TEMA 31

LENGUAJE C: CARACTERÍSTICAS GENERALES. ELEMENTOS DEL LENGUAJE. ESTRUCTURA DE UN PROGRAMA. FUNCIONES DE LIBRERÍA Y USUARIO. ENTORNO DE COMPILACIÓN. HERRAMIENTAS PARA LA ELABORACIÓN Y DEPURACIÓN DE PROGRAMAS EN LENGUAJE C.

ÍNDICE

1. INTRODUCCIÓN

El lenguaje C se puede considerar como una alternativa al lenguaje ensamblador más que un lenguaje de alto nivel.

Esto se debe a sus especiales características. C es un lenguaje que ofrece recursos análogos a los de un ensamblador, permitiendo accesos y operaciones que por lo general no son posibles con lenguajes de alto nivel, ya que éstos están definidos hacia tareas concretas, gestión, cálculo matemático, etc., mientras que C es un lenguaje diseñado para la creación de herramientas informáticas. Sin embargo, C es un lenguaje que permite abordar cualquier tarea de programación.

2. ELEMENTOS DEL LENGUAJE C

2.1. Caracteres de C

Letras, dígitos y carácter de subrayado

- Letras mayúsculas del alfabeto inglés

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Letras minúsculas del alfabeto inglés

a b c d e f g h i j k l m n o p q r s t u v w x y z

- Dígitos decimales

0123456789

- Carácter de subrayado (_)

Estos caracteres son utilizados para formar las *constantes*, los *identificadores* y las *palabras clave* de C.

El compilador C trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo, los identificadores Pi y PI son diferentes.

Espacios en blanco. Espacio en blanco, tabulador Horizontal (HT), tabulador vertical (VT), avance de página (FF), y nueva línea (LF o CR + LF) son caracteres denominados *espacios en blanco*, porque la labor que desempeñan es la misma que la del espacio en blanco, es decir, actúan como separadores entre los elementos de un programa. Los espacios en blanco en exceso son ignorados por el compilador, lo cual nos permite escribir programas más legibles.

Caracteres especiales y signos de puntuación. Se utilizan de diferentes formas y son los siguientes:

, . ; : ? ‘ “ () [] {} < ! | / \ ~ + # % & ^ * - = >

Secuencias de escape. Los caracteres también pueden ser representados por secuencias de escape. Una secuencia de escape está formada por el carácter \ seguido de una "letra" o de una "combinación de dígitos". Son utilizadas para acciones como nueva línea, tabular y para representar caracteres no imprimibles.

El lenguaje C tiene predefinidas las siguientes secuencias de escape:

SECUENCIA	NOMBRE
\n	Nueva línea
\t	Tab horizontal
\v	Tab vertical (sólo para impresora)
\b	Backspace (retroceso)
\r	Retorno de carro
\f	Alimentación de página (sólo para impresora)
\a	Bell (alerta, pitido)
\’	Comilla simple
\”	Comilla doble
\\	Backslash (barra invertida)
\ddd	Carácter ASCII. Representación octal
\xdd	Carácter ASCII. Representación hexadecimal

2.2. Tipos de datos

Hay dos clases de tipos: tipos fundamentales y tipos derivados.

Tipos fundamentales. Los podemos clasificar en:

Tipos enteros: char, short, int, long y enum.

Tipos reales: float, double y long double.

Otros: void.

Cada tipo entero puede ser clasificado por las palabras clave "signed" o "unsigned", lo que da lugar a tener disponibles los siguientes tipos extras:

signed char, unsigned char

signed short, unsigned short

signed int, unsigned int

signed long, unsigned long

Un entero calificado "signed" es un entero con signo, esto es, un número entero positivo o negativo. Un número entero calificado "unsigned" es un número entero sin signo, el cual es manipulado como un número entero positivo.

Tipos derivados. Los tipos derivados se construyen a partir de los tipos fundamentales. Algunos de ellos son los siguientes: punteros, estructuras, uniones, arrays y funciones.

