Preparador Informática

www.preparadorinformatica.com

TEMA 31. INFORMÁTICA

TEMA 34. S.A.I

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

TEMA 31 INF/ TEMA 34 SAI: 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.

1. INTRODUCCION

A lo largo de la historia de la informática, han ido apareciendo diferentes lenguajes de programación como Fortran, Cobol, Basic, Pascal, C, C++, Java, C#, etc. Concretamente el lenguaje de programación C fue uno de los primeros lenguajes en aparecer (a principios de la década de los 70) y su importancia radica en que después de cinco décadas, hoy en día sigue siendo un lenguaje de programación de referencia ocupando el segundo puesto de los lenguajes de programación más utilizados, solo por detrás de Java y a muy poca distancia de este *(datos extraídos de la consultora Tiobe en noviembre de 2019).*

Se trata de un lenguaje que tiene una biblioteca extensa y además es rápido y bien estructurado. Estas características permiten que tenga aplicaciones en una gran parte de dominios en el mundo de la programación. Para hacernos una idea de la importancia del lenguaje C, hoy en día es utilizado en: sistemas operativos, gráficos y juegos, plataformas de computación, escritura de software embebido y desarrollo de nuevos lenguajes. El sistema operativo Linux, por ejemplo, está escrito en C; además lenguajes de programación como Java están basados en C.

En el presente tema se estudian algunas de las principales características del lenguaje C. Este lenguaje es uno de los más importantes en la historia de la informática por su trascendencia en la misma, de ahí la importancia de estudiarlo en el presente tema.

2. LENGUAJE C: CARACTERÍSTICAS GENERALES

El lenguaje de programación C fue creado por Brian Kernighan y Dennis Ritchie a mediados de los años 70. Es un lenguaje de **propósito general** ya que es muy flexible y es utilizado en numerosos ámbitos. Es un **lenguaje para programadores** en el sentido de que proporciona una gran flexibilidad de programación y una muy baja comprobación de incorrecciones. Se considera como un lenguaje de **nivel medio**, puesto que combina elementos de lenguajes de alto nivel (Pascal, Basic...) con el funcionalismo del lenguaje ensamblador. Es estructurado, pero no es estructurado por bloques, o sea, no es posible declarar subrutinas dentro de otras subrutinas. Además, el lenguaje C no es rígido en la comprobación de tipos de datos, permitiendo fácilmente la conversión entre diferentes tipos de datos y la asignación entre tipos de datos diferentes.

El lenguaje C **posee un número reducido de palabras reservadas** que define el standard ANSI-C. Estas palabras reservadas se escriben en minúsculas (debemos tener en cuenta esto ya que C **distingue entre mayúsculas y minúsculas**). Como ejemplo de palabras reservadas están "break, case, char, double, for, if, int...".

Además, otra característica importante de C es que los **programas son muy portables**. La mayoría de los programas se pueden compilar y ejecutar en muchas computadoras diferentes con muy pocos o ninguna modificación. También dispone de **multitud de librerías** que facilitan la labor del programador

3. ELEMENTOS DEL LENGUAJE

3.1. COMENTARIOS

Los comentarios pueden aparecer en cualquier parte del programa, mientras estén situados entre los identificadores /* y */ o entre // y final de línea. Los comentarios son útiles para identificar los elementos principales de un programa o para la explicación del mismo.

