ESCUELA DE PREPARACIÓN DE OPOSITORES
E. P. O.
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

INF31 – SAI34

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.

Esquema.

1 Introducción. Características generales.

C es un lenguaje de programación de empleo general. Aunque ha sido estrechamente asociado con el sistema operativo UNIX, ya que su desarrollo se realizó en este sistema y debido a que tanto UNIX como su software fueron escritos en C, no está ligado a ningún sistema concreto y, aunque se lo ha llamado "lenguaje de programación de sistemas" a causa de su utilidad en la escritura de sistemas operativos, ha sido utilizado con el mismo éxito para escribir programas numéricos, programas de procesamiento de textos y bases de datos.

C es un lenguaje de relativo "bajo nivel". Esto significa que C trabaja con la misma clase de objetos que la mayoría de las computadoras: caracteres, números y direcciones, que pueden ser combinados con los operadores aritméticos y lógicos, utilizados normalmente en las máquinas.

C no contiene operaciones para trabajar directamente con objetos compuestos, tales como cadenas de caracteres, conjuntos, listas, o arrays, considerados como un todo. El lenguaje no tiene definida ninguna posibilidad de realizar asignación de memoria aparte de las definiciones estáticas y el manejo de pilas para las variables locales de las funciones. Finalmente, C no cuenta con operaciones de entrada/salida: no existen proposiciones READ o WRITE, ni métodos propios para el acceso a archivos. Todos estos mecanismos de alto nivel deben ser aportados por funciones llamadas explícitamente.

De la misma manera, C sólo ofrece proposiciones (sentencias) de control de flujo sencillas, secuenciales, de selección, de iteración, bloques y subprogramas.

C posee un alto grado de portabilidad (movilidad). Debido a que los tipos de datos y estructuras de control que ofrece son manejados directamente por la mayoría de las computadoras, la biblioteca de tiempo de ejecución necesaria para realizar programas independientes es minúscula. Por supuesto, cada versión provee una extensa y compatible biblioteca de funciones para llevar a cabo las operaciones de E/S, manejo de cadenas de caracteres y asignación de memoria; pero como son llamadas únicamente en forma explícita, estas funciones pueden omitirse en caso necesario. Pueden incluso estar escritas en el mismo C.

Debido también a que el lenguaje refleja las posibilidades de las computadoras actuales, los programas en C tienden a ser lo suficientemente eficientes como para que no haya necesidad de escribirlos en lenguaje ensamblador.

Aunque las posibilidades de C corresponden a las de muchas computadoras, el lenguaje es independiente de cualquier arquitectura de máquina en particular y así, con algo de cuidado, es fácil escribir programas "portátiles", es decir, programas que pueden ejecutarse sin cambios en una amplia variedad de computadoras.

Muchas de las ideas principales de C provienen del lenguaje BCPL. La influencia de BCPL le llega indirectamente a través del lenguaje B, escrito por Ken Thompson en 1970 para el primer sistema UNIX en una PDP-7. Aunque tiene varias características comunes con BCPL, C no es propiamente un dialecto suyo. Tanto BCPL como B son lenguajes "sin tipos". En C los objetos fundamentales son caracteres, enteros de varios tamaños y números en punto flotante. Además, existe una jerarquía de tipos de datos derivados, creados con punteros, arrays, estructuras, uniones y funciones.

C posee las construcciones fundamentales de control de flujo sin las cuales no es posible escribir programas bien estructurados: agrupamiento de sentencias, toma de decisiones (`if`), ciclos (bucles), con comprobación de la condición de terminación al principio (`while`, `for`) o al final (`do`) y selección entre un conjunto de casos posibles (`switch`).

C incluye punteros y capacidad de aritmética de direcciones. Los argumentos de las funciones se pasan copiando su valor, siendo imposible que la función llamada altere el valor del argumento. Cuando se desea realizar una llamada por referencia, se puede pasar un puntero explícitamente, pudiendo la función cambiar el valor del objeto al que se apunta.

Cualquier función puede ser llamada recursivamente y sus variables locales son normalmente "automáticas" o creadas de nuevo en cada invocación. Las definiciones de función no pueden anidarse, pero las variables pueden ser declaradas con una estructura de bloques. Las funciones de un programa C pueden compilarse por separado. Las variables pueden ser internas a una función, externas (pero conocidas únicamente a lo largo de un solo archivo fuente) y completamente globales. Las variables internas son automáticas o estáticas. Las variables automáticas pueden colocarse en un registro para incrementar su eficiencia; pero su declaración como registro es únicamente una indicación al compilador y no se refiere a registros específicos de la máquina.

Por último, cabe destacar que el lenguaje C ha sido estandarizado por ANSI (*American National Standards Institute*), siendo conocida la última revisión del mismo como C99 (*ISO/IEC 9899:1999 standard*).

2 Elementos del lenguaje.

2.1 Comentarios.

Los comentarios se utilizan para documentar el código fuente del programa. En C, un comentario es cualquier cadena delimitada por los caracteres `/*` y `*/`, y puede ocupar más de una línea.

De igual modo, y desde la aparición de C99, los caracteres // introducen un comentario que incluye a todos los caracteres excepto el carácter de nueva línea, esto es, hasta el final de la línea.

```
/*
Esto es un comentario
*/
// Esto es otro comentario
```

2.2 Variables.

Las variables son los objetos básicos que se manipulan en un programa. Las declaraciones indican las variables que se van a utilizar y establecen su tipo y, quizá, su valor inicial.

2.2.1 Nombres de variables.

Los nombres se construyen con letras y dígitos. El primer carácter debe ser una letra. El carácter de subrayado "_" se considera igual que una letra. Las letras mayúsculas y minúsculas son diferentes; la práctica tradicional en C utiliza minúsculas para nombres de variables y mayúsculas para nombres de constantes simbólicas. Las palabras clave como `if`, `else`, `int`, `float`, etc., están reservadas: no se pueden usar como nombres de variables.

2.2.2 Tipos de datos.

La siguiente tabla muestra los tipos de datos existentes en C:

`char`	Un byte (octeto) capaz de contener un carácter del juego de caracteres de la instalación local.
`int`	Un entero, normalmente del tamaño de los enteros de la instalación local.
`float`	Un número en punto flotante de precisión normal.
`double`	Un número en punto flotante de doble precisión.

Además, hay una serie de calificadores que se aplican a los enteros: `short`, `long`, `signed` y `unsigned`:

- `short` y `long` se aplican a enteros, en cuyo caso puede omitirse la palabra `int` de la declaración, y la intención es que proporcionen diferentes tamaños de enteros; normalmente `int` refleja el tamaño "natural" de una máquina en particular. Cada compilador puede interpretar `short` y `long` de la forma más apropiada para su computadora. Sobre todo, lo único seguro es que `short` no es mayor que `long`.
- `signed` y `unsigned` se aplican a `char` o a cualquier entero. Los números `unsigned` son siempre positivos o 0.
- `long` puede aplicarse también sobre el tipo `double` y especifica punto flotante de precisión extendida.
- `long long`, desde la aparición de C99 puede aplicarse a `int` para declarar enteros mayores que `long int`.