Un **puntero** es una dirección de memoria que indica dónde se localiza un objeto de un tipo especificado. Para definir una variable de tipo puntero se utiliza el operador de indirección *.

Ejemplo: int *p;

Este ejemplo declara un puntero p a un valor entero.

Variables. La sintaxis correspondiente a la declaración de una variable es la siguiente:

tipo identif, ...

"tipo" determina el tipo de la variable (char, int, float, double, ...)

"identif" indica el nombre de la variable.

Una variable declarada fuera de todo bloque (conjunto de sentencias encerradas entre {}) es, por defecto, *global* y es accesible en el resto del fichero fuente en el que está declarada. Por el contrario, una variable declarada dentro de un bloque, es por defecto *local* y es accesible solamente dentro de éste.

Declaración de constantes. A la declaración de un objeto, se puede anteponer el calificador "const", con el fin de hacer que dicho objeto sea, en lugar de una variable, una constante.

Ejemplo: const int k = 12;

1.3. Operadores

Los operadores son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, lógicos, relacionales, unitarios, lógicos para manejo de bits, de asignación, operador ternario para expresiones condicionales y otros.

Operadores aritméticos

Operador	Operación
+	Suma. Los operandos pueden ser enteros o reales
-	Resta. Los operandos pueden ser enteros o reales
*	Multiplicación. Los operandos pueden ser enteros o reales
/	División. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real
%	Módulo o resto de una división entera. Los operandos tienen que ser enteros

Operadores lógicos

Operador	Operación
&&	**And**. Da como resultado el valor lógico 1 si ambos operandos son distintos de cero. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado
||	**Or**. El resultado es 0 si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1. Si el primer operando es distinto de cero, el segundo operando no es evaluado
!	**Not**. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario

Operadores de relación

Operador	Operación
<	Primer operando "menor que" el segundo
>	Primer operando "mayor que" el segundo
<=	Primer operando "menor o igual que" el segundo
>=	Primer operando "mayor o igual que" el segundo
==	Primer operando "igual que" el segundo
!=	Primer operando "distinto que" el segundo

Operadores unitarios

Operador	Operación
-	Cambia de signo al operando (complemento a dos). El operando puede ser entero o real
~	Complemento a 1. El operando tiene que ser entero (carácter ASCII 126)

Operadores lógicos para manejo de bits

Operador	Operación
&	Operación AND al nivel de bits
|	Operación OR al nivel de bits (ASCII 124)
^	Operación XOR al nivel de bits
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha

Operadores de asignación

Operador	Operación
++	Incremento
--	Decremento
=	Asignación simple
*=	Multiplicación más asignación
/=	División más asignación
%=	Módulo más asignación
+=	Suma más asignación
-=	Resta más asignación
<<=	Desplazamiento a izquierdas más asignación
>>=	Desplazamiento a derechas más asignación
&=	Operación AND sobre bits más asignación
|=	Operación OR sobre bits más asignación
^=	Operación XOR sobre bits más asignación

Otros operadores

Operador coma. Un par de expresiones separadas por una coma se evalúa de izquierda a derecha. Todos los efectos de la expresión de la izquierda son ejecutados antes de evaluar la expresión de la

derecha, a continuación el valor de la expresión de la izquierda es descartado. El tipo y el valor del resultado son el tipo y el valor del operando de la derecha.

Ejemplo:

```
aux = v1, v1 = v2, v2 = aux;

for (a = 256, b = 1; b < 512; a/=2, b*=2)
```

Ejemplo:

```
fx(a, (b = 4, b+3), c, d)
```

Hay cuatro argumentos, de los cuales el segundo tiene un valor 7.

Operador de indirección (*****). Este operador accede a un valor indirectamente a través de un puntero. El resultado es el valor direccionado por el operando.

Operador de dirección-de (**&**). Este operador da la dirección de su operando. Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el calificador "register".

