INTRODUCCIÓN

El lenguaje de programación C se caracteriza por ser de uso general, con una sintaxis sumamente compacta y de alta portabilidad.
Se le suele clasificar dentro de los leguajes de bajo nivel, aunque lo correcto es decir de "medio nivel", esto significa que el progamador puede relacionarse con la máquina tanto con llamadas al sistema, como con elementos de programación de un lenguaje de alto nivel.

C se puede definir como el lenguaje por excelendia de los 90, sus capacidades de interacción con el sistema, programación estructurada, portabilidad, secillez, etc le convierten en el preferido para programar otros lenguajes y sistemas operativos.

ELEMENTOS DEL LENGUAJE C

Para comprender la estructura de los programas en C, observemos un ejemplo sencillo:

```
#include <stdio.h>

int main()

{
        /* El programa imprime lo siguiente */
      printf("Ejemplo de programa sencillo en 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 linea" que hace avanzar al cursor a la posición extrema izquierda de la línea siguiente. Más adelante analizaremos otras secuencias de escapehabituales.

La segunda sentencia (return 0) termina el programa y devuelve un valor al Sistema operativo, por lo general cero si la ejecución fue 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.

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 < > 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. Si 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 los directorios includes que tenga el sistema definidos. El nombre de estos ficheros siempre estará 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.

La directiva "#include" no es una sentencia de programa sino una orden de que se incluya literalmente un archivo .h sobre el código a la hora de la compilación ,por lo que no es necesario terminarla con ";"

COMENTARIOS

La inclusión de comentarios en un programa es casi obligatorio cuando desarrollamos aplicaciones Linux que seguramente leerá otra persona (posiblemente en otro idioma diferente al castellano). 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 comprendido entre los simbolos: /* y */ . Los comentarios pueden ocupar varios renglones, por ejemplo:

```
/* comentario simple */

/* comentario de varias líneas

segunda linea del comentario */
```

Se debe tener en cuenta que no es posible anidar comentarios, es decir, lo siguiente sería inválido:

```
/* This is a comment
   /* Pongo el comentario en inglés para q lo lea Alan Cox */

End of the comment */
```

También podemos utilizar la notación de comentarios de C++ para líneas simples, comenzándolas con el símbolo //. Ej:

```
// Esto es otro comentario al estilo C++
```

VARIABLES

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;          /* multiplicador es un entero */
  int multiplicando;          /* multiplicando es un entero */
  int resultado;              /* resultado es un entero */
  multiplicador = 1000 ;      /* asignación de valores */
  multiplicando = 2 ;
  resultado = multiplicando * multiplicador ;
  printf("Resultado = %d\n", resultado);          /* muestramos el resultado */
  return 0;
}
```

En las primeras líneas de texto dentro de main() defino mis variables como enteros, es decir del tipo "int" seguido de un identificador (nombre) del mismo. Este identificador puede tener una cantidad de caracteres variable, pero dependiendo del compilador que se use, éste tomará como significantes sólo los n primeros 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 en C escribir los nombres de las variables y las funciones con minúsculas, y las constantes en mayúsculas.

SALIDA DEL EJEMPLO 2: Resultado = 2000

INICIALIZACION DE VARIABLES

Las variables del mismo tipo pueden definirse mediante una definición múltiple separandolas mediante comas ",":

```
int multiplicador, multiplicando, resultado;
```

Esta sentencia es equivalente a las tres definiciones separadas en el ejemplo anterior. También podemos inicializar las variables en el momento de definirse.

```
int multiplicador = 1000, multiplicando = 2, resultado;
```

De esta manera el EJEMPLO 2 podría escribirse:

```
EJEMPLO 2 SIMPLIFICADO
#include <stdio.h>
main()
{
  int multiplicador=1000,multiplicando=2;
  printf("Resultado = %d\n", multiplicando * multiplicador);
  return 0;
}
```

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 a 256^2, es decir, el rango entre -32.768 y +32.767 (incluyendo 0).

En caso de necesitar un rango más amplio, puede definirse la variable como "long int" ó en forma más abreviada "long".

Declarada de esta manera, la 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 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.

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
unsigned 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 no se pone el calificador delante del tipo de la variable entera, éste se adopta por defecto 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, desafortunadamente 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.

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 aumentadas a su nivel 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,"valorX" a la variable a la izquierda del signo igual y "valorY" a la expresión a la derecha del mismo, es decir:

"valorX" = "valorY" ;

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

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

CONVERSIONES EXPLÍCITAS (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)

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;
```

También funcionaría correctamente definida 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 podria 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 letra_a = 97 ;
```

Para que c contenga el valor ASCII de la letra "a", aunque también podemos hacer la misma operación de la siguiente forma:

```
char letra_a = 'a' ;
```

Existen una serie de caracteres que no son imprimibles, como por ejemplo el caso típico sería el de "nueva linea" ó ENTER. Para tener acceso a esos caracteres especiales, se han definido una serie de caracteres de escape tal y como se puede observar en la TABLA 3. Su uso es idéntico al de los caracteres normales. Para añadir un caracter de nueva línea se usaría:

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

TABLA 3 SECUENCIAS DE ESCAPE

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

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. Para conocer con seguridad cuál es el tamaño de un determinado tipo en memoria utilizaremos un operador llamado "sizeof" que calcula sus requerimientos de almacenaje de forma que el código sería independiente de la plataforma o complidor que se utilice. También se puede usar sizeof sobre una variable:

```
sizeof(int)
sizeof(char)
sizeof(long double), etc.
```

Devolvería como salida el número de bytes de memoria que requiere el tipo.

DEFINICION DE NUEVOS TIPOS (typedef)

Hay veces que necesitamos la creación de nuevas variables o redefinir alguna existente por otra que se adecue más a nuestros requerimientos, 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

Aquellos valores que no necesitamos que cambien durante la ejecución del programa, podemos definirlas como CONSTANTES.

