DISCA
DEPARTAMENT D'INFORMÀTICA DE SISTEMES I COMPUTADORS

EL LENGUAJE DE PROGRAMACIÓN C

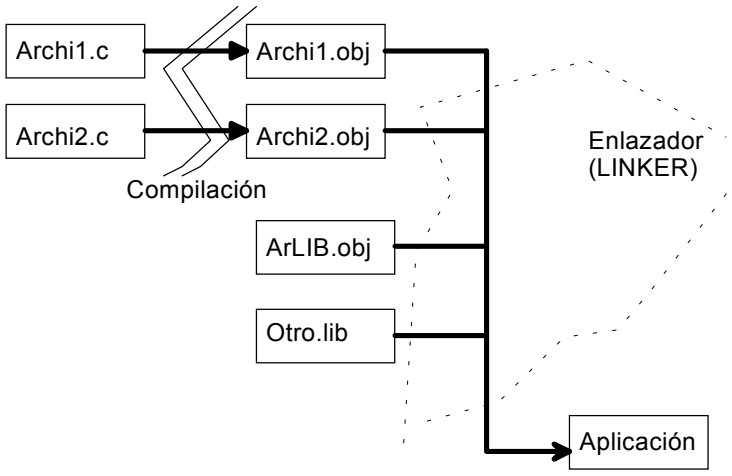
HISTORIA

El padre del lenguaje **C** es Dennis Ritchie, trabajador de los laboratorios BELL. En el ánimo de Ritchie no estaba el crear un lenguaje revolucionario, novedoso, con todo lo que un programador pudiera soñar y mucho más. El único objetivo de Ritchie era tener un lenguaje práctico y flexible para el desarrollo de los proyectos en los que estaba inmerso. Curiosamente el principal proyecto en el que se encontraba inmerso era la creación de un nuevo sistema operativo: UNIX.

Citando un párrafo de [K&R91] : "El **C** es un lenguaje de programación de propósito general que ofrece economía de expresión, estructuras de flujo y control modernos y un conjunto rico de operadores. El **C** no es un lenguaje de muy alto nivel, tampoco un gran lenguaje, y no está especializado en ningún área de aplicación. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y efectivo para muchas tareas que lenguajes supuestamente más poderosos".

Creación de un programa C

La creación de un programa escrito en C se divide en varias etapas que se pueden ver en la siguiente figura:



El primer paso es **crear** los ficheros que tienen nuestro código, que se guardarán con la extensión ".c". Como se puede apreciar en el dibujo, un solo programa se puede dividir en varios archivos siguiendo una serie de normas que veremos en los siguientes puntos.

El siguiente paso es **compilar** los archivos que componen nuestro programa, tras lo cual obtendremos un conjunto de archivos 'objeto'. Estos archivos suelen contener el código máquina del programa, pero todavía no son ejecutable.

La tarea de compilación se divide en dos en el caso de C: el preprocesamiento del código y la compilación.

En el pre procesamiento se modifican los archivos fuente de acuerdo con una serie de indicaciones y CONSTANTES DEL COMPILADOR. La salida de este proceso es un nuevo código fuente, sigue siendo C. Este código fuente obtenido es el que realmente se compila obteniendo el archivo objeto.

En los archivos objeto se puede hacer referencia a funciones o variables que están en otros módulos (archivos objeto) y no se han definido los espacios necesarios para las variables, el código del programa, la pila... Por estos motivos los archivos objeto no son ejecutables.

El siguiente y último paso es enlazar todos los archivos objeto de nuestro programa, creando el archivo ejecutable de la aplicación. Un programa escrito en C se compone de uno o más módulos.

Existen una serie de módulos que se encuentran en unos archivos especiales. Los archivos de librería, aquellos con extensión ".lib", no son más que una colección de pequeños archivos objeto, unidos para que sea más eficiente su búsqueda. Una colección de estas librerías acompaña a cada compilador de C.

Formato de un programa C

La estructura más habitual de un archivo C es la siguiente:

- Un pequeño comentario de descripción del fichero. (Punto 1)
- Ordenes para el pre procesador. (Punto 10)
- Definición de variables globales. (Punto 2)
- Definición de funciones. (Punto 6)

El lenguaje de programación C **distingue entre mayúsculas y minúsculas**.

A diferencia de otros lenguajes no hay una estructura estricta, los elementos mencionados pueden aparecer en cualquier orden. Tampoco hay una palabra reservada para indicar donde empieza la ejecución de un programa, aunque se reserva el nombre de función main() para este objetivo.

```
#include <stdlib.h>   /* Esto es un comentario */
int funcion(int a)
{
     if (a%2)
          /* no es multiplo de 2  */
          a+=3;
     else
          funcion2(&a);
     return a*5;
}

main()
{
  int a,b,c,d;
  b=7;c=14;

  a=funcion(b);
  printf("b(%4d) --> a(%4d)\n",b,a);
  a=funcion(c);
  printf("c(%4d) --> a(%4d)\n",c,a);
  a=funcion(127);
  printf(" (%4d) --> a(%4d)\n",127,a);
}
```

1 Comentarios

Los comentarios en C empiezan con "/*" y acaban con "*/". Se pueden comentar varias líneas a la vez. El efecto de un bloque de comentario es equivalente a eliminar el texto entre la definición de inicio y el primer fin de comentario. Por lo tanto:

NO SE PUEDEN ANIDAR COMENTARIOS

Esto error pueda dar lugar a errores en la compilación, que pueden ser difíciles de detectar si lo que se había comentado era un bloque de código. En cualquier caso hay que tenerlo en cuenta porque es fácil cometerlo.