3.2. DIRECTIVAS DEL PREPROCESADOR

En un programa escrito en C, es posible incluir diversas instrucciones para el compilador dentro del código fuente del programa. Estas instrucciones dadas al compilador son llamadas directivas del preprocesador y son las siguientes:

- **#include:** Incluye el fichero, cuyo nombre se indica, para su compilación con el resto del código. Los ficheros se suelen denominar .h y contienen declaraciones de tipos, variables y prototipos. El nombre del archivo fuente a incluir se coloca normalmente entre paréntesis de ángulo. (#include <stdio.h>)
- **#define:** Define macros o símbolos, es decir, cada ocurrencia de ese símbolo es sustituida por su definición.
- **#undef:** Permite quitar una definición que se realizó con anterioridad.
- **#error:** Fuerza a parar la compilación del programa, a la vez que muestra un mensaje de error.
- **#if, #ifdef, #ifndef, #else, #elif y #endif:** Son directivas condicionales.
- **#line:** Permite cambiar la cuenta de líneas del compilador.
- **#pragma:** Permite dar instrucciones al compilador sobre cómo debe realizar la compilación del código fuente.

3.3. IDENTIFICADORES

En el lenguaje C, un identificador es cualquier palabra no reservada que comience por una letra o por un subrayado, pudiendo contener en su interior letras, números y subrayados. Se usan para referenciar las variables, las funciones, las etiquetas y otros objetos definidos por el usuario.

3.4. TIPOS DE DATOS BÁSICOS

Los tipos de datos básicos disponibles en C son:

- **Número entero:** Identificado como int, que tiene distintas variantes dependiendo de su longitud (Corto: short ó short int, Normal: int, Largo: long ó long int) y dependiendo del signo (Sin signo: unsigned, Con signo: si no se indica nada respecto al signo, se entiende que tiene signo).
- **Número real:** Todos los números reales ó en coma flotante tienen signo. Dependiendo de la longitud se identificarán como float, double o long double.
- **Caracteres:** Los caracteres son todos del mismo tamaño, así que se pueden agrupar en función del signo (Normal: se identifican como char, Con signo: se identifican como signed char, Sin signo: se identifican como unsigned char).

3.5. TIPOS DE DATOS AVANZADOS

- **Enumeraciones:** Es un tipo de dato entero con valores constantes definidos por el usuario o, dicho de otro modo, un tipo enumerado es una lista de valores enteros constantes. Por ejemplo: enum boolean {FALSE, TRUE}. Salvo que se especifique otra cosa, comienzan en 0, así que FALSE será 0 y TRUE 1.
- **Arrays:** Los arrays son bloques de elementos del mismo tipo. El tipo base de un array puede ser un tipo fundamental o derivado. Los elementos individuales del array van a ser accesibles mediante una secuencia de índices. Los índices, para acceder al array deben ser variables o constantes de tipo entero. Se define la dimensión de un array como el total de índices que necesitamos para acceder a un elemento particular del array. Los arrays pueden tener una o varias dimensiones. El array más común en C es la cadena, que simplemente es un array de caracteres terminado por uno nulo.
- **Punteros:** Un puntero es una variable que contiene una dirección de memoria. Normalmente esa dirección es una posición de memoria de otra

variable, por lo cual se suele decir que el puntero "apunta" a la otra variable.

La sintaxis de la declaración de una variable puntero es: tipo *nombre;

El tipo base de la declaración sirve para conocer el tipo de datos al que pertenece la variable a la cual apunta la variable de tipo puntero.

C proporciona dos operadores relacionados con las direcciones de memoria:

- * Operador indirección: A partir de una variable tipo puntero nos proporciona el dato apuntado.
- & Operador dirección: A partir de una variable nos da la dirección de memoria donde se almacena dicha variable

- **Estructuras:** Una estructura es un agregado de tipos fundamentales o derivados que se compone de varios campos. Cada elemento de la estructura puede ser de un tipo diferente, cosa que no es posible en los arrays:

```
struct nombre_estructura{
      tipo1 campo1;
      …
      tipoN campoN,
};
```

- **Uniones:** Se definen de forma parecida a las estructuras y se emplean para almacenar en un mismo espacio de memoria variables de distintos tipos. El tamaño de la unión es igual al tamaño del mayor de sus campos:

```
union nombre_union{
      tipo1 campo1;
      …
      tipoN campoN,
};
```