El compilador, en el momento de crear el ejecutable, reemplazará el símbolo (PI en el ejemplo) por el valor asignado.
Para dar un símbolo a una constante bastará, en cualquier lugar del programa (antes de ser usada, claro) poner la directiva: #define. Ej:

```
#define PI 3.1416
```

OPERADORES

INTRODUCCION

Si analizamos la sentencia siguiente:

```
var1 = var2 + var3;
```

Estamos diciéndole al programa, por medio del operador +, que calcule 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 se pueden catalogar dentro de 6 categorias: aritméticos, relacionales, lógicos, incremento y decremento, manejo de bits y asignación

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 más un operador adicional: 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.

El orden de evaluación cuanto más bajo sea el 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.

El último de los operadores aritméticos es el de SIGNO. Su función es la de cambiarle la "polaridad" a un número, es decir, -a, siendo a -30 daría como resultado 30.

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

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 parae aumentar la legilibilidad del código.

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).

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

La evaluación de las operaciones lógicas se realiza de izquierda a derecha y se interrumpe cuando se ha asegurado el resultado, de forma que no se continúa analizando el resto de la expresión.

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

```
!( a >> b )    equivale a    ( a < b )
!( a == b )      "     "     ( a != b )
```

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++;
```

hacen lo mismo, de la misma forma que

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

es decir incrementan o decrementan la variable en uno.
Las sentencias siguientes son equivalentes:

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

Pero la diferencia es cuándo se realiza la comprobación. Veamos un ejemplo:

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

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.

OPERADORES DE ASIGNACION

Como ya se ha comentado anteriormente, este operador asigna el valor de la derecha del igual a la varible de la izquierda, por ejemplo:

```
a = 17 ;
```

pero no es aceptado, en cambio

```
17 = a ;
```

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

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 es posible escribir

```
c = ( a = b );
```

Aunque podemos obviar los paréntesis de la siguiente forma:

```
c = a = b = 17 ;
```

con lo que las tres variables resultarán iguales al valor de la contante .

También podemos usar la variable a ambos lados de la asignación de la siguiente forma:

```
a = a + 17 ;
```

lo que significaría que el valor que tiene a es el que tenia anteriormente más 17 unidades

Podemos abreviar esto de la siguiente manera, no resulta muy legible pero simplifica las operaciones:

```
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 */
```

TABLA 8 OPERADORES DE ASIGNACION

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

La asignación condicional funciona como un IF, THEN, ELSE:

lvalue = (operación relacional ó logica) ? (rvalue 1) : (rvalue 2) ;

c = (a < b)?a:b;

Lo que haría que c tomase el valor del menor de las variables a o b. Se interpretaría así:

c = Si a es menor que b entonces devuelve a, sino devuelve b

BLOQUE DE SENTENCIAS

INTRODUCCION

Un BLOQUE DE SENTENCIAS es el 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 que ya conocemos es el cuerpo del programa principal de la función main():

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

PROPOSICION IF - ELSE

Esta proposición sirve para ejecutar ciertas sentencias de programa, si una expresión resulta CIERTA ú otro grupo de sentencias, si resulta FALSA.
Su interpretación literal sería : SI es CIERTA haz tal, sino haz otra cosa.

El caso más sencillo sería :

```
if(expresión)
        sentencia;
o

if(expresión) sentencia ;

o

if(expresión)
{
        sentencia;
}
```

Cuando la sentencia que sigue al IF es única, las tres 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) Esta segunda expresión es de idéntica interpretación que la
primera, podemos omitir el !=0
```

Otras dos condiciones IF idénticas:

```
if(a)
if(a != 0)
```

Y otras dos más:

```
if(!a) if(a == 0 )
```

También podemos utilizar un bloque de sentencias después de un IF:

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

El bloque se ejecutará en su conjunto si la expresion resulta CIERTA. El bloque de sentencias debajo de la palabra reservada "else" (si no...) se ejecuta siempre que la condioción de entrada del IF sea FALSA

Su aplicación puede verse en el ejemplo siguiente :

```
if(expresión)
{
        sentencia 1a;
        sentencia 2a;
}
else
{
        sentencia 1b;
        sentencia 2b;
}
sentencia 3;
sentencia 4;
sentencia 5;
```

Esto funcionaría de la siguiente manera. Si expresión es CIERTA entonces se ejecutan las sentencias 1a y 1b, en caso contrario 1b y 2b. sentencias 3,4 y 5 se ejecutarían siempre.

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\tA = AÑADIR");
        printf("\n\tB = BORRAR");
        printf("\n\tO = ORDENAR");
        printf("\n\tI = IMPRIMIR");
        printf("\n\nESCRIBA SU OPCIÓN Y PULSE INTRO: ");

        if( (c = getchar()) != '\n')
        {
                if( c == 'A')
                        printf("\nSELECCIONÓ AÑADIR");
                else
                        if( c == 'B')
                                printf("\nSELECCIONÓ BORRAR");
                        else
                                if( c == 'O' )
                                        printf("\nSELECCIONO ORDENAR");
                                else
                                        if( c == 'I' )
                                                printf("\nSELECCIONO IMPRIMIR");
                                        else
                                                printf("\n\a\aVALOR INCORRECTO" );
        }
        else
                printf("\niNO HA SELECCIONADO NADA!");
}
```

Cuando las opciones son muchas, el código es difícil de entender y escribir. El mismo programa utilizando SWITCH, quedaría mucho más claro de leer, y sencillo de escribir, como se puede ver en el EJEMPLO siguiente:

```
#include <stdio.h>
main()
{
        int c;

        printf("\nMENU :");
        printf("\n       A = AÑADIR");
        printf("\n       B = BORRAR");
        printf("\n       O = ORDENAR");
        printf("\n       I = IMPRIMIR");
        printf("\n\nESCRIBA SU OPCIÓN Y PULSE INTRO: ");
        c = getchar();

        switch (c)
        {
                case 'A':
                        printf("\nSELECCIONÓ AÑADIR") ;
                        break;
                case 'B':
                        printf("\nSELECCIONO BORRAR") ;
                        break;
                case 'O':
                        printf("\nSELECCIONO ORDENAR") ;
                        break;
                case 'I':
                        printf("\nSELECCIONO IMPRIMIR") ;
                        break;
                case '\n':
                        printf("\niNO HA SELECCIONADO NADA!");
                        break;
                default:
                        printf("\n\a\aVALOR INCORRECTO" ) ;
                        break; /* Este último break es inútil */
        }
}
```

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. Si llega al final de los case sin encontrar coincidencia ejecutaría el bloque contendido tras "default" ya que esa es sufunción.

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("ESCRIBIÓ X , Y , ó  Z ") ;
        break ;
...
```

La forma de interrumpir la ejecución luego de haber encontrado un CASE cierto es por medio del BREAK.

