Universidad Tecnológica Nacional
Facultad Regional Rosario

SINTAXIS Y SEMANTICA DE LOS LENGUAJES

LENGUAJE C DE PROGRAMACION

LIC MONICA JÄGER -- LIC SUSANA MARTÍN

1. INTRODUCCIÓN AL LENGUAJE C

1.1 PRINCIPIOS DE C

1.1.1 ORIGENES DEL C

C es el resultado de un proceso de desarrollo comenzado con un lenguaje anterior denominado BCPL, e influenció otro lenguaje denominado B, que en los años '70 llevó al desarrollo del **"C".** Fue creado e implementado por primera vez por Dennis Ritchie en un DEC-PDP11 bajo sistema operativo UNIX.-

En 1978 se editó la primera publicación de C por Kernighan y Ritchie. Con la popularidad de las microcomputadoras se crearon muchas implementaciones de C y aparecieron discrepancias. Para remediar la situación el Instituto de Estándares Americanos (ANSI) estableció un comité a principios del verano de 1983 para crear un estándar que definiera de una vez por todas el lenguaje, y éste se aprueba en 1988.

En 1990 aparece el **TURBO C**.

1.1.2 CARACTERISTICAS DE C

- ***C se clasifica como un lenguaje de nivel medio*** porque combina elementos de lenguajes de alto nivel con la funcionalidad del lenguaje ensamblador. Como lenguaje de nivel medio permite la manipulación de **bits, bytes y direcciones**; elementos básicos con los que funciona la computadora, esto lo hace particularmente adecuado para la programación de sistemas, y aunque comúnmente se le llama **"lenguaje de programación de sistemas"** debido a su utilidad para escribir S.O., compiladores, intérpretes, editores, etc., en la actualidad se lo utiliza para cualquier tipo de tareas debido a su portabilidad y eficiencia.-

- ***El código C es muy portable***, o sea, que es posible adaptar el software escrito para un tipo de computadora en otra. Los programas corren prácticamente sin cambios en cualquier máquina que maneje C.

- ***C proporciona distintos tipos de datos básicos***, los tipos fundamentales son caracteres, enteros y números de punto flotante de varios tamaños que se combinan con los operadores aritméticos y lógicos (Un tipo de dato define un conjunto de valores que puede tener una variable junto con un conjunto de operaciones que se pueden realizar sobre esa variable.) **C permite cualquier conversión de tipos.**

- ***C no lleva a cabo comprobaciones de errores en tiempo de ejecución***, como por ejemplo que se sobrepasen los límites de los arrays o que haya incompatibilidad de tipos en los argumentos. El programador es el único responsable de llevar a cabo esas comprobaciones.-

- ***C es un lenguaje estructurado*** que permite muchas posibilidades en programación. El principal componente estructural de C es la **función**; en C las funciones permiten definir las tareas de un programa y codificarlas por separado, haciendo que los programas sean modulares. Una vez que se ha creado una función que trabaja perfectamente, se puede aprovechar en distintas situaciones. Otra forma de estructuración de código en C viene dada por el uso de **bloques de código**. Un bloque de código es un grupo de sentencias de un programa conectadas en forma lógica que es tratado como una unidad. En C se crean bloques de código colocando una serie de sentencias entre llaves.
  Ejemplo:

```
if (x < 10) {
     contador = 1;
      a = x * 3;
}
```

en este caso las dos sentencias tras el if y entre las llaves se ejecutan juntas si x es menor que 10. Estas dos sentencias junto con las llaves representan un **bloque de código**. Se trata de una unidad lógica: no se puede ejecutar una de las sentencias sin la otra.
Además implementa directamente varias construcciones de bucles, tales como **while, do-while, for**, y si bien también incluye el **goto**, éste no es la forma normal de control.

- ***Tiene la ventaja de usar compilador en lugar de intérprete***, es más rápido.
intérprete → lee el código fuente de un programa línea a línea , realizando las instrucciones específicas contenidas en esa línea, el código fuente debe estar presente cada vez que se quiere ejecutar el programa. O sea, que cada vez que se ejecuta un programa el código debe ser interpretado. (→ lentitud).
compilador → lee el programa entero y lo convierte a código objeto, que es una traducción del código fuente del programa a una forma que puede ser ejecutada directamente por la computadora. Una vez que el programa está compilado el código fuente deja de tener sentido durante la ejecución. O sea, que una vez compilado lo único que hay que hacer es ejecutarlo escribiendo su nombre (→ mayor rapidez).-

- ***En C las mayúsculas y las minúsculas son diferentes***. Todas las palabras claves de C, que son 43 (32 del standard ANSI y 11 de TURBO C), están en minúscula, por lo tanto **else** es palabra clave, mientras que **ELSE** o **Else** no lo son. Las 32 palabras claves del standard ANSI son:

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

- ***Todos los programas en C consisten en una o más funciones*** (estándares o creadas). La única función que debe estar inevitablemente presente es la denominada **main()**, ya que el programa comienza a ejecutarse con la llamada a esta función. Aunque técnicamente main() no forma parte del lenguaje C (no es palabra clave), se la debe tratar como si lo fuera, usar main() como nombre de variable confundiría al compilador. En código C bien escrito main() contiene el esbozo de lo que el programa hace; este esbozo estará compuesto por llamadas a funciones, (programación top-down).-

- ***Se puede crear un programa en C*** que sea funcional y útil y ***que sólo contenga funciones creadas por el usuario***, pero esto es raro debido a que ***C no proporciona por sí mismo capacidades de E/S***, o sea que no hay proposiciones READ o WRITE, ni métodos propios de acceso a archivos; estos mecanismos se llevan a cabo a través de funciones contenidas en la biblioteca estándar de C (se las llama explícitamente); por este motivo la mayoría de los programas en C incluyen llamadas a varias funciones contenidas en la biblioteca estándar del lenguaje C. TURBO C incorpora una biblioteca estándar que proporciona las funciones que se necesitan para llevar a cabo las tareas más usuales. Cuando en un programa se llama a una función que no es parte de lo que se ha escrito en el propio programa, TURBO C "recuerda" el nombre de la función, y luego el enlazador combina el código escrito con el código objeto que ya se encuentra en la biblioteca estándar. Este proceso se denomina ***enlace.***-

- La mayoría de los programas cortos de C están enteramente contenidos en un archivo fuente, sin embargo, a medida que aumenta la longitud del programa también aumenta el tiempo de compilación. ***TURBO C permite partir un programa en muchos archivos y que cada uno sea compilado por separado.*** Una vez que han sido compilados todos los programas se enlazan entre sí, junto con las rutinas de la biblioteca, para formar el código objeto completo. La ventaja de la compilación separada es que un cambio en el código de uno de los archivos no requiere la recompilación del programa entero. En cualquier proyecto, salvo en los más simples, el ahorro de tiempo es sustancial.-

1.1.3 FORMA DE UN PROGRAMA EN C

Un programa en C, cualquiera sea su tamaño, está formado por funciones y variables.-

Ejemplo:

```
directivas al preprocesador
declaraciones globales
main ( )
{
        variables locales
        secuencia de sentencias
}
 f1 ( )
}
        variables locales
        secuencia de sentencias
}
  f2 ( )
{
        variables  locales
        secuencia de sentencias
 }
 .
 .
 .
 fn ( )
 {
        variables locales
        secuencia de sentencias
  }
```

función principal

funciones creadas por el usuario

Variables: almacenan los valores utilizados durante los cálculos.
Funciones: contienen proposiciones que indican las operaciones de cálculo que se van a realizar, normalmente se les puede dar cualquier nombre, excepto la función **main()**, que como ya dijimos debe estar en todo programa y es la primera en ejecutarse.-

Ejemplo:

```
#include <stdio.h>                    (1)
/* muestra la suma de 2 números */    (2)
main ( )                              (3)
{                                        (4)
      int a, b, c;                       (5)
      a = 10;                         (6)
      b = 15;
      c = a + b;                      (7)
      printf ("la suma es c = %d\n", c);   (8)
}
```

(1) → indica al compilador que debe incluir información acerca de la biblioteca estándar de E/S (Standard Input-Output).

(2) → cualquier carácter entre **/*** y ***/** es ignorado por el compilador y es tratado como un comentario, estos pueden aparecer en cualquier lugar dónde se pueda colocar un espacio en blanco, un tabulador (\t) o un enter (\n).-

Obs.: los comentarios no se pueden anidar, por ej., este fragmento de código produciría un error de compilación.

```
/*
    if (x < 10) printf ("todo bien"); /* muestra estado */
    else printf ("error");
*/
```

(3) → define la función **main( )**, que es la función principal del programa, está seguida por un par de paréntesis vacíos, lo que significa que no hay transferencia de argumentos. En otras funciones entre los paréntesis se coloca la lista de argumentos que transfieren valores entre la función que llama y la función llamada.-

(4) → las proposiciones de main() o de cualquier otra función van entre llaves { }

(5) → en C se deben declarar todas las variables antes de su uso, al principio de la función y antes de cualquier proposición ejecutable. Una declaración está formada por un nombre de tipo y una lista de variables.-

(6) → proposición de asignación, inicializa las variables. Cada proposición individual termina con "**;**".-

(7) → se asigna a la variable c el resultado de una operación aritmética.-

(8) → main llama a la función de biblioteca printf con el argumento:

("la suma es c = %d\n", c)

\n representa el carácter nueva línea y hace avanzar la impresión al margen izquierdo de la siguiente línea, printf no proporciona una nueva línea automáticamente (\n representa un solo carácter).
Una secuencia de caracteres entre comillas dobles se llama cadena de caracteres o constante de cadena.-

1.1.4 ALGUNAS CONSIDERACIONES AL ESCRIBIR UN PROGRAMA

- Utilizar comentarios
- Escoger nombres de variables con significado
- Utilizar líneas en blanco para separar las distintas secciones de la función
- Espacios en blanco entre los operadores
- Utilizar sangrado
- Mantener una línea con el posicionamiento de las llaves
- Tener en cuenta que **x** y **X** son dos caracteres distintos, que la mayor parte del programa se escribe en minúscula de imprenta y se usa la mayúscula de imprenta para escribir nombres de constantes
- Escribir una sentencia por línea. TURBO C permite escribir varias sentencias en una misma línea o, por el contrario, espaciar una sentencia en varias líneas, el ejemplo siguiente es válido:

```
main( )
{
    int
    cuatro
    ;
    cuatro
    =
    4
    ;
     printf(
    %d\n",
    cuatro);
}
```

```
main( )
{
   int cuatro; cuatro = 4;
  printf("%d\n";
}
```

ya que el C averigua dónde termina una sentencia y comienza la siguiente por medio de los puntos y coma introducidos, de todas maneras el programa quedará mucho más legible si se lo escribe teniendo en cuenta las pautas antes mencionadas, o sea:

```
main ( )
{
      int cuatro;
      cuatro = 4;
      printf ("%d\n", cuatro);
 }
```

1.2 EL MODELO DE COMPILACIÓN DE C

En la **Figura 1** se muestran las distintas etapas que cubre el compilador para obtener el código ejecutable.

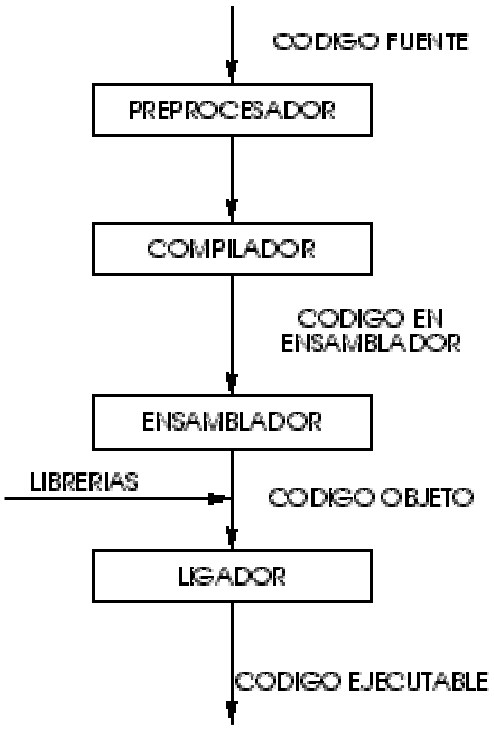


Figura 1: Modelo de compilación de C.

1.2.1 EL PREPROCESADOR

El preprocesador acepta el código fuente como entrada y es responsable de:

- quitar los comentarios
- interpretar las *directivas del preprocesador* las cuales inician con #.

1.2.1.1 DIRECTIVAS DEL PREPROCESADOR

El preprocesamiento es el primer paso en la etapa de compilación de un programa. El preprocesador tiene más o menos su propio lenguaje el cual puede ser una herramienta muy poderosa para el programador. Todas las directivas del preprocesador o comandos inician con un #.
Las ventajas que tiene usar el preprocesador son:

- los programas son más fáciles de desarrollar,
- son más fáciles de leer,
- son más fáciles de modificar
- y el código de C es más transportable entre diferentes arquitecturas de máquinas.

`#define`

La directiva `#define` se usa para definir constantes simbólicas o realizar sustitución de macros. Su formato es el siguiente:

#define `<nombre de macro> <nombre de reemplazo>`

Por ejemplo:

```
#define FALSO 0
#define VERDADERO !FALSO
```

El preprocesador también permite configurar el lenguaje. Por ejemplo, se pueden cambiar los delimitadores de bloque de código **`{ ... }`** por otros delimitadores inventados por el programador como **`inicio ... fin`** haciendo:

```
#define inicio {
#define fin }
```

Durante la compilación todas las ocurrencias de **`inicio`** y **`fin`** serán reemplazadas por su correspondiente delimitador **`{`** o **`}`** y las siguientes etapas de compilación de C no encontrarán ninguna diferencia.

La directiva `#define` tiene otra poderosa característica: el nombre de macro puede tener argumentos. Cada vez que el compilador encuentra el nombre de macro, los argumentos reales encontrados en el programa reemplazan los argumentos asociados con el nombre de la macro.

Por ejemplo:

```
#define MIN(a, b) (a < b) ? a : b

main( )
{
        int x=10, y=20;
        printf("EL mínimo es %d\n", MIN(x, y));
}
```

Cuando se compila este programa, el compilador sustituirá la expresión definida por `MIN(a, b)`, reemplazando (a, b) por `x` e `y` que son los operandos reales del programa. Así después de que el compilador hace la sustitución, la sentencia `printf` será ésta:

```
        printf("El mínimo es %d\n", (x < y) ? x : y);
```

Como se puede observar donde se coloque `MIN`, el texto será reemplazado por la definición apropiada. Por lo tanto, si en el código se hubiera puesto algo como:

```
x = MIN(q+r, s+t);
```

después del preprocesamiento, el código podría verse de la siguiente forma:

```
x = (q+r < s+t) ? q+r : s+t;
```

El uso de la sustitución de macros en lugar de funciones reales tiene un beneficio importante: incrementa la velocidad del código porque no se gasta tiempo en llamar a una función.

#undef

Se usa ***#undef*** para quitar una definición de nombre de macro que se haya definido previamente. El formato general es:

#undef `<nombre de macro>`

Por ejemplo:

```
#define LONG  100
#define ANCHO    100

char array[LONG][ANCHO];

#undef   LONG
#undef   ANCHO
/* en este momento LONG y ANCHO están indefinidas*/
```

LONG y ANCHO están definidas hasta que encuentran las sentencias ***#undef.***

El propósito principal de ***#undef*** es asignar los nombres de macro solo a aquellas secciones de código que las necesitan.

#include

La directiva del preprocesador ***#include*** hace que el compilador incluya otro archivo fuente en el que tiene la directiva **#**include. El nombre del archivo fuente que se incluirá se debe encerrar entre comillas dobles " … " o entre los signos menor que y mayor que < … >.

Por ejemplo:

```
#include <archivo>

#include "archivo"
```

Cuando se indica `<archivo>` el preprocesador busca el archivo especificado solo en el directorio establecido para contener los archivos de inclusión.

Si se usa la forma `"archivo"` el preprocesador busca primero en el directorio actual, es decir, donde el programa esta siendo ejecutado.

Los archivos de inclusión usualmente contienen los prototipos de las funciones y las declaraciones de los archivos cabecera (header files) y no tienen código de C (algoritmos).

`#if` inclusión condicional

La directiva `#if` evalúa una expresión constante entera, si la expresión constante es verdadera se compila el código asociado al `#if`. Si no es verdadera se puede evaluar otra expresión constante usando `#elif` o `#else`. Siempre se debe terminar con `#endif` para delimitar el fin de esta sentencia.

Por ejemplo:

```
#define ARG 0
#define EUA 1
#define EUR 2

#define PAÍS_ACTIVO ARG

#if PAÍS_ACTIVO == ARG
        char moneda[]="pesos";
#elif PAÍS_ACTIVO == EUA
        char moneda[]="dólar";
#else
        char moneda[]="euro";
#endif
```

Otro método de compilación condicional usa las directivas **#ifdef** (si definido) y **#ifndef** (si no definido). Estas directivas son útiles para revisar si las macros están definidas.
El formato general de **#ifdef** es:

#ifdef `<nombre de macro>`
<secuencia de sentencias>
#endif

Si el *nombre de macro* ha sido definido en una sentencia **#define**, se compilará la *secuencia de sentencias* entre el **#ifdef** y el **#endif**.

El formato general de **#ifndef** es:

#ifndef `<nombre de macro>`
<secuencia de sentencias>
#endif

Si el *nombre de macro* no ha sido definido en una sentencia **#define**, se compilará la *secuencia de sentencias* entre el **#ifndef** y el **#endif**.

Ambas sentencias, **#ifdef** y **#ifndef** pueden usar una sentencia **#else**.

Por ejemplo:

```
#include <stdio.h>
#define JUAN 10
main( )
{
    #ifdef JUAN
       printf (“Hola JUAN\n”);
    #else
       printf (“Hola, seas quien seas\n”);
    #endif
    #ifndef PABLO
       printf (“PABLO no definido\n”);
    #endif
```

imprime “hola JUAN” y “PABLO no definido”. Sin embargo, si JUAN no estuviese definido, se vería en la pantalla “Hola seas quien seas” seguido de “PABLO no definido”.

Otro ejemplo:

En C los comentarios no se pueden anidar, por ej., este fragmento de código produciría un error de compilación.

```
/*
    if (x < 10) printf ("todo bien"); /* muestra estado */
    else printf ("error");
*/
```

Aquí se trató de dejar inactiva una sección de código encerrándola entre comentarios, pero no se tuvo en cuenta que se creaba un comentario anidado; TURBO C permite esta situación e ignora el bloque completo si se lo hace de la siguiente manera:

```
# ifdef  ANULO
    if (x < 10)  printf ("todo bien"); /* muestra estado */
    else printf ("error");
# endif
```

De esta forma si ANULO es el nombre de una macro no definida, o sea, que no existe la proposición **# define ANULO 0**, todo este trozo de programa no se compila, de lo contrario sí.-

1.2.2 EL COMPILADOR

El compilador de C traduce el código fuente en código de ensamblador. El código fuente es recibido del preprocesador.

1.2.3 ENSAMBLADOR

El ensamblador crea a partir del código fuente los archivos objeto. (.obj)

1.2.4 LIGADOR

Si algún archivo fuente hace referencia a funciones de una biblioteca o funciones que están definidas en otros archivos fuentes, el *ligador* combina estas funciones con `main()` para crear un archivo ejecutable. Las referencias a variables externas son resueltas en esta etapa. (.exe)

1.2.5 USO DE LAS BIBLIOTECAS

C es un lenguaje extremadamente pequeño. Muchas de las funciones que tienen otros lenguajes no están en C, por ejemplo, no hay funciones para E/S, manejo de cadenas o funciones matemáticas.