Algunos compiladores admiten un segundo tipo de comentarios que sólo afectan a la línea en que se encuentran. Este tipo de comentario empieza por "//" y es efectivo hasta el final de la línea.

2 Tipos simples

El lenguaje C proporciona los siguientes tipos de datos básicos:

Tipo de dato	Significado
char	Entero de ocho bits con signo. Anteponiendo la palabra reservada *unsigned* tenemos un entero de ocho bits sin signo.
int	Entero con signo. La palabra reservada *unsigned* tiene el mismo efecto que sobre char. Los *int* pueden ser de 16 bits (*short*) o de 32 bits (*long*) anteponiendo esas palabras a la definición del tipo. Si sólo se pone *int* el tamaño de la variable dependerá del sistema y el compilador donde se este utilizando.
float	Número en coma flotante de simple precisión
double	Número en coma flotante de doble precisión

Se puede apreciar que el conjunto de tipos disponibles en C es reducido, con todas la opciones de tamaños y características tenemos un total de 8 tipos diferentes. En puntos siguientes veremos como se utilizan.

Los números enteros con signo utilizan codificación en complementa a 2. Por ejemplo un *char* que representa el valor -1 se guarda en memoria con una representación 1111 1111.

Al contrario que PASCAL ó MODULA2, C no hace comprobación estricta de datos. Se puede asignar un entero sin signo a uno con signo y viceversa.

2 Declaración de variables

La declaración de variables puede aparecer al principio, y sólo al principio, de cualquier bloque de código, por ejemplo al principio de un archivo. La variable definida **sólo** será válida dentro del bloque en que se ha definido. Los bloques de código que no son archivos, empiezan por una llave abierta '{' y acaban con una llave cerrada '}'. Así pues la definición de variables se puede hacer en un archivo o detrás de una llave abierta.

La sintaxis para la declaración de variables es:

modo_memoria tipo_de_dato nombre_de_var[,nombre_de_var] ;

- *[auto]* - [*unsigned*] *char*

- *static* - [*unsigned*] [*short* | long] int
- *extern* - *float*
- *register* - *double*

Las variables se pueden inicializar en el momento de su creación a un determinado valor. Esto quiere decir que el compilador insertará el código apropiado para que la primera vez que se use la variable tenga el valor adecuado. Si una variable no se inicializa no se debe hacer ninguna suposición sobre su valor.

Los valores asignados deben ser constantes. Los diferentes tipo de constantes son:

•Caracteres. Un carácter es una letra entre comillas simples: 'a', 'b'. El valor de esta constante es igual al valor ASCII del carácter representado.

•Constantes decimales. Cualquier numero entero que no empiece por cero. Si una constante decimal acaba con las letras **l** o **L** se supondrá que representa un valor largo. Si acaba con las letras **u** o **U** es que la constante representa un valor sin signo.

•Constantes octales. Números enteros que empiecen por 0 y con todas sus cifras menores que ocho.

•Constantes hexadecimales. Representan números enteros que empiecen por 0x. En la constante pueden aparecer números y las letras **a**, b**, c**, **d, e** y **f** (también es valido usar las mayúsculas).
Ejemplos de incialización:

```
char  a='a',
      b,c,d,
      f=0xff,  /* equivalencia: f=0xFF */
      g=127,
      t=077;
unsigned short int otro2= 0xfFFf,
                  uni,
                  cero=0;
long int a=1L;     /*Nos aseguramos que interpreta 1 como un long*/
```

Tras estas lineas tenemos siete variables de tipo `char` y tres del tipo `unsigned short int`. Por lo que el espacio total necesitado será de : (8*1+3*2) bytes.

Los modos de almacenamiento (**modo_memoria**) indican como reservar memoria para una variable. El significado de los principales es :

- `auto`. La variable se crea en la pila cada vez que se la ejecución pasa por su definición. Es eliminada de la pila cuando se sale del bloque donde se ha definido. Este es el modo que se toma por defecto si no se especifica otro.
- `static`. La variable siempre se encuentra en la misma posición de memoria. No tiene efecto sobre ella el entrar o salir del bloque de código donde se definió.
- `external`. La variable se ha definido en otro bloque de código. No se reserva espacio ninguno para ella, ya se hará en el bloque donde originalmente se había definido.

3 Operadores.

Los operadores básicos de C son :

=	asignación
+	suma
-	resta (o cambio de signo)
*	multiplicación
/	división

Los operadores mostrados trabajan con cualquiera de los tipos numéricos de C. Ni siquiera es necesario que los datos sean del mismo tipo. Cuando datos de distintos tipos se mezclan en una operación existe un problema de coherencia de tipos, problema que la estricta comprobación de tipos de PASCAL evita, pero que en C se permite.

Esta libertad de mezclar tipo es muy peligrosa, por lo que los compiladores de C tratan de detectarlas y avisan de que ocurren, pero sólo hacen eso avisar, es responsabilidad del programador decidir si lo permite o modifica el código.