WHILE

El WHILE es una de las tres iteraciones posibles en C. Su sintaxis es:

```
while(expresion)
        proposición 1;

o bien:

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

Esta sintaxis expresada en palabras significaria: mientras la expresión sea CIERTA, ejecuta proposición 1 o bloque.

Por lo general dentro de la proposición ó del bloque se modifican términos de la expresión condicional para controlar la duración de la iteración. De otra forma entraríamos en un bucle infinito.

DO - WHILE

Su sintaxis es:

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

Expresado en palabras significa: ejecuta las proposiciones y repitelas mientras la expresión dé un resultado CIERTO. La diferencia fundamental con el WHILE es que el DO-WHILE se ejecuta siempre AL MENOS una vez, sea cual sea el resultado de expresión. Esto es debido a que la condición de repetición del bucle se analiza al finalizar el bloque.

FOR

El FOR es simplemente una manera abreviada de expresar un WHILE. 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.
- La 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)
for (exp1;;)
for (;;)
```

Estas dos últimas expresiónes son interesantes desde el punto de vista de su carencia de término relacional, lo que implica que el programador deberá haber previsto alguna manera alternativa de salir del bucle (probablemente mediante BREAK ó RETURN como veremos ahora) ya que de otra forma, la ejecución del programa sería infinita.

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>
main()
{
        char c ;
        printf("ESTE ES UN BUCLE INFINITO");
        while(1)
        {
                printf( "DENTRO DEL BUCLE INFINITO (pulse Q para salir): ");
                if ((c = getchar()) == 'Q')
                        break;
                printf( "\nNO FUE LA TECLA CORRECTA PARA ABANDONAR EL BUCLE");
        }
        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 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 (;;) /* bucle infinito */
{
        ...
        if(expresión)
                break;  /* salida del bucle cuando la expresión sea cierta */
}
```

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.

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 el tipo de error que se cometió.

FUNCIONES

INTRODUCCIÓN

Las funciones son bloques de código utilizados para dividir un programa en partes más pequeñas, cada una de las cuáles tendrá una tarea determinada.

Su sintaxis es:

```
tipo_función nombre_función (tipo y nombre de argumentos)
{
        bloque de sentencias
}
```

tipo_función: puede ser de cualquier tipo de los que conocemos. El valor devuelto por la función será de este tipo. Por defecto, es decir, si no indicamos el tipo, la función devolverá un valor de tipo entero (int). Si no queremos que retorne ningún valor deberemos indicar el tipo vacío (void).

nombre_función: es el nombre que le daremos a la función.

tipo y nombre de argumentos: son los parámetros que recibe la función. Los argumentos de una función no son más que variables locales que reciben un valor. Este valor se lo enviamos al hacer la llamada a la función. Pueden existir funciones que no reciban argumentos.

bloque de sentencias: es el conjunto de sentencias que serán ejecutadas cuando se realice la llamada a la función.

Las funciones pueden ser llamadas desde la función main o desde otras funciones. Nunca se debe llamar a la función main desde otro lugar del programa. Por último recalcar que los argumentos de la función y sus variables locales se destruirán al finalizar la ejecución de la misma. 10.3.- Declaración de las funciones

Al igual que las variables, las funciones también han de ser declaradas. Esto es lo que se conoce como prototipo de una función. Para que un programa en C sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones.

Los prototipos de las funciones pueden escribirse antes de la función main o bién en otro fichero. En este último caso se lo indicaremos al compilador mediante la directiva #include.

En el ejemplo adjunto podremos ver la declaración de una función (prototipo). Al no recibir ni retornar ningún valor, está declarada como void en ambos lados. También vemos que existe una variable global llamada num. Esta variable es reconocible en todas las funciones del programa. Ya en la función main encontramos una variable local llamada num. Al ser una variable local, ésta tendrá preferencia sobre la global. Por tanto la función escribirá los números 10 y 5.

```
/* Declaración de funciones. */
#include <stdio.h>
void funcion(void); /* prototipo */
int num=5; /* variable global */
main() /* Escribe dos números */
{
        int num=10; /* variable local */
        printf("%d\n",num);
        funcion(); /* llamada */
}

void funcion(void)
{
        printf("%d\n",num);
}
```

PASO DE PARÁMETROS A UNA FUNCIÓN

Como ya hemos visto, las funciones pueden retornar un valor. Esto se hace mediante la instrucción return, que finaliza la ejecución de la función, devolviendo o no un valor.

En una misma función podemos tener más de una instrucción return. La forma de retornar un valor es la siguiente:

```
return ( valor o expresión );
```

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

En el ejemplo puedes ver lo que ocurre si no guardamos el valor en una variable. Fíjate que a la hora de mostrar el resultado de la suma, en el printf, también podemos llamar a la función.

