SECOND EDITION

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Prefacio

El mundo de la informática ha experimentado una revolución desde la publicación de *The C Programming Language* en 1978. Las grandes computadoras son mucho más grandes, y las computadoras personales tienen capacidades que rivalizan con las de los mainframes de hace una década. Durante este tiempo, C también ha cambiado, aunque solo modestamente, y se ha extendido mucho más allá de sus orígenes como el lenguaje del sistema operativo UNIX.

La creciente popularidad de C, los cambios en el lenguaje a lo largo de los años y la creación de compiladores por parte de grupos no involucrados en su diseño, se combinaron para demostrar la necesidad de una definición más precisa y contemporánea del lenguaje que la proporcionada por la primera edición de este libro. En 1983, el American National Standards Institute (ANSI) estableció un comité cuyo objetivo era producir "una definición inequívoca e independiente de la máquina de la lengua C", sin dejar de conservar su espíritu. El resultado es el estándar ANSI para C.

El estándar formaliza las construcciones que se insinuaron pero no se describieron en la primera edición, en particular la asignación de estructuras y las enumeraciones. Proporciona una nueva forma de declaración de función que permite la verificación cruzada de la definición con el uso. Especifica una biblioteca estándar, con un amplio conjunto de funciones para realizar tareas de entrada y salida, administración de memoria, manipulación de cadenas y tareas similares. Precisa el comportamiento de las características que no estaban detalladas en la definición original y, al mismo tiempo, establece explícitamente qué aspectos del lenguaje siguen dependiendo de la máquina.

Esta segunda edición del lenguaje de *programación C* describe C tal como lo define el estándar ANSI. Aunque hemos anotado los lugares donde ha evolucionado el idioma, hemos optado por escribir exclusivamente en la nueva forma. En su mayor parte, esto no hace una diferencia significativa; El cambio más visible es la nueva forma de declaración y definición de funciones. Los compiladores modernos ya son compatibles con la mayoría de las características del estándar.

Hemos tratado de mantener la brevedad de la primera edición. C no es un idioma grande, y no está bien servido por un libro grande. Hemos mejorado la exposición de características críticas, como los punteros, que son fundamentales para la programación en C. Hemos refinado los ejemplos originales y hemos añadido nuevos ejemplos en varios capítulos. Por ejemplo, el tratamiento de declaraciones complicadas se incrementa con programas que convierten declaraciones en palabras y viceversa. Al igual que antes, todos los ejemplos se han probado directamente a partir del texto, que está en forma legible por máquina.

El Apéndice A, el manual de referencia, no es la norma, pero nuestro intento de transmitir lo esencial de la norma en un espacio más pequeño. Está destinado a una fácil comprensión por parte de los programadores, pero no como una definición para los escritores de compiladores: ese papel pertenece propiamente al estándar en sí. El Apéndice B es un resumen de las instalaciones de la biblioteca estándar. También está destinado a ser referenciado por los programadores, no por los implementadores. El Apéndice C es un resumen conciso de los cambios con respecto a la versión original.

Como dijimos en el prefacio de la primera edición, C "se lleva bien a medida que crece la experiencia con él". Con una década más de experiencia, seguimos sintiéndolo así. Esperamos que este libro te ayude a aprender C y a utilizarlo bien.

Estamos profundamente en deuda con los amigos que nos ayudaron a producir esta segunda edición. Jon Bently, Doug Gwyn, Doug McIlroy, Peter Nelson y Rob Pike nos dieron comentarios perspicaces en casi todas las páginas de los borradores de los manuscritos. Agradecemos la lectura atenta de Al Aho, Dennis Allison, Joe Campbell, G.R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford y Chris van Wyk. También recibimos sugerencias útiles de Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo y Peter Weinberger. Dave Prosser respondió a muchas preguntas detalladas sobre el estándar ANSI. Utilizamos ampliamente el traductor de C++ de Bjarne Stroustrup para las pruebas locales de nuestros programas, y Dave Kristol nos proporcionó un compilador de C ANSI para las pruebas finales. Rich Drechsler ayudó mucho con la composición tipográfica.

Nuestro más sincero agradecimiento a todos.

Brian W. Kernighan,
Dennis M. Ritchie

Prefacio a la primera edición

C es un lenguaje de programación de propósito general con características de economía de expresión, control de flujo moderno y estructuras de datos, y un amplio conjunto de operadores. C no es un lenguaje de "muy alto nivel", ni uno "grande", y no está especializado en ninguna área de aplicación en particular. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y efectivo para muchas tareas que los lenguajes supuestamente más poderosos.

C fue diseñado e implementado originalmente en el sistema operativo UNIX en el DEC PDP-11, por Dennis Ritchie. El sistema operativo, el compilador de C y, esencialmente, todos los programas de aplicaciones de UNIX (incluido todo el software utilizado para preparar este libro) están escritos en
C. Los compiladores de producción también existen para varias otras máquinas, incluyendo el IBM System/370, el Honeywell 6000 y el Interdata 8/32. Sin embargo, C no está vinculado a ningún hardware o sistema en particular, y es fácil escribir programas que se ejecutarán sin cambios en cualquier máquina que admita C.

Este libro está destinado a ayudar al lector a aprender a programar en C. Contiene una introducción tutorial para que los nuevos usuarios comiencen lo antes posible, capítulos separados sobre cada característica principal y un manual de referencia. La mayor parte del tratamiento se basa en la lectura, la escritura y la revisión de ejemplos, más que en meras declaraciones de reglas. En su mayor parte, los ejemplos son programas completos y reales en lugar de fragmentos aislados. Todos los ejemplos se han probado directamente a partir del texto, que está en formato legible por máquina. Además de mostrar cómo hacer un uso eficaz del lenguaje, también hemos intentado, en la medida de lo posible, ilustrar algoritmos útiles y principios de buen estilo y diseño de sonido.

El libro no es un manual introductorio de programación; Supone cierta familiaridad con conceptos básicos de programación como variables, sentencias de asignación, bucles y funciones. No obstante, un programador novato debe ser capaz de leer y aprender el lenguaje, aunque el acceso a un colega más experto ayudará.

En nuestra experiencia, C ha demostrado ser un lenguaje agradable, expresivo y versátil para una amplia variedad de programas. Es fácil de aprender y se lleva bien a medida que crece la experiencia con él. Esperamos que este libro te ayude a utilizarlo bien.

Las críticas reflexivas y sugerencias de muchos amigos y colegas han contribuido en gran medida a este libro y a nuestro placer al escribirlo. En particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin y Larry Rosler leyeron varios volúmenes con cuidado. También estamos en deuda con Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson y Peter Weinberger por sus útiles comentarios en varias etapas, y con Mile Lesk y Joe Ossanna por su inestimable ayuda con la composición tipográfica.

Brian W. Kernighan,
Dennis M. Ritchie

Capítulo 1 - Introducción al tutorial

Comencemos con una introducción rápida en C. Nuestro objetivo es mostrar los elementos esenciales del lenguaje en programas reales, pero sin enredarnos en detalles, reglas y excepciones. En este punto, no estamos tratando de ser completos o incluso precisos (excepto que los ejemplos están destinados a ser correctos). Queremos llevarlo lo más rápido posible al punto en el que pueda escribir programas útiles, y para hacer eso tenemos que concentrarnos en lo básico: variables y constantes, aritmética, flujo de control, funciones y los rudimentos de entrada y salida. Estamos dejando intencionalmente fuera de este capítulo características de C que son importantes para escribir programas más grandes. Estos incluyen punteros, estructuras, la mayor parte del amplio conjunto de operadores de C, varias instrucciones de flujo de control y la biblioteca estándar.

Este enfoque y sus inconvenientes. Lo más notable es que la historia completa sobre ninguna característica en particular se encuentra aquí, y el tutorial, al ser breve, también puede ser engañoso. Y debido a que los ejemplos no usan toda la potencia de C, no son tan concisos y elegantes como podrían ser. Hemos tratado de minimizar estos efectos, pero estén advertidos. Otro inconveniente es que los capítulos posteriores necesariamente repetirán algo de este capítulo. Esperamos que la repetición te ayude más de lo que molesta.

En cualquier caso, los programadores experimentados deben ser capaces de extrapolar el material de este capítulo a sus propias necesidades de programación. Los principiantes deben complementarlo escribiendo sus propios programas pequeños y similares. Ambos grupos pueden usarlo como un marco en el que colgar las descripciones más detalladas que comienzan en el Capítulo 2.

1.1 Empezar

La única forma de aprender un nuevo lenguaje de programación es escribiendo programas en él. El primer programa a escribir es el mismo para todos los idiomas:

Imprime las palabras

```
Hola mundo
```

Este es un gran obstáculo; Para saltar sobre él, debe ser capaz de crear el texto del programa en algún lugar, compilarlo con éxito, cargarlo, ejecutarlo y averiguar a dónde fue su salida. Con estos detalles mecánicos dominados, todo lo demás es comparativamente fácil.

En C, el programa para imprimir "hola, `mundo`" es

```
#include <stdio.h>

main()
{
  printf("hola, mundo\n");
}
```

La forma de ejecutar este programa depende del sistema que esté utilizando. Como ejemplo específico, en el sistema operativo UNIX debe crear el programa en un archivo cuyo nombre termine en "`.c`", como `hello.c`, y luego compilarlo con el comando

```
cc hola.c
```

Si no ha estropeado nada, como omitir un carácter o escribir mal algo, la compilación procederá silenciosamente y creará un archivo ejecutable llamado `a.out`. Si ejecuta `a.out` escribiendo el comando

```
a.fuera
```

él imprimirá

```
Hola mundo
```

En otros sistemas, las reglas serán diferentes; Consulte con un experto local.

Ahora, algunas explicaciones sobre el programa en sí. Un programa C, sea cual sea su tamaño, consta de *funciones* y *variables*. Una función contiene *instrucciones* que especifican las operaciones de cálculo que se van a realizar, y las variables almacenan los valores utilizados durante el cálculo. Las funciones de C son como las subrutinas y funciones de Fortran o los procedimientos y funciones de Pascal. Nuestro ejemplo es una función llamada `main`. Normalmente, usted tiene la libertad de dar a las funciones los nombres que desee, pero "main" es especial: su programa comienza a ejecutarse al principio de main. Esto significa que cada programa debe tener una `principal` en alguna parte.

`main` generalmente llamará a otras funciones para ayudar a realizar su trabajo, algunas que escribió y otras de las bibliotecas que se le proporcionan. La primera línea del programa,

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

le dice al compilador que incluya información sobre la biblioteca estándar de entrada/salida; la línea aparece al principio de muchos archivos fuente de C. La biblioteca estándar se describe en el Capítulo 7 y en el Apéndice B.

Un método de comunicación de datos entre funciones es que la función que llama proporcione una lista de valores, llamados *argumentos*, a la función a la que llama. Los paréntesis después del nombre de la función rodean la lista de argumentos. En este ejemplo, `main` se define como una función que no espera argumentos, lo que se indica mediante la lista vacía `( )`.

```
#include <stdio.h>include          Información sobre la biblioteca
estándar
main()                             Defina una función llamada main
                                   que no recibió ningún valor de
                                   argumento
{                                  Las instrucciones de main se encierran entre
llaves
    printf("hola, mundo\n");       Llamadas principales Función de biblioteca
    printf
                                   Para imprimir esta secuencia de caracteres
}                                  \n representa el carácter de nueva línea
```

El primer programa C

Las instrucciones de una función se encierran entre llaves `{ }`. La función `main` contiene solo una instrucción,

```
printf("hola, mundo\n");
```

Se llama a una función nombrándola, seguida de una lista de argumentos entre paréntesis, por lo que llama a la función `printf` con el argumento `"hello, world\n"`. `printf` es una función de biblioteca que imprime la salida, en este caso la cadena de caracteres entre las comillas.

Una secuencia de caracteres entre comillas dobles, como `"hola, mundo\n"`, se denomina *cadena de caracteres* o *constante de cadena*. Por el momento, nuestro único uso de cadenas de caracteres será como argumentos para `printf` y otras funciones.

La secuencia `\n` en la cadena es la notación C para el carácter de *nueva línea*, que cuando se imprime avanza la salida al margen izquierdo en la siguiente línea. Si omite el `\n` (un experimento que vale la pena), encontrará que no hay avance de línea después de imprimir la salida. Debes usar `\n` para incluir un carácter de nueva línea en el `argumento printf`; si intentas algo como

```
printf("hola, mundo");
```

el compilador de C producirá un mensaje de error.

`printf` nunca proporciona un carácter de nueva línea automáticamente, por lo que se pueden usar varias llamadas para construir una línea de salida en etapas. Nuestro primer programa bien podría haber sido escrito

```
#include <stdio.h>

main()
{
  printf("hola, ");
  printf("mundo");
  printf("\n");
}
```

para producir una salida idéntica.

Observe que `\n` representa solo un único carácter. Una *secuencia de escape* como `\n` proporciona un mecanismo general y extensible para representar caracteres invisibles o difíciles de escribir. Entre los otros que proporciona C se encuentran `\t` para la tabulación, `\b` para la barra espaciadora, `\"` para la comilla doble y `\\` para la barra invertida en sí. Hay una lista completa en la Sección 2.3.

Ejercicio 1-1. Ejecute el programa "hello, world" en su sistema. Experimente con omitir partes del programa, para ver qué mensajes de error recibe.

Ejercicio 1-2. Experimente para averiguar qué sucede cuando la cadena de argumentos de prints contiene
c, donde *c* es algún carácter no mencionado anteriormente.

1.2 Variables y expresiones aritméticas

El siguiente programa utiliza la fórmula oC=(5/9)(oF-32) para imprimir la siguiente tabla de temperaturas Fahrenheit y sus equivalentes centígrados o Celsius:

```
1    -17
20   -6
40   4
60   15
80   26
100  37
120  48
140  60
160  71
180  82
200  93
220  104
240  115
260  126
280  137
300  148
```

El programa en sí todavía consiste en la definición de una sola función llamada `main`. Es más largo que el que imprimía "hola, mundo", pero no es complicado. Presenta varias ideas nuevas, incluidos comentarios, declaraciones, variables, expresiones aritméticas, bucles y salidas con formato.

```
#include <stdio.h>

/* imprimir Tabla Fahrenheit-
    Celsius para fahr = 0, 20,
    ..., 300 */
main()
{
  Int Fahr, Celsius;
  int inferior, superior, escalón;

  inferior = 0;/* límite inferior de la escala de
  temperatura     */ superior = 300;/* límite superior
  */
  paso = 20;      /* tamaño del paso */

  fahr = más bajo;
  while (fahr <= superior) {
      Celsius = 5 * (Fahr-32) / 9;
      printf("%d\t%d\n", fahr, celsius);
      fahr = fahr + escalón;
  }
}
```

Las dos líneas

```
/* imprimir Tabla Fahrenheit-
    Celsius para fahr = 0, 20, ...,
    300 */
```

son un *comentario*, que en este caso explica brevemente lo que hace el programa. Cualquier carácter entre `/*` y `*/` es ignorado por el compilador; pueden ser usados libremente para hacer que un programa sea más fácil de entender. Los comentarios pueden aparecer en cualquier lugar donde pueda aparecer un espacio en blanco, una tabulación o una nueva línea.

En C, todas las variables deben declararse antes de ser utilizadas, normalmente al principio de la función antes de cualquier instrucción ejecutable. Una *declaración* anuncia las propiedades de las variables; consta de un nombre y una lista de variables, como

```
Int Fahr, Celsius;
int inferior, superior, escalón;
```

El tipo `int` significa que las variables enumeradas son números enteros; en contraste con `float`, que significa punto flotante, es decir, números que pueden tener una parte fraccionaria. El rango de `int` y `float` depende de la máquina que esté utilizando; int de 16 bits , que se

encuentran entre -32768 y

+32767, son comunes, al igual que los enteros de 32 bits. `Un número flotante` suele ser una cantidad de 32 bits, con al menos seis dígitos significativos y una magnitud generalmente entre 10-38 y 1038.

C proporciona varios otros tipos de datos además de `int` y `float`, entre ellos:

`carbonizar`	carácter: un solo byte
`corto`	Entero corto
`largo`	entero largo
`doble`	Punto flotante de doble precisión

El tamaño de estos objetos también depende de la máquina. También hay *matrices*, *estructuras* y *uniones* de estos tipos básicos, *punteros* a ellos y *funciones* que los devuelven, todos los cuales conoceremos a su debido tiempo.

El cálculo en el programa de conversión de temperatura comienza con las instrucciones de *asignación*

```
menor = 0;
superior = 300;
paso = 20;
```

que establecen las variables en sus valores iniciales. Las instrucciones individuales terminan con punto y coma.

Cada línea de la tabla se calcula de la misma manera, por lo que usamos un bucle que se repite una vez por línea de salida; este es el propósito del `bucle` while

```
while (fahr <= superior) {
   ...
}
```

El bucle `while` funciona de la siguiente manera: se prueba la condición entre paréntesis. Si es verdadero (`fahr` es menor o igual que `superior`), se ejecuta el cuerpo del bucle (las tres instrucciones encerradas entre llaves). A continuación, se vuelve a probar la condición y, si es verdadera, se vuelve a ejecutar el cuerpo. Cuando la prueba se vuelve falsa (`fahr` excede `upper`), el bucle finaliza y la ejecución continúa en la instrucción que sigue al bucle. No hay más declaraciones en este programa, por lo que termina.

El cuerpo de un `while` puede ser una o más instrucciones encerradas entre llaves, como en el convertidor de temperatura, o una sola instrucción sin llaves, como en

```
Mientras que (I < J)
    i = 2 * i;
```

En cualquier caso, siempre sangraremos las sentencias controladas por el `while` mediante una tabulación (que hemos mostrado como cuatro espacios) para que puedas ver de un vistazo qué sentencias están dentro del bucle. La sangría enfatiza la estructura lógica del programa. Aunque a los compiladores de C no les importa el aspecto de un programa, la sangría y el espaciado adecuados son fundamentales para que los programas sean fáciles de leer para las personas. Recomendamos escribir solo una declaración por línea y usar espacios en blanco alrededor de los operadores para aclarar la agrupación. La posición de los aparatos ortopédicos es menos importante, aunque las personas tienen creencias apasionadas. Hemos elegido uno de varios estilos populares. Elige un estilo que se adapte a ti y luego úsalo de manera constante.

La mayor parte del trabajo se realiza en el cuerpo del bucle. La temperatura Celsius se calcula y se asigna a la variable `celsius` mediante la declaración

```
Celsius = 5 * (Fahr-32) / 9;
```

La razón para multiplicar por 5 y dividir por 9 en lugar de simplemente multiplicar por `5/9` es que en C, como en muchos otros idiomas, la división de enteros *se trunca*: cualquier parte fraccionaria se descarta. Ya que `5` y `9` son números enteros. `5/9` se truncaría a cero y, por lo tanto, todas las temperaturas Celsius se informarían como cero.

Este ejemplo también muestra un poco más de cómo `funciona printf`. `printf` es una función de formato de salida de propósito general, que describiremos en detalle en el Capítulo 7. Su primer argumento es una cadena de caracteres que se imprimirá, en la que cada `%` indica dónde se va a sustituir uno de los otros argumentos (segundo, tercero, ...) y en qué forma se va a imprimir. Por ejemplo, `%d` especifica un argumento entero, por lo que la instrucción

```
printf("%d\t%d\n", fahr, celsius);
```

Hace que se impriman los valores de los dos enteros `Fahr` y `Celsius`, con una tabulación (`\t`) entre ellos.

Cada construcción % en el primer argumento de `printf` se empareja con el segundo argumento, tercer argumento, etc. correspondiente; deben coincidir correctamente por número y tipo, o obtendrá respuestas incorrectas.

Por cierto, `printf` no es parte del lenguaje C; no hay ninguna entrada o salida definida en C. `printf` es solo una función útil de la biblioteca estándar de funciones que normalmente son accesibles para los programas C. Sin embargo, el comportamiento de `printf` está definido en el estándar ANSI, por lo que sus propiedades deben ser las mismas con cualquier compilador y biblioteca que se ajuste al estándar.

Para concentrarnos en C en sí, no hablamos mucho sobre la entrada y la salida hasta el capítulo 7. En particular, aplazaremos la entrada formateada hasta entonces. Si tiene que introducir números, lea la discusión de la función `scanf` en la Sección 7.4. `scanf` es como `printf`, excepto que lee la entrada en lugar de escribir la salida.

Hay un par de problemas con el programa de conversión de temperatura. La más simple es que el resultado no es muy bonito porque los números no están justificados a la derecha. Eso es fácil de arreglar; Si aumentamos cada `%d` en la declaración `printf` con un ancho, los números impresos se justificarán a la derecha en sus campos. Por ejemplo, podríamos decir

```
printf("%3d %6d\n", fahr, celsius);
```

para imprimir el primer número de cada línea en un campo de tres dígitos de ancho, y el segundo en un campo de seis dígitos de ancho, así:

```
  0     -17
 20      -6
 40       4
 60      15
 80      26
100      37
...
```

El problema más grave es que debido a que hemos usado aritmética de números enteros, las temperaturas Celsius no son muy precisas; por ejemplo, 0oF es en realidad alrededor de -17.8oC, no -17. Para obtener respuestas más precisas, debemos usar la aritmética de punto flotante en lugar de la número entero. Esto requiere algunos cambios en el programa. Aquí está la segunda versión:

```
#include <stdio.h>

/* imprimir Tabla Fahrenheit-Celsius
    para fahr = 0, 20, ..., 300; Versión de coma flotante
*/ main()
{
  flotador Fahr, Celsius;
  flotar más abajo, arriba,
  escalón;

  inferior = 0;/* límite inferior de la escala de
  temperatura     */ superior = 300;/* límite superior */

  paso = 20;      /* tamaño del paso */

  fahr = más bajo;
  while (fahr <= superior) {
      Celsius = (5.0/9.0) * (Fahr-32.0);
      printf("%3.0f %6.1f\n", fahr, celsius);
      fahr = fahr + escalón;
  }
}
```

Esto es muy similar a lo anterior, excepto que `fahr` y `celsius` se declaran `flotantes` y la fórmula para la conversión se escribe de una manera más natural. No pudimos usar `5/9` en la versión anterior porque la división de enteros lo truncaría a cero. Sin embargo, un punto decimal en una constante indica que es de punto flotante, por lo que `5,0/9,0` no se trunca porque es la relación de dos valores de punto flotante.