La funcionalidad de C se obtiene a través de un rico conjunto de bibliotecas de funciones.

Como resultado, muchas implementaciones de C incluyen bibliotecas ***estándar*** de funciones para varias finalidades. Para muchos propósitos básicos estas podrían ser consideradas como parte de C. Pero pueden variar de máquina a máquina.

Un programador puede también desarrollar sus propias funciones de biblioteca.

1.2.6 PROCESO DE COMPILACIÓN Y ENLAZADO EN LENGUAJE C

El objetivo de este apartado es explicar el proceso que permite obtener un *archivo ejecutable* a partir del *código fuente en lenguaje C.*

Partiendo del caso más simple podemos suponer que el programa ha sido escrito completamente en un solo módulo (esto es, hay un solo archivo de **código fuente**). Esta situación es típicamente la de los programas pequeños.

Una vez escrito el código fuente en formato texto (con las extensiones **C** o **CPP**), se lo somete al proceso de compilación. El compilador *lee* lo escrito por el programador e intenta interpretarlo según las reglas del lenguaje y la sintaxis del **C**. Si no encuentra errores, produce un módulo llamado **objeto** (con la extensión **OBJ**) (**Figura 2**). Este módulo es la *traducción a lenguaje de máquina* del código escrito originalmente en el módulo fuente.

Figura 2

Dado que el compilador **C** (como muchos otros) necesita saber con qué nombres de variables y funciones definidos por el programador va a trabajar, estos nombres deben ser declarados antes de su utilización efectiva. Cuando estos nombres son de uso general y frecuente, se los suele declarar en pequeños archivos llamados de **cabecera** (en inglés *header*), con el fin de no tener que escribir una y otra vez las mismas declaraciones. Estos archivos de cabecera (típicamente con la extensión **H**) son a su vez incluidos en cada módulo fuente en que se vaya a utilizar alguna de las funciones allí declaradas. Un archivo de cabecera puede también incluir a otros archivos de cabecera, y todo el código de cada uno de ellos pasará a formar parte del módulo fuente como si se lo hubiera escrito directamente allí (**Figura 3**)

El módulo objeto producido por el compilador no es todavía directamente ejecutable. Es necesario someterlo a su vez a otro proceso llamado **enlazado** (en inglés *link*), que producirá un archivo ejecutable (con la extensión **EXE**) (**Figura 4**).

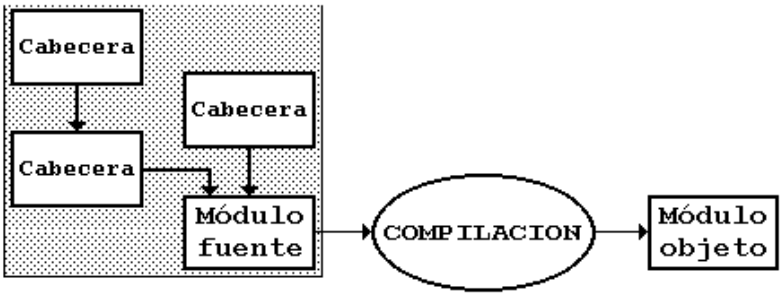


Figura 3

Figura 4

El lenguaje **C** provee de una gran cantidad de funciones ya codificadas para que el programador las utilice directamente. Este conjunto de funciones recibe el nombre de **librería**. La librería es en esencia un conjunto de módulos objeto que han sido reunidos en un único archivo (con la extensión **LIB**). El enlazador extrae de este gran depósito de código objeto aquellas funciones que realmente han sido usadas por el programador en su código fuente. Las declaraciones de estas funciones de librería están disponibles en varios archivos de cabecera, provistos por el **C**. Por lo tanto, para utilizar funciones de librería, el

programador sólo tiene que incluir el archivo de cabecera correspondiente y dejar al enlazador la tarea de hallar el código de las funciones en los archivos de librería (**Figura 5**).

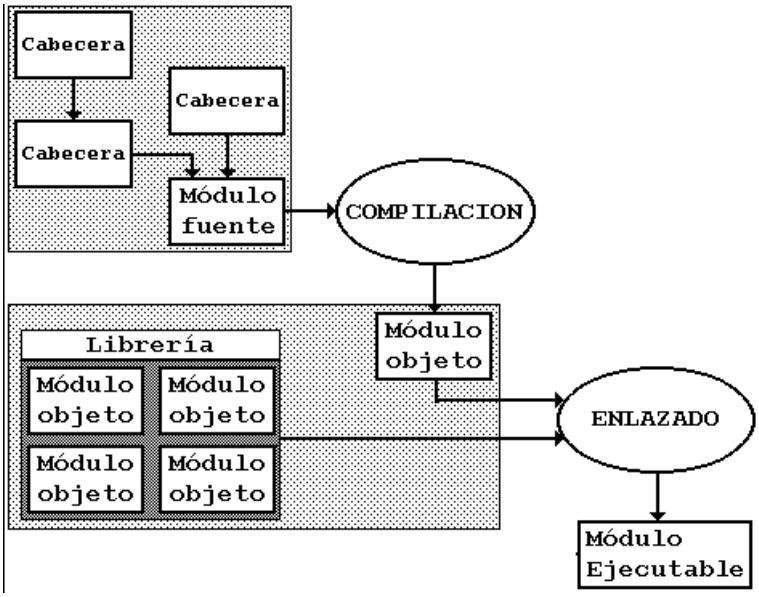


Figura 5

En el caso de los programas más grandes generalmente ocurre que el código fuente no se escribe todo en el mismo módulo, sino que se lo va distribuyendo en varios archivos de código. Cada uno de ellos incluye los archivos de cabecera necesarios y es compilado independientemente de los otros. Es tarea del enlazador reunir todos los módulos objeto así producidos y, junto con la librería del **C**, crear un único archivo ejecutable (**Figura 6**).

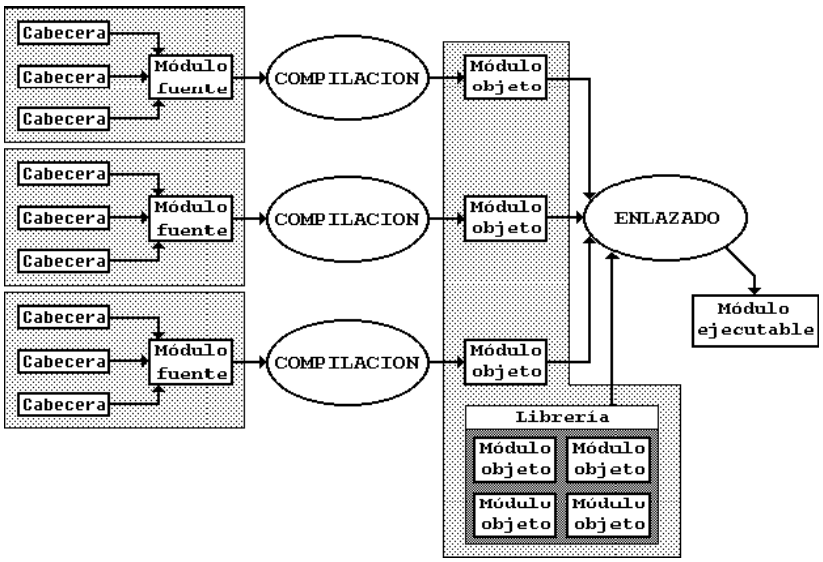


Figura 6

2. VARIABLES, CONSTANTES, OPERADORES Y EXPRESIONES

2.1 NOMBRES DE IDENTIFICADORES

Se conocen como ***identificadores*** los nombres usados para referenciar las variables, las funciones, las etiquetas y otros objetos definidos por el usuario.

Las ***reglas*** a tener en cuenta para los nombres de los identificadores son:

- Puede estar formado por letras mayúsculas, letras minúsculas, dígitos y el carácter de subrayado '_', este último es útil para mejorar la legibilidad de nombres de identificadores largos. Turbo C también permite usar el símbolo $ en un nombre de identificador, pero se recomienda no hacerlo por no tratarse de algo estándar.
- El primer carácter no puede ser un dígito, y tampoco conviene que sea el carácter de subrayado '_', puesto que las rutinas de biblioteca con frecuencia usan tales nombres. Por lo tanto conviene que el primer carácter sea una letra.
- Su longitud puede variar entre 1 y 32 caracteres. En general son significativos los primeros 31 caracteres, esto difiere según el compilador, aunque en general para nombres de funciones y variables externas, el estándar garantiza distinguir un máximo de 6 u 8 caracteres. (No distinguiría entre **contador1** y **contador2)**.
- Es conveniente elegir nombres que estén relacionados con el propósito de la variable y que no sea probable confundirlos tipográficamente.
- No debe coincidir con ninguna palabra clave.
- No debe coincidir con el nombre de ninguna función escrita por el usuario o que se encuentre en la biblioteca de Turbo C.
- Se acostumbra usar minúsculas para nombres de variables y mayúsculas para nombres de constantes simbólicas.

Los siguientes son ejemplos de nombres de identificadores:

Correctos	**Incorrectos**
cont	1cont
prueba_1	hola!
acum12	acum-12
NUM_3	NUM-3
N123	N*123

2.2 TIPOS DE DATOS

En C las variables se corresponden con una posición de memoria y poseen: un **nombre** que las identifica y permite así, referirse al contenido de una dirección particular de memoria y un **tipo** que las describe, para interpretar el valor contenido en esa dirección.

Existen cuatro tipos de datos básicos en C:

PALABRA CLAVE	TIPO	SIGNIFICADO
char	carácter	carácter
int	entero	número entero con signo
float	flotante de simple precisión	número con signo, con o sin parte fraccionaria
double	flotante de doble precisión	número con signo, con o sin parte fraccionaria con más dígitos significativos que el tipo anterior

El C proporciona además un tipo de dato de propósito especial cuyo significado es "sin valor" y cuya palabra clave es ***void***. Desde el punto de vista sintáctico este tipo se comporta como un tipo fundamental, no obstante, se lo puede utilizar sólo como parte de un tipo derivado. **No existen variables** de tipo **void**. Sirve para especificar que una función no devuelve ningún valor, o bien, para declarar explícitamente una función sin parámetros.

2.2.1 MODIFICADORES DE TIPO

Los tipos de datos básicos pueden estar precedidos por modificadores que alteran el significado del tipo base para que se ajuste más precisamente a cada necesidad. Los modificadores de tipo que existen en C son:

short ..... reduce a la mitad la longitud de almacenamiento
long ..... duplica la longitud de almacenamiento
unsigned ..... número sin signo (siempre positivo)
signed ..... número con signo (positivo o negativo)

El rango de valores que pueden representarse con cada tipo depende del diseño del hardware, en cuanto a la cantidad de bits dispuestos para cada uno. Generalmente se dispone de 16 bits para contener un entero, y de 8 para almacenar un carácter del juego de caracteres ASCII. Dado que internamente los datos en C se almacenan en ASCII, el C permite usar una variable de tipo **char** como un "entero pequeño". En este caso el entero será el número que se corresponda en el sistema decimal con la cadena de 8 cifras binarias. Según la arquitectura de la máquina ese número puede ser con o sin signo, aunque esto no afecta a la impresión del carácter en sí, ya que los caracteres que se pueden imprimir son siempre positivos.

Con la inclusión de los modificadores de tipo la variedad de tipos se amplía: **short int, long int, signed char,** etc. En el caso de los enteros permite una notación abreviada: **short, long o unsigned** pues el compilador sobreentiende el **int** por omisión, asimismo el **signed int** resulta redundante, pues normalmente el **int** es con signo.

La mayoría de los compiladores C de PC usan enteros de 16 bits. El estándar ANSI de C establece que el menor tamaño aceptable para un **int** y para un **short int** es de 16 bits; por ese motivo en muchos entornos no hay diferencias entre un **int** y un **short int**.

La diferencia entre un entero con signo (int) y un entero sin signo (unsigned int) está dada por la forma de interpretar el bit más significativo:

- **int:** el bit más significativo se usa como indicador de signo (0 para positivo, 1 para negativo), por lo tanto son 15 los bits que representan al número, y para los negativos generalmente aplica el método de complemento a 2 (tiene todos los bits del número invertidos y un 1 sumado al número).
- **unsigned int:** no existe un bit para el signo, por lo tanto los 16 bits representan al número.

Si una variable declarada como **int** contiene: 0111111111111111 será interpretado como el +32767, pero si los bits contenidos son 1111111111111111 será interpretado como el -32767. Si en cambio el 0111111111111111 está contenido en una variable declarada como **unsigned int** también será interpretado como el +32767, pero si los bits contenidos fueran 1111111111111111 se trataría del +65535.

Para elegir el tipo de dato adecuado es necesario conocer muy bien la arquitectura de la máquina; en general los conceptos fundamentales a tener en cuenta respecto al tamaño son:

tamaño char < tamaño short int < = tamaño int <= tamaño long int
tamaño float < tamaño double < tamaño long double

El siguiente cuadro muestra todas las combinaciones que se ajustan al estándar ANSI junto con sus rangos y longitudes, suponiendo palabras de 16 bits.

TIPO	TAMAÑO (en bits)	RANGO
char	8	-128 a +127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a +32767
unsigned int	16	0 a +65535
signed int	16	-32768 a +32767
short int	16	-32768 a + 32767
unsigned short int	16	0 a +65535
signed short int	16	-32768 a +32767
long int	32	-2147483648 a +2147483647
unsigned long int	32	0 a + 4294967295
signed long int	32	-2147483648 a +2147483647
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long double	64	1.7E-308 a 1.7E+308

2.3 VARIABLES

2.3.1 DECLARACIÓN DE VARIABLES

En C todas las variables deben declararse antes de su uso, generalmente al principio de la función y antes de cualquier proposición ejecutable. Al declarar una variable se le está especificando:

- al programa, cómo debe interpretar los contenidos de la memoria, y por lo tanto se establece cuánta necesita.
- al compilador, el significado de los símbolos de operación que se le apliquen a los valores contenidos en esas direcciones de memoria.

La declaración de variables se hace por medio de una sentencia que tiene el siguiente formato:

tipo lista_de_variables

dónde, **tipo** debe ser un tipo de datos válido de C y la **lista_de_variables** puede consistir en uno o más nombres de variables separados por comas.

Ejemplos:

```
int i, j, k;
unsigned suma;
char a, b, c;
double  prom1;
```

Existen tres sitios básicos donde se pueden declarar variables: dentro de las funciones, en la definición de parámetros de funciones y fuera de todas las funciones. Estas variables son respectivamente, las variables locales, los parámetros formales y las variables globales.

2.3.2 INICIALIZACIÓN DE VARIABLES

Inicializar significa especificar un valor de comienzo. Para aquellas que no se inicialicen se debe asumir que contienen valores desconocidos, "basura", porque si bien hay algunos compiladores que inicializan automáticamente a cero, no conviene tomarlo como algo seguro (globales a 0, locales con basura). La inicialización de cualquier variable se realiza mediante la asignación de un valor constante que puede realizarse en la sentencia de declaración o en una sentencia por separado. El formato general de la inicialización es:

```
tipo nombre_de_variable = constante;
```

o bien:

```
tipo nombre_de_variable;
nombre_de_variable = constante;
```

la ventaja de inicializar en la declaración en lugar de usar una sentencia de asignación separada está en que el compilador producirá un código más rápido.

Ejemplos:

```
int cant = 97;
char c = 'a';
int x = 0, y = 0;     o bien:     int x, y;        o bien:    int x, y;
                                  x = 0;                      x = y = 0;
                                  y = 0;
```

2.3.3 VARIABLES LOCALES Y GLOBALES

Según el lugar dónde se declaran las variables se clasifican en distintos tipos, a saber:

1· **Variables Locales**, también llamadas privadas o automáticas:

- Se declaran dentro de una función o de un bloque de código.
- Sólo pueden ser referenciadas por sentencias que estén dentro de esa función o de ese bloque de código, ya que fuera de los mismos no se las conoce.
- Sólo existen mientras se está ejecutando la función o el bloque de código en el que están declaradas. Se crea cuando se llama a dicha función o se entra a dicho bloque y se destruye cuando se sale de ellos, por lo tanto no conservan los valores entre llamadas.
- Su nombre puede ser usado por otras funciones.

Ejemplos:

a- La variable entera **x** se define dos veces, una vez en **func1( )** y otra en **func2( )**, pero una no tiene relación con la otra, ya que cada **x** sólo es conocida en la propia función en la que fue declarada.

```
func1( )
 {
    int x;
    x = 10;
      .
      .
 }
 func2( )
 {
```

```
    int x;
    x = 35;
       .
       .
  }
```

b- La variable **s** se define dentro de **un bloque de código,** fuera de él la variable **s** es desconocida

```
{
   int t;

      if (t > 0) {
         char s;  /* esta existe únicamente en este bloque */

      }
      /* aquí  no se conoce a  la variable s  */
}
```

2· <u>Variables Globales</u>:

- Se declaran fuera de todas las funciones.
- Son conocidas a lo largo de todo el programa y por lo tanto se pueden usar en cualquier parte del código
- Mantienen su valor durante toda la ejecución del programa.
- Si una variable local y una variable global tienen el mismo nombre, toda referencia a ese nombre de variable dentro de la función dónde se ha declarado la variable local se refiere a ésta y no tiene efecto sobre la variable global.

Ejemplo:

```
#include <stdio.h>
void f1(void );
void f2(void );
int var;   /* var es global */
main( )
{
        var = 10;
        f1( );

}
void f1( )
{
        int h;
        h = var;
        f2( );
        printf ("var es %d\n", var);  /* imprimirá 10 */
}
void f2( )
{
        int var;  /* var es local */
        var = 34;
        x = var * 2;
        printf ("x = %d\n", x);  /* imprimirá 68 */
}
```

Las variables globales son muy útiles cuando muchas funciones del programa usan los mismos datos, sin embargo, se debe evitar el uso innecesario de variables globales por las siguientes 3 razones:

* las variables globales usan memoria todo el tiempo que el programa está en ejecución, no sólo cuando se necesitan (importante si la memoria es un recurso escaso).

* usar una variable global dónde podría ir una local hace a la función menos general, ya que depende de algo que debe estar definido fuera de ella.

* usar muchas variables globales puede acarrear problemas de programación (especialmente en programas grandes) cuando se modifica el valor en algún lugar del programa.

2.3.4 ESPECIFICADORES DE CLASE DE ALMACENAMIENTO

Indican al compilador cómo debe almacenar la variable que le sigue, precede al resto de la declaración de la variable, y su forma general es:

especificador_de_almacenamiento tipo nombre_de_variable

Existen 4 tipos, a saber:

- ***auto***: sirve para declarar variables locales, por lo general no se usa ya que se asume por omisión.

- ***extern***: la forma de hacer conocer a todos los archivos (caso compilación por separado) las variables globales requeridas por el programa es declarando la totalidad de las mismas en un archivo, y usar declaraciones *extern* en los otros. Este especificador indica al compilador que los nombres y tipos de variables que siguen ya han sido declarados en otra parte y por lo tanto no crea un nuevo almacenamiento para ellas. Cuando una declaración crea almacenamiento se dice que es una *definición*; las sentencias extern son declaraciones pero no definiciones. Si todo está dentro de un mismo archivo fuente (definición y declaración) puede omitirse la palabra extern en la declaración.