```
/* Paso de parámetros. */

#include <stdio.h>

int suma(int,int); /* prototipo */
main() /* Realiza una suma */
{
        int a=10,b=25,t;
        t=suma(a,b); /* guardamos el valor */
        printf("%d=%d",suma(a,b),t);
        suma(a,b); /* el valor se pierde */
}

int suma(int a,int b)
{
        return (a+b);
}
```

PUNTEROS

INTRODUCCION

Los punteros en el Lenguaje C son variables que "apuntan", es decir que poseen la dirección de memoria de otras variables, y por medio de ellos tendremos un potente método de acceso a todas ellas.

Son una herramienta muy cómoda y directa para el manejo de variables complejas, argumentos, parámetros, etc.

La declaración de un puntero es de la siguiente forma:

```
tipo_de_variable_apuntada *nombre_del_puntero;
```

Ejemplos:

```
int *pint;  //Puntero a un entero

double *pfloat; //Puntero a un float

char *letra, *codigo, *caracter ; //Punteros a caracteres o a cadenas
```

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. Un puntero debe ser inicializado antes de usarse, y una de las formas de hacerlo es la siguiente:

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

int *pint ;    /* declaro (y creo en memoria) 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 */
```

&nombre_de_una_variable implica la dirección de la misma .

En la declaración del puntero está implicito el tamaño (en bytes) de la variable apuntada.

El símbolo & o 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, la de referenciación del mismo, se puede utilizar para hallar el valor contenido por la variable apuntada. Es lo mismo:

```
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". En este ejemplo se verá mejor:

```
#include <stdio.h>
main()
{
        char var1 ;    /*una variable del tipo caracter */
        char *pchar;  /* un puntero a una variable del tipo caracter */
        pchar = &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 aqui que en el FOR incrementa el valor de la variable y luego para imprimirla usamos la de-referenciación 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 sólo caso en el que la 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) ... o if(print != NULL) para comprobar la validez del resultado devuelto por una función.

Hay una relación muy cercana entre los punteros y los arrays. El nombre de un array es equivalente a la dirección del elemento [0] del mismo. La explicación de ésto es sencilla: el nombre de un array, para el compilador de C, es un PUNTERO inicializado con la dirección del primer elemento del array.
Veamos algunas operaciones permitidas entre punteros :

```
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" . El siguiente ejemplo nos muestra un tipo de error bastante frecuente:

```
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 :

```
int *pint , array_int[5] ;
double *pdou , array_dou[6] ;
pint = array_int ;      /* pint apunta a array_int[0] */
pdou = array_dou ;      /* pdou apunta a array_dou[0] */
pint += 1 ;             /* pint apunta a array_int[1]  */
pdou += 1 ;             /* pdou apunta a array_dou[1]  */
pint++ ;                /* pint apunta a array_int[2] */
pdou++ ;                /* pdou apunta a array_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: "incrementa 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 , array_int[1] y array_dou[1].

Vemos que de ésta manera será muy facil recorrer arrays independientemente del tamaño de variables que lo compongan, permitiendo por otro lado que elprograma sea transportable a distintas arquitecturas sin preocuparnos de la diferente cantidad de bytes que pueden asignar los mismos a un dado tipo de variable.

De manera similar, en las dos instrucciones siguientes vuelven a a incrementarse los punteros, apuntando ahora a los elementos siguientes de los arrays.
Todo lo dicho es aplicable al operador de decremento -- .

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.

La aritmética más usada con punteros son las sencillas operaciones de asignación, incremento ó decremento y de-referenciación. Cualquier 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.

Otras operaciones, como la comparación de dos punteros estan permitidas. Por ejemplo (punt1==punt2) o (punt1<punt2). Sin embargo este tipo de operaciones son potencialmente peligrosas, ya que con algunos modelos de punteros pueden funcionar correctamente y con otros no.

Recordemos lo expresado en capítulo 5, sobre el ámbito o 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 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 deapilamiento 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 estálleno) devolvera un puntero inicializado con NULL .

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

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 ;
```

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 , ¿Qué"
#define TEXTO2  " tal va todo?"
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 ? .

```
#include <stdio.h>

main()
{
        char *p , palabra[20];
        printf("Escribe tu nombre: ");
        scanf("%s" , p ) ;
        palabra = "¿Como te va, ";
        printf("%s%s?", palabra, p);
}
```

Hay dos errores: el primero ya fue analizado antes , la expresión scanf("%s" , p) es correcta pero el error es no haber inicializado al puntero p, sólo fué definido , pero aun no apunta a ningun lado válido . El segundo error está dado por la expresión : palabra = " ¿Como te va " ; (también visto anteriormente) ya que el nombre del array es una constante y no puede ser asignado a otro valor .

Así estaría correcto:

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

main()
{
        char *p, palabra[20];
        p = (char *)malloc(sizeof(char));
        printf("Escriba su nombre: ");
        scanf("%s",p);
        strcpy(palabra,"¿Cómo te 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 strcpy().

Otra forma de resolverlo sería:

```
#include <stdio.h>
main()
{
        char p[20] , *palabra ;
        printf("Escriba su nombre : ") ;
        scanf("%s" , p ) ;
        palabra = "¿ Como le va " ;
        printf("%s%s" , palabra , p ) ;
}
```

Este ejemplo es idéntico al primero con la salvedad que se han invertido las declaraciones de las variables: ahora el puntero es palabra y el array es p.

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 .

INICIALIZACIÓN 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[] = {
                "día no válido",
                "lunes"   ,
                "martes" ,
                "miercoles"       ,
                "jueves"  ,
                "viernes" ,
                "sabado" ,
                "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.