Existen, sin embargo, una serie de "mezclas" que el compilador puede resolver con un conversión de tipo de alguna de las variables o constantes implicadas. Estos cambios siguen siendo peligrosas si el programador no es consciente de ellas. A continuación se expone una tabla de conversiones que realiza un compilador (sólo operadores aritméticos).

	char int(short) (signed)	**unsigned**	**long**	**float**	**double**
char int(short) (signed)	int	unsigned	long	float	double
unsigned	unsigned	unsigned	long	float	double
long	long	long	long	float	double
float	float	float	float	float	double
double	double	double	double	double	double

En la tabla se muestran los posibles tipos de los operandos usados y el resultado de la operación. Si tenemos varias operaciones el orden de preferencia de los operandos es el mismo que el de PASCAL.

El operador de asignación tiene dos características especiales:

1. Permite la concatenación de operaciones. Ejemplo

```
{
short a=100;
unsigned short int z,y,x;
z=y=x= a + 22;/* equivalente z=(y=(x=(a+22))) */
}
```

En este caso las variables *x*, *y* y *z* tendrán un valor final de 122. El operador devuelve el valor asignado a la variable.

2. Realiza un cambio automático entre tipos numéricos. Este cambio sólo es posible si no existen problemas de rango entre los dos tipos implicados. Por ejemplo:

```
{
```

```
unsigned short int z,y,x;
float a=1000000.98,b=-1.1,c=12.1;
z=a;/* oper. 1*/
y=b;/* oper. 2*/
x=c;/* oper. 3*/
}
```

De estas tres operaciones sólo obtendremos el valor esperado en la operación 3.

3.2 Operador módulo

Para obtener el resto de una división entera se puede usar el operador módulo **%**. Este operador se puede usar sólo cuando los dos operandos son de tipo entero. En caso contrario se recibe un error de tipos al intentar compilar el programa.

Este operador es equivalente al **MOD** de PASCAL o MODULA2.

3.3 Operadores a nivel de bit

En C existen una serie de operadores que trabajan a nivel de bit. Como el operador de módulo sólo se pudren usar con valores de tipo entero.

&	'Y' bit a bit
|	'O' bit a bit
^	'O exclusiva' bit a bit
~	negación bit a bit (sólo un operando)
>>	desplazamiento a derecha (con extensión de signo)
<<	desplazamiento a izquierda (con extensión de signo)

El operador desplazamiento a la derecha equivale a una división por una potencia de 2 y el operador desplazamiento a la izquierda a una multiplicación. El primer operando se desplaza tantos bits como se indiquen en el segundo operando.

Ejemplos:

```
{
unsigned short int z=12,y=128,x=0xf3ff,r2;
short a=0x0ff0,b=0xf3ff,c=0x0fff,r1; /* Por defecto signed */

r2= x >> 4;  /* resultado r2=0x0f3f */
r1= b >> 4;  /* resultado r2=0xff3f */

r1= y & b;/* resultado r1=128    */
r1= a | z;/* resultado r1=0x0ffc */
r1= a ^ c;/* resultado r1=0x000f */
r1= ~c;/* resultado r1=0xf00f */
}
```

3.6 Operadores de asignación especiales

Todos los operadores binarios anteriores (que necesitan dos operandos) se pueden combinar con el operador de asignación para obtener unos operadores que realizan la operación y luego la asignan sobre una variable. Como primer operando se toma esa variable. Es decir, se realiza la operación y luego se asigna.

Los operadores obtenidos son: +=, -=, *=, /= , %= , &=, |= , ^=, <<= , >>=.

Ejemplos:

```
{
```

```
unsigned short int z=12,y=128,x=0xf3ff,r2;
short a=0x0ff0,b=0xf3ff,c=0x0fff,r1; /* Por defecto signed */

x >>= 4;/* resultado x=0x0f3f */
z <<= 2;/* resultado z=48 */
a -= 0x00f0 ; /* resultado a= 0x0f00 */
y *= 5;/* y= 640*/
}
```

3.7 Operadores de incremento/decremento

Existen dos operadores cuyo funcionamiento depende del lugar en que se coloquen. Estos operadores siempre van ligados a una variable, pudiendo aparecer antes o después del nombre de dicha variable. Los operadores son :

++	auto incrementar
--	auto decrementar

Estos operadores van a incrementar, o decrementar, en una unidad la variable a la que acompañan. Esto se realiza siempre. Lo que cambia es valor que se devuelve como resultado de la operación. Si el operador está antes de la variable se devuelve el nuevo valor de la variable, es equivalente a un pre incremento, si el operador se coloca después de la variable el valor devuelto es el valor que tenía la variable antes.

Tomando como ejemplo el incremento y suponiendo que a=1 y b=3 las siguientes expresiones son equivalentes:

a=b++;	a=b; b+=1;
a=++b;	b+=1; a=b;

En el primer caso **a** acaba valiendo 3, mientras que en el segundo su valor final es 4. En ambos casos el valor final de **b** es 4.

NOTA: estos operadores son difíciles de utilizar al principio, pero son MUY usados en todos los programas de C, por ello es esencial que se comprendan BIEN.

3.8 Operador de expresiones

El operador "," se utiliza para separar expresiones. Las expresiones se evalúan todas pero sólo se devuelve su último valor. Este operador es útil en aquello lugares donde la sintaxis obliga a poner una sola expresión pero necesitamos colocar más de una.