  Ejemplos:

a-

Archivo 1

```
int x, y;
char c;

main ( )
{
    .
    .
}
func1 ( )
{
   x = 123;
 }
```

Archivo 2

```
extern int x, y;
extern char c;

func2 ( )
{
     x = y / 10;
}
func3 ( )
{
     y = 10;
        }
```

b-

```
int uno, dos;   /* definición global de uno y dos  */

 main ( )
{
   extern int uno, dos;    /* uso opcional de la declaración extern  */

  }
```

- ***static*** dentro de su propia función o archivo las variables static son permanentes. Difieren de las variables globales en que no son conocidas fuera de su función o archivo, aunque mantienen sus valores entre llamadas. Tienen efectos diferentes siendo locales que globales.

***** Variables estáticas locales: · el compilador crea un almacenamiento permanente para ellas.
· sólo es conocida en el bloque en que está declarada.
· es una variable local que mantiene su valor entre llamadas

* Variables estáticas globales: sólo es conocida en el archivo en el que se declara, o sea, que las rutinas de otros archivos no la reconocerán ni alterarán su contenido directamente.

- ***register***: sólo se aplica a variables de tipo int y char y sólo a variables locales y parámetros formales. Hace que Turbo C mantenga el valor de la variable así declarada de forma que se permita un acceso más rápido a la misma. Por lo general se guarda en CPU en lugar de hacerlo en memoria, por lo tanto no se requiere acceso a memoria para determinar o modificar su valor.

2.4 CONSTANTES

C permite declarar *constantes*. Declarar una constante es parecido a declarar una variable, excepto que el valor no puede ser cambiado.
La palabra clave **const** se usa para declarar una constante, como se muestra a continuación:

```
const int a = 1;
```

Notas:

- Se puede usar **const** antes o después del tipo.
- Es usual inicializar una constante con un valor, ya que no puede ser cambiada *de alguna otra forma*.

La directiva del preprocesador #define es un método más flexible para definir *constantes* en un programa, a las constantes así definidas se las denomina **constantes simbólicas**.

Frecuentemente se ve la declaración **const** en los parámetros de una función. Lo anterior simplemente indica que la función no cambiará el valor del parámetro. Por ejemplo, la siguiente función usa este concepto:

```
char *strcpy (char *dest, const char *orig);
```

El segundo argumento orig es una cadena de C que no será alterada, cuando se use la función de la biblioteca para copiar cadenas.

2.4.1 CONTANTES NUMÉRICAS

Pueden ser **enteras** (contienen un número entero) o **de coma flotante** (contienen un número en base 10 que contiene un punto decimal o un exponente o ambos).
Ejemplos de **Constantes**:

Enteras	de Coma Flotante
5280	0.2
32767	1.
0	2e-8
743	12.3e-5

Cuando se introducen constantes numéricas en un programa, el compilador decide que tipo de constantes son. Por omisión elige el tipo de dato compatible más pequeño que pueda contener a una constante numérica., salvo en el caso de las constantes de punto flotante que las supone de tipo double. Así 20 será int, aunque hubiera podido caber en un char y el 63000 será un unsigned int.

2.4.2 CONTANTES NO NUMÉRICAS

Pueden ser de 2 tipos:

- ***de carácter:*** es **un carácter** delimitado **entre comillas simples.** El valor de una constante de carácter es el número correspondiente al código del procesador (generalmente el ASCII). Hay algunos caracteres no imprimibles que se representan como constantes de carácter mediante una secuencia de escape que aparece como 2 caracteres pero en realidad son sólo uno (‘\n’, ‘\0’, etc.).

- ***cadena de caracteres:*** es una secuencia de **ninguno, uno o más caracteres encerrada entre comillas dobles**. El compilador ubica un carácter nulo (‘\0’) al final de cada cadena para que los programas puedan encontrar el final. Técnicamente una cadena de caracteres es un vector cuyos elementos son caracteres.

NOTA: no es lo mismo ‘x’ que “x”; en el primer caso se trata de un único carácter, x, cuyo valor es el que le corresponde en el conjunto de caracteres del procesador, y en el segundo caso se trata de una cadena que tiene 2 caracteres, una x y un \0.

2.4.3 CÓDIGOS DE BARRA INVERTIDA

Son constantes de carácter que se utilizan para representar caracteres que son imposibles de introducir desde el teclado. Se utilizan igual que las demás constantes de carácter, o sea, pueden asignarse a una variable o bien utilizarlas dentro de una cadena de caracteres; (c = ‘\t’, “error\n”). El siguiente cuadro muestra la lista de códigos de barra invertida:

Código	*Significado*
\b	Espacio atrás
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\”	Comillas dobles
\’	Comillas simples
\0	Nulo
\\	Barra invertida
\v	Tabulador vertical
\a	Alerta
\o	Constante octal
\x	Constante hexadecimal

2.5 OPERADORES

El C define muchos más operadores que otros lenguajes. Combinados con distintos operandos pueden generar expresiones que siguen las reglas del álgebra, pudiendo los paréntesis alterar el orden de evaluación que le correspondería por la prioridad propia de cada operador.

2.5.1 ARITMÉTICOS

Suma ..... **+**
Resta ..... **–**
Multiplicación .. *****
División ..... /
Módulo ..... **%** (resto de la división entera-**no** puede aplicarse a operandos float o double)
Incremento…… ++
Decremento….. --

Incremento y decremento

C contiene dos operadores muy útiles que no existen generalmente en otros lenguajes. Son el *incremento* y el *decremento*, **++ y --**.

++ ..... aumenta en 1 el valor del operando
-- ..... disminuye en 1 el valor del operando

El formato de la expresión que resulta de aplicar el operador incremento o decremento es:

operador variable

variable operador

dónde operador actúa distinto según su ubicación respecto a la variable:

***** operador variable (prefijo)
1º se modifica en 1 el valor de la variable
2º la variable usa su nuevo valor

***** variable operador (sufijo)
1º la variable usa su valor
2º se modifica en 1 el valor de la variable

Ejemplos:

++n equivale a n = n + 1

z = b++ equivale a z = b y luego b = b + 1

x = --r equivale a r = r – 1 y luego x = r

Con éstos se pueden generar expresiones aritméticas cuyos operandos pueden ser *constantes, variables u otro tipo de expresiones.*

- Cuando en una expresión se combinan distintos tipos de operadores, la prioridad en la evaluación es:

  1º ++ -- - (monario)
  2º ***** / **%**
  3º + -

- Cuando en una expresión se combinan operadores de igual prioridad, la evaluación es de izquierda a derecha. Los paréntesis alteran este orden de evaluación.

2.5.2 RELACIONALES

Mayor ..... >
Menor ..... <
Mayor o igual ..... >=
Menor o igual ..... <=
Igual ..... = =
Distinto ..... **!=**

Con éstos se pueden generar expresiones relacionales cuyos operandos pueden ser *constantes, variables u otro tipo de expresiones.*

- Cuando en una expresión se combinan distintos tipos de operadores, la prioridad en la evaluación es:

  1º > < >= <=
  2º = = **!=**

- Cuando en una expresión se combinan operadores de igual prioridad, la evaluación es de izquierda a derecha. Los paréntesis alteran este orden de evaluación.

- El resultado de una operación relacional es un número entero: **1 para verdadero y 0 para falso**.

2.5.3 LÓGICOS

Conjunción (y)..... **&&**
Opción (o) ..... ||
Negación (no) ..... **!**

Con éstos se pueden generar expresiones lógicas cuyos operandos pueden ser *constantes, variables u otro tipo de expresiones.*

- Cuando en una expresión se combinan distintos tipos de operadores, la prioridad en la evaluación es:

  1º **!**
  2º **&&**
  3º ||

- Cuando en una expresión se combinan operadores de igual prioridad, la evaluación es de izquierda a derecha. Los paréntesis alteran este orden de evaluación.

- El resultado de una operación lógica es un número entero: **1 para verdadero y 0 para falso**.

En C *cierto* es cualquier valor distinto de 0 y *falso* es 0

2.6 OTROS OPERADORES

2.6.1 LA COMA COMO OPERADOR (,)

Como operador la coma encadena varias expresiones, su función es la de indicarle al compilador la sucesión de cosas que debe hacer (hacer esto y esto y esto).

Cuando se usa en la parte derecha de una sentencia de asignación, el valor asignado es el valor de la última expresión de la lista separada por comas.

Ejemplo:

```
x = (y = 3, y + 1);  ===>  x = 4

y = 10;
x = (y = y – 5, 25 / y);  ===>  x = 5
```

los ( ) son necesario por la precedencia: = > ,

2.6.2 OPERADOR DE ASIGNACIÓN (=)

Provoca el almacenamiento de un valor en una variable. El formato general resulta:

variable operador valor

dónde valor puede ser una constante, otra variable (ya inicializada), o una expresión.

Como en otros lenguajes la operación de asignación es una sentencia, pero a diferencia de muchos de ellos se trata, en C, de un operador muy potente.

Variaciones en la sentencia de asignación:

- En la inicialización puede asignársele el mismo valor a distintas variables, no así en la declaración, la forma sería:

variable1 = variable2 = variable3 = valor

- Permite usar abreviaturas cuando una expresión aritmética involucra a la misma variable de ambos lados del signo igual con la forma:

variable = variable operador expresión

y el operador involucrado es: +, -, *, / o %; puede reemplazarse por:

variable operador = expresión

2.6.3 OPERADOR CONDICIONAL (?:)

C contiene un operador muy potente y conveniente que puede usarse para sustituir ciertas sentencias de la forma if ... else. El operador ternario ?: toma la forma general:

Exp1 ? Exp2 : Exp3;

El **operador ?:** actúa de la siguiente manera: evalúa ***Exp1***, si es cierta evalúa ***Exp2*** y toma ese valor para la expresión. Si ***Exp1*** es falsa evalúa ***Exp3*** tomando su valor para la expresión.

Ejemplo:

```
y = x > 9 ?  100 : 200
```

en este caso si **x** es mayor que 9, a **y** se le asigna el valor 100, de lo contrario a **y** se le asigna el valor 200.

2.6.4 EL OPERADOR sizeof

El operador de tiempo de compilación sizeof es un operador monario que devuelve la longitud, en bytes, de la variable o del especificador de tipo entre paréntesis al que precede.

Ejemplo:

```
main( )
{
    float f;
    printf ( "%f\n", sizeof f );
    printf ( "%f\n", sizeof (int));
}
```

Nota: para calcular el tamaño de un tipo, el nombre debe ir entre ( ). No así los nombres de variables.

2.7 EXPRESIONES Y SENTENCIAS

Una **expresión** representa una unidad de datos simple, tal como un número o un carácter. La expresión puede consistir en una entidad simple, como una constante, una variable, un elemento de un arreglo o una referencia a una función. También puede consistir en alguna combinación de tales identidades interconectadas por uno o más operadores. Por ejemplo:

a + b *x = y* *c = a + b* *x = = y*

Una **sentencia** consiste en una expresión seguida de un punto y coma (**;**) y hace que la computadora lleve a cabo alguna acción. La ejecución de la sentencia de expresión hace que se evalúe la expresión. Algunos ejemplos de sentencias son:

a = 3; *++ i;* *c = a + b;* *suma += 10;*

2.7.1 CONVERSIÓN DE TIPO DE DATOS

Los operandos de tipos distintos que aparecen en una expresión se convierten a un mismo tipo en forma automática de acuerdo con ciertas reglas o en forma forzada.

2.7.1.1 CONVERSIÓN AUTOMÁTICA DE TIPO IMPLÍCITA

Hay que diferenciar entre el tipo de dato del resultado de una expresión (***regla nº 1)*** y el tipo de dato con que ese resultado se almacena en la variable ***(regla nº 2)***.

Regla nº 1: en una expresión aritmética dónde se vinculan por lo menos 2 operandos, si estos son de distinto tipo, el compilador C, previo a la resolución de la operación igualará los tipos de datos convirtiéndolos automáticamente según una escala de dominio de tipos que podríamos expresar como:

char ---> int ---> float ---> double

Esto implica decir que los char se convierten a int tomando el valor ASCII del carácter en cuestión, los int se convierten a float y estos a double.

Regla nº 2: en una asignación, independientemente de los tipos de datos que intervienen se almacenará el resultado de la expresión convirtiéndose automáticamente al tipo de la variable de asignación. Expresando las asignaciones por su tipo podríamos resumir:

int = char.....se almacena el código ASCII del carácter
int = float.....se almacena como entero truncando los decimales
float = double.....se almacena en simple precisión redondeando los decimales
float = int.....se almacena el valor entero con parte decimal 0
float = char.....se almacena el código ASCII del carácter con su parte decimal 0
char = float.....se almacena el carácter correspondiente truncando los decimales

Ejemplo:

```
char    ch;
int     i;
float   f;
double d;

res = (ch / i) + (f  *  d) - (f  + i);
        \   |     \     /     \    /
          int     double      float
           \         |         |
            \       /         /
             \     /         /
              \   /         /
              double       /
                 \        /
                  double
```

A partir de acá y dependiendo del tipo de dato con que se halla declarado la variable res se producirá o no pérdida de información al almacenarse el resultado de la operación

Otros ejemplos:

- a = 7 + 3.8

1º El resultado es 10.8
2º si **a** es **int** se almacena 10
si **a** es **float** se almacena 10.800000
si **a** es **char** se almacena el carácter “nueva línea”

- a = ‘A’ + 5.0

1º El resultado es 70.0
2º si **a** es **int** se almacena 70
si **a** es **float** se almacena 70.000000
si **a** es **char** se almacena el carácter F

- a = ‘A’ + 5.9

1º El resultado es 70.9
2º si **a** es **int** se almacena 70
si **a** es **float** se almacena 70.900000
si **a** es **char** se almacena el carácter F

Nota: cuando una expresión aritmética que se asigna a una variable char da un resultado mayor que 255 o menor que 0, el carácter que se almacena es el que corresponde a los 8 bits menos significativos.

2.7.1.2 CONVERSIÓN FORZADA DE TIPO EXPLÍCITA O MOLDES (CAST)

Si algún operando de una expresión debe ser usado momentáneamente con otro tipo, distinto al con el que fue declarado o más allá del que adquirió automáticamente de acuerdo con las reglas anteriores, es posible provocar esa conversión. El operando será afectado por el nuevo tipo de dato de la siguiente forma:

(nuevo tipo de dato) operando

En el siguiente cuadro se muestra la posible pérdida de información que se puede producir en las conversiones de tipos más usuales asumiendo palabras de 16 bits (2 bytes):

TIPO DESTINO	TIPO EXPRESIÓN	POSIBLE PÉRDIDA DE INFORMACIÓN
signed char	char	Si valor > 127 destino negativo
char	short int	8 bits más significativos
char	int	8 bits más significativos
char	long int	24 bits más significativos
short int	int	Nada
short int	long int	16 bits más significativos
int	long int	16 bits más significativos
int	float	Parte fraccional y más
float	double	Precisión, result.redondeado
double	long double	Precisión, result.redondeado

2.8 TABLA GENERAL DE PRECEDENCIAS

Mayor	( ) [ ] ->
	! - ++ -- (tipo) * & sizeof
	* / %
	+ -
	< <= > >=
	= = !=
	&&
	||
	?:
	= += -= *= /= %=
Menor	,

3. SENTENCIAS DE CONTROL DE PROGRAMA

3.1 CONSIDERACIONES GENERALES

CIERTO Y FALSO EN EL LENGUAJE C

La mayoría de las sentencias de control de programa se basan en una prueba condicional que determina la acción a seguir, o sea, especifican el orden en que se realiza el procesamiento. Una prueba condicional produce un valor cierto o falso; en C cualquier valor distinto de cero es cierto, incluyendo los números negativos. "El 0 (*cero*) es el único valor falso en C".

PROPOSICIONES Y BLOQUES

- El **;** **es un terminador de sentencias o proposiciones**, o sea, expresiones como x = 0 ó i++ ó printf(...) cuando van seguidas de **;** se convierten en proposiciones.

- Las llaves **{ }** se emplean para agrupar declaraciones y proposiciones dentro de una **proposición compuesta o bloque**, de modo que son sintácticamente equivalentes a una proposición sencilla. No se coloca "**;**" después de la llave derecha que termina un bloque.

- En **C** una sentencia puede consistir en una única sentencia, un bloque de sentencias o nada (en el caso de sentencias vacías).

3.2 SENTENCIAS CONDICIONALES

El lenguaje **C** soporta dos tipos de sentencias condicionales: ***if*** (decisión simple) y ***switch*** (decisión múltiple). Además, el operador condicional ***?:*** (ya desarrollado en el capítulo 2) es una alternativa para if en ciertas situaciones.

3.2.1 SENTENCIA *if*

Se utiliza para expresar decisiones. La forma general de la sentencia ***if*** es:

```
if (expresión) sentencia_1;
else sentencia_2;
```

dónde **sentencia_1** y **sentencia_2** pueden ser una sentencia simple o un bloque y la cláusula **else** es opcional.

La forma general del **if** con bloques de sentencias es:

```
if (expresión) {
    secuencia de sentencias_1
}
else {
    secuencia de sentencias_2
}
```

En cualquiera de los dos casos la *expresión se evalúa*, si es *verdadera* (cualquier valor != 0) *sentencia_1* se ejecuta, si es *falsa* (= 0) y existe una parte else *sentencia_2* se ejecuta. Recordar que sólo se ejecuta el código asociado al if o al else, nunca ambos.

Ejemplo:

```
if (num > 10)
 num = num * 2;
else
    printf (“num = %d\n”, num);
```

♦ if ANIDADOS

Un **if** anidado es un **if** que es el objeto de otro **if** o **else**. La razón por la que los **if** anidados son tan problemáticos es que debido a que el **else** es optativo puede ser difícil saber qué **else** se asocia con cuál **if.**

En **C** esto se simplifica, ya que el **else** siempre se refiere al **if** precedente sin **else** más próximo, o sea, el **else** va con el **if** anterior sin **else** más cercano, a menos que haya llaves que indiquen lo contrario.

Ejemplo:

```
if (número > 6)                          Nº        Rta.
   if (número < 12)              5          ---
      printf (“correcto\n”);               10      correcto
    else                                   15       error
       printf (“error\n”);
```

Si esto no es lo que se desea se deben utilizar llaves para forzar la asociación correcta:

```
if (número > 6) {                        Nº        Rta.
   if (número < 12)              5        error
      printf (“correcto\n”);}    10      correcto
 else                                    15        ---
    printf (“error\n”);
```

Si no coloco las llaves {} por más que el sangrado muestra en forma inequívoca lo que se desea, el resultado sería igual al anterior.

♦ LA ESCALA if – else – if

Una construcción común en programación es la escala *if-else-if*. Su forma general es:

```
if (expresión_1)
    sentencia_1;
else if (expresión_2)
    sentencia_2;
else if (expresión_3)
    sentencia_3;
  .
  .
else
    sentencia_n;
```

Esta secuencia de proposiciones **if** es la forma más general de escribir una *decisión múltiple*. Las expresiones se evalúan en orden de arriba abajo; tan pronto como se encuentra una expresión cierta la sentencia asociada con ella se ejecuta y el resto de la escala se pasa por alto. Si ninguna de las condiciones es verdadera se ejecuta el **else** final, o sea, el último **else** maneja el caso “ninguno de los anteriores”; es decir, actúa como *condición implícita*, si todas las pruebas condicionales fallan, se ejecuta la última sentencia **else**. En algunos casos no hay una acción explícita para la omisión, en este caso el **else sentencia** del final puede omitirse o utilizarse para detección de errores al encontrar una condición imposible.

