CAPITULO 1: INTRODUCCION

1. INTRODUCCION

El lenguaje de programación C está caracterizado por ser de uso general, con una sintaxis sumamente compacta y de alta portabilidad.
Es común leer que se lo caracteriza como un lenguaje de "bajo nivel". No debe confundirse el término "bajo" con "poco", ya que el significado del mismo es en realidad "profundo", en el sentido que C maneja los elementos básicos presentes en todas las computadoras: caracteres, números y direcciones.
Esta particularidad, junto con el hecho de no poseer operaciones de entrada-salida, manejo de arreglo de caracteres, de asignación de memoria, etc , puede al principio parecer un grave defecto; sin embargo el hecho de que estas operaciones se realicen por medio de llamadas a Funciones contenidas en Librerías externas al lenguaje en sí, es el que confiere al mismo su alto grado de portabilidad, independizándolo del "Hardware" sobre el cual corren los programas, como se irá viendo a lo largo de los siguientes capítulos.
La descripción del lenguaje se realiza siguiendo las normas del ANSI C, por lo tanto, todo lo expresado será utilizable con cualquier compilador que se adopte; sin embargo en algunos casos particulares se utilizaron funciones Compilador ó Sistema Operativo-dependientes, explicitándose en estos casos la singularidad de las mismas.

2. ANATOMIA DE UN PROGRAMA C

Siguiendo la tradición, la mejor forma de aprender a programar en cualquier lenguaje es editar, compilar, corregir y ejecutar pequeños programas descriptivos. Analicemos por lo tanto el primer ejemplo :

EJEMPLO 1

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

{

        printf("Bienvenido a la Programación en lenguaje C \n");

        return 0;

}
```

FUNCION main()

Dejemos de lado por el momento el análisis de la primer linea del programa, y pasemos a la segunda.
La función main() indica donde empieza el programa, cuyo cuerpo principal es un conjunto de sentencias delimitadas por dos llaves, una inmediatamente después de la declaración main() " { ", y otra que finaliza el listado " } ". Todos los programas C arrancan del mismo punto: la primer sentencia dentro de dicha función, en este caso printf (".....").
En el EJEMPLO 1 el programa principal está compuesto por sólo dos sentencias: la primera es un llamado a una función denominada printf(), y la segunda, return, que finaliza el programa retornando al Sistema Operativo.
Recuérdese que el lenguaje C no tiene operadores de entrada-salida por lo que para escribir en video es necesario llamar a una función externa. En este caso se invoca a la

función printf(argumento) existente en la Librería y a la cual se le envía como argumento aquellos caracteres que se desean escribir en la pantalla. Los mismos deben estar delimitados por comillas. La secuencia \n que aparece al final del mensaje es la notación que emplea C para el caracter "nueva línea" que hace avanzar al cursor a la posición extrema izquierda de la línea siguiente. Más adelante analizaremos otras secuencias de escape habituales.
La segunda sentencia (return 0) termina el programa y devuelve un valor al Sistema operativo, por lo general cero si la ejecución fué correcta y valores distintos de cero para indicar diversos errores que pudieron ocurrir. Si bien no es obligatorio terminar el programa con un return, es conveniente indicarle a quien lo haya invocado, sea el Sistema Operativo o algún otro programa, si la finalización ha sido exitosa, o no. De cualquier manera en este caso, si sacamos esa sentencia el programa correrá exactamente igual, pero al ser compilado, el compilador nos advertirá de la falta de retorno.
Cada sentencia de programa queda finalizada por el terminador "; ", el que indica al compilador el fin de la misma. Esto es necesario ya que, sentencias complejas pueden llegar a tener más de un renglón, y habrá que avisarle al compilador donde terminan. Es perfectamente lícito escribir cualquier sentencia abarcando los renglones que la misma necesite, por ejemplo podría ser:

```
printf("Bienvenido a la Programacion"

                    "en lenguaje C \n");
```

3. ENCABEZAMIENTO

Las líneas anteriores a la función main() se denominan ENCABEZAMIENTO (HEADER) y son informaciones que se le suministran al Compilador.
La primera línea del programa está compuesta por una directiva: " #include " que implica la orden de leer un archivo de texto especificado en el nombre que sigue a la misma (<stdio.h>) y reemplazar esta línea por el contenido de dicho archivo.
En este archivo están incluidas declaraciones de las funciones luego llamadas por el programa (por ejemplo printf()) necesarias para que el compilador las procese. Por ahora no nos preocupemos por el contenido del archivo ya que más adelante, en el capítulo de funciones, analizaremos exhaustivamente dichas declaraciones.
Hay dos formas distintas de invocar al archivo, a saber, si el archivo invocado está delimitado por comillas (por ejemplo "stdio.h") el compilador lo buscará en el directorio activo en el momento de compilar y si en cambio se lo delimita con los signos <.....> lo buscará en algun otro directorio, cuyo nombre habitualmente se le suministra en el momento de la instalación del compilador en el disco (por ejemplo C:\TC\INCLUDE). Por lo general estos archivos son guardados en un directorio llamado INCLUDE y el nombre de los mismos está terminado con la extensión .h.
La razón de la existencia de estos archivos es la de evitar la repetición de la escritura de largas definiciones en cada programa.
Nótese que la directiva "#include" no es una sentencia de programa sino una orden de que se copie literalmente un archivo de texto en el lugar en que ella está ubicada ,por lo que no es necesario terminarla con "; ".

4. COMENTARIOS

La inclusión de comentarios en un programa es una saludable práctica, como lo reconocerá cualquiera que haya tratado de leer un listado hecho por otro programador ó por sí mismo, varios meses atrás. Para el compilador, los comentarios son inexistentes, por lo que no generan lineas de código, permitiendo abundar en ellos tanto como se desee.

En el lenguaje C se toma como comentario todo caracter interno a los símbolos: /* */ .
Los comentarios pueden ocupar uno o más renglones, por ejemplo:
COMENTARIOS

```
/* este es un comentario corto */

/* este otro

        es mucho

        más largo

        que el anterior */
```

Todo caracter dentro de los símbolos delimitadores es tomado como comentario incluyendo a " * " ó " (" , etc.

CAPITULO 2: VARIABLES Y CONSTANTES

1. DEFINICION DE VARIABLES

Si yo deseara imprimir los resultados de multiplicar un número fijo por otro que adopta valores entre 0 y 9 , la forma normal de programar esto sería crear una CONSTANTE para el primer número y un par de VARIABLES para el segundo y para el resultado del producto. Una variable , en realidad , no es más que un nombre para identificar una (o varias) posiciones de memoria donde el programa guarda los distintos valores de una misma entidad . Un programa debe DEFINIR a todas las variables que utilizará , antes de comenzar a usarlas , a fin de indicarle al compilador de que tipo serán , y por lo tanto cuanta memoria debe destinar para albergar a cada una de ellas. Veamos el EJEMPLO 2:

EJEMPLO 2

```
#include <stdio.h>
main()
{
  int multiplicador;          /* defino multiplicador como un entero */
  int multiplicando;          /* defino multiplicando como un entero */
  int resultado;              /* defino resultado como un entero */
  multiplicador = 1000 ;      /* les asigno valores */
  multiplicando = 2 ;
  resultado = multiplicando * multiplicador ;
  printf("Resultado = %d\n", resultado);          /* muestro el resultado */
  return 0;
}
```

En las primeras líneas de texto dentro de main() defino mis variables como números enteros , es decir del tipo "int" seguido de un identificador (nombre) de la misma . Este identificador puede tener la cantidad de caracteres que se desee , sin embargo de acuerdo al Compilador que se use , este tomará como significantes sólo los primeros n de ellos ; siendo por lo general n igual a 32 . Es conveniente darle a los identificadores de las variables , nombres que tengan un significado que luego permita una fácil lectura del programa. Los identificadores deben comenzar con una letra ó con el símbolo de subrayado "_" , pudiendo continuar con cualquier otro caractér alfanumérico ó el símbolo "_" . El único símbolo no alfanumérico aceptado en un nombre es el "_" . El lenguaje C es sensible al tipo de letra usado ; así tomará como variables distintas a una llamada "variable" , de otra escrita como "VARIABLE". Es una convención entre los programadores de C escribir los nombres de las variables y las funciones con minúsculas, reservando las mayúsculas para las constantes.

El compilador dará como error de "Definición incorrecta" a la definición de variables con nombres del tipo de :

4pesos $variable primer-variable !variable etc.etc

NOTA: Los compiladores reservan determinados términos ó palabras claves (Keywords) para el uso sintáctico del lenguaje, tales como: asm, auto, break, case, char, do, for, etc. Si bien estas palabras están definidas para el ANSI C, los distintos compiladores extienden esta definición a OTROS términos, por lo que es aconsejable leer la tabla completa de palabras reservadas del compilador que se vaya a usar, para no utilizarlas en nombres de variables.

Vemos en las dos lineas subsiguientes a la definición de las variables, que puedo ya asignarles valores (1000 y 2) y luego efectuar el cálculo de la variable "resultado". Si prestamos ahora atención a la función printf(), ésta nos mostrará la forma de visualizar el valor de una variable. Insertada en el texto a mostrar, aparece una secuencia de control de impresión "%d" que indica, que en el lugar que ella ocupa, deberá ponerse el contenido de la variable (que aparece luego de cerradas las comillas que marcan la finalización del texto , y separada del mismo por una coma) expresado como un un número entero decimal. Así, si compilamos y corremos el programa, obtendremos una salida:

SALIDA DEL EJEMPLO 2

Resultado = 2000

---

2. INICIALIZACION DE VARIABLES

Las variables del mismo tipo pueden definirse mediante una definición múltiple separándolas mediante " , " a saber :
int multiplicador, multiplicando, resultado;
Esta sentencia es equivalente a las tres definiciones separadas en el ejemplo anterior.
Las variables pueden también ser inicializadas en el momento de definirse .
int multiplicador = 1000, multiplicando = 2, resultado;
De esta manera el EJEMPLO 2 podría escribirse:
EJEMPLO 2 BIS

```
#include <stdio.h>

main()

{

  int multiplicador=1000 , multiplicando=2 ;

  printf("Resultado = %d\n", multiplicando * multiplicador);

  return 0;

}
```

Obsérvese que en la primer sentencia se definen e inicializan simultaneamente ambas variables.La variable "resultado" la hemos hecho desaparecer ya que es innecesaria. Si analizamos la función printf() vemos que se ha reemplazado "resultado" por la operación entre las otras dos variables. Esta es una de las particularidades del lenguaje C : en los parámetros pasados a las funciones pueden ponerse operaciones (incluso llamadas a otras funciones) , las que se realizan ANTES de ejecutarse la función , pasando finalmente a esta el valor resultante de las mismas. El EJEMPLO 2 funciona exactamente igual que antes pero su código ahora es mucho más compacto y claro.

3. TIPOS DE VARIABLES

VARIABLES DEL TIPO ENTERO

En el ejemplo anterior definimos a las variables como enteros (int).
De acuerdo a la cantidad de bytes que reserve el compilador para este tipo de variable,

queda determinado el "alcance" ó máximo valor que puede adoptar la misma.
Debido a que el tipo int ocupa dos bytes su alcance queda restringido al rango entre -32.768 y +32.767 (incluyendo 0).
En caso de necesitar un rango más amplio, puede definirse la variable como "long int nombre_de_variable" ó en forma más abreviada "long nombre_de_variable"
Declarada de esta manera, nombre_de_variable puede alcanzar valores entre -2.347.483.648 y +2.347.483.647.
A la inversa, si se quisiera un alcance menor al de int, podría definirse "short int " ó simplemente "short", aunque por lo general, los compiladores modernos asignan a este tipo el mismo alcance que "int".
Debido a que la norma ANSI C no establece taxativamente la cantidad de bytes que ocupa cada tipo de variable, sino tan sólo que un "long" no ocupe menos memoria que un "int" y este no ocupe menos que un "short",los alcances de los mismos pueden variar de compilador en compilador , por lo que sugerimos que confirme los valores dados en este parágrafo (correspondientes al compilador de Borland C++) con los otorgados por su compilador favorito.
Para variables de muy pequeño valor puede usarse el tipo "char" cuyo alcance está restringido a -128, +127 y por lo general ocupa un único byte.
Todos los tipos citados hasta ahora pueden alojar valores positivos ó negativos y, aunque es redundante, esto puede explicitarse agregando el calificador "signed" delante; por ejemplo:

signed int

signed long

signed long int

signed short

signed short int

signed char

Si en cambio, tenemos una variable que sólo puede adoptar valores positivos (como por ejemplo la edad de una persona) podemos aumentar el alcance de cualquiera de los tipos , restringiéndolos a que sólo representen valores sin signo por medio del calificador "unsigned" . En la TABLA 1 se resume los alcances de distintos tipos de variables enteras

TABLA 1 VARIABLES DEL TIPO NUMERO ENTERO

TIPO	BYTES	VALOR MINIMO	VALOR MAXIMO
signed char	1	-128	127
unsigned char	1	0	255
signed short	2	-32.768	+32.767
unsigned short	2	0	+65.535
signed int	2	-32.768	+32.767
unsigned int	2	0	+65.535
signed long	4	-2.147.483.648	+2.147.483.647
unsigned long	4	0	+4.294.967.295

NOTA: Si se omite el calificador delante del tipo de la variable entera, éste se adopta por omisión (default) como "signed".

VARIABLES DE NUMERO REAL O PUNTO FLOTANTE

Un número real ó de punto flotante es aquel que además de una parte entera, posee fracciones de la unidad. En nuestra convención numérica solemos escribirlos de la siguiente manera : 2,3456, lamentablemente los compiladores usan la convención del PUNTO decimal (en vez de la coma) . Así el numero Pi se escribirá : 3.14159 Otro formato de escritura, normalmente aceptado, es la notación científica. Por ejemplo podrá escribirse 2.345E+02, equivalente a 2.345 * 100 ó 234.5

De acuerdo a su alcance hay tres tipos de variables de punto flotante , las mismas están descriptas en la TABLA 2

TABLA 2 TIPOS DE VARIABLES DE PUNTO FLOTANTE

TIPO	BYTES	VALOR MINIMO	VALOR MAXIMO
float	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	3.4E+4932

Las variables de punto flotante son SIEMPRE con signo, y en el caso que el exponente sea positivo puede obviarse el signo del mismo.

5. CONVERSION AUTOMATICA DE TIPOS

Cuando dos ó mas tipos de variables distintas se encuentran DENTRO de una misma operación ó expresión matemática , ocurre una conversión automática del tipo de las variables. En todo momento de realizarse una operación se aplica la siguiente secuencia de reglas de conversión (previamente a la realización de dicha operación):

- 1) Las variables del tipo char ó short se convierten en int
- 2) Las variables del tipo float se convierten en double
- 3) Si alguno de los operandos es de mayor precisión que los demás , estos se convierten al tipo de aquel y el resultado es del mismo tipo.
- 4) Si no se aplica la regla anterior y un operando es del tipo unsigned el otro se convierte en unsigned y el resultado es de este tipo.

Las reglas 1 a 3 no presentan problemas, sólo nos dicen que previamente a realizar alguna operación las variables son promovidas a su instancia superior. Esto no implica que se haya cambiado la cantidad de memoria que las aloja en forma permanente

Otro tipo de regla se aplica para la conversión en las asignaciones.

Si definimos los términos de una asignación como,"lvalue" a la variable a la izquierda del signo igual y "rvalue" a la expresión a la derecha del mismo, es decir:

"lvalue" = "rvalue" ;

Posteriormente al cálculo del resultado de "rvalue" (de acuerdo con las reglas antes descriptas), el tipo de este se iguala al del "lvalue". El resultado no se verá afectado si el tipo de "lvalue" es igual ó superior al del "rvalue", en caso contrario se efectuará un truncamiento ó redondeo, segun sea el caso.

Por ejemplo, el pasaje de float a int provoca el truncamiento de la parte fraccionaria, en cambio de double a float se hace por redondeo.

5. ENCLAVAMIENTO DE CONVERSIONES (casting)

Las conversiones automáticas pueden ser controladas a gusto por el programador,

imponiendo el tipo de variable al resultado de una operación. Supongamos por ejemplo tener:

```
double  d , e , f = 2.33 ;

int i = 6 ;

e = f * i ;

d = (int) ( f * i ) ;
```

En la primer sentencia calculamos el valor del producto (f * i) , que según lo visto anteriormente nos dará un double de valor 13.98 , el que se ha asignado a e. Si en la variable d quisiéramos reservar sólo el valor entero de dicha operación bastará con anteponer, encerrado entre paréntesis, el tipo deseado. Así en d se almacenará el número 13.00.
También es factible aplicar la fijación de tipo a una variable, por ejemplo obtendremos el mismo resultado, si hacemos:

```
d = (int) f * i ;
```

En este caso hemos convertido a f en un entero (truncando sus decimales)

6. VARIABLES DE TIPO CARACTER

El lenguaje C guarda los caracteres como números de 8 bits de acuerdo a la norma ASCII extendida , que asigna a cada caracter un número comprendido entre 0 y 255 (un byte de 8 bits) Es común entonces que las variables que vayan a alojar caracteres sean definidas como:

```
char c ;
```

Sin embargo, también funciona de manera correcta definirla como

```
int c ;
```

Esta última opción desperdicia un poco más de memoria que la anterior ,pero en algunos casos particulares presenta ciertas ventajas . Pongamos por caso una función que lee un archivo de texto ubicado en un disco. Dicho archivo puede tener cualquier caracter ASCII de valor comprendido entre 0 y 255. Para que la función pueda avisarme que el archivo ha finalizado deberá enviar un número NO comprendido entre 0 y 255 (por lo general se usa el -1 , denominado EOF, fin de archivo ó End Of File), en este caso dicho número no puede ser mantenido en una variable del tipo char, ya que esta sólo puede guardar entre 0 y 255 si se la define unsigned ó no podría mantener los caracteres comprendidos entre 128 y 255 si se la define signed (ver TABLA 1). El problema se obvia facilmente definiéndola como int.
Las variables del tipo caractér también pueden ser inicializadas en su definición, por ejemplo es válido escribir:

```
char c = 97 ;
```

para que c contenga el valor ASCII de la letra "a", sin embargo esto resulta algo engorroso , ya que obliga a recordar dichos códigos . Existe una manera más directa de asignar un caractér a una variable ; la siguiente inicialización es idéntica a la anterior :

```
char c = 'a' ;
```

Es decir que si delimitamos un caracter con comilla simple , el compilador entenderá que debe suplantarlo por su correspondiente código numérico .
Lamentablemente existen una serie de caracteres que no son imprimibles , en otras palabras que cuando editemos nuestro programa fuente (archivo de texto) nos resultará difícil de asignarlas a una variable ya que el editor las toma como un COMANDO y no como un caracter . Un caso típico sería el de "nueva linea" ó ENTER .

Con el fin de tener acceso a los mismos es que aparecen ciertas secuencias de escape convencionales . Las mismas estan listadas en la TABLA 3 y su uso es idéntico al de los caracteres normales , asi para resolver el caso de una asignación de "nueva linea " se escribirá:

```
char c = '\n' ;          /* secuencia de escape */
```

TABLA 3 SECUENCIAS DE ESCAPE

CODIGO	SIGNIFICADO	VALOR ASCII (decimal)	VALOR ASCII (hexadecimal)
'\n'	nueva línea	10	0x0A
'\r'	retorno de carro	13	0x0D
'\f'	nueva página	2	x0C
'\t'	tabulador horizontal	9	0x09
'\b'	retroceso (backspace)	8	0x08
'\''	comilla simple	39	0x27
'\"'	comillas	4	0x22
'\\ '	barra	92	0x5C
'\? '	interrogación	63	0x3F
'\nnn'	cualquier caracter (donde nnn es el código ASCII expresado en octal)		
'\xnn'	cualquier caracter (donde nn es el código ASCII expresado en hexadecimal)		

7. TAMAÑO DE LAS VARIABLES (sizeof)

En muchos programas es necesario conocer el tamaño (cantidad de bytes) que ocupa una variable, por ejemplo en el caso de querer reservar memoria para un conjunto de ellas. Lamentablemente, como vimos anteriormente este tamaño es dependiente del compilador que se use, lo que producirá, si definimos rigidamente (con un número dado de bytes) el espacio requerido para almacenarlas, un problema serio si luego se quiere compilar el programa con un compilador distinto del original

Para salvar este problema y mantener la portabilidad, es conveniente que cada vez que haya que referirse al TAMAÑO en bytes de las variables, se lo haga mediante un operador llamado "sizeof" que calcula sus requerimientos de almacenaje

Está también permitido el uso de sizeof con un tipo de variable, es decir:

