Capítulo

6

Aspectos avanzados

En el sexto capítulo se describen otros aspectos avanzados del lenguaje como un tipo especial de estructura llamada *unión,* el manejo de bits, el preprocesador, la entrada/salida con ficheros y los punteros a funciones.

Uniones (*union)*

Hay otro tipo de estructuras que son las uniones (*unions*). También están formadas por campos, pero en este caso todos los campos están en la misma posición de memoria, reservando el compilador memoria suficiente para almacenar el mayor de los campos (miembros).

La declaración de las mismas es idéntica a la de las estructuras:

```
union varios
{
      int numero;
      float real;
      char caracter;
} ;
```

Y su uso también, ya que se utilizará el operador ".", para variables y el operador "->" para punteros a uniones (en el ejemplo anterior la *union* tendría el tamaño del *float*).

Es responsabilidad del programador obtener el tipo de variable adecuado que está almacenado en la estructura. No se puede almacenar un char e intentar obtener un float. Además si se inicializa, sólo puede ser con el tipo del primer miembro de la unión.

Normalmente este tipo de datos se usan para almacenar variables que pueden ser de distintos tipos o en cuestiones de bajo nivel (nivel hardware).

Manejo de bits

El C es un lenguaje de medio nivel que se realizó para sustituir, si es posible, al ensamblador, por lo que debe permitir controlar casi todos los aspectos de la máquina y proporcionar herramientas para poder trabajar en el ámbito del bit.

Estos bits sólo se pueden usar en bytes o palabras de la máquina donde esté implementado el compilador, por lo que sólo se pueden utilizar en los tipos de datos *char* (byte) e *int* (palabra).

Los operadores de bits del C son los siguientes:

Operador	Significado	Ejemplo
~	Negación	~(11111111) == (00000000)
&	AND	23 & 11 == 3
|	OR	65 | 54 == 119
^	XOR	62 ^ 75 == 117
<<	Desplazamiento a la izquierda	(10010101) << 2 == (01010100)
>>	Desplazamiento a la derecha	(10010101) >> 2 == (00100101)

Tendremos las operación básicas (no confundir con las operaciones lógicas) y además desplazamientos en ambos sentidos, para lo cual se introducen ceros en el sentido contrario.

Otra forma de trabajar con bits es construyendo estructuras donde cada miembro, llamado campo, es una agrupación de bits. Detrás de él se pone su dimensión en bits:

```
struct {
      unsigned enable : 1;
      unsigned strobe : 2;
      unsigned allow  : 1; } registro;
```

Una variable que se defina como *registro* (podría ser el registro de un controlador) se almacenará en un entero (palabra natural), pero ocupará 4 bits: *enable* 1, *strobe* 2 y *allow* 1. Se puede acceder a los bits con el operador pertenencia "." pero sólo se podrán utilizar valores binarios (0 ó 1):

```
      registro.allow = 0;
      registro.strobe = 3;     /* es 11 en binario */
```

También se pueden poner etiquetas vacías para separar bits y etiquetas nulas para pasar al siguiente entero (es útil cuando accedemos a registros hardware de ese formato):

```
struct {
      unsigned enable :  1;
                      :  2;     /* hueco de 2 bits */
      unsigned strobe :  2;
                      :  0;     /* nulo: siguiente entero */
      unsigned allow  :  1; } registro;
```

en el primer byte habrá 3 bits útiles (el 0 y el 3,4) y en el segundo sólo 1 (el 0).

El preprocesador

El proceso de compilación de un programa C, en términos generales (ver apéndices), consta de dos fases: por un lado está la acción del preprocesador, y por otro, la propia compilación. En la primera, se transforma el código fuente para que pueda ser compilado, fundamentalmente sustituyendo constantes simbólicas, añadiendo o quitando código según unas determinadas condiciones y añadiendo los ficheros de cabecera. El carácter "#" es típico de las instrucciones del preprocesador y es antepuesto en todas ellas.

Para la definición de constantes se utiliza la instrucción del preprocesador *#define*, que consta de dos partes, por un lado los caracteres que se tienen que encontrar en el código fuente y por otro

lado por lo que se tienen que sustituir (como estilo se suelen usar las mayúsculas para definir las constantes simbólicas del preprocesador y las minúsculas para el código C). Un ejemplo de definición sencillo sería:

```
#define MAXIMO 100
```

El analizará el código y siempre que se encuentre *MAXIMO* lo sustituirá por 100 antes de la compilación. Hay una excepción a esta regla y es cuando esos caracteres están encerrados entre comillas en el código, en tal caso no se producirá dicha sustitución.