3.2.2 SENTENCIA *switch*

Si bien con la escala *if-else-if* se puede escribir una decisión múltiple, este código puede ser bastante difícil de seguir. Por este motivo **C** incorpora una sentencia de decisión de ramificación múltiple denominada **switch**. Esta sentencia compara sucesivamente una variable con una lista de constantes enteras o de caracteres, cuando se encuentra una correspondencia traslada el control adecuadamente y se ejecuta una sentencia o bloque de sentencias. La forma general de la sentencia **switch** es:

```
switch (variable) {
      case constante_1:
            secuencia de sentencias
            break;
      case constante_2:
            secuencia de sentencias
            break;
      case constante_3:
            secuencia de sentencias
            break;
        .
        .
        .
      default:
            secuencia de sentencias
            break;
}
```

dónde la sentencia **default** se ejecuta si no se encuentra ninguna correspondencia. La parte **default** es opcional, y si no aparece, no se lleva a cabo ninguna acción al fallar todas las pruebas. Cuando se encuentra una coincidencia, se ejecuta la secuencia de sentencias asociadas con ese case hasta encontrar la sentencia break o, en el caso de default (o el último case si no hay default), el final del switch con lo que sale de la estructura.

Resumiendo, cada caso empieza con un case y acaba donde hay un break. Si no ponemos break aunque haya un case el programa sigue hacia delante hasta encontrar un break o la llave de cierre del switch. Esto puede parecer una desventaja pero a veces es conveniente. Por ejemplo cuando dos case deben tener el mismo código, si no tuviéramos esta posibilidad tendríamos que escribir dos veces el mismo código.

La forma en que se puede simular la prueba de más de una constante por case, es no teniendo sentencias asociados a un case, es decir, teniendo una sentencia nula donde sólo se pone el caso, con lo que se permite que el flujo del programa *caiga* al omitir las sentencias, como se muestra a continuación:

```
switch (letra)
{
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    numvocal++;
    break;
```

```
  case ' ':
    numesp++;
    break;

  default:
    numotras++;
    break;
}
```

Puntos importantes a tener en cuenta sobre la sentencia switch:

- Con switch sólo se puede comprobar la igualdad, no se pueden evaluar expresiones relacionales o lógicas, mientras que con if puede usarse cualquier operador relacional.

- Dentro de un switch no puede haber dos constantes case que tengan los mismos valores. Por supuesto una sentencia switch contenida en otra sentencia switch (switch anidados) puede tener constantes case que sean iguales.

- Si se utilizan constantes de tipo carácter en la sentencia switch, se convierten automáticamente a sus valores enteros.

3.3 SENTENCIAS DE ITERACIÓN

Las sentencias de iteración o bucles son estructuras que permiten ejecutar partes del código de forma repetida mientras se cumpla una condición. Esta condición puede ser simple o compuesta de otras condiciones unidas por operadores lógicos.

3.3.1 LA SENTENCIA *for*

La forma gral. de la sentencia ***for*** es:

```
for (inicialización; condición; incremento)
{
        sentencia1;
        sentencia2;
}
```

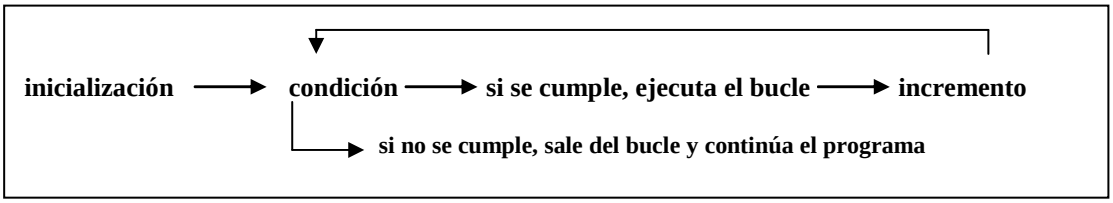
Este bucle permite agrupar las tres acciones en un sólo lugar:

- La ***inicialización*** se realiza una única vez al principio del bucle, normalmente es una sentencia de asignación que se utiliza para inicializar la variable de control del bucle.-

- La ***condición*** es una expresión relacional que determina cuando finaliza el bucle, se evalúa antes de cada ejecución potencial del bucle, cuando la expresión es falsa, o sea, = 0, el bucle finaliza. Esto permite que el for se ejecute de 0 a n veces, ya que si la condición es falsa de entrada no se ejecuta nunca.-

- El ***incremento*** define como cambia la variable de control cada vez que se repite el bucle, se ejecuta al final de cada iteración.-

El flujo del bucle **for** transcurre de la siguiente forma:



Observaciones:

- Estas tres secciones principales deben estar separadas por **;** (punto y coma)
- Cualquiera de la tres secciones pueden omitirse, pero deben permanecer los **;**
- Se utiliza el operador 'coma' para permitir que en el bucle se inicialicen y/o se incrementen dos o más variables de control.

```
for (a=1, b=100; a!=b; a++, b- -){
        …….. }
```

- Se utilizan los operadores condicionales para permitir que en el bucle se evalúen dos o mas condiciones de finalización

```
for (x = 0; x < 10 && z == 0; ++x){
        …… }
```

- El C permite generar bucles for sin cuerpo, generalmente se utilizan para generar retardos

```
for (t = 0; t <= valor; ++t);
```

- El C permite generar bucles infinitos, esta iteración infinita será interrumpida por una sentencia break.

```
for ( ;  ; ) {
  …… }
```

3.3.2 LA SENTENCIA *while*

La sentencia ***while*** se utiliza para generar bucles. La forma general de la sentencia ***while*** es:

```
while ( expresión) {
   sentencia;

        }
```

donde ***sentencia*** puede ser una sentencia única o un bloque de sentencias que se ejecutarán repetidamente mientras el valor de *expresión* no sea 0. Esta sentencia debe incluir algo que altere el valor de *expresión* (MUY IMPORTANTE PARA NO HACER UN BUCLE INFINITO). Cuando el flujo del programa llega a esta instrucción, primero se revisa si la *expresión* es verdadera para ejecutar la(s) sentencia(s), y después el ciclo while se repetirá mientras el valor de *expresión* no sea cero. Cuando llega a ser falsa, el control del programa pasa a la línea que sigue al ciclo.

Al igual que el for este bucle puede ejecutarse de 0 a n veces, si la *expresión* es falsa de entrada el bucle no se ejecuta nunca.

En el siguiente ejemplo se muestra una rutina de entrada desde el teclado, la cual se cicla mientras no se pulse A:

```
main()
{
   char carac;

   carac = '\0';
   while( carac != 'A') carac = getchar();
}
```

Antes de entrar al ciclo se inicializa la variable carac a nulo. Después pasa a la sentencia while donde se comprueba si carac no es igual a 'A', como sea verdad entonces se ejecuta la sentencia del bucle (carac = getchar();. La función getchar() lee el siguiente carácter del flujo estándar (teclado) y lo devuelve, que en nuestro ejemplo es el carácter que haya sido tecleado. Una vez que se ha pulsado una tecla, se asigna a carac y se comprueba la condición nuevamente. Después de pulsar **A**, la condición llega a ser falsa porque carac es igual a **A**, con lo que el ciclo termina.

De lo anterior, se tiene que tanto el ciclo for, como el ciclo while comprueban la condición en lo alto del ciclo, por lo que el código dentro del ciclo no se ejecuta siempre.
A continuación mostramos otro ejemplo que genera la siguiente salida en pantalla:

```
x = 3
x = 2
x = 1
```

```
main()
{
   int x=3;

   while( x>0 )
   {
      printf("x = %d\n", x);
      x--;
   }
}
```

Como se observa, dentro del ciclo tenemos más de una sentencia, por lo que se requiere usar la llave abierta y la llave cerrada { ... } para que el grupo de sentencias sean tratadas como una unidad.

Como el ciclo while puede aceptar también expresiones, y no solamente condiciones lo siguiente es válido:

```
while ( x-- );

while ( x = x + 1 );

while ( x += 5 );
```

Si se usan este tipo de expresiones, solamente cuando el resultado de x--, x=x+1 o x+=5 sea cero, la condición fallará y se podrá salir del ciclo.

De acuerdo a lo anterior, podemos realizar una operación completa dentro de la expresión. Por ejemplo:

```
main()
{
   char carac;

   carac = '\0';
   while ( (carac = getchar()) != 'A' )
      putchar(carac);
}
```

En este ejemplo se usan las funciones de la biblioteca estándar getchar(), lee un carácter del teclado y putchar(,) escribe un carácter dado en pantalla. El ciclo while procederá a leer del teclado y lo mostrará hasta que el carácter **A** sea leído.

3.3.3 LA SENTENCIA *do-while*

Al contrario de los ciclos for y while que comprueban la condición en lo alto del bucle, el bucle ***do ... while*** la examina en la parte baja del mismo. Esta característica provoca que un ciclo do ... while siempre se ejecute al menos una vez. La forma general del ciclo es:

```
do {
    sentencia1;
    sentencia2;
} while(expresión );
```

Aunque no son necesarias las llaves cuando sólo está presente una sentencia, se usan normalmente por legibilidad y para evitar confusión (respecto al lector, y no del compilador) con la sentencia while.

En el siguiente programa se usa un ciclo do ... while para leer números desde el teclado hasta que uno de ellos es menor que o igual a 100:

```
main()
{
    int num;
    do
    {
        scanf("%d", &num);
    } while (num>100 );
}
```

Otro uso común de la estructura do ... while es una rutina de selección en un menú, ya que siempre se requiere que se ejecute al menos una vez.

```
main()
{
    int opc;
    printf("1. Derivadas\n");
    printf ("2. Limites\n");
    printf("3. Integrales\n");
    do
    {
        printf("   Teclear una opción:  ");
        scanf("%d", &opc);
        switch(opc)
        {
            case 1:
                printf("\Opción 1 seleccionada\n\n");
                break;
            case 2:
                printf("\tOpcion 2 seleccionada\n\n");
                break;
            case 3:
                printf("\tOpcion 3 seleccionada\n\n");
                break;
            default:
                printf("\tOpcion no disponible\n\n");
                break;
        }
    } while( opc != 1  &&  opc != 2  &&  opc != 3);
}
```

Se muestra un ejemplo donde se reescribe usando do ... while uno de los ejemplos ya mostrados.

```
main()
{
    int x=3;
    do
    {
        printf("x = %d\n", x--);
    } while( x>0 ) ;
}
```

3.3.4 USO DE break, continue, exit y goto

- ***break:*** Como se comento uno de los usos de la sentencia break es terminar un case en la sentencia switch. Otro uso es forzar la terminación inmediata de un ciclo, saltando la prueba condicional del ciclo.
  Cuando se encuentra la sentencia break en un bucle, la computadora termina inmediatamente el ciclo y el control del programa pasa a la siguiente sentencia del ciclo.

  Por ejemplo:

```
main()
{
  int t;
  for(t=0; t<100; t++)
  {
    printf("%d ", t);
    if (t==10) break;
  }
}
```

  Este programa muestra en pantalla los números del 0 al 10, cuando alcanza el valor 10 se cumple la condición de la sentencia if, se ejecuta la sentencia break y sale del ciclo.

- ***continue:*** La sentencia continue funciona de manera similar a la sentencia break. Sin embargo, en vez de forzar la salida, continue fuerza la siguiente iteración, por lo que salta el código que falta para llegar a probar la condición. Por ejemplo, el siguiente programa visualizará sólo los números pares:

```
main()
{
    int x;

    for( x=0; x<100; x++)
    {
        if (x%2)
            continue;
        printf("%d ",x);
    }
}
```

  Finalmente se considera el siguiente ejemplo donde se leen valores enteros y se procesan de acuerdo a las siguientes condiciones:

  a. Si el valor que se ha leído es negativo, se desea imprimir un mensaje de error y se abandona el ciclo.
  b. Si el valor es mayor que 100, se ignora y se continúa leyendo
  c. Si el valor es cero, se desea terminar el ciclo.

```
main()
{
    int valor;

    while( scanf("%d", &valor) == 1 && valor != 0)
    {
        if ( valor<0 )
        {
            printf ("Valor no valido\n");
            break;
            /* Salir del ciclo */
        }

        if ( valor >100)
        {
            printf ("Valor no valido\n");
            continue;
            /* Pasar al principio del ciclo nuevamente */
        }

        printf ("Se garantiza que el valor leido esta entre 1
y 100");
    }
}
```

- ***exit:*** Permite salir anticipadamente de un programa. El prototipo de la función *exit* es: *void exit (int estado);* y se encuentra en el archivo de cabecera <stdlib.h>. El valor de *estado* se devuelve al sistema operativo.
  Se utiliza frecuentemente **exit()** cuando no se satisface una condición obligatoria en la ejecución de un programa. Por ejemplo, imagine un juego de computadora que necesita una tarjeta de gráficos en color. La función **main()** de este juego puede ser algo parecida a ésta:

```
#include<stdlib.h>
main(void)
{
  if(!tarjeta_color()) exit(l);
  jugar( );
}
```

  donde **tarjeta_color()** es una función definida por el usuario que devuelve cierto si encuentra la tarjeta de gráficos en color. Si no se encuentra en el sistema, **tarjeta_color()**devuelve falso, o sea 0, el programa finaliza.

- ***goto:*** La sentencia *goto* (*ir a*) nos permite hacer un salto a la parte del programa que deseemos. En el programa debemos poner etiquetas, estas etiquetas no se ejecutan, es como poner un nombre a una parte del programa. Estas etiquetas son las que nos sirven para indicar a la sentencia *goto* dónde tiene que saltar. El *goto* sólo se puede usar dentro de funciones, y no se puede saltar desde una función a otra.

  Nota: el uso del *goto* es un tema en discusión, el caso es no abusar de ella y tener cuidado.

  Ejemplo:

```
#include <stdio.h>

main()
{
   printf( "Línea 1\n" );
   goto linea3; /*El goto busca la etiqueta linea3 */
   printf( "Línea 2\n" );
   linea3:          /* Esta es la etiqueta */
   printf( "Línea 3\n" );
}
```

Este programa muestra: *Línea 1*
Línea 3
y no se ejecuta el *printf de Línea 2* porque nos lo hemos saltado con el *goto*

4. ARREGLOS Y CADENAS

En el siguiente capítulo se presentan los arreglos y las cadenas. Las cadenas se consideran como un arreglo de tipo *char*.

4.1 ARREGLOS UNIDIMENSIONALES Y MULTIDIMENSIONALES

4.1.1 DEFINICIÓN

Los arreglos son una colección de variables del mismo tipo que se referencian utilizando un nombre común. Un arreglo consta de posiciones de memoria contigua. La dirección más baja corresponde al primer elemento y la más alta al último. Un arreglo puede tener una o varias dimensiones. Para acceder a un elemento en particular de un arreglo se usa un índice.

Resumiendo:

Un arreglo es un conjunto de variables del mismo tipo que tienen el mismo nombre y se diferencian en el índice.

El formato para declarar un arreglo unidimensional es:

```
tipo_de_datos nombre_arreglo [ tamaño ]
```

El ***tipo_de_dato*** es uno de los tipos de datos conocidos (int, char, float...) o de los definidos por el usuario con typdef.

El ***nombre_ arreglo*** es el nombre que damos al arreglo,.este nombre debe seguir las mismas reglas que siguen los nombres de variables.

tamaño es el número de elementos que tiene el arreglo.

Es decir que al declarar un arreglo reservamos en memoria tantas variables del *tipo_de_dato* como las indicadas en *tamaño*

Por ejemplo, para declarar un arreglo de enteros llamado *listanum* con diez elementos se hace de la siguiente forma:

```
int listanum[10];
```

En C, todos los arreglos usan cero como índice para el primer elemento. Por tanto, el ejemplo anterior declara un arreglo de enteros con diez elementos desde **listanum[0]** hasta **listanum[9]**.

La forma en que se puede acceder a los elementos de un arreglo, es la siguiente:

```
listanum[2] = 15; /* Asigna 15 al 3er elemento del arreglo listanum*/

num = listanum[2]; /* Asigna el contenido del 3er elemento a la variable num */
```

El lenguaje C no realiza comprobación de contornos en los arreglos. En el caso de que sobrepase el final durante una operación de asignación, entonces se asignarán valores a otra variable o a un trozo del código, esto es, si se dimensiona un arreglo de tamaño *N*, se puede referenciar el arreglo por encima de *N* sin provocar ningún mensaje de error en tiempo de compilación o ejecución, incluso aunque probablemente se provoque el fallo del programa. Como programador se es responsable de asegurar que todos los arreglos sean lo suficientemente grandes para guardar lo que pondrá en ellos el programa.

NOTA: No hay que confundirse. En la declaración de un arreglo el número entre corchetes es el número de elementos, en cambio cuando ya usamos el arreglo el número entre corchetes es el índice.

También podemos declarar un arreglo bidimensional, para ello usaremos el siguiente formato:

```
tipo_de_dato nombre_del_arreglo[ filas ] [ columnas ];
```

Como se puede apreciar, la declaración es igual que la de un arreglo unidimensional al que le añadimos una nueva dimensión

Por ejemplo un arreglo de enteros bidimensionales se escribirá como:

```
int tabladenums[50][50];
```

Observar que para declarar cada dimensión lleva sus propios corchetes.

Para acceder los elementos se procede de forma similar al ejemplo del arreglo unidimensional, esto es,

```
tabladenums[2][3] = 15; /* Asigna 15 al elemento de la 3ª fila y la 4ª columna*/
num = tabladenums[25][16];
```

De la misma manera C permite arreglos multidimensionales, el formato general es:

```
tipo_de_dato nombre_del_arreglo [tam1][tam2] ... [tamN];
```

EN TODOS LOS CASOS EL NOMBRE DEL ARREGLO ES UN PUNTERO AL PRIMER ELEMENTO DEL ARREGLO.

4.1.2 INICIALIZACIÓN DE ARREGLOS

En C se permite la inicialización de arreglos, al igual que hacíamos con las variables. Recordemos que se podía hacer:

```
int hojas = 34;
```

Pues con arreglos se puede hacer:

```
int temp [10] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25 };
```

Ahora el elemento 0 (que será el primero), es decir temp [0] valdrá 15. El elemento 1 (el segundo) valdrá 18 y así con todos.

Si al inicializar un arreglo se asignan menos valores que los definidos por el tamaño, los elementos a los que no se les asigno valor asumirán el valor 0.

Si se da el caso contrario, metemos más datos de los reservados, dependiendo del compilador obtendremos un error o al menos un warning (aviso). En unos compiladores el programa se creará y en otros no, pero aún así nos avisa del fallo. Debe indicarse que estamos intentando guardar un dato de más, no hemos reservado memoria para él.

Con los arreglos multidimensionales también se puede. Por ejemplo:

```
int num[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
```

Otra forma sería agrupar entre { } cada fila.

```
int temp [3][5] = {
                { 15, 17, 20, 25, 10 },
                { 18, 20, 21, 23, 18 },
                { 12, 17, 23, 29, 16 } };
```

El formato genérico a utilizar sería el siguiente:

```
tipo_de_dato nombre_del_arreglo[ n_filas ][
m_columnas ] = {
            { m_columnas de la fila 1},
            { m_columnas de la fila 2},
            ... ,
            { m_columnas de la fila n},
            };
```

No debemos olvidar el '**;**' al final.