Ejemplos:

```
{
short a=1,b=3,c=-12,d=25,e=1234;
a= b++,d+e; /* a tomara el valor de d+e y b se incrementa*/
}
```

3.9 Operador *sizeof()*

El operador sizeof() sirve para conocer el tamaño en bytes que ocupa una determinada variable. Su utilización es similar a cualquier función de PASCAL.

```
{
short a,b=3,c=-12,d=25,e=1234;
a= sizeof(b);/* a valdra ...*/
}
```

4 Tipos de datos compuestos.

El lenguaje C permite tres tipos de datos compuestos : las estructuras, las uniones y los vectores.

4.1 Enumeraciones

Además de los tres tipos compuestos añade un tipo especial: la **enumeración**. Este tipo sirve para definir constantes en un determinado rango de valores **enteros**. Es utilizada cuando se quieren definir una serie de etiquetas. Su sintaxis es:

```
enum EtiquetaTipo{
    EtiConstante1=valor_1,
    EtiConstante2,
    EtiContstante3=valor_3
} var_1,var2;
```

Tras este código podremos usar cada una de la etiquetas o nombres definidos como si fuesen la constante numérica que representan. El valor de cada una de ellas se toma del valor asignado o de la etiqueta anterior sumándoles uno.

Para definir nuevas variables de este tipo enumerado en otro lugar se puede hacer con la siguiente sintaxis:

```
enum EtiquetaTipo vari1,vari2=EtiConstante1;
```

Las variables de tipo enum son en realidad variables enteras por lo que no todos los compiladores avisarán si se les asigna un valor que no es igual a alguna de las etiquetas. Sin embargo es una forma elegante de definir valores enteros constantes.

4.2 Estructuras

Las **estructuras** son el equivalente en C a los registros de PASCAL. Para su definición se utiliza el siguiente patrón:

```
struct ETI_NOMBRE{
      definición_tipo campo1,campo2;
      definición_tipo campo1,campo2;
      } variable1,variable2;
```

La definición empieza con la palabra reservada `struct` para indicar el inicio de una estructura. Tras esta palabra puede aparecer una etiqueta o identificador del tipo de estructura. Este identificador servirá para poder crear variables de este tipo en otros lugares del código.

Las llaves indican el inicio y el fin de la definición de los campos de la estructura. La forma de definición de los campos es la misma que la de definición de cualquier variable.

A diferencia de PASCAL, la definición de tipos y variables no está separada, por lo que, tras la llave de cierre, que indica que se ha acabado la definición del tipo, se pueden declarar variables. Estas variables se usaran para acceder a la estructura.

En memoria los campos que componen una estructura ocupan posiciones consecutivas en memoria. El tamaño total de una estructura es la suma de los tamaños de cada uno de sus campos.

Ejemplo:

```
struct MIO1{
      short int a,b;
      long  b1,b2
      } var1;         /* los structs no pueden inicializarse */
....
struct MIO1 var2,var3,var4;
static MIO1 var6;             /* INCORRECTO */
static struct MIO1 var7;      /* CORRECTO   */
...
var2.a=var1.b+var3.a;
```

```
...
var2.b1=var1.b1+var3.b3;
...
```

Se puede ver en el ejemplo la definición de la estructura MIO1. La estructura se compone de cuatro campos. Tras la definición de la estructura se declara la variable `var1`. En otro punto del código se quieren definir nuevas variables de este tipo, para ello se vuelve a colocar la palabra `struct`, el nombre del tipo de estructura y las variables que se quieren declarar.

ES IMPRESCINDIBLE COLOCAR LA PALABRA `struct`

Las estructuras tienen un uso especial: la definición de **campos de bits**. Al definir una estructura podemos 'empaquetar' una información en unos pocos bits. Como ejemplo:

```
struct Pal1 {
unsigned nibble1:4;
unsigned nibble2:4;
} variable;
```

Esta estructura ocupara un total de 8 bits. Lo que no podemos saber es cual de los dos campos ocupa la parte alta y cual la parte baja de esos ocho bits. Se accede a los bits como si fueran campos de una estructura normal.

4.3 Uniones

Las **uniones** son muy parecidas a las estructuras. La diferencia fundamental entre una estructura y una unión es que, mientras en la primera cada campo ocupada una posición de memoria diferente, en las uniones todos los campos ocupan la misma posición de memoria, es decir, todos empiezan en la misma posición. El tamaño de una unión se define como el tamaño del mayor de sus campos

Son útiles cuando queremos acceder a posiciones de memoria cuyo contenido debe ser interpretado con tipos diferentes. Como ejemplo:

```
union Palabra32{
      struct {
            unsigned char  a,b,c,d;
      } octetos;
      unsigned long  todo;
      struct{
            unsigned short bajo,alto;
      } doble_octeto;
} vari_32;
```

En esta unión tenemos tres campos : `octetos`, `todo` y `doble_octeto`. Si se calcula se verá que todos estos tres campos tienen una lóngitud de 32 bits, luego la unión ocupará precisamente eso 32 bits.

La alineación de cada campo depende de la máquina en que se esté trabajando, por lo que no se puede asegurar que al acceder a `vari_32.octetos.a` se vaya a acceder al primer byte de la estructura o que este coincida con byte de mayor peso de `vari_32.todo` .

Si posteriormente queremos definir nuevas variables del tipo `union Palabra32` tendremos que poner tanto el nombre de la unión como la palabra `union`. Ejemplo:

```
union Palabra32  a,b;
union { unsigned short a; short b} var1;
```

La variable var1 es de un tipo de unión al que no le hemos dado nombre. Más tarde no podremos definir variables de este tipo sin tener que repetir la definición del mismo.