Si un operador aritmético tiene operandos enteros, se realiza una operación entera. Sin embargo, si un operador aritmético tiene un operando de punto flotante y un operando entero, el entero se convertirá en punto flotante antes de que se realice la operación. Si hubiéramos escrito `(fahr-32)`, el `32` se convertiría automáticamente en coma flotante. Sin embargo, escribir constantes de punto flotante con puntos decimales explícitos, incluso cuando tienen valores enteros, enfatiza su naturaleza de punto flotante para los lectores humanos.

Las reglas detalladas para cuando los números enteros se convierten en coma flotante se encuentran en el Capítulo 2. Por ahora, observe que la tarea

```
fahr = más bajo;
```

y la prueba

```
while (fahr <= superior)
```

También funciona de forma natural: el `int` se convierte en `flotador` antes de que se realice la operación.

La `especificación de conversión` printf `%3.0f dice que un número de coma flotante (aquí` fahr`) debe imprimirse de al menos tres caracteres de ancho, sin punto decimal ni dígitos de fracción.`
`%6.1f` describe otro número (`Celsius`) que debe imprimirse de al menos seis caracteres de ancho, con 1 dígito después del punto decimal. El resultado es el siguiente:

```
  0   -17.8
 20    -6.7
 40     4.4
...
```

El ancho y la precisión se pueden omitir de una especificación: `%6f` dice que el número debe tener al menos seis caracteres de ancho; `%.2f` especifica dos caracteres después del punto decimal, pero el ancho no está restringido; y `%f` simplemente dice que se imprima el número como punto flotante.

`%d`	Imprimir como entero decimal
`%6d`	imprimir como entero decimal, de al menos 6 caracteres de ancho
`%f`	Imprimir como punto flotante
`%6f`	Imprimir como punto flotante, de al menos 6 caracteres de ancho
`%.2f`	imprimir como punto flotante, 2 caracteres después del punto decimal
`%6.2f`	Imprimir como punto flotante, al menos 6 de ancho y 2 después del punto decimal

Entre otros, `printf` también reconoce `%o` para octal, `%x` para hexadecimal, `%c` para carácter, `%s` para la cadena de caracteres y `%%` para sí mismo.

Ejercicio 1-3. Modifique el programa de conversión de temperatura para imprimir un encabezado encima de la tabla.

Ejercicio 1-4. Escriba un programa para imprimir la tabla correspondiente de Celsius a Fahrenheit.

1.3 La instrucción for

Hay muchas formas diferentes de escribir un programa para una tarea en particular. Probemos una variación en el convertidor de temperatura.

```
#include <stdio.h>

/* imprimir tabla Fahrenheit-Celsius */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Esto produce las mismas respuestas, pero ciertamente se ve diferente. Un cambio importante es la eliminación de la mayoría de las variables; Solo `queda Fahr`, y lo hemos convertido en un `int`. Los límites inferior y superior y el tamaño del paso aparecen solo como constantes en la `instrucción for`, que en sí misma es una nueva construcción, y la expresión que calcula la temperatura Celsius ahora aparece como el tercer argumento de `printf` en lugar de una instrucción de asignación separada.

Este último cambio es una instancia de una regla general: en cualquier contexto en el que se permita usar el valor de algún tipo, puede usar una expresión más complicada de ese tipo. Dado que el tercer argumento de `printf` debe ser un valor de punto flotante para que coincida con % `6.1f`, aquí puede aparecer cualquier expresión de punto flotante.

La instrucción `for` es un bucle, una generalización del `while`. Si lo comparas con el anterior, su funcionamiento debería ser claro. Dentro de los paréntesis, hay tres partes, separadas por punto y coma. La primera parte, la inicialización

```
Fahr = 0
```

se realiza una vez, antes de que se ingrese al bucle propiamente dicho. La segunda parte es la prueba o condición que controla el bucle:

```
Fahr <= 300
```

Se evalúa esta condición; si es verdadera, se ejecuta el cuerpo del bucle (en este caso un solo `ptintf`). A continuación, el paso de incremento

```
Fahr = Fahr + 20
```

se ejecuta y se vuelve a evaluar la condición. El bucle finaliza si la condición se ha convertido en falsa. Al igual que con el `while`, el cuerpo del bucle puede ser una sola instrucción o un grupo de instrucciones encerradas entre llaves. La inicialización, la condición y el incremento pueden ser cualquier expresión.

La elección entre `while` y `for` es arbitraria, en base a cuál parece más clara. El `for` suele ser adecuado para bucles en los que la inicialización y el incremento son instrucciones únicas y están relacionadas lógicamente, ya que es más compacto que `while` y mantiene las instrucciones de control de bucle juntas en un solo lugar.

Ejercicio 1-5. Modifique el programa de conversión de temperatura para imprimir la tabla en orden inverso, es decir, de 300 grados a 0.

1.4 Constantes simbólicas

Una última observación antes de dejar la conversión de temperatura para siempre. Es una mala práctica enterrar
Los "números mágicos" como el 300 y el 20 en un programa; transmiten poca información a alguien que podría tener que leer el programa más tarde, y son difíciles de cambiar de manera sistemática. Una forma de lidiar con los números mágicos es darles nombres significativos. Una `línea #define` define un *nombre simbólico o una constante simbólica* para que sea una cadena particular de caracteres:

`Lista de reemplazo de` *nombres #define*

A partir de entonces, cualquier aparición de *nombre* (que no esté entre comillas y que no forme parte de otro nombre) será reemplazada por el *texto de reemplazo* correspondiente. El *nombre* tiene la misma forma que el nombre de una variable: una secuencia de letras y dígitos que comienza con una letra. El *texto de reemplazo* puede ser cualquier secuencia de caracteres; no se limita a números.

```
#include <stdio.h>

#define INFERIOR 0/* límite inferior de la mesa
*/ #define SUPERIOR  300/* límite superior */

#define PASO   20    /* tamaño del paso */

/* imprimir tabla Fahrenheit-Celsius */
main()
{
    int fahr;

    for (fahr = INFERIOR; fahr <= SUPERIOR; fahr = fahr + PASO)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Las cantidades `LOWER, UPPER` y `STEP` son constantes simbólicas, no variables, por lo que no aparecen en las declaraciones. Los nombres de constantes simbólicas se escriben convencionalmente en mayúsculas para que puedan distinguirse fácilmente de los nombres de variables en minúsculas. Observe que no hay punto y coma al final de una `línea #define`.

1.5 Entrada y salida de caracteres

Vamos a considerar una familia de programas relacionados para el procesamiento de datos de caracteres. Encontrará que muchos programas son solo versiones ampliadas de los prototipos que discutimos aquí.

El modelo de entrada y salida soportado por la librería estándar es muy sencillo. La entrada o salida de texto, independientemente de dónde se origine o a dónde vaya, se trata como secuencias de caracteres. Un *flujo de texto* es una secuencia de caracteres divididos en líneas; cada línea consta de cero o más caracteres seguidos de un carácter de nueva línea. Es responsabilidad de la biblioteca hacer que cada flujo de entrada o salida confirme este modelo; el programador de C que usa la biblioteca no necesita preocuparse por cómo se representan las líneas fuera del programa.

La biblioteca estándar proporciona varias funciones para leer o escribir un carácter a la vez, de las cuales `getchar` y `putchar` son las más simples. Cada vez que se llama, `getchar` lee el *siguiente carácter de entrada* de una secuencia de texto y lo devuelve como su valor. Es decir, después de

```
c = getchar();
```

La variable `c` contiene el siguiente carácter de entrada. Los caracteres normalmente provienen del teclado; La entrada de archivos se discute en el Capítulo 7.

La función `putchar` imprime un carácter cada vez que se llama:

```
putchar(c);
```

Imprime el contenido de la variable entera `c` como un carácter, normalmente en la pantalla. Las llamadas a `putchar` y `printf` pueden ser intercaladas; la salida aparecerá en el orden en que se realizan las llamadas.

1.5.1 Copia de archivos

Dado `getchar` y `putchar`, puede escribir una cantidad sorprendente de código útil sin saber nada más sobre la entrada y la salida. El ejemplo más simple es un programa que copia su entrada a su salida un carácter a la vez:

```
Leer un carácter
    while (el carácter no es el indicador de fin
        de archivo) genera el carácter que se
        acaba de leer
        Leer un carácter
```

Al convertir esto en C, se obtiene:

```
#include <stdio.h>

/* copiar la entrada a la salida; 1ª
versión */ main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

El operador relacional `!=` significa "no igual a".

Lo que parece ser un carácter en el teclado o en la pantalla es, por supuesto, como todo lo demás, almacenado internamente como un patrón de bits. El tipo `char` está diseñado específicamente para almacenar dichos datos de caracteres, pero se puede usar cualquier tipo entero. Usamos `int` por una razón sutil pero importante.

El problema es distinguir el final de la entrada de los datos válidos. La solución es que `getchar` devuelve un valor distintivo cuando no hay más entrada, un valor que no se puede confundir con ningún carácter real. Este valor se denomina `EOF`, por "fin del archivo". Debemos declarar `que c` es un tipo lo suficientemente grande como para contener cualquier valor que `devuelva getchar`. No podemos usar `char` ya que `c` debe ser lo suficientemente grande como para contener `EOF` además de cualquier posible `char`. Por lo tanto, usamos `int`.

`EOF` es un número entero definido en <stdio.h>, pero el valor numérico específico no importa siempre que no sea el mismo que cualquier `valor char`. Al usar la constante simbólica, nos aseguramos de que nada en el programa depende del valor numérico específico.

El programa para copiar sería escrito de manera más concisa por programadores experimentados en C. En C, cualquier asignación, como

```
c = getchar();
```

es una expresión y tiene un valor, que es el valor del lado izquierdo después de la asignación. Esto significa que una asignación puede aparecer como parte de una expresión más grande. Si la asignación de un carácter a `c` se coloca dentro de la parte de prueba de un `bucle while`, el programa de copia se puede escribir de esta manera:

```
#include <stdio.h>

/* copiar la entrada a la salida; 2ª
versión */ main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

El `while` obtiene un carácter, lo asigna a `c` y, a continuación, comprueba si el carácter era la señal de fin de archivo. Si no lo fue, se ejecuta el cuerpo del `while`, imprimiendo el carácter. A continuación, el while se repite. Cuando finalmente se alcanza el final de la entrada, while termina y también lo hace `main`.

Esta versión centraliza la entrada - ahora solo hay una referencia a `getchar` - y reduce el programa. El programa resultante es más compacto y, una vez que se domina el idioma, es más fácil de leer. Verás este estilo a menudo. (Sin embargo, es posible dejarse llevar y crear código impenetrable, una tendencia que intentaremos frenar).

Los paréntesis alrededor de la asignación, dentro de la condición, son necesarios. La *precedencia* de `!=` es mayor que la de `=`, lo que significa que en ausencia de paréntesis, la prueba relacional `!=` se realizaría antes de la asignación `=`. Por lo tanto, la declaración

```
c = getchar() != EOF
```

es equivalente a

```
c = (getchar() != EOF)
```

Esto tiene el efecto no deseado de establecer `c` en 0 o 1, dependiendo de si la llamada de `getchar` devolvió el final del archivo. (Más sobre esto en Capítulo 2.)

Ejercicio 1-6. Verifique que la expresión `getchar() != EOF` sea 0 o 1.

Ejercicio 1-7. Escriba un programa para imprimir el valor de `EOF`.

1.5.2 Conteo de caracteres

El siguiente programa cuenta los caracteres; Es similar al programa de copia.

```
#include <stdio.h>

/* contar caracteres en la entrada; 1ª versión
*/ main()
{
    nc largo;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

La declaración

```
++nc;
```

presenta un nuevo operador, `++`, que significa *incremento en uno*. En su lugar, podría escribir `nc = nc + 1` pero `++nc` es más conciso y, a menudo, más eficiente. Hay un operador correspondiente: para disminuir en 1. Los operadores `++` y `--` pueden ser operadores de prefijo (`++nc`) u operadores de sufijo (`nc++);` estas dos formas tienen valores diferentes en las expresiones, como se mostrará en el Capítulo 2, pero `++nc` y `nc++` ambos incrementan `nc`. Por el momento nos ceñiremos a la forma del prefijo.

El programa de conteo de caracteres acumula su conteo en una `variable long` en lugar de un int. `Los` enteros largos son de al menos 32 bits. Aunque en algunas máquinas, `int` y `long` tienen el mismo tamaño, en otras `un int` es de 16 bits, con un valor máximo de 32767, y se necesitaría relativamente poca entrada para desbordar un contador int. La especificación de conversión `%ld` le dice a `printf` que el argumento correspondiente es un `entero largo`.

Puede ser posible hacer frente a números aún más grandes mediante el uso de un `doble` (flotador de `doble precisión`). También usaremos una `declaración for` en lugar de un `while`, para ilustrar otra forma de escribir el bucle.

```
 #include <stdio.h>

/* contar caracteres en la entrada; 2ª versión
*/ main()
{
    doble NC;

    for (nc = 0; gechar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

`printf` usa `%f` tanto para float como para `double`; `%.0f` suprime la impresión del punto decimal y la parte de la fracción, que es cero.

El cuerpo de este `bucle for` está vacío, porque todo el trabajo se realiza en las partes de prueba e incremento. Pero las reglas gramaticales de C requieren que un enunciado for tenga un cuerpo. El punto y coma aislado, denominado instrucción *nula*, está ahí para satisfacer ese requisito. Lo colocamos en una línea separada para que sea visible.

Antes de abandonar el programa de conteo de caracteres, observe que si la entrada no contiene caracteres, la `prueba while` o `for` falla en la primera llamada a `getchar`, y el programa produce cero, la respuesta correcta. Esto es importante. Una de las cosas buenas de `while` and `for` es que prueban en la parte superior del bucle, antes de continuar con el cuerpo. Si no hay nada que hacer, no se hace nada, incluso si eso significa nunca pasar por el cuerpo del bucle. Los programas deben actuar de manera inteligente cuando se les da una entrada de longitud cero. Las `instrucciones while` y `for` ayudan a garantizar que los programas hagan cosas razonables con condiciones de límite.

1.5.3 Conteo de líneas

El siguiente programa cuenta las líneas de entrada. Como mencionamos anteriormente, la biblioteca estándar garantiza que un flujo de texto de entrada aparezca como una secuencia de líneas, cada una terminada por una nueva línea. Por lo tanto, contar líneas es solo contar nuevas líneas:

```
#include <stdio.h>

/* contar líneas en la
entrada */ main()
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

El cuerpo del `while` ahora consta de un `if`, que a su vez controla el incremento `++nl`. La instrucción `if` prueba la condición entre paréntesis y, si la condición es verdadera, ejecuta la instrucción (o grupo de instrucciones entre llaves) que sigue. Hemos vuelto a sangrar para mostrar qué es controlado por qué.

El doble signo igual `==` es la notación C para "es igual a" (como el simple = de Pascal o el .EQ `de Fortran`.). Este símbolo se utiliza para distinguir la prueba de igualdad del = simple que C utiliza para la asignación. Una advertencia: los recién llegados a C ocasionalmente escriben `=` cuando quieren decir `==`. Como veremos en el capítulo 2, el resultado suele ser una expresión legal, por lo que no recibirás ninguna advertencia.

Un carácter escrito entre comillas simples representa un valor entero igual al valor numérico del carácter en el juego de caracteres de la máquina. A esto se le llama constante de *caracteres*, aunque es solo otra forma de escribir un número entero pequeño. Así, por ejemplo, `'A'` es una constante de caracteres; en el conjunto de caracteres ASCII su valor es 65, la representación interna del carácter
`Una`. Por supuesto, es preferible `la 'A'` a la `65`: su significado es obvio y es independiente de un conjunto de caracteres en particular.

Las secuencias de escape utilizadas en las constantes de cadena también son legales en las constantes de caracteres, por lo que `'\n'` representa el valor del carácter de nueva línea, que es 10 en ASCII. Debes tener en cuenta que `'\n'` es un solo carácter, y en las expresiones es solo un número entero; por otro lado, `'\n'` es una constante de cadena que contiene solo un carácter. El tema de las cadenas frente a los caracteres se discute con más detalle en el Capítulo 2.

Ejercicio 1-8. Escriba un programa para contar espacios en blanco, tabulaciones y saltos de línea.

Ejercicio 1-9. Escriba un programa para copiar su entrada a su salida, reemplazando cada cadena de uno o más espacios en blanco por un solo espacio en blanco.

Ejercicio 1-10. Escriba un programa para copiar su entrada a su salida, reemplazando cada tabulación por `\t`, cada retroceso por `\b` y cada barra invertida por `\\`. Esto hace que las pestañas y los retrocesos sean visibles de forma inequívoca.

1.5.4 Conteo de palabras

El cuarto de nuestra serie de programas útiles cuenta líneas, palabras y caracteres, con la definición vaga de que una palabra es cualquier secuencia de caracteres que no contiene un espacio en blanco, una tabulación o una nueva línea. Esta es una versión básica del programa wc de UNIX.

```
#include <stdio.h>

#define IN1 /* dentro de una palabra
*/ #define OUT 0 /* fuera de una
palabra */

/* contar líneas, palabras y caracteres en la
entrada */ main()
{
    INT C, NL, NW, NC, ESTADO;

    estado = FUERA;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' | c == '\n' || c = '\t')
            estado = FUERA;
        else if (estado == FUERA)
            { estado = DENTRO;
            ++NO;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Cada vez que el programa encuentra el primer carácter de una palabra, cuenta una palabra más. La variable `state` registra si el programa está actualmente en una palabra o no; inicialmente está "not in a word", a lo que se le asigna el valor `OUT`. Preferimos las constantes simbólicas `IN` y `OUT` a los valores literales 1 y 0 porque hacen que el programa sea más legible. En un programa tan pequeño como este, hace poca diferencia, pero en programas más grandes, el aumento de la claridad bien vale la pena el modesto esfuerzo adicional de escribirlo de esta manera desde el principio. También descubrirá que es más fácil hacer cambios extensos en programas donde los números mágicos aparecen solo como constantes simbólicas.

La línea

```
nl = nw = nc = 0;
```

Establece las tres variables en cero. No se trata de un caso especial, sino de una consecuencia del hecho de que una asignación es una expresión con el valor y las asignaciones asociadas de derecha a izquierda. Es como si hubiéramos escrito

```
nl = (nw = (nc = 0));
```

El operador `||` significa O, por lo que la línea

```
if (c == ' ' || c == '\n' || c = '\t')
```

dice "si `c` es un espacio en blanco *o* `c` es una nueva línea *o* `c` es una tabulación". (Recordemos que la secuencia de escape `\t` es una representación visible del carácter de tabulación). Hay un operador correspondiente `&&` para AND; su precedencia es justo mayor que `||`. Expresiones conectadas por `&&` o `||` se evalúan de izquierda a derecha, y se garantiza que la evaluación se detendrá tan pronto como se conozca la verdad o la falsedad. Si `c` es un espacio en blanco, no es necesario probar si es una nueva línea o una tabulación, por lo que no se realizan estas pruebas. Esto no es particularmente importante aquí, pero es significativo en situaciones más complicadas, como pronto veremos.

En el ejemplo también se muestra else, que especifica una acción alternativa si la parte de la condición de una `instrucción if` es false. La forma general es

```
if (expresión)
    Afirmación1
más
    Declaración2
```

Se realiza una y solo una de las dos instrucciones asociadas con un `if-else`. Si la *expresión* es verdadera, *se ejecuta la declaración1*; si no, se ejecuta *la declaración2*. Cada *enunciado* puede ser un solo enunciado o varios entre llaves. En el programa de conteo de palabras, el que sigue al `else` es un `if` que controla dos sentencias entre llaves.

Ejercicio 1-11. ¿Cómo probarías el programa de conteo de palabras? ¿Qué tipos de entrada tienen más probabilidades de descubrir errores, si los hay?

Ejercicio 1-12. Escriba un programa que imprima su entrada una palabra por línea.

1.6 Matrices

Vamos a escribir un programa para contar el número de ocurrencias de cada dígito, de los caracteres de espacio en blanco (en blanco, tabulación, nueva línea) y de todos los demás caracteres. Esto es artificial, pero nos permite ilustrar varios aspectos de C en un mismo programa.

Hay doce categorías de entrada, por lo que es conveniente usar una matriz para contener el número de ocurrencias de cada dígito, en lugar de diez variables individuales. Esta es una versión del programa:

```
#include <stdio.h>

/* contar dígitos, espacios en blanco,
otros */ main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nblanco = otro = 0;
    para (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndígito[c-'0'];
        else if (c == ' ' | c == '\n' || c == '\t')
            ++nblanco;
        más
            ++notro;

    printf("dígitos =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", espacio en blanco = %d, otro =
        %d\n", nblanco, otro);
}
```

La salida de este programa en sí mismo es

```
dígitos = 9 3 0 0 0 0 0 0 0 0 1, espacio en blanco = 123, otros = 345
```

La declaración

```
int ndigit[10];
```

Declara que `ndigit` es una matriz de 10 enteros. Los subíndices de la matriz siempre comienzan en cero en C, por lo que los elementos son `ndigit[0], ndigit[1], ..., ndigit[9]`. Esto se refleja en los bucles for que inicializan e imprimen la matriz.

