El Lenguaje de Programación "C"

Contents

Capítulo 1

Introducción al Lenguaje de Programación C

1.1 Introducción

C es un lenguaje de programación de propósitos generales, es decir, no esta especializado en ningún area en particular. Se caracteriza por tener un flujo de control moderno y potentes estructuras de datos. Si bien es un lenguaje de alto nivel, C tiene sentencias de bajo nivel, las cuales, solo se encuentran en lenguajes de maquina o ensambladores.

C fue diseñado originariamente para el Sistema Operativo UNIX en la PDP 11 de Digital Equipment Corporation. Tanto el sistema operativo como el compilador C y casi todos los programas que componen UNIX están desarrollados en este lenguaje.

Las razones de la popularidad del lenguaje C pueden sintetizarse en las siguientes:

- Lenguaje estructurado de alto nivel y flexible.
- Posee potentes instrucciones de bajo nivel.
- Generación de programas objeto eficientes.
- Disponibilidad de compiladores C para todos los soportes de hardware.
- Transportabilidad.
- Disponibilidad de gran cantidad de bilbiotecas.

El objetivo del curso es introducir los principales conceptos de este potente lenguaje de programación.

1.2 Identificadores

Los identificadores son nombres que permiten señalar, mencionar o hacer referencia a los diferentes objetos tratados por un programa. En particular hacen referencia a:

- Constantes simbólicas.
- Variables.
- Funciones.

Un identificador está constituido por una serie de letras y dígitos en cualquier orden excepto que el primero debe ser obligatoriamente un carácter distinto de un numero. Se pueden utilizar letras mayúsculas y minúsculas, pero estas son distintas. El carácter de subrayado '_' también se puede incluir y se puede considerar como una letra.

La mayoría de las implementaciones de C reconocen 32 caracteres para un identificador. El estándar de ANSI reconoce 32 caracteres. En C hay ciertas palabras reservadas que se denominan palabras claves que tienen un significado especial y no pueden ser usadas como identificadores definidos por el usuario (notar que son palabras en inglés y en minúsculas).

Estos palabras reservadas 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

Algunos compiladores pueden reconocer además de éstas otras palabras claves, por lo que hay que realizar una consulta al manual de referencia del mismo.

1.3 Tipos de Datos

En C existen distintos tipos de datos, cada uno de los cuales se puede encontrar representado de forma diferente en la memoria de la máquina. Los tipos de datos básicos son:

Tipo de Dato	Descripción	Memoria Bajo Dos
int	Cantidad entera	2 bytes
char	Carácter	1 byte
float	Número de coma flotante	4 bytes
double	Número de coma flotante doble precisión	8 bytes

Algunos tipos de datos básicos se pueden ampliar agregando calificadores de tipos de datos. Estos calificadores son:

- short (corto)
- long (largo)
- signed (con signo)
- unsigned (sin signo)

Por ejemplo, se puede definir enteros de la siguiente forma:

- enteros cortos (short int)
- enteros largos (long int)
- enteros con signo (signed int)
- enteros sin signo (unsigned int)

En particular, bajo el Sistema Operativo DOS, `unsigned int` ocupará la misma memoria que `int` con la diferencia que este ultimo reserva el bit más significativo para el signo y el tipo `unsigned int` no, por lo tanto un tipo de dato `int` puede almacenar valores en el rango de -32768 a 32767, en cambio `unsigned int` almacena valores entre 0 y 65535.

El calificador **unsigned** puede aplicarse a otros tipos de datos **int** ya calificados, por ejemplo,

```
unsigned short int
```

El tipo **char** se utiliza para representar caracteres individuales, puede considerarse que el tipo char es un tipo entero pequeño. Asimismo algunos compiladores permiten la calificación **long float** (equivalente a doble) o **long double** (equivalente a doble extra largo), pero esto no es en la generalidad de los casos.

1.4 Constantes

El lenguaje C tiene los siguientes tipos básicos de constantes:

- constantes enteras
- constantes de coma flotante
- constantes de caracter
- constantes de cadena de caracteres.

1.4.1 Constantes enteras

Las constantes enteras se pueden escribir en tres sistemas numéricos diferentes: decimal (base 10), octal (base 8) y hexadecimal (base 16). Una constante entera decimal puede ser cualquier combinación de dígitos tomados del conjunto de 0 a 9.

Ejemplo:

```
0               1               5547
```

Una constante entera octal está formada por cualquier combinación de dígitos tomados del conjunto de 0 a 7 pero el primer dígito debe ser cero (0).

Ejemplo:

```
01              047             0553
```

Una constante entera hexadecimal está formada por cualquier combinación de dígitos tomados del conjunto de 0 a 9 y desde la letra A hasta la F (tanto en minúsculas como en mayúsculas). Las letras A hasta la F representan las cantidades $10, 11, \cdots 15$. La constante entera hexadecimal debe comenzar con $0x$ o $0X$.

Ejemplo:

```
0x1             0x78            0xab
```

1.4.2 Constantes de coma flotante

Una constante de coma flotante es un número que contiene un punto decimal, un exponente o ambos, por ejemplo:

```
5000.           0.2             0.06E-3
```

Una constante de coma flotante con exponente se le conoce también con el nombre de notación científica, excepto que se sustituye la base 10 por la letra E o e.

Típicamente la magnitud de una constante de coma flotante puede variar entre un valor mínimo aproximado de $3.4E-38$ y un valor máximo de $3.4E+38$. El número de dígitos significativos es de al menos 6. De todos modos, el usuario debe averiguar estos valores para su instalación particular del compilador C que esté utilizando.

1.4.3 Constantes caracter

Una constante caracter es un sólo caracter encerrado entre apóstrofes. Por ejemplo,

```
'a'          '&'          'Z'
```

Las constantes caracter tienen valores enteros determinados por el valor numérico que le corresponde en el conjunto de caracteres de la computadora (código). En general el código que utilizan la mayoría de la computadoras del tipo IBM PC es el código ASCII (American Estándar Code for Information Interchange). Por lo tanto, el valor numérico de una constante caracter puede variar de una computadora a otra, sin embargo, las constantes en si son independientes del conjunto de caracteres.

1.4.4 Secuencias de escape

Ciertos caracteres especiales, no imprimibles, como por ejemplo, las comillas, el apóstrofe, el signo de interrogación o la barra invertida (backslash), pueden expresarse como constantes caracter en términos de secuencias de escape. Una secuencia de escape comienza con una seguida por uno o más caracteres especiales. Ejemplos de éstos son:

Función	Secuencia de Escape
Sonido	$\backslash a$
Form feed	$\backslash f$
Backslash	$\backslash\backslash$
Backspace	$\backslash b$
Retorno de carro	$\backslash r$
Nulo	$\backslash 0$
Tab horizontal	$\backslash t$
Tab vertical	$\backslash v$
Comillas	$\backslash$”
apóstrofe	$\backslash'$
Nueva línea	$\backslash n$
Interrogación	$\backslash ?$

En particular la secuencia de escape \0 representa un caracter nulo y se utiliza para indicar el final de una cadena de caracteres (string). También se puede expresar una secuencia de escape en términos de uno, dos o tres dígitos octales que representan patrones de bits correspondientes a un caracter. Algunos compiladores también aceptan dígitos hexadecimales. Por ejemplo,

$'\backslash 014'$ que indica el caracter form feed.

1.4.5 Constantes de cadena de caracteres

Una constante de cadena de caracteres (**string**) consta de cualquier número de caracteres consecutivos encerrados entre comillas. Si se deben incluir dentro de un string comillas, signos de interrogación o barras invertidas, deben hacerse mediante la correspondiente secuencia de escape. El compilador inserta automáticamente el caracter \0 al final de la cadena.

No se debe confundir una constante caracter con una constante de cadena de caracteres, pues no son equivalentes. De hecho, la constante caracter $'A'$ y su correspondiente constante de cadena de caracteres "A" no son equivalentes. La primera solo posee un caracter, mientras que la segunda posee dos, el código ASCII de la letra A y el caracter nulo ($\backslash 0$).

1.5 Variables y Arreglos

Una variable es un objeto que se utiliza para representar cierto tipo de información dentro de un programa. Como su nombre lo indica el valor de este objeto no es fijo sino que puede variar a lo largo de la ejecución del programa. A las variables se les da un nombre (un identificador) a través del cual hacemos referencia al valor de la misma. En algún punto del mismo se le asignará a la variable un determinado valor, el cual se puede recuperar simplemente haciendo referencia al nombre que la identifica. La información representada por una variable puede cambiar durante la ejecución de un programa, sin embargo el tipo datos que representa no.

Un arreglo es otra clase de variable la cual hace referencia a una colección de datos. Cada uno de estos datos se representa por un elemento dentro del arreglo correspondiente. Se pueden definir distintos tipos de arreglos. Por ejemplo, arreglos de caracteres, enteros, flotantes, unidimensionales, multidimensionales según veremos mas adelante.

1.6 Declaraciones

Una declaración en C asocia un tipo de datos determinado a un grupo de variables. En un programa en C se deben declarar todas las variables antes de que aparezcan sentencias ejecutables.

Una declaración consiste en un tipo de datos seguido de uno o más nombres de variables separadas por coma (**,**) y finalizando con punto y coma (**;**). Por ejemplo,

```
int a,b,c;         /* declara los enteros a,b,c */
short int z,l;     /* declara los enteros cortos z,l */
long int r,k;     /* declara los enteros largos r,k */
```

Toda variable arreglo se declara con el nombre del arreglo seguido de un par de corchetes [] con un entero encerrado entre estos que especifica el tamaño del mismo, su dimensión. Por ejemplo:

```
char j, linea[80];   /* declara el char j y el arreglo de 80 caracteres linea */
```

En el momento de la declaración se le pueden asignar valores a las variables. Para esto, la declaración consiste en un tipo de datos seguido de un nombre de variable, un signo = y una constante de tipo apropiado finalizando con **;**. Por ejemplo,

```
int c = 12;       /* declara la variable entera c y le asigna el valor 12 */
float t = 3.5e2; /* declara la variable flotante t y le asigna el valor 350 */
```

Un arreglo tipo cadena de caracteres puede inicializarse sin un tamaño implícito, de la siguiente manera,

```
char ejemplo[] = "CURSO DE C";
```

Este es un arreglo que contiene 11 caracteres donde el último es el caracter de final de cadena \0 (también denominado carácter de fin de string). Cualquier asignación posterior sólo tomará 10 elementos.

1.7 Expresiones y Sentencias

Una expresión representa una unidad de datos, tal como un número o un caracter. Una expresión consiste en una entidad simple, una referencia a una función o alguna combinación de tales entidades interconectadas por operadores. También puede representar condiciones lógicas. Ejemplo de expresiones son:

```
x = y;     /* esta expresion asigna el valor de la variable y a la variable x */
c = a + b; /* esta expresion asigna a c la suma de las variables a y b */
++i;       /* esta expresion es equivalente a i = i + 1*/
```

Una sentencia hace que la computadora ejecute alguna acción. En C existen tres tipos de sentencias: sentencias expresión, sentencias compuestas y sentencias de control.

Una sentencia expresión consiste de una expresión seguida de "**;**". La ejecución de la sentencia hace que se evalúe la expresión. Por ejemplo:

```
a = 3;
c = a + b;
++i;
printf("Area = %f",a);
```

La última sentencia hace que se evalúe la función **printf** que es una función de la biblioteca estándar del C (**stdio.h**) y permite visualizar datos en la pantalla (salida estándar).

Una sentencia compuesta está formada por varias sentencias simples encerradas entre (llaves). Ejemplo de una sentencia compuesta es:

```
{
pi = 3.141593;
long_circunf = 2. * pi * radio;
area_circunf =pi * radio * radio;
}
```

Debe observarse que una sentencia compuesta no finaliza con punto y coma después de la llave de cierre.

Las sentencias de control se utilizan para dirigir la acción de un programa, esto es, toma de decisiones, repeticiones, etc. Por ejemplo:

```
while (i < 20){
    printf("i= %d",i);
    ++i;
}
```

1.8 Constantes simbólicas

Una constante simbólica es un nombre que sustituye una secuencia de caracteres. Cuando el programa se compila, cada aparición de la constante simbólica es reemplazada por su secuencia de caracteres. Este tipo de constantes se suelen definir al inicio del programa. La forma de definir una constante simbólica es:

```
#define nombre texto
```

donde nombre representa un nombre simbólico que generalmente se lo suele definir en mayúsculas y texto representa la secuencia de caracteres asociada al nombre simbólico. Ejemplos de este tipo de constantes se muestran a continuación:

```
#define PI      3.14159
#define NUMERO  5
#define LETRAS  "alfabeto"
```

Debe observarse que las definiciones de las constantes simbólicas no finalizan con **";"**. Este tipo de constantes no son imprescindibles en C pero permiten escribir programas más claros y ordenados.

1.9 Operadores y Expresiones

Los datos sobre los que actúan los operadores se denominan operandos. Existen operadores que actúan sobre un operando (denominados unarios o manarios) y otros sobre dos operandos (denominados binarios). Describiremos cada uno de ellos.

Operadores Aritméticos

En lenguaje C existen cinco operadores aritméticos, ellos son:

Operador	Propósito
+	Suma
−	Resta
*	Producto
/	Cociente
%	Resto de división entera

Los operadores aritméticos actúan sobre operandos numéricos y por lo tanto los mismos deben ser numéricos, esto es enteros, de coma flotante o tipo carácter (este último por que los tipos **char** son casos particulares de los **int**).

Si uno de los operandos es negativo, el resultado tomará el signo dado por las reglas del álgebra común. La operación de resto de división entera no es del todo clara, pero para evaluar el signo correcto debe satisfacerse que:

```
a = (a / b) * b + (a % b)
```

Por ejemplo si,

```
a = -5 y b = 2
```

la operación resto dará como resultado -1 pues:

```
a  = (a / b) * b + (a % b)
-5 =   -2    * 2 + x
x  =  -5  + 4  = -1
```

Cuando una operación posee operadores de distintos tipos, estos pueden sufrir conversión de tipo antes de resolver la operación. En general, el resultado final se expresará con la mayor precisión posible. Si se desea, se puede convertir el valor resultante de una expresión a un tipo de datos diferente. Esto se realiza de la siguiente manera:

```
(tipo de dato) expresion
```

Este tipo de construcción se llama conversión de tipo o **cast**.

Ejemplo: Sea **i** una variable de tipo entero y **f** una de tipo flotante, la operación

```
(i+f) % 4
```

es inválida, pues la operación modulo se aplica a datos de tipo entero, en cambio **i+f** es flotante. La operación,

```
((int)(i+f)) % 4
```

es válida. El tipo de dato asociado a la expresión no sufre el cambio por un **cast** sino que es el valor de la expresión el que sufre la transformación. En C los operadores se agrupan jerárquicamente de acuerdo con su precedencia u orden de evaluación. Las operaciones con mayor precedencia se ejecutan primero. Este orden se puede alterar mediante el agregado de paréntesis en la operaciones.

Los operadores aritméticos $*$, / y % se encuentran en un grupo de precedencia y en otro de nivel inferior se encuentran los operadores + y −.

La asociatividad (orden en que se realizan las operaciones entre operadores de igual precedencia) es de izquierda a derecha.

1.9.1 Operadores monarios (unarios)

Existen ciertos operadores que actúan sobre un sólo operando. Estos se denominan monarios. El operador monario más frecuentemente utilizado es el signo − que se utiliza para definir un valor negativo. Otros operadores monarios son, el operador incremento ++ y el operador decremento −−. Los mismos se utilizan para incrementar o decrementar una variable. Por ejemplo,

```
++i;  /* es equivalente a i = i + 1 */
--j;  /* es equivalente a j = j - 1 */
```

Los operadores incremento y decremento pueden usarse de dos formas, uno precediendo a la variable lo que significa que se incrementa antes de ser utilizado, o bien después de la variable lo que implica que se incrementará luego de ser usada.

Otro operador monario es el operador *sizeof* que devuelve el tamaño del operando en bytes. Por ejemplo, si se solicita el tamaño en bytes de un entero en alguna máquina en particular, este operador devolverá el número de bytes realmente ocupados por la variable.

Los operadores monarios tienen mayor precedencia que los operadores aritméticos y su asociatividad es de derecha a izquierda.

1.9.2 Operadores relacionales y lógicos

En lenguaje C existen cuatro operadores relacionales:

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

Todos estos operadores tienen el mismo nivel de precedencia, que es menor a la de los operadores monarios y aritméticos. La asociatividad es de izquierda a derecha. Otro grupo de operadores asociados a los anteriores son los operadores de igualdad.

Operador	Significado
==	Igual que
!=	No igual que

La precedencia de estos operadores es menor que la de los relacionales y se evalúan de izquierda a derecha.

Por ejemplo:

```
(cuenta == 10)
```

si el valor de la variable cuenta es 10 el resultado será **verdadero**, sino será **falso**. Se debe tener especial cuidado en el uso del operador == pues se lo confunde con el operador de asignación =. Para evitar esta posible fuente de errores se suele utilizar la expresión anterior en la forma

```
(10 == cuenta).
```

Estos seis operadores se usan para formar expresiones lógicas que pueden resultar verdaderas o falsas. La expresión resultante es de tipo entero ya que todo valor verdadero está representado por un valor entero distinto de cero y el falso por el valor cero.

1.9.3 Operadores lógicos

Además de los operadores relacionales y de igualdad, C posee dos operadores lógicos. Ellos se muestran en la siguiente tabla.

Operador	Significado
&&	Producto lógico
||	Suma lógica

Los operadores lógicos actúan sobre operandos lógicos. La operación lógica && (Y) será verdadera si y solo si los dos operandos son verdaderos. En cambio la operación lógica || (O) será verdadera si alguno de los operandos es verdadero.

Los operadores lógicos tiene precedencia propia. La Y lógica tiene mayor precedencia que la O lógica. La precedencia de estos operadores se encuentra por debajo de la de los operadores de igualdad.

Existe otro operador lógico monario que niega el valor de una expresión lógica. Este operador es ~.

Las expresiones lógicas compuestas unidas por los operadores lógicos && y || se evalúan de izquierda a derecha pero sólo hasta que se ha establecido el valor cierto o falso del conjunto. Esto implica que puede suceder que la expresión no sea evaluada en su totalidad. Por ejemplo:

```
(error > 0.001) && (cuenta < 100)
```

Si error no es mayor que .001, toda la expresión es falsa y no se evaluará si cuenta es menor que 100.

Por otro lado, la expresión,

```
(error > 0.001) || (cuenta < 100)
```

si error es mayor 0.001, la expresión completa será cierta independientemente del valor de la variable cuenta.

1.9.4 Operadores de asignación

El operador de asignación se utiliza para asignar el valor de una expresión a una variable. El operador de asignación mas usado es el signo =. Las expresiones de asignación son de la forma:

```
variable = expresion
```

Si los dos operandos de una sentencia de asignación son de diferente tipo, el valor de la expresión de la derecha se convertirá automáticamente al tipo del identificador de la izquierda. En lenguaje C están permitidas las asignaciones múltiples de la forma:

```
variable1 = variable2 = expresion
```

lo cual es equivalente a:

```
variable1 = (variable2 = expresion)
```

Además existen cinco operadores de asignación más. Estos son:

$+=$	$-=$	$*=$	$/=$	$\%=$

Por ejemplo la expresión de asignación

```
expresion1 += expresion2
```

es equivalente a:

```
expresion1 = expresion1 + expresion2
```

La siguiente tabla resume la forma de operar de estos operadores de asignación,

Expresión Equivalente

```
i += 5          i = i + 5
f -= g          f = f - g
j *= (i - 3)    j = j * (i - 3)
f /= 3          f = f / 3
i %= (j - 2)    i = i % (j - 2)
```

Los operadores de asignación tienen menor precedencia que el resto de los operadores vistos anteriormente.

1.9.5 Operador condicional

Este operador permite realizar operaciones de toma de decisión simples. Permite reemplazar la tradicional sentencia **if-else**. Este operador tiene la forma,

```
expresion1 ? expresion2 : expresion3
```

Cuando se ejecuta el operador condicional, primero se evalúa expresión 1, si es cierta, se ejecuta expresión 2. Si expresión 1 es falsa se evalúa expresión 3. De esta forma queda fijada su precedencia. Por ejemplo, la sentencia

```
z = (a > b) ? a : b;
```

es equivalente a asignar a z el máximo valor entre a y b.

1.9.6 Precedencia y Orden de Evaluación

El siguiente cuadro resume los operadores vistos listados en orden de precedencia con la correspondiente asociatividad.

Operadores		**Asociatividad**
Monarios	$-, ++, --, sizeof()$	$D \rightarrow I$
Producto, División y Resto	$*, /, \%$	$I \rightarrow D$
Suma y Resta	$+, -$	$I \rightarrow D$
Relacionales	$<, <=, >, >=$	$I \rightarrow D$
Igualdad	$==, !=$	$I \rightarrow D$
Producto Lógico	&&	$I \rightarrow D$
Suma Lógica	$\|\|$	$I \rightarrow D$
Condicional	? :	$D \rightarrow I$
Asignación	$=, +=, -=, *=, /=, \%=$	$D \rightarrow I$

1.10 Sentencias de Control

En todo programa es necesario la toma de decisiones, la realización de repeticiones, la ejecución condicional de cierto grupo de instrucciones, etc. Ello se logra por medio de las sentencias de control que estudiaremos a continuación.

1.10.1 Sentencia while

Esta sentencia se utiliza para generar bucles o repeticiones. La estructura de esta sentencia es,

```
while (expresion) sentencia compuesta
```

La sentencia incluida se ejecutará repetidamente mientras el valor de la expresión no sea falsa (cero). Esta sentencia puede ser simple o compuesta. En la generalidad de los casos es. Por ejemplo la visualización en pantalla de los dígitos del 0 al 9 puede realizarse de la siguiente forma:

```
#include <stdio.h>
void main(void) {
    int digito = 0;
    while (digito <=9) {
        printf("%d\n",digito);
    ++digito;
    }
}
```

La primera instrucción que aparece en el programa permite incluir el archivo de cabecera stdio.h que proporciona información sobre las funciones de la biblioteca entrada salida estándar (en este caso printf). Así mismo la instrucción void main(void) debe colocarse en todo programa. Estos dos temas serán tratados con posterioridad.

En el tercer renglón se define el entero dígito y se inicializa en cero. La sentencia while ejecuta las operaciones encerradas entre llaves visualizando el dígito y posteriormente incrementándolo. Esto se ejecuta hasta tanto la variable dígito no supere el valor 9.

Este programa puede realizarse en forma más concisa como se muestra a continuación:

```
#include <stdio.h>
void main(void) {
    int digito = 0;
    while (digito <=9)
        printf("%d\n",digito++);
}
```

En este caso el lazo contiene una sentencia simple, se imprime el dígito y posteriormente se incrementa con el operador monario ++.