A continuación se muestra un ejemplo que asigna al primer elemento de un arreglo bidimensional cero, al siguiente 1, y así sucesivamente.

```
main()
{
    int t, i, num[3][4];

    for(t=0; t<3; ++t)
        for(i=0; i<4; ++i)
            num[t][i]=(t*4)+i*1;

    for(t=0; t<3; ++t)
    {
        for(i=0; i<4; ++i)
            printf("num[%d][%d]=%d  ", t, i, num[t][i]);
        printf("\n");
    }
}
```

4.2 CADENAS

A diferencia de otros lenguajes de programación que emplean un tipo denominado *string* para manipular un conjunto de símbolos, en C, se debe simular mediante un arreglo de caracteres, en donde la terminación de la cadena se debe indicar con nulo. Un nulo se especifica como `'\0'`. Por lo anterior, cuando se declare un arreglo de caracteres se debe considerar un carácter adicional a la cadena más larga que se vaya a guardar. Por ejemplo, si se quiere declarar un arreglo `cadena` que guarde una cadena de diez caracteres, se hará como:

```
char cadena[11];
```

Se pueden hacer también inicializaciones de arreglos de caracteres en donde automáticamente C asigna el carácter nulo al final de la cadena, de la siguiente forma:

```
char nombre_arr[ tam ]="cadena";
```

Por ejemplo, el siguiente fragmento inicializa cadena con ``hola":

```
char cadena[5]="hola";
```

El código anterior es equivalente a:

```
char cadena[5]={'h','o','l','a','\0'};
```

Para asignar la entrada estándar a una cadena se puede usar la función scanf con la opción `%s`, de igual forma para mostrarlo en la salida estándar, utilizando printf.
Por ejemplo:

```
main()
{
    char nombre[15], apellido [30];

    printf("Introduce tu nombre: ");
    scanf("%s",nombre);
    printf("Introduce tu apellido: ");
    scanf("%s",apellido);
    printf("Usted es %s %s\n", nombre, apellido);
}
```

Vemos cosas curiosas como por ejemplo que en el scanf no se usa el símbolo **&**. No hace falta porque es un arreglo, y ya sabemos que escribir el nombre del arreglo es equivalente a poner &nombre[0].

El lenguaje C no maneja cadenas de caracteres, como se hace con enteros o flotantes, por lo que lo siguiente no es válido:

```
main()
{
    char nombre[40], apellidos[40], completo[80];

    nombre="José María";                /* Ilegal */
    apellidos="Morelos y Pavón";        /* Ilegal */
    completo="Gral."+nombre+apellidos; /* Ilegal */
}
```

5. OTROS TIPOS DE DATOS

En este capítulo se revisa la forma como pueden ser creados y usados en C tipos de datos más complejos y estructuras.

5.1 ESTRUCTURAS

En C una estructura es una colección de variables que se referencian bajo el mismo nombre. Una estructura proporciona un medio conveniente para mantener junta información que se relaciona. Una *definición de estructura* forma una plantilla que se puede usar para crear variables de estructura. Las variables que forman la estructura son llamados *campos, elementos, o miembros de la estructura.*

5.1.1 DEFINICIÓN

El formato genérico para definir una estructura sería:

```
struct nombre_de_la_estructura {
        campos_de_estructura;
        };
```

NOTA: Es importante no olvidar el '**;**' del final, si no a veces se obtienen errores extraños.

Generalmente, todos los campos en la estructura están relacionados lógicamente unos con otros. Por ejemplo, se puede representar una lista de nombres de correo en una estructura. Mediante la palabra clave struct se le indica al compilador que defina una plantilla de estructura.

```
struct datos_contacto
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
};
```

Con el trozo de código anterior *no ha sido declarada ninguna variable*, tan sólo se ha definido el formato. Para declarar una variable, se hará como sigue:

```
struct datos_contacto contacto;
```

dónde **contacto** sería el nombre de la variable de tipo estructura **datos_contacto.**

Se pueden declarar una o más variables cuando se define una estructura, esto debe hacerse entre el **)** y el **;**. Por ejemplo:

```
struct datos_contacto
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
} contacto1, contacto2, contacto3;
```

observar que **datos_contacto** es una *etiqueta* para la estructura que sirve como una forma breve para futuras declaraciones. Como en esta última declaración se indican las variables con esta estructura, se puede omitir el nombre de la estructura tipo.

Las estructuras pueden ser también inicializadas en la declaración:

```
struct datos_contacto contacto1={"Vicente Fernández","Fantasía
2000","Dorado","MMX",12345};
```

5.1.2 REFERENCIA A LOS ELEMENTOS DE UNA ESTRUCTURA

Para referenciar o acceder a un miembro (o campo) de una estructura, C proporciona el operador punto ., por ejemplo, para asignar a **contacto1** otro código, lo hacemos como:

```
contacto1.codigo=54321;
```

5.1.3 DEFINICIÓN DE NUEVOS TIPOS DE DATOS

El **`typedef`** se puede usar para definir nuevos nombres de datos explícitamente, usando algunos de los tipos de datos de C, donde su formato es:

typedef *<tipo> <nombre>;*

Se puede usar `typedef` para crear nombres para tipos más complejos, como una estructura, por ejemplo:

```
typedef struct datos_contacto
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
    char estado[3];
    unsigned int codigo;
} identificacion;

identificacion contacto={"Vicente Fernández","Fantasía
2000","Dorado","MMX",12345};
```

en este caso **identificacion** sirve como una *etiqueta* a la estructura y es opcional, ya que ha sido definido un nuevo tipo de dato, por lo que la etiqueta no tiene mucho uso, en donde **identificacion** es el nuevo tipo de datos y **contacto** es una variable del tipo **identificacion**, la cual es una estructura.

5.1.4 ARREGLOS DE ESTRUCTURAS

Con C también se pueden tener arreglos de estructuras:

```
typedef struct datos_contacto
{
    char nombre[30];
    char calle[40];
    char ciudad[20];
```

```
    char estado[3];
    unsigned int codigo;
} identificacion;

identificacion contacto[1000];
```

por lo anterior, contacto tiene 1000 elementos del tipo identificacion. Lo anterior podría ser accedido de la siguiente forma:

```
contacto[50].codigo=22222;
```

5.1.5 ESTRUCTURA DE ESTRUCTURAS (ANIDADAS)

Es posible crear estructuras que tengan como miembros otras estructuras. Esto tiene diversas utilidades, por ejemplo tener la estructura de datos más ordenada. Imaginemos la siguiente situación: una tienda de música quiere hacer un programa para el inventario de los discos, cintas y cd's que tienen. Para cada título quiere conocer las existencias en cada soporte (cinta, disco, cd), y los datos del proveedor (el que le vende ese producto). Podría pensar en una estructura así:

```
struct inventario {
        char titulo[30];
        char autor[40];
        int existencias_discos;
        int existencias_cintas;
        int existencias_cd;
        char nombre_proveedor[40];
        char telefono_proveedor[10];
        char direccion_proveedor[100];
        };
```

Sin embargo utilizando estructuras anidadas se podría hacer de una forma más ordenada:

```
struct estruc_existencias {
       int discos;
       int cintas;
       int cd;
       };

struct estruc_proveedor {
       char nombre_proveedor[40];
       char telefono_proveedor[10];
       char direccion_proveedor[100];
       };

struct estruc_inventario {
        char titulo[30];
        char autor[40];
        struct estruc_existencias existencias;
        struct estruc_proveedor proveedor;
        } inventario;
```

Ahora para acceder al número de cd de cierto título usaríamos lo siguiente:

```
inventario.existencias.cd
```

y para acceder al nombre del proveedor:

```
inventario.proveedor.nombre
```

5.2 UNIONES

Hemos visto que las estructuras toman una parte de la memoria y se la reparten entre sus miembros. Cada miembro tiene reservado un espacio para él solo. El tamaño total que ocupa una estructura en memoria es la suma del tamaño que ocupa cada uno de sus miembros.

Las uniones tienen un aspecto similar en cuanto a cómo se definen, pero tienen una diferencia fundamental con respecto a las estructuras: los miembros comparten el mismo trozo de memoria. El espacio que ocupa en memoria una unión es el espacio que ocupa el campo más grande. Para entenderlo mejor vamos a ver un ejemplo:

Primero vamos a ver cómo se define una unión:

```
union nombre_de_la_unión
   {
       campos_de_la_unión
   };
```

NOTA: No se debe olvidar el '**;**' después de cerrar llaves '**}**'.

Y aquí va el ejemplo:

```
union numero
{
    short  shortnumero;
    long   longnumero;
    double floatnumero;
} unumero;
```

con lo anterior se define una unión llamada número y una instancia de esta llamada unumero. *numero* es la etiqueta de la unión y tiene el mismo comportamiento que la etiqueta en la estructura.

Los miembros pueden ser accedidos de la siguiente forma:

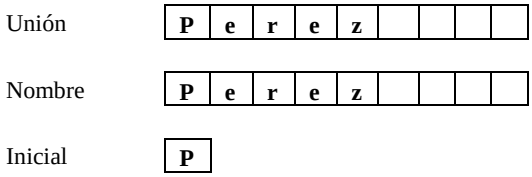
```
printf("%ld\n",unumero.longnumero);
```

con la llamada a la función **printf** se muestra el valor de longnumero.

Cuando el compilador de C esta reservando memoria para las uniones, siempre creará una variable lo suficientemente grande para que quepa el tipo de variable más largo de la unión.

```
union _persona
   {
      char nombre [10];
      char inicial;
   };
```

Creamos una unión y sus elementos son un *nombre* de 10 bytes (nombre[10]) y la inicial (1 byte). Como hemos dicho la unión ocupa el espacio de su elemento más grande, en este caso *nombre*. Por lo tanto la unión ocupa 10 bytes. Las variables *nombre* e *inicial* comparten el mismo sitio de la memoria. Si accedemos a *nombre* estaremos accediendo a los primeros 10 bytes de la unión (es decir, a toda la unión), si accedemos a *inicial* lo que tendremos es el primer byte de la unión.



```
#include <stdio.h>
union _persona
   {
      char sobrenombre [10];
      char inicial;
   } pers;

main()
  {
     printf("Escribe tu nombre: ");
     gets(pers.nombre);
     printf("\nTu nombre es: %s\n", pers.nombre);
     printf("Tu inicial es: %c\n", pers.inicial);
  }
```

Ejecutando el programa:

```
Escribe tu nombre: Perez

Tu nombre es: Perez
Tu inicial es: P
```

Para comprender mejor eso de que comparten el mismo espacio en memoria vamos a ampliar el ejemplo. Si añadimos unas líneas al final que modifiquen sólo la inicial e imprima el nuevo nombre:

```
#include <stdio.h>
union _persona
  {
      char nombre[10];
      char inicial;
  } pers;

main()
  {
     printf("Escribe tu nombre: ");
     gets(pers.nombre);
     printf("\nTu nombre es: %s\n", pers.nombre);
     printf("Tu inicial es: %c\n", pers.inicial);
     /* Cambiamos la inicial */
     pers.inicial='Z';
     printf("\nAhora tu nombre es: %s\n", pers.nombre);
     printf("y tu inicial es: %c\n", pers.inicial);
  }
```

Tendremos el siguiente resultado:

```
Escribe tu nombre: Perez

Tu nombre es: Perez
Tu inicial es: P

Ahora tu nombre es: Zerez
y tu inicial es: Z
```

Aquí queda claro que al cambiar el valor de la inicial estamos cambiando también el nombre porque la inicial y la primera letra del nombre son la misma posición de la memoria.

5.3 ENUMERACIONES

En el capítulo 1 vimos que se podía dar nombre a las constantes con *#define*. Utilizando esta técnica podríamos hacer un programa en que definiríamos las constantes *primero...quinto*.

```
#include <stdio.h>

#define primero 1
#define segundo 2
#define tercero 3
#define cuarto  4
#define quinto  5

main()
{
    int posición;
    posición=segundo;
    printf("posición = %i\n", posición);
}
```

Sin embargo existe otra forma de declarar estas constantes y es con las enumeraciones. Las enumeraciones se crean con enum:

```
enum nombre_de_la_enumeración
  {
        nombres_de_las_constantes,
  };
```

Por ejemplo:

```
enum { primero, segundo, tercero, cuarto, quinto };
```

De esta forma hemos definido las constantes *primero, segundo,... quinto*. Si no especificamos nada la primera constante (primero) toma el valor 0, la segunda (segunda) vale 1, la tercera 2,... Podemos cambiar estos valores predeterminados por los valores que deseemos:

```
enum { primero=1, segundo, tercero, cuarto, quinto };
```

Ahora primero vale 1, segundo vale 2, tercero vale 3,... Cada constante toma el valor de la anterior más uno. Si por ejemplo hacemos:

```
enum { primero=1, segundo, quinto=5, sexto, séptimo };
```

Tendremos: primero=1, segundo=2, quinto=5, sexto=6, séptimo=7.

Con esta nueva técnica podemos volver a escribir el ejemplo de antes:

```
#include <stdio.h>

enum { primero=1, segundo, tercero, cuarto, quinto } posición;
main()
{
    posición=segundo;
    printf("posición = %i\n", posición);
}
```

Las constantes definidas con enum **sólo** pueden tomar valores enteros (pueden ser negativos). Son equivalentes a las variables de tipo *int*.

Un error habitual suele ser pensar que es posible imprimir el nombre de la constante:

```
#include <stdio.h>

enum { primero=1, segundo, tercero, cuarto, quinto } posición;

main()
{
    printf("posición = %s\n", segundo);
}
```

Este ejemplo contiene errores

Es habitual pensar que con este ejemplo tendremos como resultado: *posición = segundo*. Pero no es así, en todo caso tendremos: *posición = (null)*

6. PUNTEROS

6.1 INTRODUCCIÓN

Los punteros son una parte fundamental de C. Si usted no puede usar los punteros apropiadamente entonces esta perdiendo la potencia y la flexibilidad que C ofrece básicamente. El secreto para C esta en el uso de punteros.

C usa los punteros en forma extensiva. ¿Por qué?

- Es la única forma de expresar algunos cálculos.
- Se genera código compacto y eficiente.
- Es una herramienta muy poderosa. .

C usa punteros explícitamente con:

- Arreglos,
- Estructuras y
- Funciones

6.1.1 DEFINICIÓN DE UN PUNTERO

Un puntero es una variable que contiene una dirección de memoria. Normalmente, esa dirección es la posición de otra variable de memoria. Si una variable contiene la dirección de otra variable, entonces se dice que la primera variable apunta a la segunda.

6.1.2 DECLARACIÓN

Si una variable va a contener un puntero, entonces tiene que declararse como tal. Una declaración de un puntero consiste en un tipo base, un * y el nombre de la variable. La forma general es:

```
tipo *nombre;
```

Donde *tipo* es cualquier tipo válido y *nombre* es el nombre de la variable puntero. El tipo base del puntero define el tipo de variables a las que puede apuntar. Técnicamente, cualquier tipo de puntero puede apuntar a cualquier dirección de la memoria, sin embargo, toda la aritmética de punteros esta hecha en relación a sus tipos base, por lo que es importante declarar correctamente el puntero.

6.1.3 OPERADORES DE PUNTEROS

Existen dos operadores especiales de punteros: **&** y *****. El *operador de dirección* (&) devuelve la dirección de memoria de su operando. El *operador de indirección* (*) devuelve el contenido de la dirección apuntada por el operando.

6.1.4 INICIALIZACIÓN

Después de declarar un puntero, pero antes de asignarle un valor, éste contiene un valor desconocido; si en ese instante intenta utilizarlo, probablemente se producirá un error, no sólo en el programa sino también en el sistema operativo. Por convenio, se debe asignar el valor nulo a un puntero que no este apuntando a ningún sitio, aunque esto tampoco es seguro.

La inicialización de punteros puede realizarse de dos formas diferentes:

1. **Forma estática**

   Esto es, asignarle la dirección de memoria de una variable que ya ha sido definida.

   Como en el caso de cualquier otra variable, un puntero puede utilizarse a la derecha de una declaración de asignación para asignar su valor a otro puntero.

Por ejemplo:

```
int x;            /*define la variable entera x*/
int *p1,*p2;      /*define los punteros a entero p1 y p2*/
p1=&x;            /*asigno al puntero p1 la dirección de la
                    variable x*/
p2=p1;      /*asigno al puntero p2 el contenido del puntero
              p1*/
```

Tanto p1 como p2 apuntan a x.
Para tener una mejor idea, considerar el siguiente código:

```
main( )
{
    int x = 1, y = 2;
    int *ap;

    ap = &x;

    y = *ap;

    x = ap;

    *ap = 3;
}
```

Cuando se compile el código se mostrará el siguiente mensaje:

```
warning: assignment makes integer from pointer without a cast.
```

Con el objetivo de entender el comportamiento del código supongamos que la variable x esta en la localidad de la memoria 100, y en 200 y ap en 1000.

Nota: un puntero es una variable, por lo tanto, sus valores necesitan ser guardados en algún lado.

int x = 1, y = 2;
*int *ap;*

ap = &x;

100		200		1000	
x	1	y	2	ap	100

Las variables *x* e *y* son declaradas e inicializadas con 1 y 2 respectivamente, *ap* es declarado como un puntero a entero y se le asigna la dirección de *x* (&x). Por lo que ap se carga con el valor 100.

*y = *ap;*

100		200		1000	
x	1	y	1	ap	100

Después *y* obtiene el contenido de ,a variable por ap. En el ejemplo ap apunta a la localidad de memoria 100 -- la localidad de *x*. Por lo tanto, y obtiene el valor de *x* -- el cual es 1.

x = ap;

100		200		1000	
x	100	y	1	ap	100

Como se ha visto C no es muy estricto en la asignación de valores de diferente tipo (puntero a entero). Así que es perfectamente legal (aunque el compilador genera un aviso de cuidado) asignar el valor actual de ap a la variable x. El valor de ap en ese momento es 100.

**ap = 3;*

100		200		1000	
x	3	y	1	ap	100

Finalmente se asigna un valor al contenido de un puntero (*ap).

Importante: Cuando un puntero es declarado apunta a algún lado. Se debe inicializar el puntero antes de usarlo.

2. Forma dinámica

Existe un segundo método para inicializar un puntero que consiste en asignarle una dirección de memoria utilizando las funciones de asignación de memoria malloc(), calloc(), realloc() y free(), que se verán en detalle en el capítulo siguiente.

6.1.5 PUNTERO NULL

Un puntero nulo no apunta a ninguna dirección de memoria en particular, es decir, no direcciona ningún dato válido en memoria. Para declarar un puntero nulo se utiliza la macro NULL que se encuentra definida en los archivos de cabecera `<stdio.h>, <stdef.h>, <stdlib.h> y <string.h>`.

También se la puede definir en el archivo fuente de la siguiente manera:

```
#define NULL 0
```

Tener en cuenta que nunca se debe utilizar un puntero nulo para referenciar un valor, generalmente se los utiliza en un test condicional para determinar si un puntero ha sido inicializado.

6.1.6 PUNTERO VOID

En C se puede declarar un puntero de modo que apunte a variables de cualquier tipo de dato, el método es declarar el puntero como un puntero void, denominado puntero genérico.

*void *punt; /*declara a punt como un puntero void, puntero genérico*/*

6.1.7 ARITMÉTICA DE PUNTEROS

Existen sólo dos operaciones aritméticas que se puedan usar con punteros: *la suma y la resta.*

No pueden realizarse otras operaciones aritméticas sobre los punteros más allá de la suma y resta entre un puntero y un entero. En particular, no se pueden multiplicar o dividir punteros y no se puede sumar o restar el tipo float o el tipo double a los punteros.

Por ejemplo:

```
main()
{
    float *flp, *flq, a;
    flp= &a;
    *flp = *flp + 10;
    ++*flp;
    (*flp)++;
    flq = flp;
}
```

NOTA: Un puntero a cualquier tipo de variables es una dirección en memoria, la cual es una dirección entera, pero un puntero NO es un entero.