Como no son instrucciones de C, no tienen que terminar en punto y coma, aunque acompañándolas se pueden usar comentarios y operaciones (aunque dichas operaciones no se realizarán en el preprocesado).

Si la definición a sustituir es muy larga, se puede utilizar el carácter \ para continuar en otra línea. Además esa sustitución se puede realizar en función de otros #define anteriores.

```
#define     DOS         2
#define     CUATRO      Dos * Dos
#define     Mensaje     "Esto será incorporado"
#define     Px          printf("x es %d.\n",x)
```

También se pueden definir macros con los *#define* para ello se dan los argumentos de las mismas a continuación:

```
#define CUADRADO(x)     x * x
```

de tal manera que cuando se encuentre *cuadrado* se sustituirá el argumento por el *argumento * argumento*:

```
z = CUADRADO(x);        ->          z = x * x;
z = CUADRADO(2);        ->          z = 2 * 2;
```

Y también se pueden hacer indefiniciones de los *#define* con el comando *#undef*, sobre todo para grandes programas en los que se tienen que redefinir temporalmente determinadas constantes simbólicas.

Otro comando muy útil, que ya hemos utilizado, es el que se usa para hacer inclusiones de ficheros. Cuando el preprocesador se encuentra el comando *#include*, tomará el fichero que viene a continuación y lo incluirá en el código fuente. De tal manera que podemos hacer un *#include* de un fichero que a su vez tenga *#define* dentro.

El fichero se puede delimitar por <> que indicará que se busque en el directorio definido por defecto en el proceso de compilación o por " " que lo buscará en el propio directorio.

También se puede realizar la inclusión o exclusión de código, que será muy útil en la depuración de un programa y en la adaptación a distintos compiladores. Se utilizan los comandos del preprocesador: *#if, #else, y #endif*, e *#ifdef, #ifndef*.

Las tres primeras instrucciones son sentencias condicionales con la misma función que las del lenguaje, pero se diferencian de ellas en que la expresión será una constante definida por el preprocesador con #define (que puede valer 0 o algo distinto de 0). Si es 0, el código C entre ellas no se compilará. Las dos últimas son idénticas a las anteriores, con la diferencia que sólo comprueban la existencia de la definición de la constante simbólica y no su valor:

```
#if COMPILADOR == "ANSI C"
   #include "este.fichero"
#endif
```

```
#ifdef COMPILADOR
   #include "datos.h"
#else
   #include "otrosdatos.h"
#endif
```

En ANSI C hay algunas macros predefinidas (nótense los dos subrayados):

```
Macro           Description
__DATE__        The current date as a character literal in "MMM DD YYYY" format
__TIME__        The current time as a character literal in "HH:MM:SS" format
__FILE__        This contains the current filename as a string literal.
__LINE__        This contains the current line number as a decimal constant.
__STDC__        Defined as 1 when the compiler complies with the ANSI standard.
```

Entrada / salida con ficheros

Podemos considerar un fichero como una zona de almacenamiento permanente, normalmente en disco, a la que nos referimos a través de un nombre (se supone que el lector está familiarizado con el manejo de ficheros). Para el lenguaje C internamente los ficheros son una estructura (*struct*) que se declara en la librería *stdio.h*, en la cual, se hace un *#define* de la estructura, al identificador *FILE* (también hay otros sistemas que definen el *FILE* como un *typedef*). Este identificador será usado para definir una variable de tipo puntero a FILE, que es la que usaremos en el programa para referirnos al fichero.

Las operaciones que se pueden hacer con los ficheros son las habituales. Así, tendremos operaciones para abrir y cerrar ficheros, para leer y escribir en ellos distintos tipos de datos y para poder movernos (acceso aleatorio) en su información.

La función para abrir ficheros es la *fopen*, que será de tipo puntero a FILE (puntero a *struct*), pero que no hay que declarar donde se utilice porque ya está hecho en *stdio.h*. Esta función admite dos argumentos: el nombre del fichero y el modo de apertura, ambos como tira de caracteres (los dos pueden ser punteros a *char*). El modo puede ser: "r" para sólo lectura, "w" para escritura (borrando el fichero si existe) y "a" para escritura pero añadiendo datos, no borrando el fichero existente. Existen otros modos menos usuales como "r+", "w+", "a+" para hacer ambas cosas. También existen los modos equivalentes para ficheros binarios: "rb", "wb", "ab" y, "r+b", "w+b", "a+b".