A continuación se muestra otro ejemplo. El mismo calcula la media de una lista de n números.

```
#include <stdio.h>
void main(void) {
    int n,cont = 1;
    float dato, media, suma = 0.;
    printf("Cuantos numeros va a promediar?");
    scanf("%d", &n);
    while (cont <= n) {
        printf("Dato=  ");
        scanf("%f",&dato);
        suma += dato;
        ++cont;
    }
    media = suma/n;
    printf("Valor medio = %f\n",media);
}
```

La función **scanf** permite leer datos introducidos a través del teclado. Esta función se estudiara mas adelante.

1.10.2 Sentencia do-while

La sentencia while vista anteriormente realiza el test de la condición antes de la ejecución de la sentencia compuesta. Hay situaciones donde se requiere que el test se realice al final de la sentencia compuesta. La sentencia do-while permite esto. Su forma es:

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

Como puede observarse sentencia se ejecutará al menos una vez. Veremos los mismos ejemplos anteriores con esta nueva sentencia.

```
#include <stdio.h>
void main(void) {
    int digito = 0;
    do {
        printf("%d\n",digito++);
    }while (digito <= 9);
}
```

```
#include <stdio.h>
void main(void) {
    int n,cont = 1;
    float dato, media, suma = 0.;
    printf("Cuantos numeros va a promediar?");
    scanf("%d", &n);
    do{
        printf("Dato= ");
        scanf("%f",&dato);
        suma += dato;
        ++cont;
    }  while (cont <= n);
    media = suma/n;
    printf("Valor medio = %f\n",media);
}
```

1.10.3 Sentencia for

Esta es una de las sentencias más utilizadas. Su forma general es:

```
for (expresion1; expresion2; expresion3) sentencia compuesta
```

Expresión 1 se utiliza para inicializar un parámetro (una variable) que se denomina índice (variable de control) que controla la repetición del bucle. Expresión 2 representa una condición que debe ser verdadera para que se continue con la ejecución del bucle. Expresión 3 se utiliza para modificar el valor de la variable de control inicialmente asignado por expresión 1. Cuando esta sentencia se ejecuta, expresión 2 se evalúa y se comprueba antes de cada pasada, mientras que expresión 3 se evalúa al final de cada pasada. La sentencia for es equivalente a:

```
expresion1;
while (expresion2) {
    sentencias
    expresion3;
}
```

Desde el punto de vista sintáctico, en la sentencia for no es necesario que estén presentes las tres expresiones, aunque es requisito indispensable los punto y coma. Las expresiones primera y tercera pueden omitirse siempre y cuando sean reemplazadas por otras sentencias. La segunda expresión si se omite se asume igual a 1 y el lazo continuará infinitamente a menos que se use algún mecanismo para interrumpir el lazo. Este último caso se verá a posteriori.

Nota: Este tipo de sentencia, se usa principalmente cuando se conoce el numero de veces que el lazo se debe repetir.

A continuación se muestran los ejemplos anteriores con la instrucción for.

```
#include <stdio.h>
void main(void) {
    int digito;
    for (digito = 0; digito <=9; ++digito)
        printf("%d\n",digito);
}
```

```
#include <stdio.h>
void main(void) {
    int n,cont;
    float dato, media, suma = 0.;
    printf("Cuantos numeros va a promediar?");
    scanf("%d", &n);
    for (cont = 0; cont < n;++cont) {
        printf("Dato= ");
        scanf("%f",&dato);
        suma += dato;
    }
    media = suma/n;
    printf("Valor medio = %f\n",media);
}
```

```
#include <stdio.h>
void main(void) {
    int digito = 0;
    for (;digito <=9;)
        printf("%d\n",digito++);
}
```

Otro ejemplo del uso de la sentencia for se muestra a través de la función **toupper** que se encarga de convertir a mayúsculas una cadena de caracteres.

```
#include <stdio.h>
void main(void) {
    char letras[80];
    int aux, cont;
    for (cont = 0;(letras[cont]=getchar())!='\r';++cont)
        ; //cuerpo del for vacio
    aux=cont;
    for (cont =0; cont <aux;++cont)
        putchar(toupper(letras[cont]));
}
```

Nota: **getchar** y **putchar** son funciones del C que permiten leer y escribir caracteres en el dispositivo estándar (en este caso el monitor).

En el renglón 5 debe observarse que existe una instrucción nula (;). Esta es necesaria ya que cuando se introduce el caracter $\backslash r$ que se materializa con la tecla $< enter >$ del teclado, la variable **cont** no se incrementa debido a que se satisface la condición, y la variable **aux** quedaría definida con un valor de cuenta que difiere en uno del real.

1.10.4 Repeticiones anidadas

Las repeticiones se pueden definir una dentro de otra (anidar), es decir se puede poner una sentencia de control dentro de otra. Es esencial que una repetición se encuentre completamente incluido dentro de otra. Además, cada repetición debe ser controlada por índices diferentes.

Por ejemplo, codigo válido

```
while(bandera == 0){
          for(i=0; i< 100;i++){
                  ............
          } //fin del for
}// fin del while
```

Inválido

```
while(bandera == 0){
          for(i=0; i< 100;i++){
                  ............
} // fin del while
          } // fin del for
```

Ejemplo: Leer el valor del lado de un cuadrado entre los valores 1 y 20 y graficar el mismo con caracteres $'*'$.

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

void main(void) {
    int lado,i,j;
    printf("Ingrese el valor del lado: ");
    scanf("%d", &lado);
    for(i=0;i < lado; i++){
        for(j=0; j < lado; j++){
            if (i == 0 || i == lado-1){
                printf("*");
            }
            else {
                if (j == 0 || j == lado-1){
```

```
                    printf("*");
                }
                else {
                    printf(" ");
                }
            }
        }
        printf("\n");
    }
    getch();
}
```

1.10.5 Sentencia if-else.

Esta sentencia se utiliza para realizar una toma de decisión a través de la evaluación de una expresión y llevar a cabo una de dos posibles acciones dependiendo de su valor verdadero o falso. Esta sentencia no repite la sentencia compuesta. La forma general de esta sentencia es:

```
if (expresion) sentencia compuesta
```

La expresión siempre se debe encontrar entre paréntesis. Sentencia sólo se ejecutara si expresión es cierta. Esta sentencia tiene una forma más general que incluye la cláusula else.

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

Si expresión tiene un valor verdadero se ejecutará sentencia 1. Si la expresión es falsa se ejecutará sentencia 2.

Nota: Esta sentencia no implica un bucle o lazo de repetición, es una sentencia de tome de decisión.

```
    if (x < 100)
        printf("X= %d",x);


    if (indicador !=0) {
        printf("Se han Producido: %d Iteraciones",indicador);
        valor = 0;
    }


    if(j != 0) {
```

```
        suma+=78.9;
        --z;
    }
    else {
        printf("Acumulado %f",suma);
        z=0;
        k+=3;
    }
```

1.10.6 Sentencia switch

Esta sentencia permite seleccionar un grupo de sentencias entre varias disponibles. La selección se basa en el valor de una expresión que se incluye en la sentencia switch. La forma general de esta sentencia es:

```
switch (expresion) sentencia compuesta
```

expresión debe evaluar necesariamente a tipo entero y puede ser de tipo caracter ya que este tipo de variables es un caso particular de los enteros.

Sentencia generalmente es una secuencia de opciones posibles a seguir. Para cada opción, la sentencia del grupo debe ser precedida por una o mas etiquetas case. Las etiquetas case identifican los diferentes grupos de sentencias y distinguen una de otras. Cada grupo de sentencias se escribe de forma general como:

```
case expresion:
    sentencia 1
    sentencia 2
        .
        .
    sentencia n
```

o en el caso de varias etiquetas case

```
case expresion1:
case expresion2:
case expresion3:
        sentencia1
case expresion4:
        sentencia 2
```

La palabra reservada **default** identifica uno de los grupos de sentencias dentro de **switch**. Esta etiqueta se utiliza si ninguna de las etiquetas case coincide con el valor de expresión.

Por ejemplo, el siguiente programa cuenta la ocurrencia de dígitos, espacios en blanco y todos los otros caracteres.

```
#include <stdio.h>
void main(void) {
    int c, i, nblancos, notros, ndigitos[10];
    nblancos = notros = 0;
    for(i = 0; i < 10; i++)
        ndigitos[i] = 0;
    while((c = getchar()) != '\r')
        switch(c) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                ndigitos[c - '0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nblancos++;
                break;
            default:
                notros++;
                break;
        }
        printf("Digitos =");
        for(i =0; i < 10; i++)
            printf(" %d", ndigitos[i]);
        printf("\nEspacios en blanco = %d, otros = %d\n",nblancos, notros);
}
```

Nota: La sentencia **break** debe ser utilizada para finalizar la ejecución de la sentencia **switch**.

1.10.7 Sentencia break

Esta sentencia se utiliza para terminar la ejecución de bucles o salir de una sentencia switch. Se puede utilizar dentro de una sentencia while, do-while, for o switch. La forma general de esta

sentencia es,

```
break;
```

Es decir,

```
switch (variable) {
    case 1:
        printf("Valor: %d",i);
        break;
    case 2:
        printf("Valor: %d",i);
        break;
}
```

```
switch (eleccion = getchar()) {
    case 'r':
    case 'R':
        printf("Rojo\n");
        break;
    case 'v':
    case 'V':
        printf("Verde\n");
        break;
    case 'a':
    case 'A':
        printf("Azul\n");
        break;
    default:
        printf("Desconocido\n");
}
```

1.10.8 Sentencia continue

La sentencia continue es utilizada para obviar el resto de las instrucciones de un lazo. Un lazo no termina cuando se encuentra una sentencia continue sino que no se ejecutan las sentencias que se encuentran a continuación de ella, saltando a la siguiente pasada a través del bucle.

```
do {
    scanf("%f",&x);
    if (x < 0) {
        printf("Valor Negativo\n");
        continue;
```

```
    }
    suma += x;
} while (x<=100);
```

En este ejemplo no se sumarán los valores negativos de x y el lazo se ejecutará hasta tanto no se lea un valor de x de 100 o mayor.

1.10.9 Operador coma (,)

El Operador **,** se utiliza principalmente en la sentencia **for**. Este permite que aparezcan dos expresiones en situaciones en donde se utilizaría sólo una expresión. La forma general es la siguiente:

```
for (expresion1a, expresion1b; expresion2; expresion3)
        sentencia compuesta
```

En este caso se inicializan dos índices dentro de la instrucción for.

1.11 Estructura de un Programa C

1.11.1 Tipos de Almacenamiento

Existen dos formas de caracterizar una variable, por su tipo de dato y por su tipo o modo de almacenamiento. El tipo de dato significa el tipo de información que la variable puede almacenar, por ejemplo un entero, un caracter, un número en punto flotante, etc.

El tipo de almacenamiento se refiere a la permanencia (vida) de la variable y a su ámbito o alcance dentro del programa (lugar donde se puede utilizar la variable) en el cual se reconoce, en otras palabras el ámbito de una variable esta formado por él o los programas desde los cuales se puede acceder a ella.

Existen muchas especificaciones de tipo de almacenamiento, algunas de ellas son:

- Automática (auto)
- Externa (extern)
- Estática (static)

Variable Automáticas Las variables automáticas se declaran siempre dentro de la función y son locales (o internas) a ella. Las variables automáticas declaradas en funciones diferentes, serán independientes entre sí, aún cuando tengan el mismo nombre.

Toda declaración (incluido los parámetros formales) que se encuentre dentro de una función se interpretará como automática, salvo que explícitamente se especifique otro tipo de almacenamiento.

Las variables automáticas deben ser inicializada al comenzar la función donde ellas están declaradas. Si no existe una inicialización explícita en la declaración la variable igualmente toma un valor impredecible y probablemente sin significado para n ustros propósitos.

Una variable automática pierde su valor y se destruye (se muere) cuando se sale de la función donde ha sido declarada.

Variables Externas Estas variables, a diferencia de las automáticas, no están destinadas a funciones simples. Su alcance se extiende desde el punto de definición hasta el resto del programa. Estas variables son por naturaleza de tipo global, por lo tanto se puede acceder a ellas desde cualquier función que esté dentro de su ámbito de definición.

Además, se puede acceder y modificar el valor de una variable externa en cualquier función que esté en el ámbito de alcance. Debido a esto a veces se usan este tipo de variables para transferir información entre funciones sin utilizar pasajes de parámetros. Esto es particularmente útil para transferir datos múltiples, por ejemplo cuando una función requiere numerosos datos de entrada.

Es necesario introducir ahora los conceptos de definición y declaración de variables externas. Para que una función pueda utilizar una variable externa son necesarios dos pasos: el primero es que la variable haya sido definida previamente y el segundo es que la variable haya sido declarada en la función que la va a utilizar.

La definición de una variable externa se escribe de igual manera que una variable común. Una variable externa se debe definir antes que las funciones que van a utilizarla. El Compilador C, al encontrar su definición reserva espacio de memoria para su almacenamiento. Si es necesario se puede inicializar la variable en esta definición. No se requiere agregar el prefijo extern en la definición pues una variable se reconoce como externa por la ubicación de la definición en el programa. Por ejemplo, supongamos el siguiente programa que lee varias líneas de texto y determina el numero medio de caracteres por línea.

```
#include <stdio.h>

/* Variables externas */
int sum = 0;            /* numero total de caracteres */
int lineas = 0;         /* numero de lineas */
int contlinea(void);    /* declaracion de la funcion */

/* Programa Principal */
void main (void) {
    int n;              /*numero de caracteres en una linea */
    float media;        /* media de caracteres por linea */
    printf ("Introducir el texto debajo:\n");

    /* leer una linea de texto y actualizar los contadores */
    while(( n = contlinea()) > 0) {
        sum += n;
        ++lineas;
```

```
    }
    media = (float) sum/lineas;
    printf("\nMedia de caracteres por linea: %5.2f",media);
}

int contlinea(void) {
/* lee una linea de texto y cuenta los caracteres */
    char linea[80];
    int cont = 0;
    while((linea[cont] = getchar()) != '\r')
        ++cont;
    return(cont);
}
```

Una declaración de una variable externa debe comenzar con el prefijo extern. El nombre de la variable externa y su tipo deben coincidir con su correspondiente definición.

La aparición de una declaración de variable externa no reserva espacio de almacenamiento, pues este ya fue reservado en la definición. Tampoco se puede inicializar una variable en una declaración.

Es importante tener presente que una modificación en una variable externa repercutirá en otras funciones y a veces estos efectos colaterales traen problemas difíciles de detectar.

Variable Estáticas Las variables estáticas pueden ser internas (automáticas) o externas. Las variables estáticas internas son locales a una función particular de la misma manera que las automáticas, pero a diferencia de estas últimas, no pierden su valor y lo mantienen durante todo el programa, pero solo pueden ser usadas en la función donde fueron definidas. Esto significa que las variables estáticas internas proveen a una función con variables privadas y permanentes.

Una variable estática externa es conocida dentro del resto del archivo fuente donde fue declarada, pero no en otro archivo. De esta manera se provee un medio de ocultar nombres de variables.

Para definir variables de este tipo se debe utilizar la palabra static, por ejemplo,

```
static int dato = 0;
```

1.11.2 Estructura General de un Programa en C

Un programa en C posee un conjunto de bloques que se pueden estructurar de la siguiente manera:

```
* Inclusion de bibliotecas (archivos .h)
* Declaracion de funciones globales
* Declaracion de variables globales
* Declaracion de constantes simbolicas
```

```
void main(void) {
    * Declaracion de Variables locales
    * Declaracion de Funciones locales

    ................................
    * Cuerpo ejecutable del programa en C
    ................................

} // fin del programa
```

Capítulo 2

Funciones

2.1 Introducción

El uso de funciones definidas por el programador permite dividir un programa grande en un cierto número de componentes más pequeñas, cada una de las cuales posee un propósito único e identificable. Por lo tanto, un programa en C se puede modularizar mediante el uso inteligente de las funciones.

Esto es útil para la depuración del programa, evitar redundancia de código, generación de bibliotecas propias para posteriormente usarlas en otro programa, etc.

Igualmente importante es la claridad lógica resultante de la descomposición de un programa en varias funciones concisas, representando cada función alguna parte bien definida del problema global. Estos programas son más fáciles de escribir y depurar, y su estructura lógica es más fácil de entender que la de aquellos que adolecen de este tipo de estructuras. La mayoría de los programas en C se modularizan de esta manera, aun cuando no impliquen la ejecución repetida de las mismas tareas. De hecho, la descomposición de un programa en módulos individuales se considera generalmente parte importante de la buena práctica de la programación.

Existen dos tipos de funciones para el Lenguaje de programación C:

- Las funciones definidas por el usuario (nuestras funciones)
- Las funciones de bibliotecas (definidas por otros usuarios y que podemos utilizar)

Una función es un segmento de programa que realiza determinadas tareas bien definidas. 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 las instrucciones contenidas en main.

El uso de funciones dentro de un programa en C consta de tres partes perfectamente definidas:

1. La definición de la función: Esta parte contiene el código ejecutable de la misma. Aquí es donde se define la tarea que realizará la función.

2. La declaración de la función: con esta sentancia le indicamos al compilador que existe una función definida por el usuario. Normalmente a la decalración de una función se le denomina "prototipo de función".

3. La invocación o llamada de la fucnión: es cuando queremos hacer uso de la función, es la setencia que debemos usar para que el procesador ejecute la fucnión.

Si un programa contiene varias funciones, sus definiciones deben aparecer en cualquier orden, pero deben ser independientes unas de otras. Esto es, una definición de una función no puede estar incluida en otra. Cuando se accede a una función desde alguna determinada parte del programa (o sea, cuando se invoca la función), se ejecutan todas las instrucciones de que consta. Se puede acceder a una función desde distintos lugares de un programa.

Una vez que se ha completado la ejecución de una función, se devuelve el control al punto desde el que se accedió a ella. Generalmente, una función procesará la información que le es pasada desde el punto del programa en donde se accedío a ella y devolverá un solo valor.

La información se le pasa a la función mediante identificadores especiales denominados argumentos formales (también llamados parámetros) y es devuelta por la sentencia return. Sin embargo, algunas funciones aceptan información pero no devuelven nada (void), mientras que otras devuelven un valor a través de la sentencia return.

2.1.1 Definición de una Función

La definición de una función tiene los siguientes componentes principales: la primera línea, la declaración de variables locales y el cuerpo de la función. En términos generales, la definición de una función se puede escribir de la siguiente manera:

```
tipo_devuelto nombre_funcion(tipo y lista de argumentos formales) {
    definiciones de variables locales;
    sentencias;
}
```

La primera línea de la definición contiene la especificación del tipo de valor devuelto por la función, seguido del nombre de la misma y (opcionalmente) un conjunto de argumentos (tipo y nombre), denominados argumentos formales, separados por comas y encerrados entre paréntesis. La especificación del tipo de dato devuelto por la función se puede omitir si la misma devuelve un entero o un caracter. Deben seguir al nombre de la función un par de paréntesis vacíos si la definición de la función no incluye ningún argumento. Los argumentos formales permiten que se transfiera información desde el punto del programa en donde se llama a la función a ésta. Los argumentos correspondientes en la invocación de la función se denominan argumentos actuales o reales (llamada de la función). Las variables utilizadas como argumentos formales son locales en el sentido de que no son reconocidos fuera de ella.

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.

El resto de la definición de la función (el cuerpo) esta constituida por la definición de las variables locales a la función y una sentencia compuesta que define las acciones que debe realizar.

Se devuelve información desde la función hasta el punto del programa desde donde se llamó mediante la sentencia **return**.

La sentencia **return** se puede escribir de la siguiente forma:

```
return(expresion);
```

o simplemente,

```
return expresion;
```

No es obligatorio que la sentencia return lleve una expresión asociada. Tampoco es necesario que aparezca esta sentencia en el cuerpo de la función, en este caso la función no retorna valor.

return solo puede tener como argumento un único valor (dato simple). El valor de retorno debe ser del mismo tipo que el especificado en la definición de la función. Si no es del mismo tipo el compilador C intentará aplicar alguna regla de conversión de tipo, si esto fracasa dará un mensaje de error.

El siguiente ejemplo muestra la forma en que se declara una función que devuelve un valor long. Se debe calcular el factorial de un entero positivo dado n, esta tarea es realizada por una función llamada factorial a la cual se le debe entregar como argumento el numero del cual se desea calcular el factorial.

```
#include <stdio.h>
/* declaracion de la funcion */
long int factorial (int);

void main(void) {
    int n;          /* numero n */
    long int fact;  /* factorial */

    printf("Introducir el numero: ");
    scanf("%d",&n);

    /*invocacion de la funcion con los parametros actuales */
    fact = factorial(n);

    printf("\nSu factorial es: %ld\n", fact);
```

```
}

long int factorial (int n) { /* primera linea de la definicion */
    /* declaracion de variables locales*/
    int i;
    long int prod=1L;

    /*Cuerpo de la funcion*/
    if(n < 0) {
        return (-1L);
    }
    else {
        if (n <= 1) {
            return (1L);
        }
        else {
            for (i = 2, i <= n , ++i) {
                prod *= i;
            }
        }
    }
    return(prod);
}
```

Si el tipo de datos especificado en la primera línea de la definición de la función no coincide con la expresión de retorno, el compilador intentará convertir la cantidad representada por la expresión al tipo de datos especificado en la primera línea. El resultado de esto puede ser un error de compilación o una pérdida parcial de datos (por ejemplo, debido al truncamiento). En cualquier caso se deben evitar incongruencias de este tipo.

La mayoría de los compiladores C permiten que aparezca la palabra clave void como especificador de tipo de datos cuando se define una función que no devuelve ningún valor.

El siguiente ejemplo calcula el máximo de dos cantidades enteras:

```
void maximo(int x , int y) {
    /* Definicion de una funcion que no retorna valor */
    int z;
    z = ( x >= y) ? x : y;
    printf("\n\nValor maximo = %d",z);
}
```

otra forma sería,

```
void maximo(int x , int y) {
```

```
    /* Definicion de una funcion que no retorna valor */
    int z;
    if( x >= y)
        z =  x;
    else
        z =  y;
    printf("\n\nValor maximo = %d",z);
}
```

El lenguaje C no permite definir una función dentro de otra. Si se invoca una función desde una parte del programa anterior a la definición de tal función, debe entonces aparecer en el módulo invocante la declaración de la función invocada (declaración por adelanto o declaración forward), esto informará al compilador que la función será llamada antes de que sea definida.

Como ejemplo consideremos la función que convierte caracteres de minúscula a mayúscula.