Ejemplo:

```
int *pa, a=2, b;      /* pa es un puntero a un valor entero */
pa = &a;              /* en el puntero pa se almacena */
                      /* la dirección de a */
b = *pa;              /* a b se le asigna el valor almacenado en la dirección especificada por pa */
```

2.4. Conversión de tipos

Cuando los operandos dentro de una expresión son de tipos diferentes, se convierten a un tipo común, de acuerdo con las reglas que se exponen a continuación.

1. Cualquier operando de tipo "float" es convertido a tipo "double".

2. Si un operando es de tipo "long double", el otro operando es convertido a tipo "long double".

3. Si un operando es de tipo "double", el otro operando es convertido a tipo "double".

4. Cualquier operando de tipo "char" o "short" es convertido a tipo "int".

5. Cualquier operando de tipo "unsigned char" o "unsigned short" es convertido a tipo "unsigned int".

6. Si un operando es de tipo "unsigned long", el otro operando es convertido a "unsigned long".

7. Si un operando es de tipo "long", el otro operando es convertido a tipo "long".

8. Si un operando es de tipo "unsigned int", el otro operando es convertido a tipo "unsigned int".

3. ESTRUCTURA DE UN PROGRAMA

Un programa fuente C es una colección de cualquier número de directrices para el compilador, declaraciones, definiciones, expresiones, sentencias y funciones.

Todo programa C debe contener una función denominada **main ()**, donde el programa comienza a ejecutarse. Las llaves ({}) que incluyen el cuerpo de esta función principal, definen el principio y el final del programa.

Un programa C, además de la función principal main (), consta generalmente de otras funciones que definen rutinas con una función específica en el programa. Esto quiere decir que la solución de cualquier problema, no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de elementos naturales del problema mismo, abstraídos de alguna manera, que darán lugar al desarrollo de las funciones mencionadas.

Directriz **# include**. La directriz # include "fichero" le dice al compilador que incluya el fichero especificado, en el programa fuente. Esto es necesario porque estos ficheros aportan, entre otras declaraciones, las funciones prototipo de las funciones de la librería estándar que utilizamos en nuestros programas.

Directriz **# define**. Mediante la directriz # define "identificador valor" se le indica al compilador, que toda aparición en el programa de "identificador", debe ser sustituida por "valor".

Declaraciones y definiciones. Toda variable debe ser declarada antes de ser utilizada. En general, las variables no son inicializadas por C, pero si se desea, pueden ser inicializadas en la propia declaración.

La definición de una variable, declara la variable y además le asigna memoria; la definición de una función, declara la función y además incluye el cuerpo de la misma.

Sentencias. Una sentencia es la unidad ejecutable más pequeña de un programa C. Las sentencias controlan el flujo u orden de ejecución. Una sentencia C consta de una palabra clave (for, while, if ... else, etc.), expresiones, declaraciones, o llamadas a funciones.

Toda *sentencia simple* termina con un punto y coma (;).

Dos o más sentencias pueden aparecer sobre una misma línea, separadas por punto y coma.

Una *sentencia nula* consta solamente de un "punto y coma".

Sentencia compuesta o bloque. Una sentencia compuesta o bloque, es una colección de sentencias incluidas entre ({}). Un bloque puede contener otros bloques.

Funciones. Una función es una colección de sentencias que ejecutan una tarea específica. Una función no puede contener a otra función.

Declaración de una función. La declaración de una función, también conocida como "función prototipo", tiene la forma:

Tipo_resultado nombre_función ([lista de tipos de argumentos]);

Ejemplo:

float conversión (int c);

Se observa, que en la declaración de la función, se dan sus características pero no se define su contenido. Una función puede ser declarada implícitamente o con una "declaración forward" (*función prototipo*).

Definición de una función. La definición de una función consta de una cabecera de función y del cuerpo de la función encerrado entre llaves:

```
Tipo_resultado nombre_función ([parámetros formales])
{
declaraciones de variables locales;
sentencias
[return(expresión)]
}
```

Las variables locales declaradas dentro del cuerpo de la función, por definición solamente pueden utilizarse dentro del mismo.