En todos los modos, si la operación falla se devolverá el valor NULL (definido en *stdio.h*). La utilización es:

```
FILE *fichero;
fichero = fopen("nombre", "r");
```

La función para cerrar ficheros es la *fclose*, que tomará como argumento un puntero a fichero. Es una función entera, por lo tanto, si la operación es correcta, devolverá un 0 y si falla un -1.

```
fclose(fichero);
```

Estas funciones trabajan con ficheros con buffer o *stream* (almacenamiento intermedio en memoria), de tal manera que al cerrarlo, se vacía el buffer al fichero[1]. Cuando un programa termina normalmente (final de *main* o llamada a *exit*) se cierran todos sus ficheros automáticamente, pero esto no ocurre así cuando el programa termina anormalmente, por lo que la información del buffer asociado al fichero sin cerrar se pierde, por eso siempre es conveniente cerrarlos cuando se dejen de usar en el programa.

Al igual que existen funciones para la entrada/salida por los canales estándar, debido a la equivalencia entre dispositivos y ficheros dada por el sistema operativo, también existen funciones para la entrada/salida por ficheros, de hecho son las mismas con distintos nombres (la entrada estándar es *stdin*, la salida es *stdout*, y la de error *stderr*).

[1] Se hace así por eficiencia, si tratamos de leer o escribir datos de longitud pequeña, el manejo asociado a esas operaciones puede ser costoso en tiempo y es más eficaz escribir en memoria estas variables y después pasar de un solo golpe toda esa memoria al fichero.

Para la salida (escritura) de distintos tipos de datos está la equivalente a *printf* que se llama *fprintf*. Esta función tiene tres argumentos, el primero es el puntero a fichero, el segundo la tira de control y el tercero las variables a escribir.

```
fichero = fopen("nombre","w");
int edad = 7;
fprintf(fichero, "Pedro tiene %d meses \n", edad);
```

Para la lectura de ficheros, como se intuye, está la función *fscanf*, con la cual hay que tener las mismas precauciones que con *scanf* en lo que se refiere a la lectura de strings y caracteres. También lo que se lee con ella son punteros a variables, no las propias variables, por lo que se deberá utilizar el operador de dirección "&".

```
fichero = fopen("nombre","r");
int edad, *punt_entero;
fscanf(fichero, "%d", &edad);
fscanf(fichero, "%d", punt_entero);
```

Por la misma razón que existen las funciones *getchar* y *putchar*, también existen sus equivalentes *getc* y *putc* para ficheros, a las cuales habrá que darlas como último argumento el puntero a fichero.

```
ch = getc(fichero);
putc(ch, fichero);
```

Por último, al igual que la entrada/salida de strings, también existe la equivalencia para ficheros; estas funciones son *fgets* y *fputs*. La primera tiene como primer argumento el puntero a carácter (tira de caracteres) donde se quiere realizar la lectura, después el número de caracteres que se quieren leer y por último el puntero a fichero. La segunda tiene como primer argumento la tira (puntero) de caracteres que se quiere escribir y como último argumento el puntero a fichero.

Como ejemplo de utilización de ficheros, vamos a ver como con las dos funciones de caracteres podemos ya hacer un “vuelca el contenido” de un fichero (el fichero si es de lectura debe existir):

```
#include <stdio.h>
main ()
{
      FILE *entrada;    /* fichero de entrada */
      int ch;
      /* abre fichero para lectura que se llama nombre */
      if ( (entrada = fopen("nombre", "r")) != NULL)
      {
            /* lee hasta final de fichero */
            while ( (ch = getc(entrada)) != EOF)
                  /* stdout es la pantalla */
                  putc(ch, stdout);
            fclose(entrada);
      }
      else
            puts("No se puede abrir");
}
```

Además *fgets* es de tipo puntero a carácter y devuelve un NULL si encuentra el EOF y a diferencia de *gets* no convierte el carácter '\n', y *fputs* devuelve un EOF si hay un error, quita el nulo del final, pero no añade el carácter '\n'.