La razón por la cual se asocia un puntero a un tipo de dato, es por que se debe conocer en cuantos bytes esta guardado el dato. De tal forma, que cuando se incrementa un puntero, se incrementa el puntero por un ``bloque" de memoria, en donde el bloque esta en función del tamaño del dato.

Por lo tanto para un puntero a un char, se agrega un byte a la dirección y para un puntero a entero o a flotante se agregan 4 bytes. De esta forma si a un puntero a flotante se le suman 2, el puntero entonces se mueve dos posiciones float que equivalen a 8 bytes.

Cada vez que se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se decrementa, apunta a la posición del elemento anterior. Con punteros a caracteres parece una aritmética normal, sin embargo, el resto de los punteros aumentan o decrecen la longitud del tipo de datos a los que apuntan.

6.2 PUNTEROS CONSTANTES Y PUNTEROS A CONSTANTES

Cuando trabajamos con punteros dos elementos son los que están en juego: el puntero y el objeto apuntado. La palabra clave *const* puede tener dos significados diferentes según el lugar que ocupe en la declaración, calificando como constante al puntero o al objeto apuntado.

Punteros constantes

Para crear un puntero constante se debe utilizar el siguiente formato:

tipo de dato *const nombre puntero;

Cuando se declara un puntero como constante la dirección a la que apunta no puede cambiar en todo el programa. El contenido apuntado si se puede cambiar.

Ejemplo:

```
int x = 10,  j;
int *const punt = &x;   /* puntero constante que apunta a x */
*punt = 20;       /* operación válida */
punt = &j;        /* operación inválida */
```

Punteros a constantes

Para crear un puntero a constante se debe utilizar el siguiente formato:

const tipo de dato * nombre puntero;

Cuando se declara un puntero a constante el contenido apuntado no se puede cambiar en todo el programa. La dirección a la que apunta si se puede cambiar.

Ejemplo:

```
const int x = 10,  j = 29;
const int * punt = &x;  /* puntero que apunta a la constante entera
                           x */
*punt = 20;             /* operación inválida */
punt = &j;              /* operación válida */
```

6.3 PUNTEROS Y ARREGLOS

Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un índice a la dirección de comienzo del arreglo. En esencia, el nombre de un arreglo es un puntero al arreglo. Considerar lo siguiente:

```
int a[10], x;
int *ap;

ap = &a[0];    /* ap apunta a la direccion de a[0] */

x = *ap;       /* A x se le asigna el contenido de ap (a[0] en este
                  caso) */

*(ap + 1) = 100;  /* Se asigna al segundo elemento de 'a' el valor
                     100 usando ap*/
```

Como se puede observar en el ejemplo la sentencia **a[1]** es idéntica a **ap+1**. Se debe tener cuidado ya que C no hace una revisión de los límites del arreglo, por lo que se puede ir fácilmente más allá del arreglo en memoria y sobrescribir otras cosas.

C sin embargo es mucho más sutil en su relación entre arreglos y punteros. Por ejemplo se puede teclear solamente:
ap = a; en vez de *ap = &a[0];* y también **(a + i)* en vez de *a[i]*, esto es, *&a[i]* es equivalente con *a+i.*

Y como se ve en el ejemplo, el direccionamiento de punteros se puede expresar como:
a[i] que es equivalente *a *(ap + i)*

Sin embargo los punteros y los arreglos son diferentes:

- Un puntero es una variable. Se puede hacer ap = a y ap++.
- Un arreglo NO ES una variable. Hacer a = ap y a++ ES ILEGAL.

Esta parte es muy importante, asegúrese haberla entendido.

Es correcto inicializar un puntero con la dirección de un array, en cambio no se puede cambiar la dirección de un array definido en forma estática, pues el array es un puntero constante, esto es, se puede modificar su contenido pero no la dirección que le fue asignada en su definición

Con lo comentado se puede entender como los arreglos son pasados a las funciones. Cuando un arreglo es pasado a una función lo que en realidad se le está pasando es la localidad de su elemento inicial en memoria.

Por lo tanto:

strlen(s) es equivalente a *strlen (&s[0])*

Esta es la razón por la cual se declara la función como:
int strlen(char s[]); y una declaración equivalente es *int strlen(char *s);*
ya que *char s[]* es igual que *char *s.*

La función *strlen()* es una función de la *biblioteca estándar* que regresa la longitud de una cadena. Se muestra enseguida la versión de esta función que podría escribirse:

```
int strlen(char *s)
{
    char *p = s;

    while ( *p!= '\0')
        p++;
    return p - s;
}
```

Se muestra enseguida una función para copiar una cadena en otra. Al igual que en el ejercicio anterior existe en la biblioteca estándar una función que hace lo mismo.

```
void strcpy(char *s, char *t)
{
    while ( (*s++ = *t++)!= '\0' );
}
```

Nota: Se emplea el uso del carácter nulo con la sentencia while para encontrar el fin de la cadena.

6.4 ARREGLOS DE PUNTEROS

En C se pueden tener arreglos de punteros ya que los punteros son variables.

A continuación se muestra un ejemplo de su uso: **ordenar las líneas de un texto de diferente longitud**.

Los arreglos de punteros son una representación de datos que manejan de una forma eficiente y conveniente líneas de texto de longitud variable.

¿Cómo se puede hacer lo anterior?

1. Guardar todas las líneas en un arreglo de tipo char grande. Observando que \n marca el fin de cada línea. Ver **Figura 8.**
2. Guardar los punteros en un arreglo diferente donde cada puntero apunta al primer carácter de cada línea.
3. Comparar dos líneas usando la función de la biblioteca estándar strcmp().
4. Si dos líneas están desacomodadas -- intercambiar (swap) los punteros (no el texto).

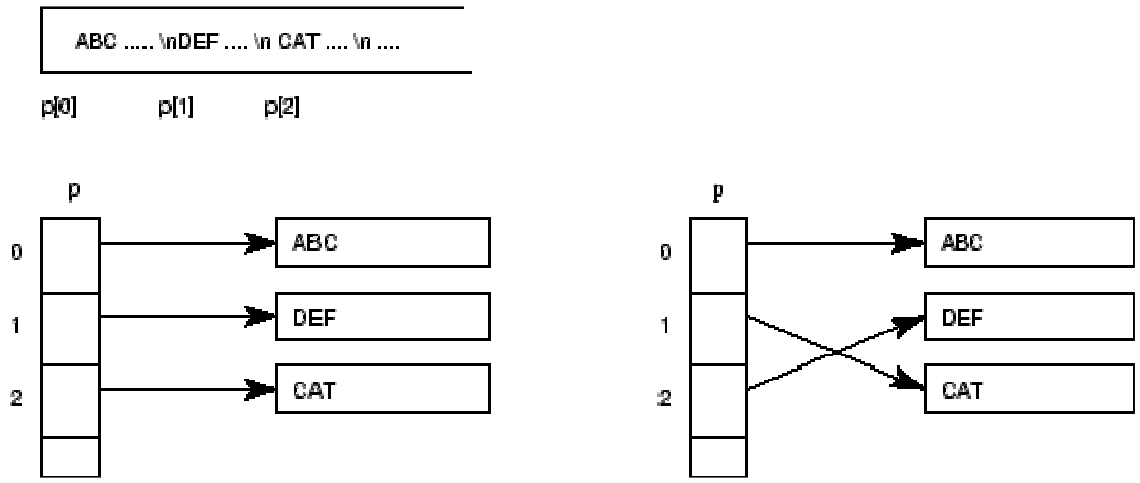


Figura 8: Arreglos de punteros (Ejemplo de ordenamiento de cadenas).

Con lo anterior se elimina:

- el manejo complicado del almacenamiento.
- alta sobrecarga por el movimiento de líneas.

6.5 PUNTEROS A PUNTEROS

Se puede hacer que un puntero apunte a otro puntero que apunte a un valor de destino. Esta situación se denomina indirección múltiple o punteros a punteros. Una variable que es puntero a puntero tiene que declararse como tal. Esto se hace colocando un * adicional en frente del nombre de la variable. Por ejemplo, la siguiente declaración inicial indica al compilador que ptr es un puntero a puntero de tipo float: float **ptr;

6.6 ARREGLOS MULTIDIMENSIONALES Y PUNTEROS

Un arreglo multidimensional puede ser visto en varias formas en C, por ejemplo:
Un arreglo de dos dimensiones es un arreglo de una dimensión, donde cada uno de los elementos es en sí mismo un arreglo.
Por lo tanto, la notación

a[n][m]

nos indica que los elementos del arreglo están guardados renglón por renglón.

Cuando se pasa una arreglo bidimensional a una función se debe especificar el número de columnas -- el número de renglones es irrelevante.

La razón de lo anterior, es nuevamente los punteros. C requiere conocer cuantas son las columnas para que pueda brincar de renglón en renglón en la memoria.

Considerando que una función deba recibir int a[5][35], se puede declarar el argumento de la función como:

f(int a[][35]) { ..... } o aún *f(int (*a)[35]) { ..... }*

En el último ejemplo se requieren los paréntesis (*a) ya que [ ] tiene una precedencia más alta que *.

Por lo tanto:

- int (*a)[35]; declara un puntero a un arreglo de 35 enteros, y por ejemplo si hacemos la siguiente referencia a+2, nos estaremos refiriendo a la dirección del primer elemento que se encuentran en el tercer renglón de la matriz supuesta, mientras que
- int *a[35]; declara un arreglo de 35 punteros a enteros.

Ahora veamos la diferencia (sutil) entre punteros y arreglos. El manejo de cadenas es una aplicación común de esto.
Considera:

```
char *nomb[10];

char anomb[10][20];
```

En donde es válido hacer nomb[3][4] y anomb[3][4] en C.
Sin embargo:

- anomb es un arreglo *verdadero* de 200 elementos de dos dimensiones tipo char.

- El acceso de los elementos anomb en memoria se hace bajo la siguiente fórmula *20*renglón + columna + dirección_base*

- En cambio nomb tiene 10 punteros a elementos.

NOTA: si cada puntero en nomb indica un arreglo de 20 elementos entonces y solamente entonces 200 char estarán disponibles (10 elementos).
Con el primer tipo de declaración se tiene la ventaja de que cada puntero puede apuntar a arreglos de diferente longitud.
Considerar:

*char *nomb[] = {"No mes", "Ene", "Feb", "Mar", ... };*

char anomb[][15] = { "No mes", "Ene", "Feb", "Mar", ... };

Lo cual gráficamente se muestra en la **Figura 9.** Se puede indicar que se hace un manejo más eficiente del espacio haciendo uso de un arreglo de punteros que usando un arreglo bidimensional.

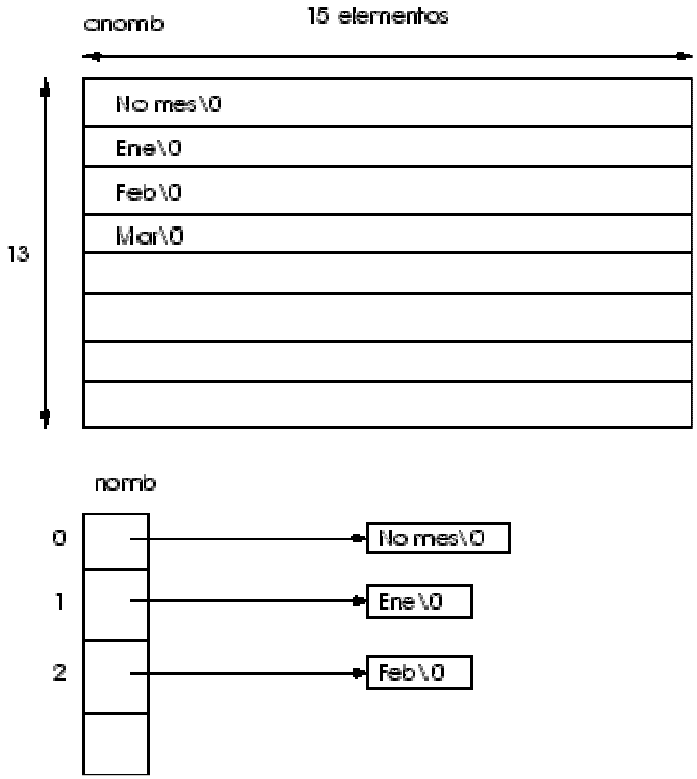


Figura 9: Arreglo de 2 dimensiones VS. arreglo de punteros.

6.7 INICIALIZACIÓN ESTÁTICA DE ARREGLOS DE PUNTEROS

La inicialización de arreglos de punteros es una aplicación ideal para un arreglo estático interno, por ejemplo:

```
func_cualquiera()
{
    static char *nomb[ ] = {"No mes", "Ene", "Feb", "Mar", .... };
}
```

Recordando que con el especificador de almacenamiento de clase *static* se reserva en forma permanente memoria el arreglo, mientras el código se esta ejecutando

6.8 PUNTEROS Y ESTRUCTURAS

Los punteros a estructuras se definen fácilmente y en una forma directa. Considerar lo siguiente:

```
main ( )
{
    struct {COORD  float x, y, z; } punto;

    struct COORD *ap_punto;

    punto.x = punto.y = punto.z = 1;

    ap_punto = &punto;  /* Se asigna punto al puntero */
    ap_punto -> x++;   /*Con el operador -> se accede a los miem
                         bros */
    ap_punto -> y+=2;  /*de la estructura apuntados por ap_punto */
    ap_punto -> z=3;
}
```

Otro ejemplo son las listas ligadas:

```
typedef struct {
    int valor;
    struct ELEMENTO *sig;
} ELEMENTO;

ELEMENTO n1, n2;

n1.sig = &n2;
```

La asignación que se hace corresponde a la **Figura 10**

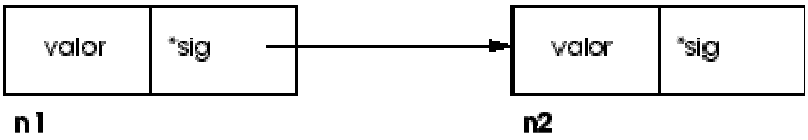


Figura 10: Esquema de una lista ligada con 2 elementos.

Nota: Solamente se puede declarar sig como un puntero tipo ELEMENTO. No se puede tener un elemento del tipo variable ya que esto generaría una definición recursiva la cual no esta permitida. Se permite poner una referencia a un puntero ya que los bytes se dejan de lado para cualquier puntero.

6.9 FALLAS COMUNES CON PUNTEROS

A continuación se muestran dos errores comunes que se hacen con los punteros:

- **No asignar un puntero a una dirección de memoria antes de usarlo**
  *int *x*

  **x = 100;*

lo adecuado será, tener primeramente una localidad física de memoria, digamos

*int *x, y;*
x = &y;
**x = 100;*

- **Indirección no válida**

Supongamos que se tiene una función llamada *malloc()* la cual trata de asignar memoria dinámicamente (en tiempo de ejecución), la cual regresa un puntero al bloque de memoria requerida si se pudo o un puntero a nulo en otro caso.

*char *malloc()*

Supongamos que se tiene un puntero *char *p, entonces:*

```
*p = (char *) malloc(100);  /* pide 100 bytes de la memoria */

*p = 'y';
```

Existe un error en el código anterior. ¿Cuál es?
El * en la primera línea ya que malloc regresa un puntero y *p no apunta a ninguna dirección.
El código correcto deberá ser:

*p = (char *) malloc(100);*

Ahora si malloc no puede regresar un bloque de memoria, entonces p es nulo, y por lo tanto no se podrá hacer:

```
*p = 'y';
```

Un buen programa en C debe revisar lo anterior, por lo que el código anterior puede ser reescrito como:

```
p = (char *) malloc(100):    /* pide 100 bytes de la memoria */
if ( p == NULL )
{
   printf("Error: fuera de memoria\n");

}

else *p = 'y';
```

8. FUNCIONES

8.1 FUNCIONES DE BIBLIOTECA

8.1.1 CONCEPTO

Una biblioteca es una colección de funciones. A diferencia de un archivo objeto, un archivo de biblioteca guarda el nombre de cada función, el código objeto de la función y la información de reubicación necesaria para el proceso de enlace. Cuando un programa referencia a una función contenida en una biblioteca, el enlazador toma esa función y añade el código objeto al programa. De esta forma, sólo se añadirán al archivo ejecutable aquellas funciones que realmente se utilicen en el programa.

El C del estándar ANSI tiene definido el contenido y la forma de la biblioteca estándar de C. Todos los compiladores de Turbo/Borland C/C++ proporcionan todas las funciones definidas por el estándar ANSI. Sin embargo, para permitir el mayor uso y control posible de la computadora, Turbo/Borland C/C++ incorpora muchas funciones adicionales. Mientras que no se transporten los programas que se escriban a un nuevo entorno, es perfectamente válido utilizar esas funciones no estándar.

8.1.2 ARCHIVOS DE CABECERA

Muchas funciones que se encuentran en la biblioteca trabajan con sus propios tipos de datos y con estructuras a las que los programas deben acceder. Estas estructuras y tipos están definidos en los *archivos de cabecera* proporcionados por el compilador, debiéndose incluir (utilizando **#include)** en cualquier archivo que utilice funciones especificas que los referencien. Además, todas las funciones de la biblioteca tienen sus prototipos definidos en un archivo de cabecera. Esto obedece a dos razones. Primero, en C++ todas las funciones han de tener un prototipo. Segundo, aunque en C la inclusión de prototipos es opcional, se aconseja su uso para proporcionar mejores mecanismos para la comprobación de tipos. Incluyendo los archivos de cabecera que corresponden a las funciones estándar utilizadas en un programa en C, se pueden detectar posibles errores de discordancia de tipos.

8.1.3 FUNCIONES ESPECÍFICAS

8.1.3.1 FUNCIONES DE ENTRADA SALIDA <*stdio.h*> <*conio.h*>

- **FUNCIONES DE E/S CON FORMATO**

La biblioteca estándar de Turbo C contiene dos funciones que realizan entrada y salida con formato sobre los tipos de datos incorporados: ***printf( )*** y ***scanf( ).*** El término *con formato* se refiere al hecho de que estas funciones lean y escriban datos en varios formatos bajo control del programador.

- ✓ **FUNCIÓN *printf( )***

Es una función de salida por pantalla con formato, escribe datos en la misma con un formato determinado. Esta función puede operar sobre cualquiera de los tipos de datos existentes, incluyendo caracteres, cadenas y números. El formato de llamada a esta función es:

```
printf ( "cadena de control", arg1, arg2, arg3,.....);
```

La cadena de control es un conjunto de caracteres entrecomillados que contiene dos tipos de elementos:

- caracteres ordinarios que desean imprimirse y que se mostrarán por pantalla tal cual.
- especificadores de conversión, cada uno de los cuales causa la conversión e impresión de cada uno de los argumentos de la lista que aparecen a continuación de la cadena de control (arg1, arg2, arg3, etc.).

La lista de argumentos (arg1, arg2, etc.) está formada por constantes, expresiones o variables cuyos valores se desean imprimir.