Además, C permite la utilización de punteros. Un puntero es una variable que contiene la dirección de otra variable. Los punteros se utilizan con abundancia en C, debido en parte a que a veces son la única manera de expresar un cálculo y en parte porque con ellos se obtiene un código más compacto y eficiente.

2.2.3 Declaraciones.

Todas las variables deben ser declaradas antes de utilizarlas, aunque ciertas declaraciones se realizan implícitamente por el contexto. Una declaración especifica un tipo, y le sigue una lista de una o más variables de ese tipo, como en:

```
int lower, upper, step;
char c, line[1000];
```

Las variables se pueden inicializar en su declaración, aunque existen algunas restricciones. Si el nombre va seguido por un signo igual y una constante, esto constituye un inicializador.

La declaración de un puntero es :

```
int *px;
```

2.2.4 Conversiones de tipos.

Los operandos de tipos diferentes que aparecen en una expresión se convierten a un mismo tipo de acuerdo con unas cuantas reglas. En general, las únicas conversiones que se realizan automáticamente son las que tienen sentido, como la conversión de un entero en punto flotante a una expresión como `f+i`. No están permitidas las expresiones que carecen de sentido; por ejemplo, utilizar como subíndice un valor del tipo `float`.

En primer lugar, los caracteres y enteros (`char` e `int`) se mezclan libremente en las expresiones aritméticas: todo `char` de una expresión se convierte automáticamente en `int`. Esto produce bastante flexibilidad en ciertos tipos de transformaciones de caracteres. En la definición de C se garantiza que ningún carácter del juego de caracteres de la instalación será negativo.

Otra utilidad de la conversión automática de tipos está relacionada con expresiones como `i>j` y expresiones lógicas que utilizan `&&` y `||`, en las que se ha establecido que devuelvan el valor 1, en caso de que el resultado tenga el valor cierto, o que devuelvan 0 si el valor es falso. La asignación `isdigit=c>='0' && c<='9';` pone en `isdigit` el valor 1, si c es un dígito, o un 0 si no lo es. (En la condición de un `if`, `while`, `for`, etc. "cierto" es equivalente a "distinto de cero".)

Por regla general, si un operador como + o * con dos operandos tiene los operandos de tipos distintos, el de tipo "inferior" es ascendido al tipo "superior", antes que se realice la operación. El resultado es del tipo "superior". A todo operador aritmético se le aplica la siguiente secuencia de reglas de conversión:

- `char` y `short` se convierten en `int`, y `float` se convierte en `double`.
- Si algún operando es de tipo `double`, el otro se convierte en `double`, y el resultado es `double`.
- Si no se aplica la regla anterior, y si algún operando es del tipo `long`, el otro se convierte en `long`, y el resultado es `long`.
- Si no se aplica la regla anterior y algún operando es de tipo `unsigned`, el otro se convierte en `unsigned`, y el resultado es `unsigned`.
- Si no se puede aplicar ninguna de las reglas anteriores, los operandos deben ser del tipo `int` y el resultado será `int`.

Todos los valores `float` de una expresión se convierten en `double`. Toda la aritmética de punto flotante en C se realiza en doble precisión.

Las conversiones también tienen lugar en las asignaciones; el valor de la parte derecha se convierte al tipo de la izquierda, que es el tipo del resultado. Los caracteres se convierten en enteros. En la operación inversa, de `int` a `char`, los bits de orden superior en exceso son sencillamente eliminados.

Por último, se puede forzar la conversión explícita del tipo "coaccionada" de una expresión mediante una construcción denominada *cast*. En la construcción

`(nombre_de_tipo) expresión`, la expresión se convierte al tipo citado mediante las anteriores reglas de conversión.

2.3 Constantes.

Una constante es un valor que no varía a lo largo del proceso de ejecución del programa.

Las constantes enteras son números con valor entero. Las constantes `long` se escriben según la forma 123L. Cualquier constante entera cuyo tamaño sea demasiado grande para caber en una variable `int` se toma como `long`.

Hay una notación para constantes octales y hexadecimales: un cero que encabece una constante `int` indica octal; el grupo 0x o 0X indica hexadecimal. Las constantes hexadecimales y octales también pueden acabar en L para hacerlas `long`.

Una constante de carácter es un único carácter escrito entre apóstrofos. El valor de una constante de carácter es el valor numérico del carácter en el conjunto de caracteres de la instalación. Ciertos caracteres no imprimibles se representan como constantes de carácter mediante secuencias de escape como \n (nueva línea), \t (tabulador), \0 (nulo), \\ (barra invertida), \' (apóstrofo), etc., que aparecen como dos caracteres, pero que en realidad son sólo uno. Además, cualquier carácter arbitrario se puede generar escribiendo '\ddd' donde ddd puede tener de 1 a 3 dígitos octales. La constante '\0' representa el carácter con valor cero. '\0' se suele escribir en lugar de 0 para poner de relieve la naturaleza de alguna expresión.

Una expresión constante es una expresión formada únicamente por constantes. Estas expresiones se evalúan en tiempo de compilación, en lugar de hacerlo en tiempo de ejecución y, por consiguiente, se emplean en cualquier posición en que vaya una constante.

Una cadena de caracteres constante es una secuencia de cero o más caracteres, delimitados por comillas. Las comillas no forman parte de la cadena, sólo la delimitan. Las mismas secuencias de escape empleadas en las constantes de caracteres se aplican en las cadenas; \" representa el carácter comilla.

Técnicamente, una cadena es un array cuyos elementos son caracteres. El compilador coloca automáticamente el carácter nulo \0 al final de cada cadena para que los programas puedan encontrar el final. Con esta representación no tiene límite real la longitud de una cadena, pero los programas deben explorarla completamente para determinar su longitud. La memoria física requerida es una posición más que el número de caracteres entre las comillas.

Mediante la construcción #define, se puede definir un nombre simbólico o constante simbólica como determinada cadena de caracteres.

2.4 Operadores.

2.4.1 Operadores aritméticos.

Los operadores aritméticos binarios son +, -, *, / y el operador módulo `%`. Hay un - unitario, pero no hay + unitario. La división entera trunca cualquier parte fraccionaria. La expresión `x%y` produce el resto de dividir x entre y. El operador `%` no puede utilizarse con operandos del tipo `float` o `double`.

2.4.2 Operadores relacionales y lógicos.

Los operadores relacionales son: >, >=, <, y <=. Todos tienen la misma precedencia. Les siguen en precedencia los operadores de igualdad: ==, y != que tienen la misma precedencia. Los operadores relacionales tienen menor precedencia que los aritméticos.

Más interesantes son las conectivas lógicas `&&` y `||`. Las expresiones en las que aparecen `&&` o `||` se evalúan de izquierda a derecha, y la evaluación se interrumpe en cuanto se conoce el resultado de falsedad o certeza. Estas propiedades son indispensables para escribir programas que funcionen correctamente.