Su uso sería:

```
fichero = fopen("nombre","r");
(fgets(tira, 81, fichero) != NULL);

fichero = fopen("nombre","a");
control = fputs("Ya era hora", fichero);
```

Hay otras funciones menos importantes para trabajar con ficheros:

- *rewind*() coloca el índice del fichero al comienzo del mismo y toma como argumento un puntero a fichero.
- *remove*() borra un fichero y también toma como argumento el fichero.
- *rename ()* toma como argumentos dos tiras de caracteres con el nombre viejo y el nuevo y cambia el nombre.
- *feof()* nos dice si para el fichero indicado se ha llegado a su final.
- *ferror()* nos dice si para el fichero indicado ha habido algún error.
- *fflush()* para el fichero indicado fuerza que el bufer interno se vuelque a fichero.
- *fseek*() que sirve para moverse por el fichero. Tiene tres argumentos: un puntero a fichero, un offset de tipo *long* (distancia donde se busca, positiva o negativa) y un modo de búsqueda. El modo puede ser 0 para el comienzo del fichero, 1 para la posición actual y 2 para el final del fichero, por lo que el offset se toma con referencia a este modo. La función devuelve un 0 si todo va bien.

```
fseek(fichero, 32L, 1);
```

Punteros a funciones

Hemos dicho que un puntero es sólo una variable que contiene una dirección. Pero la dirección puede apuntar o bien a un dato o bien a código. Si es código la forma natural de referenciarlo es con una función, de ahí que existan punteros a funciones.

La forma de definirlos empieza a ser algo barroca pero sintetizando la podemos resumir en:

```
<tipo> (*<identificador>)(<lista_de_parámetros>);
```

Ejemplos:

1. Declara un puntero a función “func” que no devuelve nada y tomas dos argumentos de tipo puntero a entero:

```
void (*func) (int *,int *);   /* declaraciones */
```

2. Declara un puntero, "pfuncion1" a una función que devuelve un int y no acepta parámetros:

```
int (*pfuncion1)(void);
```

3. Declara un puntero, "pfuncion2" a una función que no devuelve valor y que acepta un parámetro de tipo int.

```
void (*pfuncion2)(int);
```

4. Una función que devuelve un puntero a float y admite dos parámetros: un puntero a char y un int.

```
float *(*pfuncion3)(char*, int);
```

5. Declara una función "pfuncion4" que no devuelve nada y acepta un parámetro. Ese parámetro debe ser un puntero a una función que tampoco devuelve valor y admite como parámetro un int.

```
void (*pfuncion4)(void (*)(int));
```

6. Declara un array de punteros a función, cada una de ellas devuelve un int y admite como parámetro un int.

```
int (*pfuncion5[10])(int);
```

El uso de los punteros en una asignación o una llamada sería (el propio nombre de una función es un puntero):

```
func = intercambia;          /* asignación de valor */
(*func)(&a,&b);              /* llamada */

pfuncion1 = funcion;         /* asignación de valor */
int x = pfuncion1;           /* llamada */
```

Dada la relativa complejidad de la declaración de un puntero a función, en algunos casos para simplificarlo y usarlo posteriormente se usan los *typedef*:

```
int una_función(char);       /* entera con char */
typedef int (*PUN_FUN)(char);   /* tipo de dato */
PUN_FUN la_funcion = una_funcion;   /* uso del tipo */
```

Una de las utilidades de los punteros a funciones es el poder pasarles como argumentos de otras funciones (*callback*) para que sean más genéricas. Por ejemplo, la función `qsort` definida en ANSI C (stdlib.h) toma cuatro argumentos y tendría un prototipo como:

```
void qsort(void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void *));
```

Básicamente ordena un array `base` de cualquier tipo, con `nmemb` elementos y `size` tamaño de elemento, pero necesita una función para comparar los elementos, que precisamente se pasa como último argumento. Como se ve, toma dos punteros a comparar que no se pueden modificar y devuelve un entero. Una función para enteros podría ser:

```
int int_cmp(const void *p1, const void *p2)
{
    int a = *(int *) p1;
    int b = *(int *) p2;

    if (a > b)
        return 1;
    if (b < a)
        return -1;
    else
        return 0;
}

/* otra manera */
int cmpfunc (const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}
```

Ejemplos

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

/* prototipo de la funcion para comparar enteros */
int int_cmp(const void *p1, const void *p2);

int main()
{
    /* puntero a la function int_cmp */
    int (*cmp)(const void *, const void *) = int_cmp;

    int arr[]  = {5, 4, 3, 2, 10, 1, 7};
    int nmemb = 7;
    int i;

    for (i=0; i<nmemb; i++)
        printf("%d ", arr[i]);


    // los tres casos son equivalentes
    // qsort(arr, nmemb, sizeof(int), int_cmp);
    // qsort(arr, nmemb, sizeof(int), &int_cmp);
    qsort(arr, nmemb, sizeof(int), cmp);


    printf("\n");
    for (i=0; i<nmemb; i++)
        printf("%d ", arr[i]);

    return 0;

}

```