Un subíndice puede ser cualquier expresión entera, que incluye variables enteras como `i` y constantes enteras.

Este programa en particular se basa en las propiedades de la representación de caracteres de los dígitos.
Por ejemplo, la prueba

```
si (c >= '0' && c <= '9')
```

Determina si el carácter de C es un dígito. Si es así, el valor numérico de ese dígito es

```
c - '0'
```

Esto solo funciona si `'0', '1', ..., '9'` tienen valores crecientes consecutivos. Afortunadamente, esto es cierto para todos los conjuntos de caracteres.

Por definición, `los caracteres son solo números enteros pequeños, por lo que` las variables y constantes char son idénticas a `los enteros en expresiones aritméticas. Esto es natural y conveniente; Por ejemplo, C-'0'` es una expresión entera con un valor entre 0 y 9 que corresponde al carácter `'0'` a `'9'` almacenado en `C` y, por lo tanto, un subíndice válido para la matriz `Ndigit`.

La decisión de si un carácter es un dígito, un espacio en blanco u otra cosa se toma con la secuencia

```
si (c >= '0' && c <= '9')
    ++ndígito[c-'0'];
else if (c == ' ' | c == '\n' || c == '\t')
    ++nblanco;
más
    ++notro;
```

El patrón

```
if (condición1)
    Afirmación1
else if (condición2)
    Declaración2
    ...
    ...
más
    Declaraciónn
```

Ocurre con frecuencia en los programas como una forma de expresar una decisión multidireccional. Las *condiciones* se evalúan en orden desde la parte superior hasta que se cumple alguna condición; en ese momento se ejecuta la parte de la *declaración* correspondiente y se termina toda la construcción. (Cualquier *puede* ser varias instrucciones encerradas entre llaves). Si no se cumple ninguna de las condiciones, se *ejecuta la instrucción* después de la `else` final si está presente. Si se omiten las `sentencias else` y final de la *sentencia*, como en el programa de conteo de palabras, no se lleva a cabo ninguna acción. Puede haber cualquier número de

`else if`(*condición*)
declaración

se agrupa entre el `if` inicial y el `else final`.

Por una cuestión de estilo, es aconsejable formatear esta construcción como hemos mostrado; si cada una de ellas estuviera sangrada más allá de la `anterior`, una larga secuencia de decisiones marcharía por el lado derecho de la página.

La instrucción `switch`, que se discutirá en el Capítulo 4, proporciona otra forma de escribir una rama multidireccional que es particularmente adecuada cuando la condición es si algún número entero o expresión de carácter coincide con uno de un conjunto de constantes. Para contrastar, presentaremos una versión switch de este programa en la Sección 3.4.

Ejercicio 1-13. Escriba un programa para imprimir un histograma de las longitudes de las palabras en su entrada. Es fácil dibujar el histograma con las barras horizontales; Una orientación vertical es más desafiante.

Ejercicio 1-14. Escriba un programa para imprimir un histograma de las frecuencias de diferentes caracteres en su entrada.

1.7 Funciones

En C, una función es equivalente a una subrutina o función en Fortran, o a un procedimiento o función en Pascal. Una función proporciona una forma conveniente de encapsular algunos cálculos, que luego se pueden usar sin preocuparse por su implementación. Con funciones correctamente diseñadas, es posible ignorar *cómo* se hace un trabajo; saber *lo que* se hace es suficiente. C hace que la demanda de funciones sea fácil, conveniente y eficiente; A menudo verá una función corta definida y llamada solo una vez, solo porque aclara algún fragmento de código.

Hasta ahora solo hemos usado funciones como `printf`, `getchar` y `putchar` que se nos han proporcionado; ahora es el momento de escribir algunas de las nuestras. Dado que C no tiene un operador de exponenciación como el `**` de Fortran, ilustremos la mecánica de la definición de funciones escribiendo una función `power(m,n)` para elevar un entero `m` a una potencia entera positiva `n`. Es decir, el valor de la `potencia(2,5)` es 32. Esta función no es una rutina práctica de exponenciación, ya que solo maneja potencias positivas de números enteros pequeños, pero es lo suficientemente buena para la ilustración. (La biblioteca estándar contiene una función `pow(x,y)` que calcula *xy*.)

Aquí está la potencia de la función y un programa principal para ejercerla, para que pueda ver toda la estructura a la vez.

```
#include <stdio.h>

int potencia (int m, int n);

 /* función de potencia de
 prueba */ main()
 {
     Int I;

     para (i = 0; i < 10; ++i)
         printf("%d %d %d\n", i, potencia(2,i), potencia(-
     3,i)); devuelve 0;
 }

 /* potencia: eleva la base a la n-ésima
 potencia; n >= 0 */ int potencia(int base, int
 n)
 {
     int i, p;

     p = 1;
     para (i = 1; i <= n;
         ++i) p = p * base;
     retorno p;
 }
```

Una definición de función tiene esta forma:

```
return-type nombre-de-función(declaraciones de parámetros, si las hay)
{
   Declaraciones
   Declaraciones
}
```

Las definiciones de funciones pueden aparecer en cualquier orden, y en un archivo de origen o en varios, aunque ninguna función se puede dividir entre archivos. Si el programa fuente aparece en varios archivos, es posible que tenga que decir más para compilarlo y cargarlo que si aparece todo en uno, pero eso es una cuestión del sistema operativo, no un atributo del lenguaje. Por el momento, asumiremos que ambas funciones están en el mismo archivo, por lo que lo que haya aprendido sobre la ejecución de programas C seguirá funcionando.

La potencia de la función se llama dos veces por `la red principal`, en la línea

```
   printf("%d %d %d\n", i, potencia(2,i), potencia(-3,i));
```

Cada llamada pasa dos argumentos a `power`, que cada vez devuelve un número entero para formatearlo e imprimirlo. En una expresión, `power(2,i)` es un número entero al igual que `2` e `i` . (No todas las funciones producen un valor entero; vamos a tomar esto en Capítulo 4.)

La primera línea de `poder` en sí misma,

```
int power(int base, int n)
```

Declara los tipos y nombres de parámetros, así como el tipo del resultado que devuelve la función. Los nombres utilizados por `power` para sus parámetros son locales para `power`, y no son visibles para ninguna otra función: otras rutinas pueden usar los mismos nombres sin conflicto. Lo mismo ocurre con las variables `i` y `p`: la `i` en `potencia` no está relacionada con la `i` en `main`.

Por lo general, usaremos *el parámetro* para una variable nombrada en la lista entre paréntesis de una función. Los términos *argumento formal* y *argumento real* se utilizan a veces para la misma distinción.

La instrucción return: devuelve al `valor que` power calcula a `main`. Cualquier expresión puede seguir `a return`:

```
expresión de retorno;
```

No es necesario que una función devuelva un valor; Una instrucción return sin expresión hace que el control, pero no el valor útil, se devuelva al llamador, al igual que "falling off the end" de una función al alcanzar la llave derecha de terminación. Y la función de llamada puede omitir un valor devuelto por una función.

Es posible que haya notado que hay una declaración return al final de `main`. Dado que `main` es una función como cualquier otra, puede devolver un valor a su llamador, que es en efecto el entorno en el que se ejecutó el programa. Normalmente, un valor devuelto de cero implica una terminación normal; Los valores distintos de cero indican condiciones de terminación inusuales o erróneas. En aras de la simplicidad, hemos omitido las declaraciones return de nuestras `funciones principales` hasta este punto, pero las incluiremos más adelante, como un recordatorio de que los programas deben devolver el estado a su entorno.

La declaración

```
int power(int base, int n);
```

Justo antes `de que main` diga que `power` es una función que espera dos argumentos `int` y devuelve un `int`. Esta declaración, que se denomina *prototipo* de función, tiene que concordar con la definición y los usos del `poder`. Es un error si la definición de una función o cualquier uso de la misma no concuerda con su prototipo.

No es necesario que los nombres de los parámetros coincidan. De hecho, los nombres de los parámetros son opcionales en un prototipo de función, por lo que para el prototipo podríamos haber escrito

```
int poder(int, int);
```

Sin embargo, los nombres bien elegidos son una buena documentación, por lo que a menudo los usaremos.

Una nota de historia: el mayor cambio entre ANSI C y las versiones anteriores es cómo se declaran y definen las funciones. En la definición original de C, la función de `potencia` se habría escrito así:

```
/* potencia: eleva la base a la n-ésima potencia; n >= 0 */
/*         (versión antigua) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    para (i = 1; i <= n;
        ++i) p = p * base;
    retorno p;
}
```

Los parámetros se nombran entre paréntesis, y sus tipos se declaran antes de abrir la llave izquierda; los parámetros no declarados se toman como `int`. (El cuerpo de la función es el mismo que antes).

La declaración de `poder` al comienzo del programa habría sido la siguiente:

```
int poder();
```

No se permitía ninguna lista de parámetros, por lo que el compilador no podía comprobar fácilmente que se llamaba correctamente a power. De hecho, dado que de forma predeterminada se habría supuesto que power devolvería un `int`, bien podría haberse omitido toda la declaración.

La nueva sintaxis de los prototipos de funciones hace que sea mucho más fácil para un compilador detectar errores en el número de argumentos o sus tipos. El estilo antiguo de declaración y definición sigue funcionando en ANSI C, al menos durante un período de transición, pero se recomienda encarecidamente usar la nueva forma cuando tenga un compilador que la admita.

Ejercicio 1.15. Reescriba el programa de conversión de temperatura de la Sección 1.2 para usar una función para la conversión.

1.8 Argumentos: llamada por valor

Un aspecto de las funciones de C puede ser desconocido para los programadores que están acostumbrados a otros lenguajes, particularmente Fortran. En C, todos los argumentos de la función se pasan "por valor". Esto significa que a la función llamada se le dan los valores de sus argumentos en variables temporales en lugar de los originales. Esto conduce a algunas propiedades diferentes a las que se ven con lenguajes de "llamada por referencia" como Fortran o con `parámetros var` en Pascal, en los que la rutina llamada tiene acceso al argumento original, no a una copia local.

Sin embargo, la opción de compra por valor es un activo, no un pasivo. Por lo general, conduce a programas más compactos con menos variables extrañas, porque los parámetros se pueden tratar como variables locales convenientemente inicializadas en la rutina llamada. Por ejemplo, aquí hay una versión de `power` que hace uso de esta propiedad.

```
/* potencia: eleva la base a la n-ésima potencia; n >= 0;
Versión 2 */ int power(int base, int n)
{
    int p;

    para (p = 1; n > 0; --
        n) p = p * base;
    retorno p;
}
```

El parámetro `n` se utiliza como una variable temporal, y se cuenta hacia atrás (un `bucle for` que se ejecuta hacia atrás) hasta que se convierte en cero; ya no es necesaria la variable `i`. Lo que sea que se haga a `n` dentro `del poder` no tiene ningún efecto sobre el argumento de que el `poder` fue originalmente llamado.

Cuando sea necesario, es posible disponer que una función modifique una variable en una rutina de llamada. El autor de la llamada debe proporcionar la *dirección* de la variable que se va a establecer (técnicamente, un *puntero* a la variable), y la función llamada debe declarar que el parámetro es un puntero y acceder a la variable indirectamente a través de él. Cubriremos los consejos en el Capítulo 5.

La historia es diferente para las matrices. Cuando se usa el nombre de una matriz como argumento, el valor pasado a la función es la ubicación o la dirección del comienzo de la matriz: no hay copia de los elementos de la matriz. Al subescribir este valor, la función puede acceder y modificar cualquier argumento de la matriz. Este es el tema de la siguiente sección.

1.9 Matrices de caracteres

El tipo más común de matriz en C es la matriz de caracteres. Para ilustrar el uso de matrices de caracteres y funciones para manipularlas, escribamos un programa que lea un conjunto de líneas de texto e imprima el más largo. El esquema es bastante simple:

```
while (hay otra línea)
    if (es más largo que el anterior más largo)
        (guárdalo)
        (guarde su
longitud) imprima la línea
más larga
```

Este esquema deja claro que el programa se divide naturalmente en partes. Una pieza obtiene un nueva línea, otra la guarda y el resto controla el proceso.

Dado que las cosas se dividen tan bien, sería bueno escribirlas de esa manera también. En consecuencia, primero escribamos una función `getline separada` para obtener la siguiente línea de entrada. Intentaremos que la función sea útil en otros contextos. Como mínimo, `getline` tiene que devolver una señal sobre el posible final del archivo; un diseño más útil sería devolver la longitud de la línea, o cero si se encuentra el final del archivo. Cero es una devolución aceptable de fin de archivo porque nunca es una longitud de línea válida. Cada línea de texto tiene al menos un carácter; Incluso una línea que contiene solo una nueva línea tiene una longitud 1.

Cuando encontramos una línea que es más larga que la línea más larga anterior, debemos guardarla en algún lugar. Esto sugiere una segunda función, `copiar`, para copiar la nueva línea en un lugar seguro.

Por último, necesitamos un programa principal para controlar `getline` y `copy`. Aquí está el resultado.

```
#include <stdio.h>
#define MAXLINE 1000   /* longitud máxima de la línea de entrada */

int getline(char línea[], int maxline);
void copy(char to[], char from[]);

/* imprime la línea de entrada más
larga */ main()
{
    int len;            /* longitud de línea actual */
    int max;/* longitud máxima vista hasta ahora  */
    línea de caracteres[MAXLINE];/* línea de entrada
    actual */
    char más largo[MAXLINE]; /* línea más larga guardada aquí */

    máx. = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            máx = len; copia (línea
            más larga);
        }
    if (max > 0) /* había una línea */
        printf("%s", la más larga);
    devuelve 0;
}

/* getline: lee una línea en s, devuelve length */
int getline(char s[],int lim)
{
    int c, i;

    para (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    Retorno I;
}

/* copy: copia 'de' en 'hacia'; assume to es lo
suficientemente grande */ void copy(char to[], char from[])
{
    Int I;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Las funciones `getline` y `copy` se declaran al principio del programa, que suponemos que está contenido en un archivo.

`main` y `getline` se comunican a través de un par de argumentos y un valor devuelto. En `getline`, los argumentos se declaran mediante la línea

```
tú getline(char s[], eres lim);
```

que especifica que el primer argumento, `s`, es una matriz, y el segundo, `lim`, es un número entero. El propósito de proporcionar el tamaño de una matriz en una declaración es reservar almacenamiento. La longitud de una matriz `s` no es necesaria en `getline` ya que su tamaño se establece en `main`. `getline` utiliza return to

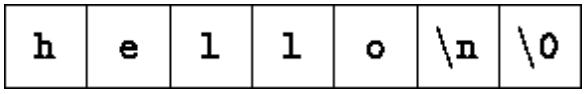
Envíe un valor de vuelta al autor de la llamada, tal como lo hizo la función `power` . Esta línea también declara que
`getline` devuelve un `int`; dado que `int` es el tipo de retorno predeterminado, podría omitirse.

Algunas funciones devuelven un valor útil; otras, como `copiar`, se usan solo por su efecto y no devuelven ningún valor. El tipo de devolución de `copy` es `void`, que indica explícitamente que no se devuelve ningún valor.

`getline` coloca el carácter `'\0'` (el *carácter nulo*, cuyo valor es cero) al final de la matriz que está creando, para marcar el final de la cadena de caracteres. Esta conversión también es utilizada por el lenguaje C: cuando una constante de cadena como

```
"Hola\n"
```

aparece en un programa C, se almacena como una matriz de caracteres que contiene los caracteres de la cadena y termina con un `'\0'` para marcar el final.



La `especificación de formato` %s en `printf` espera que el argumento correspondiente sea una cadena representada de esta forma. `copy` también se basa en el hecho de que su argumento de entrada termina con `'\0'` y copia este carácter en la salida.

Vale la pena mencionar de paso que incluso un programa tan pequeño como este presenta algunos problemas de diseño pegajosos. Por ejemplo, ¿qué debe `hacer main` si encuentra una línea que es más grande que su límite? `getline` funciona de forma segura, en el sentido de que deja de recopilar cuando la matriz está llena, incluso si no se ha visto ninguna nueva línea. Al probar la longitud y el último carácter devuelto, `main` puede determinar si la línea era demasiado larga y luego arreglárselas como desee. En aras de la brevedad, hemos hecho caso omiso de esta cuestión.

No hay forma de que un usuario de `getline` sepa de antemano qué tan larga puede ser una línea de entrada, por lo que `getline` comprueba si hay desbordamiento. Por otro lado, el usuario de copy ya sabe (o puede averiguar) el tamaño de las cadenas, por lo que hemos optado por no añadirle la comprobación de errores.

Ejercicio 1-16. Revise la rutina principal del programa de líneas más largas para que imprima correctamente la longitud de las líneas de entrada largas arbitrarias y, en la medida de lo posible, el texto.

Ejercicio 1-17. Escriba un programa para imprimir todas las líneas de entrada que tengan más de 80 caracteres.

Ejercicio 1-18. Escriba un programa para eliminar los espacios en blanco y las tabulaciones finales de cada línea de entrada, y para eliminar completamente las líneas en blanco.

Ejercicio 1-19. Escriba una función `inversa(s)` que invierta la cadena de caracteres `s`. Úselo para escribir un programa que invierta su entrada una línea a la vez.

1.10 Variables externas y alcance

Las variables de `main`, como `line, longest,` etc., son privadas o locales de `main`. Debido a que se declaran dentro de `main`, ninguna otra función puede tener acceso directo a ellos. Lo mismo ocurre con las variables de otras funciones; Por ejemplo, la variable `i` en `getline` no

está relacionada con la `i`

en copia. Cada variable local de una función comienza a existir solo cuando se llama a la función y desaparece cuando se sale de la función. Esta es la razón por la que tales variables generalmente se conocen como variables automáticas, siguiendo la terminología de otros idiomas. De ahora en adelante usaremos el término automático para referirnos a estas variables locales. (En el capítulo 4 se describe la clase de almacenamiento estático, en la que las variables locales conservan sus valores entre llamadas).

Dado que las variables automáticas van y vienen con la invocación de funciones, no conservan sus valores de una llamada a la siguiente y deben establecerse explícitamente en cada entrada. Si no están configurados, contendrán basura.

Como alternativa a las variables automáticas, es posible definir variables que son externas a todas las funciones, es decir, variables a las que se puede acceder por nombre desde cualquier función. (Este mecanismo es bastante parecido a las variables COMMON o Pascal de Fortran declaradas en el bloque más externo). Dado que las variables externas son accesibles globalmente, se pueden usar en lugar de listas de argumentos para comunicar datos entre funciones. Además, dado que las variables externas permanecen en existencia permanentemente, en lugar de aparecer y desaparecer a medida que se llaman y se cierran funciones, conservan sus valores incluso después de que las funciones que las establecieron hayan regresado.

Una variable externa debe ser *definida*, exactamente una vez, fuera de cualquier función, lo que le reserva el almacenamiento. La variable también debe ser *declarada* en cada función que quiera acceder a ella; esto indica el tipo de la variable. La declaración puede ser una instrucción `externa` explícita o puede estar implícita en el contexto. Para concretar la discusión, reescribamos el programa de línea más larga con `line`, `longest` y `max` como variables externas. Esto requiere cambiar las llamadas, las declaraciones y los cuerpos de las tres funciones.