El operador unitario de negación `!` convierte en 0 un operando distinto de cero, o de valor cierto, y en 1 un operando cero o de valor falso.

2.4.3 Operadores de incremento y decremento.

El operador de incremento ++ le suma 1 a su operando; el operador -- le resta 1. La característica especial de estos operadores es que ++ y -- se emplean como prefijos (antes de la variable) o bien como sufijos (después de la variable). En los dos casos, produce el efecto de incrementar n. Pero la expresión ++n incrementa n antes de utilizar su valor, mientras que n++ la incrementa después de que se ha empleado su valor.

2.4.4 Operadores lógicos para manejo de bits.

C cuenta con un conjunto de operadores para la manipulación de bits. Estos operadores no se aplican a operandos de tipo `float` o `double`. Los operadores para la manipulación de bits son: `&` (Y lógico a nivel de bits), | (O lógico a nivel de bits), ^ (O lógico exclusivo de bits), << (rotación o desplazamiento a la izquierda), >> (rotación a la derecha), y ~ (complemento a 1).

2.4.5 Operadores y expresiones de asignación.

Las expresiones como `i=i+2` en las que el miembro izquierdo se repite en el derecho pueden escribirse en la forma comprimida `i+=2` utilizando un operador de asignación como +=. La mayor parte de los operadores binarios (operadores como + que tienen operando izquierdo y derecho) poseen un correspondiente operador de asignación en `op=`, donde `op` es uno de +, -, *, /, %, <<, >>, &, ^, |.

2.4.6 Expresiones condicionales.

En la expresión `e1?e2:e3` primero se evalúa e1. Si no es cero (cierto), se evalúa la expresión e2, y éste es el valor de la expresión condicional. De lo contrario se evalúa e3, y éste será su valor. Únicamente se evalúa una de las expresiones e2 y e3. Entonces, para asignar a z el máximo de a y b `z=(a>b)?a:b;`.

2.4.7 Operadores de punteros.

Puesto que un puntero contiene la dirección de un objeto, se puede acceder al objeto "indirectamente" a través de él. Supongamos que `x` es una variable, de tipo `int`, y que `px` es un puntero. El operador unitario `&` devuelve la dirección de un objeto, por lo que la proposición `px=&x;` asigna la dirección de `x` a la variable `px`.

El operador unitario * toma su operando como una dirección y accede a esa dirección para obtener su contenido. Si `y` también es de tipo `int`, `y=*px;` asigna a `y` el

contenido de cualquier parte a donde apunte `px`. La secuencia `px=&x; y=*px;` asigna a `y` el mismo valor que `y=x;`

En expresiones como `y=*px+1;` los operadores unitarios `*` y `&` tienen mayor precedencia que los operadores aritméticos, por lo que al evaluar la expresión se toma el valor del objeto al que apunta `px`, se le suma 1, y el resultado se asigna a `y`.

También pueden aparecer referencias a punteros en la parte izquierda de una asignación. Si `px` apunta a `x`, entonces `*px=0` pone `x` a cero, y `*px+=1` incrementa el valor de `x` al igual que `(*px)++`. En este último ejemplo se necesitan los paréntesis; sin ellos la expresión incrementaría `px` y no al objeto al que apunta, ya que los operadores unitarios como `*` y `++` se evalúan de derecha a izquierda.

Y por último, puesto que los punteros son variables, se pueden manipular igual que cualquier otra variable. Si `px` es otro puntero a enteros, `py=px` copia el contenido de `px` en `py`, con lo que se consigue que `py` apunte al mismo objeto que `px`.

2.5 Funciones.

Las funciones dividen grandes trabajos de computación en partes más breves, y aprovechan la labor realizada por otras personas en lugar de partir de cero.

2.5.1 Conceptos básicos.

Una función tiene la forma.

```
[inline] tipo_del_valor_devuelto nombre_función(declaración de
argumentos)
{
   declaraciones y proposiciones si existen
}
```

La claúsula `inline`, introducida en C99, indica al compilador que la llamada a la función debe ser tan rápida como sea posible, siendo específico de cada compilador el cómo conseguirlo.

La sentencia `return` es el mecanismo para devolver un valor desde la función llamada a su llamador; `return` puede ir seguido de una expresión. No es obligatorio que `return` vaya seguido de una expresión. En este caso no se devuelve ningún valor. También se devuelve el control al llamador sin ningún valor cuando la ejecución llega al final de la función al encontrar la llave de cierre.

2.5.2 Argumentos de funciones.

Los argumentos de las funciones se pasan por valor, es decir, la función llamada recibe una copia temporal, privada, de cada argumento, no una dirección. Esto significa que la función no puede alterar el valor del argumento original en la función que la llama. En una función, cada argumento es efectivamente una variable local inicializada con el valor con que se llamó la función.

Cuando aparece el nombre de un array como argumento de una función se pasa la dirección de comienzo del mismo; no se copian los elementos. La función puede alterar los elementos del array indexando a partir de esa posición. Ello hace que el array se pase por referencia.

2.5.3 Punteros como argumentos de funciones.

Los punteros como argumentos suelen emplearse en el caso de funciones que devuelven más de un valor simple dado que C pasa los argumentos de las funciones por valor. La técnica consiste en usar como argumentos punteros a las variables que se desean modificar, de forma que lo que realmente es pasado por valor es la dirección de memoria de la variable, pudiendo de esta forma modificar el contenido de dichas direcciones.

2.5.4 Variables externas.

Un programa C consta de un conjunto de objetos externos, que son variables o funciones. El adjetivo externo se usa principalmente en contraste con interno, que describe los argumentos y las variables automáticas definidas dentro de las funciones. Las variables externas se definen fuera de cualquier función y, por tanto, son potencialmente utilizables por muchas funciones. También éstas son siempre externas, pues C no permite definir funciones dentro de otras. Por definición, las variables externas son también globales, por lo que todas las referencias a tales variables del mismo nombre son referencias a la misma cosa.

Las variables externas son accesibles globalmente, y por ello ofrecen una alternativa a los argumentos de las funciones y valores de retorno para comunicación de datos en ellas. Cualquier función puede acceder a una variable externa referenciándola por su nombre, si ha sido declarada de algún modo.

Las variables automáticas son internas a una función; comienzan a existir al entrar en la rutina y desaparecen al acabar. Por otro lado, las variables externas son permanentes. No aparecen y desaparecen, sino que mantienen su valor entre una invocación de la función y la siguiente.

2.5.5 Reglas sobre campo o ámbito de validez (*scope*).

El campo de validez de un nombre es la parte del programa en la que está definido. Para una variable declarada al principio de una función, el campo de validez es la función en la que se declara el nombre. Eso mismo vale para los argumentos de las funciones.

El campo de validez de una variable externa abarca desde el punto de su declaración en un archivo fuente hasta el fin del archivo.

Por otro lado, si se ha de hacer referencia a una variable externa antes de su definición o si está definida en un archivo fuente que no es aquel en que se usa, es obligatoria una declaración `extern`.