Un uso adicional de las uniones es la de definir variables de distinto tipo pero que ocupan la misma posición de memoria. Para hacerlo se construye una unión sin nombre de la que no se definen variables, lo que se denomina una unión anónima:

```
union { unsigned short a; signed short b};
/* utilizamos a y b */
a = 65535;
b = b+1;
```

En este código se utiliza la misma posición de memoria para almacenar las variables a y b.

- Escribe un programa y comprueba el valor de a y b tras el código anterior

4.4 Vectores

Los **vectores** o *arrays* se definen como una variable normal, pero indicando el número de elementos que van a contener.

Ejemplo:

```
int a[10];  /* a es un vector de 10 enteros */
```

Para acceder a los elementos de un vector se utiliza el nombre del vector y entre corchetes el índice del elemento a acceder.

IMPORTANTE: los índices de los vectores en C empiezan siempre en cero, por lo que el rango de valores posibles es desde cero hasta S-1, siendo S el valor de elementos con el que se ha definido el vector.

CUIDADO: C no hace comprobación de rango al acceder a elementos de un vector. Podemos acceder a elementos fuera del vector. Está 'cualidad' puede ocasionar problemas de violación de espacio de memoria en alguno sistemas operativos.

Para crear vectores de varias dimensiones lo que se crean son vectores de vectores, es decir, un vector cuyos elementos son vectores:

```
int a[5][10]
```

Esta definición crea un vector de 5 componentes, con cada componente siendo un vector de diez enteros. Para acceder a un elemento usaríamos la expresión :

```
a[1][9]=0;
```

Podemos inicializar los elementos de un vector:

```
int a[7]={1,2,3,4,5,6,7};
float b[3][5]={{ 1.0 ,0.0 ,0.1, 0.75, 0.75},
               { 0.0 ,1.1,2.1,3,1000},
               {1,1,1,1,1}
               };
```

Si un vector se va a inicializar se puede dejar **su última dimensión** sin definir. Las definiciones siguientes son equivalentes a las anteriores :

```
int a[]={1,2,3,4,5,6,7};
float b[3][]={{ 1.0 ,0.0 ,0.1, 0.75, 0.75},
               { 0.0 ,1.1,2.1,3,1000},
```

```
                {1,1,1,1,1}
                };
```

Sin embargo la siguiente definición NO es válida:

```
    float b[][5]={
                { 1.0 ,0.0 ,0.1, 0.75, 0.75},
                { 0.0 ,1.1,2.1,3,1000},
                {1,1,1,1,1}
                };
```

5 Control de ejecución.

Pasamos a describir las construcciones de control de programa: las estructuras de selección, el operador condicional y las estructuras de repetición. Para poder hacerlo se van a utilizar expresiones lógicas y sentencias.

Las expresiones lógicas, a partir de ahora expresiones, serán la base para tomar decisiones sobre el flujo de un programa. Las sentencias son las acciones realizadas por un programa.

5.1 Sentencias

Existen dos tipos de sentencias, las simples y las compuestas. Las sentencias simples son una línea de ejecución, como una suma o una llamada a una función. Una sentencia simple acaba en punto y coma.

Las sentencias compuestas están delimitadas por llaves, son un bloque de código. Al principio de una sentencia compuesta puede aparecer la definición de nuevas variables.

Una sentencia compuesta está compuesta por una combinación de sentencias simples y sentencias compuestas. Tras la llave final de una sentencia compuesta no se pone punto y coma.

Existe una sentencia simple especial denominada sentencia vacía. La existencia de esta sentencia permite poner un punto y coma tras la llave de fin de una sentencia compuesta o tras otro punto y coma.

CUIDADO: Tras la llave de fin de una función NO VA PUNTO Y COMA.

5.2 Expresiones Lógicas

Una Expresión Lógica en C es **cualquier expresión entera**. Si el resultado de la expresión es 0, se considera FALSO y si es distinto de 0, se considera VERDADERO.

En una expresión puede aparecer el operador "**,**". En este caso el valor de toda expresión es igual al valor de la última expresión de la cadena de comas.

Los operadores relacionales en C devuelven por tanto estos valores enteros, y son:

==	cierto (distinto de 0) si iguales
!=	cierto si distintos
>	mayor
>=	mayor o igual

<	menor
<=	menor o igual

Los operadores lógicos son:

&&	'Y' lógico: cierto si y solo si ambos ciertos
||	'O' cierto si alguno cierto.
!	negación (no)

Los operadores lógicos y los operadores bit-a-bit no tienen el mismo significado lógico.

5.3 Estructuras de selección

La estructura condicional simple tiene la siguiente sintaxis :

```
if (expresión)
      sentencia
[else
      sentencia]
```

La parte de "else" es opcional. Se aconseja utilizar siempre sentencias compuestas (entre llaves) en la estructura condicional. De esta forma el código quedará mucho más legible.
Además de la estructura condicional simple existe una estructura de selección múltiple. A diferencia de PASCAL las comparaciones han de ser fijas, es decir, las entradas para la selección múltiple deben de ser valores constantes y no expresiones.

```
switch (valor)
{
      case valor1:      sentencias1
                  break;
      case valor2:      sentencias2

      case valor3:      sentencias3
                  break;
      .....

      [default:   sentencias;]
}
```

Cuando la ejecución llega al "*switch*" se evalúa *valor*, siguiendo la ejecución en la sentencia asociada al ese valor de entre los especificados en las sentencias "*case*". Si no aparece la palabra reservada "*break*" al final de la sentencia asociada a un valor, se continuará con la sentencia asociada al siguiente valor de la lista.