```
#include <stdio.h>

#define MAXLINE 1000    /* tamaño máximo de línea de entrada */

int max;/* longitud máxima vista hasta ahora      */
línea de caracteres[MAXLINE];/* línea de entrada actual
*/
char más largo[MAXLINE]; /* línea más larga guardada aquí */

int
getline(vacío);
copia nula(nula);

/* imprime la línea de entrada más larga; Versión
especializada */ main()
{
    int len;
    extern int max;
    extern char más largo[];

    máx. = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            máx = len;
            copiar();
        }
    if (max > 0) /* había una línea */
        printf("%s", la más larga);
    devuelve 0;
}
```

```
/* getline: versión especializada */
int getline(void)
{
    int c, i;
    línea de caracteres externa[];

    para (i = 0; i < MAXLINE - 1
         && (c=getchar)) != EOF && c != '\n'; ++i)
             línea[i] = c;
    if (c == '\n') {
        línea[i] = c;
        ++i;
    }
    línea[i] = '\0';
    Retorno I;
}

/* copia: versión especializada */
copia nula (nula)
{
    Int I;
    línea de caracteres externa[], más larga[];

    i = 0;
    while ((más largo[i] = línea[i]) != '\0')
        ++i;
}
```

Las variables externas en `main`, `getline` y `copy` se definen mediante las primeras líneas del ejemplo anterior, que indican su tipo y hacen que se les asigne almacenamiento. Desde el punto de vista sintáctico, las definiciones externas son como las definiciones de variables locales, pero como se producen fuera de las funciones, las variables son externas. Antes de que una función pueda usar una variable externa, el nombre de la variable debe ser dado a conocer a la función; La declaración es la misma que antes, excepto por la palabra clave extern agregada.

En determinadas circunstancias, se puede omitir la declaración externa. Si la definición de la variable externa se produce en el archivo de código fuente antes de su uso en una función determinada, entonces no hay necesidad de una `declaración extern` en la función. Por lo tanto, `las declaraciones` extern en `main`, `getline` y `copy` son redundantes. De hecho, la práctica común es colocar las definiciones de todas las variables externas al principio del archivo de código fuente y, a continuación, omitir todas las declaraciones externas.

Si el programa está en varios archivos fuente, y una variable está definida en *file1* y se usa en *file2* y *file3*, entonces `se necesitan declaraciones` extern en *file2* y *file3* para conectar las ocurrencias de la variable. La práctica habitual es recopilar `declaraciones externas de variables y funciones en un archivo separado, históricamente llamado` encabezado, *que se incluye mediante* #include en la parte delantera de cada archivo de código fuente. El sufijo `.h` es convencional para los nombres de encabezado. Las funciones de la biblioteca estándar, por ejemplo, se declaran en encabezados como `<stdio.h>`. Este tema se discute en detalle en el Capítulo 4, y la biblioteca misma en el Capítulo 7 y el Apéndice B.

Dado que las versiones especializadas de `getline` y `copy` no tienen argumentos, la lógica sugeriría que sus prototipos al principio del archivo deberían ser `getline()` y `copy()`. Pero para la compatibilidad con programas C más antiguos, el estándar toma una lista vacía como una declaración de estilo antiguo y desactiva toda la verificación de la lista de argumentos; La palabra `void` debe usarse para una lista explícitamente vacía. Discutiremos esto más a fondo en el Capítulo 4.

Debe tener en cuenta que estamos utilizando las palabras *definición* y *declaración* con

cuidado cuando nos referimos a variables externas en esta sección. "Definición" se refiere al lugar donde se encuentra la variable

almacenamiento creado o asignado; "declaración" se refiere a los lugares donde se indica la naturaleza de la variable pero no se asigna almacenamiento.

Por cierto, hay una tendencia a hacer que todo lo que está a la vista sea una `variable externa` porque parece simplificar las comunicaciones: las listas de argumentos son cortas y las variables siempre están ahí cuando las desea. Pero las variables externas siempre están ahí, incluso cuando no las quieres. Depender demasiado de variables externas está plagado de peligros, ya que conduce a programas cuyas conexiones de datos no son todas obvias: las variables se pueden cambiar de manera inesperada e incluso inadvertida, y el programa es difícil de modificar. La segunda versión del programa de la línea más larga es inferior a la primera, en parte por estas razones, y en parte porque destruye la generalidad de dos funciones útiles al escribir en ellas los nombres de las variables que manipulan.

En este punto hemos cubierto lo que podría llamarse el núcleo convencional de C. Con este puñado de bloques de construcción, es posible escribir programas útiles de tamaño considerable, y probablemente sería una buena idea si hiciera una pausa lo suficiente para hacerlo. Estos ejercicios sugieren programas de cierta complejidad que los anteriores en este capítulo.

Ejercicio 1-20. Escriba una `tabulación de programa` que reemplace las tabulaciones de la entrada con el número adecuado de espacios en blanco para espaciar hasta la siguiente tabulación. Supongamos un conjunto fijo de tabulaciones, digamos cada *n* columnas. ¿Debería *n* ser una variable o un parámetro simbólico?

Ejercicio 1-21. Escriba una `pestaña` de programa que reemplace las cadenas de espacios en blanco por el número mínimo de tabulaciones y espacios en blanco para lograr el mismo espaciado. Utilice las mismas tabulaciones que para `detab`. ¿Cuándo una tabulación o un solo espacio en blanco serían suficientes para llegar a una tabulación, a cuál se debe dar preferencia?

Ejercicio 1-22. Escriba un programa para "doblar" las líneas largas de entrada en dos o más líneas más cortas después del último carácter no en blanco que aparece antes de la n-ésima columna de entrada. Asegúrese de que su programa haga algo inteligente con líneas muy largas, y si no hay espacios en blanco o tabulaciones antes de la columna especificada.

Ejercicio 1-23. Escriba un programa para eliminar todos los comentarios de un programa C. No olvide manejar correctamente las cadenas entrecomilladas y las constantes de caracteres. Los comentarios de C no se anidan.

Ejercicio 1-24. Escriba un programa para comprobar si un programa C tiene errores de sintaxis rudimentarios, como paréntesis, corchetes y llaves no coincidentes. No te olvides de las citas, tanto simples como dobles, las secuencias de escape y los comentarios. (Este programa es difícil si lo haces en total generalidad).

Capítulo 2 - Tipos, operadores y expresiones

Las variables y constantes son los objetos de datos básicos manipulados en un programa. Las declaraciones enumeran las variables que se van a usar e indican qué tipo tienen y, quizás, cuáles son sus valores iniciales. Los operadores especifican lo que se les va a hacer. Las expresiones combinan variables y constantes para producir nuevos valores. El tipo de un objeto determina el conjunto de valores que puede tener y las operaciones que se pueden realizar en él. Estos bloques de construcción son los temas de este capítulo.

El estándar ANSI ha realizado muchos pequeños cambios y adiciones a los tipos y expresiones básicos. Ahora hay `formas con y sin signo` de todos los tipos enteros, y notaciones para constantes sin signo y constantes de caracteres hexadecimales. Las operaciones de coma flotante se pueden realizar con precisión simple; También hay un tipo doble largo para una mayor precisión. Las constantes de cadena se pueden concatenar en tiempo de compilación. Las enumeraciones se han convertido en parte del lenguaje, formalizando una característica de larga data. Los objetos se pueden declarar `const`, lo que impide que se cambien. Las reglas para las coerciones automáticas entre tipos aritméticos se han aumentado para manejar el conjunto más rico de tipos.

2.1 Nombres de variables

Aunque no lo dijimos en el Capítulo 1, existen algunas restricciones sobre los nombres de las variables y las constantes simbólicas. Los nombres se componen de letras y números; El primer carácter debe ser una letra. El guión bajo "_" cuenta como una letra; a veces es útil para mejorar la legibilidad de los nombres largos de las variables. Sin embargo, no comience los nombres de las variables con guiones bajos, ya que las rutinas de biblioteca a menudo usan tales nombres. Las letras mayúsculas y minúsculas son distintas, por lo que `x` y `X` son dos nombres diferentes. La práctica tradicional de C es usar minúsculas para los nombres de las variables y todas las mayúsculas para las constantes simbólicas.

Al menos los primeros 31 caracteres de un nombre interno son significativos. En el caso de los nombres de función y las variables externas, el número puede ser menor que 31, ya que los nombres externos pueden ser utilizados por ensambladores y cargadores sobre los que el lenguaje no tiene control. Para los nombres externos, el estándar garantiza la unicidad solo para 6 caracteres y un solo caso. Las palabras clave como `if`, `else`, `int`, `float`, etc., están reservadas: no se pueden usar como nombres de variables. Deben estar en minúsculas.

Es aconsejable elegir nombres de variables que estén relacionados con el propósito de la variable y que sea poco probable que se mezclen tipográficamente. Tendemos a usar nombres cortos para las variables locales, especialmente los índices de bucle, y nombres más largos para las variables externas.

2.2 Tipos y tamaños de datos

Solo hay unos pocos tipos de datos básicos en C:

`carbonizar` Un solo byte, capaz de contener un carácter en el juego de caracteres local

`Int` Un número entero, que normalmente refleja el tamaño natural de los números enteros en la máquina host

`float` de precisión simple de punto flotante

`Doble` coma flotante de doble precisión

Además, hay una serie de calificadores que se pueden aplicar a estos tipos básicos. `corto` y `long` se aplican a los números enteros:

```
int sh corto;
contador de
enteros largos;
```

La palabra `int` se puede omitir en tales declaraciones, y normalmente lo es.

La intención es que `el corto` y `el largo` proporcionen diferentes longitudes de números enteros cuando sea práctico; `int` normalmente será el tamaño natural de una máquina en particular. `short` suele tener una longitud de 16 bits e `int` de 16 o 32 bits. Cada compilador es libre de elegir los tamaños apropiados para su propio hardware, sujeto solo a la restricción `de que los shorts e ints sean de al menos 16 bits, los longs sean de al menos 32 bits y short` no sea largo que `int`, que no sea más largo que `long`.

El calificador `signed` o `unsigned` se puede aplicar a `char` o a cualquier número entero. `Los números sin signo` son siempre positivos o cero, y obedecen a las leyes de la aritmética módulo *2n*, donde *n* es el número de bits del tipo. Así, por ejemplo, si `los caracteres son de 8 bits, las variables char sin signo` tienen valores entre 0 y 255, mientras que `los caracteres con signo tienen valores entre -128 y 127 (en una máquina de complemento a dos). El hecho de que los caracteres sin formato estén firmados o sin signo depende del equipo,` pero los caracteres imprimibles siempre son positivos.

El tipo `long double` especifica punto flotante de precisión extendida. Al igual que con los enteros, los tamaños de los objetos de punto flotante se definen en función de la implementación; `Float`, `Double` y `Long Double` pueden representar uno, dos o tres tamaños distintos.

Los encabezados estándar `<limits.h>` y `<float.h>` contienen constantes simbólicas para todos estos tamaños, junto con otras propiedades de la máquina y el compilador. Estos se analizan en el Apéndice B.

Ejercicio 2-1. Escriba un programa para determinar los rangos de `las variables char`, `short`, `int` y `long`, tanto `con signo` como `sin signo`, imprimiendo los valores apropiados de los encabezados estándar y mediante cálculo directo. Más difícil si los calcula: determine los rangos de los distintos tipos de punto flotante.

2.3 Constantes

Una constante entera como `1234` es una `int`. Una `constante larga` se escribe con una `l` terminal (ell) o `L`, como en `123456789L`; una constante entera demasiado grande para caber en una `int` también se tomará como larga. Las constantes sin signo se escriben con un terminal `u` o `u`, y el sufijo `ul` o `UL` indica `long sin signo`.

Las constantes de coma flotante contienen un punto decimal (`123.4`) o un exponente (`1e-2`) o ambos; su tipo es `double`, a menos que tenga un sufijo. Los sufijos `f` o `F` indican una constante `de flotación`; `l` o `L` indican un `doble largo`.

El valor de un entero se puede especificar en octal o hexadecimal en lugar de decimal. Un líder `0` (cero) en una constante entera significa octal; un `0x` inicial o `0X` significa hexadecimal. Por ejemplo, el decimal 31 se puede escribir como `037` en octal y `0x1f` o `0x1F` en hexadecimal. Octal y

Las constantes hexadecimales también pueden ir seguidas de `L` para hacerlas `largas` y `U` para hacerlas
`sin signo`: `0XFUL` es una constante *larga sin signo* con un valor de 15 decimales.

Una `constante de caracteres` es un número entero, escrito como un carácter entre comillas simples, como `'x'`. El valor de una constante de caracteres es el valor numérico del carácter en el juego de caracteres de la máquina. Por ejemplo, en el juego de caracteres ASCII, la constante de caracteres `'0'` tiene el valor 48, que no está relacionado con el valor numérico 0. Si escribimos `'0'` en lugar de un valor numérico como 48 que depende del conjunto de caracteres, el programa es independiente del valor particular y más fácil de leer. Las constantes de caracteres participan en las operaciones numéricas al igual que cualquier otro número entero, aunque se utilizan con mayor frecuencia en comparaciones con otros caracteres.

Ciertos caracteres se pueden representar en constantes de caracteres y cadenas mediante secuencias de escape como `\n` (nueva línea); estas secuencias parecen dos caracteres, pero representan solo uno. Además, se puede especificar un patrón de bits arbitrario de tamaño de byte mediante

```
'\ooo'
```

donde *ooo* es de uno a tres dígitos octales (0...7) o por

```
'\xhh'
```

donde *hh* es uno o más dígitos hexadecimales (`0...9, a... f, A... F`). Así que podríamos escribir

```
#define VTAB '\013'/* Tabulación vertical ASCII
*/ #define BELL '\007'/* Carácter de campana
ASCII */
```

o, en hexadecimal,

```
#define VTAB '\xb'/* Tabulación vertical ASCII
*/ #define BELL '\x7'/* Carácter de campana
ASCII */
```

El conjunto completo de secuencias de escape es

`\un`	Carácter de alerta (campana)	`\\`	barra invertida
`\b`	retroceso	`\?`	signo de interrogación
`\f`	Avance de encofrado	`\'`	Cita simple
`\n`	Newline	`\"`	comilla doble
`\r`	Retorno de carro	`\`*ooo*	Número octal
`\t`	Pestaña horizontal	`\xhh`	Número hexadecimal
`\v`	Pestaña vertical		

La constante de caracteres `'\0'` representa el carácter con valor cero, el carácter nulo. `A menudo se escribe` '\0' en lugar de `0` para enfatizar la naturaleza de carácter de alguna expresión, pero el valor numérico es simplemente 0.

Una *expresión constante* es una expresión que implica solo constantes. Dichas expresiones se pueden evaluar durante la compilación en lugar de en tiempo de ejecución y, en consecuencia, se pueden usar en cualquier lugar donde pueda ocurrir una constante, como en

```
#define línea de
```

```
caracteres MAXLINE
1000 [MAXLINE+1];
```

o

```
#define BISIESTO 1 /* en años bisiestos */
días intermedios[31 + 28 + SALTOS + 31 + 30 + 31 + 31 + 31 + 30 + 31 + 31];
```

Una *constante de cadena*, o *literal* de cadena, es una secuencia de cero o más caracteres entre comillas dobles, como en

```
"Soy una cuerda"
```

o

```
"" /* la cadena vacía */
```

Las comillas no forman parte de la cadena, sino que solo sirven para delimitarla. Las mismas secuencias de escape que se usan en las constantes de caracteres se aplican en las cadenas; `\"` representa el carácter de comillas dobles. Las constantes de cadena se pueden concatenar en tiempo de compilación:

```
"Hola" "Mundo"
```

es equivalente a

```
"Hola, mundo"
```

Esto es útil para dividir cadenas largas en varias líneas de origen.

Técnicamente, una constante de cadena es una matriz de caracteres. La representación interna de una cadena tiene un carácter nulo `'\0'` al final, por lo que el almacenamiento físico requerido es uno más que el número de caracteres escritos entre las comillas. Esta representación significa que no hay límite para la longitud que puede tener una cadena, pero los programas deben escanear una cadena completamente para determinar su longitud. La función de biblioteca estándar `strlen(s)` devuelve la longitud de su argumento de cadena de caracteres `s`, excluyendo el terminal `'\0'`. Esta es nuestra versión:

```
/* strlen: longitud de retorno de
s */ int strlen(char s[])
{
    Int I;

    while (s[i] != '\0')
        ++i;
    Retorno I;
}
```

`strlen` y otras funciones de cadena se declaran en el encabezado estándar `<string.h>`.

Tenga cuidado de distinguir entre una constante de caracteres y una cadena que contiene un solo carácter: `'x'` no es lo mismo que `'x'`. El primero es un número entero, que se utiliza para producir el valor numérico de la letra *x* en el juego de caracteres de la máquina. Este último es una matriz de caracteres que contiene un carácter (la letra *x*) y un `'\0'`.

Hay otro tipo de constante, la *constante de enumeración*. Una enumeración es una lista de valores enteros constantes, como en

```
enumeración booleana { NO, SÍ };
```

El primer nombre de una `enumeración` tiene el valor 0, el siguiente 1, y así sucesivamente, a menos que se especifiquen valores explícitos. Si no se especifican todos los valores, los valores no especificados continúan la progresión desde el último valor especificado, como en el segundo de estos ejemplos:

```
los escapes de enumeración { BELL = '\a', RETROCESO = '\b',
                TAB = '\t', NEWLINE = '\n', VTAB = '\v',
                RETURN = '\r' };

enum meses { ENE = 1, FEB, MAR, ABR, MAY, JUN,
              JUL, AGO, SEP, OCT, NOV, DIC };
                    /* FEB = 2, MAR = 3, etc. */
```

Los nombres de las diferentes enumeraciones deben ser distintos. No es necesario que los valores sean distintos en la misma enumeración.

Las enumeraciones proporcionan una manera cómoda de asociar valores constantes con nombres, una alternativa a `#define` con la ventaja de que los valores se pueden generar automáticamente. Aunque se pueden declarar variables de tipos de enumeración, no es necesario que los compiladores comprueben que lo que se almacena en una variable de este tipo es un valor válido para la enumeración. Sin embargo, las variables de enumeración ofrecen la posibilidad de verificarlas y, por lo tanto, a menudo son mejores que `#defines`. `Además, un depurador puede` imprimir valores de variables de enumeración en su forma simbólica.

2.4 Declaraciones

Todas las variables deben declararse antes de su uso, aunque ciertas declaraciones se pueden hacer implícitamente por contenido. Una declaración especifica un tipo y contiene una lista de una o varias variables de ese tipo, como en

```
int inferior, superior,
escalón; char c,
línea[1000];
```

Las variables se pueden distribuir entre las declaraciones de cualquier manera; Las listas anteriores bien podrían escribirse como

```
int inferior;
int superior;
int paso;
char c;
línea de caracteres[1000];
```

Esta última forma ocupa más espacio, pero es conveniente para agregar un comentario a cada declaración para modificaciones posteriores.

Una variable también puede ser inicializada en su declaración. Si el nombre va seguido de un signo igual y una expresión, la expresión actúa como inicializador, como en

```
char esc = '\\';
int   i = 0;
Int   límite =
MAXLINE+1; EPS de
flotación = 1.0E-5;
```

Si la variable en cuestión no es automática, la inicialización se realiza una sola vez, conceptualmente antes de que el programa comience a ejecutarse, y el inicializador debe ser una expresión constante. Una variable automática inicializada explícitamente se inicializa cada vez que se introduce la función o el bloque en el que se encuentra; El inicializador puede ser cualquier expresión. Las variables externas y estáticas se inicializan en cero de forma predeterminada. Las variables automáticas para las que no hay un inicializador explícito tienen valores indefinidos (es decir, no utilizados).

El calificador `const` se puede aplicar a la declaración de cualquier variable para especificar que su valor no se cambiará. Para una matriz, el calificador const dice que los elementos no se modificarán.

```
const doble e = 2.71828182845905;
const char msg[] = "advertencia:
";
```

La declaración `const` también se puede usar con argumentos de matriz, para indicar que la función no cambia esa matriz:

```
int strlen(const char[]);
```

El resultado está definido por la implementación si se intenta cambiar una `const`.

2.5 Operadores aritméticos

Los operadores aritméticos binarios son `+`, `-`, `*`, `/` y el operador de módulo `%`. La división de enteros trunca cualquier parte fraccionaria. La expresión

```
x % y
```

produce el resto cuando `x` se divide por `y` y, por lo tanto, es cero cuando `y` divide a `x` exactamente. Por ejemplo, un año es bisiesto si es divisible por 4 pero no por 100, excepto que los años divisibles por 400 *son* bisiestos. Por lo tanto

```
if ((año % 4 == 0 && año % 100 != 0) || año % 400 == 0)
    printf("%d es un año bisiesto\n", año);
más
    printf("%d no es un año bisiesto\n", año);
```

El operador `%` no se puede aplicar a un `float` o `double`. La dirección de truncamiento para `/` y el signo del resultado para `%` dependen de la máquina para los operandos negativos, al igual que la acción realizada en el desbordamiento o subdesbordamiento.

Los operadores `binarios +` y `-` tienen la misma prioridad, que es menor que la prioridad de `*`, `/` y `%`, que a su vez es menor que las unarias `+` y `-`. Los operadores aritméticos se asocian de izquierda a derecha.

La Tabla 2.1 al final de este capítulo resume la precedencia y asociatividad para todos los operadores.

2.6 Operadores relacionales y lógicos

Los operadores relacionales son:

```
>   >=   <   <=
```

Todos tienen la misma precedencia. Justo debajo de ellos en prioridad están los operadores de igualdad:

```
==   !=
```

Los operadores relacionales tienen menor prioridad que los operadores aritméticos, por lo que una expresión como `i < LIM-1` se toma como `< (LIM-1)`, como era de esperar.

Más interesantes son los operadores lógicos `&&` y `||`. Expresiones conectadas por `&&` o `||` se evalúan de izquierda a derecha, y la evaluación se detiene tan pronto como se conoce la verdad o falsedad del resultado. La mayoría de los programas de C se basan en estas propiedades. Por ejemplo, aquí hay un bucle de la función de entrada `getline` que escribimos en el Capítulo 1:

```
para (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Antes de leer un nuevo carácter es necesario comprobar que hay espacio para almacenarlo en el array `s`, por lo que primero se `debe hacer la prueba` *i < lim-1*. Además, si esta prueba falla, no debemos seguir leyendo otro carácter.

Del mismo modo, sería desafortunado si `c` se probara contra `EOF` antes de llamar a `getchar`; por lo tanto, la llamada y la asignación deben ocurrir antes de que se pruebe el carácter en `c`.

La precedencia de `&&` es mayor que la de `||`, y ambos son menores que los operadores relacionales y de igualdad, por lo que expresiones como

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

No es necesario poner paréntesis adicionales. Pero dado que la precedencia de `!=` es mayor que la asignación, se necesitan paréntesis en

```
(c=getchar()) != '\n'
```

para lograr el resultado deseado de la asignación a `C` y luego la comparación con `'\n'`.

Por definición, el valor numérico de una expresión relacional o lógica es 1 si la relación es verdadera y 0 si la relación es falsa.

El operador de negación unaria `!` Convierte un operando distinto de cero en 0 y un operando cero en 1. Un uso común de `!` está en construcciones como

```
if (!válido)
```

en lugar de

```
if (válido == 0)
```

Es difícil generalizar sobre qué forma es mejor. Las construcciones como `!valid` se leen bien ("si no es válido"), pero las más complicadas pueden ser difíciles de entender.