Es importante distinguir entre la declaración de una variable externa y su definición. Una declaración da a conocer las propiedades de una variable; una definición causa también una asignación de memoria. Sólo debe haber una definición de una variable externa entre todos los archivos que constituyen el programa fuente. Los otros archivos podrán contener declaraciones `extern` para acceder a él. La inicialización de una variable externa sólo puede acompañar a la definición.

En grandes programas, la facilidad de incluir archivos en `#include`, permite mantener una sola copia de las declaraciones `extern` del programa, pero ha de insertarse en cada archivo fuente que se compile.

2.5.6 Variables estáticas.

Las variables `static` pueden ser internas o externas. Las variables `static` internas son locales a una función en la misma forma que las automáticas pero, a diferencia de ellas, su existencia es permanente, en lugar de aparecer y desaparecer al activar la función. Esto significa que las variables `static` internas proporcionan un medio de almacenamiento permanente y privado en una función.

Una variable `static` externa es accesible en el resto del archivo fuente en el que está declarada, pero no en otro. Por tanto, el almacenamiento `static` externo proporciona un medio de ocultar nombres, pero serán externas para poder ser compartidas.

Normalmente, las funciones son objetos externos: sus nombres son conocidos globalmente. Sin embargo, puede declararse una función `static`, y así su nombre será inaccesible fuera del archivo en que se declare.

En C, `static` no sólo indica permanencia, sino también cierto grado de privacidad. Los objetos `static` internos son conocidos solamente dentro de una función. Los objetos `static` externos (variables o funciones) se conocen sólo en el archivo fuente en que aparecen y sus nombres no interfieren con variables ni funciones que llevan el mismo nombre en otros archivos.

Las variables y funciones `static` externas proporcionan un medio para ocultar objetos y rutinas que las manipulan de modo que otras rutinas y datos no puedan entrar en conflicto con ellas, ni siquiera inadvertidamente.

2.5.7 Variables registro.

Una declaración `register` avisa al compilador que la variable será muy usada. Cuando es posible, las variables `register` se colocan en los registros de la máquina, lo que producirá programas más cortos y más rápidos. La declaración register es del tipo

```
register int x;
register char c;
```

La parte `int` puede omitirse; `register` sólo se aplica a variables automáticas y a los parámetros formales de una función.

En la práctica existen algunas restricciones en las variables registro, que reflejan la realidad del hardware subyacente. Sólo unas pocas variables de cada función se pueden mantener en registros, y sólo se permiten algunos tipos. La palabra `register` se ignora si hay declaraciones excesivas o no permitidas. No es posible tomar la dirección de una variable registro.

2.5.8 Estructura de bloques.

C no es un lenguaje estructurado en bloques, y en él no pueden definirse funciones dentro de otras. Por otro lado, las variables sí se pueden definir en una forma estructurada en bloques. Las declaraciones de variables se colocan después de la llave de apertura que introduce cualquier sentencia compuesta, no solamente al comienzo de una función. Las variables declaradas así se solapan con cualquier variable del mismo nombre en bloques externos, y permanecen hasta que se alcanza la llave de cierre.

2.5.9 Inicialización.

En ausencia de una inicialización explícita, se garantiza que las variables externas y estáticas tendrán inicialmente el valor cero. Las variables automáticas y registro tienen valores indefinidos. Las variables simples (ni arrays ni estructuras) se inicializan al declararse, con sólo escribir luego de su nombre un signo igual y una expresión constante.

En las variables externas estáticas la inicialización se realiza una sola vez, en tiempo de compilación. Para las automáticas y registro se efectúa cada vez que se entra en la función o bloque.

En las variables automáticas y registro el inicializador no se limita a una constante: de hecho puede ser cualquier expresión válida que contenga valores definidos previamente, incluso llamadas a función.

Los arrays automáticos no pueden ser inicializados. Los arrays externos y estáticos se pueden inicializar haciendo que a la declaración le siga una lista de inicializadores entre llaves, separados por comas.

2.6 Macros.

Una definición de la forma

```
#define SI 1
```

indica que debe efectuarse una sustitución del tipo más sencillo (reemplazamiento de un nombre por una cadena de caracteres). En `#define` los nombres tienen la misma forma que los identificadores. El texto para reemplazarlos es arbitrario. Normalmente el texto de reemplazo está constituido por el resto de la línea. Una definición larga puede continuarse si se coloca un \ al final de la línea. El "ámbito" de un nombre definido mediante `#define` se entiende desde el punto de su definición hasta el final del archivo fuente. Los nombres pueden redefinirse y las definiciones contener otras realizadas previamente. En las cadenas entrecomilladas no se efectúan sustituciones.

También es posible definir macros con argumentos en los que el texto de reemplazo depende de la forma en que se llama la macro. Por ejemplo

```
#define max(A,B) ((A)>(B)?(A):(B))
```

Ahora la línea `x=max(p+q,r+s);` se sustituirá por `x=((p+q)>(r+s)?(p+q):(r+s));`. Esto proporciona una función que expande el código en línea en lugar de efectuar una llamada a función. Si los argumentos se tratan consistentemente, esta macro servirá para cualquier tipo de datos.

Sin embargo, en la anterior expansión, la expresión se evalúa dos veces. Esto es malo en caso de que existan efectos secundarios involucrados, tales como llamadas a función y operadores de incremento.

2.7 Control de flujo.

2.7.1 Proposiciones y bloques.

Una expresión como `x=0` o `i++` o `printf(...)` se convierte en una proposición cuando va seguida por punto y coma. En el lenguaje C, el punto y coma es un terminador de sentencia, y no un separador. Con las llaves `{` y `}` se agrupan declaraciones y sentencias en una proposición compuesta o bloque, y son

sintácticamente equivalentes a una proposición simple. Las llaves que rodean las proposiciones de una función son un ejemplo de ello. Las situadas alrededor de proposiciones múltiples después de `if`, `else`, `while` o `for` son otro ejemplo. Nunca se pone punto y coma después de la llave derecha que cierra un bloque.

2.7.2 If-Else.

La proposición `if-else` sirve para tomar decisiones. La sintaxis es:

```
if (expresión)
   proposición 1
else
   proposición 2
```

donde la parte `else` es opcional. Se evalúa la expresión. Si es "cierta" (valor distinto de cero), se ejecuta la `proposición 1`. Si es "falsa" (cero) y existe la parte `else`, se ejecuta `proposición 2`.

2.7.3 Switch.

La proposición `switch` es una herramienta especial para decisiones múltiples que comprueba si una expresión iguala uno entre varios valores constantes, y se bifurca a partir de ellos.

```
switch (expresión)
{
   case valor 1:
                proposición 1
                break;
   case valor 2:
                proposición 2
                break;
   ...
   default:
                proposición
                break;
}
```

El `switch` evalúa la expresión entre paréntesis y compara su valor con todos los casos. Cada opción debe ser etiquetada con un entero, constante de carácter o expresión constante. Si algún `case` corresponde al valor de la expresión en el `switch`, la ejecución comenzará en la proposición etiquetada por ese `switch`. La opción etiquetada `default` se ejecuta si no se satisface ninguna de las otras. La parte `default` es opcional. En caso de no existir y no verificarse ninguna opción, no se realiza ninguna acción. Los `case` y `default` pueden aparecer en cualquier orden. Las opciones deben ser todas diferentes.