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 repasar la TABLA 13 donde se indican las precedencias de los operadores, a fín 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 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 escribirés:

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

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

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 PARÁMETROS 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 FUNCIÓN

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 ) ;
```

MANEJO DE FICHEROS

Ahora veremos la forma de almacenar datos que podremos recuperar cuando deseemos. Estudiaremos los distintos modos en que podemos abrir un fichero, así como las funciones para leer y escribir en él.

APERTURA

Antes de abrir un fichero necesitamos declarar un puntero de tipo FILE, con el que trabajaremos durante todo el proceso. Para abrir el fichero utilizaremos la función fopen( ).

Su sintaxis es:

```
FILE *puntero;
puntero = fopen ( nombre del fichero, "modo de apertura" );
```

Donde puntero es la variable de tipo FILE, nombre del fichero es el nombre que daremos al fichero que queremos crear o abrir. Este nombre debe ir encerrado entre comillas. También podemos especificar la ruta donde se encuentra o utilizar un array que contenga el nombre del archivo (en este caso no se pondrán las comillas). Algunos ejemplos:

```
puntero=fopen("DATOS.DAT","r");
puntero=fopen("C:\\TXT\\SALUDO.TXT","w");
```

Un archivo puede ser abierto en dos modos diferentes, en modo texto o en modo binario. A continuación lo veremos con más detalle.

Modo texto

w	crea un fichero de escritura. Si ya existe lo crea de nuevo.
w+	crea un fichero de lectura y escritura. Si ya existe lo crea de nuevo.
a	abre o crea un fichero para añadir datos al final del mismo.
a+	abre o crea un fichero para leer y añadir datos al final del mismo.
r	abre un fichero de lectura.
r+	abre un fichero de lectura y escritura.

Modo binario

wb	crea un fichero de escritura. Si ya existe lo crea de nuevo.
w+b	crea un fichero de lectura y escritura. Si ya existe lo crea de nuevo.
ab	abre o crea un fichero para añadir datos al final del mismo.
a+b	abre o crea un fichero para leer y añadir datos al final del mismo.
rb	abre un fichero de lectura.
r+b	abre un fichero de lectura y escritura.

La función fopen devuelve, como ya hemos visto, un puntero de tipo FILE. Si al intentar abrir el fichero se produjese un error (por ejemplo si no existe y lo estamos abriendo en modo lectura), la función fopen devolvería NULL. Por esta razón es mejor controlar las posibles causas de error a la hora de programar. Un ejemplo:

```
FILE *pf;
pf=fopen("datos.txt","r");
if (pf == NULL) printf("Error al abrir el fichero");
```

freopen( )

Esta función cierra el fichero apuntado por el puntero y reasigna este puntero a un fichero que será abierto. Su sintaxis es:

```
freopen(nombre del fichero,"modo de apertura",puntero);
```

Donde nombre del fichero es el nombre del nuevo fichero que queremos abrir, luego el modo de apertura, y finalmente el puntero que va a ser reasignado.

CIERRE

Una vez que hemos acabado nuestro trabajo con un fichero es recomendable cerrarlo. Los ficheros se cierran al finalizar el programa pero el número de estos que pueden estar abiertos es limitado. Para cerrar los ficheros utilizaremos la función fclose( );.

Esta función cierra el fichero, cuyo puntero le indicamos como parámetro. Si el fichero se cierra con éxito devuelve 0.

```
fclose(puntero);
```

Un ejemplo ilustrativo aunque de poca utilidad:

```
FILE *pf;
pf=fopen("AGENDA.DAT","rb");
if ( pf == NULL ) printf ("Error al abrir el fichero");
else fclose(pf);
```

ESCRITURA Y LECTURA

A continuación veremos las funciones que se podrán utilizar dependiendo del dato que queramos escribir y/o leer en el fichero.

Un caracter

```
fputc( variable_caracter , puntero_fichero );
```

Escribimos un caracter en un fichero (abierto en modo escritura). Un ejemplo:

```
FILE *pf;
char letra='a';
if (!(pf=fopen("datos.txt","w"))) /* otra forma de controlar si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else fputc(letra,pf);
fclose(pf);
fgetc( puntero_fichero );
```

Lee un caracter de un fichero (abierto en modo lectura). Deberemos guardarlo en una variable. Un ejemplo:

```
FILE *pf;
char letra;
if (!(pf=fopen("datos.txt","r"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        letra=fgetc(pf);
        printf("%c",letra);
        fclose(pf);
}
```

Un número entero

```
fputw( variable_entera, puntero_fichero );
```

Escribe un número entero en formato binario en el fichero. Ejemplo:

```
FILE *pf;
int num=3;
if (!(pf=fopen("datos.txt","wb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fputw(num,pf); /* también podíamos haber hecho directamente: fputw(3,pf);
*/
        fclose(pf);
}
```

```
fgetw( puntero_fichero );
```

Lee un número entero de un fichero, avanzando dos bytes después de cada lectura. Un ejemplo:

```
FILE *pf;
int num;
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        num=getw(pf);
        printf("%d",num);
        fclose(pf);
}
```

Una cadena de caracteres

```
fputs( variable_array, puntero_fichero );
```

Escribe una cadena de caracteres en el fichero. Ejemplo:

```
FILE *pf;
char cad="Me llamo Vicente";
if (!(pf=fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fputs(cad,pf); /* o también así: fputs("Me llamo Vicente",pf); */
        fclose(pf);
}
```

```
fgets( variable_array, variable_entera, puntero_fichero );
```

Lee una cadena de caracteres del fichero y la almacena en variable_array. La variable_entera indica la longitud máxima de caracteres que puede leer. Un ejemplo:

```
FILE *pf;
char cad[80];
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fgets(cad,80,pf);
        printf("%s",cad);
        fclose(pf);
}
```

Con formato

```
fprintf( puntero_fichero, formato, argumentos);
```

Funciona igual que un printf pero guarda la salida en un fichero. Ejemplo:

```
FILE *pf;
char nombre[20]="Santiago";
int edad=34;
if (!(pf=fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fprintf(pf,"%20s%2d\n",nombre,edad);
        fclose(pf);
}
fscanf( puntero_fichero, formato, argumentos );
```

Lee los argumentos del fichero. Al igual que con un scanf, deberemos indicar la dirección de memoria de los argumentos con el símbolo & (ampersand). Un ejemplo:

```
FILE *pf;
char nombre[20];
int edad;
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fscanf(pf,"%20s%2d\",nombre,&edad);
        printf("Nombre: %s  Edad: %d",nombre,edad);
        fclose(pf);
}
```

Estructuras

fwrite(*buffer, tamaño, nº de veces, puntero_fichero);

Se utiliza para escribir bloques de texto o de datos, estructuras, en un fichero. En esta función, *buffer será la dirección de memoria de la cuál se recogerán los datos; tamaño, el tamaño en bytes que ocupan esos datos y nº de veces, será el número de elementos del tamaño indicado que se escribirán.

fread(*buffer, tamaño, nº de veces, puntero_fichero);

Se utiliza para leer bloques de texto o de datos de un fichero. En esta función, *buffer es la dirección de memoria en la que se almacenan los datos; tamaño, el tamaño en bytes que ocupan esos datos y nº de veces, será el número de elementos del tamaño indicado que se leerán.

Puedes encontrar ejemplos sobre la apertura y cierre de ficheros, así como de la lectura y escritura de datos, en el archivo IMAGECAT.C. Se trata de un programa que crea un catálogo en formato HTML a partir de las imágenes que se encuentran en un directorio determinado.

Otras funciones para ficheros

```
rewind( puntero_fichero );
```

Sitúa el puntero al principio del archivo.

```
fseek( puntero_fichero, long posicion, int origen );
```

Sitúa el puntero en la posicion que le indiquemos. Como origen podremos poner:

```
0 o SEEK_SET, el principio del fichero
1 o SEEK_CUR, la posición actual
2 o SEEK_END, el final del fichero
```

```
rename( nombre1, nombre2 );
```

Su función es exactamente la misma que la que conocemos en MS-DOS. Cambia el nombre del fichero nombre1 por un nuevo nombre, nombre2.

```
remove( nombre );
```

Como la función del DOS del, podremos eliminar el archivo indicado en nombre.

Detección de final de fichero

```
feof( puntero_fichero );
```

Siempre deberemos controlar si hemos llegado al final de fichero cuando estemos leyendo, de lo contrario podrían producirse errores de lectura no deseados. Para este fin disponemos de la función feof( ). Esta función retorna 0 si no ha llegado al final, y un valor diferente de 0 si lo ha alcanzado.

ASIGNACIÓN DINÁMICA DE MEMORIA

INTRODUCCIÓN

En este tema estudiaremos las posibilidades que ofrece el lenguaje C a la hora de trabajar dinámicamente con la memoria dentro de nuestros programas, esto es, reservar y liberar bloques de memoria dentro de un programa.

Además en este tema se introducirá el concepto de tipo abstracto de dato y la forma de dividir un gran programa en otros más pequeños.

ASIGNACIÓN DINÁMICA Y ESTÁTICA DE MEMORIA.

Hasta este momento solamente hemos realizado asignaciones estáticas del programa, y más concretamente estas asignaciones estáticas no eran otras que las declaraciones de variables en nuestro programa. Cuando declaramos una variable se reserva la memoria suficiente para contener la información que debe almacenar. Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (función main). Realmente las variables locales de las funciones se crean cuando éstas son llamadas pero nosotros no tenemos control sobre esa memoria, el compilador genera el código para esta operación automáticamente.

En este sentido las variables locales están asociadas a asignaciones de memoria dinámicas, puesto que se crean y destruyen durante la ejecución del programa.

Así entendemos por asignaciones de memoria dinámica,aquellas que son creadas por nuestro programa mientras se están ejecutando y que por tanto, cuya gestión debe ser realizada por el programador.

¿CÓMO SE RESERVA MEMORIA DINÁMICAMENTE?

El lenguaje C dispone, como ya indicamos con anterioridad, de una serie de librerías de funciones estándar. El fichero de cabeceras stdlib.h contiene las declaraciones de dos funciones que nos permiten reservar memoria, así como otra función que nos permite liberarla.

1.- Reserva de memoria

Las dos funciones que nos permiten reservar memoria son:

```
malloc (cantidad_de_memoria);
calloc (número_de_elementos, tamaño_de_cada_elemento);
```

Estas dos funciones reservan la memoria especificada y nos devuelven un puntero a la zona en cuestión. Si no se ha podido reservar el tamaño de la memoria especificado devuelve un puntero con el valor 0 o NULL. El tipo del puntero es, en principio void, es decir, un puntero a cualquier cosa. Por tanto, a la hora de ejecutar estás funciones es aconsejable realizar una operación cast (de conversión de tipo) de cara a la utilización de la aritmética de punteros a la que aludíamos anteriormente. Los compiladores modernos suelen realizar esta conversión automáticamente.

Antes de indicar como deben utilizarse las susodichas funciones tenemos que comentar el operador sizeof. Este operador es imprescindible a la hora de realizar programas portables, es decir, programas que puedan ejecutarse en cualquier máquina que disponga de un compilador de C.

El operador sizeof(tipo_de_dato), nos devuelve el tamaño que ocupa en memoria un cierto tipo de dato, de esta manera, podemos escribir programas independientes del tamaño de los datos y de la longitud de palabra de la máquina. En resumen si no utilizamos este operador en conjunción con las conversiones de tipo cast probablemente nuestro programa sólo funciones en el ordenador sobre el que lo hemos programado.

Por ejemplo, el los sistemas PC, la memoria está orientada a bytes y un entero ocupa 2 posiciones de memoria, sin embargo puede que en otro sistema la máquina esté orientada a palabras (conjuntos de 2 bytes, aunque en general una máquina orientada a palabras también puede acceder a bytes) y por tanto el tamaño de un entero sería de 1 posición de memoria, suponiendo que ambas máquinas definan la misma precisión para este tipo.

Con todo lo mencionado anteriormente veamos un ejemplo de un programa que reserva dinámicamente memoria para algún dato.

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

main()
{
int    *p_int;
float  *mat;

p_int = (int *)  malloc(sizeof(int));
mat   = (float *)calloc(20,sizeof(float));

if ((p_int==NULL)||(mat==NULL))
      {
            printf ("\nNo hay memoria");
            exit(1);
      }

/* Aquí irían las operaciones sobre los datos */
/* Aquí iría el código que libera la memoria */

}
```

Este programa declara dos variables que son punteros a un entero y a un float. A estos punteros se le asigna una zona de memoria, para el primero se reserva memoria para almacenar una variable entera y en el segundo se crea una matriz de veinte elementos cada uno de ellos un float. Obsérvese el uso de los operadores cast para modificar el tipo del puntero devuelto por malloc y calloc, así como la utilización del operador sizeof.