```
# include <stdio.h>
void main(void) {
    char minusc, mayusc;
    /* declaracion de la funcion */
    char minusc_a_mayusc(char minusc);
    printf("Introducir una letra minuscula: ");
    scanf("%c", &minusc);
    /* llamada a la funcion, con parametros actuales */
    mayusc = minusc_a_mayusc(minusc);
    printf("\nLa mayuscula equivalente es %c\n\n", mayusc);
}

/* definicion de la funcion */
char minusc_a_mayusc(char c1)   {
    char c2;
    c2 = ((c1 >= 'a') && (c1 <= 'z')) ? ('A'+ c1 - 'a') : c1;
    return(c2);
}
```

2.2 Paso de Argumentos a una Función

Cuando a una función se le pasa un valor mediante un argumento actual, se copia el valor de este en el argumento formal correspondiente de la función. Esto significa que se puede modificar el valor del argumento formal dentro de la función, pero el valor del argumento actual en el punto de la llamada no cambiará. Este procedimiento para pasar el valor de un argumento a una función se denomina paso por valor. Veamos un ejemplo sencillo para ejemplificar esto:

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

```
void modificar (int);
void main(void) {
    int a = 2;
    printf("\nEl valor de a antes del llamado es %d\n",a);
    modificar(a);
    printf("\nEl valor de a despues del llamado es %d\n",a);
}

void modificar(int a) {
    a *= 3;
    printf("\n\tEl valor de a dentro de la funcion es %d\n", a);
}
```

Al ejecutar este programa se genera la siguiente salida:

El valor de a antes del llamado es 2

El valor de a dentro de la función es 6

El valor de a después del llamado es 2

En conclusión en lenguaje C todos los argumentos se pasan por "por valor". De esta manera en C la función llamada no puede alterar el valor de una variable de la función llamadora, únicamente puede cambiar el valor de la copia privada de esa variable. Como veremos posteriormente modificar el valor de un argumento actual dentro de una función solo es posible en casos especiales.

2.3 Especificación del Tipo de Datos de los Argumentos

El compilador estándar ANSI permite el siguiente formato de especificación de argumentos en definiciones y declaraciones de funciones:

```
tipo nombre (tipo1 arg1, tipo2 arg2, ..., tipon argn)
```

en donde *arg1, arg2, ..., argn* hacen referencia al primer argumento, al segundo, y así sucesivamente. Las declaraciones de funciones escritas de esta forma se llaman prototipos de funciones. Por ejemplo,

```
int mostrar( int, int);
float func(int, float);
void f(char, long, double);
```

El siguiente ejemplo aclara los conceptos:

```
int func(int ,int );  /* declaracion de funcion */
```

```
void main(void) {
    int a,b,c,d,i,j;    /* declaracion de variables */
    . . .
    i=func(a,b);
    . . .
    j=func(c,d);
}

int func(int a,int b) { /* definicion de funcion */
    /* cuerpo de la funcion */
    . . .
}
```

2.4 Funciones en programas de varios archivos

Hasta ahora hemos restringido nuestra atención a programas en C que están enteramente contenidos en archivos simples. Sin embargo, muchos programas están compuestos por varios archivos. Esto es especialmente cierto en programas que utilizan funciones largas, donde cada función puede ocupar un archivo separado.

Si existen muchas funciones pequeñas separadas relacionadas entre sí dentro de un programa, puede ser deseable colocar unas pocas funciones dentro de cada uno de los diversos archivos. Entonces los archivos individuales se pueden compilar por separado y posteriormente unir para formar un programa objeto ejecutable. Esto hace mas fácil la depuración y la edición del programa, pues cada archivo se mantiene con un tamaño manejable.

Los programas de varios archivos permiten una mayor flexibilidad al programador al definir el ámbito de las funciones y de las variables. Sin embargo, las reglas asociadas con los tipos de almacenamiento se vuelven más complicadas, por que se aplican a funciones así como a variables, y existen más opciones disponibles en el uso tanto de variables externas como estáticas.

La definición de una función dentro de un programa de varios archivos puede ser tanto estática (static) como externa (**extern**).

Una función externa será reconocida a lo largo de todo el programa, una función estática será reconocida solo dentro del archivo que se defina. En cada caso, el tipo de almacenamiento se establece colocando el tipo adecuado de asignación (externa o estática) al comienzo de la definición de la función. Se asume que la función es externa si la designación del tipo de almacenamiento no aparece.

En términos generales, la primera línea de una definición de función se puede escribir como:

```
<tipo-almacenamiento> tipo-dato nombre( tipo1 arg1,..., tipon argn)
```

donde, **argi** se refiere al argumento formal **i-esimo**.

2.5 Algunas funciones de la biblioteca estándar

A lo largo de los diversos ejemplos que se han utilizado se han visto algunas funciones predefinidas del lenguaje C, en particular la función de salida **printf**. Trataremos ahora específicamente algunas de las funciones de entrada/salida más usadas. En particular se estudiarán las funciones **getch**, **putch** que se encuentran en la biblioteca *conio.h*, **scanf**, **printf**, **gets** y **puts**. La función **getch** permite ingresar un caracter a través del teclado y la función **putch** mostrar (exteriorizar) un caracter en la pantalla. Por otra parte las funciones **scanf** y **printf** son más complicadas y permiten el ingreso y exteriorización de caracteres, valores numéricos y cadena de caracteres; **gets** y **puts** permiten la entrada y salida de cadenas de caracteres (strings).

La mayoría de las versiones de C incluyen una colección de archivos cabecera que proporcionan la información necesaria para las distintas funciones de biblioteca. Estos archivos se incluyen en un programa mediante la sentencia **#include** al comienzo del programa. Por ejemplo,

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

2.5.1 Entrada de un caracter - Función getch y getche

Esta función permite el ingreso de un caracter a través del teclado. Esta función es parte de la biblioteca de consola estándar del C. La misma devuelve el caracter leído desde el dispositivo estándar de entrada (normalmente el teclado). No requiere argumento y su forma general es:

```
variable_caracter = getch();
```

donde *variable_caracter* es alguna variable previamente definida. La función **getche** trabaja de igual forma que **getch** pero el caracter ingresado (tecleado) por el usuario es mostrado en la pantalla (eco).

2.5.2 Salida de un caracter - Función putch

Esta función es el complemento de la función **getch** y permite visualizar un caracter en el monitor. Al igual que la función anterior es parte de la biblioteca de consola estándar del C. Su formato es:

```
putch(variable_caracter);
```

donde *variable_caracter* hace referencia a una variable tipo caracter previamente definida cuyo contenido se quiere exteriorizar.

A continuación se muestra un programa que hace uso de estas dos funciones. El mismo lee caracteres desde el teclado hasta que encuentra un caracter de retorno de carro ($\backslash r$). Todos los caracteres leídos los va almacenando en un arreglo denominado letras. Posteriormente se imprimen los caracteres leídos pero convertidos a mayúsculas (utiliza la función **toupper**, biblioteca **ctype.h**).

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

void main(void) {
    char letras[80];
    int aux, cont;
    for (cont = 0;(letras[cont]=getch())!='\r';++cont)
        ; /* cuerpo del for vacio */
    aux=cont;
    for (cont =0; cont < aux;++cont)
        putch(toupper(letras[cont]));
}
```

2.5.3 Función scanf

Esta función también llamada función de entrada con formato permite introducir cualquier combinación de valores numéricos, caracteres y cadena de caracteres. Esta función devuelve el numero de datos que se han conseguido introducir correctamente. Se encuentra junto con **printf** en la biblioteca cuya cabecera es **stdio.h**.

Su forma general se puede escribir:

```
int scanf(string de formato,arg1,arg2,..,argn);
```

en donde string de formato hace referencia a una cadena de caracteres que contiene información sobre el formato de los datos y *arg1,arg2,...,argn* son argumentos que representan los datos (En realidad los argumentos representan punteros que indican direcciones de memoria en donde se encuentran los datos).

En el string de formato se incluyen grupos de caracteres, un grupo por cada dato de entrada. Cada grupo de caracteres debe comenzar con el signo %. En su forma mas sencilla , un grupo de caracteres estará formado por el signo % seguido de un caracter de conversión que indica el tipo del dato correspondiente. Dentro de la del string de formato no se permite la inclusión de mensajes, tales como carteles de aclaración de que tipo de información requiere el programa.

Dentro de la cadena de control se pueden encontrar varios caracteres seguidos o pueden estar separados por caracteres de espaciado (espacios en blanco, tabuladores, etc).

En la siguiente tabla se muestran los caracteres de conversión de uso más frecuente. Los argumentos pueden ser variables o arreglos y sus tipos deben coincidir con los indicados por los grupos de caracteres correspondientes en el string de formato. Cada nombre de variable debe ser precedido por el caracter & ya que los argumentos son en realidad punteros que indican donde están situadas las variables en memoria, a no ser que la variable sea un puntero.

Caracter	Significado
c	El dato ha ingresar es un caracter.
d	El dato ha ingresar es un entero decimal con signo.
e	El dato ha ingresar es un valor en coma flotante con exponente.
f	El dato ha ingresar es un valor en coma flotante sin exponente.
g	El dato ha ingresar es un valor en coma flotante.
i	El dato ha ingresar es un entero con signo.
o	El dato ha ingresar es un entero octal, sin el cero inicial.
s	El dato ha ingresar es una cadena de caracteres.
u	El dato ha ingresar es un entero decimal sin signo.
x	El dato ha ingresar es un entero hexadecimal sin el prefijo 0x.
ld	El dato ha ingresar es un entero largo con signo.
lu	El dato ha ingresar es un entero largo sin signo.

Ejemplos,

```
#include <stdio.h>
void main(void) {
    char letras[20];
    int entero;
    float flotante;
        . . . .
        . . . .
    scanf("%s %d %f",letras,&entero,&flotante);
        . . . .
        . . . .
}
```

El string de formato es en este caso "$\%s$ $\%d$ $\%f$". Contiene tres grupos de caracteres. El primer grupo de caracteres, $\%s$ indica que el primer argumento (letras) representa una cadena de caracteres y como se verá más adelante el nombre de un arreglo es un puntero y por lo tanto no se debe colocar el símbolo &. Además cuando se lee una cadena de caracteres como en el ejemplo se debe tener en cuenta que no se permiten espacios en blanco. El segundo grupo, $\%d$, indica que el segundo argumento (&entero) representa un valor entero decimal, y el tercer grupo, $\%f$, indica que el tercer argumento (&flotante) representa un valor de coma flotante.

Otro ejemplo,

```
#include <stdio.h>
void main(void) {
    char letras[80];
    scanf("%[ ABCDEFGHIJKLMNOPQRSTUWXYZ]",letras);
    ......
    ......
}
```

En este caso se leerá un arreglo de caracteres, pero únicamente se permiten caracteres blancos y letras del abecedario en mayúsculas.

También es posible limitar el número de caracteres especificando una longitud de campo. El dato puede estar compuesto por menos caracteres que los que especifique la longitud de campo, pero no se leerán los que se encuentran mas allá de este valor. Estos caracteres sobrantes pueden ser interpretados de forma incorrecta como los componentes del siguiente dato. Ejemplo

```
#include <stdio.h>
void main(void) {
    char letras[20];
    int entero;
    float flotante;
    scanf("%s %3d %5f",letras,&entero,&flotante);
    .....
    .....
}
```

2.5.4 Función printf

Como ya se ha visto esta función permite escribir datos en el dispositivo estándar de salida, es análoga en su uso a la función **scanf**. En términos generales, la función se escribe como:

```
printf(string de formato, arg1,arg2,...,argn);
```

en donde string de formato hace referencia a una cadena de caracteres que contiene información sobre el formato de la salida y *arg1, arg2,..., argn* son argumentos que representan los datos de salida. Los argumentos pueden ser constantes, variables simples o nombres de arreglos, o expresiones mas complicadas. En contraste con la función **scanf**, los argumentos de **printf** no representan direcciones de memoria y por lo tanto no deben ser precedidos por el símbolo &. Básicamente el string de formato tiene una estructura igual al string de formato de la función **scanf**. La lista de caracteres de conversión para esta función se muestra en la siguiente tabla. Recordar que cada uno de estos caracteres debe estar precedido del signo %.

En esta función puede especificarse el número de decimales a imprimir. Un número de coma flotante se redondeará si se debe recortar para ajustarse a la precisión especificada. Además de la longitud, la precisión y el caracter de conversión, cada grupo puede incluir un indicador que afecta la forma en que aparecerán los datos en la salida. El indicador debe colocarse inmediatamente a continuación del signo %. Algunos compiladores permiten que aparezcan dos o más indicadores seguidos dentro del mismo grupo de caracteres. En la siguiente tabla aparecen los indicadores de uso más común.

Caracter	Significado
c	El dato es visualizado como un caracter.
d	El dato es visualizado como un entero decimal con signo.
e	El dato es visualizado como un valor en coma flotante con exponente.
f	El dato es visualizado como un valor en coma flotante sin exponente.
g	El dato es visualizado como un valor en coma flotante.
i	El dato es visualizado como un entero con signo.
o	El dato es visualizado como un entero octal, sin el cero inicial.
s	El dato es visualizado como una cadena de caracteres.
u	El dato es visualizado como un entero decimal sin signo.
x	El dato es visualizado como un entero hexadecimal sin el prefijo 0x.
ld	El dato es visualizado como un entero largo con signo.
lu	El dato es visualizado como un entero largo sin signo.

Indicador	Significado
+	Cada dato numérico es precedido por un signo + o − según corresponda.
−	El dato se ajusta a la izquierda dentro del campo.
0	En lugar de espacios vacíos se muestran ceros.
#	Hace que los datos Octales y Hexadecimales sean precedidos por 0 o 0x.

El siguiente programa permite visualizar un numero en coma flotante con distintas precisiones.

```
#include <stdio.h>
void main(void) {
    float x = 123.456;
    printf("%7f %7.3f %7.1f\n\n", x, x, x);
    printf("%12e %12.5e %12.3e", x, x, x);
}
```

Cuando se ejecuta este programa, se genera la siguiente salida:

```
123.456000      123.456         123.5

1.234560e+002   1.23456e+002    1.235e+002
```

2.5.5 Funciones gets y puts

Estas funciones aceptan un solo argumento. El mismo debe ser un dato que represente una cadena de caracteres. La misma puede contener espacios. Estas funciones son alternativas sencillas respecto al uso de **scanf** y **printf** respectivamente.

```
#include <stdio.h>
void main(void) {
    char linea[80];
```

```
    gets(linea);
    puts(linea);
}
```

2.5.6 Funciones rand, randomize y srand

Estas funciones pertenecen a la biblioteca estándar **stdlib.h**. La función **rand** genera numeros enteros pseudo aleatorios en el rango 0 hasta 65535. El formato de esta función es,

```
int rand(void);
```

Cada vez que esta función es ejecutada genera un numero en el rango antes citado. Estas funciones se basan en generar secuencias a partir de un numero inicial llamado semilla. La función **rand** siempre comienza a partir de la misma semilla por lo que siempre genera la misma secuencia. Existen dos funciones que modifican esta semilla y permiten que la función **rand** genere secuencias diferentes. La función **randomize**, cuyo formato es,

```
void randomize(void);
```

toma la hora de la computadora como valor inicial para modificar el valor de la semilla. Por lo tanto debe utilizarse la biblioteca **time.h** junto a **randomize**. Por su parte, la función **srand** con formato,

```
void srand(unsigned semilla);
```

también modifica el valor de la semilla, pero en este caso se le entrega a la función un valor **unsigned int** como argumento, a partir de este valor se genera la secuencia. Si este valor permanece fijo, también lo hará la secuencia generada.

Para ejemplificar el uso de estas funciones analizemos los siguientes programas:

Ejemplo: Programa que simula la tirada de una moneda un cantidad de veces especificada por el operador. El resultado es la cantidad y el porcentaje de Caras y Secas.

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

void main(void) {
    int moneda;
    long int i, ContC=0L, ContS=0L, ContT; /* Contador de caras y Secas */
    float porC, porS; /* Porcentaje de Caras y Secas */
```

```
    clrscr(); /* Borra la pantalla */
    printf("Ingrese la Cantidad de veces que desea arrojar la moneda: ");
    scanf("%ld", &ContT);
    randomize(); /* Inicializa el generador de numeros aleatorios */
    for(i = 0;i < ContT;i++) {
        moneda = rand()%2; /* esta expresion genera numeros 0 o 1 */
        (0 == moneda)? ContC++ : ContS++;
    }
    printf("Cantidad de Caras y Secas para %ld Tiradas\n\n", ContT);
    printf("Caras       :%ld\t\tSecas        :%ld\n",ContC,ContS);
    porC = ((float)ContC)/ContT;
    porS = ((float)ContS)/ContT;
    printf("Por. Caras  :%ld\t\tPor. Secas   :%ld\n",porC,porS);
    getch();
}
```

Ejemplo: Programa que simula la tirada de un dado una cantidad de veces especificada por el operador. El resultado es la cantidad de veces que ocurre cada y su porcentaje.

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

void main(void) {
    int dado;
    long int i, Cont1, Cont2, Cont3, Cont4, Cont5, Cont6, ContT;
    /* Contadores de caras*/
    Cont1 = Cont2 = Cont3 = Cont4 = Cont5 = Cont6 = 0L;

    clrscr(); /* Borra la pantalla */
    printf("Ingrese la Cantidad de veces que desea arrojar el dado: ");
    scanf("%ld", &ContT);
    randomize(); /* Inicializa el generador de numeros aleatorios */
    for(i = 0;i < ContT;i++) {
        dado = rand()%6 + 1; /* esta expresion genera numeros de 1 a 6 */
        switch(dado){
            case 1:
                    Cont1++;
                    break;
            case 2:
                    Cont2++;
                    break;
            case 3:
```

```
                    Cont3++;
                    break;
            case 4:
                    Cont4++;
                    break;
            case 5:
                    Cont5++;
                    break;
            case 6:
                    Cont6++;
                    break;
        }
    }
    printf("Cantidad de tiradas del dado: %ld\n\n", ContT);
    printf("Cara 1:%ld veces\t\tPorcentaje :%f\n",Cont1,((float)Cont1)/ContT);
    printf("Cara 2:%ld veces\t\tPorcentaje :%f\n",Cont2,((float)Cont2)/ContT);
    printf("Cara 3:%ld veces\t\tPorcentaje :%f\n",Cont3,((float)Cont3)/ContT);
    printf("Cara 4:%ld veces\t\tPorcentaje :%f\n",Cont4,((float)Cont4)/ContT);
    printf("Cara 5:%ld veces\t\tPorcentaje :%f\n",Cont5,((float)Cont5)/ContT);
    printf("Cara 6:%ld veces\t\tPorcentaje :%f\n",Cont6,((float)Cont6)/ContT);
    getch();
}
```

Capítulo 3

Arreglos

3.1 Introducción

Muchas aplicaciones requieren el procesamiento de datos múltiples que tienen características comunes, por ejemplo un conjunto de datos numéricos, representados por $x_1, x2, \cdots, x_n$. En tales situaciones es a menudo conveniente colocar los datos en un arreglo, donde todos comparten el mismo nombre,x para el ejemplo. Los datos individuales pueden ser caracteres, enteros, números en coma flotante, etcétera. Sin embargo, todos deben ser del mismo tipo y con el mismo tipo de almacenamiento.

Cada elemento del arreglo (cada dato individual) es individualizado especificando el nombre del arreglo seguido por uno o mas índices, con cada índice encerrado entre corchetes. Cada índice debe ser expresado como un entero no negativo que varia desde 0 a $n-1$. Así en un arreglo x de n elementos, cada uno de los elementos son $x[0], x[1], x[2], ...x[n]$, como se ilustra en el diagrama,

$x[0]$	$x[1]$	$x[2]$	$\cdots$	$x[n-2]$	$x[n-1]$

El numero de índices determinan la dimensión del arreglo. Por ejemplo, $x[1]$ refiere a un elemento de un arreglo unidimensional cuyo nombre es x. Similarmente, $y[i][j]$ se refiere a un elemento genérico en un arreglo bidimensional y. Se pueden formar arreglos de mayor dimensión, añadiendo índices adicionales de la misma manera.

3.2 Definición de un Arreglo

Los arreglos se definen en gran parte como las variables ordinarias, excepto que en cada arreglo se debe indicar la dimensión del mismo, es decir el número de elementos. Para un arreglo unidimensional, el tamaño se especifica con una expresión entera positiva encerrada entre corchetes. La expresión es generalmente una constante entera positiva. En términos generales, un arreglo unidimensional puede expresarse como:

```
<tipo_almacenamiento> tipo_dato nombre_arreglo[expresion];
```

donde **tipo_almacenamiento** se refiere al tipo de almacenamiento del arreglo, **tipo_dato** es el tipo de datos del arreglo, **nombre_arreglo** es el nombre del arreglo y expresión es una expresión entera positiva que indica el numero de elementos del arreglo (su dimensión). Como con variables básicas el tipo_almacenamiento es opcional; los valores por defecto son automático para los arreglos definidos dentro de una función o bloque, y externo para arreglos definidos fuera de una función.

Algunos ejemplos de definición de arreglos son:

```
int x[100];                 /* Arreglo de 100 elementos enteros */
char texto[80];             /* Arreglo de 80 caracteres */
static char mensaje[25];    /* Arreglo estatico de 25 caracteres */
static float n[12];         /* Arreglo estatico de 12 numeros reales */
```

Algunas veces es conveniente definir el tamaño de un arreglo en términos de una constante simbólica en vez de una cantidad entera fija. Esto hace mas fácil modificar un programa que utiliza arreglos, ya que todas las referencias al tamaño máximo del arreglo puede alterarse cambiando simplemente el valor de la constante simbólica.

Ejemplo: Convertir texto en minúsculas a mayúsculas.

```
#include <stdio.h>
#define TAMANO  80
void main(void) {
    char letra[TAMANO];
    int cont;
    for(cont=0; cont < TAMANO; cont++) {
        letra[cont] = getche();
    }
    for(cont=0; cont < TAMANO; ++cont) {
        putch(toupper(letra[cont]));
    }
}
```

Notar que la constante simbólica **TAMANO** tiene asignado el valor 80. En la declaración del arreglo y en las dos sentencias for aparece esta constante en lugar de su valor. Por tanto para alterar el programa acomodándolo a distintos tamaños del arreglo, sólo debe cambiarse la sentencia #define.

Los arreglos pueden incluir, si se desea, la asignación de valores iniciales. Los valores iniciales deben aparecer en el ordenen que serán asignados a los elementos individuales del arreglo, encerrados entre llaves y separados por comas. La forma general es:

```
<tipo-almac> tipo-dato nombre[expresion] = {valor1, valor2, ..., valorn};
```

donde **valor1** se refiere al valor a ser asignado al primer elemento del arreglo, **valor2** al segundo y así sucesivamente. La presencia de expresión, que indica el número de elementos del arreglo es opcional cuando los valores iniciales están presentes.

A continuación se dan algunos ejemplos de inicialización de arreglos:

```
int digitos[10] = {0,1,2,3,4,5,6,7,8,9};
static float x[] = {0,0.25,0,-0.50,0,0};
int talle[35] = {38,39,40};
char color[] = {'R','V','A'};
char parrafo[] = {"Curso de Informatica II"};
```

Todos los elementos del arreglo que no tienen asignados valores iniciales explícitos serán puestos automáticamente en cero.