```
sizeof(int)

sizeof(char)

sizeof(long double) , etc.
```

8. DEFINICION DE NUEVOS TIPOS (typedef)

A veces resulta conveniente crear otros tipos de variables , ó redefinir con otro nombre las existentes , esto se puede realizar mediante la palabra clave "typedef" , por ejemplo:

```
typedef unsigned long double  enorme  ;
```

A partir de este momento ,las definiciones siguientes tienen idéntico significado:

```
unsigned long double nombre_de_variable ;

enorme nombre_de_variable ;
```

9. CONSTANTES

Aquellos valores que , una vez compilado el programa no pueden ser cambiados , como por ejemplo los valores literales que hemos usado hasta ahora en las inicializaciones de las variables (1000 , 2 , 'a' , '\n' , etc), suelen denominarse CONSTANTES .
Como dichas constantes son guardadas en memoria de la manera que al compilador le resulta más eficiente suelen aparecer ciertos efectos secundarios , a veces desconcertantes , ya que las mismas son afectadas por las reglas de RECONVERSION AUTOMATICA DE TIPO vista previamente.
A fin de tener control sobre el tipo de las constantes, se aplican la siguientes reglas :

- Una variable expresada como entera (sin parte decimal) es tomada como tal salvo que se la siga de las letras F ó L (mayúsculas ó minúsculas) ejemplos :
  1 : tomada como ENTERA
  1F : tomada como FLOAT
  1L : tomada como LONG DOUBLE
- Una variable con parte decimal es tomada siempre como DOUBLE, salvo que se la siga de la letra F ó L
  1.0 : tomada como DOUBLE
  1.0F : tomada como FLOAT
  1.0L : tomada como LONG FLOAT
- Si en cualquiera de los casos anteriores agregamos la letra U ó u la constante queda calificada como UNSIGNED (consiguiendo mayor alcance) :
  1u : tomada como UNSIGNED INT
  1.0UL : tomada como UNSIGNED LONG DOUBLE
- Una variable numérica que comienza con "0" es tomado como OCTAL asi : 012 equivale a 10 unidades decimales
- Una variable numérica que comienza con "0x" ó "0X" es tomada como hexadecimal : 0x16 equivale a 22 unidades decimales y 0x1A a 26 unidades decimales.

10. CONSTANTES SIMBOLICAS

Por lo general es una mala práctica de programación colocar en un programa constantes en forma literal (sobre todo si se usan varias veces en el mismo) ya que el texto se hace dificil de comprender y aún más de corregir, si se debe cambiar el valor de dichas constantes.
Se puede en cambio asignar un símbolo a cada constante, y reemplazarla a lo largo del programa por el mismo, de forma que este sea más legible y además, en caso de querer modificar el valor, bastará con cambiarlo en la asignación.
El compilador, en el momento de crear el ejecutable, reemplazará el símbolo por el valor asignado.
Para dar un símbolo a una constante bastará, en cualquier lugar del programa (previo a su uso) poner la directiva: #define, por ejemplo:

```
#define VALOR_CONSTANTE 342

#define PI 3.1416
```

CAPITULO 3: OPERADORES

1. INTRODUCCION

Si analizamos la sentencia siguiente:
var1 = var2 + var3;
estamos diciéndole al programa, por medio del operador +, que compute la suma del valor de dos variables , y una vez realizado ésto asigne el resultado a otra variable var1. Esta última operación (asignación) se indica mediante otro operador, el signo =.
El lenguaje C tiene una amplia variedad de operadores, y todos ellos caen dentro de 6 categorias , a saber : aritméticos , relacionales, lógicos, incremento y decremento, manejo de bits y asignacion. Todos ellos se irán describiendo en los párrafos subsiguientes.

2. OPERADORES ARITMETICOS

Tal como era de esperarse los operadores aritméticos ,mostrados en la TABLA 4 , comprenden las cuatro operaciones basicas , suma , resta , multiplicación y división , con un agregado , el operador módulo .

TABLA 4 OPERADORES ARITMETICOS

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVALUACION
+	SUMA	a + b	3
-	RESTA	a - b	3
*	MULTIPLICACION	a * b	2
/	DIVISION	a / b	2
%	MODULO	a % b	2
-	SIGNO	-a	2

El operador módulo (%) se utiliza para calcular el resto del cociente entre dos ENTEROS , y NO puede ser aplicado a variables del tipo float ó double .
Si bien la precedencia (orden en el que son ejecutados los operadores) se analizará más adelante, en este capítulo, podemos adelantar algo sobre el orden que se realizan las operaciones aritméticas.
En la TABLA 4, última columna, se da el orden de evaluación de un operador dado. Cuanto más bajo sea dicho número mayor será su prioridad de ejecución. Si en una operación existen varios operadores, primero se evaluarán los de multiplicación , división y módulo y luego los de suma y resta . La precedencia de los tres primeros es la misma , por lo que si hay varios de ellos, se comenzará a evaluar a aquel que quede más a la izquierda . Lo mismo ocurre con la suma y la resta .
Para evitar errores en los cálculos se pueden usar paréntesis , sin limitación de anidamiento, los que fuerzan a realizar primero las operaciones incluidas en ellos . Los paréntesis no disminuyen la velocidad a la que se ejecuta el programa sino que tan sólo obligan al compilador a realizar las operaciones en un orden dado dado, por lo que es una buena costumbre utilizarlos ampliamente .
Los paréntesis tienen un orden de precedencia 0, es decir que antes que nada se evalúa lo que ellos encierran .
Se puede observar que no existen operadores de potenciación, radicación, logaritmación, etc, ya que en el lenguaje C todas estas operaciones (y muchas otras) se realizan por medio de llamadas a Funciones.
El último de los operadores aritméticos es el de SIGNO . No debe confundirselo con el de resta, ya que este es un operador unitario que opera sobre una única variable cambiando el signo de su contenido númerico. Obviamente no existe el operador +

unitario, ya que su operación sería DEJAR el signo de la variable, lo que se consigue simplemente por omisión del signo.

3. OPERADORES RELACIONALES

Todas las operaciones relacionales dan sólo dos posibles resultados : VERDADERO ó FALSO . En el lenguaje C, Falso queda representado por un valor entero nulo (cero) y Verdadero por cualquier número distinto de cero En la TABLA 5 se encuentra la descripción de los mismos .

TABLA 5 OPERADORES RELACIONALES

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVALUACION
<	menor que	(a < b)	5
>	mayor que	(a >b)	5
< =	menor o igual que	(a < = b)	5
>=	mayor o igual que	(a >>= b)	5
= =	igual que	(a = = b)	6
! =	distinto que	(a != b)	6

Uno de los errores más comunes es confundir el operador relacional IGUAL QUE (= =) con el de asignacion IGUAL A (=). La expresión a=b copia el valor de b en a, mientras que a = = b retorna un cero , si a es distinto de b ó un número distinto de cero si son iguales.

Los operadores relacionales tiene menor precedencia que los aritméticos , de forma que a < b + c se interpreta como a < (b + c), pero aunque sea superfluo recomendamos el uso de paréntesis a fin de aumentar la legilibilidad del texto.

Cuando se comparan dos variables tipo char el resultado de la operación dependerá de la comparación de los valores ASCII de los caracteres contenidos en ellas. Asi el caracter a (ASCII 97) será mayor que el A (ASCII 65) ó que el 9 (ASCII 57).

4. OPERADORES LOGICOS

Hay tres operadores que realizan las conectividades lógicas Y (AND) , O (OR) y NEGACION (NOT) y están descriptos en la TABLA 6 .

TABLA 6 OPERADORES LOGICOS

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVALUACION
&&	Y (AND)	(a>b) && (c < d)	10
||	O (OR)	(a>b) || (c < d)	11
!	NEGACION (NOT)	!(a>b)	1

Los resultados de la operaciones lógicas siempre adoptan los valores CIERTO ó FALSO. La evaluación de las operaciones lógicas se realiza de izquierda a derecha y se interrumpe cuando se ha asegurado el resultado .

El operador NEGACION invierte el sentido lógico de las operaciones , así será

!(a >> b) equivale a (a < b)
!(a == b) " " (a != b)
etc.

En algunas operaciones suele usárselo de una manera que se presta a confusión , por ejemplo : (!i) donde i es un entero. Esto dará un resultado CIERTO si i tiene un valor 0 y un resultado FALSO si i es distinto de cero .

5. OPERADORES DE INCREMENTO Y DECREMENTO

Los operadores de incremento y decremento son sólo dos y están descriptos en la TABLA 7

TABLA 7 OPERADORES DE INCREMENTO Y DECREMENTO

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVALUACION
++	incremento	++i ó i++	1
--	decremento	--i ó i--	1

Para visualizar rapidamente la función de los operadores antedichos , digamos que las sentencias :

```
a = a + 1 ;
```

```
a++ ;
```

tienen una acción idéntica , de la misma forma que

```
a = a - 1 ;
```

```
a-- ;
```

es decir incrementa y decrementa a la variable en una unidad

Si bien estos operadores se suelen emplear con variables int , pueden ser usados sin problemas con cualquier otro tipo de variable . Así si a es un float de valor 1.05 , luego de hacer a++ adoptará el valor de 2.05 y de la misma manera si b es una variable del tipo char que contiene el caracter 'C' , luego de hacer b-- su valor será 'B' .

Si bien las sentencias

```
i++ ;
++i ;
```

son absolutamente equivalentes, en la mayoria de los casos la ubicación de los operadores incremento ó decremento indica CUANDO se realiza éste .

Veamos el siguiente ejemplo :

```
int i = 1 , j , k ;
j = i++ ;
k = ++i ;
```

acá j es igualado al valor de i y POSTERIORMENTE a la asignación i es incrementado por lo que j será igual a 1 e i igual a 2 , luego de ejecutada la sentencia . En la siguiente instrucción i se incrementa ANTES de efectuarse la asignacion tomando el valor de 3 , él que luego es copiado en k .

6. OPERADORES DE ASIGNACION

En principio puede resultar algo futil gastar papel en describir al operador IGUAL A (=) , sin embargo es necesario remarcar ciertas características del mismo .

Anteriormente definimos a una asignación como la copia del resultado de una expresión (rvalue) sobre otra (lvalue) , esto implica que dicho lvalue debe tener LUGAR (es decir poseer una posición de memoria) para alojar dicho valor .

Es por lo tanto válido escribir

```
a = 17 ;
```

pero no es aceptado , en cambio

```
17 = a ;      /* incorrecto */
```

ya que la constante numérica 17 no posee una ubicación de memoria donde alojar al valor de a .

Aunque parezca un poco extraño al principio las asignaciones , al igual que las otras operaciones , dan un resultado que puede asignarse a su vez a otra expresión .

De la misma forma que (a + b) es evaluada y su resultado puedo copiarlo en otra

variable : c = (a + b) ; una asignación (a = b) da como resultado el valor de b , por lo que es lícito escribir
c = (a = b) ;
Debido a que las asignaciones se evalúan de derecha a izquierda , los paréntesis son superfluos , y podrá escribirse entonces :
c = a = b = 17 ;
con lo que las tres variables resultarán iguales al valor de la contante .
El hecho de que estas operaciones se realicen de derecha a izquierda también permite realizar instrucciones del tipo :
a = a + 17 ;
significando esto que al valor que TENIA anteriormente a , se le suma la constante y LUEGO se copia el resultado en la variable .
Como este último tipo de operaciones es por demás común , existe en C un pseudocódigo , con el fin de abreviarlas .
Asi una operación arítmetica o de bit cualquiera (simbolizada por OP)
a = (a) OP (b) ;
puede escribirse en forma abreviada como :
a OP= b ;
Por ejemplo

```
a += b   ;    /* equivale :  a = a + b */
a -= b   ;    /* equivale :  a = a - b */
a *= b   ;    /* equivale :  a = a * b */
a /= b   ;    /* equivale :  a = a / b */
a %= b   ;    /* equivale :  a = a % b */
```

Nótese que el pseudooperador debe escribirse con los dos símbolos seguidos , por ejemplo += , y no será aceptado +(espacio) = .
Los operadores de asignación estan resumidos en la TABLA 8 .

TABLA 8 OPERADORES DE ASIGNACION

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVALUACION
=	igual a	a = b	13
op=	pseudocodigo	a += b	13
=?:	asig.condicional	a = (c>b)?d:e	12

Vemos de la tabla anterior que aparece otro operador denominado ASIGNACION CONDICIONAL . El significado del mismo es el siguiente :
lvalue = (operación relacional ó logica) ? (rvalue 1) : (rvalue 2) ;
de acuerdo al resultado de la operación condicional se asignará a lvalue el valor de rvalue 1 ó 2 . Si aquella es CIERTA será lvalue = rvalue 1 y si diera FALSO , lvalue = rvalue 2 .
Por ejemplo, si quisiéramos asignar a c el menor de los valores a ó b , bastará con escribir :
c = (a < b) ? a : b ;

7. OPERADORES DE MANEJO DE BITS

Estos operadores muestran una de las armas más potentes del lenguaje C , la de poder manipulear INTERNAMENTE , es decir bit a bit , las variables .
Debemos anticipar que estos operadores sólo se aplican a variables del tipo char , short , int y long y NO pueden ser usados con float ó double ,
Sabemos que las computadoras guardan los datos organizados en forma digital , en bytes , formado por números binarios de 8 bits y como se vió anteriormente cuando se analizó el tamaño de las variables , un char ocupará un byte de 8 bits , mientras que los

short e int se forman con dos bytes (16 bits) y los long por cuatro bytes (32 bits).
Para el manejo de dichos bits , contamos con los operadores descriptos en la TABLA 9 .

TABLA 9 OPERADORES DE MANEJO DE BITS

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVAL.
&	Y ó AND (bit a bit)	a & b	7
|	O ú OR INCLUSIVA	a | b	9
^	O ú OR EXCLUSIVA	a ^ b	8
<<	ROTACION A LA IZQUIER	a << b	4
>>	ROTACION A LA DERECHA	a >> b	4
~	COMPLEMENTO A UNO	~a	1

Describiremos mediante unos pocos ejemplos la operatoria de manejo de bits.
Analicemos primero como funciona el operador Y, también llamado BITWISE AND , las reglas para la operación son las dadas en la TABLA 10 .

TABLA 10 REGLAS PARA LA OPERACION Y (BITWISE AND)

bit a	&	bit b	=	bit c
0	&	0	=	0
0	&	1	=	0
1	&	0	=	0
1	&	1	=	1

Si suponemos tener dos variables del tipo char, una de ella de valor 85 (hex. 55), otra de valor 71 (hex. 47) y realizamos el AND a nivel bits de ellas, obtendremos :

```
      bits                  decimal hexadecimal
      0 1 0 1 0 1 0 1         85          55
      &                     &           &
      0 1 0 0 0 1 1 1         71          47
-------------------------     -------   -------
      0 1 0 0 0 1 0 1         69          45
```

Nótese que la operación es del tipo lógico entre bits, por lo que los resultados numéricos tienen poco ó ningún significado y sólo se han puesto con el fin de ejemplificar .
De la misma manera para la operacion O INCLUSIVA, cuyas reglas se dan en la TABLA 11, será:

TABLA 11 REGLAS PARA LA OPERACION O INCLUSIVA (BITWISE OR)

bit a	|	bit b	=	bit c
0	|	0	=	0
0	|	1	=	1
1	|	0	=	1
1	|	1	=	1

Para las mismas variables anteriores obtendremos :
0 1 0 1 0 1 1 1 87 57
Analizando ahora la O EXCLUSIVA (ó EXOR) tendremos :

TABLA 12 REGLAS PARA LA OPERACION O EXCLUSIVA (EXOR)

bit a	^	bit b	=	bit c
0	^	0	=	0
0	^	1	=	1
1	^	0	=	1
1	^	1	=	0

Para las mismas variables anteriores obtendremos :

0 0 0 1 0 0 1 0 18 12

Veamos ahora las operaciones de desplazamiento , la sentencia

c = a << b

implica asignarle a c, el valor de a con sus bits corridos a la izquierda en b lugares , los bits que van "saliendo" por la izquierda , se desechan ; y los bits que van quedando libres a la derecha se completan con cero .

Se procede de la misma manera para el corrimiento a la derecha >>.

El operador COMPLEMENTO A UNO es del tipo unitario , es decir que realiza una operación sobre una única variable , y su efecto es dar a la variable un valor igual a restar de (-1) el valor que traía . Quizás es más visible decir que este operador cambia los bits en 1 de la variable en 0 y viceversa.

TABLA 13 PRECEDENCIA DE LOS OPERADORES

PRECEDENCIA	OPERADORES	ASOCIATIVIDAD
0	()[] -> .	izq. a derecha
1	sizeof (tipo) ! ~ ++ -- signo* &	derecha a izq.
2	* / %	izq. a derecha
3	+ -	izq. a derecha
4	>	izq. a derecha
5	>=	izq. a derecha
6	== !=	izq. a derecha
7	&	izq. a derecha
8	^	izq. a derecha
9	|	izq. a derecha
10	&&	izq. a derecha
11	||	izq. a derecha
12	?:	derecha a izq.
13	= += -= *= etc	derecha a izq.

NOTA: en el renglón de los operadores de precedencia cero hemos agregado ubicándolos a la derecha del mismo para diferenciarlos, tres operadores , [] ->> y . que serán analizados más adelante, de la misma manera en el renglón siguiente hemos colocado al final a dos operadores: * y & ya que aunque coinciden en símbolo con los de PRODUCTO y AND A NIVEL BITS, son OTRO tipo de operadores que se describirán en capítulos sucesivos. En ese mismo renglón se ha consignado como SIGNO al unitario - .

CAPITULO 4: PROPOSICIONES PARA EL CONTROL DE FLUJO DE PROGRAMA

1. INTRODUCCION

En lo que sigue de este capítulo, denominaremos BLOQUE DE SENTENCIAS al conjunto de sentencias individuales incluídas dentro un par de llaves. Por ejemplo :

```
{
sentencia 1 ;
sentencia 2 ;
.....
sentencia n ;
}
```

Este conjunto se comportará sintacticamente como una sentencia simple y la llave de cierre del bloque NO debe ir seguida de punto y coma .

Un ejemplo de bloque ya visto , es el cuerpo del programa principal de la función main() .

