Introducción al Lenguaje C

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 básicospresentes 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.

ANATOMIA DE UN PROGRAMA C

Siguiendo la tradicion, 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>
int main()
{
  printf("Bienvenido a la Programacion en lenguaje C \n");
  return 0;
}
```

FUNCION main()

Dejemos de lado por el momento el análisis de la primer línea del programa, y pasemos a la segunda.
La funcion main() indica donde empieza el programa, cuyo cuerpo principal es un conjunto de sentencias delimitadas por dos llaves, una inmediatamente después de la declaracion main() " { ", y otra que finaliza el listado "} ". Todos los programas C arrancan del mismo punto: la primer sentencia dentro de dicha funcion, en este caso printf (". ").

En el EJEMPLO 1 el programa principal está compuesto por solo dos sentencias: la primera es un llamado a una funcion 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 funcion externa. En este caso se invoca a la funcion 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 notacion que emplea C para el caracter "nueva linea" que hace avanzar al cursor a la posicion extrema izquierda de la línea siguiente.
La segunda sentencia (return 0) termina el programa y devuelve un valor al Sistema operativo, por lo general cero si la ejecucion 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 finalizacion 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 renglon, y habrá que avisarle al compilador donde terminan.

ENCABEZAMIENTO

Las líneas anteriores a la funcion 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.
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 algún otro directorio, cuyo nombre habitualmente se le suministra en el momento de la instalacion 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 extension .h.

La razon de la existencia de estos archivos es la de evitar la repeticion de la escritura de largas definiciones en cada programa.

Notese 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 "; ".

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 o por sí mismo, varios meses atrás. Para el compilador, los comentarios son inexistentes, por lo que no generan líneas de codigo, permitiendo abundar en ellos tanto como se desee.

En el lenguaje C se toma como comentario todo caracter interno a los simbolos: /* */ .
Los comentarios pueden ocupar uno o más renglones. También podemos tener comentarios sobre una misma líneas mediante la inclusion de doble barras //

COMENTARIOS

```
suma = suma + k  // comentario sobre la línea
/* este es un comentario corto */
/* .....
..... */
```

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.

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 solo 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 o con el símbolo de subrayado "_" , pudiendo continuar con cualquier otro caractér alfanumérico o 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 convencion 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 "Definicion incorrecta" a la definicion de variables con nombres del tipo de : 4pesos $variable primer-variable !variable etc.etc

Vemos en las dos líneas subsiguientes a la definicion de las variables, que puedo ya asignarles valores (1000 y 2) y luego efectuar el cálculo de la variable "resultado". Si prestamos ahora atencion a la funcion 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 impresion "%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 finalizacion 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 :

INICIALIZACION DE VARIABLES

Las variables del mismo tipo pueden definirse mediante una definicion 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 funcion printf() vemos que se ha reemplazado "resultado" por la operacion 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 funcion , pasando finalmente a esta el valor resultante de las mismas.

El EJEMPLO 2 funciona exactamente igual que antes pero su codigo ahora es mucho más compacto y claro.

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" o 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" o 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 " o simplemente "short", aunque por lo general, los compiladores modernos asignan a este tipo el mismo alcance que "int".

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 o 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 solo puede adoptar valores positivos (como por ejemplo la edad de una persona) podemos aumentar el alcance de cualquiera de los tipos, restringiéndolos a que solo 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 omision (default) como "signed".

Variables de número real o punto flotante

Un número real o de punto flotante es aquel que además de una parte entera, posee fracciones de la unidad. En nuestra convencion numérica solemos escribirlos de la siguiente manera : 2,3456, lamentablemente los compiladores usan la convencion del PUNTO decimal (en vez de la coma) . Así el numero Pi se escribirá : 3.14159

Otro formato de escritura, normalmente aceptado, es la notacion científica. Por ejemplo podrá escribirse 2.345E+02, equivalente a 2.345 * 100 o 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.

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 o End Of File), en este caso dicho número no puede ser mantenido en una variable del tipo char, ya que esta solo puede guardar entre 0 y 255 si se la define unsigned o no podría mantener los caracteres comprendidos entre 128 y 255 si se la define signed (ver TABLA 1).
El problema se obvia fácilmente definiéndola como int.
Las variables del tipo caractér también pueden ser inicializadas en su definicion, 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 inicializacion 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" o ENTER .
Con el fin de tener acceso a los mismos es que aparecen ciertas secuencias de escape convencionales . Las mismas están listadas en la TABLA 3 y su uso es idéntico al de los caracteres normales , así para resolver el caso de una asignación de "nueva línea " 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
'\? '	interrogacion	63	0x3F
'\nnn'	cualquier caracter (donde nnn es el codigo ASCII expresado en octal)		
'\xnn'	cualquier caracter (donde nn es el codigo ASCII expresado en hexadecimal)		

TAMAÑO DE LAS VARIABLES

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 rígidamente (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.

DEFINICION DE NUEVOS TIPOS (typedef)

A veces resulta conveniente crear otros tipos de variables , o 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 ;

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 asignacion.
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: por ejemplo :
#define VALOR_CONSTANTE 342
#define PI 3.1416

OPERADORES

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 asignación. Todos ellos se irán describiendo en los párrafos subsiguientes.

OPERADORES ARITMETICOS

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

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 modulo (%) se utiliza para calcular el resto del cociente entre dos ENTEROS , y NO puede ser aplicado a variables del tipo float o double .

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 modulo 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 solo obligan al compilador a realizar las operaciones en un orden 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 confundírselo con el de resta, ya que este es un operador unitario que opera sobre una única variable cambiando el signo de su contenido numérico. 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.

OPERADORES RELACIONALES

Todas las operaciones relacionales dan solo dos posibles resultados: VERDADERO o 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 asignación 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 o 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. Así el carácter a (ASCII 97) será mayor que el A (ASCII 65) o que el 9 (ASCII 57).

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 o 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.

OPERADORES DE INCREMENTO Y DECREMENTO

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

TABLA 7 OPERADORES DE INCREMENTO Y DECREMENTO

SIMBOLO	DESCRIPCION	EJEMPLO	ORDEN DE EVALUACION
++	Incremento	++i o i++	1
--	Decremento	--i o i--	1

Para visualizar rápidamente 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 o 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 asignación tomando el valor de 3 , él que luego es copiado en k .

OPERADORES DE ASIGNACION

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

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 pseudocodigo , con el fín de abreviarlas . Así una operación aritmética 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 */
```