Si la evaluación de *valor* no es igual a ninguno de los valores de la lista se ejecutara la sentencia asociada a *default*.

5.4 Operador condicional

```
(expresión1) ? expresión2 : expresión3
```

El valor de la expresión es expresión2 si expresión1 es cierto en otro caso es expresión3.

Ejemplo:

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

asigna a **c** el mínimo de **a** y **b**;

Este operador forma parte de lo que algunos autores denominan taquigrafía de C. Formas de escribir expresiones que tratan de dar el máximo significado con la cantidad mínima de caracteres. Algunas no son muy usuales y su utilización es mejor reservarlas para cuando se está seguro. Sin embargo hay que conocer su sintaxis para poder 'leer' algunos programas.

5.5 Estructuras de repetición

En C existen tres estructuras de repetición o tipos de bucles diferentes :

```
while (expresión)
      sentencia   /* SI simple debe acabar en ";" */


do
      sentencia   /* SI simple debe acabar en ";" */
while (expresión);


for (sent_inicialización; expresion_test; sent_actualización)
      sentencia   /* SI simple debe acabar en ";" */
```

Tanto el bucle while como el do..while tienen un equivalente en PASCAL. El bucle *for*, sin embargo, se comporta de manera diferente al de PASCAL. Para entender su funcionamiento hay que tener en cuenta que es equivalente a:

```
inicialización;         /* Una sentencia simple    */
while (test){           /* test es una expresión */
      sentencia
      actualización;    /* Una sentencia simple */
}
```

Recordar que donde hay una sentencia simple se pueden colocar varias separándolas con el operador ','.

Existen dos palabras reservadas que permiten modificar el funcionamiento normal de las estructuras de repetición:

- *break* permite salir de cualquier bucle. Sólo sale de un bucle cada vez.
- *continue* permite ignorar el resto de sentencias hasta el final de un bucle y comenzar una nueva vuelta. **Vuelve al punto de test del bucle en ejecución.**

6 Declaración de funciones

Existen dos sintaxis diferentes para definir funciones en C: la de K&R y la estándar ANSI. Hoy día se utiliza más el estándar ANSI, sin embargo en las prácticas a realizar usaremos la de K&R, ya que la teoría se ve con este formato y es la que más difiere de PASCAL.

Las funciones se pueden definir en cualquier parte de un fichero, pero nunca dentro de otra función.

LAS FUNCIONES NO PUEDEN CONTENER OTRAS FUNCIONES.

La declaración de una función según K&R:

```
tipo_del_resultado nombre_de_la_función(nombre_de_los_parámetros)
declaración de los parámetros;
{
      declaración_de_variables_locales;

      sentencias;
...
      return a;
....
      return b;
}
```

y según el estándar ANSI:

```
tipo_del_resultado nombre_de_la_función(declaración de los parámetros)
{
      declaración_de_variables_locales;
      sentencias;
...
      {     /* Nuevo bloque de sentencias */
            int a=5;
            a=a+2;      /* En este bloque a es una variable entera */

      }
      /* Cuando salimos del bloque a vuelve a ser tipo_del_resultado */
...
      return  a;
}
```

Los parámetros siempre se pasan por valor. Si queremos simular un paso por referencia tendremos que usar punteros, más adelante se verá como se tratan en C los punteros.

Es conveniente, aunque no necesario, declarar las funciones antes de usarlas. La declaración difiere entre el C de K&R y el estándar, pero en general consiste en decir que la función existe e informar de al menos el tipo devuelto por la función.

Los archivos cabecera, archivos con extensión ".h", suelen incluir estas declaraciones. Como ejemplos de archivos cabecera ampliamente utilizados se sugiere buscar y leer en el sistema los archivos "stdio.h" y "stdlib.h"

7 Punteros.

Como en otros lenguajes, un puntero contiene la dirección de memoria en la que se encuentra un dato. Las operaciones básicas con punteros son dos:

Operador *: Lo apuntado por. (contenido del puntero)

Operador &: La dirección de. (devuelve un puntero)

Para definir un puntero, se define el tipo de lo apuntado:

```
int *a;     /* Lo apuntado por a es de tipo entero */
            /* luego a es un puntero a int.        */
```

Podemos asignar direcciones a punteros con el operador dirección:

```
int main(void)
{
      int i, *p;
      i = 5;
      p = &i; /* p apunta a i. */
      printf("Lo apuntado por p vale %d\n",*p);
      *p = *p+1;
      printf("El valor de i es ahora %d\n",i);
      return 0;
}
```

Sin embargo C añade nuevas características a los punteros, todas ellas relacionadas con lo que se denomina ***aritmética de punteros***. La aritmética de punteros consiste en que a un puntero se le pueden sumar y restar valores enteros, dando como resultado un nuevo puntero. Así si a un puntero a entero le sumamos dos, le estamos indicando que queremos aumentar el puntero en tantas unidades como sea falta para desplazarnos dos enteros en memoria. Esto es muy útil para el acceso a memoria a bajo nivel.