Ejercicio 2-2. Escriba un bucle equivalente al `bucle for` anterior sin usar `&&` o `||`.

2.7 Conversiones de tipo

Cuando un operador tiene operandos de diferentes tipos, se convierten en un tipo común de acuerdo con un pequeño número de reglas. En general, las únicas conversiones automáticas son aquellas que convierten un operando "más estrecho" en uno "más ancho" sin perder información, como convertir un número entero en punto flotante en una expresión como `f + i`. Las expresiones que no tienen sentido, como usar un `float` como subíndice, no están permitidas. Las expresiones que pueden perder información, como la asignación de un tipo entero más largo a uno más corto o un tipo de punto flotante a un entero, pueden generar una advertencia, pero no son ilegales.

Un `carácter` es solo un número entero pequeño, por lo que `los caracteres se pueden usar libremente en expresiones aritméticas. Esto permite una flexibilidad considerable en ciertos tipos de transformaciones de caracteres. Un ejemplo es esta implementación ingenua de la función atoi`, que convierte una cadena de dígitos en su equivalente numérico.

```
/* atoi: convertir s en entero */
int atoi(char s[])
{
    int i, n;

    n = 0;
    para (i = 0; s[i] >= '0' && s[i] <= '9';
        ++i) n = 10 * n + (s[i] - '0');
    retorno n;
}
```

Como discutimos en el Capítulo 1, la expresión

```
s[i] - '0'
```

da el valor numérico del carácter almacenado en `s[i]`, porque los valores de `'0'`, `'1'`, etc., forman una secuencia creciente contigua.

Otro ejemplo de conversión de `char` a `int` es la función `lower`, que asigna un solo carácter a minúsculas *para el conjunto de caracteres ASCII*. Si el carácter no es una letra mayúscula, `lower` , lo devuelve sin cambios.

```
/* lower: convierte c a minúsculas; Solo ASCII */
int lower (int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    más
        Retorno C;
}
```

Esto funciona para ASCII porque las letras mayúsculas y minúsculas correspondientes están a una distancia fija como valores numéricos y cada alfabeto es contiguo: no hay nada más que letras entre `la A` y `la Z`. Sin embargo, esta última observación no es cierta para el conjunto de caracteres EBCDIC, por lo que este código convertiría algo más que letras en EBCDIC.

El encabezado estándar `<ctype.h>`, descrito en el Apéndice B, define una familia de funciones que proporcionan pruebas y conversiones que son independientes del conjunto de caracteres. Por ejemplo, la función `alower` es un reemplazo portátil de la función `lower` que se muestra arriba. Del mismo modo, la prueba

```
c >= '0' && c <= '9'
```

puede ser reemplazado por

```
isdigit(c)
```

A partir de ahora `utilizaremos las` funciones <ctype.h>.

Hay un punto sutil sobre la conversión de caracteres a números enteros. El lenguaje no especifica si las variables de tipo `char` son cantidades con signo o sin signo. Cuando un `char` se convierte en un `int`, ¿puede producir un entero negativo? La respuesta varía de una máquina a otra, lo que refleja las diferencias en la arquitectura. En algunas máquinas, un `char` cuyo bit más a la izquierda es 1 se convertirá en un entero negativo ("extensión de signo"). En otros, un `char` se promueve a un int agregando ceros en el extremo izquierdo y, por lo tanto, siempre es positivo.

La definición de C garantiza que cualquier carácter del juego de caracteres de impresión estándar de la máquina nunca será negativo, por lo que estos caracteres siempre serán cantidades positivas en las expresiones. Pero los patrones de bits arbitrarios almacenados en variables de caracteres pueden parecer negativos en algunas máquinas, pero positivos en otras. Para la portabilidad, especifique `signed` o `unsigned` si los datos que no son de caracteres se van a almacenar en `variables char`.

Expresiones relacionales como `i > j` y expresiones lógicas conectadas por `&&` y `||` se definen para tener el valor 1 si es verdadero y 0 si es falso. Por lo tanto, la cesión

```
d = c >= '0' && c <= '9'
```

Establece `d` en 1 si `c` es un dígito y 0 si no lo es. Sin embargo, funciones como `isdigit` pueden devolver cualquier valor distinto de cero para true. En la parte de prueba de `if`, `while`, `for`, etc., "true" solo significa "distinto de cero", por lo que esto no hace ninguna diferencia.

Las conversiones aritméticas implícitas funcionan de forma muy parecida a lo esperado. En general, si un operador como + o * que toma dos operandos (un operador binario) tiene operandos de diferentes tipos, el tipo "inferior" se *promueve* al tipo "superior" antes de que la operación continúe. El resultado es del tipo entero. En la sección 6 del apéndice A se establecen con precisión las reglas de conversión. Sin embargo, si no hay operandos sin signo, bastará con el siguiente conjunto informal de reglas:

- Si alguno de los operandos es long double, convierta el otro en long double.
- De lo contrario, si alguno de los operandos es double, convierta el otro en double.
- De lo contrario, si alguno de los operandos es float, convierta el otro en float.
- De lo contrario, convierta char y short en int.
- A continuación, si alguno de los operandos es long, convierta el otro en long.

Tenga en cuenta que los flotantes de una expresión no se convierten automáticamente en dobles; se trata de un cambio con respecto a la definición original. En general, las funciones matemáticas como las de <math.h> usarán doble precisión. La razón principal para usar float es ahorrar almacenamiento en matrices grandes o, con menos frecuencia, ahorrar tiempo en máquinas donde la aritmética de doble precisión es particularmente costosa.

Las reglas de conversión son más complicadas cuando se trata de operandos sin signo. El problema es que las comparaciones entre valores con signo y sin signo dependen de la máquina, ya que dependen de los tamaños de los distintos tipos de enteros. Por ejemplo, supongamos que int es de 16 bits y long es de 32 bits. Entonces -1L < 1U, porque 1U, que es un entero sin signo, se promueve a un largo con signo. Pero -1L > 1UL porque -1L se promueve a largo sin signo y, por lo tanto, parece ser un número positivo grande.

Las conversiones se llevan a cabo en todas las asignaciones; El valor del lado derecho se convierte al tipo del lado izquierdo, que es el tipo del resultado.

Un carácter se convierte en un número entero, ya sea por extensión de signo o no, como se ha descrito anteriormente.

Los enteros más largos se convierten en enteros más cortos o en caracteres eliminando el exceso de bits de orden superior. Por lo tanto, en

```
Int I;
char c;

i = c;
c = i;
```

El valor de c no cambia. Esto es cierto tanto si se trata de una extensión de signo como si no. Sin embargo, invertir el orden de las asignaciones podría perder información.

Si x es float e i es int, entonces x = i e i = x causan conversiones; float a int provoca el truncamiento de cualquier parte fraccionaria. Cuando un valor double se convierte en float, el hecho de que el valor se redondee o se trunque depende de la implementación.

Dado que un argumento de una llamada de función es una expresión, la conversión de tipos también tiene lugar cuando los argumentos se pasan a las funciones. En ausencia de un prototipo de función, char y short se convierten en int, y float se convierte en double. Esta es la razón por la que hemos declarado que los argumentos de la función son int y double incluso cuando la función se llama con char y float.

Por último, las conversiones de tipos explícitas se pueden forzar ("coaccionar") en cualquier expresión, con un operador unario llamado cast. En la construcción

(*nombre de tipo*) *expresión*

La *expresión* se convierte en el tipo con nombre mediante las reglas de conversión anteriores. El significado preciso de una conversión es como si la *expresión* se asignara a una variable del tipo especificado, que luego se usa en lugar de toda la construcción. Por ejemplo, la rutina de la biblioteca `sqrt` espera un `argumento doble` y producirá tonterías si se maneja inadvertidamente otra cosa. (`sqrt` se declara en `<math.h>`.) Entonces, si `n` es un número entero, podemos usar

```
sqrt((doble) n)
```

para convertir el valor de `n` en `double` antes de pasarlo a `sqrt`. Tenga en cuenta que la conversión produce el *valor* de `n` en el tipo adecuado; `n` en sí mismo no se altera. El operador de conversión tiene la misma prioridad que otros operadores unarios, como se resume en la tabla al final de este capítulo.

Si los argumentos son declarados por un prototipo de función, como debería ser normalmente, la declaración provoca la coerción automática de cualquier argumento cuando se llama a la función. Por lo tanto, dado un prototipo de función para `sqrt`:

```
Doble sqrt (doble)
```

La convocatoria

```
raíz2 = sqrt(2)
```

Coacciona el entero `2` en el `valor` doble `2.0` sin necesidad de una conversión.

La biblioteca estándar incluye una implementación portátil de un generador de números pseudoaleatorios y una función para inicializar la semilla; El primero ilustra un reparto:

```
unsigned long int next = 1;

/* rand: devuelve un número entero pseudoaleatorio en
0..32767 */ int rand(void)
{
    siguiente = siguiente * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: establece la semilla
para rand() */ void srand(unsigned
int seed)
{
    siguiente = semilla;
}
```

Ejercicio 2-3. Escriba una función `htoi(s)`, que convierte una cadena de dígitos hexadecimales (incluido un `0x` o `0X opcional`) en su valor entero equivalente. Los dígitos permitidos son del `0` al `9`, de `la a` a la `f` y de `la A` a la `F`.

2.8 Operadores de incremento y decremento

C proporciona dos operadores inusuales para incrementar y disminuir variables. El operador de incremento `++` agrega 1 a su operando, mientras que el operador de decremento `--` resta 1. Con frecuencia hemos usado `++` para incrementar variables, como en

```
if (c == '\n')
    ++nl;
```

El aspecto inusual es que `++` y `--` se pueden usar como operadores de prefijo (antes de la variable, como en `++n`) u operadores de sufijo (después de la variable: `n++`). En ambos casos, el efecto es incrementar `n`. Pero la expresión `++n` incrementa `n` *antes de* que se use su valor, mientras que `n++` incrementa `n` *después de* que se haya usado su valor. Esto significa que en un contexto en el que se utiliza el valor, no solo el efecto, `++n` y `n++` son diferentes. Si `n` es 5, entonces

```
x = n++;
```

establece `x` en 5, pero

```
x = ++n;
```

Establece `x` en 6. En ambos casos, `n` se convierte en 6. Los operadores de incremento y decremento solo se pueden aplicar a variables; Una expresión como `(I+J)++` es ilegal.

En un contexto en el que no se desea ningún valor, solo el efecto incremental, como en

```
if (c == '\n')
    nl++;
```

El prefijo y el sufijo son lo mismo. Pero hay situaciones en las que se requiere específicamente uno u otro. Por ejemplo, considere la función `squeeze(s,c)`, que elimina todas las apariciones del carácter `c` de la cadena `s`.

```
/* exprimir: borrar todo c de s */
void squeeze(char s[], int c)
{
   int i, j;

   para (i = j = 0; s[i] != '\0'; i++)
       si (s[i] != c)
           s[j++] = s[i];
   s[j] = '\0';
}
```

Cada vez que se produce un no `c`, se copia en la posición `j` actual, y solo entonces es `j` incrementado para estar listo para el siguiente carácter. Esto es exactamente equivalente a

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Otro ejemplo de una construcción similar proviene de la función `getline` que escribimos en el Capítulo 1, donde podemos reemplazar

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

por los más compactos

```
if (c == '\n')
   s[i++] = c;
```

Como tercer ejemplo, considere la función estándar `strcat(s,t)`, que concatena la cadena `t` al final de la cadena `s`. `strcat` supone que hay suficiente espacio en `s` para contener la combinación. Tal y como lo hemos escrito, `strcat` no devuelve ningún valor; la versión estándar de la biblioteca devuelve un puntero a la cadena resultante.

```
/* strcat: concatenar t al final de s; s debe ser lo
suficientemente grande */ void strcat(char s[], char t[])
{
```

```
        int i, j;

        i = j = 0;
        while (s[i] != '\0') /* encuentra el final
            de s */ i++;
        while ((s[i++] = t[j++]) != '\0') /* copiar t */
            ;
    }
```

A medida que cada miembro se copia de `t` a `s`, el sufijo `++` se aplica tanto a i como a `j` para asegurarse de que están en posición para el siguiente paso a través del bucle.

Ejercicio 2-4. Escriba una versión alternativa de `squeeze(s1,s2)` que elimine cada carácter en `s1` que coincida con cualquier carácter de la *cadena* `s2`.

Ejercicio 2-5. Escriba la función `any(s1,s2)`, que devuelve la primera ubicación de una cadena `s1` donde aparece cualquier carácter de la cadena `s2`, o `-1` si `s1` no contiene caracteres de `s2`. (La función estándar de la biblioteca `strpbrk` hace el mismo trabajo, pero devuelve un puntero a la ubicación).

2.9 Operadores bit a bit

C proporciona seis operadores para la manipulación de bits, que sólo pueden aplicarse a operandos enteros, es decir, `char`, `short`, `int` y `long`, con o sin signo.

`&`	bit a bit Y
`|`	O inclusivo bit a bit
`^`	bit a bit exclusivo OR
`<<`	Desplazamiento a la izquierda
`>>`	cambio a la derecha
`~`	Complemento (unario)

El operador AND bit a bit `&` se usa a menudo para enmascarar algún conjunto de bits, por ejemplo

```
n = n y 0177;
```

Establece a cero todos los bits de n, excepto los 7 bits de n de orden inferior.

El operador OR bit a bit `|` se utiliza para encender bits:

```
x = x | SET_ON;
```

Establece en uno en `x` los bits que se establecen en uno en `SET_ON`.

El operador OR exclusivo bit a bit `^` establece un uno en cada posición de bit donde sus operandos tienen bits diferentes, y cero donde son iguales.

Hay que distinguir los operadores bit a bit `&` y `|` de los operadores lógicos `&&` y `||`, que implican una evaluación de izquierda a derecha de un valor de verdad. Por ejemplo, si `x` es 1 e `y es` 2, entonces `x e y` es cero mientras que `x & y` es uno.

Los operadores de desplazamiento `<<` y `>>` realizan desplazamientos a la izquierda y a la derecha de su operando izquierdo por el número de posiciones de bits dado por el operando derecho, que debe ser no negativo. Por lo tanto, `x << 2` turnos

el valor de `x` en dos posiciones, llenando los bits vacíos con cero; esto es equivalente a la multiplicación por 4. El desplazamiento a la derecha de una `cantidad sin signo` siempre se ajusta a los bits vacantes con cero. Al desplazar a la derecha, una cantidad con signo se llenará con signos de bits ("desplazamiento aritmético") en algunas máquinas y con 0 bits ("desplazamiento lógico") en otras.

El operador unario `~` produce el complemento uno de un entero, es decir, convierte cada bit 1 en un bit 0 y viceversa. Por ejemplo

```
x = x & ~077
```

Establece los últimos seis bits de `x` en cero. Tenga en cuenta que `x & ~077` es independiente de la longitud de la palabra y, por lo tanto, es preferible a, por ejemplo, `x y 0177700`, que supone que `x` es una cantidad de 16 bits. El formulario portátil no implica ningún costo adicional, ya `que ~077` es una expresión constante que se puede evaluar en tiempo de compilación.

Como ilustración de algunos de los operadores de bits, considere la función `getbits(x,p,n)` que devuelve el campo de `n` bits (ajustado a la derecha) `de x` que comienza en la posición `p`. Suponemos que la posición del bit 0 está en el extremo derecho y que `n` y `p` son valores positivos sensatos. Por ejemplo, `getbits(x,4,3)` devuelve los tres bits en las posiciones 4, 3 y 2, ajustados a la derecha.

```
/* getbits: obtener n bits de la posición p */
getbits sin signo(x, int p, int n) sin signo
{
    retorno (x >> (p+1-n)) & ~(~0 << n);
}
```

La expresión `x >> (p+1-n)` mueve el campo deseado al extremo derecho de la palabra. `~0` es todo de 1 bit; al desplazarlo a la izquierda `n` posiciones con `~0<<n` se colocan ceros en los n bits más a la derecha ; complementando eso con `~` se crea una máscara con unos en los n bits más a la derecha .

Ejercicio 2-6. Escriba una función `setbits(x,p,n,y)` que devuelva `x` con los `n` bits que comienzan en la posición `p` establecidos en los n bits más a la derecha de `y`, dejando los otros bits sin cambios.

Ejercicio 2-7. Escriba una función `invert(x,p,n)` que devuelva `x` con los `n` bits que comienzan en la posición `p` invertidos (es decir, 1 cambiado a 0 y viceversa), dejando los demás sin cambios.

Ejercicio 2-8. Escriba una función `rightrot(x,n)` que devuelva el valor del entero `x` girado a la derecha en `n` posiciones.

2.10 Operadores de asignación y expresiones

Una expresión como

```
i = i + 2
```

en el que la variable del lado izquierdo se repite inmediatamente a la derecha, se puede escribir en forma comprimida

```
i += 2
```

El operador `+=` se denomina *operador de asignación*.

La mayoría de los operadores binarios (operadores como `+` que tienen un operando izquierdo y derecho) tienen un operador de asignación correspondiente *op*=, donde *op* es uno de los

siguientes operadores.

```
+   -   *   /   %   <<   >>   &   ^   |
```

Si *expr1* y *expr2* son expresiones, entonces

```
expr1 op= expr2
```

es equivalente a

```
expr1 = (expr1) en (expr2)
```

excepto que *expr1* se calcula solo una vez. Fíjate en los paréntesis alrededor de *expr2*:

```
x *= y + 1
```

medio

```
x = x * (y + 1)
```

en lugar de

```
x = x * y + 1
```

Por ejemplo, la función `bitcount` cuenta el número de bits 1 en su argumento entero.

```
/* bitcount: cuenta 1 bit en x */
int bitcount(unsigned x)
{
    int b;

    para (b = 0; x != 0; x >>= 1)
        si (x y 01)
            b++;
    Retorno B;
}
```

Declarar el argumento `x` como `unsigned` garantiza que cuando se desplaza a la derecha, los bits vacíos se rellenarán con ceros, no con bits de signo, independientemente de la máquina en la que se ejecute el programa.

Aparte de la concisión, los operadores de asignación tienen la ventaja de que se corresponden mejor con la forma de pensar de las personas. Decimos "suma 2 a `i`" o "incrementa `i` en 2", no "toma `i`, suma 2 y luego vuelve a poner el resultado en `i`". Por lo tanto, la expresión `i += 2` es preferible a `i = i+2`. Además, para una expresión complicada como

```
yyval[yypv[p3+p4] + yypv[p1]] += 2
```

El operador de asignación hace que el código sea más fácil de entender, ya que el lector no tiene que comprobar minuciosamente que dos expresiones largas son realmente iguales, o preguntarse por qué no lo son. Y un operador de asignación puede incluso ayudar a un compilador a producir código eficiente.

Ya hemos visto que la declaración de asignación tiene un valor y puede ocurrir en expresiones; El ejemplo más común es

```
while ((c = getchar()) != EOF)
    ...
```

Los otros operadores de asignación (`+=`, `-=`, etc.) también pueden aparecer en expresiones, aunque esto es menos frecuente.

En todas estas expresiones, el tipo de una expresión de asignación es el tipo de su operando izquierdo y el valor es el valor después de la asignación.

Ejercicio 2-9. En un sistema numérico de complemento a dos, `x &= (x-1)` elimina el bit 1 más a la derecha en `x`. Explique por qué. Use esta observación para escribir una versión más rápida de `bitcount`.

2.11 Expresiones condicionales

Las declaraciones

```
Si (a > b)
    z = a;
más
    z = b;
```

Calcule en `z` el máximo de `a` y `b`. La *expresión condicional*, escrita con el operador ternario "`?:`", proporciona una forma alternativa de escribir esta y otras construcciones similares. En la expresión

```
expr1 ? expr2 : expr3
```

En primer lugar, se evalúa la expresión *expr1*. Si es distinto de cero (true), se evalúa la expresión *expr2* y ese es el valor de la expresión condicional. De lo contrario, se evalúa *expr3* y ese es el valor. Solo se evalúa uno de *expr2* y *expr3*. Por lo tanto, para poner `z` al máximo de `a` y `b`,

```
z = (a > b) ? a : b;     /* z = máx(a, b) */
```

Cabe señalar que la expresión condicional es de hecho una expresión, y se puede usar donde sea que pueda estar cualquier otra expresión. Si *expr2* y *expr3* son de tipos diferentes, el tipo del resultado viene determinado por las reglas de conversión descritas anteriormente en este capítulo. Por ejemplo, si `f` es un `float` y `n` un `int`, entonces la expresión

```
(n > 0) ? f : n
```

es de tipo `float` independientemente de si `n` es positivo.

Los paréntesis no son necesarios alrededor de la primera expresión de una expresión condicional, ya que la precedencia de `?:` es muy baja, justo por encima de la asignación. Sin embargo, son recomendables de todos modos, ya que hacen que la parte de la condición de la expresión sea más fácil de ver.