La proposición `break` hace que se produzca una salida inmediata de la instrucción `switch`. Debido a que cada `case` actúa como una etiqueta, después de la ejecución continúa en el siguiente, a no ser que se tome alguna acción explícita para abandonarlo. Las proposiciones `break` y `return` son los métodos más normales de acabar un `switch`. También puede utilizarse una proposición `break` para provocar una terminación inmediata de los ciclos `while`, `for` y `do`.

2.7.4 Iteraciones: while y for.

En

```
while (expresión)
   proposición
```

se evalúa expresión. Si no es cero, se ejecuta la proposición y se vuelve a evaluar la expresión. El ciclo continúa hasta que expresión llega a valer cero.

La proposición `for`

```
for (expr1;expr2;expr3)
   proposición
```

es equivalente a

```
expr1;
while (expr2)
{
   proposición
   expr3;
}
```

Gramaticalmente, los tres componentes de un `for` son expresiones. Más comúnmente, expr1 y expr3 son asignaciones o llamadas a función y expr2 es una expresión de relación. Puede omitirse cualquiera de las tres partes, aunque deben permanecer los puntos y coma. Si es necesario, simplemente se eliminan en la expresión expr1 o expr2. Si la comprobación, expr2, no está presente, se supone siempre que es cierta.

Otro operador de C es la coma ",", que casi siempre encuentra aplicación en la sentencia `for`. Una pareja de expresiones separadas por una coma se evalúa de izquierda a derecha, y el tipo del resultado es el mismo que el del operando derecho. Por tanto, en una sentencia `for` es posible situar varias expresiones en cada una de sus partes; por ejemplo, para procesar varios índices de forma simultánea.

2.7.5 Iteraciones: do-while.

Las iteraciones `while` y `for` comparten la útil característica de comprobar la condición de terminación al comienzo en lugar de hacerlo al final. El tercer tipo de iteración en C, `do-while`, hace la comprobación al final después de cada pasada a través del cuerpo del ciclo. Y éste se ejecuta siempre al menos una vez. La sintaxis es:

```
do
   proposición
while (expresión);
```

Primero se ejecuta `proposición` y a continuación se evalúa `expresión`. En caso de ser cierta, se ejecuta `proposición` de nuevo y así sucesivamente. La iteración termina cuando la expresión se convierte en falsa.

2.7.6 Break.

A veces es necesario tener la posibilidad de salir de un ciclo por un lugar distinto al de las comprobaciones del comienzo o del final. La sentencia `break` proporciona una salida forzada de `for`, `while` y `do`, en la misma forma que `switch`. Una sentencia `break` obliga a una salida inmediata del ciclo (o `switch`) más interior.

2.7.7 Continue.

La proposición `continue` obliga a ejecutar la siguiente iteración del ciclo (`for`, `while`, `do`) que la contiene. En `while` y `do`, significa que la parte de comprobación de la condición se ejecute inmediatamente; en `for`, el control pasa a la etapa de reinicialización.

2.7.8 Saltos (goto) y etiquetas.

El lenguaje C contiene la sentencia `goto`, junto con etiquetas a las que saltar.

2.8 Punteros y arrays.

En C existe una estrecha relación entre punteros y arrays, suficientemente estrecha como para que se los trate simultáneamente. Cualquier operación que se pueda realizar mediante la indexación de un array se puede realizar también con punteros.

La declaración

```
int a[10];
```

define un array de tamaño 10, es decir un bloque de 10 objetos consecutivos denominados `a[0]`, `a[1]`, ... , `a[9]`. La notación `a[i]` significa el elemento del array que se encuentra a i posiciones del comienzo. Si `pa` es un puntero a un entero, entonces la asignación `pa=&a[0]` hace que `pa` apunte al elemento cero de `a`; es decir, `pa` contiene la dirección de `a[0]`. Ahora la asignación `x=*pa` copiará el contenido de `a[0]` en `x`.

Si `pa` apunta a un elemento particular de un array `a`, entonces por definición `pa+1` apunta al siguiente elemento, y en general `pa-i` apunta a `i` elementos antes de `pa`, y `pa+i` apunta `i` elementos después.

La correspondencia entre indexación y aritmética de punteros es muy estrecha. El compilador convierte toda referencia a un array en un puntero al comienzo del array. El efecto es que el nombre de un array es una expresión de tipo puntero. Sin embargo, hay una diferencia entre el nombre de un array y un puntero y se la debe tener en cuenta. Un puntero es una variable, por lo que operaciones como `pa=a` y `pa++` son correctas. Pero el nombre de un array es una constante, no una variable; de ahí que las construcciones como `a=pa` o `a++` o `p=&a` sean ilegales.

Cuando se pasa el nombre de un array a una función, se pasa la dirección del comienzo del array. En la función, este argumento es una variable, igual que cualquier otra, por lo que el nombre de un array como argumento es un puntero, o sea una variable que contiene una dirección.

Las definiciones de parámetros formales `char s[]` y `char *s` son completamente equivalentes; la forma como deben escribirse depende del tipo de expresiones en que aparezcan. Cuando se pasa el nombre de un array a una función, la función puede establecer a su conveniencia si va a manejar un array o un puntero y hacer las manipulaciones de acuerdo con esto.

Por último, indicar que C99 introdujo la posibilidad de incluir arrays de longitud variable. Estos arrays se declaran igual que el resto de arrays, con la salvedad de que el tamaño del mismo no es una expresión constante.

2.9 Estructuras.

La palabra clave struct introduce la declaración de una estructura que no es más que una lista de declaraciones encerrada entre llaves. Opcionalmente puede seguir un nombre a la palabra clave struct denominado nombre de la estructura. Los elementos de una estructura se denominan miembros. La llave de cierre que termina la lista de miembros puede ir seguida de una lista de variables, como si se tratara de un tipo básico.

```
struct
{
   ...
} x,y,z;
```

Se accede a uno de sus miembros mediante los operadores "." o "->", dependiendo de si accedemos mediante una variable del tipo estructura o mediante un puntero a dicha estructura.

3 Estructura de un programa.

Un programa es un conjunto de definiciones de funciones. La comunicación entre las funciones se efectúa a través de argumentos y valores devueltos por la función. También se puede realizar mediante variables externas.

Las funciones aparecen en cualquier orden en el archivo fuente, y el programa fuente puede estar dividido en varios archivos. Sin embargo, las funciones son indivisibles. `main` es una de estas funciones. Normalmente, se puede dar a las funciones cualquier nombre, pero `main` es un nombre especial (el programa comienza a ejecutarse al principio de `main`). Esto significa que todos los programas deben tener una función `main` en algún sitio. Casi siempre `main` hará uso de otras funciones para efectuar su trabajo. Algunas estarán en el mismo programa y otras en bibliotecas de funciones escritas previamente.