El trabajo con *arrays* y punteros es muy similar, de hecho un *array* queda definido por la dirección del primer elemento y por el tipo de los componentes. Así el siguiente código:

```
{
int a[10];
a[5]=a[0];
}
```

podríamos reescribirlo como:

```
{
int a[10];
*(a+5)=*a;  /* *a es lo mismo que *(a+0)  */
}
```

Veamos otro ejemplo:

```
int main(void)
{
      int i, a[10], *p;

      for(i=0;i<10;i++)
            a[i]=i; /* inicializamos el array */

      p = a; /* p apunta a a[0].
              * notese que poner 'a' es lo mismo que '&a[0]'
              */

      printf("Elemento 5: %d, %d, %d\n",a[5],*(p+5), p[5]);
      for (i=0;i<10;i++){
printf("El elemento %d vale %d\n",i,*p++);
}
      return 0;
}
```

Un bucle típico para copiar un vector definido sobre un tipo simple es :

```
{
      int a[10],b[10],*c,*d;
...
/* En algun siti ose inicializa a */
```

```
      /* Quiero copiar a en b           */
      for(i=0,c=a,d=b;i<10;i++)
            *d++=*c++;
      }
```

8 Tipos de datos definidos por el usuario

La forma de definir variables que representan estructuras, uniones y vectores descrita anteriormente, es equivalente al mecanismo de PASCAL para definir una variable y al asignarle un tipo definir en ese instante la estructura de la misma. En PASCAL este método se utiliza sólo cuando se requieren pocas variables con esa estructura.

PASCAL	C
VAR A: RECORD c1,c2: INTEGER; c3 : REAL; END;	struct { int c1,c2; float c3 } A;

¿Qué ocurre cuando, por ejemplo, se quiere definir en multitud de lugares de mi programa vectores sobre el mismo tipo y todos de la misma longitud?

En PASCAL se soluciona declarando un nuevo tipo de datos. En C no existe esta posibilidad como tal, pero se permite crear *sinónimos* para tipos de datos. Los sinónimos no son nuevos tipos de datos estrictamente hablando, pero permiten ahorrar el escribir el nombre completo del tipo, incluyendo `struct` o `union` si son necesarias. La sintaxis es:

typedef tipo_de_dato nombre_de_tipo;

El nuevo nombre de tipo se coloca donde iría el nombre de la variable a definir. Ejemplos:

```
typedef int MiTipo[8];
typedef struct TIPOESTRUCTURA{ int a,b,c} TipoEstructura;

MiTipo a;
struct TIPOESTRUCTURA dosVar;
TipoEstructura unaVar;
```

9 Cambio de tipo de una variable o "*casting*"

Al principio de este punto se comentó que el compilador decidía el tipo de un resultado en función de los operandos implicados en la misma, para evitar problemas de compatibilidad de tipos. Excluyendo estos cambios automáticos el compilador nos avisara cuando los operandos de una operación o los parámetros de una función no coincidan.

Existen situaciones en las que es "deseable" saltarse esta comprobación de tipos, para tales situaciones se dispone de los " cast operators", los operadores de cambio de tipo.

La forma de estos operadores es muy simple, se antepone a una expresión el nombre del tipo al que se quiere llegar entre parentesis. De esta forma se interpreta la posición de memoria ocupada por la expresión como si contuviera el otro tipo.

Ejemplo:

```
int main(void)
{
short int i, a[]={1001,1002,1003,1004,1005,6,7,8,9,-1}, b[20];
char *a1;

a1=(unsigned char*)a;   /* Conversión con cast */
printf("Tamanyos %d   %d\n",sizeof(*b),sizeof(*a1));
for (i=0;i<20;i++,a1++)
          b[i]=*a1;           /* Conversión automática */
```

```
}
```

Cómo lo que se hace es interpretar la posición de memoria no se traduce su contenido para que el valor siga siendo el mismo. Este hecho provoca efectos laterales que en ocasiones no se esperaban, por ello no se deben utilizar más que cuando sea estrictamente necesario.

Ejercicio. Hacer un cast desde un puntero a un valor númerico sin signo de su mismo tamaño. Que crees que representa el valor así obtenido. Realiza un pequeño programa en C.

10 Ordenes del pre procesador

Estas ordenes modifican el texto de entrada al compilador, es decir, el efecto de las mismas es sobre el texto del archivo, por lo que podríamos sustituirlas por el texto al que equivalen.

Todas empiezan por el carácter "#", que debe estar obligatoriamente en la primera columna.

Las ordenes de preprocesador más usadas son :

```
#include "fichero"
#include <fichero>
#define etiqueta valor
#define macro(parametros) sentencias
```

La orden *#include* inserta un fichero dentro de otro. Su uso más común es insertar **ficheros de cabecera**. Los ficheros de cabecera, con extensión ".h", contienen la definición de variables, tipos y funciones cuya implementación se encuentra en otro archivo. Es normal que las primeras líneas de código en cualquier programa C sean una serie de ordenes #include.

La orden #define permite definir etiquetas de texto que se sustituirán con su valor cada vez que aparezcan en el texto. Las etiquetas pueden ser cualquier combinación de caracteres alfanuméricos que empiecen por una letra o por "_".