La expresión condicional a menudo conduce a un código sucinto. Por ejemplo, este bucle imprime `n` elementos de una matriz, 10 por línea, con cada columna separada por un espacio en blanco y con cada línea (incluida la última) terminada por una nueva línea.

```
para (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

Se imprime una nueva línea después de cada décimo elemento y después de la n-ésima. Todos los demás elementos van seguidos de un espacio en blanco. Esto puede parecer complicado, pero es más compacto que el equivalente `if- else`. Otro buen ejemplo es

```
printf("Tienes %d artículos%s.\n", n, n==1 ? "" : "s");
```

Ejercicio 2-10. Reescriba la función `lower`, que convierte las letras mayúsculas en minúsculas, con una expresión condicional en lugar de `if-else`.

2.12 Precedencia y orden de evaluación

En la tabla 2.1 se resumen las reglas de precedencia y asociatividad de todos los operadores, incluidos aquellos que aún no hemos discutido. Los operadores de la misma línea tienen la misma prioridad; Las filas están en orden de prioridad decreciente, por lo que, por ejemplo, `*`, `/` y `%` tienen la misma prioridad, que es mayor que la de los binarios `+` y `-`. El "operador" `()` se refiere a la llamada a la función. Los operadores `->` y `.` se utilizan para acceder a los miembros de las estructuras; se tratarán de

Capítulo 6, junto con `sizeof` (tamaño de un objeto). En el capítulo 5 se analiza `*` (direccionamiento indirecto a través de un puntero) y `&` (dirección de un objeto), y en el capítulo 3 se analiza el operador de coma.

Operadores	Asociatividad
`() [] -> .`	De izquierda a derecha
`! ~ ++ -- + - *` (*tipo*) `tamañode`	De derecha a izquierda
`* / %`	De izquierda a derecha
`+ -`	De izquierda a derecha
`<< >>`	De izquierda a derecha
`< <= > >=`	De izquierda a derecha
`== !=`	De izquierda a derecha
`&`	De izquierda a derecha
`^`	De izquierda a derecha
`|`	De izquierda a derecha
`&&`	De izquierda a derecha
`||`	De izquierda a derecha
`?:`	De derecha a izquierda
`= += -= *= /= %= &= ^= |= <<= >>=`	De derecha a izquierda
`,`	De izquierda a derecha

Unary & +, -, y * tienen mayor prioridad que las formas binarias.

Tabla 2.1: *Precedencia y asociatividad de operadores*

Tenga en cuenta que la precedencia de los operadores bit a bit `&`, `^` y `|` cae por debajo `de ==` y `!=`. Esto implica que las expresiones de prueba de bits como

```
if ((x & MASK) == 0) ...
```

debe estar completamente entre paréntesis para dar resultados adecuados.

C, como la mayoría de los idiomas, no especifica el orden en el que se evalúan los operandos de un operador. (Las excepciones son: `&&`, `||`, `?:`, y ','.) Por ejemplo, en una instrucción como

```
x = f() + g();
```

`f` puede evaluarse antes que `g` o viceversa; por lo tanto, si f o `g` alteran una variable de la que depende la otra, `x` puede depender del orden de evaluación. Los resultados intermedios se pueden almacenar en variables temporales para garantizar una secuencia particular.

Del mismo modo, no se especifica el orden en el que se evalúan los argumentos de función, por lo que la instrucción

```
printf("%d %d\n", ++n, power(2, n));    /*INCORRECTO*/
```

puede producir diferentes resultados con diferentes compiladores, dependiendo de si `n` se incrementa antes de llamar a power. La solución, por supuesto, es escribir

```
++n;
```

```
printf("%d %d\n", n, power(2, n));
```

Las llamadas a funciones, las instrucciones de asignación anidadas y los operadores de incremento y decremento causan
"efectos secundarios" - alguna variable se cambia como un subproducto de la evaluación de una expresión. En cualquier expresión que implique efectos secundarios, puede haber dependencias sutiles en el orden en que se actualizan las variables que participan en la expresión. Una situación infeliz es tipificada por la declaración

```
a[i] = i++;
```

La cuestión es si el subíndice es el antiguo valor de `i` o el nuevo. Los compiladores pueden interpretar esto de diferentes maneras y generar diferentes respuestas dependiendo de su interpretación. La norma deja intencionadamente la mayoría de estos asuntos sin especificar. Cuando se producen efectos secundarios (asignación a variables) dentro de una expresión se deja a discreción del compilador, ya que el mejor orden depende en gran medida de la arquitectura de la máquina. (El estándar especifica que todos los efectos secundarios de los argumentos tienen efecto antes de que se llame a una función, pero eso no ayudaría en la llamada a `printf` arriba.)

La moraleja es que escribir código que depende del orden de evaluación es una mala práctica de programación en cualquier lenguaje. Naturalmente, es necesario saber qué cosas evitar, pero si no sabe *cómo* se hacen en varias máquinas, no tendrá la tentación de aprovechar una implementación en particular.

Capítulo 3 - Flujo de control

El flujo de control de un lenguaje especifica el orden en el que se realizan los cálculos. Ya hemos conocido las construcciones de flujo de control más comunes en ejemplos anteriores; Aquí completaremos el conjunto, y seremos más precisos sobre los comentados anteriormente.

3.1 Sentencias y bloques

Una expresión como `x = 0` o `i++` o `printf(...)` se convierte en una *instrucción* cuando va seguida de un punto y coma, como en

```
x = 0;
I++;
printf(...);
```

En C, el punto y coma es un terminador de instrucción, en lugar de un separador como lo es en lenguajes como Pascal.

Las llaves `{` y `}` se utilizan para agrupar declaraciones e instrucciones en una *instrucción compuesta*, o *bloque*, de modo que sean sintácticamente equivalentes a una sola instrucción. Las llaves que rodean las declaraciones de una función son un ejemplo obvio; Entre llaves alrededor de varias instrucciones después de un `if`, `else`, `while` o `for` son otro. (Las variables se pueden declarar dentro de *De* esto hablaremos en el capítulo 4.) No hay punto y coma después de la llave derecha que termina un bloque.

3.2 Si-Else

La instrucción `if-else` se utiliza para expresar decisiones. Formalmente, la sintaxis es

```
if (expresión)
    Afirmación1
más
    Declaración2
```

donde la parte `else` es opcional. Se evalúa la *expresión*; si es verdadera (es decir, si la *expresión* tiene un valor distinto de cero), *se ejecuta statement1*. Si es falso (*la expresión* es cero) y si hay una parte `else`, *se ejecuta statement2 en su lugar.*

Dado que un `if` prueba el valor numérico de una expresión, son posibles ciertos atajos de codificación.
La más obvia es la escritura

```
if (expresión)
```

En lugar de

```
if (expresión != 0)
```

A veces esto es natural y claro; En otras ocasiones puede ser críptico.

Debido a que la parte else de un `if-else` es opcional, existe una ambigüedad cuando se omite un else if de una secuencia if anidada. Esto se resuelve asociando el `else` con el `else-less` if `anterior más cercano`. Por ejemplo, en

```
si (n > 0)
    Si (a > b)
        z = a;
    más
        z = b;
```

el `else` va al `si interno`, como hemos demostrado por sangría. Si eso no es lo que quieres, debes usar aparatos ortopédicos para forzar la asociación adecuada:

```
if (n > 0) {
    Si (a > b)
         z = a;
}
más
    z = b;
```

La ambigüedad es especialmente perniciosa en situaciones como esta:

```
si (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            Retorno I;
        }
más         /*INCORRECTO*/
    printf("error -- n es negativo\n");
```

La sangría muestra inequívocamente lo que desea, pero el compilador no recibe el mensaje y asocia el `else` con el `if interno`. Este tipo de error puede ser difícil de encontrar; Es una buena idea usar llaves cuando hay `condiciones if anidadas`.

Por cierto, fíjate que hay un punto y coma después de `z = a` en

```
Si (a > b)
    z = a;
más
    z = b;
```

Esto se debe a que, gramaticalmente, una *declaración* sigue al `if`, y una declaración de expresión como
"`z = a;`" siempre termina con un punto y coma.

3.3 De lo contrario-si

La construcción

```
if (expresión)
    declaración
else if (expresión)
    declaración
else if (expresión)
    declaración
else if (expresión)
    declaración
más
    declaración
```

ocurre con tanta frecuencia que vale la pena una breve discusión por separado. Esta secuencia de enunciados if es la forma más general de escribir una decisión multidireccional. Las *expresiones* se evalúan en orden; si una *expresión* es verdadera, se ejecuta la instrucción asociada a ella, y esto termina toda la cadena. Como siempre, el código de cada *instrucción* es una sola instrucción o un grupo de ellas entre llaves.

La última `parte else` maneja el caso "ninguno de los anteriores" o el caso predeterminado donde no se cumple ninguna de las otras condiciones. A veces no hay ninguna acción explícita para el valor predeterminado; En ese caso, el arrastre

```
más
    declaración
```

se puede omitir, o se puede usar para la verificación de errores para detectar una condición "imposible".

Para ilustrar una decisión de tres vías, aquí hay una función de búsqueda binaria que decide si un valor particular `x` ocurre en la matriz ordenada `v`. Los elementos de `v` deben estar en orden creciente. La función devuelve la posición (un número entre 0 y `n-1`) si `x` aparece en `v` y -1 si no lo está.

La búsqueda binaria compara primero el valor de entrada `x` con el elemento central de la matriz `v`. Si `x` es menor que el valor medio, la búsqueda se centra en la mitad inferior de la tabla, de lo contrario, en la mitad superior. En cualquier caso, el siguiente paso es comparar `x` con el elemento central de la mitad seleccionada. Este proceso de dividir el rango en dos continúa hasta que se encuentra el valor o el rango está vacío.

```
/* binsearch: encuentra x en v[0] <= v[1] <= ... <= v[n-
1] */ int binsearch(int x, int v[], int n)
{
    int bajo, alto, medio;

    bajo = 0;
    alto = n - 1;
    while (bajo <= alto) {
        medio =
        (bajo+alto)/2; if
        (x < v[medio])
            alto = medio + 1;
        else if (x > v[mid])
            bajo = medio + 1;
        más  /* coincidencia
            encontrada */
            devuelve mid;
    }
    retorno -1;  /* sin coincidencia */
}
```

La decisión fundamental es si `x` es menor, mayor o igual que el elemento medio `v[mid]` en cada paso; esto es natural para `else-if`.

Ejercicio 3-1. Nuestra búsqueda binaria realiza dos pruebas dentro del bucle, cuando una sería suficiente (al precio de más pruebas fuera). Escriba una versión con una sola prueba dentro del bucle y mida la diferencia en tiempo de ejecución.

3.4 Interruptor

La instrucción `switch` es una decisión multidireccional que prueba si una expresión coincide con uno de varios *valores enteros constantes y se bifurca en consecuencia.*

```
switch (expresión) {
    case const-expr: sentencias
    case const-expr: sentencias
    default: sentencias
}
```

Cada caso se etiqueta mediante una o varias constantes o expresiones constantes con valores enteros. Si un caso coincide con el valor de la expresión, la ejecución se inicia en ese caso. Todas las expresiones de mayúsculas y minúsculas deben ser diferentes. El caso etiquetado como `default` se ejecuta si no se cumple ninguno de los otros casos. Un `valor predeterminado` es opcional; si no está allí y si ninguno de los casos coincide, no se lleva a cabo ninguna acción. Los casos y la cláusula de incumplimiento pueden aparecer en cualquier orden.

En el Capítulo 1 escribimos un programa para contar las ocurrencias de cada dígito, espacio en blanco y todos los demás caracteres, usando una secuencia de `si ... de lo contrario, si ... de lo contrario`. Aquí está el mismo programa con un `interruptor`:

```
#include <stdio.h>

main() /* contar dígitos, espacio en blanco, otros */
{
    int c, i, nwhite, nother, ndigit[10];

    nblanco = otro = 0;
    para (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
        Caso '0': Caso '1': Caso '2': Caso '3': Caso '4':
        Caso '5': Caso '6': Caso '7': Caso '8': Caso '9':
            ndigit[C-'0']++;
            quebra
        r; caso '
        ':
        Caso '\n':
        Caso '\t':
            nblanco++;
            quebrar;
        predeterminado:
            Otro++;
            quebrar;
        }
    }
    printf("dígitos =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", espacio en blanco = %d, otro =
        %d\n", nblanco, otro);
    devuelve 0;
}
```

La instrucción `break` provoca una salida inmediata del `switch`. Dado que los casos sirven solo como etiquetas, una vez finalizado el código de un caso, la ejecución *pasa* al siguiente a menos que se realice una acción explícita para escapar. `Break` y `Return` son las formas más comunes de salir de un `interruptor`. Una `instrucción break` también se puede usar para forzar una salida inmediata de los bucles `while`, `for` y `do`, como se discutirá más adelante en este capítulo.

Caer a través de los casos es una bendición mixta. En el lado positivo, permite adjuntar varios casos a una sola acción, como ocurre con los dígitos de este ejemplo. Pero también implica que, normalmente, cada caso debe terminar con una `pausa` para evitar pasar al siguiente. Pasar de un caso a otro no es robusto, siendo propenso a la desintegración cuando se modifica el programa. Con la excepción de varias etiquetas para un solo cálculo, las caídas deben usarse con moderación y comentarse.

Como una buena forma, ponga un `descanso` después del último caso (el `valor predeterminado` aquí) aunque sea lógicamente innecesario. Algún día, cuando se agregue otro caso al final, este poco de programación defensiva te salvará.

Ejercicio 3-2. Escriba una función `escape(s,t)` que convierta caracteres como newline y tab en secuencias de escape visibles como `\n` y `\t` a medida que copia la cadena `t` a `s`. Usa un `interruptor`. Escribe también una función para la otra dirección, convirtiendo las secuencias de escape en los personajes reales.

3.5 Bucles - Mientras y Para

Ya nos hemos encontrado con los bucles `while` y `for`. En

```
while (expresión)
    declaración
```

Se evalúa la *expresión*. Si es distinto de cero, *se ejecuta la instrucción y* se vuelve a evaluar la expresión. Este ciclo continúa hasta *que la expresión* se convierte en cero, momento en el que la ejecución se reanuda después de la *instrucción*.

La `instrucción` for

```
para (expr1; expr2; expr3)
```

es equivalente a

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

excepto por el comportamiento de `continue`, que se describe en la Sección 3.7.

Gramaticalmente, los tres componentes de un `bucle for` son expresiones. Por lo general, *expr1* y *expr3* son asignaciones o llamadas a funciones y *expr2* es una expresión relacional. Se puede omitir cualquiera de las tres partes, aunque se debe mantener el punto y coma. Si *se omite expr1* o *expr3*, simplemente se quita de la expansión. Si la prueba, *expr2*, no está presente, se toma como permanentemente verdadera, por lo que

```
para (;;) {
    ...
}
```

es un bucle "infinito", que presumiblemente se romperá por otros medios, como una `ruptura` o un `retorno`.

Ya sea que se use `durante` o `para` es en gran medida una cuestión de preferencia personal. Por ejemplo, en

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
    ;   /* Omitir caracteres de espacio en blanco */
```

No hay inicialización ni reinicialización, por lo que el `mientras` es más natural.

El `for` es preferible cuando hay una inicialización e incremento simples, ya que mantiene las instrucciones de control del bucle juntas y visibles en la parte superior del bucle. Esto es más obvio en

```
para (i = 0; i < n; i++)
    ...
```

que es el modismo C para procesar los primeros `n` elementos de una matriz, el análogo del bucle DO de Fortran o el para. Sin embargo, la analogía no es perfecta, ya que la variable de índice `i` conserva su valor cuando el bucle termina por cualquier motivo. Dado que los componentes de for son expresiones arbitrarias, los bucles for no se restringen a progresiones aritméticas. No obstante, es de mal estilo forzar cálculos no relacionados en la inicialización y el incremento de un `for`, que es mejor reservar para operaciones de control de bucle.

Como un ejemplo más grande, aquí hay otra versión de `atoi` para convertir una cadena a su equivalente numérico. Este es un poco más general que el del Capítulo 2; se ocupa de un espacio en blanco inicial opcional y un signo `+` o `-` opcional. (En el capítulo 4 se muestra `atof`, que hace la misma conversión para los números de coma flotante).

La estructura del programa refleja la forma de la entrada:

Omita el espacio en
blanco, si hay algún
signo de obtención, si lo
hay
Obtener la parte entera y convertirla

Cada paso hace su parte y deja las cosas en un estado limpio para el siguiente. Todo el proceso termina en el primer carácter que no puede formar parte de un número.

```
#include <tipo.h>

/* atoi: convierte s en entero; version 2 */
int atoi(char s[])
{
    int i, n, signo;

    for (i = 0; isspace(s[i]); i++) /* omitir espacio en blanco */
        ;
    signo = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-') /* saltar signo
        */ i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    signo de retorno * n;
}
```

La biblioteca estándar proporciona una función más elaborada `strtol` para la conversión de cadenas a enteros largos; véase la Sección 5 del Apéndice B.

Las ventajas de mantener centralizado el control de bucles son aún más obvias cuando hay varios bucles anidados. La siguiente función es una ordenación de Shell para ordenar una matriz de enteros. La idea básica de este algoritmo de clasificación, que fue inventado en 1959 por D. L. Shell, es que en las primeras etapas, se comparan elementos distantes, en lugar de adyacentes como en clasificaciones de intercambio más simples. Esto tiende a eliminar grandes cantidades de trastorno rápidamente, por lo que las etapas posteriores tienen menos trabajo por hacer. El intervalo entre los elementos comparados se reduce gradualmente a uno, momento en el que la ordenación se convierte efectivamente en un método de intercambio adyacente.

```
/* shellsort: ordenar v[0]... v[n-1] en orden creciente */
void shellsort(int v[], int n)
{
```

```
    int gap, i, j, temp;

    para (espacio = n/2; espacio > 0;
        espacio /= 2) para (i =
        espacio; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] =
                temperatura;
            }
}
```

Hay tres bucles anidados. El más externo controla el espacio entre los elementos comparados, reduciéndolo de `n/2` por un factor de dos en cada pasada hasta que se convierte en cero. El bucle central camina a lo largo de los elementos. El bucle más interno compara cada par de elementos que están separados por `un espacio` e invierte los que están desordenados. Dado que el `espacio` finalmente se reduce a uno, todos los elementos finalmente se ordenan correctamente. Observe cómo la generalidad del `for` hace que el bucle exterior encaje en la misma forma que los demás, aunque no sea una progresión aritmética.

Un último operador C es la coma ",", que se usa con mayor frecuencia en la `instrucción for`. Un par de expresiones separadas por una coma se evalúa de izquierda a derecha, y el tipo y el valor del resultado son el tipo y el valor del operando derecho. Por lo tanto, en una instrucción for, es posible colocar varias expresiones en las distintas partes, por ejemplo, para procesar dos índices en paralelo. Esto se ilustra en la función `reverse(s)`, que invierte la cadena `s` en su lugar.

```
#include <cadena.h>

/* reverse: cuerda inversa s en su lugar
*/ void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Las comas que separan los argumentos de la función, las variables en las declaraciones, etc., *no son* operadores de coma y no garantizan la evaluación de izquierda a derecha.

Los operadores de coma deben usarse con moderación. Los usos más adecuados son para construcciones fuertemente relacionadas entre sí, como en el `bucle for` a `la inversa`, y en macros donde un cálculo de varios pasos tiene que ser una sola expresión. Una expresión de coma también puede ser apropiada para el intercambio de elementos a `la inversa`, donde el intercambio se puede pensar en una sola operación:

```
 for (i = 0, j = strlen(s)-1; i < j; i++, j--)
        c = s[i], s[i] = s[j], s[j] = c;
```

Ejercicio 3-3. Escriba una función `expand(s1,s2)` que expanda las notaciones abreviadas como `a-z` en la cadena `s1` en la lista completa equivalente `abc... xyz` en `s2`. Permita letras de cualquiera de las mayúsculas y minúsculas y dígitos, y esté preparado para manejar casos como `a-b-c` y `a-z0-9` y `-a-z`. Organice que un adelanto o un final `-` se tome literalmente.

3.6 Bucles - Hacer mientras

Como discutimos en el Capítulo 1, los bucles `while` y `for` prueban la condición de terminación en la parte superior. Por el contrario, el tercer bucle en C, el `do-while`, se prueba en la parte inferior *después de* hacer cada paso a través del cuerpo del bucle; el cuerpo siempre se ejecuta al menos una vez.

La sintaxis del `do` es

```
hacer
    declaración
while (expresión);
```

Se ejecuta la *instrucción* y, a continuación, se evalúa la expresión. Si es verdadera, *la instrucción* se evalúa de nuevo, y así sucesivamente. Cuando la expresión se convierte en falsa, el bucle termina. Excepto por el sentido de la prueba, `do-while` es equivalente a la instrucción `repeat-until` de Pascal.

La experiencia demuestra que `el do-while` se usa mucho menos que `el while` y `for`. Sin embargo, de vez en cuando es valioso, como en la siguiente función `itoa`, que convierte un número en una cadena de caracteres (el inverso de `atoi`). El trabajo es un poco más complicado de lo que podría pensarse en un principio, porque los métodos sencillos de generar los dígitos los generan en el orden incorrecto. Hemos elegido generar la cadena al revés y luego invertirla.