El tipo_resultado especifica qué tipo de datos retorna la función. Éste, puede ser cualquier tipo fundamental, o tipo definido por el usuario, pero no puede ser un array o una función. Por defecto, el valor retornado es int. Este valor es devuelto a la sentencia de llamada, por medio de la sentencia:

```
return (expresión)
```

Esta sentencia puede ser o no la última, y puede aparecer más de una vez en el cuerpo de la función. En el caso de que la función no retorne un valor, se omite.

Los *parámetros formales* de una función son las variables que reciben los valores de los argumentos en la llamada a la función; consisten en una lista de identificadores con sus tipos, separados por comas.

Pasando argumentos a funciones. Cuando se llama a una función, el valor del primer parámetro actual es pasado al primer parámetro formal, el valor del segundo parámetro actual es pasado al segundo parámetro formal y así sucesivamente. Todos los argumentos, excepto los arrays, son pasados *por valor*. Esto es, a la función se pasa una copia del argumento, no su dirección. Esto hace que la función C, no pueda alterar los contenidos de las variables pasadas.

Si se desea poder alterar los contenidos de los argumentos en la llamada, entonces hay que pasarlos *por referencia*. Es decir, a la función, se pasa la dirección del argumento y no su valor por lo que el parámetro formal correspondiente tiene que ser un puntero. Para pasar la dirección de un argumento, utilizaremos el operador &.

4. INSTRUCCIONES EN C

4.1. Sentencia de asignación

Una sentencia de asignación tiene la forma:

variable operador_de_asignación expresión

Ejemplo:

```
total = 0;
área = 3,141592 * r * r;
cuenta + = 1;
```

La sentencia de asignación es asimétrica. Esto quiere decir que la expresión de la derecha es evaluada, y el resultado es asignado a la variable especificada a la izquierda. De acuerdo con esta definición, no sería válida la sentencia:

```
3,141592 * r * r = área;
```

Si la variable es de tipo "puntero", solamente se le puede asignar una dirección de memoria, la cual será siempre distinta de 0. Un valor 0 (se escribe NULL) sirve para indicar que esa variable puntero no apunta a un dato válido.

4.2. Entrada y salida standard

Las operaciones de entrada y salida no forman parte del conjunto de sentencias de C, sino que pertenecen al conjunto de funciones de la librería estándar de C. Por ello, todo fichero fuente que utilice funciones de entrada/salida correspondientes a la librería estándar de C, necesita de las funciones prototipo correspondientes a éstas, por lo que deberá contener la línea:

 # include <stdio.h>

Salida con formato. Función printf(). La función printf() escribe con formato, una serie de caracteres, o un valor, en el fichero de salida estándar stdout. Esta función devuelve un valor entero igual al número de caracteres escritos.

#include <stdio>
*int printf (const char * formato [,argumento]...);*

- *formato*: especifica como va a ser la salida. Está formado por caracteres ordinarios, secuencias de escape y especificaciones de formato. El formato se lee de izquierda a derecha. Cada argumento debe tener su correspondiente especificación y en el mismo orden. Si hay más argumentos que especificaciones de formato, los argumentos en exceso se ignoran.

- *argumento*: representa el valor o valores a escribir.

Entrada con formato. Función scanf(). La función scanf() lee datos de la entrada estándar stdin, los interpreta de acuerdo con el formato indicado y los almacena en los argumentos especificados. Cada argumento debe ser un puntero a una variable cuyo tipo debe corresponderse con el tipo especificado en el formato. Esta función devuelve un entero correspondiente al número de datos leídos y asignados de la entrada. Si este valor es cero, significa que no han sido asignados datos. Cuando se intenta leer un end-of-file (marca de fin de fichero) la función scanf() retorna un EOF, constante definida en el fichero stdio.h.

#include <stdio.h>
*int scanf (const char * formato [,argumento]...);*

- *formato*: interpreta cada dato de entrada. Está formado por caracteres en blanco (“, \t, \n), caracteres ordinarios y especificaciones de formato. El formato se lee de izquierda a derecha. Cada argumento debe tener su correspondiente especificación de formato y el mismo orden. Si un carácter en la entrada estándar no se corresponde con la entrada especificada por el formato, se interrumpe la entrada de datos.