Por ejemplo consideremos el siguiente ejemplo,

```
static float x[6] = {-0.3,0,0.25};
```

dará como resultado de asignación:

```
x[0]=-0.3   x[1] = 0   x[2] = 0.25   x[3] = 0   x[4] = 0   x[5] = 0
```

3.3 Procesamiento de un arreglo

En lenguaje C no se permiten operaciones que impliquen arreglos completos. Así, si a y b son dos arreglos similares (mismo tipo de datos, misma dimensión, mismo tamaño), operaciones de asignación, de comparación, etcétera, deben realizarse elemento por elemento. Esto se hace normalmente dentro de una repetición donde cada paso se usa para procesar un elemento del arreglo. Obviamente que el número de pasos del lazo debe ser igual al número de elementos del arreglo.

3.4 Paso de Arreglos a Funciones

El nombre de un arreglo se puede utilizar como argumento de una función, permitiendo así que el arreglo completo sea pasado a la función.

Para usar un arreglo como argumento de una función, el nombre del arreglo debe aparecer solo, sin corchetes o índices, como un argumento actual dentro de la llamada a la función.

El correspondiente argumento formal se escribe de la misma manera, pero debe ser declarado como un arreglo dentro de la declaración de argumentos formales. Cuando se declara un arreglo unidimensional como un argumento formal, el arreglo se escribe con un par de corchetes vacíos. El tamaño del arreglo no se especifica dentro de la declaración de argumentos formales.

El siguiente ejemplo ilustra el paso de arreglos a funciones,

```
float media(int, float []);     /* Declaracion de funcion */

void main(void){
    int n;                      /* Declaracion de variable */
    float med;                  /* Declaracion de variable */
    float lista[100];           /* Definicion de arreglo */
    ...
    med = media(n,lista);       /* Llamada a la funcion */
    ....
}

/* Definicion de funcion */
float media(int a, float x[]){
    .....
}
```

Dentro de main se invoca la función media. Esta llamada a función contiene dos argumentos actuales: la variable entera n y el arreglo unidimensional en coma flotante *lista*. Notar que *lista* aparece como una variable ordinaria dentro de la llamada a función.

En la primera línea de la función tenemos dos argumentos formales, llamados a y x. La declaración formal de argumentos establece que a es una variable entera y x un arreglo unidimensional en coma flotante. Así hay una correspondencia entre el argumento actual n y el argumento formal a. Similarmente hay una correspondencia entre el argumento actual *lista* y el argumento formal x. Notar que el tamaño de x no se especifica dentro de la declaración formal de argumentos.

Se precisa tener algún cuidado cuando se escriben declaraciones de funciones que incluyen especificaciones de tipo de argumentos. Si alguno de los argumentos es un arreglo, el tipo de datos de cada arreglo debe estar seguido por un par de corchetes, indicando así que el argumento es un arreglo. En el caso de prototipos de funciones, un par de corchetes vacíos debe seguir al tipo de cada argumento arreglo. Similarmente, si la primera línea de una definición de función incluye declaraciones formales de argumentos, cada nombre de arreglo que aparezca como argumento formal debe estar seguido por un par de corchetes vacíos.

Hemos discutido que los argumentos son pasados a la función por valor cuando los argumentos son variables ordinarias. Sin embargo, cuando se pasa un arreglo a una función, los valores de los elementos del arreglo no son pasados a la función. En vez de esto, el nombre del arreglo se interpreta como la dirección del primer elemento del arreglo (la dirección de memoria conteniendo el primer elemento del arreglo). Esta dirección se asigna al correspondiente argumento formal cuando se llama a la función. El argumento formal se convierte por tanto en un puntero al primer elemento del arreglo. Los argumentos pasados de esta manera se dice que son pasados por referencia en vez de por valor.

Cuando se hace una referencia a un elemento del arreglo dentro de la función, el índice del elemento se añade al valor del puntero para indicar la dirección del elemento especificado. Por tanto, cualquier

elemento del arreglo puede ser accedido dentro de la función. Además, si un elemento del arreglo es alterado dentro de la función, esta alteración será reconocida en la porción del programa desde la que fue invocada (en realidad, en todo el ámbito de la definición del arreglo).

El siguiente ejemplo pasa un arreglo de tres elementos enteros a una función que son modificados. Los valores de los elementos del arreglo se escriben en tres partes del programa para ilustrar los efectos de las alteraciones.

```
#include <stdio.h>
void modificar(int []);     /* Declaracion de la funcion */

void main(void) {
    int cont, a[3];         /* Definicion del arreglo */
    printf("\nDesde main, antes de llamar a la funcion: \n");
    for(cont = 0; cont <= 2; ++cont) {
        a[cont] = cont + 1;
        printf("a[%d] = %d\n", cont, a[cont]);
    }
    modificar(a);
    printf("\nDesde main, despues de llamar a la funcion: \n");
    for(cont = 0; cont <= 2; ++cont)
        printf("a[%d] = %d\n", cont, a[cont]);
}

/* Definicion de funcion */
void modificar(int a[]) {
    int cont;
    printf("\nDesde la funcion, despues de modificar los valores:\n");
    for(cont = 0; cont <= 2; ++cont) {
        a[cont] = -9;
        printf("a[%d] = %d\n", cont, a[cont]);
    }
}
```

A los elementos del arreglo se les asignan los valores $a[0] = 1$, $a[1] = 2$, $a[2] = 3$ en el primer for del main. Estos valores se muestran después de ser asignados. Entonces el arreglo se pasa a la función modificar, donde se les asigna el valor -9 a cada uno de los elementos. Estos valores se muestran dentro de la función. Finalmente, los valores del arreglo se muestran de nuevo desde main, una vez que el control ha sido transferido desde la función modificar hasta main.

El hecho de que un arreglo pueda ser modificado globalmente dentro de una función proporciona un mecanismo adecuado para mover múltiples datos de o desde una función a la porción llamante del programa. Simplemente se pasa el arreglo a la función y se alteran sus elementos dentro de ella. O, si el arreglo debe ser preservado, se copia el arreglo (elemento por elemento) dentro de la porción llamante del programa, se pasa la copia a la función y se realizan las alteraciones. El programador

debe tener cierto cuidado al alterar un arreglo dentro de una función pues es muy fácil alterarlo de modo no intencionado fuera de la función.

Consideremos el problema de ordenar una lista de n enteros en una secuencia de valores algebraicos crecientes. Escribamos un programa que realice la reordenación de modo que no se use un almacenamiento innecesario. Por tanto, el programa contendrá solo un arreglo unidimensional de enteros llamado x, en el que se reordenará un elemento cada vez. La reordenación comenzará recorriendo todo el arreglo para encontrar el menor de los números (algebraicamente). Este número será intercambiado con el primer elemento del arreglo, colocando así el menor elemento al principio de la lista. El resto de los $n-1$ elementos del arreglo se recorrerán para encontrar el menor, que se intercambiara con el segundo y así sucesivamente, hasta que el arreglo completo sea reordenado. La reordenación completa requerirá de $n-1$ pasos a lo largo del arreglo, pero cada vez la parte del arreglo a considerar es menor.

Para encontrar el menor de los números en cada paso, comparamos secuencialmente cada número del arreglo, $x[i]$, con el número de comienzo, $x[elem]$, donde $elem$ es una variable entera que se usa para identificar un elemento particular del arreglo. Si $x[i]$ es menor que $x[elem]$, entonces se intercambian los dos números; en otro caso se dejan los dos números en sus posiciones originales.

El programa en C que ejecuta esta tarea se escribe de la siguiente manera:

```
#include <stdio.h>
#define TAM 100

void reordenar(int , int []); /* Declaracion global de la funcion */

void main(void) {
    int i, n, x[TAM];
    /* Leer un valor para n */
    printf("\nCuantos numeros seran introducidos? ");
    scanf("%d",&n);
    fflush(stdin); /* Esta funcion limpia el buffer de teclado */
    printf("\n");
    /* Leer la lista de numeros */
    for(i = 0; i < n; ++i) {
        printf("i = %d  x = ", i + 1);
        scanf("%d", &x[i]);
        fflush(stdin);
    }
    /* reordenar todos los elementos del arreglo */
    reordenar(n,x);
    /* mostrar el arreglo reordenado */
    printf("\n\nLista de numeros reordenada: \n\n");
    for(i = 0; i < n; ++i)
        printf("i = %d  x = %d\n", i + 1, x[i]);
}
```

```
void reordenar(int n, int x[]) {
    int i, elem, temp;
    for (elem = 0;  elem < n-1; ++elem) {
        /* encontrar el menor del resto de los elementos */
        for (i = elem + 1; i < n; ++i) {
            if ( x[i] < x[elem]) {
                /* intercambiar los dos elementos */
                temp = x[elem];
                x[elem] = x[i];
                x[i] = temp;
            }
        }
    }
}
```

3.5 Arreglos multidimensionales

Los arreglos multidimensionales son definidos prácticamente de la misma manera que los arreglos unidimensionales, excepto que se requiere un par separado de corchetes para cada índice. Así un arreglo bidimensional requerirá dos pares de corchetes, un arreglo tridimensional requerirá tres y así sucesivamente.

En términos generales, la definición de un arreglo multidimensional puede escribirse como:

```
<tipo-almac> tipo-dato nombre[expresion1] [expresion2] ... [expresion n]
```

donde **tipo-almac** se refiere al tipo de almacenamiento del arreglo, **tipo-dato** es su tipo de dato, **nombre** especifica el nombre del arreglo y **expresion1, expresion2, ..., expresionn** son expresiones enteras positivas que indican el numero de elementos del arreglo asociados con cada índice. Hemos visto que un arreglo unidimensional de n elementos puede ser visto como una lista de valores. Similarmente un arreglo bidimensional de $m \times n$ elementos puede ser visto como una tabla de valores que tiene m filas y n columnas.

Columna1	**Columna**2	**Columna**3	$\cdots$	**Columna**$n-2$	**Columna**$n-1$
$x[0][0]$	$x[0][1]$	$x[0][2]$	$\cdots$	$x[0][n-2]$	$x[0][n-1]$
$x[1][0]$	$x[1][1]$	$x[1][2]$	$\cdots$	$x[1][n-2]$	$x[1][n-1]$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$x[m-1][0]$	$x[m-1][1]$	$x[m-1][2]$	$\cdots$	$x[m-1][n-2]$	$x[m-1][n-1]$

Las siguientes son definiciones típicas de arreglos multidimensionales,

```
float tabla[50][50];
```

```
char pagina[24][80];
static double registros[100][66][255];
static double registros[L][M][N];
```

Si la definición de un arreglo multidimensional incluye la asignación de valores iniciales, se debe tener cuidado en el orden en que estos valores iniciales serán asignados. La regla es que el último índice se incrementa mas rápidamente, y el primer índice se incrementa mas lentamente. De esta manera en un arreglo bidimensional deben ser asignados por filas, esto es, primero serán asignados los elementos de la primera fila, luego los de la segunda, y así sucesivamente. Considerar la siguiente definición de arreglo bidimensional:

```
int val[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

El arreglo val se puede interpretar como una matriz de tres filas y cuatro columnas. Debido a que los valores se asignan por filas, el resultado de esta asignación será como sigue:

$$\begin{array}{llll} val[0][0] = 1 & val[0][1] = 2 & val[0][2] = 3 & val[0][3] = 4 \\ val[1][0] = 5 & val[1][1] = 6 & val[1][2] = 7 & val[1][3] = 8 \\ val[2][0] = 9 & val[2][1] = 10 & val[2][2] = 11 & val[2][3] = 12 \end{array}$$

El orden natural en el que los valores iniciales son asignados puede alterarse formando grupos de valores iniciales encerrados entre llaves ($\{\cdots\}$). Los valores dentro de cada par interno de llaves serán asignados a los elementos del arreglo cuyo último índice varíe mas rápidamente. Si hay pocos elementos dentro de cada par de llaves, al resto de los elementos de cada fila se le asignarán ceros. Por otra parte, el número de valores dentro de cada para de llaves no puede exceder del tamaño de la fila definido.

El siguiente ejemplo produce las mismas asignaciones iniciales que el último ejemplo.

```
int val[3][4] = {
    { 1, 2, 3, 4},
    { 5, 6, 7, 8},
    { 9, 10, 11, 12}
};
```

Si el arreglo es de mayor dimensión, el concepto se aplica sin modificación:

```
int t[10][20][30] = {
    { /* tabla 1 */
        { 1, 2, 3, 4},      /* fila 1 */
        { 5, 6, 7, 8},      /* fila 2 */
        { 9, 10, 11, 12}    /* fila 3 */
    },
```

```
    { /* tabla 2*/
        { 21, 22, 23, 24},  /* fila 1 */
        { 25, 26, 27, 28},  /* fila 2 */
        { 29, 30, 31, 32}   /* fila 3 */
    }
};
```

Este arreglo hay que interpretarlo como una colección de 10 tablas, con 20 filas y 30 columnas cada una. El grupo de valores iniciales asignados será la siguiente:

$$
\begin{array}{llll}
t[0][0][0] = 1 & t[0][0][1] = 2 & t[0][0][2] = 3 & t[0][0][3] = 4 \\
t[0][1][0] = 5 & t[0][1][1] = 6 & t[0][1][2] = 7 & t[0][1][3] = 8 \\
t[0][2][0] = 9 & t[0][2][1] = 10 & t[0][2][2] = 11 & t[0][2][3] = 12 \\
t[1][0][0] = 21 & t[1][0][1] = 22 & t[1][0][2] = 23 & t[1][0][3] = 24 \\
t[1][1][0] = 25 & t[1][1][1] = 26 & t[1][1][2] = 27 & t[1][1][3] = 28 \\
t[1][2][0] = 29 & t[1][2][1] = 30 & t[1][2][2] = 31 & t[1][2][3] = 32
\end{array}
$$

El resto de los elementos del arreglo tendrán asignados ceros.

Los arreglos multidimensionales se procesan de la misma manera que los arreglos unidimensionales, sobre la base de elemento a elemento. Sin embargo, se requiere algún cuidado cuando se pasan arreglos multidimensionales a una función. En particular, las declaraciones de argumentos formales dentro de la definición de función debe incluir especificaciones explícitas de tamaño en todos los índices excepto en el primero. Estas especificaciones deben ser consistentes con las correspondientes especificaciones de tamaño en el programa llamante. El primer índice puede ser escrito como un par de corchetes vacíos, como en un arreglo unidimensional. Los prototipos correspondientes de función deben escribirse de la misma manera.

El siguiente programa ilustra los conceptos antes mencionados. Supóngase que se desea leer dos matrices de números enteros y calcular su suma, esto es,

```
c[i][j] =  a[i][j] + b[i][j]
```

Para lograr esto supondremos que las matrices tiene igual número de filas y columnas y que no exceden de 20 filas y 30 columnas. El programa se modulariza escribiendo funciones separadas para leer los arreglos, calcular la suma de los elementos del arreglo y escribir el arreglo. Llamaremos a estas funciones leematriz, sumamatriz, escribirmatriz.

```
#include <stdio.h>
#define MAXF  20
#define MAXC  30
/* prototipos de funcion */
void leematriz (int a[][MAXC], int nfilas, int ncols);
void sumamatriz(int a[][MAXC],int b[][MAXC],int c[][MAXC],int nfilas,int ncols);
```

```
void escribirmatriz (int c[][MAXCOL], int nfilas, int ncols);

void main(void) {
    int nfilas, ncols; /* numero de filas y numero de columnas */
    /* definiciones de arreglos */
    int a[MAXF][MAXC], b[MAXF][MAXC], c[MAXF][MAXC];

    printf ("Cuantas filas? ");
    scanf ("%d", &nfilas);
    printf ("\nCuantas columnas? ");
    scanf ("%d", &ncols);

    printf ("\n\nPrimera Matriz: \n");
    leematriz(a, nfilas, ncols);
    printf("\n\nSegunda Matriz: ");
    leematriz(b, nfilas, ncols);
    sumamatriz(a, b, c, nfilas, ncols);
    printf ("\n\nSuma de los Elementos: \n\n");
    escribirmatriz(c, nfilas, ncols);
}

/* funcion para leer una matriz */
void leermatriz(int a[][MAXC], int m, int n) {
    int fila, col;  /* contador de numero de filas y columnas */
    for (fila = 0; fila < m; ++fila) {
        printf ("\nIntrod. datos para la fila nro. %2d\n", fila +1);
        for (col = 0; col < n; ++col)
            scanf ("%d", &a[fila][col]);
    }
}

/* funcion para sumar dos matrices */
void sumamatriz ( int a[][MAXC], int b[][MAXC, int c[][MAXC], int m, int n) {
    int fila, col;

    for (fila = 0; fila < m; ++fila)
        for (col = 0; col < n; ++col)
            c[fila][col] = a[fila][col] + b[fila][col];
}

/* funcion para escribir una matriz en la salida estandar */
void escribirmatriz (int a[][MAXC], int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila) {
        for (col = 0; col < n; ++col)
```

```
            printf ("%4d", a[fila][col]);
        printf("\n");
    }
}
```

Notar la manera de escribir las declaraciones formales de argumentos dentro de cada definición de función.

3.6 Arreglos y cadenas de caracteres

Ya se ha visto que una cadena de caracteres (strings) puede ser representada por un arreglo unidimensional de caracteres. Cada caracter de la cadena será almacenado en un elemento del arreglo. Algunos problemas requieren que los caracteres de la cadena sean procesados individualmente. No obstante, hay muchos otros en los que se requiere que las cadenas se procesen como entidades completas. Tales problemas pueden simplificarse considerablemente usando funciones especiales de biblioteca orientadas a cadenas.

Por ejemplo, la mayoría de los compiladores C incluyen funciones de biblioteca que permiten comparar cadenas, copiarlas o concatenarlas (combinadas una detrás de la otra). Otras funciones de biblioteca permiten operaciones sobre caracteres individuales dentro de cadenas, por ejemplo permiten encontrar caracteres individuales dentro de una cadena, etc. La biblioteca estándar para el manejo de cadena de caracteres es **string.h**. Dentro de esta biblioteca existen un gran número de funciones entre las cuales podemos citar:

```
char* strcpy(char destino[ ], char fuente[ ]);
```

Esta función permite copiar cadenas de caracteres, desde la cadena fuente a la cadena destino. Notar que la cadena destino debe ser de un tamaño mayor o igual que la cadena fuente.

```
char* strcat(char destino[ ], char fuente[ ]);
```

Esta función permite concatenar (unir) dos cadenas de caracteres, es decir agregarle a una cadena otra. Al igual que la función anterior, la cadena destino debe poseer un tamaño tal que pueda alojar las dos cadenas.

```
int strcmp(char s1[ ],char s2[ ]);
```

Esta función permite comparar dos cadenas de caracteres y devuelve un valor entero en consecuencia. Si el resultado es positivo significa que la cadena *s1* es mayor a la cadena *s2*, es decir se encuentra después en orden alfabético. Si el resultado es negativo entonces la cadena *s1* estará antes alfabéticamente que *s2*, y si el resultado es cero, las dos cadenas son iguales. Recordemos que la forma de trabajo de esta función es la comparación de los códigos ASCII de cada carácter, por lo tanto las mayúsculas y las minúsculas son tratadas como letras diferentes.

```
int strlen(char s1[ ]);
```

Finalmente, esta función permite averiguar la longitud de una cadena de caracteres.

El siguiente ejemplo ilustra el uso de algunas de estas funciones de biblioteca.

Ordenar una lista de cadenas de caracteres. Supongamos que queremos introducir una lista de cadenas en la computadora, reordenarlas alfabéticamente y escribir la lista reordenada. La estrategia para hacer esto es similar a la usada para reordenar números. Sin embargo, ahora existe la complicación adicional de comparar cadenas de caracteres completas, en lugar de valores numéricos simples. Por tanto almacenamos las cadenas en un arreglo bidimensional de caracteres. Cada cadena será almacenada en una fila distinta del arreglo.

Para simplificar el proceso hacemos uso de las funciones de biblioteca **strcmpi** y **strcpy**. Estas funciones se usan para comparar dos cadenas y para copiar una cadena en otra, respectivamente. (Notar que **strcmpi** es una variación de la función mas común **strcmp**, que compra cadenas pero sin diferenciar entre caracteres en mayúsculas y minúsculas).

Por tanto, si **strcmpi(cadena1, cadena2)** retorna un valor positivo, indicaría que la cadena2 debe ir antes que cadena1 para colocar las cadenas en orden alfabético.

El programa completo es similar al programa de ordenación de números presentado anteriormente. Sin embargo, ahora se permite que el programa acepte un número no especificado de cadenas hasta que se introduce una cadena igual a "FIN" (en minúsculas o mayúsculas). El programa contará las cadenas según sean introducidas, ignorando la última cadena que contiene "FIN".

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

void ordenar (int n, char x[][12]); /* prototipo de funcion */

void main(void){
    int i, n = 0;
    char x[10][12];
    printf ("Introducir debajo cada cadena en una linea separada\n\n");
    printf("Escribir \'FIN\' para terminar\n\n");
    /* Leer la lista de cadenas */
    do  {
        printf ("cadena %d: ", n + 1);
        gets(x[n]);
    } while(strcmpi( x[n++], "FIN"));
    /* ordenar la lista de cadenas */
    ordenar (--n, x);
    /* mostrar la lista ordenada de cadenas */
    printf ("\n\nLista reordenada de cadenas: \n");
    for ( i = 0; i < n; ++i)
```

```
        printf("\ncadena %d: %s", i+1, x[i]);
}

/* funcion para reordenar la lista de cadenas */
void ordenar( int n, char x[][12]){
    char temp[12];
    int i, elem;

    for (elem = 0; elem < n - 1; ++elem){
    /* encontrar la menor de las cadenas restantes */
        for (i = elem + 1; i < n; ++i)
            if (strcmpi (x[elem], x[i]) > 0){
            /* intercambiar las dos cadenas */
                strcpy (temp, x[elem]);
                strcpy (x[elem], x[i]);
                strcpy (x[i], temp);
            }
    }
}
```

Capítulo 4

Punteros

4.1 Introducción

Un puntero es una variable que representa la posición (la dirección) de otro dato, tal como una variable o un elemento de un arreglo. Los punteros son usados frecuentemente en C y tienen gran cantidad de aplicaciones. El uso cuidadoso de los punteros provee claridad y simplicidad al programa. Por ejemplo pueden usarse para pasar información entre una función y sus puntos de llamada. En particular proporcionan una forma de devolver varios datos desde una función mediante los argumentos de la misma. Además permiten asignar dinamicamente memoria, manejo de listas encadenadas y acceso a distintas posiciones dentro de una misma estructura de datos por ejemplo arreglos, cadenas, estructuras, etc.

Los punteros están muy relacionados con los arreglos y proporcionan una vía alternativa de acceso a los elementos individuales del arreglo. Es más, proporcionan una forma conveniente para representar arreglos multidimensionales.