- **Tipos definidos por el usuario:** Es una declaración que permite definir tipos de datos que ya tenemos con nombres diferentes y también con nuevos tipos de datos. La sintaxis es la siguiente: `typedef tipo nuevo_tipo;`
- **Campos de bits:** Un campo de bit es un método predefinido por C para poder acceder a un bit de un byte. Se basa en la estructura, pues un campo de bit no es más que un tipo especial de estructura al que le añadimos el número de bits que se desea reservar para cada elemento. El formato es:

```
struct nombre_campo_bit
{
      tipo nombre1 : longitud;
      ...
      tipo nombreN : longitud;
}variables_campo_bit;
```

3.6. CONSTANTES

Una expresión constante es un valor que no puede variar a lo largo del programa. Es costumbre escribir las constantes en mayúsculas. Ej: `#define MINIMO 10`

Existen, además, algunos tipos de constantes, distintos a los anteriores, que es necesario resaltar de forma particular. Estos tipos de constantes son:

- Las **constantes hexadecimales y octales** son constantes enteras, pero definidas en base 16 (hexadecimales) o en base 8 (octales). Las constantes de tipo hexadecimal comienzan por los caracteres 0x seguidas del número deseado. Las constantes de tipo octal comienzan por un cero (0). Por ejemplo, es constante hexadecimal (65535 decimal); y octal 0173 (123 decimal).
- Las **constantes de cadena o literales** son conjuntos de caracteres que se encierran entre comillas dobles. Por ejemplo: "El programa ha comenzado".
- Las **constantes de caracteres de barra invertida** se usan para introducir caracteres que es imposible introducir por el teclado. Estas constantes son:

Código	Significado
\b	Retroceso
\f	Alimentación de hoja
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador horizontal
\"	Doble comilla
\'	Simple comilla

Código	Significado
\0	Nulo
\\	Barra invertida
\v	Tabulador vertical
\a	Alerta
\o	Constante octal
\x	Constante hexadecimal

3.7. VARIABLES

Una variable es un espacio de memoria identificado por un nombre, donde se van a almacenar valores de un tipo determinado y que sí pueden variar a lo largo del programa. Para declarar una variable de un tipo determinado, se debe indicar en primer lugar el tipo de dato del que se desea crear la variable, seguido del nombre que se quiere dar a la variable, y con el que luego nos referimos a ella. Por ejemplo: `int NumHijos;`

Mediante el operador de asignación "=", podemos asignar un valor a una variable. Si empleamos este operador en la declaración de la variable, su valor inicial será el indicado por medio del operador: `int NumHijos=1;`

En C, las variables pueden tener distintos tipos dependiendo del alcance de las mismas. Si se declaran fuera de todas las funciones del programa, son las

llamadas variables **globales**, accesibles desde cualquier parte del programa. Si se declaran dentro de una función o como parámetros a la función, son las llamadas variables **locales**, accesibles tan solo por la función en las que se declaran. Además, las variables pueden tener distintos modos de almacenamiento: auto, extern, static y register.

3.8. OPERADORES

C es un lenguaje muy rico en operadores. Se definen varios tipos:

- **Operador de asignación:** Se representa con el carácter de igualdad (=).

```
variable = expresión;
```

Donde variable es el nombre de la variable donde almacenamos el valor como resultado de la expresión. Por ejemplo: `valor = 25;`

- **Operadores aritméticos:** Los operadores aritméticos son:

Operador		Operador		Operador	
++	Incremento	--	Decremento		
-	Menos unario				
*	Multiplicación.	/	División	%	Módulo
+	Suma	-	Resta		

- **Operadores lógicos:** Los operadores lógicos en C son los siguientes:

OPERADOR	SIGNIFICADO
!	Not (no lógico)
&&	And (y lógico)
||	Or (ó lógico)

- **Operadores relacionales:** Estos operadores son los siguientes:

OPERADOR	SIGNIFICADO
==	Igual a
!=	No igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