- *argumento*: es un puntero a la variable que se quiere leer.

Ejemplo:

int a; float b; char c;

sentencia	entrada de datos
scanf (“%d %f %c”, &a, &b, &c);	5 23.4 b
scanf (“%d, %f, %c”, &a, &b, &c);	5, 23.4, b
scanf (“%d: %f: %c”, &a, &b, &c);	5: 23.4 :b

Entrada de caracteres. Getchar(). Lee un carácter de la entrada estándar stdin y avanza la posición de lectura al siguiente carácter a leer.

Ejemplo:

car = getchar(); / lee un carácter y lo almacena en la variable car */*

Salida de caracteres. Putchar(). Escribe un carácter en la salida estándar stdout en la posición actual y avanza a la siguiente posición de escritura.

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

Esta función devuelve el carácter escrito, o un EOF si ocurre un error.

Ejemplo:

```
putchar (car);     /* escribe el carácter contenido en la variable car */
```

4.3. Sentencias de control

Sentencia if. Toma una decisión referente a la acción a ejecutar en un programa, basándose en el resultado (verdadero o falso) de una expresión.

```
if (expresión)
sentencia1;
[else
sentencia2]:
```

expresión	debe ser una expresión numérica, relacional o lógica. El resultado que se obtiene al evaluar la expresión es "verdadero" (no-cero) o "falso" (cero).
sentencia l/2	representan una sentencia simple o compuesta. Cada sentencia simple debe estar separada de la anterior por un punto y coma.

Anidamiento de sentencias if. Las sentencias "if ... else" pueden estar anidadas. Esto quiere decir que como "sentencia1" o "sentencia2", de acuerdo con el formato, puede escribirse otra "sentencia if".

Ejemplo:

```
if (expresiónl)
{
if (expresión2)
sentencia1;
} else
sentencia2;
```

Sentencia switch. Esta sentencia permite ejecutar una de varias acciones, en función del valor de una expresión.

```
switch (expr-test)
{
[declaraciones]
case cte.l:
[sentencia1;]
[case cte.2:]
[sentencia2;]
[case cte.3:]
[sentencia3;]
.
.
[default:]
[sentenciaN;]
}
```

expr-test es una expresión entera.

cte.i es una constante entera, una constante de un solo carácter o una expresión constante; en todos los casos, el valor resultante tiene que ser entero.

sentencia es una sentencia simple o compuesta.

Al principio del cuerpo de la sentencia "switch", pueden aparecer declaraciones. Las inicializaciones, si las hay, son ignoradas.

La sentencia "switch" evalúa la expresión entre paréntesis y compara su valor con las constantes de cada "case". La ejecución de las sentencias del cuerpo de la sentencia "switch", comienza en el "case" cuya constante coincida con el valor de la "expr-test" y continúa hasta el final del cuerpo o hasta una sentencia que transfiera el control fuera del cuerpo (por ejemplo "break"). La sentencia "switch" puede incluir cualquier número de cláusulas "case".

Sentencia break. Esta sentencia finaliza la ejecución de una sentencia "do", "for", "switch" o "while" en la cual aparece break.

Cuando estas sentencias están anidadas, la sentencia "break" solamente finaliza la ejecución de la sentencia donde está incluida.

Sentencia while. Ejecuta una sentencia, simple o compuesta, cero o más veces, dependiendo del valor de una expresión.

while (expresión)
sentencia;

expresión es cualquier expresión numérica, relacional o lógica.
sentencia es una sentencia simple o compuesta.

La ejecución de la sentencia "while" sucede así:

1. Se evalúa la expresión.
2. Si el resultado de la expresión es cero (falso), la sentencia no se ejecuta y se pasa a ejecutar la siguiente sentencia en el programa.
3. Si el resultado de la expresión es distinto de cero (verdadero), se ejecuta la sentencia y el proceso se repite comenzando en el punto 1.