Habrá un *especificador de conversión* por cada uno de los argumentos de la lista, si no hay suficientes o no son del tipo correcto se producen resultados confusos. La sintaxis es la siguiente:

* comienza con el carácter **%**

* termina con un **carácter de conversión**, que es una letra que indica con que forma se imprimirán los argumentos de la lista. Si el carácter después del **%** no es una especificación de conversión válida, el comportamiento no está definido. Los *especificadores de conversión* válidos son:

Código	Formato
%c	Un único carácter
%d	Número entero decimal
%i	Número entero decimal
%e	Notación científica
%f	Dec. en pto. flotante (predeterminado a 6 dec.)
%g	Usa %e ó %f, el más corto (pred. a 6 dec.)
%o	Octal
%s	Cadena de caracteres (imprime hasta el '\0')
%u	Número decimal sin signo
%x	Hexadecimales
%%	Imprime un signo %
%p	Muestra un puntero
%n	El argumento asociado es un puntero a entero al que se asigna el número de caracteres escritos hasta entonces

Entre el % y el carácter de conversión puede haber modificadores que especifiquen la longitud de campo, el número de decimales y el ajuste a la izquierda. Estos modificadores, en orden son:

- **Un signo menos**, *que especifica el ajuste a la izquierda del argumento convertido*. Por omisión, todas las salidas están ajustadas a la derecha: si la longitud del campo es mayor que la del dato a imprimir, el dato se situará en la parte derecha del campo.Se puede forzar a que la salida sea ajustada a la izquierda situando un signo menos directamente después del %. Por ejemplo, **%-30s**
- **Un número**, *que especifica el ancho mínimo de campo*. El argumento convertido será impreso dentro de un campo de al menos este ancho. Si es necesario será llenado de blancos a la izquierda (o a la derecha, si se requiere ajuste a la izquierda) para completar la amplitud del campo. Si el argumento (cadena o número) es más largo que el mínimo, el mismo se imprimirá en toda su longitud, aunque sobrepase el mínimo. *Si se quiere rellenar con ceros*, se ha de poner **un 0 antes del especificador de longitud de campo**. Por ejemplo, **%05d** rellenará con ceros un número con menos de 5 dígitos para que su longitud total sea cinco.
- **Un punto**, que separa el ancho de campo de la precisión.

- **Un número**, *la precisión*, que especifica el número de posiciones decimales que se han de imprimir para un número en punto flotante; si se aplica a cadenas o a enteros, el numero que sigue al punto especifica la longitud máxima del campo, si el argumento es más largo que la longitud máxima de campo especificada, se truncarán los caracteres finales a la derecha. Por ejemplo, **%10.4f** imprime un número de al menos 10 caracteres con 4 posiciones decimales; **%5.7s** imprime una cadena de al menos cinco caracteres de longitud y no más de siete.
- **Una h** si el entero se imprimirá como *short* (hd, hi, ho, hu, hx) y una **l** si se imprimirá como *long* (ld, li, lo, lu, lx), o como *double* (le, lf, lg).

Se debe tener en cuenta que existen 2 métodos para manejar sentencias *printf( ) largas*:

1 – Repartir una sentencia en dos líneas, teniendo en cuenta que podemos partir la línea entre argumentos, pero no en la mitad del texto entre comillas (cadena de control).

2 – Emplear dos sentencias printf() para imprimir las misma línea.

Sentencia printf( )	**Salida**											
(":%-5.2f:", 123.234)	:	1	2	3	.	2	3	:				
(":%5.2f:", 3.234)	:		3	.	2	3	:					
(":%10s:", "hola")	:							h	o	l	a	:
(":%-10s:", "hola")	:	h	o	l	a							:
(":%5.7s:", "123456789")	:	1	2	3	4	5	6	7	:			
(":%d:%c:", 15, 15)	:	1	5	:	*	:						
(":%d:%c:", 'A', 'A')	:	6	5	:	A	:						
(":%12.5s:", "Esto es una prueba")	:						E	s	t	o		:
(":%05d:", 336)	:	0	0	3	3	6	:					
(":%-05d:", 336)	:	3	3	6			:					

✓ **FUNCIÓN *scanf( )***

Esta función se comporta como la inversa de *printf( )*, puede leer todos los tipos de datos que suministra el compilador y convierte los números automáticamente al formato interno apropiado. El formato de llamada a esta función es:

scanf("cadena de control", lista de argumentos);

La lista de argumentos (arg1, arg2, etc.) está formada por *punteros a variables* que indican dónde deberá almacenarse la entrada correspondientemente convertida. Las reglas a seguir son:

- Si desea leer un valor perteneciente a cualquiera de los tipos básicos, coloque el nombre de la variable precedido por un **&**.
- Si lo que desea es leer una variable de tipo string (tira de caracteres), **no** anteponga el **&**.

La cadena de control es un conjunto de caracteres entrecomillados que consta de 3 tipos de elementos:

- Blancos o tabuladores, los cuales son ignorados.

- Caracteres ordinarios (no %), que se espera coincidan con el siguiente carácter que no sea espacio en blanco del flujo de entrada.
- Especificadores de conversión, que indican a scanf( ) que tipo de dato se va a leer a continuación.

Habrá un *especificador de conversión* por cada uno de los argumento de la lista. La sintáxis es la siguiente:

- ★ comienza con el carácter **%**
- ★ un carácter optativo de supresión de asignación *****
- ★ un **número** optativo que especifica la longitud máxima de campo
- ★ una **h** para indicar que se ingresará un *entero corto* (hd, hi, ho, hu, hx), una **l** para indicar que se ingresará un *entero largo* (ld, li, lo, lu, lx) o un *double* (le, lf)
- ★ termina con un **carácter de conversión**, que es una letra que indica la interpretación que se hará del campo de entrada. Los especificadores de conversión válidos son:

Código	Formato
%c	Leer un único carácter
%d	Leer un número entero decimal
%i	Leer un número entero decimal
%e	Leer un número en coma flotante
%f	Leer un número en coma flotante
%o	Leer un número octal
%s	Leer una cadena de caracteres
%h	Leer un entero corto
%x	Leer un número hexadecimal
%p	Leer un puntero
%n	Recibe un valor entero igual al número de caracteres leídos hasta entonces

- Un carácter en blanco en la cadena de control de scanf() hace que salte uno o más caracteres en blanco en la secuencia de entrada.
- Un carácter distinto de espacio en blanco en la cadena de control hace que la función scanf() lea y descarte los caracteres que coincidan con el mismo en la secuencia de entrada.
- Los elementos de datos de entrada deben estar separados por espacios en blanco, tabuladores, o saltos de líneas; los símbolos de puntuación tales como las comas, los ; y similares no cuentan como separadores.
- Un ***** entre el % y el especificador de conversión hace que se lea el dato pero que no lo asigne.
- scanf() se detiene cuando termina con su lista de argumentos, o cuando alguna entrada no coincide con la especificación de control. La siguiente llamada a scanf() continúa la búsqueda inmediatamente después del último carácter que ya fue convertido.
- Cuando se utiliza un número *n* como modificador de longitud máxima de campo, si la secuencia de entrada en más extensa que esta longitud, la primera llamada a scanf() asigna hasta el carácter que ocupa la posición ***n*** incluído, en la próxima llamada a scanf() los restantes caracteres serán asignados.

- La asignación a un campo puede terminar antes de lo especificado si se encuentra un espacio en blanco, si lo hubiera continúa con el siguiente campo.

```
char cad[30];
scanf("%20s", cad);
 /* si la secuencia de entrada es: abcdefghijklmnopqrstuvwxyz, la 1ª llamada a scanf() asigna hasta el carácter t incluído,si se vuelve a llamar a scanf() los caracteres uvwxyz se asignarán a cad, desde la posición 0.  */
```

scanf( )	***Ingresa***	***a***	***b***	***c***
("%d", &a)	10	10		
("%d%d", &a, &b)	10 30	10	30	
("%d%*c%d", &a, &c)	10/20	10		20
("%d/%d/%d",&a,&b,&c)	12/10/98	12	10	98
("%d%c%d",&a,&b,&c)	12$36	12	$	36
("%d%d%s", &a, &b, &c)	1 2 hola	1	2	hola

***** Otra posibilidad de **scanf()** viene dada por el *juego de inspección*. Un juego de inspección define un conjunto de caracteres que pueden leerse utilizando **scanf()** y ser asignados a la correspondiente cadena de caracteres. La función **scanf()** lee caracteres y los asigna a la cadena mientras que sean miembros del juego de inspección. Cuando se introduce un carácter que no se corresponde con ninguno del juego de inspección, **scanf()** termina en nulo la cadena y pasa al siguiente campo. En un juego de inspección puede especificarse un rango utilizando un guión, o un juego inverso poniendo como primer carácter del juego un circunflejo (^). Cuando se pone ^ se indica a **scanf()** que ha de aceptar cualquier carácter que no esté definido en el juego de inspección.

Ejemplos:

"%[xyz]%d", s1, s2 / indica que solo leerá caracteres x, y ,z; al 1º distinto pasa al siguiente campo */*

"%[0123456789]%s", s1, s2 / usa un juego de inspección para leer dígitos en s1. Tan pronto como se introduce un carácter que no sea un dígito, se termina s1 con un nulo y se leen caracteres en s2 hasta que encuentra el siguiente carácter de espacio en blanco. */*

"%[a-zA-Z]%[0-9]", s1, s2 / aceptará sólo letras, minúsculas o mayúsculas en s1 y sólo dígitos en s2*/*

"%[^0-9]%[^a-zA-Z]", s1, s2 / aceptará no números en s1 y no letras en s2 – por el ^ que antecede al juego de caracteres de cada uno de los rangos */*

"%[^\n]", s1 / aceptará todo tipo da caracteres incluído el espacio en blanco – por el ^ que antecede al \n el ingreso se cortará cuando se tipee enter*/*

➢ FUNCIONES DE ENTRADA / SALIDA SIN FORMATO

La biblioteca estándar de Turbo C proporciona varias funciones para leer o escribir un carácter a la vez, de las cuales **getchar()** y **putchar()** son las más simples. También veremos la función **getche()**, que es similar a getchar(). En términos de complejidad y potencia le siguen **gets()** y **puts()**, que permiten leer y escribir cadenas de caracteres.

✓ **FUNCIÓN *getchar( )***

Cada vez que se invoca esta función toma un solo carácter del teclado y lo entrega a un programa en ejecución. La función ***getchar( )*** carece de argumentos, simplemente captura el siguiente carácter y se otorga a sí misma el valor de dicho carácter. El carácter devuelto por la función ***getchar( )*** puede ser asignado a una variable, esto es, después de hacer

c = getchar( );

la variable c contiene el siguiente carácter de entrada.

La función ***getchar( )*** funciona con buffer de línea, no procesa hasta que no se pulsa *<enter>*.

✓ **FUNCIÓN *putchar( )***

Cada vez que se invoca, esta función toma un carácter de un programa en ejecución y lo envía a la pantalla. La función ***putchar( )*** posee argumento, éste puede ser un carácter, una variable o una función cuyo valor sea un único carácter. Cada vez que se la invoca muestra el contenido del argumento.

Ejemplos:

```
putchar('S');
putchar(c);
putchar('007');
putchar(getchar());
```

Ejemplos:

a- El siguiente programa toma un carácter del teclado y lo muestra por pantalla.

```
#include <stdio.h>
main( )
{
    char ch;
    ch = getchar();
    putchar(ch);
}
```

b- Una forma totalmente compacta de escribir este programa y que no utiliza variables sería.

```
#include <stdio.h>
main( )
{
   putchar(getchar( ));
}
```

NOTA: en ambos casos se procesa un único carácter, ya que se ha llamado a la función getchar() una sola vez.-

✓ **FUNCIONES *getche( ) y getch( ) <conio.h>***

Su funcionamiento es similar al de *getchar()*, la diferencia radica en que como no tiene buffer de entrada, el eco de la tecla pulsada aparece en la pantalla inmediatamente, no espera un retorno de carro. Al igual que *getchar* no posee argumento. Requiere el archivo de cabecera <**conio.h**>.-

La función *getch( )* es igual a la anterior, excepto que no hace eco de la pulsación de tecla en pantalla.

✓ **FUNCIÓN *gets(argumento)***

Cada vez que se invoca, esta función lee una cadena de caracteres introducida por teclado. Con ***gets( )*** se pueden introducir caracteres hasta que se pulse [enter], esto hace que se coloque un terminador nulo al final de la cadena y ***gets( )*** termina. Esta cadena se guarda en el arreglo que tiene ***gets( )*** como argumento. Se puede corregir la entrada con la tecla de retroceso antes de pulsar *[enter]*.

✓ **FUNCIÓN *puts(argumento)***

Cada vez que se invoca, esta función imprime una cadena en la pantalla, la que tiene como argumento; además reconoce los mismos códigos de barra invertida que printf( ), tales como \n, \t, \b, etc. El argumento de ***puts( )*** puede ser una secuencia de escape, un puntero a una cadena de caracteres (nombre de la cadena) o una constante de cadena.

La función ***puts( )***, luego de mostrar pasa automáticamente el cursor a la línea siguiente, por lo tanto, si se coloca \n dejará una línea en blanco.

Ejemplos:

```
puts('\n');
puts("hola");
puts(cadena);
puts(cadena[3]);
```

Otro ejemplo:

El siguiente programa declara un arreglo de caracteres, lo carga y luego realiza distintas salidas por medio de la función *puts( ).*

```
#include<stdio.h>
main( )
{
    char nombre[20];              /* declaro un arreglo de 20caracteres*/
    gets(nombre);                 /* cargo la cadena */
    puts("Muestro la cadena");/* muestro un cartel */
    puts(nombre);                 /* muestro la cadena completa */
    puts(&nombre[7]);  /*muestro la cadena a partir del elemento indicado */
}
```

8.1.3.2 FUNCIONES DE MANIPULACIÓN DE PANTALLA

<conio.h>

- **clrscr( )** ➔ clear screen, limpia la pantalla activa.
- **clreol( )** ➔ clear end of line, borra desde donde está el cursor hasta el final de la línea actual. Por lo general se utiliza después de gotoxy(x,y).
- **gotoxy(x,y)** ➔ posiciona el cursor en la columna **x** (varía de 1 a 80) y LA fila **y** (varía de 1 a 25) especificadas. Por lo general se utiliza conjuntamente con alguna función de E/S.
- **delline( )** ➔ borra la línea de la ventana activa que contenga el cursor
- **insline( )** ➔ inserta una línea en blanco en la posición actual del cursor.
- **flushall( )** ➔ limpia el buffer de entrada
- **kbhit( )** ➔ se utiliza para determinar si se ha pulsado una tecla o no. Si el usuario ha pulsado una tecla, esta función devuelve verdadero (!= 0), pero no lee el carácter. Si hay una pulsación de tecla esperando se puede leer con getche( ) o getch( ). Si no hay pendiente ninguna pulsación de tecla *kbhit( )* devuelve falso (0)
- **textbackground(color)** ➔ establece el color de fondo de una pantalla de texto
- **textcolor(color)** ➔ establece el color de los caracteres en una pantalla de texto
- **wherex( )** ➔ retorna la columna en la que se encuentra el cursor
- **wherey( )** ➔ retorna la fila en la que se encuentra el cursor

<dos.h>

- **sleep(segundos)** ➔ interrumpe temporalmente la ejecución del programa durante la cantidad de segundos que se le especifique.
- **delay(miliseg)** ➔ interrumpe temporalmente la ejecución del programa durante la cantidad de milisegundos que se le especifique.

8.1.3.3 FUNCIONES PARA MANEJO DE CARACTERES <*ctype.h*>

Las funciones de caracteres están declaradas de forma que tomen un argumento entero y el valor de retorno es un int. Sin embargo, sólo se utiliza el byte menos significativo. Por tanto, se puede usar un argumento de tipo carácter, ya que automáticamente se convierte a **int** en el momento de la llamada. Las funciones regresan **verdadero** (diferente de 0) si el argumento satisface la condición, y **falso** (cero) si no lo hace.

Función	Tipo	Propósito
isalpha(c)	int	Determina si el argumento es alfabético
isdigit(c)	int	Determina si el argumento es un dígito
isalnum(c)	int	Determina si el argumento es alfanumérico
isspace(c)	int	Determina si el argumento es un espacio en blanco
ispunct(c)	int	Determina si el argumento es un carácter de puntuación o un espacio
islower(c)	int	Determina si el argumento es una minúscula
isupper (c)	int	Determina si el argumento es una mayúscula
tolower(c)	int	Convierte **c** a minúscula y retorna el carácter cambiado
toupper(c)	int	Convierte **c** a mayúscula y retorna el carácter cambiado

8.1.3.4 FUNCIONES PARA MANEJO DE CADENAS <*string.h*>

Función	Tipo	Propósito
strcmp(s1,s2)	int	Compara alfabéticamente las cadenas **s1** y **s2.** Retorna un valor negativo si **s1<s2,** 0 si son iguales y un valor positivo si **s1>s2.**
strcmpi(s1,s2)	int	Igual a la anterior pero no diferencia mayúsculas de minúsculas.
strncmp(s1,s2,n)	int	Compara hasta **n** caracteres de **s1** con **s2.** Retorna un valor negativo si **s1<s2,** 0 si son iguales y un valor positivo si **s1>s2.**
strlen(s1)	int	Retorna la longitud de **s1** sin contar el \0
strcpy(s1,s2)	char *	Copia la cadena **s2** en la cadena **s1.** Retorna la cadena **s1**
strncpy(s1,s2,n)	char *	Copia hasta **n** caracteres de la cadena **s2** a la cadena **s1.** Retorna **s1**
strrev(s1)	char *	Invierte la cadena **s1** sobre si misma y la retorna
strcat(s1,s2)	char *	Concatena la cadena **s2** al final de **s1** y retorna **s1**
strncat(s1,s2,n)	char *	Concatena hasta n caracteres de **s2** al final de **s1.** Retorna **s1**
strlwr(s1)	char *	Convierte la cadena **s1** a minúscula y la retorna
strupr(s1)	char*	Convierte la cadena **s1** a mayúscula y la retorna
strstr(s1,s2)	char*	Retorna un puntero a la primera ocurrencia de la cadena **s2** en **s1**, o NULL si no está presente
strchr(s1, c)	char *	Retorna un puntero a la 1ª ocurrencia de **c** en **s1,** o NULL si no está
strrchr(s1,c)	char*	Retorna un puntero a la última ocurrencia de **c** en **s1,** o NULL si no está
strpbrk(s1,s2)	char*	Retorna un puntero a la 1ª ocurrencia en **s1** de cualquier carácter de **s2**, o NULL si no está
strtok(s1,s2)	char*	Busca en **s1** *tokens* delimitados por caracteres de **s2.** Retorna un puntero al *token* o NULL cuando ya no encuentra *tokens*

8.1.3.5 FUNCIONES MATEMÁTICAS <*math.h*>

En este *header* se declaran funciones y macros matemáticas. En la siguiente tabla **x** e **y** son de tipo double, n es un int y todas las funciones regresan double.

Función	Tipo	Propósito
pow (x, y)	double	Calcula y retorna x^y (error si x = 0 y y ≤ 0, o x < 0 y y no es entero)
sqrt(x)	double	Calcula y retorna la raíz cuadrada de x, x ≥0
fabs(x)	double	Calcula y retorna el valor absoluto de x
modf(x,double*p)	double	Divide x en parte entera y fraccionaria
fmod(x, y)	double	Retorna el resto de de la división entera entre los flotantes x e y

8.1.3.6 FUNCIONES VARIAS <*stdlib.h*>

En este *header* se declaran funciones para conversión numérica, asignación de memoria y tareas semejantes

Función	Tipo	Propósito
atof (s1)	double	Convierte la cadena **s1** a double
atoi (s1)	int	Convierte la cadena **s1** a entero
atol (s1)	long	Convierte la cadena **s1** a entero largo
itoa(n,s1,base)	char*	Convierte un valor numérico entero en una cadena de texto
rand ( )	int	Genera y devuelve un entero aleatorio entre 0 y RAND_MAX

randomize ( )	void	Inicializa el generador de nros. aleatorios usando la función time( )
random(num)	int	Devuelve un nro. aleatorio que se encuentra entre 0 y num-1

8.1.3.7 FUNCIONES DE FECHA Y HORA <*time.h*>

En este *header* se declaran los tipos y funcionespara manipulación de fecha y hora.