4.2 Conceptos básicos.

Dentro de la memoria de la computadora cada dato almacenado ocupa una o mas celdas de memoria contiguas. El número de celdas de memoria requeridas para almacenar un dato depende de su tipo (es decir si el dato es char, int, float, etc). Por ejemplo, un caracter será almacenado normalmente en un byte, un entero en dos bytes contiguos, un flotante puede necesitar cuatro y una cantidad en doble precisión puede requerir ocho bytes contiguos. Suponga que `v` es un variable que representa cierto dato. El compilador automáticamente asignará celdas de memoria para ese dato. Se puede acceder al dato si conocemos la localización (la dirección) de la primera celda de memoria. La dirección de memoria de la variable `v` puede determinarse mediante la expresión `&v`, donde `&` es un operador monario, llamado el operador de dirección, que proporciona la dirección del operando. Ahora vamos a asignar la dirección de `v` a otra variable, `pv`. Así,

```
pv = &v;
```

Esta nueva variable (pv) es un puntero a la variable v. Recordar, sin embargo, que pv representa la dirección de v y no su valor. De esta forma pv se refiere como variable puntero.

El dato representado por v puede ser accedido por la expresión *pv, donde * es un operador monario, llamado el operador indirección, que opera solo sobre una variable puntero. Por lo tanto, *pv y v representan el mismo dato (el contenido de las mismas celdas de memoria). Además, si escribimos,

```
pv = &v;
u = *pv;
```

entonces u y v representan el mismo valor (se asume que u y v están declaradas como variables del mismo tipo de dato).

Los operadores monarios & y * son miembros del mismo grupo de precedencia que los otros operadores monarios, por ejemplo -, ++, --, !, sizeof(tipo), que fueron presentados en el capitulo 2. Se debe recordar que este grupo de operadores tiene mayor precedencia que el de los operadores aritméticos y la asociatividad de los operadores monarios es de derecha a izquierda. El operador & sólo puede actuar sobre operandos con dirección única como variables ordinarias o elementos de un arreglo. Por consiguiente, el operador dirección no puede actuar sobre expresiones aritméticas, tales como 2 * (u + v). El operador indirección * sólo puede actuar sobre operandos que sean punteros. Sin embargo, si pv apunta a v (pv = &v), entonces una expresión como *pv puede ser intercambiable con la correspondiente variable v. Así una referencia indirecta puede aparecer en lugar de una variable dentro de una expresión mas complicada.

Considere el siguiente ejemplo,

```
#include <stdio.h>
void main(void ) {
    int u1, u2;
    int v = 3;
    int *pv; /* puntero a enteros */
    u1 = 2 * (v + 5); /* expresion ordinaria */
    pv = &v; /* pv apunta a v */
    u2 = 2 * (*pv + 5); /*expresion alternativa */
    printf("\nu1 = %d  u2 = %d", u1, u2);
}
```

En este programa la salida u1 y u2 son iguales ya que v y *pv representan al mismo dato.

Las variables puntero pueden apuntar a variables numéricas o de caracteres, arreglos, funciones u otras variables puntero, y a otras estructuras de datos como veremos posteriormente. Por tanto, a una variable puntero se le puede asignar la dirección de una variable ordinaria (por ejemplo pv = &v). También se le puede asignar la dirección de otra variable puntero (por ejemplo pv = px), siempre que ambas variables puntero apunten al mismo tipo de dato. Además, a una variable puntero se le puede asignar un valor nulo (cero), como se explicará posteriormente. Por otra parte, a variables ordinarias no se les puede asignar direcciones arbitrarias (una expresión como &x no puede aparecer en la parte izquierda de una sentencia de asignación).

4.3 Declaración de Punteros

Los punteros como cualquier otra variable, deben ser declarados antes de ser usados dentro de un programa en C. Sin embargo, la interpretación de una declaración de puntero es un poco diferente a la declaración de otras variables. Cuando una variable puntero es definida, el nombre de la variable debe ir precedida por un `*`. Este identifica que la variable es un puntero. El tipo de dato que aparece en la declaración se refiere al tipo de dato al que hará referencia el puntero.

El formato de la declaración de un puntero es el siguiente:

```
<tipo_apuntado> *<nbre_puntero>;
```

donde `nbre-puntero` es el nombre de la variable puntero y `tipo-apuntado` se refiere al tipo de dato apuntado por el puntero.

Por ejemplo,

```
int *px;
float *pv;
```

Esta declaración limita el tipo de los datos a los que puede apuntar el puntero. Por ejemplo en el caso anterior:

```
int *px;
```

significa que `px` solo debe apuntar a datos de tipo entero o convertibles a ese tipo.

Es posible hacer la siguientes asignaciones:

```
*px=0;  /* asigna cero a la direccion donde apunta px */
*px+=1; /* incrementa en uno el contenido apuntado por px */
px=py;  /* asigna la direccion de py a px  */
```

Las siguientes asignaciones no son equivalentes:

```
(*px)++; /* Incrementa en uno el contenido apuntado por px */
*px++;   /* Incrementa en uno el puntero y toma el valor apuntado por px */
```

El siguiente ejemplo muestra la declaración de variables punteros y su asignación:

```
void main(void) {
    int ivar,*iptr;
    iptr=&ivar;  /* la direccion de ivar es asignada a iptr */
```

```
    ivar=122;
    printf("Direccion de ivar      : %p\n",&ivar);
    printf("Contenido de ivar      : %d\n",ivar);
    printf("Contenido de iptr      : %p\n",iptr);
    printf("Valor apuntado por iptr: %d\n",*iptr);
}
```

Tras la ejecución de este programa los resultado serán:

```
Direccion de ivar      : 166E
Contenido de ivar      : 122
Contenido de iptr      : 166E
Valor apuntado por iptr: 122
```

La función main declara dos variables `ivar` e `iptr`. La primera es una variable entera, mientras que la segunda es un puntero a un dato de tipo entero. Dentro de la función printf, en la cadena de control aparece el caracter de formato `%p` que se utiliza para exteriorizar el contenido de variables puntero.

Dentro de la declaración de variable, una variable puntero puede ser inicializada asignándole la dirección de otra variable. Tener en cuenta que la variable cuya dirección se asigna al puntero debe estar previamente declarada en el programa. Por ejemplo, un programa contiene las siguientes declaraciones,

```
float u, v;
float *pv = &v;
```

Las variables `u` y `v` son declaradas como variables de punto flotante y `pv` es declarada como una variable puntero que apunta a cantidades de punto flotante y se le asigna la dirección de la variable `v`.

4.4 Paso de punteros a una función

Debido a que el lenguaje C pasa los argumentos por valor, no existe forma directa de que una función llamada altere una variable de la función llamadora. Como hacer si hubiera que modificar realmente el contenido de los argumentos pasados?. Por ejemplo la función `intercambio()`, la cual tiene como objetivo intercambiar el contenido de dos variables enteras. En un primer intento podemos escribir la siguiente función:

```
void intercambio(int x,int y) {
    int temp;
    temp=x;
```

```
    x=y;
    y=temp;
}
```

Esta función no cumple con los requerimientos de modificación de las variables pasadas como parámetros actuales. Pues los parámetros formales `x` e `y` reciben el valor y no la dirección de los argumentos actuales. De esta forma no alteran las variables pasadas por la función llamadora. Tratando de corregir los errores anteriores escribimos la siguiente versión de la función `intercambio()`.

```
void intercambio(int *px, int *py) {
    int temp;
    temp=*x;
    *px=*py;
    *py=temp;
}
```

Se puede apreciar ahora que los parámetros pasados son direcciones de variables enteras, de esta forma utilizando punteros a ellas se logra el objetivo, es decir alterar los valores de las variables pasadas por la función llamadora.

En este caso se debe invocar la función `intercambio()` de la siguiente manera,

```
intercambio(&x,&y);
```

Un ejemplo que ya hemos visto es la función de biblioteca `scanf`, que requiere que sus argumentos vayan precedidos por `&`. Así la función `scanf` requiere que sea especificada la dirección de los elementos que vayan a ser introducidos en la memoria de la computadora. El operador `&` (ampersand) proporciona entonces un medio eficaz para acceder a las direcciones de variables ordinarias univaluadas.

4.5 Punteros y arreglos unidimensionales

En C existe una estrecha relación entre punteros y arreglos. Las operaciones que se realizan con uno de ellos se pueden también hacer con el otro.

Recuerde que el nombre de un arreglo es realmente un puntero al primer elemento de ese arreglo. Sin embargo, si `x` es un arreglo unidimensional, entonces la dirección del primer elemento del arreglo puede expresarse tanto como `&x[0]` o simplemente como `x`. Además la dirección del segundo elemento del arreglo se puede escribir como `&x[1]` o como `(x + 1)`, y así sucesivamente. En general la dirección del elemento `(i + 1)` del arreglo se puede expresar o bien como `&x[i]` o como `(x + i)`. Por lo tanto, se tienen dos modos diferentes de escribir la dirección de cualquier elemento de un arreglo: se puede escribir el elemento real precedido del operador `&`, o se puede escribir una expresión en la cual el índice se añade al nombre del arreglo.

Debe entenderse que se esta tratando con un tipo especial e inusual de expresión. En la expresión `(x + i)`, por ejemplo, `x` representa una dirección e `i` representa una cantidad entera. Además `x` es el nombre de un arreglo cuyos elementos pueden ser caracteres, enteros, etc. Por lo tanto, no se está simplemente sumando valores numéricos. Mas bien se esta especificando una dirección que está un cierto número de posiciones mas allá de la dirección del primer elemento. En términos simples, se está especificando una localización que está `i` elementos del arreglo detrás del primero. Cuando el índice `i` se utiliza de esta manera recibe el nombre de desplazamiento. Puesto que `&x[i]` y `(x + i)` representan la dirección del `i-esimo` elemento del arreglo `x`, sería razonable que `x[i]` y `*(x + i)` representen el contenido de esa dirección, o sea el valor del `i-esimo` elemento del arreglo `x`. Además esto es cierto. Los dos términos son intercambiables. Cada término puede utilizarse en cualquier aplicación particular. La elección depende únicamente de las preferencias del programador.

El siguiente ejemplo ilustra la relación entre los elementos de un arreglo y sus direcciones,

```
#include <stdio.h>
void main(void) {
    static int x[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
    int i;
    for (i = 0; i <= 9; ++i)
        printf ("\nx[%d]= %d\t*(x + %d)= %d\t&x[%d]= %p\tx + %d=%p",
                i,x[i],i,*(x + i),i,&x[i],i,x + i);
}
```

Ejecutando el programa resulta la siguiente salida,

```
x[0]= 10    *(x + 0)= 10    &x[0]= 0194 x + 0= 0194
x[1]= 11    *(x + 1)= 11    &x[1]= 0196 x + 1= 0196
x[2]= 12    *(x + 2)= 12    &x[2]= 0198 x + 2= 0198
x[3]= 13    *(x + 3)= 13    &x[3]= 019A x + 3= 019A
x[4]= 14    *(x + 4)= 14    &x[4]= 019C x + 4= 019C
x[5]= 15    *(x + 5)= 15    &x[5]= 019E x + 5= 019E
x[6]= 16    *(x + 6)= 16    &x[6]= 01A0 x + 6= 01A0
x[7]= 17    *(x + 7)= 17    &x[7]= 01A2 x + 7= 01A2
x[8]= 18    *(x + 8)= 18    &x[8]= 01A4 x + 8= 01A4
x[9]= 19    *(x + 9)= 19    &x[9]= 01A6 x + 9= 01A6
```

Cuando asignamos un valor a un elemento de un arreglo, como `x[i]`, la parte izquierda de la asignación puede escribirse como `x[i]` o como `*(x + i)`. De esta manera, un valor puede asignarse directamente a un elemento del arreglo o al área de memoria cuya dirección es la del elemento del arreglo. Por otra parte, a veces es necesario asignar una dirección a un identificador. En tales situaciones una variable puntero puede aparecer en la parte izquierda de la asignación. No es posible asignar una dirección arbitraria a un nombre de arreglo o a un elemento del mismo. Por lo tanto, expresiones como `x`, `(x + i)` y `&x[i]` no pueden aparecer en la parte izquierda de una asignación. Además, la dirección de un arreglo no puede alterarse arbitrariamente, por lo que

expresiones como `++x` no están permitidas. Para ejemplificar algunos de los aspectos enunciados consideremos el siguiente esqueleto de programa,

```
#include <stdio.h>
void main(void) {
    int linea[80];
    int *pl;
    ...
    /*asignar valores */
    linea[2] = linea[1];
    linea[2] = *(linea + 1);
    *(linea + 2) = linea[1];
    *(linea + 2) = *(linea + 1);
    /* asignar direcciones */
    pl = linea;
    pl = &linea[0];
    pl = &linea[1];
    pl = linea + 1;
}
```

Cada una de las primeras cuatro sentencias de asignación le asigna el valor del segundo elemento del arreglo (`linea[1]`) al tercer elemento del arreglo (`linea[2]`), por lo tanto las cuatro son sentencias equivalentes. Cada una de las cuatro ultimas sentencias asignan direcciones. Las dos primeras asignan la dirección del primer elemento del arreglo al puntero `pl`. Las dos ultimas asignan la dirección del segundo elemento del arreglo al puntero `pl`.

Hemos establecido que un nombre de arreglo es en realidad un puntero al primer elemento dentro del arreglo. Por lo tanto, debe ser posible definir un arreglo como una variable puntero en lugar de definirlo como un arreglo convencional. Sintácticamente, las dos definiciones son equivalentes. Sin embargo, la definición convencional de un arreglo produce la reserva de un bloque fijo de memoria al principio de la ejecución de un programa, mientras que esto no ocurre si el arreglo se representa en términos de una variable puntero. Como resultado de uso de una variable puntero para representar un arreglo se necesita algún tipo de asignación inicial de memoria, antes de que los elementos del arreglo sean procesados. Generalmente, tales tipos de reserva de memoria se realizan usando la función de biblioteca `malloc` (Memory ALLOCation), pero el método exacto varia de una aplicación a otra. Si el arreglo ha sido definido como una variable puntero, no se le pueden asignar valores iniciales numéricos. Por tanto, será necesario una definición convencional si se quiere asignar valores a los elementos de un arreglo numérico.

Para ilustrar el uso de punteros, considérese de nuevo el problema de ordenar una lista de enteros que fue descrito en el capitulo de arreglos.

```
#include <stdio.h>
void reordenar(int n, int *x);
void main(void){
```

```
    int i, n, *x;
    /* Leer el valor para n */
    printf ("\nCuantos numeros seran introducidos? ");
    scanf ("%d", &n);
    printf("\n");
    /* reserva de memoria */
    x = (int *) malloc(n * sizeof(int));
    /* leer la tabla de numeros */
    for (i = 0; i < n; ++i){
        printf("\ni= %d   x = ", i + 1);
        scanf ("%d", x + i);
        }
    /* reordena todos los elementos del arreglo */
    reordenar (n, x);
    /* mostrar la tabla reordenada */
    printf ("\n\n La Tabla de numeros reordenada es: ");
    for (i = 0; i < n; ++i){
        printf ("i = %d   x = %d\n", i + 1, *(x + i));
    }
    free(x); /*se libera la memoria solicitada */
}
void reordenar (int n, int *x){
    int i, elem, temp;
    for (elem = 0; elem < n - 1; ++elem)
        /* buscar el menor del resto de los elementos */
        for (i = elem + 1; i < n; ++i)
            if (*(x + i) < *(x + elem)) {
                /* intercambiar los elementos */
                temp = *(x + elem);
                *(x + elem) = *(x + i);
                *(x + i) = temp;
                }
}
```

4.6 Operaciones con punteros

En la sección anterior se ha visto que se puede sumar un valor entero al nombre de un arreglo para acceder a un elemento individual del mismo. El valor entero es interpretado como el índice del arreglo. Esto funciona ya que todos los elementos del arreglo son del mismo tipo y por lo tanto cada elemento ocupa el mismo número de celdas de memoria. El número de celdas que separan a dos elementos del arreglo dependerán del tipo de datos del arreglo, pero de esto se encarga automáticamente el compilador y por tanto no concierne al programador directamente. Este concepto puede extenderse a variables puntero. En particular, un valor entero puede sumarse o restarse a una variable puntero, pero el resultado de la operación debe interpretarse con cuidado.

Supongamos por ejemplo, que `px` es una variable puntero que representa la dirección de una variable `x`. Podemos escribir expresiones como `++px`, `--px`, `(px + 3)`, `(px + i)` y `(px - i)`, donde `i` es una variable entera. Cada expresión representará una dirección localizada a cierta distancia de la posición original representada por `px`. La distancia exacta será el producto de la cantidad entera por el número de bytes que ocupa cada elemento al cual apunta `px`. Por ejemplo, si `px` apunta a un entero, como un entero requiere 2 bytes de memoria, entonces la expresión `(px + 3)` resultará en una dirección de 6 bytes más allá del entero apuntado por `px`.

Considere el siguiente programa en C para ilustrar lo antes mencionado,

```
#include <stdio.h>
void main(void) {
    int *px;  /* Puntero a un entero */
    int i = 1;
    float f = 0.3;
    double d = 0.005;
    char c = '*';
    px = &i; /* Asigna la direccion de i al puntero px */
    printf ("Valores: i=%d f=%f d=%f c=%c\n\n", i, f, d, c);
    printf ("Direcciones: &i=%X &f=%X &d=%X &c=%X\n\n", &i, &f, &d, &c);
    printf ("Valores de punteros: px=%X px+1=%X px+2=%X px+3=%X\n",
             px, px+1,px+2,px+3);
}
```

Este programa muestra los valores y las direcciones asociados con cuatro tipos diferentes de variables. El programa también hace uso de una variable puntero, `px`, que representa la dirección de `i`. También se muestran los valores de `px`, `px+1`, `px+2`, `px+3` para que puedan ser comparados con las direcciones de las diferentes variables. La ejecución del programa podría producir la siguiente salida,

```
Valores:              i=1        f=0.300000      d=0.005000      c=*
Direcciones:          &i=117E    &f=1180         &d=1186         &c=118E
Valores de punteros: px=117E    px+1=1180       px+2=1182       px+3=1184
```

Una variable puntero puede restarse de otra siempre que ambas variables apunten a elementos del mismo arreglo. El valor resultante indica el numero de palabras o bytes que separan a los correspondientes elementos del arreglo. En el siguiente programa, dos variables puntero apuntan al primero y al último elemento de un arreglo de enteros.

```
#include <stdio.h>
void main(void) {
    int *px, *py; /* punteros a enteros */
    int a[6] = { 1, 2, 3, 4, 5, 6};
    px = &a[0];
```

```
    py = &a[5];
    printf ("px=%X py=%X", px, py);
    printf ("\n\npy - px = %X", py - px);
}
```

Ejecutando el programa se produce la siguiente salida,

```
px= 52   py= 5C
py - px = A
```

La primera linea indica que la dirección de `a[0]` es 52 y la dirección de `a[5]` es $5C$. La diferencia entre estos dos numeros hexadecimales es 10 en decimal (A en hexadecimal).

Las variables puntero pueden comparase siempre que estas apunten a un mismo tipo de dato. Tales comparaciones pueden ser útiles cuando ambas variables apunten a elementos de un mismo arreglo. Las comparaciones pueden probar la igualdad o desigualdad. Además, una variable puntero puede comparase con cero (normalmente expresado como la constante simbólica `NULL`).

Suponga que `px` y `py` son variables puntero que apuntan a elementos de un mismo arreglo. Algunas expresiones relacionando esas variables se muestran a continuación. Todas son sintácticamente correctas.

```
(px < py)
(px >= py)
(px == py)
(px != py)
(px == NULL)
```

Estas expresiones pueden usarse de la misma manera que cualquier otra expresión lógica.

```
if (px < py)
    printf("px < py");
else
    printf("px >= py);
```

Finalmente, debe quedar claro que las operaciones discutidas previamente sólo pueden realizarse con punteros. Estas operaciones están resumidas a continuación:

1. A una variable puntero se le puede asignar la dirección de una variable ordinaria (`pv = &v`).
2. A una variable puntero se le puede asignar el valor de otra variable puntero, (por ejemplo (`px = py`)), siempre que ambos apunten al mismo tipo de dato.
3. A una variable puntero se le puede asignar un valor nulo (cero)(por ejemplo `pv = NULL`).

4. Una cantidad entera puede ser sumada o restada a una variable puntero (por ej, `++px`, `pv + 3`).

5. Una variable puntero puede ser restada de otra con tal que ambas apunten a elementos del mismo arreglo.

6. Dos variables puntero pueden compararse siempre y cuando apunten a datos del mismo tipo.

No se permiten otras operaciones aritméticas con punteros. Así una variable puntero no puede ser multiplicada por una constante, dos punteros no pueden sumarse, etc.

4.7 Asignación dinámica de memoria

En la mayoría de los algorítmos que un programador debe realizar, normalmente no se dispone de la cantidad de datos con las cuales debe trabajar. Esto trae aparejado el problema de dimensionar las estructuras de datos (por ejemplo arreglos) adecuadamente, lo que no es sencillo de realizar. Por ejemplo un programador puede dimensionar un arreglo para que contenga 100 elementos de tipo entero y durante la ejecución del programa el usuario podría trabajar unicamente con 10 desperdiciando 90 valiosos lugares de memoria. Por el contrario también podría suceder que el usuario quisiera trabajar con 300 datos en cuyo caso no le sería posible pues el programa admite como máximo 100. Nos preguntamos entonces si no existirá algún mecanismo que me permita asignar memoria en el momento de la ejecución del programa y no en el momento de la compilación como sucede con los arreglos?. La respuesta es que si existe un mecanismo de este tipo que se denomina asignación dinámica de memoria, pues permite reservar memoria en tiempo de ejecución.

El Lenguaje de Programación C dispone de una biblioteca de funciones para este fin, el nombre de tal biblioteca es `<alloc.h>`. Existen varias funciones dentro de esta biblioteca para el fin específico mencionado, podemos citar a 2 de ellas, la función `malloc()` y la función `free()`.

La función `malloc` está declarada de la siguiente manera:

```
(void *) malloc(int cantidad_bytes)
```

Esta función lo que hace es solicitarle al Sistema Operativo un cierto bloque de memoria de longitud `cantidad_bytes`. El bloque de memoria debe venir expresado en bytes y no en datos. El Sistema Operativo responde a nuestro programa entregando la dirección del primer byte del bloque de memoria asignada o en su defecto la constante `NULL` para indicar que lo solicitado es imposible de ser asignado.

Supongamos el siguiente ejemplo aclaratorio:

```
.....
int *pv;
int n=100;
```

```
if((pv = (int *)malloc(100 * sizeof(int)))==NULL) {
    printf("No hay memoria disponible\n");
    exit(1);
}
...
free(pv);
```

En este ejemplo se solicita memoria para 100 datos de tipo entero y a través de la sentencia `if` se verifica si el Sistema Operativo accedió al pedido o no. En la sentencia se puede ver que la función `malloc` retorna un puntero (`pv`) al cual se le realiza un `cast` de tipo para indicar que el bloque de memoria solicitado es para datos de tipo entero.