Sentencia do. Ejecuta una sentencia, simple o compuesta, una o más veces, dependiendo del valor de una expresión.

do
sentencia;
while (expresión);

expresión es cualquier expresión numérica, relacional o lógica.
sentencia es una sentencia simple o compuesta.

La ejecución de una sentencia "do" sucede de la siguiente forma:

1. Se ejecuta la sentencia o cuerpo de la sentencia "do".
2. Se evalúa la expresión.
3. Si el resultado de la expresión es cero (falso), se pasa a ejecutar la siguiente sentencia del programa.
4. Si el resultado de la expresión es distinto de cero (verdadero), el proceso se repite comenzando en el punto 1.

Sentencia for. Cuando se desea ejecutar una sentencia simple o compuesta, repetidamente un número de veces conocido, la construcción adecuada es la sentencia "for".

for ([v1=e1, [v2=e2]...];[condición];[progresión-cond])sentencia;

vi=ei	vi representa una variable que será inicializada con el valor de la expresión ei.
condición	es una expresión de Boole (operandos unidos por operadores relacionales y/o lógicos). Si se omite, se supone siempre que es verdadera.
progresión-cond	es una expresión cuyo valor evoluciona hasta que se de la condición para finalizar la ejecución de la sentencia "for".
sentencia	es una sentencia simple o compuesta.

La ejecución de la sentencia "for" sucede de la siguiente forma:

1. Se inicializan las variables "vi".
2. Se evalúa la expresión de Boole (condición).
   2.1. Si el resultado es distinto de cero (verdadero), se ejecuta la sentencia, se evalúa la expresión que de lugar a la "progresión de la condición" y se vuelve al punto 2.
   2.2. Si el resultado es cero (falso), la ejecución de la sentencia "for" se da por finalizada y se continúa en siguiente sentencia del programa.

Ejemplo:

```
for (x = 1; x<21; x++) { ... bloque del for ... }
```

Sentencia continue. Esta sentencia, estando dentro de una sentencia "do", "while", o "for", pasa el control para que se ejecute la siguiente iteración.

continue;

Sentencia goto y etiquetas. La sentencia "goto" transfiere el control a una línea específica del programa, identificada por una etiqueta.

goto etiqueta;
etiqueta: sentencia;

4.4. Sentencias de preprocesador

Las directrices para el preprocesador son utilizadas para hacer programas fuente fáciles de cambiar y de compilar en diferentes situaciones. Una directriz comienza con el símbolo # como primer carácter, distinto de blanco, en una línea, e indica al preprocesador una acción específica a ejecutar. Pueden aparecer en cualquier parte del fichero fuente, pero solamente se aplican desde su punto de definición hasta el final del programa fuente.

Las instrucciones para el preprocesador son:

1. **# define**. Sustitución de símbolos.
2. **# undef**. Borra la definición del identificador especificado.
3. **# ifdef**.
4. **# line**.

5. HERRAMIENTAS PARA LA ELABORACIÓN Y DEPURACIÓN

5.1. Turbo C y Turbo C++ de Borland

El paquete de software de Turbo C contiene todos los archivos ejecutables y las librerías que se necesitan para crear, compilar, "linkar" (enlazar) y ejecutar programas escritos en C. También contiene programas de ejemplos, la utilidad independiente MAKE, los volúmenes de documentación, y el programa autodemo TCTOUR que enseña interactivamente el funcionamiento del entorno, incluyendo la gestión de ventanas y ejemplos de compilación.

Turbo C y Turbo C++, se ejecutan bajo MS/DOS y puede generar códigos para los microprocesadores 8088/8086, 80286/80386, poseyendo librerías para emular los coprocesadores 8087/80287 y 80387. Soporta el estándar ANSI, totalmente compatible con las definiciones Kernighan & Ritchie.

Los programas fuente pueden compilarse de dos formas distintas: utilizando el IDE (el entorno integrado), o bien, el *compilador de líneas de comandos independiente* (TCC).