```
/* itoa: convierte n en caracteres en s
*/ void itoa(int n, char s[])
{
    int i, signo;

    if ((signo = n) < 0) /* signo de registro */
        n = -n;/* hacer n positivo */ i = 0;

    hacer {/* genera dígitos en orden inverso */
        s[i++] = n % 10 + '0'; /* obtiene el
        siguiente dígito */
    } while ((n /= 10) > 0);/* borrarlo */ if
    (signo < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverso(s);
}
```

El `do-while` es necesario, o al menos conveniente, ya que al menos un carácter debe estar instalado en la matriz `s`, incluso si `n` es cero. También usamos llaves alrededor de la declaración única que compone el cuerpo del `do-while`, aunque son innecesarias, para que el lector apresurado no confunda la parte `while` con el *comienzo* de un `bucle while`.

Ejercicio 3-4. En una representación de número de complemento a dos, nuestra versión de `itoa` no maneja el número negativo más grande, es decir, el valor de `n` igual a -(2wordsize-1). Explique por qué no. Modifíquelo para imprimir ese valor correctamente, independientemente de la máquina en la que se ejecute.

Ejercicio 3-5. Escriba la función `itob(n,s,b)` que convierte el entero `n` en una representación de carácter base `b` en la cadena `s`. En particular, `itob(n,s,16)` da formato `s` como un entero hexadecimal en `s`.

Ejercicio 3-6. Escriba una versión de `itoa` que acepte tres argumentos en lugar de dos. El tercer argumento es un ancho de campo mínimo; El número convertido debe rellenarse con espacios en blanco a la izquierda si es necesario para que sea lo suficientemente ancho.

3.7 Pausa y continúa

A veces es conveniente poder salir de un bucle que no sea probando en la parte superior o inferior. La instrucción `break` proporciona una salida anticipada de `for`, `while` y `do`, al igual que from `switch`. Una `interrupción` hace que el bucle o conmutador envolvente más interno salga inmediatamente.

La siguiente función, `trim`, elimina los espacios en blanco, las tabulaciones y las saltos de línea finales del final de una cadena, utilizando un `salto` para salir de un bucle cuando se encuentra la línea no en blanco, que no es una tabulación y que no es una nueva línea más a la derecha.

```
/* trim: eliminar espacios en blanco finales,
pestañas, saltos de línea */ int trim(char s[])
{
    int n;

    para (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' & s[n] != '\t' && s[n] != '\n')
            interrupción;
    s[n+1] = '\0';
    retorno n;
}
```

`strlen` devuelve la longitud de la cadena. El bucle `for` comienza al final y escanea hacia atrás en busca del primer carácter que no sea un espacio en blanco, una tabulación o una nueva línea. El bucle se rompe cuando se encuentra uno, o cuando `n` se vuelve negativo (es decir, cuando se ha escaneado toda la cadena). Debe comprobar que este es el comportamiento correcto incluso cuando la cadena está vacía o contiene solo caracteres de espacio en blanco.

La instrucción `continue` está relacionada con `break`, pero se usa con menos frecuencia; hace que comience la siguiente iteración del `bucle for`, `while` o `do` que la envuelve. En el `while` y `do`, esto significa que la parte de prueba se ejecuta inmediatamente; en el `for`, el control pasa al paso de incremento. La instrucción `continue` solo se aplica a los bucles, no al `switch`. Un `continue` dentro de un `modificador` dentro de un bucle provoca la siguiente iteración del bucle.

Por ejemplo, este fragmento procesa solo los elementos no negativos de la matriz `a`; se omiten los valores negativos.

```
para (i = 0; i < n; i++)
    if (a[i] < 0)/* omitir elementos negativos
        */ continuar;
    ... /* hacer elementos positivos */
```

La instrucción `continue` se usa a menudo cuando la parte del bucle que sigue es complicada, de modo que revertir una prueba y aplicar sangría a otro nivel anidaría el programa demasiado profundamente.

3.8 Ir a y etiquetas

C proporciona la instrucción goto, de la que se puede abusar infinitamente , y las etiquetas a las que bifurcar. Formalmente, la `instrucción goto` nunca es necesaria, y en la práctica casi siempre es fácil escribir código sin ella. No hemos usado `goto` en este libro.

Sin embargo, hay algunas situaciones en las que `los gotos pueden encontrar un lugar. Lo más común es abandonar el` procesamiento en alguna estructura profundamente anidada, como salir de dos o más bucles a la vez. La instrucción `break` no se puede usar directamente, ya que solo sale del bucle más interno. Así:

```
    para ( ... )
        para ( ... ) {
            ...
            if (desastre)
                Error de ir a;
        }
    ...
error:
    /* limpiar el desorden */
```

Esta organización es útil si el código de control de errores no es trivial y si los errores pueden ocurrir en varios lugares.

Una etiqueta tiene la misma forma que el nombre de una variable y va seguida de dos puntos. Se puede adjuntar a cualquier instrucción en la misma función que el `goto`. El alcance de una etiqueta es toda la función.

Como otro ejemplo, considere el problema de determinar si dos matrices `a` y `b` tienen un elemento en común. Una posibilidad es

```
    para (i = 0; i < n; i++)
        para (j = 0; j < m; j++)
            si (a[i] == b[j])
                goto encontrado;
    /* no encontró ningún elemento común */
    ...
fundar:
    /* tiene uno: a[i] == b[j] */
    ...
```

El código que involucra un `goto` siempre se puede escribir sin uno, aunque tal vez a costa de algunas pruebas repetidas o una variable adicional. Por ejemplo, la búsqueda de matrices se convierte en

```
encontrado = 0;
for (i = 0; i < n && !found; i++)
    para (j = 0; j < m && !encontrado;
        j++) si (a[i] == b[j])
            encontrado = 1;
if (encontrado)
    /* Obtuve uno: A[I-1] == B[J-1] */
    ...
más
    /* no encontró ningún elemento común */
    ...
```

Con algunas excepciones como las citadas aquí, el código que se basa en `instrucciones goto` suele ser más difícil de entender y mantener que el código sin `gotos`. `Aunque no somos dogmáticos sobre el asunto, parece que las declaraciones goto` deberían usarse raramente, si es que se usan.

Capítulo 4 - Funciones y estructura del programa

Las funciones dividen las grandes tareas informáticas en otras más pequeñas y permiten a las personas construir sobre lo que otros han hecho en lugar de empezar de cero. Las funciones apropiadas ocultan los detalles de la operación de las partes del programa que no necesitan conocerlas, aclarando así el conjunto y aliviando el dolor de hacer cambios.

C ha sido diseñado para hacer que las funciones sean eficientes y fáciles de usar; Los programas C generalmente constan de muchas funciones pequeñas en lugar de unas pocas grandes. Un programa puede residir en uno o más archivos fuente. Los archivos fuente se pueden compilar por separado y cargar juntos, junto con las funciones compiladas previamente de las bibliotecas. Sin embargo, no entraremos en ese proceso aquí, ya que los detalles varían de un sistema a otro.

La declaración y definición de funciones es el área en la que el estándar ANSI ha realizado la mayoría de los cambios en C. Como vimos por primera vez en el Capítulo 1, ahora es posible declarar el tipo de argumentos cuando se declara una función. La sintaxis de la declaración de función también cambia, de modo que las declaraciones y definiciones coinciden. Esto hace posible que un compilador detecte muchos más errores de los que podía antes. Además, cuando los argumentos se declaran correctamente, se realizan automáticamente las coerciones de tipos adecuadas.

La norma aclara las reglas sobre el alcance de los nombres; En particular, requiere que haya una sola definición de cada objeto externo. La inicialización es más general: ahora se pueden inicializar matrices y estructuras automáticas.

El preprocesador C también ha sido mejorado. Las nuevas instalaciones de preprocesador incluyen un conjunto más completo de directivas de compilación condicional, una forma de crear cadenas entrecomilladas a partir de argumentos de macro y un mejor control sobre el proceso de expansión de macros.

4.1 Conceptos básicos de las funciones

Para empezar, diseñemos y escribamos un programa para imprimir cada línea de su entrada que contenga un "patrón" o cadena de caracteres en particular. (Este es un caso especial del programa UNIX `grep`.) Por ejemplo, buscar el patrón de letras "ould" en el conjunto de líneas

```
¡Ah amor! ¿Podríamos tú y yo conspirar con el
Destino para aprehender este triste Esquema
de las Cosas en su totalidad, ¿No lo
romperíamos en pedazos y luego lo volveríamos
a moldear más cerca del Deseo del Corazón?
```

producirá la salida

```
¡Ah amor! ¿No podríamos tú y yo conspirar
con el Destino? ¿No lo romperíamos en
pedazos y luego lo volveríamos a moldear
más cerca del Deseo del Corazón?
```

El trabajo se divide perfectamente en tres partes:

```
while (hay otra línea)
    Si (la línea contiene el patrón)
        imprímelo
```

Aunque ciertamente es posible poner el código para todo esto en `main`, una mejor manera es usar la estructura a su favor haciendo que cada parte sea una función separada. Es mejor tratar con tres piezas pequeñas que con una grande, porque los detalles irrelevantes pueden quedar enterrados en las funciones y se minimiza la posibilidad de interacciones no deseadas. Y las piezas pueden ser incluso útiles en otros programas.

"While there's another line" es `getline`, una función que escribimos en el Capítulo 1, y "print it" es `printf`, que alguien ya nos ha proporcionado. Esto significa que solo necesitamos escribir una rutina para decidir si la línea contiene una ocurrencia del patrón.

Podemos resolver ese problema escribiendo una función `strindex(s,t)` que devuelva la posición o el índice en la cadena `s` donde comienza la cadena `t` , o `-1` si `s` no contiene `t`. Dado que las matrices C comienzan en la posición cero, los índices serán cero o positivos, por lo que un valor negativo como `-1` es conveniente para señalar un error. Cuando más adelante necesitemos una coincidencia de patrones más sofisticada, solo tenemos que reemplazar `strindex`; el resto del código puede permanecer igual. (La biblioteca estándar proporciona una función `strstr` que es similar a `strindex`, excepto que devuelve un puntero en lugar de un índice).

Teniendo en cuenta tanto diseño, rellenar los detalles del programa es sencillo. Aquí está todo, para que veas cómo encajan las piezas. Por ahora, el patrón que se va a buscar es una cadena literal, que no es el más general de los mecanismos. Volveremos en breve a una discusión sobre cómo inicializar matrices de caracteres, y en el Capítulo 5 mostraremos cómo hacer que el patrón sea un parámetro que se establece cuando se ejecuta el programa. También hay una versión ligeramente diferente de `getline`; puede que le resulte instructivo compararla con la del Capítulo 1.

```
#include <stdio.h>
#define MAXLINE 1000 /* longitud máxima de la línea de entrada */

tú getline(char line[], eres max)
int strindex(char fuente[], char searchfor[]);

patrón de char[] = "ould"; /* patrón a buscar */

/* encontrar todas las líneas que
coincidan con el patrón */ main()
{
    línea de
    caracteres[MAXLINE];
    int encontrado = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(línea, patrón) >= 0) {
            printf("%s", línea);
            encontrado++;
        }
    devolución encontrada;
}

/* getline: Poner la línea en s, devolver la
longitud */ int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
```

```
        s[i] = '\0';
        Retorno I;
    }

    /* strindex: devuelve el índice de t en s, -1 si no
    hay ninguno */ int strindex(char s[], char t[])
    {
        int i, j, k;

        for (i = 0; s[i] != '\0'; i++) {
            para (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
                ;
            if (k > 0 && t[k] == '\0')
                devuelve i;
        }
        retorno -1;
    }
```

Cada definición de función tiene la forma

```
return-type nombre-función(declaraciones de argumentos)
{
    Declaraciones y comunicados
}
```

Varias partes pueden estar ausentes; Una función mínima es

```
muñeco() {}
```

que no hace nada y no devuelve nada. Una función de no hacer nada como esta a veces es útil como marcador de posición durante el desarrollo del programa. Si se omite el tipo de valor devuelto, `se asume` int.

Un programa no es más que un conjunto de definiciones de variables y funciones. La comunicación entre las funciones se realiza mediante argumentos y valores devueltos por las funciones, y a través de variables externas. Las funciones pueden aparecer en cualquier orden en el archivo de código fuente, y el programa de origen se puede dividir en varios archivos, siempre y cuando no se divida ninguna función.

La instrucción `return` es el mecanismo para devolver un valor de la función llamada a su llamador. Cualquier expresión puede seguir `a return`:

```
expresión de retorno;
```

La *expresión* se convertirá al tipo de valor devuelto de la función si es necesario. Los paréntesis se utilizan a menudo alrededor de la *expresión*, pero son opcionales.

La función de llamada es libre de ignorar el valor devuelto. Además, no es necesario que haya ninguna expresión después `de la devolución`; en ese caso, no se devuelve ningún valor al autor de la llamada. El control también regresa al autor de la llamada sin valor cuando la ejecución "se cae del final" de la función al alcanzar la llave derecha de cierre. No es ilegal, pero probablemente sea una señal de problemas, si una función devuelve un valor de un lugar y ningún valor de otro. En cualquier caso, si una función no puede devolver un valor, su
El "valor" seguramente será basura.

El programa de búsqueda de patrones devuelve un estado de `main`, el número de coincidencias encontradas. Este valor está disponible para su uso por parte del entorno que llamó al programa

La mecánica de cómo compilar y cargar un programa C que reside en múltiples archivos fuente varía de un sistema a otro. En el sistema UNIX, por ejemplo, el comando `cc` mencionado en el Capítulo 1 hace el trabajo. Supongamos que las tres funciones se almacenan en tres archivos denominados `main.c`, `getline.c` y `strindex.c`. A continuación, el comando

```
cc main.c getline.c strindex.c
```

Compila los tres archivos, colocando el código objeto resultante en los archivos `main.o`, `getline.o` y `strindex.o`, y luego los carga todos en un archivo ejecutable llamado `a.out`. Si hay un error, por ejemplo en `main.c`, el archivo se puede recompilar por sí mismo y el resultado se puede cargar con los archivos objeto anteriores, con el comando

```
cc main.c getline.o strindex.o
```

El comando `cc` utiliza la convención de `nomenclatura ''.c`" frente a "`.o`" para distinguir los archivos de origen de los archivos de objeto.

Ejercicio 4-1. Escriba la función `strindex(s,t)` que devuelve la posición del ocurrencia de `t` en `s`, o `-1` si no hay ninguno.

4.2 Funciones que devuelven números no enteros

Hasta ahora, nuestros ejemplos de funciones no han devuelto ningún valor (`void`) o un `int`. ¿Qué pasa si una función debe devolver algún otro tipo? Muchas funciones numéricas como `sqrt`, `sin` y `cos` devuelven `double`; otras funciones especializadas devuelven otros tipos. Para ilustrar cómo lidiar con esto, escribamos y usemos la función `atof(s)`, que convierte la cadena `s` en su equivalente de punto flotante de doble precisión. `atof` es una extensión de `atoi`, de la que mostramos versiones en los capítulos 2 y 3. Maneja un signo opcional y un punto decimal, y la presencia o ausencia de parte o fracción de parte. Nuestra versión *no es* una rutina de conversión de entrada de alta calidad; eso tomaría más espacio del que nos gustaría usar. La biblioteca estándar incluye un `atof`; el encabezado `<stdlib.h>` lo declara.

En primer lugar, `atof` debe declarar el tipo de valor que devuelve, ya que no es `int`. El nombre del tipo precede al nombre de la función:

```
#include <tipo.h>

/* atof: convierte la cadena s en double
*/ double atof(char s[])
{
    doble val, poder;
    int i, signo;

    for (i = 0; isspace(s[i]); i++) /* omitir espacio en blanco */
        ;
    signo = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        potencia *= 10;
    }
    signo de retorno * val / poder;
}
```

En segundo lugar, e igual de importante, la rutina de llamada debe saber que `atof` devuelve un valor que no es int. Una forma de garantizar esto es declarar `atof` explícitamente en la rutina de llamada. La declaración se muestra en esta calculadora primitiva (apenas adecuada para el equilibrio de la chequera), que dice

Un número por línea, opcionalmente precedido por un signo, y los suma, imprimiendo la suma acumulada después de cada entrada:

```
#include <stdio.h>

#define MAXLINE 100

/* calculadora rudimentaria */
main()
{
    suma doble, atof(char []);
    línea de
    caracteres[MAXLINE];
    obtienes línea(línea de caracteres[], eres máximo);

    suma = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    devuelve 0;
}
```

La declaración

```
suma doble, atof(char []);
```

Dice que `sum` es una `variable doble`, y que `atof` es una función que toma un argumento `char[]` y devuelve un doble.

La función `atof` debe declararse y definirse de forma coherente. Si `atof` itself y la llamada a él en `main` tienen tipos inconsistentes en el mismo archivo fuente, el error será detectado por el compilador. Pero si (como es más probable) `atof` se compilara por separado, la discordancia no se detectaría, `atof` devolvería un `doble` que `main` trataría como un `int`, y se producirían respuestas sin sentido.

A la luz de lo que hemos dicho acerca de cómo las declaraciones deben coincidir con las definiciones, esto podría parecer sorprendente. La razón por la que puede ocurrir una discordancia es que si no hay un prototipo de función, una función se declara implícitamente por su primera aparición en una expresión, como

```
suma += atof(línea)
```

Si un nombre que no se ha declarado previamente aparece en una expresión y va seguido de un paréntesis izquierdo, el contexto lo declara como un nombre de función, se supone que la función devuelve un `int` y no se supone nada sobre sus argumentos. Además, si una declaración de función no incluye argumentos, como en

```
doble atof();
```

Esto también significa que no se debe asumir nada sobre los argumentos de `ATOF`; todas las comprobaciones de parámetros están desactivadas. Este significado especial de la lista de argumentos vacía está destinado a permitir que los programas C más antiguos se compilen con compiladores nuevos. Pero es una mala idea usarlo con nuevos programas en C. Si la función toma argumentos, declárelos; Si no acepta argumentos, use `void`.

Dado `atof`, correctamente declarado, podríamos escribir `atoi` (convertir una cadena en `int`) en términos de ello:

```
/* atoi: convierte la cadena s a entero usando atof
*/ int atoi(char s[])
{
    doble atof(char s[]);
```

```
    retorno (int) atof(s);
}
```

Observe la estructura de las declaraciones y la instrucción return. El valor de la expresión en

```
expresión de retorno;
```

se convierte en el tipo de la función antes de que se tome la devolución. Por lo tanto, el valor de `atof`, un `doble`, se convierte automáticamente en `int` cuando aparece en este `retorno`, ya que la función `atoi` devuelve un `int`. Sin embargo, esta operación descarta potencialmente la información, por lo que algunos compiladores advierten de ella. La conversión indica explícitamente que la operación está prevista y suprime cualquier advertencia.

Ejercicio 4-2. Extender `atof` para manejar la notación científica de la forma

```
123.45e-6
```

donde un número de coma flotante puede ir seguido de `e` o `e` y un exponente con signo opcional.

4.3 Variables externas

Un programa C consiste en un conjunto de objetos externos, que son variables o funciones. El adjetivo "externo" se usa en contraste con "interno", que describe los argumentos y variables definidos dentro de las funciones. Las variables externas se definen fuera de cualquier función y, por lo tanto, están potencialmente disponibles para muchas funciones. Las funciones en sí mismas son siempre externas, porque C no permite que las funciones se definan dentro de otras funciones. De forma predeterminada, las variables y funciones externas tienen la propiedad de que todas las referencias a ellas con el mismo nombre, incluso las de funciones compiladas por separado, son referencias a lo mismo. (El estándar llama a esta propiedad *Vinculación externa*). En este sentido, las variables externas son análogas a los bloques COMMON de Fortran o a las variables del bloque más externo de Pascal. Más adelante veremos cómo definir variables y funciones externas que solo son visibles dentro de un único archivo fuente. Dado que las variables externas son accesibles globalmente, proporcionan una alternativa a los argumentos de función y devuelven valores para comunicar datos entre funciones. Cualquier función puede acceder a una variable externa refiriéndose a ella por su nombre, si el nombre ha sido declarado de alguna manera.

Si se debe compartir un gran número de variables entre funciones, las variables externas son más convenientes y eficientes que las largas listas de argumentos. Sin embargo, como se señaló en el Capítulo 1, este razonamiento debe aplicarse con cierta precaución, ya que puede tener un efecto negativo en la estructura del programa y conducir a programas con demasiadas conexiones de datos entre funciones.

Las variables externas también son útiles debido a su mayor alcance y vida útil. Las variables automáticas son internas a una función; Nacen cuando se ingresa a la función y desaparecen cuando se deja. Las variables externas, por otro lado, son permanentes, por lo que pueden retener valores de una invocación de función a la siguiente. Por lo tanto, si dos funciones deben compartir algunos datos, pero ninguna llama a la otra, a menudo es más conveniente si los datos compartidos se mantienen en variables externas en lugar de pasarse dentro y fuera a través de argumentos.

Examinemos este tema con un ejemplo más amplio. El problema es escribir un programa de cálculo que proporcione los operadores `+`, `-`, `*` y `/`. Debido a que es más fácil de implementar, la calculadora usará la notación polaca inversa en lugar de infijo. (La notación polaca inversa es utilizada por algunas calculadoras de bolsillo y en idiomas como Forth y Postscript).

En la notación polaca inversa, cada operador sigue a sus operandos; una expresión infija como

```
(1 - 2) * (4 + 5)
```

se introduce como

```
1 2 - 4 5 + *
```

Los paréntesis no son necesarios; La notación es inequívoca siempre y cuando sepamos cuántos operandos espera cada operador.

La implementación es sencilla. Cada operando se inserta en una pila; Cuando llega un operador, se extrae el número adecuado de operandos (dos para operadores binarios), se les aplica el operador y el resultado se devuelve a la pila. En el ejemplo anterior, por ejemplo, 1 y 2 se empujan y luego se reemplazan por su diferencia, -1. A continuación, se empujan 4 y 5 y luego se reemplazan por su suma, 9. El producto de -1 y 9, que es -9, los reemplaza en la pila. El valor en la parte superior de la pila se extrae e imprime cuando se encuentra el final de la línea de entrada.

La estructura del programa es, por lo tanto, un bucle que realiza la operación adecuada en cada operador y operando a medida que aparece:

```
while (el operador u operando siguiente no es el indicador de
    fin de archivo) if (número)
        empújalo
    de lo contrario,
        si los
        operandos pop
        (operador)
        realizan la
        operación push
        result
    else if (nueva línea)
        Pop e impresión en la parte superior de la pila
    más
        error
```

La operación de empujar y hacer estallar una pila es trivial, pero para cuando se agregan la detección y recuperación de errores, son lo suficientemente largas como para que sea mejor poner cada una en una función separada que repetir el código a lo largo de todo el programa. Y debe haber una función separada para obtener el siguiente operador u operando de entrada.

La principal decisión de diseño que aún no se ha discutido es dónde está la pila, es decir, qué rutinas acceden a ella directamente. Lo más posible es mantenerlo en `main`, y pasar la pila y la posición actual de la pila a las rutinas que lo empujan y lo hacen estallar. Pero `main` no necesita conocer las variables que controlan la pila; solo realiza operaciones de empuje y pop. Por lo tanto, hemos decidido almacenar la pila y su información asociada en variables externas accesibles a las funciones `push` y `pop`, pero no a `main`.

Traducir este esquema a código es bastante fácil. Si por ahora pensamos que el programa existe en un archivo fuente, se verá así:

```
#includes
#defines
```

Declaraciones de función para

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

Variables externas para `push` *y* `pop`

```
void empuje( doble f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }
```

Rutinas convocadas por `Getop`

Más adelante discutiremos cómo se podría dividir esto en dos o más archivos fuente.

La función `principal` es un bucle que contiene un `gran interruptor` en el tipo de operador u operando; este es un uso más típico del `interruptor` que el que se muestra en la Sección 3.4.