```
main()
{
bloque de sentencias
}
```

En las proposiciones de control de flujo de programa , trabajaremos alternativamente con sentencias simples y bloques de ellas .

2. PROPOSICION IF - ELSE

Esta proposición sirve para ejecutar ciertas sentencias de programa , si una expresión resulta CIERTA ú otro grupo de sentencias, si aquella resulta FALSA. Su interpretación literal sería : SI es CIERTA tal cosa , haga tal otra , si no lo es salteéla .

El caso más sencillo sería :

```
if(expresión)
 sentencia ;
```

ó

```
if(expresión)  sentencia ;
```

Cuando la sentencia que sigue al IF es única, las dos formas de escritura expresadas arriba son equivalentes . La sentencia sólo se ejecutará si el resultado de "expresión" es distinto de cero (CIERTO) , en caso contrario el programa salteará dicha sentencia , realizando la siguiente en su flujo.

Veamos unos ejemplos de las distintas formas que puede adoptar la "expresión" dentro de un IF :

if(a > b)	if((a > b) != 0)	las dos expresiones son idénticas, aunque a veces resulta más claro expresarla de la segunda manera, sobre todo en los primeros contactos con el lenguaje.
if(a)	if(a != 0)	
if(!a)	if(a == 0)	Las dos superiores son idénticas entre sí , al igual que las dos inferiores Obsérvese que (!a) dará un valor CIERTO sólo cuando a sea FALSO. (ver operador NEGACION en

		el capítulo anterior)
if(a == b)		
if(a = b)	if(a == b)	La primera es una expresión correcta , el IF se realizará sólo si a es igual a b. En cambio la segunda es un error , ya que no se está comparando a con b , sino ASIGNANDO el valor de esta a aquella . Sin embargo, a veces puede usarse como un truco (un poco sucio) de programacion , ya que primero se realiza la asignación y luego se evalúa el resultado de esta para realizar el IF , es entonces equivalente a escribir :
		a = b ;
if(a)	.....	con el ahorro de una linea de programa (a costa de la legibilidad del mismo).

En casos más complejos que los anteriores , la proposición IF puede estar seguida por un bloque de sentencias :

if(expresión)	if(expresión) {
{	sentencia 1 ;
sentencia 1 ;	sentencia 2 ;
sentencia 2 ;	.....
.....	}
}	

Las dos maneras son equivalentes , por lo que la posición de la llave de apertura del bloque queda librada al gusto del programador . El indentado de las sentencias (sangría) es también optativo , pero sumamente recomendable ,sobre todo para permitir la lectura de proposiciones muy complejas ó anidadas , como se verá luego. El bloque se ejecutará en su conjunto si la expresion resulta CIERTA. El uso del ELSE es optativo , y su aplicación resulta en la ejecución de una , ó una serie de sentencias en el caso de que la expresión del IF resulta FALSA.

Su aplicación puede verse en el ejemplo siguiente :

if(expresión)	if(expresión)
{	{
sentencia 1 ;	sentencia 1 ;
sentencia 2 ;	sentencia 2 ;
}	}

```
  sentencia 3 ;              else

  sentencia 4 ;              {

  sentencia 5 ;              sentencia 3 ;

                         sentencia 4 ;

                         }

                         sentencia 5 ;
```

En el ejemplo de la izquierda no se usa el ELSE y por lo tanto las sentencias 3 , 4 y 5 se ejecutan siempre . En el segundo caso , las sentencias 1 y 2 se ejecutan solo si la expresión es CIERTA , en ese caso las 3 y 4 NO se ejecutarán para saltarse directamente a la 5 , en el caso de que la expresión resulte FALSA se realizarán las 3 y 4 en lugar de las dos primeras y finalmente la 5 .
La proposición ELSE queda siempre asociada al IF más cercano , arriba de él .
Es común también , en caso de decisiones múltiples , el uso de anidamientos ELSE-IF de la forma indicada abajo:

```
if(exp.1)                  if(exp.1)

 sentencia1 ;               sentencia1 ;

else  if(exp.2)            else  if(exp.2)

 sentencia2 ;                   sentencia2 ;

else if(exp.3)                  else if(exp.3)

 sentencia3 ;                        sentencia3 ;

else                            else

 sentencia5 ;                       sentencia5 ;
```

Si bién se suele escribir según la modalidad de la izquierda , a la derecha hemos expresado las asociaciones entre los distintos ELSE é IF por medio del indentado del texto.

3. PROPOSICION SWITCH

El SWITCH es una forma sencilla de evitar largos , tediosos y confusos anidamientos de ELSE-IF .
Supongamos que estamos implementando un Menu , con varias elecciones posibles . El esqueleto de una posible solución al problema usando if-else podría ser el siguiente :

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

main()

{
```

```
int c  ;
printf("\nMENU :") ;
printf("\n       A = ADICIONAR A LA LISTA ") ;
printf("\n       B = BORRAR DE LA LISTA   ") ;
printf("\n       O = ORDENAR LA LISTA     ") ;
printf("\n       I = IMPRIMIR LA LISTA    ") ;
printf("\n\nESCRIBA SU SELECCION , Y LUEGO <<ENTER>> : ") ;
if( (c = getchar()) != '\n' )
 {
  if( c == 'A')
    printf("\nUD. SELECCIONO AGREGAR") ;
  else
    if( c == 'B')
     printf("\nUD. SELECCIONO BORRAR") ;
    else
     if( c == 'O' )
      printf("\nUD. SELECCIONO ORDENAR") ;
     else
      if( c == 'I' )
       printf("\nUD. SELECCIONO IMPRIMIR") ;
      else
       printf("\n\a\aUD. APRETO UN CARACTER ILEGAL" ) ;
 }
else
  printf("\n¡ UD. NO HA SELECCIONADO NADA !" ) ;
```

```
}
```

Como es fácil de ver , cuando las opciones son muchas, el texto comienza a hacerse difícil de entender y engorroso de escribir.
El mismo programa, utilizando un SWITCH , quedaría mucho más claro de leer, y sencillo de escribir, como se aprecia en el EJEMPLO siguiente.

```
#include <stdio.h>

#include <conio.h>

main()
{

int c  ;

printf("\nMENU :") ;

printf("\n       A = ADICIONAR A LA LISTA ") ;

printf("\n       B = BORRAR DE LA LISTA   ") ;

printf("\n       O = ORDENAR LA LISTA     ") ;

printf("\n       I = IMPRIMIR LA LISTA    ") ;

printf("\n\nESCRIBA SU SELECCION , Y LUEGO <<ENTER>> : ") ;

c = getchar() ;

switch (c)

 {

  case 'A' :

    printf("\nUD. SELECCIONO AGREGAR") ;

    break ;

  case 'B' :

    printf("\nUD. SELECCIONO BORRAR") ;

    break ;
```

```
 case 'O' :
   printf("\nUD. SELECCIONO ORDENAR") ;
   break ;
 case 'I' :
   printf("\nUD. SELECCIONO IMPRIMIR") ;
   break ;
 case '\n':
   printf("\n¡ UD. NO HA SELECCIONADO NADA !" ) ;
   break ;
 default  :
   printf("\n\a\aUD. APRETO UN CARACTER ILEGAL" ) ;
   break ;
 }
}
```

El SWITCH empieza con la sentencia : switch (expresión) . La expresión contenida por los paréntesis debe ser ENTERA , en nuestro caso un caracter ; luego mediante una llave abre el bloque de las sentencias de comparación . Cada una de ellas se representa por la palabra clave "case" seguida por el valor de comparación y terminada por dos puntos . Seguidamente se ubican las sentencias que se quieren ejecutar , en el caso que la comparación resulte CIERTA . En el caso de resultar FALSA , se realizará la siguiente comparación , y así sucesivamente .
Prestemos atención tambien a la sentencia BREAK con la que se termina cada CASE. Una característica poco obvia del SWITCH , es que si se eliminan los BREAK del programa anterior , al resultar CIERTA una sentencia de comparación, se ejecutarán las sentencias de ese CASE particular pero TAMBIEN la de todos los CASE por debajo del que ha resultado verdadero. Quizás se aclare esto diciendo que , las sentencias propias de un CASE se ejecutarán si su comparación ú otra comparación ANTERIOR resulta CIERTA . La razón para este poco "juicioso" comportamiento del SWITCH es que así se permite que varias comparaciones compartan las mismas sentencias de programa , por ejemplo :

```
 .....
 case 'X' :
 case 'Y' :
```

```
case 'Z' :

 printf(" UD. ESCRIBIO X , Y , ó  Z ") ;

 break ;

 .....
```

La forma de interrumpir la ejecución luego de haber encontrado un CASE cierto es por medio del BREAK , el que dá por terminado el SWITCH .
Al final del bloque de sentencias del SWITCH , aparece una optativa llamada DEFAULT , que implica : si no se ha cumplido ningun CASE , ejecute lo que sigue. Es algo superfluo poner el BREAK en este caso , ya que no hay más sentencias despues del DEFAULT , sin embargo , como el orden en que aparecen las comparaciones no tiene importancia para la ejecución de la instrucción, puede suceder que en futuras correcciones del programa se agregue algún nuevo CASE luego del DEFAULT , por lo que es conveniente preveerlo , agregando el BREAK , para evitar errores de laboriosa ubicación .
Más adelante volveremos sobre otros usos del BREAK.

4. LA ITERACION WHILE

El WHILE es una de las tres iteraciones posibles en C . Su sintaxis podría expresarse de la siguiente forma :

```
while(expresion)          ó          while(expresión) {

  proposición 1 ;                        proposición 1 ;

                                         proposición 2 ;

                                         .............

                                         proposición n ;

                                         }
```

Esta sintaxis expresada en palabras significaria: mientras (expresión) dé un resultado CIERTO ejecútese la proposición 1 , en el caso de la izquierda ó ejecútese el bloque de sentencias , en el caso de la derecha.
Por lo general , dentro de la proposición ó del bloque de ellas , se modifican términos de la expresión condicional , para controlar la duración de la iteración .

5. LA ITERACION DO - WHILE

Su sintaxis será :

```
do {

 proposición 1 ;

 proposición 2 ;

 .............

} while (expresión) ;
```

Expresado en palabras , esto significa : ejecute las proposiciones , luego repita la ejecución mientras la expresión dé un resultado CIERTO . La diferencia fundamental entre esta iteración y la anterior es que el DO-WHILE se ejecuta siempre AL MENOS una vez , sea cual sea el resultado de expresión.

6. ITERACION FOR

El FOR es simplemente una manera abreviada de expresar un WHILE , veamos su sintaxis :

```
for ( expresión1 ; expresión2 ; expresion3 ) {

      proposición1 ;

      proposición2 ;

      .............

}
```

Esto es equivalente a :

```
expresión1 ;

while ( expresión2 ) {

      proposición1 ;

      proposición2 ;

      .............

      expresion3   ;

}
```

La expresión1 es una asignación de una ó más variables , (equivale a una inicialización de las mismas) , la expresión2 es una relación de algun tipo que , mientras dé un valor CIERTO , permite la iteración de la ejecución y expresión3 es otra asignación , que comunmente varía alguna de las variables contenida en expresión2 .

Todas estas expresiones , contenidas en el paréntesis del FOR deben estar separadas por PUNTO Y COMA y NO por comas simples .

No es imprescindible que existan TODAS las expresiones dentro del paréntesis del FOR , pudiendose dejar en blanco algunas de ellas , por ejemplo :

```
for ( ; exp2 ; exp3)          ó

for (exp1 ; ; )               ó

for ( ; ; )
```

Estas dos últimas expresiónes son interesantes desde el punto de vista de su falta de término relacional , lo que implica que el programador deberá haber previsto alguna manera alternativa de salir del lazo (probablemente mediante BREAK ó RETURN

como veremos más adelante) ya que sinó , la ejecución del mismo es infinita (ó tan larga como se mantenga encendida la computadora) .

7. LA SENTENCIA BREAK

El BREAK , ya brevemente descripto con el SWITCH , sirve también para terminar loops producidos por WHILE , DO-WHILE y FOR antes que se cumpla la condición normal de terminación . En el EJEMPLO siguiente vemos su uso para terminar un WHILE indeterminado.

```
#include <stdio.h>

#include <conio.h>

main()

{

  char c ;

  printf("ESTE ES UN LOOP INDEFINIDO ") ;

  while(1)  {

   printf( "DENTRO DEL LOOP INDEFINIDO (apriete una tecla):" ) ;

   if( (c = getch()) == 'Q' )

    break ;

   printf( "\nNO FUE LA TECLA CORRECTA PARA ABANDONAR EL LOOP ") ;

  }

   printf("\nTECLA CORRECTA : FIN DEL WHILE ") ;

}
```

Obsérvese que la expresión while(1) SIEMPRE es cierta , por lo que el programa correrá imparable hasta que el operador oprima la tecla "secreta" Q . Esto se consigue en el IF , ya que cuando c es igual al ASCII Q se ejecuta la instrucción BREAK ,dando por finalizado el WHILE .
El mismo criterio podría aplicarse con el DO-WHILE ó con FOR , por ejemplo haciendo

```
 for (;;)  {     /* loop indefinido */

 .....

  if( expresión )
```

```
  break ;       /* ruptura del loop cuando expresión sea verdadera */

 }
```

8. LA SENTENCIA CONTINUE

La sentencia CONTINUE es similar al BREAK con la diferencia que en vez de terminar violentamente un loop , termina con la realización de una iteración particular y permitiendo al programa continuar con la siguiente.

9. LA FUNCION EXIT()

La función EXIT() tiene una operatoria mucho más drastica que las anteriores , en vez de saltear una iteración ó abandonar un lazo de programa , esta abandona directamente al programa mismo dándolo por terminado . Realiza también una serie de operaciones útiles como ser , el cerrado de cualquier archivo que el programa hubiera abierto , el vaciado de los buffers de salida , etc.

Normalmente se la utiliza para abortar los programas en caso de que se esté por cometer un error fatal é inevitable . Mediante el valor que se le ponga en su argumento se le puede informar a quien haya llamado al programa (Sistema Operativo , archivo .bat , u otro programa) el tipo de error que se cometió.

10 SENTENCIA GOTO

Si Ud. se ha admirado de que C tenga la operación GOTO , recuerde que el hecho de existir NO lo obliga a usarla , en el mismo sentido que por tener puertas los aviones no está obligado a saltar por ellas en pleno vuelo.

El uso del GOTO implica un salto incondicional de un lugar a otro del programa . Esta práctica hace que los programas sean muy dificiles de corregir ó mantener.

Si no quedara más remedio que usarlo, (y en programación estructurada SIEMPRE hay remedio) debe marcarse el destino del salto mediante un nombre seguido por dos puntos .

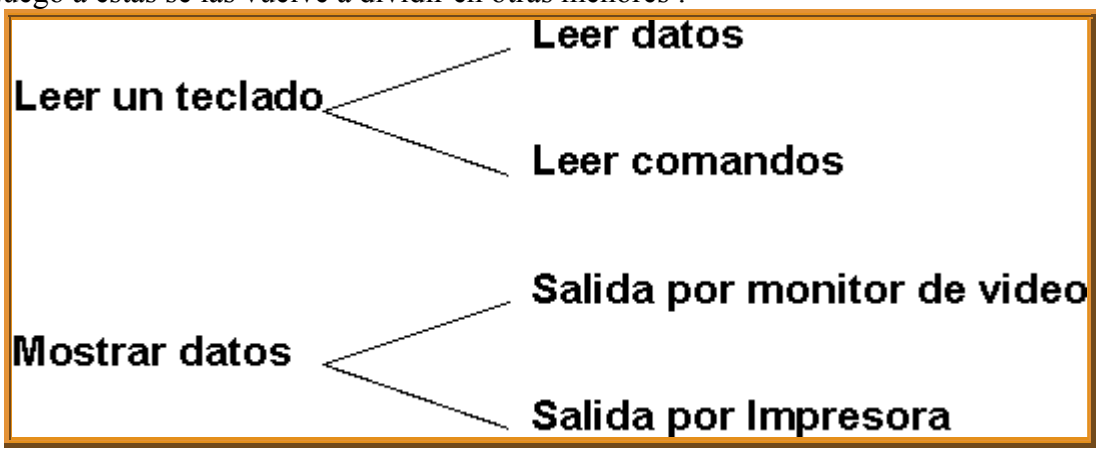
```
if( c == 0 )  goto OTRO_LADO ;
.............................
OTRO_LADO:
 printf(.....
```

En este caso si c es cero se saltean todas las sentencias entre el if y el destino , continuandose con la ejecución del printf() . El destino puede ser tanto posterior como anterior al GOTO invocante .

CAPITULO 5: FUNCIONES

1.INTRODUCCION

La forma más razonable de encarar el desarrollo de un programa complicado es aplicar lo que se ha dado en llamar "Programación Top - Down" .
Esto implica que, luego de conocer cual es la meta a alcanzar, se subdivide esta en otras varias tareas concurrentes, por ejemplo :
Leer un teclado, procesar datos, mostrar los resultados .
Luego a estas se las vuelve a dividir en otras menores :



Y así se continúa hasta llegar a tener un gran conjunto de pequeñas y simples tareas, del tipo de "leer una tecla" ó "imprimir un caracter".
Luego sólo resta abocarse a resolver cada una de ellas por separado.
De esta forma el programador, sólo se las tendrá que ver con diminutas piezas de programa, de pocas lineas, cuya escritura y corrección posterior es una tarea simple.
Tal es el criterio con que está estructurado el lenguaje C, donde una de sus herramientas fundamentales són las funciones. Todo compilador comercial trae una gran cantidad de Librerias de toda índole, matematicas, de entrada - salida, de manejo de textos, de manejo de gráficos, etc, que solucionan la mayor parte de los problemas básicos de programación .
Sin embargo será inevitable que en algún momento tenga que crear mis propias funciones, las reglas para ello son las que desarrollaremos en este capítulo .
Comencemos con algunos conceptos básicos: para hacer que las instrucciones contenidas en una función, se ejecuten en determinado momento, no es necesario más que escribir su nombre como una linea de sentencia en mi programa.
Convencionalmente en C los nombres de las funciones se escriben en minúscula y siguen las reglas dadas anteriormente para los de las variables, pero deben ser seguidos, para diferenciarlas de aquellas por un par de paréntesis .
Dentro de estos paréntesis estarán ubicados los datos que se les pasan a las funciones. Está permitido pasarles uno, ninguno ó una lista de ellos separados por comas, por ejemplo: pow10(a), getch(), strcmp(s1, s2) .
Un concepto sumamente importante es que los argumentos que se les envían a las funciones son los VALORES de las variables y NO las variables mismas. En otras palabras, cuando se invoca una función de la forma pow10(a) en realidad se está copiando en el "stack" de la memoria el valor que tiene en ese momento la variable a, la función podrá usar este valor para sus cálculos, pero está garantizado que los mismos no

afectan en absoluto a la variable en sí misma.
Como veremos más adelante, es posible que una función modifique a una variable, pero para ello, será necesario comunicarle la DIRECCION EN MEMORIA de dicha variable .
Las funciones pueden ó no devolver valores al programa invocante. Hay funciones que tan sólo realizan acciones, como por ejemplo clrscr(), que borra la pantalla de video, y por lo tanto no retornan ningun dato de interés; en cambio otras efectuan cálculos, devolviendo los resultados de los mismos.
La invocación a estos dos tipos de funciones difiere algo, por ejemplo escribiremos :

```
clrscr() ;

c = getch() ;
```

donde en el segundo caso el valor retornado por la función se asigna a la variable c. Obviamente ésta deberá tener el tipo correcto para alojarla .

2. DECLARACION DE FUNCIONES

Antes de escribir una función es necesario informarle al Compilador los tamaños de los valores que se le enviarán en el stack y el tamaño de los valores que ella retornará al programa invocante .
Estas informaciones están contenidas en la DECLARACION del PROTOTIPO DE LA FUNCION.
Formalmente dicha declaración queda dada por :
tipo del valor de retorno nombre_de_la_función(lista de tipos de parámetros)
Pongamos algunos ejemplos :

```
float mi_funcion(int i, double j ) ;

double otra_funcion(void) ;

      otra_mas(long p) ;

void   la_ultima(long double z, char y, int x, unsigned long w) ;
```

El primer término del prototipo da, como hemos visto el tipo del dato retornado por la función; en caso de obviarse el mismo se toma, por omisión, el tipo int. Sin embargo, aunque la función devuelva este tipo de dato, para evitar malas interpretaciones es conveniente explicitarlo .
Ya que el "default" del tipo de retorno es el int, debemos indicar cuando la función NO retorna nada, esto se realiza por medio de la palabra VOID (sin valor).
De la misma manera se actúa, cuando no se le enviarán argumentos.
Más adelante se profundizará sobre el tema de los argumentos y sus características.
La declaración debe anteceder en el programa a la definición de la función. Es normal, por razones de legibilidad de la documentación, encontrar todas las declaraciones de las funciones usadas en el programa, en el HEADER del mismo, junto con los include de los archivos *.h que tienen los prototipos de las funciones de Librería.
Si una ó más de nuestras funciones es usada habitualmente, podemos disponer su prototipo en un archivo de texto, e incluirlo las veces que necesitemos, según se vio en capítulos previos.

3. DEFINICION DE LAS FUNCIONES

La definición de una función puede ubicarse en cualquier lugar del programa, con sólo dos restricciones: debe hallarse luego de dar su prototipo, y no puede estar dentro de la

definición de otra función (incluida main()). Es decir que a diferencia de Pascal, en C las definiciones no pueden anidarse.
NOTA: no confundir definición con llamada; una función puede llamar a tantas otras como desee .
La definición debe comenzar con un encabezamiento, que debe coincidir totalmente con el prototipo declarado para la misma, y a continuación del mismo, encerradas por llaves se escribirán las sentencias que la componen; por ejemplo:

```
#include <stdio.h>

float mi_funcion(int i, double j ); /* DECLARACION observe que termina en  ";"  */

main()

{

float k ;

int p ;

double z ;

.....
k = mi_funcion( p, z );            /* LLAMADA a la función */

.....

}                                  /* fin de la función main() */
float mi_funcion(int i, double j )   /* DEFINICION  observe que NO lleva  ";"  */

{

float n

.....

printf("%d", i );                  /* LLAMADA  a otra función */

.....

return ( 2 * n );                  /* RETORNO devolviendo un valor float */

}
```

Pasemos ahora a describir más puntualmente las distintas modalidades que adoptan las funciones .