Es preciso aclarar, que el Turbo C++ no es una versión mejorada de Turbo C desde el punto de vista del entorno, sino que es un lenguaje orientado a objetos desarrollado a partir del Turbo C. Todos los programas fuentes escritos en Turbo C podrán ser compilados por Turbo C++, pero no a la inversa.

El entorno de desarrollo integrado (IDE). Con el entorno IDE de Turbo C no es necesario utilizar un editor de textos, compilador, linker (enlazador) y software MAKE para realizar y ejecutar programas en C. Todas estas características se encuentran completamente accesibles y visualizadas con el programa TC, sin necesidad de salir de este entorno.

Esto permite ciclos de recompilación extremadamente rápidos, lo que hace la creación, prueba y depuración de los programas más rápida y sencilla.

Cuando se ejecuta TC, aparece la pantalla principal, que se divide en cuatro partes:

- El menú principal, línea superior (más amplio en Turbo C++).
- La ventana del editor y la línea de estado (en Turbo C).
- La ventana de mensajes del compilador, línea inferior (modificable en Turbo C++).
- La ventana de ayuda de referencia rápida.

Utilización del compilador en línea de comandos. Para utilizar esta posibilidad de compilación, previamente debemos tener grabado nuestro programa fuente en formato ASCII, realizado mediante un editor de textos. El compilar de esta manera puede tener ventajas para los programadores avanzados, ya que dispone de otras posibilidades que comentaremos a continuación. El programa que realiza la compilación mediante líneas de comandos se llama TCC.EXE, y el formato de introducción de comandos sigue la siguiente regla:

TCC [opción1 opción2 ...] nombre1 [nombre2] ...

donde "opción" representa un comando, que debe entrarse precedido del signo - (menos) para su activación. Si queremos desactivar una opción, debe además ir detrás también el signo (menos). El elemento nombre puede ser:

- nombre.ASM invoca a MASM para ensamblar a OBJ.
- nombre.OBJ incluye este archivo objeto durante el enlace.
- nombre.LIB incluye este archivo librería durante el enlace.

- nombre compila este archivo fuente de C.

- nombre.C compila este archivo fuente de C.

- nombre.CCP compila este archivo fuente de C++.

5.2. Quick C y C Compiler de Microsoft

Inicialmente, el paquete de software de C Compiler incluía también el paquete de Quick C, si bien este último podía adquirirse por separado, a un precio menor.

Ambas versiones, contienen todos los ficheros ejecutables y las librerías necesarias para crear, compilar, "linkar" y ejecutar programas escritos en C, así como una serie de programas de utilidades.

Esencialmente, existen dos versiones de Quick C bien diferenciadas. La versión 1.0 que tiene como una de sus opciones, la posibilidad de compilar, linkar y ejecutar el programa en memoria, sin generar ficheros en disco, y versiones posteriores que siempre generan archivos en disco.

La versión 1.0 carga un modelo "limitado" de Quick C en memoria, por lo que algunas compilaciones se presentan mensajes de error en pantalla indicando que alguna/s función/es no están resueltas externamente (inresolved external). Esto, se soluciona fácilmente volviendo a compilar el programa con la opción de crear ficheros en disco. En la mayor parte de los casos, la compilación en esta versión (1.0) utilizando la opción "start" (compilación en memoria) no presenta ningún tipo de problemas, y sí una mayor rapidez y facilidad de uso. Por ello se ha elegido esta versión, que presenta además pocas diferencias de uso respecto a versiones posteriores. La versión elegida para C Compiler, es sin embargo la 6.00, última versión aparecida en el mercado en 1992.

Quick C y C Compiler, se ejecutan bajo MS-DOS/OS2 y puede generar códigos para los microprocesadores 8088/8086, 80286/80386, poseyendo librerías para emular los coprocesadores 8087/80287 y 80387. Soporta al estándar ANSI, totalmente compatible con las definiciones Kernighan & Ritchie.

Los programas fuente pueden compilarse de dos formas distintas: utilizando el *entorno integrado* (programas QC para Quick C y PWB para el C Compiler), o bien, el *compilador de líneas de comandos independiente* (QCL y CL, respectivamente).