Como se puede observar no resulta rentable la declaración de una variable simple (un entero, por ejemplo, como en el programa anterior) dinámicamente, en primer lugar por que aunque la variable sólo se utilice en una pequeña parte del programa, compensa tener menos memoria (2 bytes para un entero) que incluir todo el código de llamada a malloc y comprobación de que la asignación fue correcta (ésto seguro que ocupa más de dos bytes).

En segundo lugar tenemos que trabajar con un puntero con lo cual el programa ya aparece un poco más engorroso puesto que para las lecturas y asignaciones de las variables tenemos que utilizar el operador *.

Para termina un breve comentario sobre las funciones anteriormente descritas. Básicamente da lo mismo utilizar malloc y calloc para reservar memoria es equivalente:

```
mat = (float *)calloc (20,sizeof(float));
mat = (float *)malloc (20*sizeof(float));
```

La diferencia fundamental es que, a la hora de definir matrices dinámicas calloc es mucho más claro y además inicializa todos los elementos de la matriz a cero. Nótese también que puesto que las matrices se referencian como un puntero la asignación dinámica de una matriz nos permite acceder a sus elementos con instrucciones de la forma:

NOTA: En realidad existen algunas diferencias al trabajar sobre máquinas con alineamiento de palabras.

```
mat[0] = 5;
mat[2] = mat[1]*mat[6]/67;
```

Con lo cual el comentario sobre lo engorroso que resultaba trabajar con un puntero a una variable simple, en el caso de las matrices dinámicas no existe diferencia alguna con una declaración normal de matrices.

2.- Liberación de la memoria.

La función que nos permite liberar la memoria asignada con malloc y calloc es free(puntero), donde puntero es el puntero devuelto por malloc o calloc.

En nuestro ejemplo anterior, podemos ahora escribir el código etiquetado como : /* Ahora iría el código que libera la memoria */

```
free (p_int);
free(mat);
```

Hay que tener cuidado a la hora de liberar la memoria.
Tenemos que liberar todos los bloque que hemos asignado, con lo cual siempre debemos tener almacenados los punteros al principio de la zona que reservamos. Si mientras actuamos sobre los datos modificamos el valor del puntero al inicio de la zona reservada, la función free probablemente no podrá liberar el bloque dememoria.

3.- Ventajas de la asignación dinámica.

Vamos a exponer un ejemplos en el que se aprecie claramente la utilidad de la asignación dinámica de memoria.

Supongamos que deseamos programar una serie de funciones para trabajar con matrices. Una primera solución sería definir una estructura de datos matriz, compuesta por una matriz y sus dimensiones puesto que nos interesa que nuestras funciones trabajen con matrices de cualquier tamaño. Por tanto la matriz dentro de la estructura tendrá el tamaño máximo de todas las matrices con las que queramos trabajar y como tenemos almacenadas las dimensiones trabajaremos con una matriz de cualquier tamaño pero tendremos reservada memoria para una matriz de tamaño máximo.
Estamos desperdiciando memoria. Una definición de este tipo sería:

```
typedef struct
      {
            float   mat[1000][1000];
            int     ancho,alto;
      } MATRIZ;
```

En principio esta es la única forma de definir nuestro tipo de datos. Con esta definición una matriz 3x3 ocupará 1000x1000 floats al igual que una matriz 50x50.

Sin embargo podemos asignar memoria dinámicamente a la matriz y reservar sólo el tamaño que nos hace falta. La estructurasería ésta.

```
struct mat
      {
            float   *datos;
            int     ancho,alto;
      };


 typedef struct mat *MATRIZ;
```

El tipo MATRIZ ahora debe ser un puntero puesto que tenemos que reservar memoria para la estructura que contiene la matriz en sí y las dimensiones. Una función que nos crease una matriz sería algo así:

```
MATRIZ  inic_matriz (int x,int y)
{
MATRIZ  temp;

temp = (MATRIZ) malloc (sizeof(struct mat));
temp->ancho = x;
temp->alto  = y;
temp->datos = (float *) malloc (sizeof(float)*x*y);

return temp;
}
```

En esta función se ha obviado el código que comprueba si la asignación ha sido correcta. Veamos como se organizan los datos con estas estructuras.

```
temp------->datos---------->x*x elementos
            ancho,alto
```

Esta estructura puede parecer en principio demasiado compleja, pero veremos en el siguiente capítulo que es muy útil en el encapsulado de los datos.

En nuestro programa principal, para utilizar la matrizdeclararíamos algo así:

```
main()
{
MATRIZ  a;

a = inic_matriz (3,3);
...
borrar_matriz(a);
}
```

Dónde borrar_matriz(a) libera la memoria reservada en inic_matriz, teniendo en cuenta que se realizaron dos asignaciones, una para la estructura mat y otra para la matriz en sí.

Otra definición posible del problema podría ser así.

```
typedef struct mat MATRIZ;

void inic_matriz (MATRIZ *p,int x,int y)
{

p->ancho = x;
p->alto  = y;
p->datos = (float *)malloc(sizeof(float)*x*y);
}
```

Con este esquema el programa principal sería algo como ésto:

```
main()
{
MATRIZ  a;

inic_matriz (&a,3,3);

.....

borrar_matriz (&a);
}
```

Con este esquema el acceso a la matriz sería *(a.datos+x+y*a.ancho), idéntico al anterior sustituyendo los puntos por flechas ->. En el siguiente capítulo se justificará la utilización de la primera forma de implementación frente a esta segunda. En realidad se trata de la misma implementación salvo que en la primera el tipo MATRIZ es un puntero a una estructura, por tanto es necesario primero reservar memoria para poder utilizar la estructura, mientras que en la segunda implementación, el tipo MATRIZ es ya en si la estructura, con lo cual el compilador ya reserva la memoria necesaria. En el primer ejemplo MATRIZ a; define a como un puntero a una estructura struct mat, mientras que en la segunda MATRIZ a; define a como una variable cuyo tipo es struct mat.