4. FUNCIONES QUE NO RETORNAN VALOR NI RECIBEN PARAMETROS

Veamos como ejemplo la implementacion de una funcion "pausa"

```
#include <stdio.h>

void pausa(void) ;
```

```
main()

{

  int contador = 1;
  printf("VALOR DEL CONTADOR DENTRO DEL while \n");
  while (contador <= 10) {

    if(contador == 5 ) pausa();

    printf("%d\n", contador++);

  }

  pausa() ;

  printf("VALOR DEL CONTADOR LUEGO DE SALIR DEL while: %d", contador) ;

  return 0;

}

void pausa(void)

{

char c ;
printf("\nAPRIETE ENTER PARA CONTINUAR ") ;

while( (c = getchar()) != '\n') ;

}
```

Analicemos lo hecho, en la segunda linea hemos declarado la función pausa, sin valor de retorno ni parámetros.
Luego esta es llamada dos veces por el programa principal, una cuando contador adquiere el valor de 5 (antes de imprimirlo) y otra luego de finalizar el loop.
Posteriormente la función es definida. El bloque de sentencias de la misma está compuesto, en este caso particular, por la definición de una variable c, la impresión de un mensaje de aviso y finalmente un while que no hace nada, solo espera recibir un caracter igual a <ENTER>.
En cada llamada, el programa principal transfiere el comando a la función, ejecutandose, hasta que ésta finalice, su propia secuencia de instrucciones. Al finalizar la función esta retorna el comando al programa principal, continuandose la ejecución por la instrucción que sucede al llamado .
Si bien las funciones aceptan cualquier nombre, es una buena técnica de programación nombrarlas con términos que representen, aunque sea vagamente, su operatoria .
Se puede salir prematuramente de una función void mediante el uso de RETURN, sin que este sea seguido de ningun parámetro ó valor .

5. FUNCIONES QUE RETORNAN VALOR

Analicemos por medio de un ejemplo dichas funciones :

```
#include <stdio.h>

#include <conio.h>
#define FALSO 0

#define CIERTO 1
int finalizar(void);

int lea_char(void) ;
main()

{

  int i = 0;

  int fin = FALSO;

  printf("Ejemplo de Funciones que retornan valor\n");

  while (fin == FALSO) {

    i++;

    printf("i == %d\n", i);

    fin = finalizar();

  }

  printf("\n\nFIN DEL PROGRAMA.....");

  return 0;

}

int finalizar(void)

{

  int c;

  printf("Otro número ? (s/n) ");

  do {

    c = lea_char() ;
```

```
  } while ((c != 'n') && (c != 's'));

  return (c == 'n');

}
int lea_char(void)

{

int j ;
if( (j = getch()) >>= 'A' && j <<= 'Z' )

 return( j + ( 'a' - 'A') ) ;

else

 return j ;

}
```

Analicemos paso a paso el programa anterior; las dos primeras lineas incluiran, en el programa los prototipos de las funciones de librería usadas, (en este caso printf() y getch()). En las dos siguientes damos nombres simbólicos a dos constantes que usaremos en las condiciones lógicas y posteriormente damos los prototipos de dos funciones que hemos creado.
Podrían haberse obviado, en este caso particular, estas dos últimas declaraciones, ya que ambas retornan un int (default), sin embargo el hecho de incluirlas hará que el programa sea más facilmente comprensible en el futuro.
Comienza luego la función main(), inicializando dos variables, i y fin, donde la primera nos servirá de contador y la segunda de indicador lógico. Luego de imprimir el rótulo del programa, entramos en un loop en el que permaneceremos todo el tiempo en que fin sea FALSO.
Dentro de este loop, incrementamos el contador, lo imprimimos, y asignamos a fin un valor que es el retorno de la función finalizar() .
Esta asignación realiza la llamada a la función, la que toma el control del flujo del programa, ejecutando sus propias instrucciones.
Saltemos entonces a analizar a finalizar(). Esta define su variable propia, c, (de cuyas propiedades nos ocuparemos más adelante) y luego entra en un do-while, que efectúa una llamada a otra función, lea_char(), y asigna su retorno a c iterando esta operativa si c no es 'n' ó 's', note que: c != 'n' && c != 's' es equivalente a: !(c == 'n' || c == 's') .
La función lea_char() tiene como misión leer un caracter enviado por el teclado, (lo realiza dentro de la expresión relacional del IF) y salvar la ambigüedad del uso de mayúsculas ó minúsculas en las respuestas, convirtiendo las primeras en las segundas.
Es facil de ver que, si un caracter esta comprendido entre A y Z, se le suma la diferencia entre los ASCII de las minúsculas y las mayúsculas (97 - 65 = 32) para convertirlo, y luego retornarlo al invocante.
Esta conversión fué incluida a modo de ejemplo solamente, ya que existe una de Librería, tolower() declarada en ctype.h, que realiza la misma tarea.

Cuando lea_char() devuelva un caracter n ó s, se saldrá del do-while en la función finalizar() y se retornará al programa principal, el valor de la comparación lógica entre el contenido de c y el ASCII del caracter n. Si ambos son iguales, el valor retornado será 1 (CIERTO) y en caso contrario 0 (FALSO) .
Mientras el valor retornado al programa principal sea FALSO, este permanecerá dentro de su while imprimiendo valores sucesivos del contador, y llamadas a las funciones, hasta que finalmente un retorno de CIERTO (el operador presionó la tecla n) hace terminar el loop e imprimir el mensaje de despedida.
Nota: preste atención a que en la función finalizar() se ha usado un do-while .¿Cómo modificaría el programa para usar un while ?. En la función lea_char se han usado dos returns, de tal forma que ella sale por uno u otro. De esta manera si luego de finalizado el else se hubiera agregado otra sentencia, esta jamás sería ejecutada.
En el siguiente ejemplo veremos funciones que retornan datos de tipo distinto al int.
Debemos presentar antes, otra función muy común de entrada de datos: scanf(), que nos permitirá leer datos completos (no solo caracteres) enviados desde el teclado, su expresión formal es algo similar a la del printf() ,
scanf("secuencia de control", dirección de la variable) ;
Donde en la secuencia de control se indicará que tipo de variable se espera leer, por ejemplo :

%d si se desea leer un entero decimal (int)

%o " " " " " " octal "

%x " " " " " " hexadecimal "

%c " " " " " caracter

%f leerá un flot

%ld leerá un long int

%lf leerá un double

%Lf leerá un long double

Por "dirección de la variable" deberá entenderse que se debe indicar, en vez del nombre de la variable en la que se cargará el valor leido, la dirección de su ubicación en la memoria de la máquina. Esto suena sumamente apabullante, pero por ahora solo diremos, (más adelante abundaremos en detalles) que para ello es necesario simplemente anteponer el signo & al nombre de la misma .
Así, si deseo leer un entero y guardarlo en la variable "valor_leido" escribiré:
scanf("%d",&valor_leido); en cambio si deseara leer un entero y un valor de punto flotante será: scanf("%d %f", &valor_entero, &valor_punto_flotante) ;
El tipo de las variables deberá coincidir EXACTAMENTE con los expresados en la secuencia de control, ya que de no ser así, los resultados son impredecibles.
Por supuesto esta función tiene muchísimas más opciones, (consulte el Manual de Librerias de su Compilador, si tiene curiosidad) sin embargo, por simplicidad, por ahora nos conformaremos con las antedichas.
El prototipo de scanf() esta declarado en stdio.h .

Usaremos también otra función, ya citada, clrscr(). Recordemos que esta es solo válida para máquinas tipo PC compatibles y no corre bajo Windows.
Encaremos ahora un programa que nos presente primero, un menú para seleccionar la conversión de ºC a Fahrenheit ó de centímetros a pulgadas, hecha nuestra elección, nos pregunte el valor a convertir y posteriormente nos de el resultado .
Si suponemos que las funciones que usaremos en el programa serán frecuentemente usadas, podemos poner las declaraciones de las mismas, así como todas las contantes que usen, en un archivo texto, por ejemplo convers.h. Este podrá ser guardado en el subdirectorio donde están todos los demás (INCLUDE) ó dejado en el directorio activo, en el cual se compila el programa fuente de nuestro problema. Para variar, supongamos que esto último es nuestro caso .
CONVERS.H

```
#include <conio.h>

#define FALSO  0

#define CIERTO 1

#define CENT_POR_INCH  25.4
void pausa(void)              ;

void mostrar_menu(void)       ;

int seleccion(void)           ;

void cm_a_pulgadas(void)      ;

void grados_a_fahrenheit(void)  ;

double leer_valor(void)       ;
```

Vemos que un Header puede incluir llamadas a otros (en este caso conio.h). Hemos puesto tambien la definición de todas las constantes que usaran las funciones abajo declaradas. De dichas declaraciones vemos que usaremos funciones que no retornan nada, otra que retorna un entero y otra que devuelve un double .
Veamos ahora el desarrollo del programa en sí. Observe que la invocación a conversión.h se hace con comillas, por haber decidido dejarlo en el directorio activo .

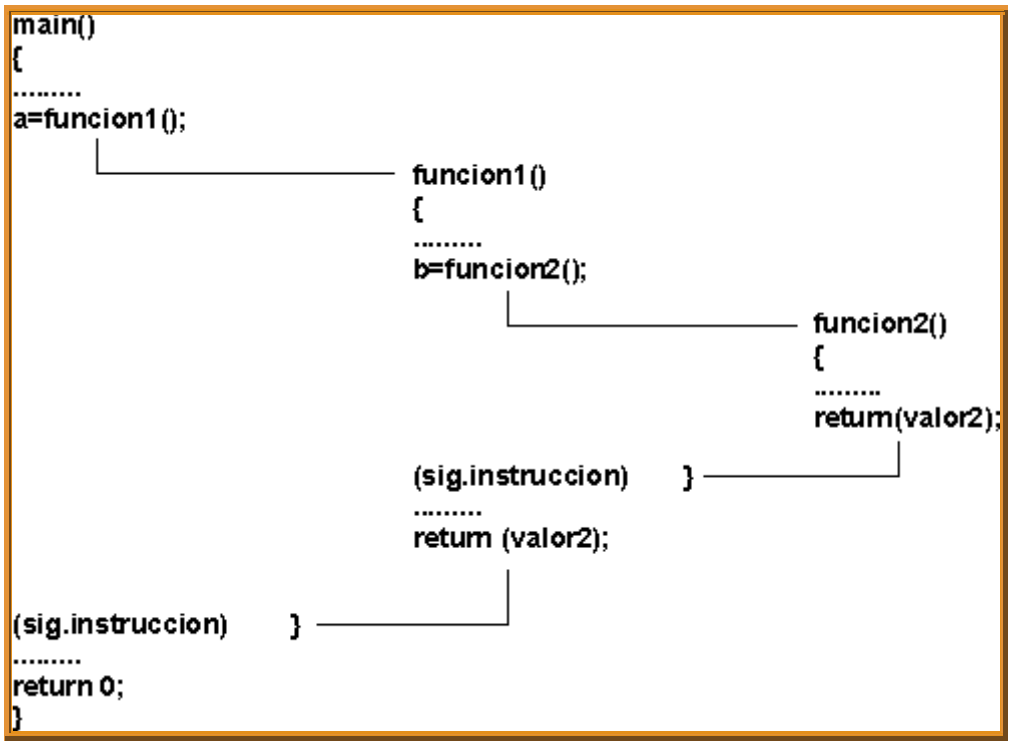
```
#include <stdio.h>

#include "convers.h"
main()

{

  int fin = FALSO;
  while (!fin) {
```

```
    mostrar_menu();
    switch(seleccion()) {
      case 1:
        cm_a_pulgadas();
        break;
      case 2:
        grados_a_fahrenheit();
        break;
      case 3:
        fin = CIERTO;
        break;
      default:
        printf("\n¡Error en la Seleccion!\a\a\n");
        pausa() ;
    }
  }
  return 0;
}
/* Funciones */
void pausa(void)
{
char c = 0;
printf("\n\n\nAPRIETE ENTER PARA CONTINUAR ") ;
while( (c = getch()) != '\r') ;
}
void mostrar_menu(void)
```

```
{
  clrscr();
  printf("\n         Menu\n");
  printf("---------------------------\n");
  printf("1:  Centimetros a pulgadas\n");
  printf("2:  Celsius a Fahrenheit\n");
  printf("3:  Terminar\n");
}
int seleccion(void)
{
  printf("\nEscriba el número de su Selección: ");
  return (getche() - '0');
}
void cm_a_pulgadas(void)
{
  double centimetros;  /* Guardará el valor pasado por leer_valor() */
  double pulgadas    ; /* Guardará el valor calculado */
  printf("\nEscriba los Centimetros a convertir: ");
  centimetros = leer_valor();
  pulgadas = centimetros  * CENT_POR_INCH;
  printf("%.3f Centimetros = %.3f Pulgadas\n", centimetros, pulgadas);
  pausa()   ;
}
void grados_a_fahrenheit(void)
{
  double grados;  /* Guardará el valor pasado por leer_valor() */
  double fahrenheit  ; /* Guardará el valor calculado */
```

```
  printf("\nEscriba los Grados a convertir: ");

  grados = leer_valor();

  fahrenheit = (((grados * 9.0)/5.0) + 32.0) ;

  printf("%.3f Grados = %.3f Fahrenheit", grados, fahrenheit);

  pausa();

}
double leer_valor(void)

{

  double valor;  /* Variable para guardar lo leido del teclado */
  scanf("%lf", &valor);

  return valor;

}
```

Veamos que hemos hecho: primero incluimos todas las definiciones presentes en el archivo convers.h que habiamos previamente creado. Luego main() entra en un loop, que finalizará cuando la variable fin tome un valor CIERTO, y dentro del cual lo primero que se hace es llamar a mostrar_menú(), que pone los rótulos de opciones . Luego se entra en un SWITCH que tiene como variable ,el retorno de la función selección() (recuerde que tiene que ser un entero), según sea éste se saldrá por alguno de los tres CASE. Observe que selección() lee el teclado mediante un getche(), (similar a getch() antes descripta, pero con la diferencia que aquella hace eco del caracter en la pantalla) y finalmente devuelve la diferencia entre el ASCII del número escrito menos el ASCII del número cero, es decir, un entero igual numericamente al valor que el operador quizo introducir .
Si este fue 1, el SWITCH invoca a la función cm_a_pulgadas() y en caso de ser 2 a grados_a_fahrenheit() .
Estas dos últimas proceden de igual manera: indican que se escriba el dato y pasan el control a leer_valor(), la que mediante scanf() lo hace, retornando en la variable valor, un double, que luego es procesado por aquellas convenientemente. Como hasta ahora la variable fin del programa principal no ha sido tocada, y por lo tanto continua con FALSO ,la iteración del while sigue realizandose, luego que se ejecuta el BREAK de finalización del CASE en cuestión. En cambio, si la selección() hubiera dado un resultado de tres, el tercer case, la convierte en CIERTO, con lo que se finaliza el WHILE y el programa termina.
Vemos en este ejemplo, la posibilidad de múltiples llamados a funciones, una llama a otra, que a su vez llama a otra, la cual llama a otra, etc ,etc, dando un esquema de flujo de programa de la forma :



6. AMBITO DE LAS VARIABLES (SCOPE)

VARIABLES GLOBALES

Hasta ahora hemos diferenciado a las variable segun su "tipo" (int, char double, etc), el cual se refería, en última instancia, a la cantidad de bytes que la conformaban. Veremos ahora que hay otra diferenciación de las mismas, de acuerdo a la clase de memoria en la que residen .

Si definimos una variable AFUERA de cualquier función (incluyendo esto a main()), estaremos frente a lo denominado VARIABLE GLOBAL. Este tipo de variable será ubicada en el segmento de datos de la memoria utilizada por el programa, y existirá todo el tiempo que esté ejecutandose este .

Este tipo de variables son automaticamente inicializadas a CERO cuando el programa comienza a ejecutarse .

Son accesibles a todas las funciones que esten declaradas en el mismo, por lo que cualquiera de ellas podrá actuar sobre el valor de las mismas. Por ejemplo :

```
#include <stdio.h>

double una_funcion(void);

double variable_global ;

main()

{

double i ;

printf("%f", variable_global );       /* se imprimirá 0 */
```

```
i = una_funcion() ;

printf("%f", i );                    /* se imprimirá 1 */

printf("%f", variable_global );      /* se imprimirá 1 */

variable_global += 1 ;

printf("%f", variable_global );      /* se imprimirá 2 */

return 0 ;

}
double una_funcion(void)

{

return( variable_global  += 1) ;

}
```

Observemos que la variable_global está definida afuera de las funciones del programa, incluyendo al main(), por lo que le pertenece a TODAS ellas. En el primer printf() del programa principal se la imprime, demostrandose que está automaticamente inicializada a cero .
Luego es incrementada por una_funcion() que devuelve ademas una copia de su valor, el cual es asignado a i ,la que, si es impresa mostrará un valor de uno, pero tambien la variable_global ha quedado modificada, como lo demuestra la ejecución de la sentencia siguiente. Luego main() tambien modifica su valor , lo cual es demostrado por el printf() siguiente.
Esto nos permite deducir que dicha variable es de uso público, sin que haga falta que ninguna función la declare, para actuar sobre ella.
Las globales son a los demás tipos de variables, lo que el GOTO es a los otros tipos de sentencias .
Puede resultar muy difícil evaluar su estado en programas algo complejos, con múltiples llamados condicionales a funciones que las afectan, dando comunmente orígen a errores muy engorrosos de corregir .

VARIABLES LOCALES

A diferencia de las anteriores, las variables definidas DENTRO de una función, son denominadas VARIABLES LOCALES a la misma, a veces se las denomina también como AUTOMATICAS, ya que son creadas y destruídas automaticamente por la llamada y el retorno de una función, respectivamente .
Estas variables se ubican en la pila dinámica (stack) de memoria ,destinandosele un espacio en la misma cuando se las define dentro de una función, y borrándose cuando la misma devuelve el control del programa, a quien la haya invocado.
Este método permite que, aunque se haya definido un gran número de variables en un programa, estas no ocupen memoria simultaneamente en el tiempo, y solo vayan incrementando el stack cuando se las necesita, para luego, una vez usadas desaparecer,

dejando al stack en su estado original .
El identificador ó nombre que se la haya dado a una variable es sólo relevante entonces, para la función que la haya definido, pudiendo existir entonces variables que tengan el mismo nombre, pero definidas en funciones distintas, sin que haya peligro alguno de confusión .
La ubicación de estas variables locales, se crea en el momento de correr el programa, por lo que no poseen una dirección prefijada, esto impide que el compilador las pueda inicializar previamente. Recuerdese entonces que, si no se las inicializa expresamente en el momento de su definición, su valor será indeterminado (basura) .

VARIABLES LOCALES ESTATICAS

Las variables locales vistas hasta ahora, nacen y mueren con cada llamada y finalización de una función, sin embargo muchas veces sería util que mantuvieran su valor, entre una y otra llamada a la función sin por ello perder su ámbito de existencia, es decir seguir siendo locales sólo a la función que las defina. En el siguiente ejemplo veremos que esto se consigue definiendo a la variable con el prefacio static.

VARIABLES DE REGISTRO

Otra posibilidad de almacenamiento de las variables locales es, que en vez de ser mantenidas en posiciones de la memoria de la computadora, se las guarde en registros internos del Microprocesador que conforma la CPU de la misma .
De esta manera el acceso a ellas es mucho más directo y rápido, aumentando la velocidad de ejecución del programa. Se suelen usar registros para almacenar a los contadores de los FOR, WHILE, etc.
Lamentablemente, en este caso no se puede imponer al compilador, este tipo de variable, ya que no tenemos control sobre los registros libres en un momento dado del programa, por lo tanto se SUGIERE, que de ser posible, ubique la variable del modo descripto. El prefacio en éste caso será :
register int var_reg ;
Hay que recalcar que esto es sólo válido para variables LOCALES, siendo imposible definir en un registro a una global. Por otra parte las variables de registro no son accesibles por dirección, como se verá más adelante .

VARIABLES EXTERNAS

Al DEFINIR una variable, como lo hemos estado haciendo hasta ahora, indicamos al compilador que reserve para la misma una determinada cantidad de memoria, (sea en el segmento de memoria de datos, si es global ó en el stack, si es local), pero debido a que en C es normal la compilación por separado de pequeños módulos, que componen el programa completo, puede darse el caso que una función escrita en un archivo dado, deba usar una variable global definida en otro archivo. Bastará para poder hacerlo, que se la DECLARE especificando que es EXTERNA a dicho módulo, lo que implica que está definida en otro lado .
Supongamos que nuestro programa está compuesto por sólo dos módulos: mod_prin.c y mod_sec.c los cuales se compilarán y enlazarán juntos, por medio del compilador y el linker, por ejemplo corriendo: bcc mod_prin.c mod_sec.c si usaramos el compilador de Borland .
Si en el primer módulo (mod_prin.c) aparece una variable global, definida como
double var1 = 5 ;
El segundo módulo, ubicado en un archivo distinto de aquel, podrá referenciarla mediante la declaración de la misma :
extern double var1 ;

Notesé que la inialización de la variable sólo puede realizarse en su DEFINICION y no en la declaración. Esta última, no reserva memoria para la variable sino que sólo hace mención que la misma ha sido definida en otro lado .
Será finalmente el Linker el que resuelva los problemas de direccionamiento de la variable al encadenar los dos módulos compilados .