- **Operadores sobre bits:** C posee operadores que actúan a nivel de bits sobre los datos, estos operadores son los siguientes:

Operador	Nombre	Operación
~	Not	Complemento a uno (NOT)
<<	Desplazamiento izquierda	Desplazamiento izquierda
>>	Desplazamiento derecha	Desplazamiento derecha
&	And	Y
^	Xor	O exclusivo (XOR)
|	Or	O

- **El operador ?:** Se usa para reemplazar las sentencias if/else. El operador ? es un operador ternario cuyo formato general es: `Exp1 ? Exp2 : Exp3;` Donde Exp1, Exp2 y Exp3 son expresiones. El operador ? evalúa la expresión Exp1, si es cierta se evalúa Exp2 y si es falsa se evalúa Exp3.
- **El operador sizeof:** Es un operador en tiempo de compilación. El operador sizeof devuelve el tamaño de una variable o tipo de dato durante la compilación, no durante la ejecución del programa.

3.9. CONVERSIÓN DE TIPOS

Conversión automática: El lenguaje C permite que en una misma expresión aparezcan diferentes tipos de datos, encargándose el compilador de realizar las operaciones de forma correcta. El compilador del lenguaje C, cuando en una misma expresión aparecen dos o más tipos de datos, convierte todos los operandos al tipo del operando más grande existente.

Conversión forzada: En C, existe también la posibilidad de forzar la conversión de un tipo de datos en otro tipo de datos. Esta conversión se llama "casts", y su sintaxis es: `(tipo)expresión`

4. ESTRUCTURA DE UN PROGRAMA

Un programa en C consta de uno o más módulos (ficheros fuentes). Cada módulo puede contener:

- Directivas del precompilador, por ejemplo, para "incluir" otros ficheros (#include) y "definir" constantes y macros (#define).
- Declaraciones de variables y prototipos de funciones.
- Una o más funciones.
- Comentarios

Básicamente podemos decir que todo programa de C consta de un conjunto de funciones, y una función llamada main que es la única absolutamente imprescindible y es la primera que se ejecuta al comenzar el programa, llamándose desde ella al resto de funciones que compongan nuestro programa. Las definiciones de las funciones adicionales pueden preceder o seguir a main. Cada función debe contener al menos:

- Una cabecera de la función, que consta del nombre de la función seguido de una lista opcional de argumentos encerrados con paréntesis.
- Una sentencia compuesta, que contiene el resto de la función.

4.1. SENTENCIAS

Una expresión seguida de un ; es una **sentencia**. Toda sentencia debe ir terminada con punto y coma. Pueden ir varias sentencias en una misma línea. Las llaves { } se emplean para agrupar sentencias en lo que se denomina **bloque**. Un bloque es sintácticamente equivalente a una sentencia. Dentro de un bloque puede haber sentencias y declaraciones.

4.2. SENTENCIAS CONDICIONALES

- **Sentencias if-else:** La forma general de la sentencia es:

```
if (condición)
      sentencia;
else
      sentencia;
```

Si la condición es verdadera se ejecuta la sentencia asociada al if, en caso de que sea falsa la condición se ejecuta la sentencia asociada al else (si existe el else). Las sentencias de control if pueden ir anidadas.

- **Sentencias switch:** La forma general de la sentencia switch es:

```
switch(variable)
{
      case const1:
            sentencia;
            break;
      ...
      default:
            sentencia;
}
```

Donde variable debe ser de tipo char o int, y donde const1, const2, ..., indican constantes de C del tipo de datos de la variable. Dichas constantes no pueden repetirse dentro del switch. La sentencia switch se ejecuta comparando el valor de la variable con el valor de cada una de las constantes, realizando la comparación desde arriba hacia abajo. En caso de que se encuentre una constante cuyo valor coincida con el valor de la variable, se empieza a ejecutar las sentencias hasta encontrar una sentencia break. En caso de que no se encuentre ningún valor que coincida, se ejecuta el default (si existe).