Notese que el pseudooperador debe escribirse con los dos símbolos seguidos, por ejemplo += , y no será aceptado
+(espacio) = .

Los operadores de asignación están 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 = (operacion relacional o logica) ? (rvalue 1) : (rvalue 2) ;
de acuerdo al resultado de la operación condicional se asignará a lvalue el valor de rvalue 1 o 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 o b , bastará con escribir : c = (a < b) ? a : b ;

PROPOSICIONES PARA EL CONTROL DE FLUJO DE PROGRAMA

En lo que sigue, 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 sintácticamente 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 funcion main() .

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

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

PROPOSICION IF - ELSE
Esta proposición sirve para ejecutar ciertas sentencias de programa, si una expresión resulta CIERTA u otro grupo de sentencias, si aquella resulta FALSA. Su interpretación literal sería:
SI es CIERTA tal cosa , haga tal otra , si no lo es saltéela .
El caso más sencillo sería :

```
if(expresion)
sentencia ; o
if(expresion) sentencia ;
```

Cuando la sentencia que sigue al IF es única, las dos formas de escritura expresadas arriba son equivalentes . La sentencia solo se ejecutará si el resultado de "expresion" 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 "expresion" 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 solo cuando a sea FALSO. (ver operador NEGACION en el capítulo anterior)
if(a == b)		
if(a = b)	/* Error */	La primera es una expresión correcta, el IF se realizará solo 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.

En casos más complejos que los anteriores, la proposicion 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 o anidadas, como se verá luego.

El bloque se ejecutará en su conjunto si la expresión resulta CIERTA. El uso del ELSE es optativo, y su aplicación resulta en la ejecución de una, o 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 expresion 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 proposicion ELSE queda siempre asociada al IF más cercano, arriba de él .

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 (expresion) .

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 también 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 u 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 , o Z ") ;
break ;
.................
```

La forma de interrumpir la ejecución luego de haber encontrado un CASE cierto es por medio del BREAK , el que da por terminado el SWITCH .

Al final del bloque de sentencias del SWITCH , aparece una optativa llamada DEFAULT , que implica : si no se ha cumplido ningún CASE , ejecute lo que sigue. Es algo superfluo poner el BREAK en este caso, ya que no hay más sentencias después 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 preverlo , agregando el BREAK , para evitar errores de laboriosa ubicación .

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

```
while(expresión)          o        while(expresión) {
  proposición 1 ;                     proposición 1 ;
                                      proposición 2 ;
                                     ..............
                                      proposición n ;
                                      }
```

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

LA ITERACION DO - WHILE

Su sintaxis será :

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

Expresado en palabras , esto significa : ejecute las proposiciones , luego repita la ejecucion 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.

ITERACION FOR

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

```
for ( expresion1 ; expresion2 ; expresion3 ) {
proposicion1 ; proposicion2 ;
.....}
```

Esto es equivalente a :

```
expresion1 ;
while ( expresion2 ) { proposicion1 ;
proposicion2 ;
.....
expresion3 ;
}
```

La expresion1 es una asignación de una o más variables , (equivale a una inicialización de las mismas) , la expresion2 es una relación de algún tipo que , mientras dé un valor CIERTO , permite la iteración de la ejecución y expresion3 es otra asignación , que comúnmente varía alguna de las variables contenida en expresion2 .

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

FUNCIONES

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" o "imprimir un caracter".
Luego solo resta abocarse a resolver cada una de ellas por separado.
De esta forma el programador, solo se las tendrá que ver con diminutas piezas de programa, de pocas líneas, 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 son las funciones. Todo compilador comercial trae una gran cantidad de Librerias de toda índole, matemáticas, de entrada - salida, de manejo de textos, de manejo de gráficos, etc, que solucionan la mayor parte de los problemas básicos de programación.

Comencemos con algunos conceptos básicos: para hacer que las instrucciones contenidas en una funcion, 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 o 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 funcion 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 funcion 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 funcion modifique a una variable, pero para ello, será necesario comunicarle la DIRECCION EN MEMORIA de dicha variable .

Las funciones pueden o no devolver valores al programa invocante. Hay funciones que tan solo 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 efectúan 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 .

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 declaracion queda dada por :
tipo del valor de retorno nombre_de_la_funcion(lista de tipos de parámetros) Pongamos algunos ejemplos :