7. ARGUMENTOS Y PARAMETROS DE LAS FUNCIONES

Supongamos que en un determinado programa debemos calcular repetidamente el valor medio de dos variables, una solución razonable sería crear una función que realice dicho cálculo, y llamarla cada vez que se necesite. Para ello será necesario, en cada llamada, pasarle los valores de las variables para que calcule su valor medio. Esto se define en la declaración de la funcion especificando, no solo su valor de retorno sino también el tipo de argumentos que recibe :

```
double valor_medio(double x, double y) ;
```

de esta declaración vemos que la función valor_medio recibe dos argumentos (x e y) del tipo double y devuelve un resultado de ese mismo tipo .
Cuando definamos a la función en sí, deberemos incluir parámetros para que alberguen los valores recibidos, así escribiremos:

```
double valor_medio(double x, double y )

{

return ( (x + y) / 2.0 )

}
```

NOTA: No es necesario que los NOMBRES de los párametros coincidan con los declarados previamente, es decir que hubiera sido equivalente escribir: double valor_medio(double a, double b) etc, sin embargo es una buena costumbre mantenerlos igual. En realidad en la declaración de la función, no es necesario incluir el nombre de los parámetros, bastaría con poner solo el tipo, sin embargo es práctica generalizada, explicitarlos a fin de hacer más legible al programa .
Aquí estamos utilizando la síntaxis moderna del lenguaje C, pudiendose encontrar en versiones arcaicas, definiciones equivalentes como :

```
double valor_medio()      ó     double valor_medio(double, double)

double x;                       double x ;

double y;                       double y ;

{                           {

.....                       .....
```

Sin embargo es preferible utilizar la nomenclatura moderna, ya que esta facilita la rápida comprensión del programa .
Veamos un ejemplo, para determinar el comportamiento de los parámetros,
Supongamos desear un programa que calcule el valor medio de dos variables incrementadas en un valor fijo, es decir:

```
( ( x + incremento ) + ( y + incremento ) ) / 2.0
```

Lo podríamos resolver de la siguiente forma :

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

```
/* Declaración de la función y el tipo de sus parámetros */

double valor_medio(double p_valor, double s_valor, double inc) ;
main()

{

double x, y, z, resultado ;
printf("Ingrese el primer valor: ") ;

scanf("%lf", &x ) ;
printf("\nIngrese el segundo valor: ");

scanf("%lf", &y ) ;
printf("\nIngrese el incremento    : ");

scanf("%lf", &z) ;
resultado = valor_medio( x, y, z );     /* llamada a la función y

                                        pasaje de argumentos   */
printf("\n\nResultado de la operación: %lf", resultado) ;
printf("\n\nValor con que quedaron las variables: ") ;

printf("\n Primer valor : %lf ", x ) ;

printf("\n Segundo valor: %lf ", y ) ;

printf("\n Incremento   : %lf ", z ) ;
}
/* Definición de la función y sus parámetros */
double valor_medio( double p_valor, double s_valor, double inc )

{

p_valor  += inc ;

s_valor += inc ;
return ( (p_valor + s_valor ) / 2.0 ) ;
}
```

Veamos primero cual seria la salida de pantalla de este programa :

SALIDA DEL EJEMPLO
Ingrese el primer valor: [SUPONGAMOS ESCRIBIR: 10.0]

Ingrese el segundo valor: [" " : 8.0]

Ingrese el incremento : [" " : 2.0]
Resultado de la operación: 11.000000

Valor con que quedaron las variables:

Primer valor : 10.000000

Segundo valor: 8.000000

Incremento : 2.000000

Vemos que luego de obtenidos, mediante scanf(), los tres datos x, y, z, los mismos son pasados a la función de calculo en la sentencia de asignación de la variable resultado. La función inicializa sus parámetros (p_valor, s_valor e inc) con los valores de los argumentos enviados (x, y, z) y luego los procesa. La unica diferencia entre un argumento y una variable local, es que ésta no es inicializada automaticamente, mientras que aquellos lo són, a los valores de los argumentos colocados en la expresión de llamada.
Acá debemos remarcar un importante concepto: éste pasaje de datos a las funciones, se realiza COPIANDO el valor de las variables en el stack y No pasandoles las variables en sí. Esto se denomina: PASAJE POR VALOR y garantiza que dichas variables no sean afectadas de ninguna manera por la función invocada. Una clara prueba de ello es que, en la función valor_medio() se incrementa p_valor y s_valor, sumandoseles el contenido del parámetro inc. Sin embargo cuando, luego de retornar al programa principal, imprimimos las variables cuyos valores fueron enviados como parametros, vemos que conservan sus valores iniciales. Veremos más adelante que otras estructuras de datos pueden ser pasadas a las funciones por direcciones en vez de por valor, pudiendo aquellas modificarlas a gusto .
Debe aclararse que, el pasaje de argumentos, es también una OPERACION, por lo que las variables pasadas quedan afectadas por las reglas de Conversión Automática de Tipo, vistas en el Capítulo 2. Como ejemplo, si x hubiera sido definida en la función main() como int, al ser pasada como argumento a valor_medio() sería promovida a double. Especial cuidado debe tenerse entonces con los errores que pueden producirse por redondeo ó truncamiento, siendo una buena técnica de programación hacer coincidir los tipos de los argumentos con los de los parámetros.

CAPITULO 6: ESTRUCTURAS DE AGRUPAMIENTO DE VARIABLES

1. CONJUNTO ORDENADO DE VARIABLES (ARRAYS)

Los arreglos ó conjuntos de datos ordenados (arrays) recolectan variables del MISMO tipo , guardandolas en forma secuencial en la memoria . La cantidad máxima de variables que pueden albergar está sólo limitada por la cantidad de memoria disponible . El tipo de las variables involucradas puede ser cualquiera de los ya vistos , con la única restricción de que todos los componentes de un array deben ser del mismo tipo .
La declaración de un array se realiza según la siguiente sintaxis :
tipo de las variables nombre[cantidad de elementos] ;

Por ejemplo :

```
int var1[10] ;

char nombre[50] ;

float numeros[200] ;

long double  cantidades[25] ;
```

Si tomamos el primer caso , estamos declarando un array de 10 variables enteras , cada una de ellas quedará individualizada por el subíndice que sigue al nombre del mismo es decir :
var1[0] , var1[1] , etc , hasta var1[9] .
Nótese que la CANTIDAD de elementos es 10 , pero su numeración vá de 0 a 9 , y nó de 1 a 10 . En resumen un array de N elementos tiene subíndices válidos entre 0 y N - 1 . Cualquier otro número usado como subíndice , traerá datos de otras zonas de memoria , cuyo contenido es impredictible .
Se puede referenciar a cada elemento , en forma individual , tal como se ha hecho con las variables anteriormente , por ejemplo :

```
var1[5] = 40 ;

contador = var1[3] + 7 ;

if(var1[0] >>= 37)

.....
```

Tambien es posible utilizar como subíndice expresiones aritméticas , valores enteros retornados por funciones , etc . Así podríamos escribir :

```
printf(" %d " , var1[ ++i] ) ;

var1[8] = var1[ i + j ] ;

.....

int una_funcion(void) ;

var1[0] = var1[ una_funcion() ] * 15 ;
```

Por supuesto los subíndices resultantes de las operaciones tienen que estar acotados a aquellos para los que el array fué declarado y ser enteros .
La inicialización de los arrays sigue las mismas reglas que vimos para los otros tipos de variables , es decir : Si se declaran como globales (afuera del cuerpo de todas las funciones) cada uno de sus elementos será automaticamente inicializado a cero . Si en cambio , su declaracion es local a una función , no se realiza ninguna inicialización , quedando a cargo del programa cargar los valores de inicio .
La inicialización de un array local , puede realizarse en su declaración , dando una lista de valores iniciales:
int numero[8] = { 4 , 7 , 0 , 0 , 0 , 9 , 8 , 7 } ;
Obsérvese que la lista está delimitada por llaves . Otra posibilidad , sólo válida cuando se inicializan todos los elementos del array , es escribir :
int numero[] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 } ;
donde se obvia la declaración de la cantidad de elementos , ya que está implícita en la lista de valores constantes .
También se puede inicializar parcialmente un array , por ejemplo :
int numero[10] = { 1 , 1 , 1 } ;
en éste caso los tres primeros elementos del mismo valdran 1 , y los restantes cero en el caso que la declaración sea global , ó cualquier valor impredecible en el caso de que sea local .

2. CONJUNTO ORDENADO DE CARACTERES (STRINGS)

Los strings son simplementes arrays de caracteres , tal como los vimos hasta ahora , con el agregado de un último elemento constante : el caracter NULL (ASCII == 0 , simbolizado por la secuencia de escape \0) . Este agregado permite a las funciones que procesan a los mismos , determinar facilmente la finalización de los datos .
Podemos generar un string , declarando :

```
char car_str[] = { 'A' , 'B' , 'C' , 'D' , 0 } ;

char car_str[] = { 'A' , 'B' , 'C' , 'D' , '\0' } ;
```

Ambas maneras son equivalentes. Sin embargo hay , en el lenguaje C , una forma más compacta de declararlos :

```
char car_str[] = "ABCD" ;

char car_str[5] = "ABCD" ;

int  texto[] =  "renglon 1  \n renglon 2  \n " ;    /* ERROR */

unsigned char  texto[] =  "renglon 1  \n renglon 2  \n " ;
```

Simplemente en la declaración del mismo se encierran los caracteres que lo componen entre comillas . Obsérvese que en la segunda declaración , se ha explicitado (no es necesario) , la cantidad de elementos que tiene el string , y és uno más que la cantidad de caracteres con que se lo inicializa , para dejar lugar al NULL . Todas éstas declaraciones agregan automáticamente el NULL como último elemento del array .
Un caso interesante es él de la tercer línea (comentada como ERROR) , con el fín de poder albergar al caracter "\n"20(ASCII 179) se intentó asignar el string a un array de enteros , Esto no es permitido por el compilador , que lo rechaza como una asignación inválida . La razón de ello se verá más adelante cuando analicemos punteros , ya que el string constante usado como rvalue es un puntero a char , y no a int . La solución mas común para este caso es , declarar el array como unsigned char , con lo que llevamos el alcance de sus elementos a 255 . Si tuvieramos el caso de tener que albergar en un string

el caracter EOF (-1) y al mismo tiempo caracteres con ASCII mayor que 127 ,se podría definir el array como int , pero su inicialización se tendrá que hacer obligatoriamente usando llaves , como vimos anteriormente .
Se deduce entonces , de lo antedicho que un string sigue siendo un array de caracteres , con la salvedad del agregado de un terminador , por lo que las propiedades que veremos a continuacion , se aplicaran indistintamente a ambos .

3. ARRAYS Y STRINGS COMO ARGUMENTOS DE FUNCIONES

Los arrays , como todos los otros tipos de variables , pueden ser pasados como argumentos a las funciones . Veamos esquematicamente como sería la sintaxis :

```
double funcion_1( float numeros[10] , char palabra[] ) ;       /*linea 1*/

.....

main()                                          /*linea 2*/

{

float numeros[10] = { 1.1 , 2.2 , 3.0 } ;                  /*linea 3*/

char palabra[] = " Lenguaje C " ;                          /*linea 4*/

double c ;                                       /*linea 5*/

.....

c = funcion_1( numeros , palabra )                          /*linea 6*/

.....

}

double funcion_1( float numeros[10] , char palabra[] )        /*linea 7*/

{

.....

}
```

Es necesario analizar con mucho detenimiento , este último ejemplo . En la primer línea declaramos el prototipo de funcion_1() que recibe como argumentos dos arrays , uno de 10 elementos del tipo float , y otro de caracteres de longitud indeterminada . En el primer caso la función necesitará saber de alguna manera cual es la longitud del array numérico recibido, mientras que en el segundo , no hace falta , ya que la función puede ser construída para que , por sí misma , detecte la finalización del string por la presencia del caracter NULL . Se podría generalizar más el programa declarando :

```
double funcion_1( double numeros[] , int longitud_array , char palabra[] ) ;
```

en donde , en la variable longitud_array se enviaría la cantidad de elementos de numero[] .
En la tercer línea se declara el array numérico , inicializandose sólo los tres primeros elementos , y en la cuarta línea se declara el string .
En la séptima línea se dá la definición de la función , de acuerdo al prototipo escrito anteriormente .
Si miramos con detenimiento la sexta línea , el llamado a la función , vemos que los argumentos pasados sólo tienen el NOMBRE de ambos arrays . Esta es la diferencia más importante entre este tipo de estructura de datos y las variables simples vistas anteriormente , ya que los arrays son pasados a las funciones por DIRECCION y nó por valor .
En el lenguaje C se prefiere , para evitar el uso abusivo del stack , cuando hay que enviar a una función una larga estructura de datos , en lugar de copiar a todos ellos , cargar el stack sólo con la dirección de la posición de memoria donde está ubicado el primero de los mismos.
El nombre de un array equivale sintácticamente a la direccion del elemento cero así será :
numero == dirección de numero[0]

palabra == direccion de palabra[0]

Esto habilita a las funciones a que puedan acceder a los arrays directamente , allí donde el programa los ha ubicado en la memoria , por lo que pueden MODIFICARLOS EN FORMA PERMANENTE aunque no hayan sido declarados como locales a la función misma ní globales al programa .
Es muy importante recordar este último concepto , a fín de evitar errores muy comunes , en los primeros intentos de programación en C .
Otra característica importante de los arrays es que , su nombre (ó dirección del primer elemento) es una CONSTANTE y nó una variable . El nombre de los arrays implican para el compilador el lugar de memoria donde empieza la estructura de datos por lo que , intentar cambiar su valor es tomado como un error , asI si escribieramos por ejemplo :

```
char titulo[] = "Primer titulo" ;

.....

titulo = "subtitulo" ;
```

La primer sentencia es correcta , ya que estamos incializando al string , pero la segunda produciría un error del tipo " LVALUE REQUERIDO " , es decir que el compilador espera ver , del lado izquierdo de una expresión , a una variable y en cambio se ha encontrado con una constante titulo (ó sea la dirección de memoria donde está almacenada la P de "Primer título") . Esto al compilador le suena similar a una expresión de la clase : 124 = j y se niega rotundamente a compilarla .

4. ARRAYS MULTIDIMENSIONALES.

Las estructuras de datos del tipo array pueden tener más de una dimensión , es bastante común el uso de arrays "planos" ó matriciales de dos dimensiones , por ejemplo :

```
int matriz[ número total de filas ] [ número total de columnas ] ;
```

Si declaramos :

```
int matriz[3][4] ;
```

esquematicamente la disposicion "espacial" de los elementos seria:

columnas:	0	1	2	3	
filas 0	[0][0]	[0][1]	[0][2]	[0][3]	matriz[0][]
1	[1][0]	[1][1]	[1][2]	[1][3]	matriz[1][]
2	[2][0]	[2][1]	[2][2]	[2][3]	matriz[2][]

Por supuesto , aunque menos usados , se pueden generar arrays de cualquier número de dimensiones .
Para inicializar arrays multidimensionales , se aplica una técnica muy similar a la ya vista , por ejemplo para dar valores iniciales a un array de caracteres de dos dimensiones , se escribirá :

```
char dia_de_la_semana[7][8] = {
                        "lunes" , "martes" , " miercoles" ,
                         "jueves" , "viernes" , "sábado" ,
                         "domingo"
                       } ;
```

Acá el elemento [0][0] será la "l" de lunes , el [2][3] la "r" de miercoles , el [5][2] la "b" de sabado, etc. Nótese que los elementos [0][5] , [1][6] ,etc estan inicializados con el caracter NULL y demas [0][6] y [0][7], etc no han sido inicializados. Si le parece que en este párrafo se nos escapó un error , está equivocado , lo que ocurre es que se olvidó de contar los índices desde 0.
Este último ejemplo también podría verse como un array unidimensional de strings.

5. ESTRUCTURAS

DECLARACION DE ESTRUCTURAS

Así como los arrays son organizaciones secuenciales de variables simples , de un mismo tipo cualquiera dado , resulta necesario en multiples aplicaciones , agrupar variables de distintos tipos , en una sola entidad . Este sería el caso , si quisieramos generar la variable " legajo personal " , en ella tendriamos que incluir variables del tipo : strings , para el nombre , apellido , nombre de la calle en donde vive , etc , enteros , para la edad , número de codigo postal , float (ó double , si tiene la suerte de ganar mucho) para el sueldo , y así siguiendo . Existe en C en tipo de variable compuesta , para manejar ésta situación típica de las Bases de Datos , llamada ESTRUCTURA . No hay limitaciones en el tipo ni cantidad de variables que pueda contener una estructura , mientras su máquina posea memoria suficiente como para alojarla , con una sóla salvedad : una

estructura no puede contenerse a sí misma como miembro .
Para usarlas , se deben seguir dos pasos . Hay que , primero declarar la estructura en sí , ésto es , darle un nombre y describir a sus miembros , para finalmente declarar a una ó más variables , del tipo de la estructura antedicha , veamos un ejemplo :

```
struct legajo {

        int edad ;

        char nombre[50] ;

        float sueldo ;

      } ;

struct legajo  legajos_vendedores , legajos_profesionales ;
```

En la primer sentencia se crea un tipo de estructura , mediante el declarador "struct",luego se le dá un nombre " legajo " y finalmente , entre llaves se declaran cada uno de sus miembros , pudiendo estos ser de cualquier tipo de variable , incluyendo a los arrays ó aún otra estructura . La única restricción es que no haya dos miembros con el mismo nombre , aunque sí pueden coincidir con el nombre de otra variable simple , (o de un miembro de otra estructura) , declaradas en otro lugar del programa. Esta sentencia es sólo una declaración , es decir que no asigna lugar en la memoria para la estructura , sólo le avisa al compilador como tendrá que manejar a dicha memoria para alojar variables del tipo struct legajo .
En la segunda sentencia , se definen dos variables del tipo de la estructura anterior ,(ésta definición debe colocarse luego de la declaración) , y se reserva memoria para ambas .
Las dos sentencias pueden combinarse en una sola , dando la definición a continuación de la declaracion :

```
struct legajo {

        int edad ;

        char nombre[50] ;

        float sueldo ;

      }   legajo_vendedor , legajo_programador ;
```

Y si nó fueran a realizarse más declaraciones de variables de éste tipo , podría obviarse el nombre de la estructura (legajo).
Las variables del tipo de una estructura , pueden ser inicializadas en su definición , así por ejemplo se podría escribir:

```
struct legajo {

        int edad ;

        char nombre[50] ;

        float sueldo ;
```

```
        char observaciones[500] ;

    }   legajo_vendedor = { 40 , "Juan Eneene" , 1200.50 ,

                                "Asignado a zona A"       } ;
struct legajo  legajo_programador = { 23 , "Jose Peres" , 2000.0 ,

                                "Asignado a zona B" } ;
```

Acá se utilizaron las dos modalidades de definición de variables , inicializandolas a ambas .

REGLAS PARA EL USO DE ESTRUCTURAS

Lo primero que debemos estudiar es el método para dirigirnos a un miembro particular de una estructura .Para ello existe un operador que relaciona al nombre de ella con el de un miembro , este operador se representa con el punto (.) , así se podrá referenciar a cada uno de los miembros como variables individuales , con las particularidades que les otorgan sus propias declaraciones , internas a la estructura.