4.3. SENTENCIAS ITERATIVAS

- **Sentencias while:** El bucle while se ejecuta mientras la condición sea verdad. La sintaxis del bucle while es:

```
while (condición)
      sentencia;
```

- **Sentencias do-while**: El bucle do/while comprueba la condición en la parte baja del mismo, lo cual provoca que el bucle se ejecute como mínimo una vez. El bucle do-while se ejecuta mientras la condición sea verdad. La sintaxis del bucle do/while es:

```
do
      sentencia;
while (condición);
```

- **Sentencias for:** El bucle for de C puede no contener inicialización, condición o incremento, o incluso pueden no existir dos e incluso las tres expresiones del bucle. El bucle for se ejecuta siempre que la condición sea verdadera, es por ello que puede llegar a no ejecutarse. La sintaxis del bucle for es:

```
for(inicialización,condición,incremento)
      sentencia;
```

4.4. SENTENCIAS DE SALTO

- **Sentencias return:** Se usa para volver de una función. Se trata de una sentencia de salto porque hace que la ejecución vuelva al punto en que se hizo la llamada a la función. La forma general es: `return expresión;`
  Donde expresión es opcional. Se pueden usar tantas sentencias return como se quiera en una función. La función termina al encontrar el primero.
- **Sentencias break y continue:** Permiten modificar y controlar la ejecución de los bucles anteriormente descritos. La sentencia break provoca la salida del bucle en el cual se encuentra y la ejecución de la sentencia que se encuentra a continuación del bucle. La sentencia continue provoca que el programa vaya a comprobar la condición del bucle en while y do/while, o bien, que ejecute el incremento y después compruebe la condición en el caso del bucle for.
- **Sentencias goto:** Se utiliza para alterar la secuencia normal de un programa, transfiriendo el control a otra parte del programa, normalmente donde hay una etiqueta.

5. FUNCIONES DE LIBRERÍA Y USUARIO

5.1. FUNCIONES DE USUARIO

Una función es un conjunto de instrucciones, que se ejecutan cuando se le llama, es decir, cuando se referencia a esta función desde otro punto del programa. Para que la función pueda procesar distintos datos o de distinta forma dependiendo del punto del programa desde el que se le llame, se puede pasar a la función la información que debe tratar en forma de parámetros y ésta puede devolver el resultado de este tratamiento como valor de retorno. La organización de un programa grande en funciones sencillas hará que el programa sea estructurado además de fácil de depurar y mantener. Podemos considerar que una función tiene tres estados: definición, declaración y llamada.

Definición de la función: Definir una función consiste en escribir el código que se ha de ejecutar cuando se llame a esa función. La sintaxis es la siguiente:

```
tipo nombre_funcion (declaración_parametros){
      variables_locales;
      proposiciones;
      return (expresión);
}
```

El tipo especifica el tipo de valor que devuelve la sentencia return de la función. El valor puede ser cualquier tipo válido, si no se especifica ninguno, se asume un resultado entero. La lista de parámetros ("declaración_parametros") es la lista de nombres de variables separados por comas con sus tipos asociados que reciben los valores de los argumentos cuando se llama a la función. Una función puede no tener parámetros, en cuyo caso la lista de parámetros está vacía pero los paréntesis es necesario ponerlos igualmente.

Declaración de la función: Normalmente necesitamos llamar a una función sin haberla definido todavía, o que se ha definido en un archivo diferente. Cuando el compilador localiza la llamada a una función comprueba que la función existe, que los parámetros que se le pasan son los adecuados y que el uso del valor del retorno es el adecuado para el tipo de dato devuelto. Si la función no está definida previamente, el compilador no puede realizar estas comprobaciones y genera un error.