La estructura típica de un programa en C sería:

```
/* CABECERA DEL PROGRAMA */
Directivas del PREPROCESADOR
Declaraciones de VARIABLES GLOBALES

/*Declaración del tipo de dato devuelto por cada función */
Tipo funcion1(tipo,tipo,...);
...
Tipo funcionN(tipo,tipo,...);

/*Definición de cada una de las funciones*/
funcion1()
{
...;
}
...
funcionN()
{
...;
}

/* función principal MAIN */
main ()
{
...;
```

```
}
```

3.1 Argumentos de la función main.

Cuando se invoca `main` al comienzo de la ejecución, se llama con dos argumentos. El primero (llamado `argc` por convenio) es el número de argumentos en la línea de comandos con que se invocó al programa; el segundo (`argv`) es un puntero a un array de cadenas de caracteres que contienen los argumentos, uno por cadena.

Por convenio, `argv[0]` es el nombre con el que se invocó el programa; por tanto, `argc` es como mínimo 1.

3.2 Ficheros cabecera y prototipo de funciones.

Ya hemos comentado que un programa C puede constar de varios ficheros de código fuente. Para evitar tener que declarar las funciones creadas en cada uno de los ficheros fuente, se introduce la declaración o prototipo de la función en un fichero cabecera, de forma que basta con realizar un #include de dicho fichero para tener todas las declaraciones. De esta forma, la estructura de cada fichero fuente podría ser la siguiente:

```
/*FICHERO DE CABECERA DEL FICHERO FUENTE 1*/
#ifndef _FICHERO_FUENTE_1_
#define _FICHERO_FUENTE_1_

/*Prototipo de las funciones */
Tipo función1(tipo,tipo,...);
...
Tipo funciónN(tipo,tipo,...);
#endif
```

```
/*FICHERO FUENTE 1*/
#include "FICHERO DE CABECERA DEL FICHERO FUENTE 1"

/*Definición de cada una de las funciones*/

funcion1()
{
 ..;
}
...
funcionN()
{
...;
}
```

La estructura del programa principal sería:

```
/*FICHERO FUENTE PRINCIPAL*/
#include "FICHERO DE CABECERA DEL FICHERO FUENTE 1"
...
#include "FICHERO DE CABECERA DEL FICHERO FUENTE 2"

main ()
{
...;
...;
}
```

4 Funciones de librería y usuario.

El proceso de generación de programas ha evolucionado constantemente con el fin de obtener una mayor eficiencia de los programas y un mejor aprovechamiento de los recursos del sistema. Inicialmente se comprobó que muchos módulos eran utilizados una y otra vez por diferentes programas, por consiguiente se llegó a la conclusión que sería bueno agrupar todas aquellas funciones de propósito general en librerías de módulos. De esa forma se puede desarrollar nuevo software reutilizando aquellos módulos ya escritos.

Una librería no es más que un archivo donde están agrupados varios módulos o funciones, de tal forma que el enlazador entiende su formato. Para un acceso más rápido del enlazador se añade un índice al principio del fichero que ayuda a identificar los módulos e identificadores que contiene sin tener que recorrer todo el fichero.

Todo archivo fuente que utilice funciones de una biblioteca deberá contener una directiva `#include` al principio para incluir el fichero de cabecera que contiene el prototipo de las funciones de la biblioteca. El fichero de declaraciones define ciertas macros y variables empleadas por la biblioteca, además de declarar las funciones y estructuras de datos manejados. El uso de los ángulos en lugar de las comillas instruye al compilador para buscar el archivo en un directorio con información estándar.

Las funciones de biblioteca no son parte del lenguaje en sí, pero las incluyen todas las implementaciones. Son llamadas a subprogramas ya hechos y localizados en las bibliotecas de C. Hay funciones de biblioteca para realizar las operaciones estándar de entrada/salida, operaciones sobre caracteres, operaciones sobre cadenas de caracteres, cálculos matemáticos, etc.

Para incluir funciones de biblioteca en nuestros programas debemos invocar el archivo de cabecera correspondiente con la directiva #include y el archivo correspondiente entre los signos < y >. Algunos de los ficheros cabecera más utilizados son: <stdio.h> (*standard I-O*), <math.h> (*mathematics functions*), etc.

Las operaciones de entrada y salida no forman parte del lenguaje C. La biblioteca estándar de entrada/salida es un conjunto de funciones diseñadas para proporcionar un sistema estándar de entrada/salida a los programas C, siendo parte del C estándar ANSI. Se pretende que las funciones presenten una interfaz conveniente para la programación, y reflejen las operaciones que proporcionan la mayoría de los sistemas operativos modernos. Las rutinas son lo suficientemente eficientes como para que los usuarios rara vez las eviten por razón de eficiencia. Por último las rutinas están destinadas a ser portables, en el sentido de que deberán existir en un formato compatible con todo sistema que disponga de C, y que los programas que limiten su interacción con el sistema a las facilidades proporcionadas por la biblioteca estándar se puedan trasladar de un sistema a otro sin cambios esenciales.

5 Herramientas para la elaboración y depuración de programas en lenguaje C.

Existen múltiples herramientas para el desarrollo de programas en C. Normalmente son entornos integrados que permiten el desarrollo, la compilación, y depuración del código. Sin embargo, suelen ser entornos creados por fabricantes de software de desarrollo concretos para sus productos. Nosotros vamos a presentar las herramientas disponibles para el desarrollo de programas en lenguaje C que se distribuyen bajo licencia GNU. Estas herramientas se encuentran disponibles en

cualquier sistema UNIX, existiendo además versiones de los mismos para otros sistemas como por ejemplo MS-DOS.

5.1 El compilador.

El compilador es la herramienta que nos ayuda a traducir el lenguaje fuente a código binario, que es el que la máquina es capaz de ejecutar. El comando para usar el compilador de lenguaje C es `cc`. En este apartado, el compilador que vamos a estudiar es el de C y C++ de GNU (`gcc` - *the GNU C compiler*). Este compilador es de libre distribución y se encuentra disponible para casi todas las plataformas existentes, incluido LINUX. Su uso es aproximadamente el mismo que el de `cc` y con las mismas opciones que éste. La principal diferencia entre ambos es que `gcc` es compatible ANSI, mientras que `cc` sólo soporta la versión Kernighan & Ritchie.

El proceso de compilación con `gcc` consta de varias etapas: preprocesado, compilación, ensamblado y enlazado.