La sintaxis para realizar ésta referencia es :

nombre_de_la_estructura.nombre_del_miembro , así podremos escribir por ejemplo , las siguientes sentencias

```
strut posicion_de {
float eje_x ;
float eje_y ;
float eje_z ;
} fin_recta , inicio_recta = { 1.0 , 2.0 , 3.0 ) ;
fin_recta.eje_x = 10.0 ;
fin_recta.eje_y = 50.0 ;
fin_recta.eje_z = 90.0 ;
if( fin_recta.eje_x == inicio_recta.eje_x )
.....
```

Es muy importante recalcar que , dos estructuras , aunque sean del mismo tipo , no pueden ser asignadas ó comparadas la una con la otra , en forma directa , sino asignando ó comparandolas miembro a miembro. Esto se vé claramente explicitado en las líneas siguientes , basadas en las declaraciones anteriores:

```
fin_recta = inicio_recta ;            /* ERROR */

if( fin_recta >>= inicio_recta );      /* ERROR */

fin_recta.eje_x = inicio_recta.eje_x ;     /* FORMA CORRECTA DE ASIGNAR */

fin_recta.eje_y = inicio_recta.eje_y ;     /* UNA ESTRUCTURA A OTRA     */

fin_recta.eje_z = inicio_recta.eje_z ;
if( (fin_recta.eje_x >>= inicio_recta.eje_x) &&     /* FORMA CORRECTA DE  */

   (fin_recta.eje_y >>= inicio_recta.eje_y) &&     /* COMPARAR UNA       */

   (fin_recta.eje_z >>= inicio_recta.eje_z) )      /* ESTRUCTURA CON OTRA */
```

Las estructuras pueden anidarse , es decir que una ó mas de ellas pueden ser miembro de otra . Las estructuras también pueden ser pasadas a las funciones como parámetros , y ser retornadas por éstas , como resultados .

6. ARRAYS DE ESTRUCTURAS

Cuando hablamos de arrays dijimos que se podían agrupar , para formarlos , cualquier tipo de variables , esto es extensible a las estructuras y podemos entonces agruparlas ordenadamente , como elementos de un array . Veamos un ejemplo :

```
typedef struct {

        char     material[50] ;

        int       existencia ;

        double costo_unitario ;

       } Item ;
Item  stock[100] ;
```

Hemos definido aquí un array de 100 elementos , donde cada uno de ellos es una estructura del tipo Item compuesta por tres variables , un int , un double y un string ó array de 50 caracteres.

Los arrays de estructuras pueden inicializarse de la manera habitual , así en una definición de stock, podríamos haber escrito:

```
Item   stock1[100] = {

            "tornillos"  , 120 , .15 ,

            "tuercas"    , 200 , .09 ,

            "arandelas"  ,  90 , .01

           } ;

Item   stock2[]    = {

             { 'i','t','e','m','1','\0' } , 10 , 1.5 ,

             { 'i','t','e','m','2','\0' } , 20 , 1.0 ,

             { 'i','t','e','m','3','\0' } , 60 , 2.5 ,

             { 'i','t','e','m','4','\0' } , 40 , 4.6 ,

             { 'i','t','e','m','5','\0' } , 10 , 1.2 ,

            } ;
```

Analicemos un poco las diferencias entre la dos inicializaciones dadas , en la primera , el array material[] es inicializado como un string , por medio de las comillas y luego en forma ordenada , se van inicializando cada uno de los miembros de los elementos del array stock1[] , en la segunda se ha preferido dar valores individuales a cada uno de los elementos del array material , por lo que es necesario encerrarlos entre llaves .

Sin embargo hay una diferencia mucho mayor entre las dos sentencias , en la primera explicitamos el tamaño del array , [100] , y sólo inicializamos los tres primeros elementos , los restantes quedarán cargados de basura si la definición es local a alguna

función , ó de cero si es global , pero de cualquier manera están alojados en la memoria , en cambio en la segunda dejamos implícito el número de elementos , por lo que será el compilador el que calcule la cantidad de ellos , basandose en cuantos se han inicializado , por lo tanto este array sólo tendrá ubicados en memoria cuatro elementos , sin posibilidad de agregar nuevos datos posteriomente .
Veremos más adelante que en muchos casos es usual realizar un alojamiento dinámico de las estructuras en la memoria , en razón de ello , y para evitar ademas el saturación de stack por el pasaje ó retorno desde funciones , es necesario conocer el tamaño , ó espacio en bytes ocupados por ella .
Podemos aplicar el operador sizeof , de la siguiente manera :

```
longitud_base_de_datos  = sizeof( stock1 ) ;

longitud_de_dato        = sizeof( Item )  ;

cantidad_de_datos       = sizeof( stock1 ) / sizeof( Item ) ;
```

Con la primera calculamos el tamaño necesario de memoria para albergar a todos datos, en la segunda la longitud de un sólo elemento (record) y por supuesto dividiendo ambas , se obtiene la cantidad de records.

7. UNIONES

Las uniones son a primera vista, entidades muy similares a las estructuras, están formadas por un número cualquiera de miembros, al igual que aquellas, pero en éste caso no existen simultaneamente todos los miembros, y sólo uno de ellos tendrá un valor válido.
Supongamos por caso, que queremos guardar datos para un stock de materiales , pero los mismos pueden ser identificados , en un caso con el número de articulo (un entero) y en otro por su nombre (un string de 10 letras como máximo). No tendría sentido definir dos variables , un int y un string , para cada artículo , ya que voy a usar una modalidad ú la otra, pero nó las dos simultaneamente. Las uniones resuelven este caso , ya que si declaro una que contenga dos miembros, un entero y un string , sólo se reservará lugar para el mayor de ellos , en estee caso, el string , de tal forma que si asigno un valor a éste se llenará ese lugar de la memoria con los caracteres correspondientes, pero si en cambio asigno un valor al miembro declarado como int éste se guardará en los dos primeros bytes del MISMO lugar de memoria. Por supuesto, en una unión, sólo uno de los miembros tendrá entonces un valor correcto .

CAPITULO 7: PUNTEROS (POINTERS)

1. INTRODUCCION A LOS PUNTEROS

Los punteros en el Lenguaje C , son variables que " apuntan " , es decir que poseen la dirección de las ubicaciones en memoria de otras variables, y por medio de ellos tendremos un poderoso método de acceso a todas ellas .
Quizás este punto es el más conflictivo del lenguaje , ya que muchos programadores en otros idiomas , y novatos en C , lo ven como un método extraño ó al menos desacostrumbrado , lo que les produce un cierto rechazo . Sin embargo , y en la medida que uno se va familiarizando con ellos , se convierten en la herramienta más cómoda y directa para el manejo de variables complejas , argumentos , parámetros , etc , y se empieza a preguntar como es que hizo para programar hasta aquí , sin ellos . La respuesta es que no lo ha hecho , ya que los hemos usado en forma encubierta , sin decir lo que eran . (Perdón por el pequeño engaño).
Veamos primero , como se declara un puntero :
tipo de variable apuntada *nombre_del_puntero ;

```
int *pint ;

double *pfloat ;

char  *letra , *codigo , *caracter ;
```

En estas declaraciones sólo decimos al compilador que reserve una posición de memoria para albergar la dirección de una variable , del tipo indicado , la cual será referenciada con el nombre que hayamos dado al puntero .
Obviamente , un puntero debe ser inicializado antes de usarse , y una de las eventuales formas de hacerlo es la siguiente:

```
int var1 ;     /* declaro ( y creo en memoria ) una variable entera ) */

int *pint ;    /*    "      "  "   "     "      un puntero que contendrá

                    la dirección de una variable entera */

pint = &var1 ;  /* escribo en la dirección de memoria donde está el

                    puntero la dirección de la variable entera */
```

Como habiamos anticipado en capítulos anteriores " &nombre_de_una_variable " implica la dirección de la misma . Si se pregunta porque no se usaron otros símbolos en vez de & y * , que se confunden con la Y lógica de bits y el producto , ... consuelese pensando que yo también me hice siempre esa pregunta . De cualquier manera es siempre obvio , en el contexto del programa el uso de los mismos .
Esquematicamente , lo que hemos hecho se puede simbolizar de la siguiente manera :
donde dentro del recuadro está el contenido de cada variable .

```
Pint          xxxxxx                    valor contenido por var1
              Dirección de var1

              yyyyyy (posición de memoria       xxxxxx (posición de memoria
```

ocupada por el puntero) ocupada por la variable)
En realidad , como veremos más adelante , en la declaración del puntero , está implicita otra información : cual es el tamaño (en bytes) de la variable apuntada.
El símbolo & , ó dirección , puede aplicarse a variables , funciones , etc , pero nó a constantes ó expresiones , ya que éstas no tienen una posición de memoria asignada.
La operación inversa a la asignación de un puntero , de referenciación del mismo , se puede utilizar para hallar el valor contenido por la variable apuntada . Así por ejemplo serán expresiones equivalentes :

```
y = var1 ;

y = *pint ;

printf("%d" , var1 ) ;

printf("%d" , *pint) ;
```

En estos casos , la expresión " *nombre_del_puntero " , implica " contenido de la variable apuntada por el mismo " . Veamos un corto ejemplo de ello :

```
#include <stdio.h>

main()

{

  char var1 ;    /*una variable del tipo caracter */

  char *pchar;      /* un puntero a una variable del tipo caracter */

  pc = &var1 ;   /*asignamos al puntero la direccion de la variable */

  for (var1 = 'a'; var1 <<= 'z'; var1++)

    printf("%c", *pchar) ;  /* imprimimos el valor de la variable apuntada */

  return 0 ;

}
```

Vemos acá , que en el FOR se incrementa el valor de la variable , y luego para imprimirla usamos la dereferenciación de su puntero.
El programa imprimirá las letras del abecedario de la misma manera que lo habría hecho si la sentencia del printf() huiera sido, printf("%c" , var1) .
Hay un error , que se comete con bastante frecuencia , y es cargar en la dirección

apuntada por un puntero a un tipo dado de variable , el contenido de otro tipo de las mismas , por ejemplo :

```
double d = 10.0 ;

int i = 7 , *pint ;

pint = &i ;

*pint = 10 ;          /* correcto,equivale a asignar a i el valor 10 */ ;

*pint = d ;          /* ERROR se pretende cargar en una variable entera un valor double */

pint = &d ;          /* INCORRECTO se pretende apuntar a una variable double con un
                     puntero declarado como apuntador a int */

pint = 4358 ;        /* ?????? */
```

El primer error , la asignación de un double , produce la pérdida de información dada por la conversión automática de tipo de variable , ya vista anteriormente , el segundo produce un llamado de atención rotulado como " asignación sospechosa de un pointer " . Resumiendo , las variables ó constantes cargadas por dereferenciación de un puntero , deben coincidir en tipo con la declaración de aquel .
La asignación de una constante a un pointer , y no a la variable apuntada por él , es un serio error , ya que debe ser el compilador , el encargado de poner en él el valor de la dirección , aquel así lo declara dando un mensaje de " conversión de puntero no transportable " . Si bien lo compila , ejecutar un programa que ha tenido esta advertencia es similar a jugar a la ruleta rusa , puede "colgarse" la máquina ó lo que es peor destruirse involuntariamente información contenida en un disco , etc.
Hay un sólo caso en el que esta asignación de una constante a un puntero es permitida , muchas funciones para indicar que no pueden realizar una acción ó que se ha producido un error de algun tipo , devuelven un puntero llamado "Null Pointer" , lo que significa que no apunta a ningun lado válido , dicho puntero ha sido cargado con la dirección NULL (por lo general en valor 0) , así la asignación : pint = NULL ; es válida y permite luego operaciones relacionales del tipo if(pint) ..... ó if(print != NULL) para convalidar la validez del resultado devuelto por una función .
Una advertencia : si bien volveremos más adelante sobre este tema , debemos desde ahora tener en cuenta que los punteros no son enteros , como parecería a primera vista , ya que el número que representa a una posición de memoria , sí lo es . Debido al corto alcance de este tipo de variable , algunos compiladores pueden , para apuntar a una variable muy lejana , usar cualquier otro tipo , con mayor alcance que el antedicho .

2. PUNTEROS Y ARRAYS

Hay una relación muy cercana entre los punteros y los arrays . Yá vimos previamente que el designador (ó nombre de un array) era equivalente a la dirección del elemento [0] del mismo . La explicación de ésto es ahora sencilla : el nombre de un array , para el compilador C , es un PUNTERO inicializado con la dirección del primer elemento del array . Sin embargo hay una importante diferencia entre ambos , que haremos notar más abajo.
Veamos algunas operaciones permitidas entre punteros :

ASIGNACION

```
float var1 , conjunto[] = { 9.0 , 8.0 , 7.0 , 6.0 , 5.0 );

float *punt ;

punt = conjunto ;    /* equivalente a hacer : punt = &conjunto [0] */

var1 = *punt ;

*punt = 25.1 ;
```

Es perfectamente válido asignar a un puntero el valor de otro , el resultado de ésta operación es cargar en el puntero punt la dirección del elemento [0] del array conjunto , y posteriormente en la variable var1 el valor del mismo (9.0) y para luego cambiar el valor de dicho primer elemento a 25.1 .
Veamos cual es la diferencia entre un puntero y el denominador de un array : el primero es una VARIABLE , es decir que puedo asignarlo , incrementarlo etc , en cambio el segundo es una CONSTANTE , que apunta siempre al primer elemento del array con que fué declarado , por lo que su contenido NO PUEDE SER VARIADO . Si lo piensa un poco , es lógico , ya que "conjunto" implica la dirección del elemento conjunto [0] , por lo que , si yo cambiara su valor , apuntaría a otro lado dejando de ser , "conjunto" .
Desde este punto de vista , el siguiente ejemplo nos muestra un tipo de error bastante frecuente:
ASIGNACION ERRONEA

```
int conjunto[5] , lista[] = { 5 , 6 , 7 , 8 , 0 ) ;

int *apuntador ;

apuntador = lista ;      /* correcto */

conjunto = apuntador;    /* ERROR ( se requiere en Lvalue no constante ) */

lista = conjunto ;       /* ERROR  ( idem ) */

apuntador = &conjunto    /* ERROR no se puede aplicar el operador & (dirección) a
                            una constante */
```

Veamos ahora las distintas modalidades del incremento de un puntero :
INCREMENTO O DECREMENTO DE UN PUNTERO

```
int *pint , arreglo_int[5] ;

double *pdou , arreglo_dou[6] ;

pint = arreglo_int ;      /* pint apunta a arreglo_int[0] */

pdou = arreglo_dou ;      /* pdou apunta a arreglo_dou[0] */

pint += 1 ;               /* pint apunta a arreglo_int[1]  */
```

```
pdou += 1 ;            /* pdou apunta a arreglo_dou[1]  */

pint++ ;               /* pint apunta a arreglo_int[2] */

pdou++ ;               /* pdou apunta a arreglo_dou[2] */
```

Hemos declarado y asignado dos punteros , uno a int y otro a double , con las direcciones de dos arrays de esas caracteristicas . Ambos estarán ahora apuntando a los elementos [0] de los arrays . En las dos instrucciones siguientes incrementamos en uno dichos punteros . ¿ adonde apuntaran ahora ?.
Para el compilador , éstas sentencias se leen como : incremente el contenido del puntero (dirección del primer elemento del array) en un número igual a la cantidad de bytes que tiene la variable con que fué declarado . Es decir que el contenido de pint es incrementado en dos bytes (un int tiene 2 bytes) mientras que pdou es incrementado 8 bytes (por ser un puntero a double) , el resultado entonces es el mismo para ambos , ya que luego de la operación quedan apuntando al elemento SIGUIENTE del array , arreglo_int[1] y arreglo_dou[1] .
Vemos que de ésta manera será muy facil "barrer" arrays , independientemente del tamaño de variables que lo compongan , permitiendo por otro lado que el programa sea transportable a distintos hardwares sin preocuparnos de la diferente cantidad de bytes que pueden asignar los mismos , a un dado tipo de variable .
De manera similar las dos instrucciones siguientes , vuelven a a incrementarse los punteros , apuntando ahora a los elementos siguientes de los arrays.
Todo lo dicho es aplicable , en identica manera , al operador de decremento -- .

ARITMETICA DE DEREFERENCIA

Debido a que los operadores * y ++ ó -- tienen la misma precedencia y se evaluan de derecha a izquierda , y los paréntesis tienen mayor precedencia que ambos , muchas operaciones que los utilizan en conjunto a todos estos operadores , pueden parecer poco claras y dar origen a un sinnúmero de errores , (revise un poco la TABLA 13 del capítulo 3) analicémoslas detalladamente , partiendo de :

```
int *p , a[] = { 0 , 10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 } ;

int var ;

p = a ;
```

A partir de aquí , el puntero está apuntando a a[0] . Veamos las distintas variantes que puede tener la siguiente instrucción:

```
*p = 27 ;
```

La más sencilla de las opciones , simplemente asignamos al elemento apuntado por p (a[0]) un valor constante . Veamos la inversa de ella:

```
var = *p ;
```

var sería asignada al valor 0 (contenido de a[0]) , y p seguiría apuntando al mismo elemento. Que hubiera pasado, si en vez de ello se hubiera escrito:

```
var = *( p + 1 ) ;
```

acá podríamos traducir el sentido de la operación como : cargue var con el contenido del elemento siguiente al apuntado por p (a[1]) . Lo interesante de remarcar acá es que p , en sí mismo , NO VARIA Y LUEGO DE ESTA SENTENCIA SEGUIRA APUNTANDO A a[0] . De la misma forma : var = *(p + 3) asignará 30 a var , sin

modificar el contenido de p .
En cambio la expresión :
var = *(p++) ;
podemos leerla como : asigne a var el valor de lo apuntado por p y LUEGO incremente éste para que apunte al proximo elemento . Así en var quedaría 0 (valor de a[0]) y p apuntaría finalmente a a[1] . Si en vez de ésto hubieramos preincrementado a p tendríamos :
var = *(++p) ;
la que puede leerse como : apunte con p al próximo elemento y asigne a var con el valor de éste . En este caso var sería igualada a 10 (a[1]) y p quedaría apuntando al mismo .
En las dos operaciones anteriores los paréntesis son superfluos ya que al analizarse los operadores de derecha a izquierda , daría lo mismo escribir :
var = *p++ ; /* sintácticamente igual a var = *(p++) */

var = *++p ; /* " " " var = *(++p) */

3. ARITMETICA DE PUNTEROS

La aritmética más frecuentemente usada con punteros son las sencillas operaciones de asignación , incremento ó decremento y dereferenciación . Todo otro tipo de aritmética con ellos está prohibida ó es de uso peligroso ó poco transportable . Por ejemplo no está permitido , sumar , restar , dividir , multiplicar , etc , dos apuntadores entre sí . Lo cual si lo pensamos un poco es bastante lógico , ya que de nada me serviría sumar dos direcciones de memoria , por ejemplo .
Otras operaciones estan permitidas , como la comparación de dos punteros , por ejemplo (punt1 == punt2) ó (punt1 < punt2) sin embargo este tipo de operaciones son potencialmente peligrosas , ya que con algunos modelos de pointers pueden funcionar correctamente y con otros no .