La sentencia final del ejemplo anterior, utiliza la función `free()` para devolverle al sistema operativo la memoria solicitada. Esta acción de devolución debe realizarse cuando los datos en la memoria no sean de interés para nuestro programa.

4.8 Punteros y arreglos multidimensionales.

Como un arreglo unidimensional puede representarse en términos de punteros, el nombre del arreglo y un desplazamiento (índice), es razonable esperar que los arreglos multidimensionales puedan representarse con una notación equivalente de punteros. En efecto este es el caso, por ejemplo, un arreglo bidimensional es en realidad un conjunto de arreglos unidimensionales.

4.8.1 Puntero a un conjunto de arreglos unidimensionales

Se puede definir un arreglo bidimensional como un puntero a un grupo contiguo de arreglos unidimensionales. La declaración sería:

```
<tipo-dato> (*ptvar)[expresion2];
```

en lugar de,

```
<tipo-dato> <nombre_arreglo> [expresion1][expresion2];
```

por ejemplo,

```
int (*x) [20];
```

en lugar de,

```
int x[10][20];
```

Generalizando este concepto para arreglos multidimensionales se tiene,

```
<tipo-dato>  (*ptvar)[expresion2][expresion3]...[expresion n];
```

que reemplaza a,

```
<tipo-dato> <nombre-arreglo> [expresion1][expresion2]...[expresion n];
```

Notar los paréntesis que rodean al nombre del arreglo y el asterisco que lo precede en la versión de punteros. Estos paréntesis deben estar presentes. Sin ellos estaremos definiendo un arreglo de punteros en lugar de un puntero a un grupo de arreglos, debido a que estos símbolos (paréntesis y asterisco) se evalúan de derecha a izquierda.

Un elemento individual del arreglo se puede acceder mediante el uso repetido del operador indirección (`*`). Sin embargo, normalmente este método es más difícil que el convencional para acceder a un elemento del arreglo. El siguiente ejemplo ilustra el uso de este operador.

Sea `x` un arreglo bidimensional con 10 filas y 20 columnas, entonces,

```
int x[10][20];
```

El elemento en la fila 2 columna 5 se puede acceder de las siguientes dos maneras,

```
x[2][5] o *(*x+2)+5)
```

Como ejemplo general de uso de esta técnica repetiremos el ejemplo de la suma de matrices resuelto anteriormente con arreglos.

```
#include <stdio.h>
#include <alloc.h>
#define MCOL 30
void leematriz(int (*a)[MCOL], int nfilas, int ncols);
void sumamatriz(int (*a)[MCOL],int (*b)[MCOL],int (*c)[MCOL], int nfilas, int ncols);
void escribematriz(int (*c)[MCOL],int nfilas, int ncols);
/* cada arreglo bidimensional se procesa como un puntero
   a un conjunto de arreglos unidimensionales */
void main(void)  {
    int nfilas, ncols;
    /* definicion de punteros */
    int (*a)[MCOL], (*b)[MCOL], (*c)[MCOL];
    printf ("Cuantas filas?");
    scanf ("%d",&nfilas);
    printf("\nCuantas columnas?");
```

```
    scanf ("%d", &ncols);
    /*reserva espacio de memoria*/
    a = (int *) malloc( nfilas * ncols * sizeof(int));
    b = (int *) malloc( nfilas * ncols * sizeof(int));
    c = (int *) malloc( nfilas * ncols * sizeof(int));
    printf("\n\nPrimera Matriz:\n");
    leematriz(a, nfilas, ncols);
    printf("\n\nSegunda Matriz:\n");
    leematriz(b, nfilas, ncols);
    sumamatriz(a, b, c, nfilas, ncols);
    printf ("\n\nMatriz Suma:\n");
    escribematriz(c, nfilas, ncols);
}
/* Lee matriz*/
void leematriz(int (*a)[MCOL], int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila) {
        printf ("\nIntroducir datos para la fila %2d\n", fila + 1);
        for (col = 0; col < n; ++col) {
            printf ("  x[%d][%d]= ",fila + 1, col + 1);
            scanf ("%d", (*(a+fila)+col));
            }
        }
}
/* Suma Matrices */
void sumamatriz(int (*a)[MCOL],int (*b)[MCOL],int (*c)[MCOL],int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila)
        for (col = 0; col < n; ++col)
            *(*(c+fila)+col) = *(*(a+fila)+col) + *(*(b+fila)+col);
}
/* Escribe el resultado */
void escribematriz(int (*c)[MCOL], int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila) {
        for (col = 0; col < n; ++col)
            printf ("  c[%d][%d]= %4d",fila + 1, col + 1,*(*(c+fila)+col));
        printf("\n");
        }
}
```

4.8.2 Arreglo de punteros

Un arreglo multidimensional puede ser expresado como un arreglo de punteros en lugar de como un puntero a un grupo contiguo de arreglos. En estos casos el nuevo arreglo será de una dimensión menor que el arreglo multidimensional. Cada puntero indicará el principio de un arreglo de dimensión $n-1$.

En general un arreglo bidimensional puede ser definido como un arreglo unidimensional de punteros escribiendo:

```
<tipo-dato>  *nombre_puntero[expresion1];
```

En lugar de la definición convencional,

```
<tipo-dato>  nombre_arreglo[expresion 1][expresion 2];
```

Similarmente un arreglo n-dimensional puede definirse como un arreglo de punteros de dimensión $(n-1)$ escribiendo,

```
<tipo-dato>  *nombre_puntero[expresion1][expresion2]...[expresion(n-1)];
```

en lugar de,

```
<tipo-dato>  nombre_arreglo[expresion1][expresion2]...[expresion n];
```

Notar que el nombre del arreglo precedido por asterisco no está encerrado entre paréntesis en este tipo de declaración. Así la regla de precedencia de derecha a izquierda asocia primero el par de corchetes con el arreglo, definiéndolo como tal. El asterisco que lo precede establece que el arreglo contendrá punteros que apuntan al tipo de dato especificado. Como ejemplo consideremos otra versión del programa que suma matrices.

```
#include <stdio.h>
#include <alloc.h>
#define MFIL 30
/* prototipos de funcion */
void leematriz(int *a[MFIL], int nfilas, int ncols);
void sumamatriz(int *a[MFIL],int *b[MFIL],int *c[MFIL],int nfilas, int ncols);
void escribematriz(int *c[MFIL],int nfilas, int ncols);
/* cada arreglo bidimensional se representa como un arreglo
   de punteros, donde cada puntero indica una fila del arreglo
   bidimensional original */
void main(void) {
```

```
    int fila, nfilas, ncols;
    /* definicion de punteros */
    int *a[MFIL], *b[MFIL], *c[MFIL];
    printf ("Cuantas filas?");
    scanf ("%d",&nfilas);
    printf("\nCuantas columnas?");
    scanf ("%d", &ncols);
    /*reserva espacio de memoria*/
    for (fila =0; fila <= nfilas; ++fila) {
        a[fila] = (int *) malloc (ncols * sizeof(int));
        b[fila] = (int *) malloc (ncols * sizeof(int));
        c[fila] = (int *) malloc (ncols * sizeof(int));
        }
    printf("\n\nPrimera Matriz:\n");
    leematriz(a, nfilas, ncols);
    printf("\n\nSegunda Matriz:\n");
    leematriz(b, nfilas, ncols);
    sumamatriz(a, b, c, nfilas, ncols);
    printf ("\n\nMatriz Suma:\n");
    escribematriz(c, nfilas, ncols);
}
/* Lee matriz*/
void leematriz(int *a[MFIL], int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila) {
        printf ("\nIntroducir datos para la fila %2d\n", fila + 1);
        for (col = 0; col < n; ++col) {
            printf ("  x[%d][%d]= ",fila + 1, col + 1);
            scanf ("%d", (a[fila]+col));
            }
        }
}
/* Suma Matrices */
void sumamatriz(int *a[MFIL],int *b[MFIL],int *c[MFIL], int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila)
        for (col = 0; col < n; ++col)
            *(c[fila]+col) = *(a[fila]+col) + *(b[fila]+col);
}
/* Escribe el resultado */
void escribematriz(int *c[MFIL], int m, int n) {
    int fila, col;
    for (fila = 0; fila < m; ++fila) {
        for (col = 0; col < n; ++col)
            printf ("  c[%d][%d]= %4d",fila + 1, col + 1,*(c[fila]+col));
```

```
        printf("\n");
    }
}
```

4.9 Punteros a funciones

Cuando una declaración de función aparece dentro de otra el nombre de la función declarada se convierte en un puntero a esa función. Así, si la función proceso se declara dentro de `main` entonces proceso se interpretará como una variable puntero dentro de `main`. Tales punteros pueden pasarse a otras funciones como argumentos.

Esto tiene el efecto de pasarle una función a otra, como si la primera función fuera una variable. La primera función se puede acceder dentro de la segunda. Pueden hacerse llamadas sucesivas a una función pasándole diferentes punteros (diferentes funciones). Cuando una función acepta el nombre de otra como argumento, la declaración formal debe identificar este argumento como un puntero a otra función. En su forma mas simple, un argumento formal que es un puntero a función puede declararse como,

```
<tipo-dato>  (*nbre-fcion)();
```

donde `tipo-dato` es el tipo del valor de retorno de la función apuntada. Se puede acceder a esta función a través del operador indirección. Para hacer esto, el operador indirección debe preceder al nombre de la función. El operador indirección y el nombre de la función deben estar encerrados entre paréntesis, esto es, (llamada de la función)

```
(*nbre-fcion) (argumento 1, argumento 2, ..., argumento n);
```

Para aclarar los conceptos observar el siguiente programa,

Este programa consta de cuatro funciones: `main`, `proceso`, `func1`, `func2`. Cada una de las tres funciones retorna un entero.

```
void main(void) {
    int i,j;
    int proceso(int(*)(int,int))
    int func1(int,int);
    int func2(int,int);
    ...
    i=proceso(func1); /* se pasa func1 a proceso; retorna un valor para i */
    ...
    j=proceso(func2); /* se pasa func2 a proceso; retorna un valor para i */
    ...
}
```

```
/* definicion de funcion */
int proceso(int (*pf)(int,int))  {
    int a,b,c;
    ...
    c=(*pf) (a,b); /* acceso a la funcion pasada a esta funcion;
                      retorna un valor para c */
    ...
    return(c);
}
/* definicion de funcion */
int func1(int a, int b) {
    int c;
    ...
    c = ...   /* usar a y b para evaluar c */
    ...
    return(c);
}
/* definicion de funcion */
int func2(int x, int y) {
    int z;
    ...
    z = ...   /* usar x e y para evaluar z */
    ...
    return(z);
}
```

Notar que `main` llama a la función `proceso` dos veces. En la primera pasa `func1` a `proceso`, mientras que en la segunda pasa `func2`. Cada llamada retorna un valor entero. Estos valores son asignados a las variables enteras `i` y `j`, respectivamente. Examinemos la definición de `proceso`. Esta función tiene un parámetro formal, `pf`, declarado como un puntero a función. La función a la que apunta `pf` devuelve un valor entero. Dentro de proceso se llama a la función a la que apunta `pf`. Se pasan dos cantidades enteras (`a` y `b`) como argumentos a la función. La función devuelve un entero que es asignado a la variable `c`. El resto de las definiciones de variables son normales (`func1` y `func2`). Estas son las funciones pasadas desde `main` hasta `proceso`. Los valores resultantes se asume que se obtienen de los argumentos, aunque los detalles de los cálculos no están especificados. Cada función presumiblemente procesa sus argumentos de entrada de modo diferente.

4.10 Mas sobre declaración de punteros

Las declaraciones de punteros pueden volverse complicadas y es necesario un cierto cuidado en su interpretación. Esto es especialmente cierto con declaraciones que involucren funciones o arreglos. Una dificultad es el uso dual de los paréntesis. Los paréntesis se usan para indicar funciones y anidaciones dentro de declaraciones mas complejas. Así la declaración,

```
int *p (int a);
```

indica una función que acepta un argumento entero y devuelve un puntero a entero. Por otra parte, la declaración,

```
int (*p)(int a);
```

indica un puntero a función que acepta un argumento entero y retorna un entero. En esta última declaración, los primeros paréntesis se usan para anidar y los segundos para indicar una función. La interpretación de declaraciones mas complejas puede hacerse molesta. Por ejemplo, considerar la siguiente declaración,

```
int *(*p)(int (*a)[]);
```

En esta declaración, `(*p)(...)` indica un puntero a función. Por esto, `int (*p)(...)` indica un puntero a función que retorna un entero. Dentro de los dos últimos paréntesis (la especificación de los argumentos de la función), `(*a)[]` indica un puntero a un arreglo. Como resultado, `int (*a)[]` representa un puntero a un arreglo de enteros. Uniendo las piezas, `(*p)(int (*a)[])` representa un puntero a una función cuyo argumento es un puntero a un arreglo de enteros. Finalmente la declaración completa,

```
int *(*p)(int (*a)[]);
```

representa un puntero a una función que acepta un puntero a un arreglo de enteros como argumentos y devuelve un puntero a entero.

Capítulo 5

Estructuras y Uniones

5.1 Introducción

En capítulos anteriores se estudiaron los arreglos, que son estructuras de datos cuyos elementos son todos del mismo tipo. Ahora, estudiaremos la estructura, que es una colección de datos cuyos elementos individuales pueden ser de distinto tipo. Así una estructura puede contener elementos enteros, flotantes, caracteres, etc. Además, punteros, arreglos y otras estructuras pueden incluirse como elementos dentro de una estructura. Estrechamente relacionada con la estructura está la unión que también contiene múltiples miembros pero a diferencia de la estructura, los miembros de una unión comparten el área de almacenamiento, incluso cuando los miembros son de diferentes tipo.

5.2 Definición de una estructura

La declaración de estructuras es algo mas complicada que la declaración de arreglos, ya que una estructura debe definirse en términos de sus miembros individuales. En general, la composición de una estructura se puede definir como,

```
struct marca {
    miembro 1;
    miembro 2;
    .......
    miembro m;
};
```

En esta declaración `struct` es la palabra clave requerida, `marca` es un nombre que identifica a la estructura de este tipo (estructuras que tengan esa composición) y `miembro 1`, `miembro 2`,..., `miembro m`, la declaración de los miembros individuales. (Nota: No existe distinción formal entre definición y declaración de estructuras, los términos son intercambiables).

Los miembros individuales de la estructura pueden ser variables ordinarias, punteros, arreglos u otras estructuras. Los nombres de los miembros dentro de una estructura particular deben ser todos diferentes, pero un miembro puede tener el mismo nombre que una variable definida fuera de la estructura. No se puede asignar un tipo de almacenamiento a un miembro individual, y tampoco puede inicializarse dentro de la declaración del tipo de la estructura. Una vez que la composición de la estructura ha sido definida, las variables individuales de ese tipo de estructura pueden declararse como sigue:

```
<tipo-almacenamiento> struct marca variable 1, variable 2, ..., variable n;
```

donde `tipo-almacenamiento` es un especificador opcional de tipo de almacenamiento, `struct` la palabra clave requerida, `marca` el nombre que aparece en la declaración de tipo de estructura y `variable 1`, `variable 2`, ..., `variable n` variables de estructura del tipo marca.

Por ejemplo,

```
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
};
```

Esta estructura se llama `legajo` (la marca es legajo) y contiene cuatro miembros, dos enteros y dos tipo char uno de los cuales es un arreglo.

Definida la estructura, se pueden ahora declarar variables de estructura como sigue:

```
struct legajo alumno, profesor;
```

Así `alumno` y `profesor` son variables tipo legajo cuya composición se identifica mediante la marca `legajo`.

Es posible combinar la declaración de la estructura con las variables de ese tipo como se muestra a continuación:

```
<tipo-almacenamiento> struct marca {
    miembro 1;
    miembro 2;
    .....
    miembro m;
} variable 1, variable 2, ..., variable n;
```

En esta situación marca es opcional. El siguiente ejemplo aclara lo dicho,

```
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
} alumno, profesor;
```

En el caso que la marca no se coloque la declaración toma la forma:

```
struct {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
} alumno, profesor;
```

Se puede definir una estructura como miembro de otra estructura. En este caso la declaración de la estructura interna debe aparecer antes que la declaración de la estructura externa.

Por ejemplo,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
} alumno, profesor;
```

A los miembros de una variable de tipo estructurado se le pueden asignar valores iniciales de la siguiente forma:

```
<tipo-almacenamiento> struct marca variable = {valor 1, valor 2, ..., valor m};
```

donde `valor 1` se refiere al valor del primer miembro, `valor 2` al valor del segundo, y así sucesivamente. El siguiente ejemplo ilustra la inicialización de una variable estructura,

```
struct fecha {
```

```
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
};
struct legajo alumno = {5436,"Juan P\'{e}rez","D",11923456,28,09,67};
```

También es posible definir un arreglo de estructuras, esto es un arreglo en el que cada elemento sea una estructura. En el siguiente ejemplo se muestra este caso,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
};
struct legajo alumno[100];
```

En esta declaración, alumno es un arreglo de 100 estructuras. Cada elemento de alumno es una estructura de tipo legajo. Se puede inicializar arreglo de estructuras de la misma manera que los arreglos, esto es,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
```

```
    struct fecha fecha_nac;
};
struct legajo alumno[] =    {
    {45,"Juan P\'{e}rez","D",10926432},
    {38,"Jos\'{e} Fern\'{a}ndez","I",11876543}
};
```

5.3 Procesamiento de los miembros de una estructura

Generalmente los miembros de una estructura se procesan en forma individual como entidades separadas. Por lo tanto para referirse a algún miembro en particular se procede de la siguiente manera:

```
variable.miembro
```

Donde variable se refiere al nombre de una variable de tipo estructura y miembro es el nombre de un miembro de la estructura. El operador "`.`" (punto) separa el nombre de la variable del nombre del miembro. Es un operador de máxima prioridad y su asociatividad es de izquierda a derecha.

Considerar las siguientes declaraciones de estructuras:

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
};
struct legajo alumno;
```

Para hacer referencia al tipo de documento de alumno se deberá escribir,

```
alumno.tipo_doc
```

Como el operador "." pertenece a un grupo de mayor precedencia, el mismo tendrá mayor precedencia que los operadores monarios, aritméticos, relacionales, lógicos y de asignación. En este caso la expresión `++variable.miembro` es equivalente a `++(variable.miembro)`, el operador `++`

se aplicará al miembro de la estructura y no a la estructura completa. A continuación se brindan algunos ejemplos de operaciones con estructuras.

Expresión	Interpretación
`++alumno.numero`	incrementa el valor de número.
`alumno.numero++`	incrementa el valor de número después de acceder a su valor.
`--alumno.numero`	decrementa el valor de número antes de acceder a su valor.
`&alumno`	accede a la dirección del primer miembro de la estructura.
`&alumno.tipo_doc`	accede a la dirección del miembro tipo de documento.

Se pueden escribir expresiones más complejas usando repetidamente el operador ".". Por ejemplo, si un miembro de una estructura es a su vez otra estructura, entonces se puede acceder al miembro de la estructura más interna de la siguiente forma:

```
variable.miembro.submiembro
```

donde miembro se refiere al nombre del miembro de la estructura externa y submiembro al de la estructura interna. En el ejemplo que estamos analizando si se quiere hacer referencia al año de nacimiento del alumno deberá escribirse,

```
alumno.fecha_nac.anio
```

Similarmente, si un miembro de una estructura es un arreglo, entonces se puede acceder a un miembro individual del arreglo de la siguiente forma,

```
variable.miembro[expresi\'{o}n]
```

donde expresión es un valor entero no negativo que indica el elemento del arreglo. El uso del operador punto puede extenderse a arreglos de estructuras escribiendo,

```
nombre_arreglo[expresi\'{o}n].miembro
```

donde `nombre_arreglo` se refiere al nombre del arreglo y `nombre_arreglo[expresion]` es un elemento individual del arreglo. Por lo tanto `nombre_arreglo[expresion].miembro` se referirá a un miembro especifico dentro de una estructura particular (un elemento del arreglo).

Considere el siguiente ejemplo,

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

```
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
};
struct legajo alumno[100];
```

En este ejemplo alumno es un arreglo que puede contener hasta 100 elementos. Cada elemento es una estructura de tipo `legajo`. Así si se desea acceder al tipo de documento del alumno 14 se debe escribir,

```
alumno[13].tipo_doc.
```

Los miembros de una estructura pueden procesarse de la misma manera que las variables ordinarias. Pueden aparecer en expresiones, funciones, etc.