- Preprocesado. Realiza el procesado de todo el conjunto de órdenes del fichero que comienzan con el carácter almohadilla (#). Para ello se hace la expansión de todas las macros que entiende el preprocesador por su correspondiente código.
- Compilación. Realiza el proceso de transformación de un lenguaje de alto nivel a ensamblador. Por cada fichero fuente se obtiene un fichero en ensamblador. Se hace un análisis del programa. Este análisis tiene por objeto comprobar que el código fuente es correcto sintácticamente y semánticamente. A continuación se realiza la generación de código ensamblador de acuerdo a la plataforma de destino.
- Ensamblado. Recoge el código en ensamblador generado por la etapa de compilación y lo transforma en código objeto, con las referencias a objetos externos todavía sin resolver.
- Enlazado. Se cogen todos los ficheros objeto que componen el programa y se enlazan junto con las librerías necesarias en un fichero ejecutable. En esta fase se resuelven todas las referencias externas a otros objetos.

Estas etapas se realizan siempre en este orden y el propio compilador da la posibilidad de parar en cualquiera de las etapas, añadiendo simplemente un parámetro en la línea de compilación.

La forma de invocar al compilador es mediante el comando:

```
$ gcc [opciones] fichero1.c [fichero2.c] [fichero3.c] ...
```

Las opciones más importantes, que se pueden dar al compilador `gcc` son:

- `-Dnombre=definición` Permite realizar definiciones equivalentes a la macro del preprocesador `#define`. Si se pone `-DPRUEBA=yes`, equivale a poner al comienzo del fichero `#define PRUEBA yes`.
- `-Unombre` Permite eliminar definiciones. Es equivalente a la sentencia `#undef`.
- `-c` Se obtiene el código objeto del fichero. No se realiza la última etapa de enlace del fichero.
- `-o nombre_fichero` Nombre del fichero donde se pondrá el fichero resultante de la compilación.

- `-g` Se añade la información necesaria al ejecutable para que el programa pueda ser depurado simbólicamente.
- `-Wall` Muestra información de los warning o avisos, que no constituyen errores, pero que pueden suponer pistas de posibles fallos en el programa.
- `-L directorio` Indica al compilador en qué directorios se encuentran las librerías para la etapa de enlazado.
- `-l librería` Indica al enlazador una nueva librería donde deberá buscar las funciones referenciadas. Si se pone `–lm` buscará dentro de las librerías matemáticas que se encuentran en el fichero `libm.a` o `libm.so`.

5.2 El preprocesador.

El preprocesado es una etapa previa al proceso de compilación propiamente dicho, en la que se incluyen archivos, se expanden macros y se toman decisiones sobre qué parte de código se pasa a compilar y qué parte no. El preprocesador recorre el fichero fuente en busca de una serie de directivas y realiza distintas operaciones a partir de las mismas.

Todas las directivas del preprocesador llevan el prefijo # (almohadilla) como primer carácter de la línea, y como no se consideran sentencias del programa en C no llevan punto y coma al final. Las directivas más importantes que entiende el preprocesador son:

- #include: Esta directiva se utiliza para incluir en un fichero fuente el código contenido en otro archivo, como por ejemplo los ficheros cabecera. A continuación de la directiva se debe especificar el nombre del archivo, entre paréntesis angulares (`<nombre_fichero>`) o entre comillas dobles. Si se hace de la primera forma, el fichero se busca en el camino especificado por el compilador, si se pone de la segunda forma el fichero se busca en el directorio actual.
- #define: Sirve para sustituir una macro dentro del fichero por un texto determinado. Se utiliza para la definición de parámetros dentro de un programa.
- #undef: Sirve para anular la definición de una macro.
- Compilación condicional (#if, #ifdef, #ifndef, #else, #elif y #endif): Estas seis directivas se utilizan de forma similar a las sentencias del lenguaje C. Si se cumple la condición se introduce la porción de código, si no se ignora.

La utilidad `cpp` es la que realiza el preprocesado, puede ser invocada desde la línea de comandos. Cuando se compila un programa, el *gcc* ya se encarga de llamar al preprocesador con las opciones adecuadas.

5.3 El enlazador.

Otro de los procesos que realiza el compilador es el de enlazado. Una vez que se ha obtenido todos los códigos objeto de los ficheros que componen un programa se invoca al enlazador. La utilidad que realiza el enlazado se llama `ld` y puede ser invocada desde la línea de comandos independientemente. Esta utilidad enlaza todos los códigos objeto y resuelve todas las referencias externas que se tienen. Para resolver las referencias, debe buscar en todos los ficheros objeto que tiene y además en las librerías disponibles. Por defecto, el enlazador busca en las librerías estándar del sistema; no obstante, se le pueden indicar nuevas librerías donde debe buscar las referencias.

5.4 La utilidad make.

Resulta engorroso el escribir, cada vez que se compila un fichero, todas las opciones del compilador, además un ejecutable puede estar compuesto por múltiples ficheros fuente, con lo cual el número de invocaciones al compilador se ve incrementado. Para evitar este problema, se suele utilizar una herramienta de construcción automática de ejecutables. La herramienta `make` determina automáticamente qué elementos de un programa necesitan ser recompilados, y genera automáticamente las órdenes necesarias para hacerlo. Aunque normalmente se utiliza para la generación de programas en C o C++, se puede utilizar para cualquier lenguaje cuyo compilador pueda ser invocado desde la línea de comandos.

Para utilizar la utilidad `make` se debe escribir un fichero llamado `makefile`, en el que se escriben las relaciones existentes entre los ficheros que componen el programa, y se proporcionan los comandos para actualizar cada fichero. Una vez creado el fichero `makefile`, cada vez que el usuario desee compilar la aplicación, bastará con que escriba `make` para que recompile sólo los ficheros que han sido modificados.

El fichero *makefile* está formado por reglas con el siguiente aspecto:

```
objetivo...: dependencias...
      comandos
...
```

El objetivo suele ser el nombre del fichero que va a ser generado. Un objetivo también puede ser el nombre de una acción a realizar. La dependencia es el fichero o conjunto de ficheros que son necesarios para crear el objetivo. Un comando es una acción que se debe llevar a cabo para obtener el objetivo. Si para obtener un objetivo se necesitan varios comandos, cada uno de ellos se pondrá en una línea con un tabulador al principio de la línea. Los comandos se ejecutarán en el orden en el que aparecen dentro de la regla.

Si se llama a la utilidad `make` con un objetivo como parámetro, la herramienta busca en el fichero `makefile` la regla que debe aplicar. Una vez encontrada la regla, comprueba si el fichero objetivo tiene fecha anterior a la fecha de las dependencias y, si es así, el fichero objetivo se considera obsoleto y vuelve a crearlo ejecutando los comandos que se indican en la regla. Si no existen dependencias el objetivo será considerado obsoleto siempre.