4. PUNTEROS Y VARIABLES DINAMICAS

Recordemos lo expresado en capítulo 5 , sobre el ámbito ó existencia de las variables , la menos duradera de ellas era la del tipo local a una función , ya que nacía y moría con ésta . Sin embargo , esto es algo relativo , en cuanto a la función main() , ya que sus variables locales ocuparán memoria durante toda la ejecución del programa.
Supongamos un caso típico , debemos recibir una serie de datos de entrada , digamos del tipo double , y debemos procesar según un determinado algoritmo a aquellos que aparecen una ó más veces con el mismo valor .
Si no estamos seguros de cuantos datos van a ingresar a nuestro programa , pondremos alguna limitación , suficientemente grande a los efectos de la precisión requerida por el problema , digamos 5000 valores como máximo , debemos definir entonces un array de doubles capaz de albergar a cinco mil de ellos , por lo que el mismo ocupará del orden de los 40 k de memoria .
Si definimos este array en main() , ese espacio de memoria permanecerá ocupado hasta el fín del programa , aunque luego de aplicarle el algoritmo de cálculo ya no lo necesitemos más , comprometiendo seriamente nuestra disponibilidad de memoria para albergar a otras variables . Una solución posible sería definirlo en una función llamada por main() que se ocupara de llenar el array con los datos , procesarlos y finalmente devolviera algún tipo de resultado , borrando con su retorno a la masiva variable de la memoria .
Sin embargo en C existe una forma más racional de utilizar nuestros recursos de memoria de manera conservadora . Los programas ejecutables creados con estos compiladores dividen la memoria disponible en varios segmentos , uno para el código (en lenguaje máquina) , otro para albergar las variables globales , otro para el stack (a

travez del cual se pasan argumentos y donde residen las variables locales) y finalmente un último segmento llamado memoria de apilamiento ó amontonamiento (Heap) .
El Heap es la zona destinada a albergar a las variables dinámicas , es decir aquellas que crecen (en el sentido de ocupación de memoria) y decrecen a lo largo del programa , pudiendose crear y desaparecer (desalojando la memoria que ocupaban) en cualquier momento de la ejecución .
Veamos cual sería la metodología para crearlas ; supongamos primero que queremos ubicar un único dato en el Heap , definimos primero un puntero al tipo de la variable deseada :

```
double *p ;
```

notemos que ésta declaración no crea lugar para la variable , sino que asigna un lugar en la memoria para que posteriormente se guarde ahí la dirección de aquella Para reservar una cantidad dada de bytes en el Heap , se efectua una llamada a alguna de las funciones de Librería , dedicadas al manejo del mismo . La más tradicional es malloc() (su nombre deriva de memory allocation) , a esta función se le dá como argumento la cantidad de bytes que se quiere reservar , y nos devuelve un pointer apuntando a la primer posición de la "pila" reservada . En caso que la función falle en su cometido (el Heap está lleno) devolvera un puntero inicializado con NULL .

```
p = malloc(8) ;
```

acá hemos pedido 8 bytes (los necesarios para albergar un double) y hemos asignado a p el retorno de la función , es decir la dirección en el Heap de la memoria reservada.
Como es algo engorroso recordar el tamaño de cada tipo variable , agravado por el hecho de que , si reservamos memoria de esta forma , el programa no se ejecutará correctamente , si es compilado con otro compilador que asigne una cantidad distinta de bytes a dicha variable , es más usual utilizar sizeof , para indicar la cantidad de bytes requerida :

```
p = malloc( sizeof(double) ) ;
```

En caso de haber hecho previamente un uso intensivo del Heap , se debería averiguar si la reserva de lugar fué exitosa:

```
if( p == NULL )

  rutina_de_error() ;
```

si no lo fué estas sentencias me derivan a la ejecución de una rutina de error que tomará cuenta de este caso . Por supuesto podría combinar ambas operaciones en una sola ,

```
if( ( p = malloc( sizeof(double) ) ) == NULL ) {

   printf("no hay mas lugar en el Heap ..... Socorro !!" ) ;

   exit(1) ;

}
```

se ha reemplazado aquí la rutina de error , por un mensaje y la terminación del programa , por medio de exit() retornando un código de error .
Si ahora quisiera guardar en el Heap el resultado de alguna operación , sería tan directo como,

```
*p  =  a * ( b + 37 ) ;
```

y para recuperarlo , y asignarselo a otra variable bastaría con escribir :

```
var = *p ;
```

5. PUNTEROS A STRINGS

No hay gran diferencia entre el trato de punteros a arrays , y a strings , ya que estos dos últimos son entidades de la misma clase . Sin embargo analicemos algunas particularidades . Así como inicializamos un string con un grupo de caracteres terminados en '\0' , podemos asignar al mismo un puntero :
p = "Esto es un string constante " ;
esta operación no implica haber copiado el texto , sino sólo que a p se le ha asignado la dirección de memoria donde reside la "E" del texto . A partir de ello podemos manejar a p como lo hemos hecho hasta ahora . Veamos un ejemplo

---

```
#include <stdio.h>

#define TEXTO1  "¿ Hola , como "

#define TEXTO2  "le va a Ud. ? "

main()

{

char palabra[20] , *p ;

int i ;

p = TEXTO1 ;

for( i = 0 ; ( palabra[i] = *p++ ) != '\0' ; i++ ) ;

p = TEXTO2 ;

printf("%s" , palabra ) ;

printf("%s" , p ) ;

return 0 ;

}
```

Definimos primero dos strings constantes TEXTO1 y TEXTO2 , luego asignamos al puntero p la dirección del primero , y seguidamente en el FOR copiamos el contenido de éste en el array palabra , observe que dicha operación termina cuando el contenido de lo apuntado por p es el terminador del string , luego asignamos a p la dirección de TEXTO2 y finalmente imprimimos ambos strings , obteniendo una salida del tipo : " ¿ Hola , como le va a UD. ? " (espero que bien) .
Reconozcamos que esto se podría haber escrito más compacto, si hubieramos recordado que palabra tambien es un puntero y NULL es cero , así podemos poner en vez del FOR
while(*palabra++ = *p++) ;

Vemos que aquí se ha agregado muy poco a lo ya sabido , sin embargo hay un tipo de error muy frecuente , que podemos analizar , fíjese en el EJEMPLO siguiente , ¿ ve algun problema ? .
(CON ERRORES)

```
#include <stdio.h>

char *p , palabra[20] ;

printf("Escriba su nombre : ") ;

scanf("%s" , p ) ;

palabra = "¿ Como le va " ;

printf("%s%s" , palabra , p ) ;

}
```

Pues hay dos errores , a falta de uno , el primero ya fue analizado antes , la expresión scanf("%s" , p) es correcta pero , el error implícito es no haber inicializado al puntero p , el cual sólo fué definido , pero aun no apunta a ningun lado válido . El segundo error está dado por la expresión : palabra = " ¿ Como le va " ; (también visto anteriormente) ya que el nombre del array es una constante y no puede ser asignado a otro valor . ¿Como lo escribiríamos para que funcione correctamente ?
(CORRECTO)

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char *p , palabra[20] ;

p = (char *)malloc(sizeof(char)128) ;

printf("Escriba su nombre : ") ;

scanf("%s" , p ) ;

strcpy(palabra , "¿ Como le va " ) ;

printf("%s%s" , palabra , p ) ;

}
```

Observe que antes de scanf() se ha inicializado a p, mediante el retorno de malloc() y a al array palabra se le copiado el string mediante la función vista anteriormente strcpy(). Debemos aclarar también que, la secuencia de control %s en el printf() impone enviar a

la pantalla un string, estando éste apuntado por el argumento siguiente al control, éste puede ser tanto el nombre de un array, como un puntero, ya que ambos explicitan direcciones.
Una forma alternativa de resolverlo , sería:
(CORRECTO)

```
#include <stdio.h>

main()

{

char p[20] , *palabra ;

printf("Escriba su nombre : ") ;

scanf("%s" , p ) ;

palabra = "¿ Como le va " ;

printf("%s%s" , palabra , p ) ;

}
```

Obsérvese , que es idéntico al primero , con la salvedad que se ha invertido las declaraciones de las variables , ahora el puntero es palabra y el array es p . Ambas soluciones son equivalentes y dependerá del resto del programa , cual es la mejor elección .

6. ARRAYS DE PUNTEROS

Es una práctica muy habitual , sobre todo cuando se tiene que tratar con strings de distinta longitud , generar array cuyos elementos son punteros , que albergarán las direcciones de dichos strings.
Si imaginamos a un puntero como una flecha , un array de ellos equivaldría a un carcaj indio lleno de aquellas .
Asi como:
char *flecha;
definía a un puntero a un caracter , la definición
char *carcaj[5];
implica un array de 5 punteros a caracteres .

INICIALIZACION DE ARRAYS DE PUNTEROS

Los arrays de punteros pueden ser inicializados de la misma forma que un array común , es decir dando los valores de sus elementos , durante su definición , por ejemplo si quisieramos tener un array donde el subíndice de los elementos coincidiera con el nombre de los días de la semana , podríamos escribir :

```
char *dias[] = {

        "número de día no válido" ,

        "lunes"                   ,
```

```
"martes"        ,

"miercoles"     ,

"jueves"        ,

"viernes"       ,

"sabado"        ,

"por fín es domingo"

}
```

Igual que antes, no es necesario en este caso indicar la cantidad de elementos , ya que el compilador los calcula por la cantidad de términos dados en la inicialización. Asi el elemento dias[0] será un puntero con la dirección del primer string, dias[1], la del segundo, etc.

7. PUNTEROS A ESTRUCTURAS

Los punteros pueden también servir para el manejo de estructuras , y su alojamiento dinámico , pero tienen además la propiedad de poder direccionar a los miembros de las mismas utilizando un operador particular , el -> , (escrito con los símbolos "menos" seguido por "mayor") .

Supongamos crear una estructura y luego asignar valores a sus miembros , por los métodos ya descriptos anteriormente :

```
struct  conjunto {

        int a     ;

        double b  ;

        char c[5] ;

       } stconj    ;
stconj.a  = 10     ;

stconj.b  = 1.15   ;

stconj.c[0]  = 'A' ;
```

La forma de realizar lo mismo , mediante el uso de un puntero, sería la siguiente :

```
struct  conjunto {

        int a     ;

        double b  ;

        char c[5] ;

       } *ptrconj    ;
```

```
ptrconj = (struct conjunto *)malloc( sizeof( struct conjunto )) ;

ptrconj->a  = 10     ;

ptrconj->b  = 1.15   ;

ptrconj->c[0]  = 'A' ;
```

En este caso vemos que antes de inicializar un elemento de la estructura es necesario alojarla en la memoria mediante malloc(), observe atentamente la instrucción: primero se indica que el puntero que devuelve la función sea del tipo de apuntador a conjunto (ésto es sólo formal), y luego con sizeof se le da como argumento las dimensiones en bytes de la estructura.
Acá se puede notar la ventaja del uso del typedef , para ahorrar tediosas repeticiones de texto, y mejorar la legilibilidad de los listados; podríamos escribir:

```
typedef struct   {

                int a     ;

                double b  ;

                char c[5] ;

               }  conj    ;

conj *ptrconj ;

ptrconj = ( conj *)malloc( sizeof( conj )) ;
```

Es muy importante acá , repasar la TABLA 13 del final del capítulo 3 , donde se indican las precedencias de los operadores , a fin de evitar comportamientos no deseados , cuando se usan simultaneamente varios de ellos .
Ya que c es un array podemos escribir :

```
x = *ptrconj -> c ;
```

la duda acá es, si nos referimos al contenido apuntado por ptrconj ó por c.
Vemos en la tabla que, el operador -> es de mayor precedencia que la de * (dereferenciación), por lo que, el resultado de la expresión es asignar el valor apuntado por c, es decir el contenido de c[0] .
De la misma forma:

```
*ptrconj -> c++ ;  incrementa el puntero c , haciendolo tener la direccion
                   de c[1] y luego extrae el valor de éste .

++ptrconj -> c ;  incrementa el valor de c[0] .
```

En caso de duda , es conveniente el uso a discreción de paréntesis , para saltar por sobre las , a veces complicadas , reglas que impone la precedencia así , si queremos por ejemplo el valor de c[3] , la forma más clara de escribir es:

```
*( ptrconj -> ( c + 4 ) ) ;
```

(Recuerde que c[3] es el CUARTO elemento del array).

8. PUNTEROS Y FUNCIONES

La relación entre los punteros y las funciones , puede verse en tres casos distintos ,

podemos pasarle a una función un puntero como argumento (por supuesto si su parámetro es un puntero del mismo tipo) , pueden devolver un puntero de cualquier tipo , como ya hemos visto con malloc() y calloc() , y es posible también apuntar a la dirección de la función , en otras palabras , al código en vez de a un dato.

PUNTEROS COMO PARAMETROS DE FUNCIONES .

Supongamos que hemos declarado una estructura , se puede pasar a una función como argumento , de la manera que ya vimos anteriormente:

```
struct  conjunto {

            int a     ;

            double b  ;

            char c[5] ;

           } datos    ;
```

```
void una_funcion( struct conjunto datos );
```

Hicimos notar, en su momento, que en este caso la estructura se copiaba en el stack y así era pasada a la función, con el peligro que esto implicaba, si ella era muy masiva, de agotarlo.

Otra forma equivalente es utilizar un puntero a la estructura :

```
struct  conjunto {

            int a     ;

            double b  ;

            char c[5] ;

           } *pdatos    ;
```

```
void una_funcion( struct conjunto *pdatos ) ;
```

Con lo que sólo ocupo lugar en el stack para pasarle la dirección de la misma. Luego en la función, como todos los miembros de la estructuras son accesibles por medio del puntero, tengo pleno control de la misma.

Un ejemplo de funciones ya usadas que poseen como parámetros a punteros son:

```
scanf(puntero_a_string_de_control , punteros_a_variables)
printf(puntero_a_string_de_control , variables )
```

En ambas vemos que los strings de control son , como no podría ser de otro modo , punteros , es decir que los podríamos definir fuera de la función y luego pasarselos a ellas :

```
p_control = "valor : %d " ;

printf( p_control , var ) ;
```

PUNTEROS COMO RESULTADO DE UNA FUNCION

Las funciones que retornan punteros son por lo general aquellas que modifican un argumento , que les ha sido pasado por dirección (por medio de un puntero) , devolviendo un puntero a dicho argumento modificado , ó las que reservan lugar en el

Heap para las variables dinámicas , retornando un puntero a dicho bloque de memoria . Así podremos declarar funciónes del tipo de:

```
char *funcion1( char * var1 ) ;

double *funcion2(int i , double j , char *k ) ;

struct item *funcion3( struct stock *puntst ) ;
```

El retorno de las mismas puede inicializar punteros del mismo tipo al devuelto , ó distinto , por medio del uso del casting . Algunas funciones , tales como malloc() y calloc() definen su retorno como punteros a void :

```
void *malloc( int tamano ) ;
```

de esta forma al invocarlas , debemos indicar el tipo de puntero de deseamos

```
p = (double *)malloc( 64 ) ;
```

CAPITULO 8: FUNCIONES DE MANEJO DE STRINGS

1. INTRODUCCION

Si bien ya hemos realizado variadas operaciones de manejo de string , dada su importancia, pues son cuando menos el medio de comunicación de los programas con el operador, trataremos acá de sintetizar los conceptos relativos a los mismos, y resumir aquellas funciones ya vistas, con el agregado de otras nuevas.
La mayoría de las que veremos a continuación, responden a la norma ANSI C, por lo que serán independientes del compilador que usemos. Estas tienen sus prototipos definidos en los archivos de encabezamiento stdio.h, stdlib.h, string.h y ctype.h.
Agregaremos tambien algunas que caen fuera de la norma, por lo que su portabilidad a otros compiladores distintos al que fueron extraidas, no es segura. Seran aquellas declaradas en Headers no citados arriba. Sin embargo, hoy en día practicamente todos los compiladores las traen ó tienen otras similares, con nombres parecidos. De cualquier forma, antes de compilar los ejemplos aquí suministrados, en caso de encontrarse alguna de estas, verifique con su manual de Libreria la existencia y compatibilidad de la misma.
Refresquemos, antes de comenzar, algunas de las características básicas de los strings. Estos pueden aparecer en un programa como una constante simbólica, de la forma siguiente:

```
#define TITULO "Capitulo 9"
```

en este caso, en cada lugar donde aparece TITULO se reemplazará esta constante simbólica por la DIRECCION de la C del texto con que fué definida .
Así, será correcto escribir:

```
char *p = TITULO ;
```

Recordemos tambien que en la memoria, el string se guardará de la siguiente forma:

67	97	112	105	116	117	108	111	20	57	0

Donde los números son el código ASCII que representa a cada caracter del string , en particular , note que 20 corresponde al espacio , terminandose con un NULL (código 0) .
A los efecttos prácticos, para las funciones de manejo de los mismos, es como si en realidad hubieramos memorizados directamente los caracteres:

C	a	p	i	t	u	l	o		9	/0

El código ASCII de los caracteres imprimibles vá entre el 31 y el 127 , reservándose los códigos entre el 0 y 30 , para los caracteres de control (retorno de carro, avance de linea, tabulador, etc).
Si en cambio , hubieramos escrito el string de una manera ortográficamente más correcta :

```
#define TITULO "Capítulo 9"
```

(con la i acentuada) estaríamos introduciendo un caracter del conjunto ASCII Extendido , ya que su código supera a 127 y está representado por 173 .
Lo correcto en este caso sería definir , aunque muchos compiladores ya lo presuponen por omisión, para asegurar la portabilidad :

```
unsigned char *p = TITULO ;
```

de esta forma se garantiza que el alcance de la variable sea de 255 , ó en su defecto :

```
int *p = TITULO ;
```
Es correcto tambien declarar el puntero , y asignarlo posteriormente
```
char *p ;

p = TITULO ;
```
Esta asignación solo dá , al contenido del puntero la dirección del string global predefinido .
Sin embargo , si en lugar de un puntero deseamos usar un array , en este caso es correcta la inicialización del mismo , pero no así su asignación posterior:
```
char nombre[] = TITULO ;        /* Correcto */

.....

char nombre[11] ;

nombre = TITULO ;               /* Incorrecto */
```
Ya que si bien, el nombre de un array es un puntero , es de índole constante , negándose el compilador a cambiar su dirección.
Si estuvieramos en el caso de ingresar un string variable , por ejemplo leyendolo desde el teclado , podríamos utilizar un array, de la siguiente forma :
```
char nombre[128] ;

scanf("%s" , nombre ) ;
```
en este caso la única precaución es que , el array tenga suficiente longitud para albergar a cualquier string escrito . En el caso de trabajar bajo DOS, basta con darle 128 caracteres, ya que el buffer de lectura de ese sistema operativo no supera dicha cantidad .
Hay que hacer notar que la longitud de un string puede ser mayor que la del texto válido contenido , ya que este termina donde se encuentra el NULL , quedando los bytes sobrantes desaprovechados .
Seria incorrecto leer este string mediante un puntero declarado , pero al que no se le ha reservado memoria:
```
char *p ;

scanf("%s" , p )      /* Incorrecto */
```
ya que la dirección contenida por p no ha sido inicializada aún con ningun valor válido .
Lo correcto en éste caso es:
```
char *p ;

p = (char *)malloc(128 * sizeof(char)) ;

scanf("%s" , p )                /* Correcto */
```
reservando memoria previamente a cargar el string.
Otro punto sobre el que quiero volver, a fín de evitar confusiones, es el sentido de la constante NULL , y el de variables nulas.
Segun éste se aplique a caracteres, strings ó punteros, su significado varia:

- Un caracter nulo tiene el valor ASCII cero. Un string siempre estará terminado por un caracter NULL .
- Un string nulo ó vacio, no tiene longitud cero, sino que su primer caracter es un NULL .

- Un puntero nulo, no corresponde a un string vacio, sinó que su contenido ha sido asignado a la dirección 0 ó NULL, es decir que no apunta a ningun string aún.

Hay que recalcar que, practicamente todas las funciones que describiremos a continuación , basan su operatoria en la suposición que los strings que se le pasan como argumento , terminan en el caracter NULL , si por error esto no fuera así , los resultados son catastróficos , produciendose generalmente la destrucción de los datos y el aborto del programa .

2. FUNCIONES DE IMPRESION DE STRINGS

Daremos un análisis de las funciones que permiten la impresión en pantalla de strings , muchas de ellas pueden obviamente , utilizarse para imprimir otro tipo de variable , pero aquí sólo describiremos su aplicación particular sobre el tema de nuestro interes.

PRINTF()

- Header : <stdio.h>
- Prototipo : int printf(const char *formato , argumento , ...)
- Portabilidad : Definida en ANSI C. No es compatible con Windows
- Comentario : Retorna un entero igual a la cantidad de caracteres que ha impresso , ó un EOF (End Of File , por lo general -1) en caso de error ú operación fallida . Tiene un uso más generalizado que el que aquí describimos , ya que por el momento veremos sólo su aplicación a strings . El string de formato puede ser construido directamente en la función , delimitándolo con comillas , ó definido antes en el listado , y pasado como argumento por medio de un puntero . Puede contener , directamente el texto a imprimir , si éste es una constante , en cuyo caso no se pasarán más argumentos , ó una mezcla de texto constante con secuencias de control de formato para la impresión del resto de los parámetros pasados. La secuencia de control comienza con el caracter % . En caso de impresión de strings , el comando debe terminarse con la letra s. Entre el comienzo (%) y el fín (s) de la secuencia de control , pueden introducirse opcionalmente modificadores cuyo sentido , en el caso de los strings , es el siguiente : % [justificación] [longitud] [.presicion] s La (longitud) dá la cantidad MINIMA de caracteres a imprimir , independientemente de cuantos caracteres tenga el string . Si este valor es mayor que la cantidad de caracteres del string , se rellenará con blancos el sobrante , colocandose los mismos a la derecha ó izquierda , segun sea la justificación . Si la (longitud) es menor que la del string , este quedará truncado . La precisión es un número , que debe estar precedido por un punto , e indica el máximo número de caracteres del string , que se imprimirán . La justificación "default" es hacia la derecha , un signo menos en este campo impone una justificacion a la izquierda . Dentro de la secuencia de comando pueden ubicarse secuencias de escape como las vistas en capítulos anteriores . Un caso especial se dá , cuando en el lugar de la longitud se coloca un asterisco (*) , éste implica que la longitud vendrá expresada por el argumento que sigue al string en la lista de los mismos (un entero) .
- Ejemplos : En los ejemplos siguientes se ha colocado el caracter | adelante y atras de la secuencia de comando , para mostrar donde empieza y donde termina la impresión del string .