Otro uso de esta orden es el compilado selectivo. Permite escoger trozos de código según el valor de una etiqueta, o incluso según una etiqueta está definida o no. Para conseguirlo se utilizan otras ordenes como #ifdef, #endif o #elif. Ejemplo:

```
#ifdef _INTEL_8086_
      ....
#elif _MOTOROLA_68000
      ....
#endif
```

Hay que tener especial cuidado con la orden #define cuando se define una macro con parámetros. Debe de tenerse presente que las ordenes de pre procesado sólo actúan sobre el texto que finalmente se compilará, sustituyendo el texto de la etiqueta por el texto equivalente de valor, sustituyendo los caracteres pasados como parámetros donde se indique. Esto puede tener efectos laterales no deseados.

Como ejemplo intentar evaluar el siguiente código:

```
#define CUADRADO(a)  a*a
#define CUADRAD(a)  (a)*(a)
...
b  =  CUADRADO(c+1) + 2 ;  /* Sutituido : c+1*c+1 +2      */
b2 =  CUADRAD(c+1)    + 1;   /* Sustituido: _____________ */
```

Los operadores de incremento son un claro ejemplo de los efectos que pueden tener las macrossi noi se las usa debidamente.

11 Tiras de caracteres.

Una tira de caracteres se define en C como un *array* de caracteres. Por convenio se indica el final de la tira con el valor 0 (cero). El acceso a los caracteres se puede hacer como elementos de un *array* o con punteros.

Ejemplo:

```
/*El siguiente programa copia una tira de caracteres en otra:*/
#include <stdio.h>
int main(void)
{
char *tira="Esto es una tira";
char tira2[80], *tira3, *tira4;

tira3 = tira; /* No nos engañemos, tira3 y tira apuntan
         * a los mismos datos, pero solo hay una
         * copia de estos
         */
tira4 = tira2; /* trabajaremos con punteros en vez de índices */
while ((*tira4++ = *tira3++) != 0); /* bucle vacio */

/* la tira ha sido copiada */
/* ahora vamos a modificar la original y la copia para poder
   distinguirlas */
/* vamos a volver a copiar punteros para verificar que realmente
   lo que se copia son punteros y no los datos */
tira3 = tira;
tira4 = tira2;
tira[0]='1'; /* esto no es muy elegante ya que la tira original
      * era una constante y por tanto no deberia ser
      * modificada, pero en este caso funciona, y sirve
      * para nuestros fines
      */
tira2[0]='2';
printf("tira:  %s\ntira2: %s\ntira3: %s\ntira4: %s\n",
tira,tira2,tira3,tira4);
return 0;
}
```

En la librería de C existen funciones diseñadas para trabajar con tiras de caracteres. Para acceder a estas hay que incluir el fichero *string.h*.

Alguna de estas funciones son:

```
char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
int strcmp(const char *s1, const char*s2);
```

El valor devuelto por esta última función hay que interpretrarlo :

si < 0 significa que s1 < s2
si == 0 significa que s1 == s2
si > 0 significa que s1 > s2

NOTA: En el directorio /home/acso1/practicas/cadenas/ se pueden encontrar el código fuente del ejemplo propuesto y de otros programas que muestran la utilización de estas funciones.

Ejemplo:

```
/* strcpy example */
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10];
```

```
   char *str1 = "abcdefghi";

   strcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

12 E/S.

El acceso a la librería standard de entrada-salida se realiza con la inclusión del fichero *stdio.h*. Podemos trabajar con la entrada y salida standard (teclado y pantalla) o sobre ficheros en disco.

Entrada/Salida standard :

```
      int printf(const char *format [, argument, ...]);
int scanf(const char *format [, address, ...]);
      int putchar(int ch);
int getchar(void);
      int puts(const char *s);
      char *gets(char *s);
```

La gestión de ficheros se verá como parte de la asignatura ACSO 2.

12 Variables dinámicas

Los punteros pueden utilizarse para trabajar con variables "dinámicas", es decir, variables cuya memoria se va reservando en tiempo de ejecución.

En C se dispone de las siguientes llamadas de reserva de memoria:

```
      void *malloc(size_t size);
      void *calloc(size_t nitems, size_t size);
      void *realloc(void *block, size_t size);
      void free(void *block);
```

Hay que incluir los ficheros *stdlib.h* y *alloc.h*

Ejemplos:

```
/* malloc example */

#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>

int main(void)
{
    char *str;
    /* allocate memory for string */
    if ((str = (char *) malloc(10)) == NULL)
    {
       printf("Not enough memory to allocate buffer\n");
       exit(1);  /* terminate program if out of memory */
    }
    /* copy "Hello" into string */
    strcpy(str, "Hello");
    /* display string */
    printf("String is %s\n", str);

    /* free memory */
    free(str);
}
```

```
/* calloc example */

 #include <stdio.h>
 #include <alloc.h>
 #include <string.h>

 int main(void)
 {
    char *str = NULL;

    /* allocate memory for string */
    str = (char *) calloc(10, sizeof(char));

    /* copy "Hello" into string */
    strcpy(str, "Hello");

    /* display string */
    printf("String is %s\n", str);

    /* free memory */
    free(str);
}
```

REFERENCIAS

"El lenguaje de programación C"

Brian W. Kernighan
2ª Edición 1991
De. Prentice-Hall Hispanoamericana
Biblioteca: 4-64/1357B

"Curso de programación C"

Francisco Javier Ceballos Sierra
Ed. ra-ma
Biblioteca: 4-64/901