PRECOMPILADOR Y COMPILADOR

GNU cc (gcc) es el compilador del Proyecto GNU. Compila programas escritos en C, C++, Objective C, e incluso fortran (a través de g77). Próximamente estarán disponibles Pascal, Modula-2, Ada 9x y otros muchos más lenguajes de desarrollo.

gcc permite al programador el control total del proceso de compilación, compuesto de cuatro etapas:

* Preprocesado
* Compilación
* Ensamblado
* Enlazado

Se puede parar el proceso después de cada etapa para ir examinando los resultados. Acepta los dialectos del lenguaje C (ANSI-C y K&R), así como C++ y Objective C. Puede incorporar en el código binario información para el depurado, así como realizar diversas optimizaciones de código. Además es un compilador cruzado, o sea, se puede desarrollar código en un procesador para se ejecutado en otro con distinta arquitectura. Por último, gcc posee extensiones de los lenguajes C y C++ que mejoran la optimización y facilitan el trabajo, a cambio de perder portabilidad (cosa poco recomendable).

Un tutorial corto
Para coger familiaridad, empecemos con el típico ejemplo en C:

```
/*
 * hello.c - Mi primer programa de C
 */
#include <stdio.h>

int main(void)
{
   fprintf(stdout, "¡Hola, linuxero!\n");
   return 0;
}
```

Para compilar y ejecutar este programa:

```
$ gcc hello.c -o hello
$ ./hello
¡Hola, Zaralinuxero!
```

La primera orden le dice al gcc que compile y enlace el fichero fuente hello.c, generando el ejecutable indicado por el argumento -o hello. La segunda orden ejecuta el programa, obteniéndose como resultado la tercera línea.

Varias cosas han pasado sin darnos cuenta. Lo primero que ha hecho gcc es manda al preprocesador (cpp) el fichero hello.c para expandir macros e incluir el contenido de los ficheros #included. Seguido, se compila el código preprocesado en código objeto, para finalmente el linkador (ld) crea el hello binario.

Veamos este proceso paso a paso. Si usamos la opción -E paramos el proceso justo después de realizarse la precompilación:

```
$ gcc -E hello.c -o hello.i
```

Podemos observar en el fichero hello.i el resultado del preprocesado, con el fichero stdio.h incrustado dentro de él. Continuando con el proceso, usaremos la opción -c para parar despúes de realizar la compilación.

```
$ gcc -c hello.i -o hello.o
```

Ya tan sólo queda enlazar el fichero objeto con las librerías del sistema para generar el ejecutable:

```
$ gcc hello.o -o hello
```

Para saber el punto de arranque gcc se fija en la extensión del fichero de entrada.. Las extensiones más comunes reconocidas por gcc son:

Extensión	Tipo
.c	Código fuente Lenguaje C
.C,.cc	Código fuente Lenguaje C++
.i	Código fuente C Preprocesado
.ii	Código fuente C++ Preprocesado
.S,.s	Código fuente Lenguaje Ensamblador
.o	Código objeto compilado
.a,.so	Código de librería compilado

Aunque lo normal es crear un ejecutable a partir del código fuente, a veces resulta conveniente parar el proceso en un punto determinado. Un caso habitual es cuando queremos obtener el código objeto para integrarlo en una librería, por lo que no es necesario la etapa de enlace. Otro caso cuando queremos obtener el código del precompilado para chequear algún conflicto que tengamos entre los ficheros includes y nuestro código.

La mayor parte de los programas de C se generan partiendo de varios fuentes, códigos objeto y librerías. gcc tomará los ficheros que le damos de entrada, los procesará convenientemente, y los enlaza para generar un único ejecutable. Dar todos y cada uno de los ficheros que componen una aplicación puede resultar demasiado tedioso. Para facilitar esta tarea se suele utilizar la utilidad make que veremos más adelante.
Opciones comunes de la línea de comandos

La lista de opciones ocuparía varias páginas, por lo que sólo veremos las opciones más comunes:

Opción

Descripción

-o <file>

Nombre del fichero de salida. Si no se especifica al generar el ejecutable, el nombre por defecto será a.out

-c

Compilar sin enlazar

-DMAC=VAL

Define el macro MAC y le asigna el valor VAL

-IDIR

Sitúa el directorio DIR delante de la lista de directorios dónde buscar los ficheros includes.

-LDIR

Sitúa el directorio DIR delante de la lista de directorios dónde buscar las librerías.

-static

Enlaza con librerías estáticas. Por defecto enlaza las librerías dinámicamente.

-lFICH

Enlaza con la librería libFICH

-g

Incluye información estándar para depurado

-ggdb

Incluye información específica para el depurador gdb

-O

Optimiza el código compilado

-ON

Especifica un nivel de optimación del código entre 0 y 3

-ansi

Soporta el C ANSI/ISO estándar

-pedantic

Emite todos los avisos requeridos por el C ANSI/ISO

-pedantic-errors

Emite todos los errores requeridos por el C ANSI/ISO

-traditional

Soporta la sintáxis Kernighan & Ritchie del lenguaje C

-w

Suprime todos los mensajes de aviso (una mala idea)

-Wall

Emite todos los avisos que el gcc pueda generar.

-werror

Convierte todos los avisos en errores, lo que parará la compilación

-MM

Crea una lista de dependencias compatible con make

-v

Muestra los comandos utilizados en cada etapa de la compilación