```
float mi_funcion(int i, double j ) ;
double otra_funcion(void) ;
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 funcion; en caso de obviarse el mismo se toma, por omisión, el tipo int. Sin embargo, aunque la funcion 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 funcion NO retorna nada, esto se realiza por medio de la palabra VOID (sin valor).

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.

DEFINICION DE LAS FUNCIONES

La definicion 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 funcion */
.....
}          /* fin de la funcion main() */

float mi_funcion(int i, double j ) /* DEFINICION observe que NO lleva ";" */
{
float n
.....
printf("%d", i ); /* LLAMADA a otra funcion */
.....
return ( 2 * n ); /* RETORNO devolviendo un valor float */
}
```

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

FUNCIONES QUE NO RETORNAN VALOR NI RECIBEN PARAMETROS

Veamos como ejemplo la implementación de una función "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 funcion 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 funcion es definida. El bloque de sentencias de la misma está compuesto, en este caso particular, por la definicion de una variable c, la impresion 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 funcion, ejecutándose, hasta que ésta finalice, su propia secuencia de instrucciones. Al finalizar la funcion esta retorna el comando al programa principal, continuandose la ejecucion por la instruccion 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 ningún parámetro o valor .

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 líneas incluirán, 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 logicas 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 fácilmente comprensible en el futuro. Comienza luego la funcion main(), inicializando dos variables, i y fin, donde la primera nos servirá de contador y la segunda de indicador logico. Luego de imprimir el rotulo 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 funcion, lea_char(), y asigna su retorno a c iterando esta operativa si c no es 'n' o 's', note que: c != 'n' && c != 's' es equivalente a: !(c == 'n' || c == 's') .

La funcion lea_char() tiene como mision leer un caracter enviado por el teclado, (lo realiza dentro de la expresion relacional del IF) y salvar la ambigüedad del uso de mayúsculas o 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 conversion 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 o s, se saldrá del do-while en la funcion finalizar() y se retornará al programa principal, el valor de la comparacion logica 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 presiono la tecla n) hace terminar el loop e imprimir el mensaje de despedida.

Nota: preste atencion a que en la funcion finalizar() se ha usado un do-while .¿Como modificaría el programa para usar un while ?. En la funcion 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 funcion muy común de entrada de datos: scanf(), que nos permitirá leer datos completos (no solo caracteres) enviados desde el teclado, su expresion formal es algo similar a la del printf() , scanf("secuencia de control", direccion 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 "direccion 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 direccion de su ubicacion 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.

El prototipo de scanf() esta declarado en stdio.h .

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 funcion (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 ejecucion 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 funcion la declare, para actuar sobre ella.

VARIABLES LOCALES

A diferencia de las anteriores, las variables definidas DENTRO de una funcion, 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 funcion, respectivamente .

Estas variables se ubican en la pila dinámica (stack) de memoria, destinándosele un espacio en la misma cuando se las define dentro de una funcion, 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 o nombre que se la haya dado a una variable es solo relevante entonces, para la funcion que la haya definido, pudiendo existir entonces variables que tengan el mismo nombre, pero definidas en funciones distintas, sin que haya peligro alguno de confusion .

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. Recuérdese entonces que, si no se las inicializa expresamente en el momento de su definición, su valor será indeterminado (basura) .

ARGUMENTOS Y PARAMETROS DE LAS FUNCIONES

Supongamos que en un determinado programa debemos calcular repetidamente el valor medio de dos variables, una solucion razonable sería crear una funcion 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 declaracion 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 declaracion vemos que la funcion valor_medio recibe dos argumentos (x e y) del tipo double y devuelve un resultado de ese mismo tipo .

Cuando definamos a la funcion 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 declaracion de la funcion, 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 .

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>
/* Declaracion de la funcion 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 funcion y pasaje de argumentos */
printf("\n\nResultado de la operacion: %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 ) ;
}
```

```
/* Definicion de la funcion 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 operacion: 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 funcion de cálculo en la sentencia de asignación de la variable resultado. La funcion 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 automáticamente, mientras que aquellos lo son, a los valores de los argumentos colocados en la expresion 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 funcion invocada. Una clara prueba de ello es que, en la funcion 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 .

Especial cuidado debe tenerse entonces con los errores que pueden producirse por redondeo o truncamiento, siendo una buena técnica de programacion hacer coincidir los tipos de los argumentos con los de los parámetros.