Para declarar una función se debe escribir el tipo del valor de retorno, seguido del nombre de la función y de los tipos de los distintos parámetros encerrados

entre paréntesis y separados por comas. La declaración finaliza con un punto y coma. Ejemplo: `int máximo (int a, int b);`

Llamadas a la función: Las funciones son llamadas para su ejecución desde cualquier parte del código. La llamada de una función se produce mediante el uso de su nombre en una sentencia, pasando una lista de argumentos que deben coincidir en número y tipo con los especificados en la declaración. Las llamadas a funciones se pueden realizar, en general, de dos formas, por valor y por referencia:

- La llamada por valor copia el valor de un argumento en el parámetro formal de la función. Así, los cambios en los parámetros de la función no afectan a las variables que usan en la llamada. Es la forma de llamada más habitual.
- La llamada por referencia copia la dirección del argumento en el parámetro. Dentro de la función se usa la dirección para acceder al argumento usado, significando que los cambios hechos a los parámetros afectan a la variable usada en la llamada.

Recursividad: Una función de C puede llamarse a sí misma. Este proceso recibe el nombre de recursividad. Los ejemplos de recursividad abundan, siendo uno de los más habituales la función factorial:

```
unsigned Factorial(unsigned num){
      if (num==0)
            return 1;
      return num*Factorial(num-1);
}
```

La recursividad es una poderosa herramienta de programación, sin embargo, presenta dos problemas importantes:

- La velocidad de ejecución de un algoritmo programado de forma recursiva es mucho más lenta que el programado de forma iterativa.
- La recursividad, si es excesiva, puede ocasionar el desbordamiento de la pila, y con ello, el fallo en la ejecución del programa.

5.2. FUNCIONES DE LIBRERÍA

El estándar ANSI C define un conjunto de funciones, así como tipos relacionados y macros, que son proporcionados para la implementación. Todas las librerías

son declaradas en un fichero cabecera. Para que sea visible al programa, se añade el comando del preprocesador #include. Por ejemplo: `#include <stdio.h>`

Cada fichero de cabecera se denomina librería. Las librerías estándar más utilizadas mostradas en orden alfabético son:

- <ctype.h>: Contiene funciones para clasificar caracteres según sus tipos o para convertir entre mayúsculas y minúsculas independientemente del conjunto de caracteres (isalpha, isdigit...).
- <math.h>: Contiene las funciones matemáticas comunes (cos, sin, tan, log...).
- <stdio.h>: Proporciona las funciones de entrada/salida (scanf, printf...).
- <stdlib.h>: Para realizar ciertas operaciones como conversión de tipos (atoi, atof...), generación de números pseudo-aleatorios (rand...), gestión de memoria dinámica (malloc, free...), control de procesos (exit, system...), funciones de entorno, de ordenación y búsqueda (bsearch...).
- <string.h>: Para manipulación de cadenas de caracteres (strcpy, strlen...)

6. ENTORNO DE COMPILACIÓN. HERRAMIENTAS PARA LA ELABORACIÓN Y DEPURACIÓN DE PROGRAMAS EN C

La compilación se produce en tres pasos:

- Se crea el código fuente como un fichero de texto estándar con extensión ".c".
- Compilación del código fuente y creación del código objeto.
- Enlazado del código objeto con el código objeto de las librerías empleadas.

Al ejecutar el programa, se crean distintas regiones de memoria:

- Código del programa: Almacena instrucciones en código máquina de las funciones y los procedimientos, y su tamaño puede fijarse en tiempo de compilación.
- Memoria estática: Almacena las variables globales y constantes (enteros, reales, strings, ...), variables estáticas (static), direcciones reservadas por el sistema operativo y código y datos estáticos cargados desde librerías o módulos precompilados.
- Memoria dinámica: Zona de tamaño fijo donde se mantienen las variables locales e información de control adicional de los procedimientos que se han

llamado y, además, se guardan las variables cuyo tamaño varía en tiempo de ejecución o que no se pueden mantener en memoria estática.