```
.....

p = "Lenguaje C" ;    /* 10 caracteres */
```

```
.....

printf("|%15s|" , p )     ;    /* imprime : |     Lenguaje C|  */

printf("|%15.8s|" , p )   ;    /*    "    : |       Lenguaje|  */

printf("|%-15s|" , p )    ;    /*    "    : |Lenguaje C     |  */

printf("|%-15.8s|" , p )  ;    /*    "    : |Lenguaje       |  */

printf("|%.6s|" , p )     ;    /*    "    : |Lengua|          */

ancho = printf("|%15s|" , p ); /* imprime : |     Lenguaje C|  */

printf("|%*.8s|" , p , ancho); /*    "    : |       Lenguaje|  */
```

Existe otra función más específica que la anterior , aunque más restrigida , puts() .
PUTS()

- Header : <stdio.h>
- Prototipo : int puts(const char *s)
- Portabilidad : Definida en ANSI C. No es compatible con Windows
- Comentario : Copia un string terminado con un NULL y apuntado por s en la salida estandar , normalmente stdout ó video . Si la salida fué exitosa retorna un valor positivo , caso contrario EOF . Luego de impreso el string agrega automaticamente un \n ó avance de línea . Es más rápida que la anterior , ya que escribe directamente el buffer de video . Solo es aplicable a variables del tipo strings
- Ejemplo :

```
#include <stdio.h>

main()

{

char p[] = "Uno" , s[] = "Dos" ;

puts(p) ;

puts(s) ;

}

/* imprime :    Uno

                Dos       */
```

---

3. FUNCIONES DE ADQUISICION DE STRING
Cuando se necesita leer un string enviado desde el teclado , se utilizará alguna de las

funciones abajo citadas , debiendose tener los recaudos descriptos antes , ya que la longitud del mismo es desconocida.

SCANF()

- Header : <stdio.h>
- Prototipo : int scanf(const char *formato , direccion , ...)
- Portabilidad : Definida en ANSI C. No es compatible con Windows
- Comentario : Esta función es la inversa del printf() y valen para ella los mismos comentarios respecto a su generalidad. La analizaremos desde el punto de vista exclusivo de los strings . Las secuencias de control dentro del string de formato , comenzarán con % y terminarán con s , siendo optativo colocar entre ambas los siguientes modificadores: % [*] [longitud] [N ó F] s El * en éste caso suprime el asignamiento del valor ingresado a la variable. Longitud es un entero que indica el máximo de caracteres a cargar en la dirección dada por el puntero . N ó F intruyen a la función a forzar al puntero entregado como parámetro a ser Near ó Far . Se pueden poner varios especificadores de formato seguidos, en ese caso , la cantidad de argumentos debe coincidir co la de aquellos , ó por lo menos no ser menor, ya que si así fuera los resultados son impredecibles y por lo general desastrosos . En cambio , si la cantidad es mayor, el excedente sera simplemente no tomado en cuenta . Si se separan los especificadores con caracteres de distanciameinto no imprimibles como , espacio , \t , \n , etc , la función esperará a que el correspondiente sea ingresado por el teclado , una vez ubicado el proximo caracterer imprimible será enviado a la dirección apuntada por el próximo parámetro , descartando los separadores anteriores . Si en cambio se usan para separar dos especificadores de formato , caracteres imprimibles , como dos puntos , coma , etc , estos serán leidos y descartados. Una condición particular puede darse en el caso de los strings : se puede especificar los caracteres ó grupos de caracteres que se desea leer , si luego del % y encerrado entre corchetes [] se coloca un grupo de ellos , solo serán enviados a la dirección del parámetro , aquellos que coincidan con los mismos . Por ejemplo %[0123456789]s solo leerá los caracteres numéricos . Esto se puede expresar , en forma más compacta como %[0-9]s Si en cambio se desea EXCLUIR dichos caracteres debrá escribirse : %[^0-9]s , indicandose la exclusión mediante el simbolo ^ . El uso anterior al de esta función , de alguna otra que lea el teclado pude dejar el buffer del mismo cargado con caracteres que luego afectan a scanf como si nuevamente hubieran sido escritos , en estos casos se impone el uso previo a la llamada a ésta función de fflush() . Retorna un entero de igual valor al de campos leidos exitosamente.
- Ejemplo : scanf("%20s" \n "%[0-9A-F]s" , p , q) ; En éste caso se leerá un primer string de nó más de 20 caracteres, y se enviará a la dirección contenida por el puntero p , luego se esperará un ENTER y se enviarán a la dirección de q todos aquellos caracteres leidos que correspondan a los números ó a letras comprendidas entre la A y F.

De la misma manera que para printf(), hay funciones menos generales, dedicadas expresamente a la lectura de strings, como gets(), que veremos a continuación .

GETS()

- Header : <stdio.h>
- Prototipo : char *gets(char *s)

- Portabilidad : Definida en ANSI C. No es compatible con Windows
- Comentario : Lee caracteres desde la entrada estandar , incluyendo los espacios hasta que encuentra un avance de linea (ENTER), este es reemplazado por un NULL, y el string resultante en cargado en la direccion indicada por s. Retorna s, ó un NULL en caso de error. Es responsabilidad del programador, que s tenga suficiente longitud como para albergar lo leído.

---

4. FUNCIONES DE CONVERSION ENTRE STRING Y VARIABLES NUMERICAS

Puede darse el caso que la información a ingresarse a un programa ejecutable , por el operador pueda ser en algunos caso un valor numérico y en otros un string de caracteres . Un problema típico enfrentamos en el ejemplo en que ingresabamos a nuestra base de datos un articulo , ya sea por su nombre ó por su número de código .
Más cómodo que escribir dos instancias del programa , una para cada una de las opciones , es dejar que el operador escriba lo que se le venga en ganas , leyendolo como un string , luego verificar si éste está compuesto exclusivamente por números ó posee algun caracter nó numérico , y actuar en consecuencia .
Para evaluar si un string cae dentro de una categoría dada , es decir si está compuesto exclusivamente por números , letras, mayúsculas , minúsculas , caracteres alfanuméricos , etc existen una serie de funciones , algunas de las cuales ya hemos usado, que describimos a continuación . Estas deben ser usadas con los strings , analizando caracter a caracter de los mismos, dentro de un FOR ó un WHILE:

```
for(i=0 ; palabra[i] != NULL ; i++) {

 if( isalnum(palabra[i] )

  .....

 }
```

IS.....()

- Header : <ctype.h>
- Prototipo :

```
int isalnum( int  c )

int isalpha( int  c )

int isascii( int  c )

int iscntrl( int  c )

int isdigit( int  c )

int islower( int  c )

int isupper( int  c )

int ispunct( int  c )
```

int isspace(int c)

int isxdigit(int c)

- Portabilidad : Definida en ANSI C
- Comentario : Retornarán un valor CIERTO (distinto de cero) si el caracter enviado como argumento cae dentro de la catewgoría fijada para la comparación y FALSO ó cero en caso contrario . Las categorias para cada función son las siguientes :

La Función	Retorna CIERTO si c es :
isalnum(c)	Alfanumérico (letras ó números)
isalpha(c)	Alfabetico , mayúscula ó minúscula
isascii(c)	Si su valor está entre 0 y 126
iscntrl(c)	Si es un caracter de control cuyo ASCII está comprendido entre 0 y 31 ó si es el código de "delete" , 127 .
islower(c)	Si es un caracter alfabético minuscula.
isupper(c)	Si es un caracter alfabético mayúscula
isdigit(c)	Si es un número comprendido entre 0 y 9
ispunct(c)	Si es un caracter de puntuación
isspace(c)	Si es el caracter espacio, tabulador, avance de línea, retorno de carro, etc.
isxdigit(c)	Si es código correspondiente a un número hexadecimal, es decir entre 0 - 9 ó A - F ó a - f .

Una vez que sabemos que un string está compuesto por números , podemos convertirlo en una variable numérica , de cualquier tipo , para poder realizar con ella los cálculos que correspondan .

ATOI() , ATOL() , ATOF()

- Header : <stdlib.h>
- Prototipo :
  int   atoi(const char *s)

  long   atol(const char *s)

  double atof(const char *s)
- Portabilidad : Definida en ANSI C
- Comentario : Convierten el string apuntado por s a un número . atoi() retorna un entero , atol() un long y atof() un double . (Algunos compiladores traen una función adicional , _atold() que retorna un long double) . El string puede tener la siguiente configuración :

[espacios , blancos , tabulador , etc] [signo] xxx
donde xxx son caracteres entre 0 y 9 , para atoi() y atol() . Para atof() en cambio , se aceptan :
[espacios , etc] [signo] xxx [.] [xxx] ó

[espacios , etc] [signo] xxx [.] [xxx] [e ó E [signo] xxx]

según se desee usar la convención de punto flotante ó cientifica.

Es posible tambien , aunque menos frecuente , realizar la operación inversa, es decir, convertir un número en un string.
ITOA() , ULTOA()

- Header : <stdlib.h>
- Prototipo:
  char *itoa(int numero , char *s , int base)

  char *ultoa(unsigned long numero , char *s , int base)
- Portabilidad : Definida en ANSI C
- Comentario : Retornan un puntero a un string formado por caracteres que representan los dígitos del número enviado como argumento . Por base se entiende la de la numeración en la que se quiere expresar el string , 10 para decimal , 8 para octal , 16 para hexadecimal , etc . itoa() convertirá un entero , mientras ultoa() lo hará con un unsigned long.

5. DETERMINACION DE LA LONGITUD DE UN STRING

Hemos aplicado anteriormente esta función, damos aquí entonces , sólo una ampliación de sus caracteristicas.
STRLEN() , _FSTRLEN

- Header : <string.h>
- Prototipo :
  size_t strlen(const char *s)

  size_t far _fstrlen(const char far *s)
- Portabilidad : Definidas en ANSI C
- Comentario : Retornan un entero con la cantidad de caracteres del string . No toma en cuenta al terminador NULL . Por lo que la memoria real necesaria para albergar al string es 1+strlen(s) . _fstrlen() dá idéntico resultado , pero acepta como argumento un puntero " far " .
- Ejemplo :
  .....

  ```
  char s[128] ;

  gets(s) ;

  p = (char *)malloc( sizeof( strlen(s) + 1 ) ;
  ```

---

6. COPIA Y DUPLICACION DE STRINGS

Vimos que el operador de asignación no está definido para strings , es decir que hacer p = q , donde p y q son dos arrays , no produce la copia de q en p y directamente la expresión no es compilada . Si en cambio p y q son dos punteros a caracteres , la expresión es compilada , pero no produce el efecto de copia , simplemente , cambia el valor de p , haciendo que apunte al MISMO string que q . Es decir que no se genera uno nuevo , por lo que todo lo operado sobre p afectará al original , apuntado por q .
Para generar entonces , una copia de un string en otro lugar de la memoria , se deben

utilizar alguna de las funciones abajo descriptas . Hay que diferenciar la copia de la duplicacion : la primera copia un string sobre un lugar PREVIAMENTE reservado de memoria (mediante malloc() , calloc() ó alguna otra función función de alocación) , en cambio la duplicación GENERA el espacio para guardar al nuevo string así creado.

STRCPY()

- Header : <string.h>
- Prototipo : char *strcpy(char *destino , const char *origen)
- Portabilidad : Definidas en ANSI C
- Comentario : Copia los caracteres del string "origen" , incluyendo el NULL , a partir de la dirección apuntada por "destino" . No verifica que haya suficiente memoria reservada para tal efecto , por lo que es responsabilidad del programador ubicar previamente suficiente memoria como para albergar a una copia de "origen" . Aunque es superfluo , su retorno es el puntero "destino" .

Existe tambien una función para realizar la copia PARCIAL . Por lo general las funciones que realizan acciones sobre una parte solamente , de los strings , llevan el mismo nombre de las que los afectan totalmente , pero con la adición de la letra "n".

STRNCPY()

- Header : <string.h>
- Prototipo : char *strncpy(char *destino , const char *origen , size_t n_char)
- Portabilidad : Definidas en ANSI C
- Comentario : Copia n_char caracteres del string "origen" , NO incluyendo el NULL , si la cantidad de caracteres copiada es menor que strlen(origen) + 1 , en la dirección apuntada por "destino" . n_char es un número entero y deberá ser menor que la memoria reservada apuntada por destino .
- Ejemplo:

```
#include <string.h>

main()

{

char strvacio[11] ;

char strorigen[] =  "0123456789" ;

char strdestino[] = "ABCDEFGHIJ" ;

.....

strncpy( strdestino , strorigen , 5 ) ;

strncpy( strvacio , strorigen , 5 ) ;

strvacio[5] = '\0' ;

.....
```

```
}
```
Los strings quedarían , luego de la copia :

strdestino[] == 0 , 1 , 2 , 3 , 4 , F , G , H , I , J , \0

strvacio[] == 0 , 1 , 2 , 3 , 4 , \0 , indefinidos

Note que en el caso de strdestino no hizo falta agregar el NULL , ya que éste se generó en la incialización del mismo , en cambio strvacio no fué inicializado , por lo que para terminar el string , luego de la copia , se deberá forzosamente agregar al final del mismo.

La función siguiente permite la duplicación de strings :

STRDUP()

- Header : <string.h>
- Prototipo : char *strdup(const char *origen)
- Portabilidad : Definida en ANSI C
- Comentario : Duplica el contenido de "origen" en una zona de memoria por ella reservada y retorna un puntero a dicha zona .
- Ejemplo :

```
#include <string.h>

main()

{

char *p ;

char q[] = "Duplicación de strings" ;

p = strdup( q ) ;

.....

}
```

Note que el retorno de la función debe ser siempre asignado a un dado puntero .

---

7. CONCATENACION DE STRINGS

Se puede, mediante la concatenación de dos ó más strings , crear otro , cuyo contenido es el agregado del de todos los anteriores .

La concatenación de varios strings puede anidarse , de la siguiente manera :

strcat(strcat(x , w) , z) ;

en la cual al x se le agrega a continuación el w , y luego el z . Por supuesto x tiene que tener suficiente longitud como para albergarlos .

STRCAT()

- Header : <string.h>
- Prototipo : char *strcat(char *destino , const char *origen)
- Portabilidad : Definida en ANSI C
- Comentario : Agrega el contenido de "origen" al final del string inicializado "destino" , retornando un puntero a este .

- Ejemplo :

```
#include <string.h>

char z[20] ;

main()

{

char p[20] ;

char q[] =   "123456789" ;

char w[] =   "AB" ;

char y[20] = "AB" ;

strcat( y , q ) ;      /* Correcto , el contenido de y[] será:
                       y[] == A,B,1,2,3,4,5,6,7,8,9,\0  */

strcat( z , q ) ;      /* Correcto , por ser global z[] quedó
                       inicializado con 20 NULLS por lo que
                       luego de la operación quedará:
                       z[] == 1,2,3,4,5,6,7,8,9,\0

strcat( p , q ) ;      /* Error ! p no ha sido inicializado por
                       lo que la función no encuentra el NULL
                       para empezar a agregar , por lo que
                       barre la memoria hasta encontrar
                       alguno, y ahí escribe con resultados,
                       generalmente catastróficos.

strcat( w , q ) ;      /* Error ! w solo tiene 3 caracteres,
                       por lo el resultado final será:
                       w[] == A,B,1  sin la terminación
                       del NULL por lo que cualquier
                       próxima operación que se haga
                       utilizando este array, como string,
                       fallará rotundamente .

 {
```

STRNCAT()

- Header : <string.h>
- Prototipo : char *strncat(char *destino , const char *origen , size_t cant)
- Portabilidad : Definida en ANSI C
- Comentario : Similar en un todo a la anterior , pero solo concatena cant caracteres del string "origen" en "destino" .

---

8. COMPARACION DE STRINGS

No debe confundirse la comparación de strings , con la de punteros , es decir
if(p == q) {

.....
sólo dará CIERTO cuando ambos apunten al MISMO string , siempre y cuando dichos punteros sean " near " ó " huge " . El caso que acá nos ocupa es más general , ya que involucra a dos ó más strings ubicados en distintos puntos de la memoria (abarca el caso anterior , como situación particular).

STRCMP()

- Header : <string.h>
- Prototipo : int strcmp(const char *s1 , const char *s2)
- Portabilidad : Definida en ANSI C
- Comentario : Retorna un entero , cuyo valor depende del resultado de la comparación
  < 0 si s1 es menor que s2

  == 0 si s1 es igual a s2

  > 0 si s1 es mayor que s2

La comparación se realiza caracter a caracter , y devuelve el resultado de la realizada entre los primeros dos que sean distintos.
La misma se efectua tomando en cuenta el código ASCII de los caracteres así será por ejemplo '9' < 'A' , 'A' < 'Z y 'Z' < 'a' .

STRCMPI()

- Header : <string.h>
- Prototipo : int strcmpi(const char *s1 , const char *s2)
- Portabilidad : Solamente válida para el DOS
- Comentario : Retorna un entero , de una manera similar a la anterior pero no es sensible a la diferencia entre mayúsculas y minúsculas , es decir que en este caso 'a' == 'A' y 'Z' > 'a' .

STRNCMP() , STRNCMPI()

- Header : <string.h>
- Prototipo : int strcmpi(const char *s1 , const char *s2 , size_t cant)
- Portabilidad : strncmpi() es solamente válida para el DOS strncmp() está definida en ANSI C
- Comentario : Retornan un entero , con caracteristicas similares a las de las funciones hermanas , descriptas arriba , pero solo comparan los primeros "cant" caracteres .

---

9. BUSQUEDA DENTRO DE UN STRING

Muchas veces dentro de un programa , se necesita ubicar dentro de un string , a un determinado caracter ó conjunto ordenado de ellos . Para simplificarnos la tarea existen

una serie de funciones de Librería , que toman por su cuenta la resolución de este problema :

STRCHR() Y STRRCHR()

- Header : <string.h>
- Prototipo :

  char *strchr(const char *s1 , int c)

  char *strrchr(const char *s1 , int c)
- Portabilidad : Definidas en ANSI C
- Comentario : Retornan un puntero , a la posición del caracter dentro del string , si es que lo encuentran , ó en su defecto NULL . strchr() barre el string desde el comienzo , por lo que marcará la primer aparición del caracter en él , en cambio strrchr() lo barre desde el final , es decir que buscará la última aparición del mismo . El terminador NULL es considerado como parte del string .

STRBRK()

- Header : <string.h>
- Prototipo : char *strbrk(const char *s1 , const char *s2)
- Portabilidad : Definidas en ANSI C
- Comentario : Similar a las anteriores , pero ubicando la primer aparición de cualquier caracter contenido en s2 , dentro de s1 .

STRSTR()

- Header : <string.h>
- Prototipo : char *strstr(const char *s1 , const char *s2)
- Portabilidad : Definidas en ANSI C
- Comentario : Busca dentro de s1 un substring igual a s2 , devolviendo un puntero al primer caracter del substring . cualquier caracter contenido en s2 , dentro de s1 .

STRTOK()

- Header : <string.h>
- Prototipo : char *strtok(const char *s1 , const char *s2)
- Portabilidad : Definidas en ANSI C
- Comentario : Busca dentro de s1 un substring igual a s2 , si lo encuentra MODIFICA a s1 reemplazando el substring por NULL, devolviendo la dirección de s1. Si se lo vuelve a invocar con NULL en lugar en el lugar del parámetro s1, continua buscando en el string original una nueva aparición de s2 . Cuando no encuentra el substring , retorna NULL. Sirve para descomponer un string en "palabras de una frase" cuando éstas estan separadas por un mismo caracter/es .
- Ejemplo :

```
puts("escriba una frase , separando las palabras con espacios") ;

gets(s) ;

p = strtok(s , " ") ;
```

```
while(p) {

 puts(p) ;

 p = strtok( NULL , " ") ;

}
```

---

10. FUNCIONES DE MODIFICACION DE STRING

Resulta conveniente a veces uniformizar los string leidos de teclado, antes de usarlos, un caso típico es tratar de independizarse de la posibilidad de que el operador escriba, algunas veces con mayúscula y otras con minúscula.

STRLWR() Y STRUPR()

- Header : <string.h>
- Prototipo :

```
char *strlwr( char *s1 )

char *strupr( char *s1 )
```

- Portabilidad : Definidas en ANSI C
- Comentario : Convierten respectivamente los caracteres comprendidos entre a y z a minúsculas ó mayúsculas , los restantes quedan como estaban .

FIN DE CURSO