Función	Tipo	Propósito
time (*p)	long int	Regresa la fecha y hora actual del calendario, -1 si no está disponible
asctime(*p)	char*	Convierte la información de las estructuras de fecha y hora a una cadena

8.2 FUNCIONES CREADAS POR EL PROGRAMADOR

8.2.1 INTRODUCCIÓN

El C utiliza funciones de biblioteca con el fin de realizar un cierto número de operaciones o cálculos de uso común. Sin embargo, C también permite al programador definir sus propias funciones que realicen determinadas tareas. Esto permite dividir un programa en cierto número de componentes más pequeñas.

8.2.1.1 DEFINICIÓN

Una ***función*** es un conjunto de declaraciones, definiciones, expresiones y sentencias que realizan una tarea específica y bien definida. Todo programa en C consta de una o más funciones. Una de estas funciones se debe llamar ***main***. La ejecución del programa siempre comenzará por la de las instrucciones contenidas en main.

8.2.1.2 CARACTERÍSTICAS

- Cuando se accede a una función desde alguna parte del programa, se ejecutan las instrucciones de que consta. Se puede acceder a una misma función desde varios puntos del programa. Una vez completada la ejecución de una función, se devuelve el control al punto desde el que se accedió a ella.
- A la función se le puede pasar información mediante unos identificadores denominados ***argumentos*** o ***parámetros***.
- Las funciones terminan y regresan automáticamente al procedimiento que las llamó.
- En C una función puede devolver datos mediante argumentos y puede devolver también un valor.
- Las funciones pueden ser llamadas desde la función main o desde otras funciones.
- Nunca se debe llamar a la función main desde otro lugar del programa.

8.2.1.3 VENTAJAS DE USO

Hay tres razones fundamentales para el uso de funciones en C:

1) Evitar la repetición innecesaria de código.

2) Independencia: separando el código en funciones modulares se facilita el diseño y la comprensión de los programas.
3) El realizar funciones independientes de la función principal e independientes entre sí permite que las funciones puedan tener sus propias variables "privadas", es decir, estas variables no pueden ser accedidas desde fuera de la función. Esto significa que el programador no necesita preocuparse por el uso accidental de los mismos nombres de variables en otros puntos del programa.

8.2.2 ESTRUCTURA DE LAS FUNCIONES

Al emplear una función se ven involucrados tres elementos del programa: el prototipo de la función, la declaración de la función y la llamada a la función.

Ejemplo: Programa que llama a una función que calcula el promedio entre dos números enteros y luego muestra el resultado

```
#include <stdio.h>
float buscaprom(int num1, int num2);    /*prototipo*/
main()
{
   int a=7, b=10;
   float resultado;
   resultado = buscaprom(a, b);          /*llamada*/
   printf("Promedio=%f\n",resultado); }
float buscaprom(int num1, int num2)    /*declaración o encabezado*/
{
   float promedio;
   promedio = (num1 + num2) / 2.0; return(promedio);
}
```

8.2.2.1 PROTOTIPO DE LA FUNCIÓN

Este es el primer elemento relacionado con una función. Está formado por una línea antes del comienzo de main( ). El formato general de la línea del prototipo es:

tipo_de_retorno nombre_función(lista_de_parámetros);

Su propósito es decirle al compilador el nombre de la función, el tipo de dato que devuelve la función (si hay alguno) y el número y tipo de los argumentos de la función (si los hay) para que este pueda chequear si los tipos de los datos enviados a la y devueltos por la función se corresponden con lo especificado.

Nota:

***** Para que un programa en C sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones.

***** Recordar que los tipos de datos del prototipo y de la declaración de la función deben coincidir.

Ejemplos:

Prototipo de una función de nombre ***factorial*** que recibe como argumento el contenido de una variable entera y devuelve un entero largo:

long int factorial (int);

Prototipo de una función de nombre ***factorial*** que recibe como argumento el contenido de una variable entera y no devuelve ningún valor:

void factorial (int);

8.2.2.2 DECLARACIÓN DE FUNCIONES

Se denomina declaración de la función a la función propiamente dicha. Es de la forma:

tipo_de_retorno nombre_función **(tipo y nombre argumentos)**
{
declaración de variables locales;
código de la función;
}

Notar que la primera línea no finaliza con punto y coma.

Tener en cuenta que los argumentos o parámetros formales de la función y sus variables locales se destruirán al finalizar la ejecución de la misma.

✓ ***tipo_de_retorno***

Indica que tipo de dato retornará la función, puede ser cualquiera de los tipos de datos válidos que conocemos. Si la función no devuelve ningún valor se debe poner ***void***, por defecto, es decir, si no indicamos el tipo de retorno, se asumirá que la función devolverá un valor de tipo entero (int).

✓ ***nombre_función***

Es el nombre que le daremos a la función. El mismo debe seguir las siguientes reglas:

- ★ Son significativos los primeros 8 caracteres.
- ★ Puede estar formado por letras, dígitos y guión bajo.
- ★ Debe comenzar con una letra.
- ★ No debe coincidir con ninguna palabra reservada.
- ★ No debe coincidir con el nombre de ninguna variable
- ★ No debe coincidir con el nombre de ninguna otra función, ya sea de biblioteca o creada por el programador.

✓ ***lista_de_parámetros***

Son los argumentos que recibe la función. Esta lista estará formada por una serie de nombres de variables acompañadas por su tipo y separadas entre sí por comas. Los argumentos de una función no son más que variables locales que reciben un valor. Este valor se lo enviamos al hacer la llamada a la función. No siempre se tienen que incluir parámetros en una función, pueden existir funciones que no reciban argumentos, en este caso se indicará poniendo ***void.***

Los argumentos, que se denominan en la definición de la función ***argumentos o parámetros formales***, permiten la transferencia de información desde el punto del programa en donde se llama a la función a ésta. (Los argumentos correspondientes en la llamada a la función se denominan ***argumentos actuales o reales***). Los identificadores utilizados como ***argumentos o parámetros formales*** son locales en el sentido de que no son reconocidos fuera de la función.

Cada argumento formal debe tener el mismo tipo de datos que el correspondiente argumento actual. Esto es, cada argumento formal debe ser del mismo tipo que el dato que recibe desde el punto de llamada.

En resúmen, los ***argumentos o parámetros formales***:

- Se declaran en la definición de parámetros de las funciones. Si una función usa argumentos, debe declarar las variables que van a aceptar los valores de los mismos. Esas variables son los parámetros formales de la función.
- Se comportan como cualquier otra variable local de la función y pueden usarse en cualquier expresión válida de C.
- Se declaran tras el nombre de la función y entre ( ).
- Al igual que las variables locales, son dinámicas y se destruyen al salir de la función.
- Deben ser del mismo tipo que los argumentos que se usan en la llamada a la función, de lo contrario los resultados pueden ser inesperados.

✓ ***código de la función***

Es el conjunto de sentencias que serán ejecutadas cuando se realice la llamada a la función. También se suele llamar *cuerpo de la función*.

Las funciones terminan y regresan automáticamente al procedimiento que las llamó cuando se encuentra la llave de cierre de la función (**}**), o bien, se puede forzar el regreso antes usando la sentencia ***return.***

★ **SENTENCIA return**

Se devuelve información desde la función hasta el punto de llamada mediante la sentencia ***return***. En una misma función podemos tener más de una instrucción ***return***.

La forma de retornar un valor es la siguiente:

return (valor o expresión);

El valor devuelto por la función debe asignarse a una variable. De lo contrario, el valor se perderá.

Se devuelve el valor de *expresión* al punto de llamada. La expresión es opcional y si se omite, la sentencia ***return*** simplemente devuelve el control al punto del programa desde donde se llamó a la función, sin ninguna transferencia de información.

Sólo se puede incluir una expresión en la sentencia ***return***. Por tanto, una función sólo puede devolver un valor al punto de llamada mediante la sentencia ***return***. Para que la función devuelva más de un valor es necesario otro mecanismo.

Declaración de una función que devuelve un valor entero largo:

```
long int factorial (int n)
 {
   long int f=1;
   int i;
   for (i=2;i<=n;++i)
   f*=i;
   return (f);
 }
```

Declaración de una función que no devuelve ningún valor:

```
void factorial (int n)
{
  long int f=1;
  int i;
  for (i=2;i<=n;++i)
  f*=i;
  printf("el factorial de %d es %ld\n",n,f);
}
```

8.2.2.3 LLAMADA DE LA FUNCIÓN

La llamada a una función se realiza de forma diferente según que la función devuelva o no un valor al punto de llamada.

Si la función no devuelve ningún valor (void) se accede a ella especificando su nombre, seguido de una lista de argumentos cerrados entre paréntesis y separados por comas.

Si la función devuelve un valor se debe acceder a ella desde una expresión.

En ambos casos si la función no requiere argumentos se deben especificar los paréntesis vacíos.

Los argumentos actuales pueden ser constantes, variables o expresiones más complejas. No obstante cada argumento actual debe ser del mismo tipo de datos que el argumento formal correspondiente.

Ejemplos de llamada a la función **factorial**:

Para el caso de la función factorial que devuelve un valor:

- ★ printf("el factorial de %d es %ld\n",n,factorial(n));
- ★ fact=factorial(n);
- ★ a = a + factorial(n);
- ★ if (factorial(n)>50)
- ★ switch(factorial(n))

Para el caso de la función factorial que no devuelve nada:

- ★ factorial(n);

/*vemos que sucede si en la llamada no guardamos el valor de la variable retornada */

```
#include <stdio.h>
int suma(int,int);                    /* prototipo */
main()                                /* Realiza una suma */
{
int a=10,b=25,t;
t=suma(a,b);                          /* guardamos el valor */
printf("%d=%d",suma(a,b),t);
suma(a,b);                            /* el valor se pierde */
}
int suma(int a,int b)
{
return (a+b);
}
```

8.2.3 PASO DE PARÁMETROS A UNA FUNCIÓN

Existen dos formas de enviar parámetros a una función:

Por valor: se copia el valor del argumento actual o real en el argumento o parámetro formal de la función. Es decir que trabajamos con una *copia*, no con el *original*, por lo tanto cualquier cambio que se realice dentro de la función en el argumento enviado, **NO** afectará al valor original de las variables utilizadas en la llamada.

Por referencia: en este caso lo que hacemos es enviar a la función la dirección de memoria donde se encuentra la variable o dato. Es decir que trabajamos directamente con el *original*, por lo tanto cualquier modificación **SI** afectará a las variables utilizadas en la llamada. En resúmen, esto permite que datos de la parte del programa en la que se llama a la función sean accedidos por la función, alterados dentro de ella y luego devueltos al programa de forma alterada. En los casos de

pasajes por referencia en la llamada se debe anteponer al nombre de la variable el operador **&** (ampersand – la direción de) y los parámetros deben ser de un tipo puntero compatible.

Para tener en cuenta:

- ◊ Las estructuras se pueden pasar por valor o por referencia. Si se pasa una estructura por valor, las modificaciones de sus campos serán locales mientras que si se pasan por referencia las modificaciones serán permanentes.
- ◊ El paso de arrays como parámetros de funciones siempre se realiza por referencia. No es posible enviar por valor los **arrays**, ya que recordemos que el nombre de un arreglo por si solo es un puntero al primer elemento del array. Esto significa que la declaración del parámetro formal debe ser de un tipo puntero compatible, ya sea declararlo como un array (tipo nombre[long]) o como un puntero (tipo *nombre). Si fuera necesario salvar esta característica de funcionamiento, se podría transferir el arreglo *adentro* de una estructura (como un campo de ella). Así, sería la estructura el argumento en la llamada y por lo tanto, al transferirla, se generaría en memoria una copia de ella y la función podría trabajar sobre la copia del arreglo sin afectar el arreglo original.

```
/* Paso por valor. */

#include <stdio.h>

void intercambio(int,int);

main()     /* Intercambio de valores */
{
     int a=1,b=2;
     printf("a=%d y b=%d",a,b);   /* valores de a y b antes de la llamada */
     intercambio(a,b);                    /* llamada */
     printf("a=%d y b=%d",a,b);  /* valores de a y b despues de la llamada */
}

void intercambio (int x,int y)
{
      int aux;
      aux=x;
      x=y;
      y=aux;
      printf("a=%d y b=%d",x,y); /* valores de a y b en la función */
}
```

★ En el ejemplo anterior podrás comprobar que antes y después de la llamada, las variables mantienen su valor. Solamente se modifica en la función **intercambio** (*paso por valor*).

★ En el siguiente ejemplo podrás ver como las variables intercambian su valor tras la llamada de la función (*paso por referencia*).

```
/* Paso por referencia. */

#include <stdio.h>

void intercambio(int *,int *);

main()     /* Intercambio de valores */
{
     int a=1,b=2;
      printf("a=%d y b=%d",a,b);  /* valores de a y b antes de la llamada */
      intercambio(&a,&b);                 /* llamada */
      printf("a=%d y b=%d",a,b);   /* valores de a y b despues de la llamada */
}
```

```
void intercambio (int *x,int *y)
{
      int aux;
      aux=*x;
      *x=*y;
      *y=aux;
      printf("a=%d y b=%d",*x,*y);   /* valores de a y b en la función */
}
```

8.2.4 TIEMPO DE VIDA DE LOS DATOS

En C, cada función es un bloque de código. El código de una función es privado a esa función y solo se puede acceder a él mediante una llamada a esa función. (No es posible, por ejemplo, utilizar un goto para saltar en medio de otra función.) El código que comprende el cuerpo de una función está oculto al resto del programa y, a no ser que se usen datos o variables globales, no puede ser afectado por otras partes del programa ni afectarlas. Dicho de otro modo, el código y los datos que están definidos dentro de una función no pueden interactuar con el código o los datos definidos dentro de otra función, porque las dos funciones tienen un ámbito diferente.

Las variables que están definidas dentro de una función se llaman ***variables locales***. Una variable local comienza a existir cuando se entra en la función y se destruye al salir de ella. Así, las variables locales no pueden conservar sus valores entre distintas llamadas a la función. La única excepción a esta regla se da cuando la variable se declara con el *especificador* de clase de almacenamiento ***static*.** Esto hace que el compilador trate a la variable como si fuese una variable global en cuanto a almacenamiento se refiere, pero siga limitando su ámbito al interior de la función.

En C, todas las funciones están al mismo nivel de ámbito. Es decir, no se puede definir una función dentro de otra función.

A continuación veremos un ejemplo en el cual se prueba el comportamiento de una variable declarada como ***static*.** Estas variables se inicializan automáticamente en 0 (cero).

```
/* prueba variable estatica */
# include <stdio.h>
void pru_est(void);
main ( )
{
     int cont; clrscr() ;
     for(cont = 1; cont <= 3; cont++) {
     printf("iteración numero %d:", cont);
     pru_est();
    }
 }
void pru_est(void)
{
     int aut = 1;
     static int estatica = 1;
     printf("auto = %d y estatica = %d\n", aut, estatica); aut++;
     estatica++;
}
```

La salida del programa ejemplo será:

```
iteracion numero 1: auto = 1 y estatica = 1
iteracion numero 2: auto = 1 y estatica = 2
iteracion numero 3: auto = 1 y estatica = 3
```

8.2.5 RECURSIVIDAD

Si una expresión en el cuerpo de una función llama a la propia función., se dice que ésta es ***recursiva***. Un ejemplo es la función fact(), que calcula el factorial de un entero. El factorial de un número N es el producto de todos los números enteros entre 1 y N. Por ejemplo, el factorial de 3 es 1 x 2 x 3. A continuación se muestra fact() y su equivalente iterativa:

```
/* Calcula el factorial de un número. */
        /* no recursiva */
        factiter(int n)
        {
            int t, resp;
            resp = 1;
            for(t=1; t<=n; t++)
           resp=resp*(t);
           return (resp) ;
        }
```

```
/* Calcula el factorial de un número. */
/* recursiva */
        fact(int n)
        {
            int resp;
            if(n<=1) return(1);
           resp = fact(n-1)*n;
           return(resp) ;
        }
```

La versión no recursiva ***factiter( )*** debe estar clara. Utiliza un bucle que empieza en 1 y termina en el número y que progresivamente multiplica cada número por el producto móvil.

La operación de la recursiva ***fact()*** es un poco más compleja. Cuando se llama a fact() con un argumento de 1, la función devuelve 1. En otro caso, devuelve el producto de fact(n-l)*n. Para evaluar esta expresión se llama a fact() con n-l. Esto ocurre hasta que n es igual 1 y las llamadas a la función empiezan a volver.

Al calcular el factorial de 2. la primera llamada a fact() provoca una segunda llamada con argumento 1. Esta llamada devuelve 1, que se multiplica entonces por (el valor original de n). La respuesta, entonces, es 2. Puede encontrar interesante incluir sentencias printf() en fact() para ver el nivel de las llamadas y las respuestas intermedias.

Cuando una función se llama a sí misma, se asigna espacio en la pila para las nuevas variables locales y parámetros, y el código de la función se ejecuta con estas nuevas variables desde el principio. Una llamada recursiva no hace una nueva copia de la función. Sólo son nuevos los argumentos. *Al* volver de una llamada recursiva, se recuperan de la pila las variables locales y los parámetros antiguos y la ejecución se reanuda tras la llamada a la función dentro de la función. Podría decirse que las funciones recursivas tienen un efecto "telescópico" que las lleva fuera y dentro de sí mismas.

La mayoría de las rutinas recursivas no minimizan significativamente el tamaño del código ni ahorran espacio de memoria. Además, las versiones recursivas de la mayoría de las rutinas se ejecutan algo más despacio que sus versiones iterativas equivalentes, debido a las repetidas llamadas a la función; pero esto no es significativo en la mayoría de los casos. Muchas llamadas recursivas a una función pueden hacer que se desborde la pila, aunque no es probable que ocurra. Debido a que los parámetros de la función y las variables locales se guardan en la pila y cada nueva llamada crea una nueva copia de estas variables, es posible que se agote el espacio de la pila. Si pasa esto, lo que se produce es un *desbordamiento de la pila*.

La principal ventaja de las funciones recursivas es que se pueden usar para crear versiones de algoritmos más claros y más sencillos que sus equivalentes iterativas.

Cuando se escriben funciones recursivas, se debe tener una sentencia if en algún sitio que fuerce a la función a volver sin que se ejecute la llamada recursiva. Si no se hace así, la función nunca devolverá el control una vez que se le ha llamado. Es un error muy común al escribir funciones recursivas.

BIBLIOGRAFÍA

Gottfried, B. "PROGRAMACIÓN EN C". Mc Graw Hill, 1997

Hillar, G.C. "ESTRUCTURA INTERNA DE LA PC". 3ra. Edición. Editorial Hispano Americana S.A.-HASA, Argentina, 2000.

Kernighan, B. Ritchie, D. "EL LENGUAJE DE PROGRAMACIÓN C". 2da. Edición. Prentice-Hall Hispanoamericana, México, 1988.

Schildt, H. "PROGRAMACIÓN EN LENGUAJE C". Osborne/McGraw Hill, México, 1985.

Schildt, H. "THE ANNOTATED ANSI C STANDARD". Osborne/McGraw Hill. 1993.

Schildt, H. "C GUÍA DE AUTOENSEÑANZA". Osborne/McGraw Hill, España, 1995.