El entorno de desarrollo integrado. Con el entorno integrado de Quick C y C Compiler de Microsoft, no es necesario utilizar un editor de textos, compilador, linker (enlazador) y software adicional para realizar y ejecutar programas en C. Todas estas características se encuentran con los programas QC (Quick C) y PWB (C Compiler), sin necesidad de salir de este entorno.

El entorno de Microsoft trata a los menús como ventanas que pueden cambiar de tamaño e, incluso, de posición. Además al soportar el ratón, la gestión de las mismas se realiza de una manera cómoda y rápida posicionándose encima de la opción y pulsando el botón izquierdo del ratón.

El menú principal. El menú principal se utiliza para la ejecución de las órdenes y comandos de Quick C y C Compiler, como por ejemplo utilización del editor, compilación de un programa, o establecimiento de las opciones del entorno.

La activación del menú principal (franja superior de la pantalla) se obtiene por la pulsación simultánea de las teclas ALT y la letra inicial del nombre de la opción, o bien con la pulsación de ENTER una vez situados con las teclas del cursor en la opción elegida. Igualmente, se obtiene la activación de la opción elegida situando el cursor en la misma con el ratón y pulsando el botón izquierdo del mismo. En algunos submenús se pasa de una opción a otra pulsando la tecla TAB y las teclas del movimiento del cursor + ENTER. El menú principal contiene las siguientes alternativas:

FILE. Carga y graba ficheros, permite la creación de archivos fuente, maneja los directorios, invoca al DOS y termina la ejecución de QC o PWB.

EDIT. Ayuda a la modificación, copia efectiva y cambios de posición de las líneas de los ficheros fuente.

VIEW. Adapta la visualización de QC/PWB en pantalla, dentro de opciones determinadas, dirigidas a las preferencias del usuario.

SEARCH. Permite la búsqueda y/o sustitución de cadenas de caracteres en el formato de edición.

MAKE (C Compiler). Es similar al comando PROJECT de Turbo C/Turbo C++.

RUN. Compila, enlaza y ejecuta los programas presentes en memoria automáticamente.

DEBUG (Quick C). Ayuda a la depuración de los programas.

CALLS (Quick C). Muestra la jerarquía de las funciones (anidamiento) que, han sido llamadas por otras funciones y asigna, mediante el movimiento del cursor, el punto donde se efectúa la llamada de alguna de las funciones visualizadas.

OPTIONS (C Compiler). Permite la selección de las opciones del compilador (como los modelos de memoria), del enlazador y del entorno.

BROWSE (C Compiler). Contiene diversas utilidades relacionadas con el entorno de PWB.

HELP. Suministra ayuda e información suplementaria sobre las sentencias de Quick C y C Compiler.

Utilización del compilados en línea de comandos. Tanto Quick C como C Compiler, permiten compilar y “linkar” un programa en "línea de comandos", desde el Sistema Operativo, incrementando la potencia y versatilidad de los resultados obtenidos.

- *Línea de comandos en Quick C*: La llamada al compilador de Quick C se realiza con el formato:

  qcl NombreFichero1/Opción1 NombreFichero2,... /link lib OpciónL

Aunque con "qcl" se puede compilar y enlazar un fichero fuente ASCII en un solo paso, esto puede realizarse por separado, compilando con la opción "/c" de "qcl" y utilizando, posteriormente, el clásico programa de enlace LINK.EXE de Microsoft.

- Línea de comandos en C Compiler: Al igual que "qcl" la utilidad CL de C Compiler compila y enlaza uno o más ficheros fuente de C. La llamada se hace con:

  CL [/Opciones] NombresFicheros NombresLibrerías [OpcionesLink]

7. BIBLIOGRAFÍA

García de Sola, J.F.
Lenguaje C y estructura de datos
Mc Graw-Hill, 1992

Salvador Senent
Gestión de Entrada/Salida en C
Anaya Multimedia, 1992