6.1. ENTORNOS DE PROGRAMACIÓN

Un entorno de programación es un programa o conjunto de programas que engloban todas las tareas necesarias (edición del programa, compilación y enlazado, ejecución y depuración) para el desarrollo de un programa o aplicación. Hay quien además incluye la creación de documentación complementaria que facilita el mantenimiento del programa dentro de estas funciones. Este tipo de entornos incorporan numerosas herramientas, utilidades, aplicaciones ya desarrolladas, ejemplos, tutoriales, etc. Todas ellas encaminadas a facilitar y mejorar el desarrollo. A los entornos de programación se les llama IDE (Integrated Development Environment). Como ejemplos de entornos de programación puedo nombrar Eclipse, NetBeans, CodeLite, etc.

6.2. COMPILADOR GCC (GNU COMPILER).

El compilador gcc es rápido, flexible y riguroso con el estándar de ANSI C. En realidad, gcc no genera código máquina, sino código ensamblado. La fase de ensamblado a código máquina la realiza el ensamblador de GNU y el enlazador de GNU se encarga de los objetos resultantes. Este proceso es transparente para el usuario ya que gcc realiza automáticamente el paso desde código en C a un fichero ejecutable. Su sintaxis es "*gcc [opciones] fichero(s)*" donde fichero(s) puede ser uno o varios ficheros fuente o ficheros objeto, y opciones puede ser:

- -o <ejecutable>. El fichero ejecutable generado por gcc es por defecto a.out. Mediante este modificador, se especifica el nombre del ejecutable.
- -Wall. No omite la detección de ningún aviso de compilación (warning).
- -g. Incluye en el ejecutable información necesaria para utilizar un depurador.
- -c. Preprocesa, compila y ensambla, pero no enlaza. El resultado es un fichero objeto con extensión .o y el mismo nombre que el fichero fuente.

6.3. DEPURADOR GDB (GNU DEBUGGER).

Se trata de un depurador asociado a gcc, que necesita que el programa sea compilado previamente con la opción -g. El depurador permite establecer puntos de ruptura condicionales y detener la ejecución del programa cuando el valor de

una expresión cambie. Su sintaxis es gdb fichero donde fichero es el nombre del ejecutable creado con gcc.

6.4. HERRAMIENTA MAKE.

Un programa en C normalmente está formado por varios ficheros fuente y ficheros cabecera. Cada vez que se modifica algún fichero fuente o cabecera, el programa debe recompilarse para crear un ejecutable actualizado. Sin embargo, no es necesario recompilar todos los ficheros, sino sólo los afectados por la modificación. La utilidad make determina automáticamente qué ficheros del programa deben ser recompilados y las órdenes que se deben utilizar para ello. Para utilizar make es necesario escribir un fichero denominado Makefile, que describe las dependencias entre ficheros y las órdenes que se deben ejecutar con cada fichero. Si en un directorio existe un fichero Makefile, la orden make se encargará de ejecutar las órdenes que contiene.

7. CONCLUSIÓN

En el presente tema hemos estudiado el lenguaje C. Este lenguaje es de suma importancia principalmente por varios motivos:

- El lenguaje de programación C sigue siendo uno de los más populares, solo por detrás de Java y a muy corta distancia de éste.
- Sigue siendo uno de los lenguajes más demandados por las empresas.
- Conocer las bases de este lenguaje facilita el aprendizaje y comprensión de otros lenguajes de programación similares, también muy implantados a nivel empresarial.

Principalmente por estos motivos y por muchos otros es tan importante conocer el lenguaje de programación C y dominarlo perfectamente.

8. BIBLIOGRAFÍA

- Kernighan, B. y Ritchie, D: **El lenguaje de programación C**. Editorial Pearson
- Ceballos, F.J.: **C/C++ Curso de Programación**. Editorial Ra-Ma
- Adiego, J. y Llanos, D. **Fundamentos de informática y programación en C**. Editorial Paraninfo.