En algunas versiones de C las estructuras debían ser procesadas miembro a miembro. Con esta restricción la única operación posible con la estructura completa era tomar su dirección. La versiones actuales (característica del ANSI C) permiten asignar una estructura completa a otra siempre y cuando tengan la misma composición. Por ejemplo, dadas las siguientes definiciones de tipos estructurados,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
}alumno[100],docente[40];
```

En la mayoría de las nuevas versiones de C es posible copiar los valores de una estructura en otra simplemente escribiendo,

```
docente[32] = alumno[87];
```

Por otra parte, algunas de las versiones mas antiguas de C requieren que los valores sean copiados individualmente, miembro a miembro, por ejemplo,

```
docente[19].fecha_nac.mes = alumno[3].fecha_nac.mes;
```

Por ejemplo: Supongamos se desea ingresar los datos relativos a un curso de alumnos y generar un listado por pantalla de esta información,

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define N 50
struct CURSO {
    int legajo;
    char nombre[40];
    char tip_doc[4];
    char doc[12];
};
void main(void) {
    int i,n;
    char aux[40];
    struct CURSO alumno[N];
    do {
        printf("Ingrese cantidad de alumnos [%d]",N);
        scanf("%d",&n);
    }while(n > N);
    for(i=0; i<n ; i++) {
        printf("\nLegajo:");
        scanf("%d",&alumno[i].legajo);
        printf("Apellido y Nombre Principal: ");
        gets(aux);
        strcpy(alumno[i].nombre,aux);
        strcat(alumno[i].nombre," ");
        gets(aux);
        strcat(alumno[i].nombre,aux);
        printf("Tipo de Documento [LE,LC,DNI,EX]: ");
        scanf("%s",alumno[i].tip_doc);
        printf("Documento: ");
        scanf("%s",alumno[i].doc);
    }
    getch();
    clrscr();
    for(i=0; i<n ; i++) {
        printf("\n\nLegajo: %d\n",alumno[i].legajo);
        printf("Apellido y Nombre Principal: %s\n",alumno[i].nombre);
        printf("Documento: %s %s",alumno[i].tip_doc,alumno[i].doc);
    }
}
```

5.4 Tipos de datos definidos por el usuario (typedef)

La característica `typedef` permite a los usuarios definir nuevos tipos de datos que sean equivalentes a los ya existentes. Una vez que el tipo de dato definido por el usuario ha sido establecido, entonces las nuevas variables, arreglos, estructuras, etc, se pueden declarar en términos de este nuevo tipo de dato. En términos generales, un nuevo tipo de dato se define como,

```
typedef tipo_viejo tipo_nuevo;
```

donde `tipo_viejo` se refiere a un tipo de dato existente (estándar o previamente definido por el usuario) y `tipo_nuevo` es el nuevo tipo definido por el usuario. Sin embargo debe quedar claro que el nuevo tipo será nuevo sólo en el nombre. En realidad, este nuevo tipo no será fundamentalmente diferente de los tipos de datos estándar. Los siguientes ejemplos aclaran los conceptos,

```
typedef int edad;
edad varon, mujer;
```

En este caso varón y mujer son del tipo edad equivalente a un tipo `int`. Las declaraciones,

```
typedef float altura[100];
altura varon, mujer;
```

definen altura como un arreglo de 100 elementos flotantes. De aquí que `varon` y `mujer` son ambos arreglos de 100 elementos de tipo flotante. Otra forma de expresar esto es,

```
typedef float altura;
altura varon[100], mujer[100];
```

La característica `typedef` es útil cuando se definen estructuras, ya que elimina la necesidad de escribir repetidamente `struct marca` en cualquier lugar que una estructura sea referenciada. En términos generales, el tipo de estructura definida por el usuario se puede escribir como,

```
typedef struct {
    miembro 1;
    miembro 2;
    ...
    miembro m;
}  tipo_nuevo;
```

donde `tipo_nuevo` es el tipo de estructura definida por el usuario. Las variables de estructura se pueden definir en términos del nuevo tipo de dato. Las siguientes declaraciones son comparables a las declaraciones de estructuras presentadas en ejemplos anteriores,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
typedef struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
}registro; // aqui registro es un nuevo tipo de datos

registro alumno[100];
```

En esta declaración se define a registro como un tipo de dato definido por el usuario. Luego la variable `alumno[100]` se define como variable de estructura del tipo registro.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#define N 50
typedef struct {
    int dia;
    int mes;
    int anio;
} FECHA;
typedef struct {
    int legajo;
    char nombre[40];
    char tip_doc[4];
    char doc[12];
    FECHA nac;
} CURSO;
void main(void) {
    int i,n;
    char aux[40];
    char temp[4];
    CURSO alumno[N];
    do{
        printf("Ingrese cantidad de alumnos [ %d]: ",N);
        scanf("%d",&n);
    }while(n > N);
    //Carga de datos
```

```
    for(i=0; i<n ; i++) {
        printf("\nLegajo: ");
        scanf("%d",&alumno[i].legajo);
        printf(""Apellido y Nombre Principal: ");
        gets(aux);
        strcpy(alumno[i].nombre,aux);
        strcat(alumno[i].nombre," ");
        gets(aux);
        strcat(alumno[i].nombre,aux);
        printf("Tipo de Documento [LE,LC,DNI,EX]: ");
        scanf("%s",alumno[i].tip_doc);
        printf("Documento: ");
        scanf("%s", alumno[i].doc);
        printf("Ingrese la fecha de nacimiento [dd/mm/aa]: ");
        scanf("%s", aux);
        temp[0]=aux[0];
        temp[1]=aux[1];
        temp[2] ='\0';
        alumno[i].nac.dia=atoi(temp);
        temp[0]=aux[3];
        temp[1]=aux[4];
        temp[2] ='\0';
        alumno[i].nac.mes=atoi(temp);
        temp[0]=aux[6];
        temp[1]=aux[7];
        temp[2] ='\0';
        alumno[i].nac.anio=atoi(temp);
        }
    getch();
    clrscr();
    //Impresion de resultados
    for(i=0; i<n ; i++) {
        printf("\n\nLegajo: %d\n",alumno[i].legajo);
        printf("Apellido y Nombre Principal: %s\n",alumno[i].nombre);
        printf("Documento: %s %s\n",alumno[i].tip_doc,alumno[i].doc);
        printf("Fecha de Nac: %d/%d/%d", alumno[i].nac.dia, alumno[i].nac.mes,
                alumno[i].nac.anio);
    }
}
```

5.5 Estructuras y Punteros

El nombre de una estructura identifica a una variable de ese tipo, a diferencia de los arreglos, el nombre de una variable de tipo estructurado no representa una dirección.

Se puede acceder entonces a la dirección de comienzo de una estructura de la misma manera que se accede a la dirección de cualquier otra variable, mediante el uso del operador dirección `&`. Así, si nombre es una variable de tipo estructura, entonces `&nombre` representa la dirección de comienzo de esa variable. Además se puede declarar una variable puntero a una estructura escribiendo,

```
tipo *ptvar;
```

donde `tipo` es el tipo de dato que identifica la composición de la estructura y `ptvar` representa el nombre de la variable puntero. Se puede asignar la dirección de comienzo de la estructura a este puntero escribiendo

```
ptvar = &variable;
```

Veamos el siguiente ejemplo,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
typedef struct {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
}legajo;

legajo alumno, *pc;
```

En este caso alumno es una variable estructurada de tipo `legajo` y `pc` un puntero que apunta a una variable estructura del tipo `legajo`. Así la dirección de comienzo de alumno se puede asignar a `pc` de la siguiente forma,

```
pc = &alumno;
```

Las declaraciones de la variable y del puntero se pueden combinar con la declaración de la estructura escribiendo,

```
typedef struct {
    miembro 1;
    miembro 2;
```

```
    ...
    miembro m;
} variable, *ptvar;
```

También, se puede acceder a un miembro de la estructura en términos de su correspondiente variable puntero escribiendo,

```
ptvar->miembro
```

donde el operador `->` es comparable al operador punto. Así la expresión `ptvar->miembro` es equivalente a escribir `variable.miembro`. El operador `->` pertenece al grupo de mayor precedencia, como el operador punto. Su asociatividad es de izquierda a derecha. Este operador puede combinarse con el operador punto para acceder a un submiembro dentro de una estructura. De aquí que un submiembro se pueda acceder escribiendo,

```
ptvar->miembro.submiembro;
```

De la misma manera, el operador `->` se puede usar para acceder a un elemento de un arreglo que es miembro de una estructura. Esto se realiza escribiendo,

```
ptvar->miembro[expresi\'{o}n];
```

donde expresión es un entero no negativo que indica el elemento del arreglo.

El siguiente ejemplo ilustra el uso de punteros a estructuras,

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct legajo {
    int c\'{o}digo_nro;
    char nombre[30];
    char tipo_doc;
    int numero;
    struct fecha fecha_nac;
} alumno, *pc = &alumno;
```

Notar que la variable puntero `pc` es inicializada asignándole la dirección de comienzo de la variable de estructura alumno. En otras palabras, `pc` apuntará a `alumno`. Si se desea acceder al número de documento del alumno, se puede escribir cualquiera de las siguientes expresiones:

```
alumno.numero       pc->numero      (*pc).numero
```

En la última expresión se necesitan los paréntesis porque el operador punto tiene mayor precedencia que el operador indirección (*). Sin los paréntesis el compilador generará un error, porque pc (un puntero) no es directamente compatible con el operador punto. De igual manera, se puede acceder al mes de la fecha de nacimiento del alumno con cualquiera de las siguientes expresiones,

```
alumno.fecha_nac.mes    pc>fecha_nac.mes    (*pc).fecha_nac.mes
```

Se puede acceder al tercer caracter del nombre del alumno escribiendo cualquiera de las siguientes expresiones,

```
alumno.nombre[2]        pc->nombre[2]       (*pc).nombre[2]
*(alumno.nombre + 2)    pc->(nombre + 2)    *((*pc).nombre +2)
```

Una estructura puede incluir también uno o más punteros como miembros. Así, si ptmiembro es a la vez un puntero y un miembro de variable, entonces *variable.ptmiembro accederá al valor al que apunta ptmiembro. De igual manera, si ptvar es una variable puntero que apunta a una estructura y ptmiembro es un miembro de esa estructura, entonces *ptvar->ptmiembro accederá al valor al que apunta ptmiembro. Esto puede observarse en el siguiente programa,

```
#include <stdio.h>
struct fecha {
    int dia;
    int mes;
    int anio;
};
typedef struct {
    int *c\'{o}digo_nro;
    char *nombre;
    char *tipo_doc;
    int *numero;
    struct fecha fecha_nac;
} CURSO;
void main(void) {
    int n =1234;
    char t = 'C';
    int d = 28;
    int m = 9;
    int a = 53;
    CURSO alumno, *pc = &alumno;
    alumno.c\'{o}digo_nro = &n;
    alumno.tipo_doc = &t;
```

```
    alumno.nombre = "Juan P\'{e}rez";
    *alumno.numero = 50;
    alumno.fecha_nac.dia = d;
    alumno.fecha_nac.mes = m;
    alumno.fecha_nac.anio = a;
    printf("%d %s %c %d %d-%d-%d\n", *pc->c\'{o}digo_nro, alumno.nombre,*pc->tipo_doc,
            *pc->numero, pc->fecha_nac.dia, pc->fecha_nac.mes, pc->fecha_nac.anio);
    printf("%d %s %c %d %d-%d-%d \n",*alumno.c\'{o}digo_nro,alumno.nombre,
            *alumno.tipo_doc,*alumno.numero,alumno.fecha_nac.dia,
            alumno.fecha_nac.mes, alumno.fecha_nac.anio);
}
```

La ejecución de este programa dará como resultado:

```
1234 Juan P\'{e}rez C 50 28-9-53
1234 Juan P\'{e}rez C 50 28-9-53
```

Como era de esperar las dos líneas son idénticas.

Como el operador (`->`) es miembro del grupo de mayor prioridad, tendrá la misma prioridad que el operador (.), con asociatividad de izquierda a derecha. Además, este operador, como el operador punto, tendrá prioridad sobre los operadores monarios, aritméticos, relacionales, lógicos o de asignación que puedan aparecer en una expresión. Sin embargo, se deben hacer ciertas consideraciones a algunos operadores monarios, como ++, cuando se aplican a variables puntero a estructura. Ya se sabe que expresiones como `++ptvar->miembro` y `++ptvar->miembro.submiembro` son equivalentes a `++(ptvar->miembro)` y `++(ptvar->miembro.submiembro)` respectivamente. Así tales expresiones producirán un incremento del valor del miembro o del submiembro. Por otra parte, la expresión `++ptvar` producirá el incremento del valor de `ptvar` en el número de bytes asociado con la estructura a la que apunta. De aquí que la dirección representada por ptvar cambiará como resultado de esta expresión. Similarmente, la expresión `(++ptvar).miembro` producirá que el valor de `ptvar` sea incrementado en ese número de bytes antes de acceder al miembro. Hay algún peligro en realizar operaciones como ésta porque `ptvar` es posible que ya no apunte a una variable estructura ya que su valor ha sido alterado.

5.6 Paso de estructuras a una función

Existen varias maneras de pasar información de una estructura a, o desde una función. Pueden transferirse miembros individuales o bien la estructura completa. El mecanismo para realizar la transferencia varía, dependiendo del tipo de transferencia y de la versión particular de C. Los miembros individuales de una estructura se pueden pasar a una función como argumentos en la llamada a la función, y un miembro de una estructura puede ser devuelto mediante la sentencia `return`. Para ello cada miembro de la estructura se trata como una variable ordinaria. El siguiente programa ilustra el pasaje de miembros de una estructura a una función.

```
struct fecha {
    int dia;
    int mes;
    int anio;
};
struct cuenta {
    int cuen_no;
    char cuen_tipo;
    char nombre[80];
    float saldo;
    struct fecha ultimopago;
} ;

void main(void) {
    struct cuenta cliente;
    float ajustar (char nombre[], int cuen_no, float saldo);
    ......
    cliente.saldo = ajustar (cliente.nombre, cliente.cuen_no, cliente.saldo);
    ......
}
float ajustar (char nombre[], int cuen_no, float saldo) {
    float nuevosaldo;   /*declaraci\'{o}n de variable local*/
    .......
    nuevosaldo = ... ;
    .....
    return(nuevosaldo);
}
```

Este ejemplo muestra como se pueden transferir miembros de una estructura a y desde una función. Una estructura completa se puede transferir a una función pasando un puntero a la estructura como argumento. En principio es similar al procedimiento utilizado para pasar un arreglo. Sin embargo, se debe utilizar notación explícita para representar que una estructura se pasa como argumento. Debe quedar claro que una estructura pasada de esta manera será pasada por referencia en vez de por valor. De aquí que, si cualquiera de los miembros de la estructura es alterado dentro de la función, las alteraciones serán reconocidas fuera de la función. De nuevo vemos una analogía directa con la transferencia de arreglo a funciones.

Considere el siguiente ejemplo,

```
#include <stdio.h>
typedef struct {
    char *nombre;
    int cuen_no;
    int cuen_tipo;
    float saldo;
```

```
} registro;
void main(void) {
    void ajustar (registro *pt); /* Declaraci\'{o}n de la funci\'{o}n */
    static registro cliente = {"Juan P\'{e}rez", 1234, 'C', 1234.54};
    printf  ("%s %d %c %.2f\n",cliente.nombre,cliente.cuen_no,
              cliente.cuen_tipo, cliente.saldo);
    ajustar(&cliente);
    printf  ("%s %d %c %.2f\n",cliente.nombre,cliente.cuen_no,
              cliente.cuen_tipo, cliente.saldo);
}

void ajustar(registro *pt) {
    pt->nombre = "Pedro P\'{e}rez";
    pt->cuen_no = 546;
    pt->cuen_tipo = 'D';
    pt->saldo = 4567.9;
    return;
}
```

Este programa ilustra la transferencia de una estructura a una función pasando la dirección de la estructura (un puntero) a la función.

Un puntero a una estructura puede ser devuelta de una función a la porción llamante del programa. Esta característica puede ser útil cuando se pasen varias estructuras a una función y solo una de ellas es devuelta.

La mayoría de las nuevas versiones de C permiten que una estructura completa sea transferida directamente a la función como un argumento y devuelta directamente por la sentencia return. En este caso la transferencia es por valor y no por referencia. Por lo tanto si cualquiera de los miembros de la estructura es alterado dentro de la función, las alteraciones no serán reconocidas fuera de la función. Sin embargo, si la estructura modificada es retornada a la parte llamante del programa entonces los cambios serán reconocidos. El siguiente programa ilustra esto,

```
#include <stdio.h>
typedef struct {
    char *nombre;
    int cuen_no;
    int cuen_tipo;
    float saldo;
} registro;

void main(void) {
    void ajustar (registro cliente);
    static registro cliente = {"Juan P\'{e}rez",1234,'C',1234.54};
    printf("%s %d %c %.2f\n",cliente.nombre,cliente.cuen_no,
            cliente.cuen_tipo, cliente.saldo);
```

```
    ajustar (cliente);
    printf("%s %d %c %.2f\n",cliente.nombre,cliente.cuen_no,
            cliente.cuen_tipo, cliente.saldo);
}

void ajustar (registro clien) {
    clien.nombre = "Pedro P\'{e}rez";
    clien.cuen_no = 546;
    clien.cuen_tipo = 'D';
    clien.saldo = 4567.9;
    return;
}
```

En este caso el resultado del programa es dos líneas con los mismos datos. Por lo tanto las nuevas asignaciones hechas dentro de la función no son reconocidas dentro de main. Suponga ahora que modificamos el programa de modo que la estructura alterada se devuelve de la función a main.

```
#include <stdio.h>
typedef struct {
    char *nombre;
    int cuen_no;
    int cuen_tipo;
    float saldo;
} registro;

void main(void) {
    registro ajustar (registro cliente);
    registro cliente = {"Juan P\'{e}rez",1234,'C',1234.54};
    printf("%s %d %c %.2f\n",cliente.nombre,cliente.cuen_no,
            cliente.cuen_tipo, cliente.saldo);
    cliente = ajustar(cliente);
    printf("%s %d %c %.2f\n",cliente.nombre,cliente.cuen_no,
            cliente.cuen_tipo, cliente.saldo);
}

registro ajustar (registro clien) {
    clien.nombre = "Pedro P\'{e}rez";
    clien.cuen_no = 546;
    clien.cuen_tipo = 'D';
    clien.saldo = 4567.9;
    return(clien);
}
```

5.7 Uniones

Las uniones como las estructuras contienen miembros cuyos tipos de datos pueden ser diferentes. Sin embargo, los miembros que componen la unión comparten dentro de la memoria el área de almacenamiento, mientras que cada miembro dentro de un tipo de datos estructurado tienen asignadas su propia áreas de almacenamiento.

Dentro de una union, la reserva de memoria requerida para almacenar miembros cuyos tipos de datos son diferentes (teniendo diferentes requerimientos de memoria) es manejado automáticamente por el compilador. Sin embargo, el usuario debe tener presente cual es el tipo de información que esta almacenado en cada momento. Una tentativa de acceso al tipo de información equivocado producirá resultados sin sentido. En términos generales, la composición de la union puede definirse como:

```
union marca {
    miembro 1;
    miembro 2;
    .....
    miembro m;
};
```

`union` es la palabra clave requerida mientras que los otros términos poseen el mismo significado que en el caso de la estructura. Las variables de tipo union pueden ser definidas como,

```
<tipo-almacenamiento> union marca variable 1, ...,variable n;
```

donde `tipo-almacenamiento` es un especificador opcional del tipo de almacenamiento. `union` la palabra clave requerida, `marca` el nombre que aparece en la definición de la unión y `variable 1`, ..., `variable n` variables unión del tipo marca. Las dos declaraciones pueden combinarse como se hizo con estructuras, escribiendo,

```
<tipo-almacenamiento> union marca {
    miembro 1;
    miembro 2;
    .....
    miembro m;
} variable 1, ..., variable n;
```

La marca es opcional en este tipo de declaración. Por ejemplo,

```
union ejem {
    char color[12];
    int talla;
} camisa, blusa;
```

En este ejemplo se tienen dos variables union (camisa y blusa) que son del tipo ejem. Cada variable puede representar una cadena de 12 caracteres (color) o un entero (talla) en un momento dado. La cadena de caracteres requerirá más área de almacenamiento dentro de la memoria de la máquina que el entero y por lo tanto se reserva un bloque de memoria suficiente para almacenar la cadena de 12 caracteres para cada variable unión, siendo el compilador el encargado de distinguir automáticamente entre el arreglo y el entero.

Una unión puede ser miembro de una estructura y una estructura puede ser miembro de una unión. Además, las estructuras y las uniones pueden ser mezcladas libremente con arreglos.

Un miembro individual de una unión se identifica de la misma manera que en la estructura. Para aclarar los conceptos vistos se muestra el siguiente ejemplo,

```
#include <stdio.h>
uni\'{o}n desc {
    char color;
    int talla;
};

struct ropa {
    char fabricante[20];
    float costo;
    uni\'{o}n desc descripci\'{o}n;
};

void main(void) {
    struct ropa camisa, blusa;
    printf("%d\n",sizeof(uni\'{o}n desc));
    camisa.descripci\'{o}n.color = 'B';
    printf("%c %d\n",camisa.descripci\'{o}n.color, camisa.descripci\'{o}n.talla);
    camisa.descripci\'{o}n.talla = 12;
    printf("%c %d\n",camisa.descripci\'{o}n.color, camisa.descripci\'{o}n.talla);
}
```

Ejecutando el programa se produce la siguiente salida,

```
 2
 B 27986
 * 42
```

La primera línea muestra la longitud en bytes de la unión. La segunda muestra dos datos. El primero es el único que tiene sentido ya que se le ha asignado el color B. El segundo dato no tiene sentido. Lo mismo ocurre en el tercer renglón donde ahora el dato válido es 42 (la talla) y no el color. Del ejemplo se desprende que las uniones deben manejarse con mucho cuidado.

Capítulo 6

Archivos de datos

6.1 Introducción

En computación un archivo es un conjunto de datos con información relacionada, la información puede ser de distinto tipo. Así un archivo puede contener una secuencia de sentencias de un determinado lenguaje de programación, o puede consistir en valores de datos, el primer archivo es un programa y el segundo se denomina archivo de datos.

Hasta el momento, las aplicaciones realizadas trataban estructuras de datos que se encontraban almacenadas en la memoria principal de la computadora, pero esto no siempre es lo más conveniente por distintas razones, entre las que podemos citar:

- La cantidad de datos a procesar por el programa es lo suficientemente grande para que no entren en la memoria principal.
- Es común que los datos que salen de un programa deban ser almacenados para poder ser tratados como entradas de otro.
- La necesidad de que la información ingresada a la computadora sea mantenida en algún medio de almacenamiento masivo para posteriores procesamientos.

En el lenguaje de programación C existe un conjunto extenso de funciones de biblioteca que permiten el manejo de archivos de datos, a lo largo de este capítulo estudiaremos la más comunes.

El lenguaje de programación C permite procesar los archivos secuenciales y de acceso directo (acceso aleatorio) en forma indistinta, como lo hacen otros lenguajes de programación. Pero existen dos tipos distintos de archivos de datos, los llamados archivos secuenciales de datos, también llamados estándar o de texto y los archivos de bajo nivel o binarios. Generalmente es más fácil trabajar con archivos de datos secuenciales.

Existen, cuatro operaciones básicas con archivos, siempre al menos debemos trabajar con tres de ellas.

Estas operaciones son:

- Abrir de un archivo: Se debe abrir un archivo para preparar el sistema de hardware para que acepte nueva información o para que pueda transferir información desde el disco hasta el programa que lo solicita. Es como abrir un canal de comunicaciones entre el disco y la CPU. Además es necesario que nuestro sistema sepa donde debe guardar la nueva información y desde donde debe leer la información solicitada. Si bien esto es necesario hacer, cabe mencionar que mientras mas tiempo un archivo permanezca abierto mas problemas podemos tener con él. Por lo tanto debemos mantener abierto un archivo el menor tiempo posible.
- Cerrar un Archivo: Así como se abre un archivo, también es necesario cerrarlo luego de trabajar con él. Es como eliminar el vínculo o canal de comunicaciones que se estableció en el momento de la apertura.
- Leer información desde un archivo: Permite extraer información desde el disco, existen muchas formas de hacerlo, dependiendo del tipo de información requerida y de la función utilizada.
- Escribir información en un archivo: Permite enviar información al disco, existen muchas formas de hacerlo, dependiendo del tipo de información a escribir y de la función utilizada.

6.1.1 Abrir un archivo

El primer paso a realizar cuando se trabaja con archivos, es establecer un área, un buffer de entrada/salida, también llamada memoria intermedia de entrada/salida. En esta zona es donde se almacena temporalmente la información mientras se produce la transferencia entre la memoria del computador y el archivo de datos. La utilización de esta memoria intermedia permite que la información se pueda leer o escribir en el archivo con mucha mayor velocidad de lo que se tendría sin su uso.