Veamos un sencillo ejemplo:

```
/* ------------------- */
/* Fichero: potencia.h */
/* ------------------- */

#ifndef _POTENCIA_H_
#define _POTENCIA_H_

long potencia(long,long);

#endif
```

```
/* ------------------- */
/* Fichero: potencia.c */
/* ------------------- */

#include "potencia.h"

long potencia(long base,long exponente)
```

```
{
   long resultado=1;

   while(exponente-- > 0)
      resultado*=base;
   return(resultado);
}
```

```
/* ------------------- */
/* Fichero: eleva.c */
/* ------------------- */

#include <stdio.h>
#include <stdlib.h>
#include "potencia.h"

int main(int argc,char *argv[])
{
   long base,exponente;

   if(argc!=3)
      fprintf(stderr,"Usage: %s base exponente\n",argv[0]);
   else
   {
      base=atol(argv[1]);
      exponente=atol(argv[2]);
      printf("El resultado es %ld\n",potencia(base,exponente));
   }
   return 0;
}
```

La línea de comando que debemos teclear para realizar la compilación es:

```
$ gcc -o eleva eleva.c potencia.c
```

Por tanto, podríamos crear el siguiente *makefile*:

```
#Makefile básico. Ejemplo

eleva : eleva.o potencia.o
      gcc -o eleva eleva.o potencia.o

eleva.o : eleva.c potencia.h
      gcc -c eleva.c

potencia.o : potencia.c potencia.h
      gcc -c potencia.c
```

Al invocar a la herramienta *make* sin ningún objetivo como parámetro, se intenta conseguir el objetivo que aparece en primer lugar en el fichero *makefile* (en realidad si se llamase a la herramienta como *make eleva*, produciría el mismo efecto).

Antes de procesar una regla, la herramienta *make* debe procesar las reglas necesarias para construir los ficheros que aparecen como dependencias en la misma. En el ejemplo, antes de realizar la regla *eleva*, deberá ejecutar las reglas en las que los ficheros *eleva.o* y *potencia.o* son objetivo. Una vez terminadas estas reglas, se procesa la regla *eleva*.

Si el usuario modificase, por ejemplo, el fichero *potencia.c*, la herramienta *make* sólo compilaría el fichero *potencia.c* y luego realizaría el enlazado de todos los ficheros

para generar de nuevo el fichero ejecutable *eleva*; no gastaría tiempo compilando los ficheros que no han sido modificados.

En el ejemplo visto anteriormente, se puede ver que en la primera regla se listan los ficheros de dependencias dos veces. Para simplificar el tener que escribir dos veces las mismas palabras, se utilizan las variables en los ficheros *makefile*. Por ejemplo, se podría definir una variable como *OBJETOS = eleva.o potencia.o* y después, en cada lugar donde deba aparecer el valor de la variable, se pondría la variable.

Como se puede ver, en el fichero *makefile* puesto de ejemplo anteriormente, la orden para compilar un código fuente es siempre la misma: *gcc -o fichero.c*. En realidad no es necesario escribir siempre el comando para compilar un fichero La herramienta *make*, tiene definidas unas reglas implícitas que aplica por defecto. La orden implícita para obtener un fichero objeto a partir de un fichero fuente en C es:

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
```

Estas órdenes implícitas hacen referencia a ciertas variables ya predefinidas, aunque pueden ser modificados sus valores, bien pasándolos como argumento al llamar al *make* o introduciendo una variable de entorno dentro del *makefile*. Algunas de las variables predefinidas dentro del make son:

- CC Compilador de C, por defecto *cc*.
- CPP Preprocesador de C, por defecto *$(CC) -E*.
- MAKE Comando para realizar un *make*, por defecto *make*.
- RM Comando para borrar un fichero, por defecto *rm -f*.
- $@ Nombre del fichero objetivo de la regla.
- $< Nombre de la primera dependencia de una regla.
- $? Nombre de las dependencias que son más recientes que el objetivo.
- $^ Nombre de todas las dependencias, con espacios entre ellas.

Existen también una serie de variables, cuyo valor por defecto es la cadena vacía, y que son utilizados por las reglas implícitas. Algunos de estos valores son: CFLAGS (opciones del compilador de C), CPPFLAGS (opciones del preprocesador), y LDFLAGS (opciones para la fase de enlazado). Los comandos implícitos se aplican cuando el usuario no provee en la regla ningún comando.

El ejemplo de makefile que se puso al principio de la sección, utilizando variables y reglas implícitas, podría quedar reducido al siguiente fichero:

```
CC=gcc
CFLAGS=
CPPFLAGS=
OBJETOS=eleva.o potencia.o

eleva : $(OBJETOS)
	$(CC) -o $@ $(LDFLAGS) $(OBJETOS)

#el comando está implícito
eleva.o : eleva.c potencia.h
potencia.o : potencia.c potencia.h
```

Si se desea añadir la opción de depuración a toda la compilación sin tener que modificar el fichero makefile, basta con invocar a la herramienta pasándole como parámetro *CFLAGS = -g*.

5.5 Depuradores de código.

El propósito de un depurador es permitirnos ver lo que está ocurriendo en un programa cuando éste está corriendo y se produce un error de ejecución. El depurador que se va a analizar es el depurador de GNU `gdb` para C y C++. Este depurador es de libre distribución y se encuentra disponible para múltiples plataformas.

Todo depurador debe soportar cuatro funcionalidades importantes: arrancar un programa especificando cualquier cosa que afecte su comportamiento y hacer que un programa pare bajo unas condiciones específicas; examinar qué ha ocurrido en el programa cuando éste ha parado; y cambiar variables del programa, para experimentar y corregir los efectos de los errores.

Para poder ejecutar un programa bajo el depurador es necesario compilar el programa con un parámetro especial, se debe añadir la opción `-g`. El fichero ejecutable generado tendrá un mayor tamaño ya que en él se incluye la información necesaria para la depuración simbólica: tipos de datos de las variables y funciones, línea de código fuente asociada a cada fragmento de código ejecutable, etc.

Una vez obtenido el fichero ejecutable se invocará al depurador mediante `gdb [fichero_ejecutable]` y aparecerá en la pantalla un prompt donde se pueden introducir los comandos necesarios para depurar el programa.

La forma de parar la ejecución de un programa es poniendo puntos de ruptura o breakpoints. Existen también unos puntos de ruptura especiales, denominados watchpoints que detienen la ejecución cuando el valor de una expresión cambia su valor, sin tener que predecir el lugar exacto donde esto sucede.

Uno de los objetivos más importantes en el proceso de depuración es la capacidad de visualizar el contenido de las variables e incluso poder modificarlas en tiempo de ejecución. Para ello se utilizan los comandos: `print VARIABLE`, `display VARIABLE`, `undisplay VARIABLE`, `set variable VARIABLE=VALOR`.

6 Conclusiones.

C es un lenguaje de programación de propósito general. Fue originalmente diseñado para sistemas de programación, pero se ha utilizado con resultados satisfactorios en otras aplicaciones como sistemas de bases de datos, análisis numérico, etc.

C trabaja con la misma clase de objetos que la mayoría de las computadoras: caracteres, números y direcciones, que pueden ser combinados con los operadores aritméticos y lógicos, utilizados normalmente en las máquinas. No contiene operaciones para trabajar directamente con objetos compuestos, tales como cadenas de caracteres, conjuntos, listas, o arrays, considerados como un todo.

C sólo ofrece proposiciones (sentencias) de control de flujo sencillas, secuenciales, de selección, de iteración, bloques y subprogramas.