El encargado de realizar el enlace entre el sistema de memoria intermedia de E/S y el archivo es el puntero al archivo. Un puntero a un archivo, es un puntero mediante el cual se puede acceder a una estructura que contiene toda la información necesaria para poder trabajar con el archivo, como lo es el nombre, estado, la dirección del buffer, la operación a realizar, etc. Los usuarios no necesitan conocer l os detalles de esta estructura, ya que está definida en el archivo de cabecera stdio.h. Un puntero de archivo es una variable de tipo FILE (mayúsculas), que esta definido también en stdio.h. Para que un programa pueda leer o escribir en un archivo, necesita utilizar un puntero al archivo, por lo tanto debe utilizar una sentencia como la siguiente:

```
FILE *ptvar; //define un puntero a archivo
```

Esta última sentencia nos dice que `ptvar` es un puntero a `FILE`, donde `FILE` es el nombre de un tipo de datos, como lo es int o float, el cual ha sido definido con un typedef, y como se especifico anteriormente, establece un área del buffer de E/S. El puntero `ptvar` indicará el principio de ese área, a este puntero comúnmente se lo denomina como puntero a archivo secuencial, o simplemente como archivo secuencial.

Un archivo de datos debe ser abierto antes de ser creado o de que se realice cualquier operación en él. La función `fopen` es la que se utiliza para abrir un archivo, al realizar esta operación se asocia el nombre del archivo con el área del buffer correspondiente, también se debe especificar el tipo de operación que se desea realizar con él, por ejemplo de solo lectura, de solo escritura, de lectura/escritura, etc. El formato de la función `fopen` es el siguiente:

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

Los dos argumentos de la función son cadenas de caracteres, el primero debe especificar el nombre del archivo, el cual debe ser consistente con las reglas establecidas por el sistema operativo para nombrar archivos, permitiéndose la especificación de caminos (tener en cuenta que en este último caso la especificación de caminos se debe hacer según las reglas de construcción de strings del lenguaje C, por ejemplo "`c:\\s\\miarchiv.dat`", la doble barra indica la secuencia de escape para la barra). El segundo argumento modo, también es una cadena de caracteres, indica el uso que se va a hacer del archivo, los modos permitidos se muestran en la siguiente tabla.

Modo	Operación a realizar
"r"	Abrir un archivo de texto solo para lectura.
"w"	Abrir un nuevo archivo de texto solo para escritura. Si existe un archivo con el mismo nombre, será destruido, y creado otro en su lugar.
"a"	Abre un archivo de texto existente para agregar datos. Si no existe uno con el nombre especificado se crea uno nuevo.
"rb"	Abrir un archivo binario solo para lectura.
"wb"	Abrir un nuevo archivo binario solo para escritura. Si existe un archivo con el mismo nombre, será destruido, y creado otro en su lugar.
"ab"	Abre un archivo binario existente para agregar datos. Si no existe uno con el nombre especificado se crea uno nuevo.
"r+"	Abrir un archivo de texto para lectura y escritura.
"w+"	Abrir un nuevo archivo de texto para lectura y escritura. Si existe un archivo con el mismo nombre, será destruido, y creado otro en su lugar.
"a+"	Abre un archivo de texto existente para agregar datos y leer. Si no existe uno con el nombre especificado se crea uno nuevo.

La función `fopen` retorna un puntero al principio del área del buffer asociada al archivo. Debe tenerse la precaución de no alterar el valor de dicho puntero en el programa. Existen algunas situaciones de error como por ejemplo: intentar leer un archivo inexistente, intentar leer un archivo sin autorización o un disco lleno o protegido, en este caso `fopen` devuelve un puntero nulo `NULL` (definido también en stdio.h)

Para abrir un archivo para escritura con el nombre prueba en un programa, se escribiría:

```
FILE *fp; fp = fopen("prueba","w");
```

Sin embargo, usualmente se verá el código escrito de esta forma:

```
FILE *fp; if (( fp = fopen ("prueba","w")) == NULL ) {
    puts (" ERROR NO SE PUEDE ABRIR EL ARCHIVO \n);
    exit (1);
}
```

Este método detecta cualquier error producido al abrir un archivo, mediante la macro `NULL`. Se utiliza un nulo porque ningún puntero podrá tener ese valor.

La función `exit` utilizada, es una función de biblioteca que permite abandonar el programa y devolver el control al sistema operativo, cuando durante su ejecución lo considere necesario. Como ejemplo de esto, si un programa necesita para su ejecución la presencia de una determinada tarjeta gráfica o de una impresora y en un chequeo no las encuentra presentes en el sistema de cómputo, la función `exit()` abortara la ejecución del mismo, como lo hace en el ejemplo al detectarse un error al abrir el archivo "prueba", considerar que esta función se encuentra en la biblioteca stdlib.h.

6.1.2 Cerrar un archivo

Un archivo de datos, debe cerrarse al final del programa. Esto se realiza mediante la función de biblioteca `fclose`. La sintaxis es simplemente:

```
fclose(ptvar);
```

Donde el argumento `ptvar` es el puntero de archivo que fue devuelto en cuando se invocó a `fopen`. La función `fclose` retorna un entero, cuando el valor de retorno es cero, indica que la operación de cierre se ha realizado sin problemas; cualquier otro valor indica un error. Generalmente en los únicos casos en que `fclose` fallará son: cuando se retira anticipadamente un diskette de la unidad o cuando no hay mas espacio en el.

Es una buena práctica en computación cerrar explícitamente los archivos mediante la función `fclose`, aunque la mayoría de los compiladores de lenguaje C cerrarán automáticamente los archivos de datos al final de la ejecución del programa, si no esta presente una llamada a `fclose`.

La función `fclose` escribe en el archivo los datos que todavía quedan en el buffer del disco y realiza un cierre formal a nivel sistema operativo en el archivo. El no cierre de un archivo produce serios problemas, incluyendo pérdida de datos, archivos destruidos, etc. Como se sabe el sistema operativo limita el número de archivos que se pueden tener abiertos en forma simultanea, por lo tanto muchas veces es necesario cerrar un archivo antes de abrir otro.

6.1.3 Escribir y leer un caracter de un archivo

La función que se utiliza para escribir un caracter en un archivo, el cual fue previamente abierto para escritura con la función `fopen`, es `putc()`; cuyo formato es el siguiente:

```
int putc(ch,fp);
```

Donde el argumento `fp` es el puntero de archivo devuelto por `fopen`, el cual le indica a `putc` cual es el archivo de disco que se va a escribir, `ch` es el caracter a escribir. La función `putc` devuelve el caracter escrito si la operación se realizo con éxito y `EOF` si se produce un error, recordar que `EOF` es una macro definida en stdio.h que indica fin de archivo.

Para leer un caracter desde un archivo que ha sido abierto para lectura mediante `fopen`, se utiliza la función `getc`, cuya sintaxis es la siguiente:

```
ch = getc(fp);
```

`fp` es un puntero a archivo de tipo `FILE` devuelto por `fopen`, La función `getc` devuelve el caracter siguiente del archivo referenciado por `fp`, y entrega `EOF` cuando se alcanza el fin de archivo.

6.1.4 Uso de las funciones fopen, getc, putc y fclose

Estas funciones constituyen el conjunto mínimo de funciones para el manejo de archivos. Mediante dos ejemplos se desarrollara la utilización de estas funciones ya descriptas en los párrafos anteriores.

En el primer ejemplo, se lee un texto caracter a caracter desde teclado, cada caracter es convertido a mayúsculas y luego almacenado en el archivo `muestra.dat`.

```
// Lee una linea de texto en minusculas y la almacena en mayusculas en un archivo
#include <stdio.h>
#include <stdlib.h>
void main(void) {
    FILE *fpt;  /* define ftp como puntero a estructura FILE */
    char c, d;
    /* abre un archivo nuevo de datos solo para escritura */
    if((fpt = fopen ("muestra.dat","w"))==NULL) {
        puts("Error al abrir el archivo\n");
        exit(1);
    }
    /* leer cada caracter y escribe su equivalente mayusculas en el archivo */
    do {
        c = getchar();
        d = toupper(c);
        putc(d,fp);
    } while (c != '\n');
    /* cierra el archivo de datos*/
    fclose(fpt);
}
```

El programa comienza definiendo el puntero a archivo secuencial `fpt`, que indicará el comienzo del área del buffer del archivo.

Se abre un nuevo archivo `muestra.dat`, sólo para escritura. A continuación el lazo `do-while` lee una serie de caracteres del teclado y escribe las mayúsculas equivalentes en el archivo. La función `putc` se utiliza para escribir cada caracter en el archivo, para lo cual se requiere que el puntero de archivo ftp se especifique como argumento. El lazo continúa hasta que se introduce desde teclado, el caracter de nueva línea '`\n`', detectado este se sale del lazo cerrándose el archivo.

Un archivo creado de esta manera puede visualizarse de distintas formas. Por ejemplo, usando alguna orden del sistema operativo como print o type. También se puede visualizar mediante un editor o un procesador de textos. Otra posibilidad, para visualizarlo es la de crear un programa que lea el contenido del archivo y lo muestre, realizando una tarea inversa a la del programa anterior, es el caso del segundo ejemplo de aplicación que se muestra a continuación.

El programa lee una línea de texto de un archivo de datos, caracter a caracter, mostrándolo en pantalla.

```
/* Lee una linea de texto de un archivo y lo muestra en pantalla */
#include <stdio.h>
#include <stdlib.h>
void main(void) {
    FILE *fpt;  /* define ftp como puntero a estructura FILE */
    char c;
    /* abre un archivo nuevo de datos solo para lectura */
    if ((fpt = fopen ("muestra.dat","r")) == NULL) {
        printf("\n ERROR - No se puede abrir el archivo indicado\n");
        exit(1);
    }
    /* lee y muestra c/u de los caracteres del archivo */
    do {
        c = getc(fpt);
        putchar (c);
    } while (c != '\n');
    /* cierra el archivo de datos*/
    fclose(fpt);
}
```

6.1.5 Las funciones getw(), putw(), fgets() y fputs()

Además de `getc()` y `putc()`, algunos compiladores C, como el Turbo C, soportan funciones similares aplicables a otro tipo de datos.

Las funciones `putw()` y `getw()` permiten leer y escribir enteros (2 bytes o 1 Word) en y desde un archivo de datos. Estas funciones trabajan exactamente igual que las ya descritas `getc()` y `putc()`, y sus definiciones están también en stdio.h. Los prototipos de función de `putw` y `getw`, son los siguientes

```
int putw (int i, FILE *fp); int getw (FILE *fp);
```

El lenguaje de programación C provee además dos funciones que permiten leer y escribir cadenas de caracteres desde los archivos, estas son las funciones `fgets` y `fputs`.

La función `fputs()` funciona como `puts()`, salvo que `fputs()` escribe la cadena de caracteres en archivo indicado por el puntero, si la acción se realiza en forma correcta, la función entrega el último caracter escrito, si ocurre algún problema retorna `EOF`.

La función `fgets()` lee una cadena de caracteres desde el archivo especificado por el puntero de archivo, hasta que lee un `\n`, o cuando el contador de caracteres llega a -1.

El formato de ambas funciones es el siguiente:

```
int  fputs( cadena,FILE *fp); char * fgets(cadena, contador, FILE
*fp);
```

Ejercicio propuesto: Realizar las modificaciones en los dos programas de los ejemplos anteriores para que en vez de transferir caracteres entre archivo y consola lo realicen con enteros y con cadenas de caracteres.

6.1.6 Las funciones fread() y fwrite()

Estas dos funciones provistas en la biblioteca, permiten leer y escribir desde y en los archivos bloques de datos. El formato general de sus declaraciones son:

```
unsigned fread(void *memoria,int num_de_bytes,int cont,FILE *fp);
unsigned fwrite(void *memoria,int num_de bytes,int cont,FILE *fp);
```

En el caso de `fread()`, el argumento memoria es un puntero a la zona de memoria que recibirá los datos leídos desde el archivo. Para `fwrite()`, memoria es un puntero a la información que se escribirá en el archivo. Para ambos, el argumento `num_de_bytes` especifica el número de bytes de cada uno de los bloques a transferir. El parámetro `cont` en ambas funciones, determina cuantos elementos o bloques, cada uno de `num_de_bytes` de longitud, se leerán o escribirán. El último argumento `fp` es un puntero de archivo el cual ha sido previamente abierto mediante la función `fopen`, como el archivo es abierto para datos binarios, tanto `fread()` como `fwrite()`, pueden procesar información de cualquier tipo. Ambas funciones están definidas en stdio.h.

Además esta funciones se utilizan conjuntamente con la función `feof()`, la cual permite averiguar si se alcanzo o no el final de un archivo. El formato general de esta función es:

```
int feof(FILE *pt)
```

Esta función retorna un valor entero distinto de 0 si se alcanzo el final del archivo, en caso contrario retorna 0. El argumento es un puntero al archivo con el que se esta operando.

El siguiente ejemplo describe el uso de la función `fwrite()`, para escribir un dato real (float) en un archivo de datos en disco.

```
/* Escribe un numero en punto flotante en un archivo de disco */
#include <stdio.h>
void main(void) {
    FILE *fp;
    float f = 145.87;
    if ((fp = fopen ("prueba","wb")) == NULL) {
        printf (" NO se puede abrir el archivo \n");
        exit(1);
    }
    fwrite (&f, sizeof(float),1,fp);
    fclose(fp);
}
```

Como se observa en el programa, el puntero a memoria que requiere la función `fwrite()`, en este caso, no es más que la dirección de una variable simple. Aplicaciones más útiles de `fwrite()` y `fread()`, son evidentemente la lectura y escritura de arreglos y estructuras en archivos. En el siguiente ejemplo, se crea un arreglo muestra de datos reales, y se escribe en un archivo del mismo nombre utilizando `fwrite()`.

```
// Este ejemplo transfiere los datos en coma flotante
// contenidos en un arreglo a un archivo
#include <stdio.h>
void main(void) {
    FILE *fp;
    float muestra [100];
    int i;
    if(( fp = fopen ("muestra","wb")) == NULL) {
        printf (" NO se puede abrir el archivo \n" );
        exit (1);
    }
    for ( i=0; i < 100 ; i++) {
        /* asigna a muestra [i] el valor de i en flotante */
        muestra [i] = (float) i;
    }
    /* graba todo el archivo entero en un solo paso */
    fwrite (muestra, sizeof(muestra),1,fp);
    fclose (fp);
}
```

Una aplicación de la función `fread()` sería leer el archivo muestra, generado en el ejemplo anterior y mostrarlo por pantalla, tarea que es realizada por el siguiente programa.

```
#include <stdio.h>
void main(void) {
```

```
    FILE *fp;
    float sample [100];
    int i;
    if (( fp = fopen ("muestra", "rb")) == NULL) {
        printf ("no se puede abrir el archivo");
        exit (1);
    }
    /* lee el archivo entero en un solo paso */
    fread(sample, sizeof (sample),1,fp);
    /* Muestra los elementos del arreglo uno por uno*/
    for (i=0;i<100;i++) {
        printf("%f\t",sample[i]);
    }
    fclose (fp);
}
```

Se propone, que en base a estos dos ejemplos se realicen programas similares para arreglos de cadena de caracteres, y para estructuras.

6.1.7 Entrada/Salida de acceso directo

Es posible realizar las operaciones de lectura y escritura directas de un dato en un archivo, mediante la utilización de la función `fseek()`, la cual permite localizar una determinada posición en el archivo. El formato general de la declaración de `fseek()` es:

```
int fseek(FILE *fp, long num_bytes, int origen);
```

El argumento `fp` es el puntero a archivo que se retorna de la invocación a `fopen()`; `num_bytes`, es un entero largo que especifica la cantidad de bytes a desplazarse desde el "origen" para obtener la nueva posición, este desplazamiento puede ser positivo o negativo, el signo indicara el sentido del desplazamiento. El argumento origen, establece una referencia para ubicar una determinada posición, y se representa por alguno de los siguientes macros definidos en stdio.h:

Nombre de la Macro	Valor real	Descripción
`SEEK_SET`	0	Comienzo del archivo
`SEEK_CUR`	1	Posición actual
`SEEK_END`	2	Fin del archivo

La función `fseek()` retorna el valor 0 si el proceso se ha realizado normalmente y -1 si ocurrió algún error.

El siguiente, es un fragmento de un programa, en el cual se lee el caracter ubicado en la posición 235 del archivo test.

```
void main(void) {
    ...
    FILE *fp;
    char ch;
    if ((fp = fopen("test", "rb")) == NULL) {
        printf ("no puede abrirse el archivo\n");
        exit(1);
    }
    fseek (fp,234L,SEEK_SET);
    ch = getc (fp); /* Lee el caracter de la posici\'{o}n 235 */
    ...
}
```

En la invocación de la función `fseek()`, al especificar el numero de bytes del desplazamiento, se le agrega el modificador "L" a la constante 234, para forzar al compilador a tratar a la constante como de tipo long (Recordar que si se usa un entero común, cuando la computadora espera un entero largo, provocará errores). Cabe acotar que la necesidad del uso de entero largo es para permitir el manejo de archivos de más de 64K bytes.

6.1.8 Los canales de entrada salida estándar

Siempre que un programa en lenguaje C comienza su ejecución, la computadora abre cinco canales de entrada/salida , también llamadas corrientes de E/S (streams), que en definitiva son punteros a archivos de tipo `FILE`. Los cuales son: Entrada estándar (`stdin`), salida estándar (`stdout`), salida de error estándar (`stderr`), salida a impresora (`stdprn`) y salida auxiliar (`stdaux`). Los primeros tres canales se refieren normalmente a la consola, salvo que el sistema operativo haya realizado algún redireccionamiento.

Estos punteros de archivos, definidos en stdio.h, pueden utilizarse en cualquier función donde se necesite un puntero a archivo de tipo `FILE`, se debe tener en cuenta que son constantes, no variables, por lo tanto no se debe alterar su valor.

Un ejemplo muy simple es el siguiente en el cual mediante la utilización de la función `putc()`, se implementa la función `putchar()`.

```
putchar (char c) {
    putc (c,stdout);
}
```

Otro ejemplo de utilización de los canales de E/S estándar lo muestra la siguiente sentencia, en la cual se utiliza la función `fputs()` para mandar una cadena de caracteres a la impresora.

```
fputs ("ESTO ES UNA PRUEBA \n", stdprn );
```

Así como la computadora crea estos punteros de archivos al comienzo del programa de usuario, los cierra al final, por lo tanto no deben cerrarse.

6.1.9 Las funciones fprintf() y fscanf()

Muchos archivos de datos contienen estructuras de datos más complejas, donde la información esta compuesta por caracteres y números. Para procesar estos archivos es conveniente utilizar las funciones de biblioteca específicas para manejo archivos de datos con formato, como lo son `fscanf()` y `fprintf()`. Estas funciones se comportan exactamente igual que las ya estudiadas `scanf()` y `printf()`. Así la función `fscanf()` permite leer uno o varios datos con formato desde un archivo especificado, y `fprintf()` permite escribir datos con formato en un archivo.

El formato general de estas funciones es el siguiente:

```
int fprintf (FILE * fp, cadena de control,lista de variables);
int fscanf (FILE * fp, cadena de control ,lista de variables);
```

Donde `fp` es el puntero de archivo, que se obtiene con la invocación de la función `fopen()`, los argumentos cadena de control y lista de variables, deben cumplir las mismas reglas que se utilizaban para las funciones `scanf()` y `printf()`. La función `fprintf()` entrega un valor entero, el cual indica la cantidad de bytes que fueron almacenados en el archivo. El valor devuelto por `fscanf()` indica la cantidad de campos que fueron debidamente leídos del archivo y almacenados en memoria.

Para mostrar la utilización de estas funciones, se presenta el siguiente ejemplo, se trata de un programa que maneja una sencilla guía telefónica en disco. Permite introducir nombres y números y se puede buscar un número correspondiente a un nombre especificado.

```
/* Un sencillo directorio de telefono */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
void agrega (void);
void mirar (void);
int menu(void);

void main(void) {
    char elecc;
    do{
        elecc = menu();
        switch (elecc) {
            case 'a':
                agrega();
```

```
                break;
            case 'm':
                mirar ();
                break;
        }
    } while(elecc != 's');
}

/* Funcion menu */
/* Muestra menu en pantalla y espera una eleccion. */
void menu (void) {
    char ch;
    do {
        clrscr();
        printf(" Ingrese una opcion \n\n");
        printf(" A  Agregar \n");
        printf(" M  Mirar  \n");
        printf(" S  Salir \n");
        ch = tolower (getche());
    }while(ch != 's' && ch != 'a' && ch != 'm');
    return ch;
}

/* Agrega un nombre y un numero */
void agrega (void) {
    FILE *fp;
    char name[80], num[15];
    int code;
    /* Abre archivo para agregar */
    if ((fp = fopen ("telefono","a")) == NULL) {
        printf(" No se puede abrir el archivo telefono");
        exit (1);
    }
    printf (" Introduzca nombre:  ");
    gets(name);
    printf ("\n\n Ingrese codigo:  ");
    scanf ("%d",&code);
    fscanf(stdin,"%c");  /* descarga CR del canal de entrada estandar stdin */
    printf ("\n\n Ingrese numero: ");
    gets(num);
    /* escribe en el archivo */
    fprintf (fp,"%s %d %s\n", name,code,num);
    fclose (fp);
}
```

```
/* Encuentra un numero dando el nombre */
void mirar (void) {
    FILE *fp;
    char name[80], name2[80], num[15];
    int code;
    /* abre el archivo para leer */
    if (( fp = fopen ("telefono","r")) == NULL) {
        printf (" No se puede abrir el archivo telefono ");
        exit (1);
    }
    clrscr();
    printf (" Ingrese el nombre \n");
    gets(name2);
    /* mira el numero */
    while (!feof(fp)){ /* feof devuelve 0 si no encuentra EOF */
        /* lee el archivo y determina si se asignaron los tres campos */
        if ((fscanf (fp,"%s %d %s", name,&code,num)) == 3) {
            if (! strcmp (name,name2)) {
                printf("%s  \t%03d\t %s \n", name,code,num);
            }
        }
    }
    getch();
    fclose (fp);
}
```

6.1.10 Borrado de archivos

La función `remove()`, permite borrar un archivo especificado como argumento. El formato general de esta función es:

```
int remove (nombre_archivo);
```

Si la acción se ejecuta en forma correcta, la función devuelve cero; si no, entrega un valor distinto de cero.

6.1.11 Las funciones ferror y rewind

La función `ferror()`, determina si en una operación con archivos se ha producido algún error, el formato general de esta función es:

```
int ferror (FILE *fp);
```

Donde el argumento `fp` es un puntero a archivo de tipo `FILE`. La función `ferror()` devuelve cero si la operación con archivos se ha realizado sin error.

La función `rewind()` coloca el puntero del archivo al comienzo del mismo. El formato de esta función es el siguiente.

```
void rewind (FILE *fp);
```

Las definiciones de ambas funciones están en stdio.h.