```
#include <stdio.h>
#include <stdlib.h> /* para atof() */

#define MAXOP100 /* tamaño máximo del operando u operador
*/ #define NUMBER '0' /* señal de que se ha encontrado un
número */

int getop(char []);
empuje de vacío
(doble); doble pop
(vacío);

/* calculadora polaca inversa */
main()
{
    tipo int;
    Doble OP2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        NÚMERO DE CASO:
            empuje(atof(s))
            ; quebrar;
        Caso '+':
            empuje (pop() +
            pop()); quebrar;
        Caso '*':
            empuje(pop() * pop());
            quebrar;
        caso '-':
            op2 = pop();
            empuje (pop() -
            op2); Quebrar;
        caso '/':
            op2 = pop();
            if (op2 != 0.0)
                empuje (pop() /
            op2); Más
                printf("error: divisor cero\n");
            quebrar;
        Caso '\n':
            printf("\t%.8g\n", pop());
            quebrar;
        predeterminado:
            printf("error: comando desconocido %s\n",
            s); quebrar;
        }
    }
    devuelve 0;
}
```

Dado que + y * son operadores conmutativos, el orden en el que se combinan los operandos saltados es irrelevante, pero para - y / se deben distinguir los operandos izquierdo y derecho. En

```
empuje(pop() - pop()); /*INCORRECTO*/
```

El orden en el que se evalúan las dos llamadas de pop no está definido. Para garantizar el orden correcto, es necesario colocar el primer valor en una variable temporal como hicimos en main.

```
#define MAXVAL 100 /* profundidad máxima de la pila de val */

int sp = 0;/* siguiente posición de pila libre */
double val[MAXVAL]; /* pila de valores */

/* empuje: empuje f en la pila de
valores * / empuje nulo (doble f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    más
        printf("error: pila llena, no se puede empujar %g\n", f);
}

/* pop: pop y devuelve el valor superior de la
pila */ double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: pila vacía\n");
        devuelve 0,0;
    }
}
```

Una variable es externa si se define fuera de cualquier función. Por lo tanto, la pila y el índice de pila que deben ser compartidos por push y pop se definen fuera de estas funciones. Pero main en sí no se refiere a la pila o a la posición de la pila: la representación se puede ocultar.

Pasemos ahora a la implementación de getop, la función que obtiene el siguiente operador u operando. La tarea es fácil. Omita los espacios en blanco y las pestañas. Si el siguiente carácter no es un dígito o un punto hexadecimal, devuélvalo. De lo contrario, recopile una cadena de dígitos (que podría incluir un punto decimal) y devuelva NUMBER, la señal de que se ha recopilado un número.

```
#include <tipo.h>

int getch(vacío);
void ungetch(int);

/* getop: obtener el siguiente carácter u operando
numérico */ int getop(char s[])
{
    int I, C;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;/* no es un número */ i
    = 0;
    if (isdigit(c))/* recoge la parte entera */
        while (isdigit(s[++i] = c = getch()))

            ;
    if (c == '.')      /* recoger la parte de la fracción */
```

```
        while (isdigit(s[++i] = c = getch()))
            ;
        s[i] = '\0';
        si (c != EOF)
            ungetch(c);
        devolver NÚMERO;
    }
```

¿Qué son `getch` y `ungetch`? A menudo se da el caso de que un programa no puede determinar que ha leído suficiente entrada hasta que ha leído demasiado. Un ejemplo es la recopilación de caracteres que componen un número: hasta que no se ve el primer no dígito, el número no está completo. Pero entonces el programa ha leído un carácter demasiado lejos, un carácter para el que no está preparado.

El problema se resolvería si fuera posible "des-leer" el carácter no deseado. Entonces, cada vez que el programa leía un carácter de más, podía empujarlo hacia atrás en la entrada, por lo que el resto del código podría comportarse como si nunca se hubiera leído. Afortunadamente, es fácil simular la obtención de un personaje, escribiendo un par de funciones cooperativas. `getch` entrega el siguiente carácter de entrada que se va a tener en cuenta; `ungetch` los devolverá antes de leer la nueva entrada.

La forma en que trabajan juntos es simple. `ungetch` coloca los caracteres empujados hacia atrás en un búfer compartido, una matriz de caracteres. `getch` lee del búfer si hay algo más y llama a `getchar` si el búfer está vacío. También debe haber una variable de índice que registre la posición del carácter actual en el búfer.

Dado que getch `y` ungetch `comparten el búfer y el índice` y deben conservar sus valores entre llamadas, deben ser externos a ambas rutinas. Por lo tanto, podemos escribir `getch`, `ungetch` y sus variables compartidas como:

```
#define BUFSIZE 100

char buf[BUFSIZE];     /* búfer para ungetch */
int bufp = 0;          /* Próxima posición libre en BUF */

int getch(void) /* obtener un carácter (posiblemente empujado hacia atrás)
*/
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)    /* empujar el carácter hacia atrás en la entrada */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: demasiados caracteres\n");
    más
        buf[bufp++] = c;
}
```

La biblioteca estándar incluye una función `ungetch` que proporciona un carácter de retroceso; lo discutiremos en el Capítulo 7. Hemos utilizado una matriz para el retroceso, en lugar de un solo carácter, para ilustrar un enfoque más general.

Ejercicio 4-3. Dado el marco básico, es sencillo ampliar la calculadora. Agregue el operador de módulo (%) y las provisiones para números negativos.

Ejercicio 4-4. Agregue los comandos para imprimir los elementos superiores de la pila sin que salten, para duplicarla e intercambiar los dos elementos superiores. Agregue un comando para borrar la pila.

Ejercicio 4-5. Agregue acceso a funciones de biblioteca como `sin`, `exp` y `pow`. Véase <math.h> en el Apéndice B, Sección 4.

Ejercicio 4-6. Agregue comandos para manejar variables. (Es fácil proporcionar veintiséis variables con nombres de una sola letra). Agregue una variable para el valor impreso más recientemente.

Ejercicio 4-7. Escriba una rutina `ungets(s)` que empuje una cadena completa a la entrada. ¿Debería `ungets` saber sobre `buf` y `bufp`, o debería simplemente usar `ungetch`?

Ejercicio 4-8. Supongamos que nunca habrá más de un carácter de retroceso. Modificar `getch` y `ungetch` en consecuencia.

Ejercicio 4-9. Nuestro `getch` y `ungetch` no manejan correctamente un `EOF` empujado hacia atrás. Decida cuáles deberían ser sus propiedades si se retrasa un `EOF` y, a continuación, implemente su diseño.

Ejercicio 4-10. Una organización alternativa usa `getline` para leer una línea de entrada completa; esto hace que
`getch` y `ungetch` innecesarios. Revise la calculadora para usar este enfoque.

4.4 Reglas de alcance

No es necesario que todas las funciones y variables externas que componen un programa C se compilen todas al mismo tiempo; El texto fuente del programa puede guardarse en varios archivos, y las rutinas previamente compiladas pueden cargarse desde las bibliotecas. Entre las preguntas de interés se encuentran:

- ¿Cómo se escriben las declaraciones para que las variables se declaren correctamente durante la compilación?
- ¿Cómo se organizan las declaraciones para que todas las piezas estén correctamente conectadas cuando se cargue el programa?
- ¿Cómo se organizan las declaraciones para que solo haya una copia?
- ¿Cómo se inicializan las variables externas?

Analicemos estos temas reorganizando el programa de la calculadora en varios archivos. En la práctica, la calculadora es demasiado pequeña para que valga la pena dividirla, pero es una buena ilustración de los problemas que surgen en los programas más grandes.

El *ámbito* de un nombre es la parte del programa dentro de la cual se puede utilizar el nombre. En el caso de una variable automática declarada al principio de una función, el ámbito es la función en la que se declara el nombre. Las variables locales del mismo nombre en diferentes funciones no están relacionadas. Lo mismo ocurre con los parámetros de la función, que son en efecto variables locales.

El ámbito de una variable externa o una función dura desde el punto en el que se declara hasta el final del archivo que se está compilando. Por ejemplo, si `main`, `sp`, `val`, `push` y `pop` se definen en un archivo, en el orden mostrado anteriormente, es decir,

```
main() { ... }

int sp = 0;
doble val[MAXVAL];

void empuje(doble f) { ... }

double pop(void) { ... }
```

Entonces, las variables `sp` y `val` se pueden usar en `push` y `pop` simplemente nombrándolas; no se necesitan más declaraciones. Pero estos nombres no son visibles en `general`, ni tampoco lo son `el push` and `pop` en sí.

Por otro lado, si se va a hacer referencia a una variable externa antes de definirla, o si se define en un archivo fuente diferente al que se está utilizando, entonces una declaración `extern` es obligatoria.

Es importante distinguir entre la *declaración* de una variable externa y su *definición*. Una declaración anuncia las propiedades de una variable (principalmente su tipo); Una definición también hace que el almacenamiento se reserve. Si las líneas

```
int sp;
doble val[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `sp` y `val`, hacen que el almacenamiento se reserve y también sirven como declaraciones para el resto de ese archivo de origen. Por otro lado, las líneas

```
extern int sp;
extern doble val[];
```

Declaran para el resto del archivo fuente que `sp` es un `int` y que `val` es una `matriz doble` (cuyo tamaño se determina en otra parte), pero no crean las variables ni reservan almacenamiento para ellas.

Solo debe haber una *definición* de una variable externa entre todos los archivos que componen el programa fuente; otros archivos pueden contener declaraciones externas para acceder a ella. (También puede haber `extern` en el archivo que contiene la definición). Los tamaños de matriz deben especificarse con la definición, pero son opcionales con una `declaración extern`.

La inicialización de una variable externa va solo con la definición.

Aunque no es una organización probable para este programa, las funciones `push` y `pop` podrían definirse en un archivo, y las variables `val` y `sp` definirse e inicializarse en otro. Entonces estas definiciones y declaraciones serían necesarias para unirlos:

En el archivo 1:

```
extern int sp;
extern doble val[];

void empuje(doble f) { ... }

double pop(void) { ... }
```

En el archivo 2:

```
int sp = 0;
doble val[MAXVAL];
```

Dado que las `declaraciones extern` de *file1* se encuentran delante y fuera de las definiciones de función, se aplican a todas las funciones; un conjunto de declaraciones es suficiente para todo *file1*. Esta misma organización también sería necesaria si la definición de `sp` y `val` siguiera su uso en un solo archivo.

4.5 Archivos de encabezado

Consideremos ahora dividir el programa de calculadora en varios archivos fuente, ya que podría ser si cada uno de los componentes fuera sustancialmente más grande. La `función principal` iría en un archivo, al que llamaremos `main.c`; `push`, `pop` y sus variables van a un segundo archivo, `stack.c`; `Getop` entra en una tercera, `Getop.C`. Finalmente, `getch` y `ungetch` van a un cuarto archivo, `getch.c`; los separamos de los demás porque provendrían de una biblioteca compilada por separado en un programa realista.

Hay una cosa más de la que preocuparse: las definiciones y declaraciones compartidas entre los archivos. En la medida de lo posible, queremos centralizar esto, de modo que solo haya una copia para obtener y conservar a medida que el programa evoluciona. En consecuencia, colocaremos este material común en un *archivo de cabecera*, `calc.h`, que se incluirá según sea necesario. (El `#include` línea se describe en la sección 4.11.) El programa resultante tiene el siguiente aspecto:

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Existe una disyuntiva entre el deseo de que cada archivo tenga acceso solo a la información que necesita para su trabajo y la realidad práctica de que es más difícil mantener más archivos de encabezado. Hasta un tamaño de programa moderado, probablemente sea mejor tener un archivo de encabezado que contenga

todo lo que se va a compartir entre dos partes cualesquiera del programa; Esa es la decisión que tomamos aquí. Para un programa mucho más grande, se necesitaría más organización y más encabezados.

4.6 Variables estáticas

Las variables `sp` y `val` en `stack.c`, y `buf` y `bufp` en `getch.c`, son para el uso privado de las funciones en sus respectivos archivos de código fuente, y no están destinadas a ser accedidas por nada más. La `declaración estática`, aplicada a una variable o función externa, limita el ámbito de ese objeto al resto del archivo de código fuente que se está compilando. Por lo tanto, la `estática externa` proporciona una forma de ocultar nombres como `buf` y `bufp` en la combinación `getch-ungetch`, que deben ser externos para que se puedan compartir, pero que no deben ser visibles para los usuarios de `getch` y `ungetch`.

El almacenamiento estático se especifica anteponiendo la palabra static a la declaración normal. Si las dos rutinas y las dos variables se compilan en un archivo, como en

```
char buf estático[BUFSIZE]; /* búfer para ungetch */
static int bufp = 0;/* siguiente posición libre en buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

Entonces ninguna otra rutina podrá acceder a `buf` y `bufp`, y esos nombres no entrarán en conflicto con los mismos nombres en otros archivos del mismo programa. De la misma manera, las variables que `push` y `pop` utilizan para la manipulación de la pila se pueden ocultar, declarando `que sp` y `val` son `estáticos`.

La declaración `estática externa` se usa con mayor frecuencia para variables, pero también se puede aplicar a funciones. Normalmente, los nombres de las funciones son globales, visibles para cualquier parte de todo el programa. Sin embargo, si una función se declara `estática`, su nombre es invisible fuera del archivo en el que se declara.

La declaración `estática` también se puede aplicar a variables internas. Las variables `estáticas internas` son locales para una función en particular al igual que las variables automáticas, pero a diferencia de las automáticas, permanecen en existencia en lugar de ir y venir cada vez que se activa la función. Esto significa que `las variables estáticas` internas proporcionan almacenamiento privado y permanente dentro de una sola función.

Ejercicio 4-11. Modifique `getop` para que no necesite usar `ungetch`. Sugerencia: use un archivo interno
`variable estática .`

4.7 Registrar variables

Una declaración de registro advierte al compilador de que la variable en cuestión se utilizará mucho. La idea es que `las variables de registro` se coloquen en registros de máquina, lo que puede resultar en programas más pequeños y rápidos. Pero los compiladores son libres de ignorar el consejo.

La declaración `de registro` tiene el siguiente aspecto:

```
registrar int x;
registrar el carácter
C;
```

y así sucesivamente. La declaración de registro solo se puede aplicar a las variables automáticas y a los parámetros formales de una función. En este último caso, parece que

```
f(registro sin signo m, registro long n)
{
    registro int i;
    ...
}
```

En la práctica, existen restricciones en las variables de registro, lo que refleja las realidades del hardware subyacente. Solo unas pocas variables de cada función se pueden mantener en registros, y solo se permiten ciertos tipos. Sin embargo, las declaraciones de registro excesivo son inofensivas, ya que la palabra `registro` se omite para las declaraciones en exceso o no permitidas. Y no es posible tomar la dirección de una variable de registro (un tema tratado en el Capítulo 5), independientemente de si la variable está realmente colocada en un registro. Las restricciones específicas sobre el número y los tipos de variables de registro varían de una máquina a otra.

4.8 Estructura de bloques

C no es un lenguaje estructurado en bloques en el sentido de Pascal o lenguajes similares, porque las funciones pueden no estar definidas dentro de otras funciones. Por otro lado, las variables se pueden definir de forma estructurada en bloques dentro de una función. Las declaraciones de variables (incluidas las inicializaciones) pueden seguir a la llave izquierda que introduce *cualquier* instrucción compuesta, no solo la que comienza una función. Las variables declaradas de esta manera ocultan cualquier variable con nombre idéntico en bloques externos y permanecen en existencia hasta la llave derecha coincidente. Por ejemplo, en

```
if (n > 0) {
    Int I; /* declarar una nueva i */

    para (i = 0; i < n; i++)
        ...
}
```

El ámbito de la variable `i` es la rama "verdadera" del `if`; este `i` no está relacionado con ningún `i` fuera del bloque. Una variable automática declarada e inicializada en un bloque se inicializa cada vez que se introduce el bloque.

Las variables automáticas, incluidos los parámetros formales, también ocultan las variables externas y las funciones del mismo nombre. Dadas las declaraciones

```
int x;
int y;

F (Doble X)
{
    doble y;
}
```

Luego, dentro de la función `f`, las ocurrencias de `x` se refieren al parámetro, que es un `doble`; fuera
`f`, se refieren al `int` externo. Lo mismo ocurre con la variable `y`.

Por una cuestión de estilo, es mejor evitar los nombres de variables que ocultan nombres en un ámbito externo; El potencial de confusión y error es demasiado grande.

4.9 Inicialización

La inicialización se ha mencionado de pasada muchas veces hasta ahora, pero siempre de forma periférica a algún otro tema. En esta sección se resumen algunas de las reglas, ahora que hemos analizado las distintas clases de almacenamiento.

En ausencia de inicialización explícita, se garantiza que las variables externas y estáticas se inicializarán en cero; Las variables automáticas y de registro tienen valores iniciales indefinidos (es decir, basura).

Las variables escalares se pueden inicializar cuando se definen, siguiendo el nombre con un signo igual y una expresión:

```
int x = 1;
char squota = '\'';
día largo = 1000L * 60L * 60L * 24L; /* milisegundos/día */
```

Para las variables externas y estáticas, el inicializador debe ser una expresión constante; la inicialización se realiza una vez, conceptualmente antes de que el programa comience a ejecutarse. Para las variables automáticas y de registro, el inicializador no se limita a ser una constante: puede ser cualquier expresión que implique valores previamente definidos, incluso llamadas a funciones. Por ejemplo, la inicialización del programa de búsqueda binaria en la Sección 3.3 podría escribirse como

```
int binsearch(int x, int v[], int n)
{
    int bajo = 0;
    int alto = n - 1;
    int mid;
    ...
}
```

En lugar de

```
    int bajo, alto,

    medio; bajo = 0;
    alto = n - 1;
```

En efecto, la inicialización de variables automáticas no es más que una abreviatura de las sentencias de asignación. Qué forma preferir es en gran medida una cuestión de gustos. Por lo general, hemos usado asignaciones explícitas, porque los inicializadores en las declaraciones son más difíciles de ver y están más alejados del punto de uso.

Una matriz se puede inicializar siguiendo su declaración con una lista de inicializadores encerrados entre llaves y separados por comas. Por ejemplo, para inicializar una matriz `days` con el número de días de cada mes:

```
int días[] = { 31, 28, 31, 30, 31, 31, 31, 30, 31, 31, 31, 31 }
```

Cuando se omite el tamaño de la matriz, el compilador calculará la longitud contando los inicializadores, de los cuales hay 12 en este caso.

Si hay menos inicializadores para una matriz que el tamaño especificado, los demás serán cero para las variables externas, estáticas y automáticas. Es un error tener demasiados inicializadores. No hay forma de especificar la repetición de un inicializador, ni de inicializar un elemento en medio de una matriz sin proporcionar también todos los valores anteriores.

Las matrices de caracteres son un caso especial de inicialización; Se puede usar una cadena en lugar de las llaves y las comas:

```
patrón de caracteres = "ould";
```

es una abreviatura de la más larga pero equivalente

```
patrón de caracteres[] = { 'o', 'u', 'l', 'd', '\0' };
```

En este caso, el tamaño de la matriz es cinco (cuatro caracteres más la terminación `'\0'`).

4.10 Recursión

Las funciones C se pueden usar de forma recursiva; Es decir, una función puede llamarse a sí misma directa o indirectamente. Considere la posibilidad de imprimir un número como una cadena de caracteres. Como mencionamos antes, los dígitos se generan en el orden incorrecto: los dígitos de orden inferior están disponibles antes que los dígitos de orden superior, pero deben imprimirse al revés.

Hay dos soluciones a este problema. On es almacenar los dígitos en una matriz a medida que se generan, luego imprimirlos en el orden inverso, como hicimos con `itoa` en la sección 3.6. La alternativa es una solución recursiva, en la que `printd` primero se llama a sí mismo para hacer frente a los dígitos iniciales y luego imprime el dígito final. De nuevo, esta versión puede fallar en el número negativo más grande.

```
#include <stdio.h>

/* printd: print n en decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    Si (N / 10)
        impreso (n / 10);
    putchar(n % 10 + '0');
}
```

Cuando una función se llama a sí misma de forma recursiva, cada invocación obtiene un nuevo conjunto de todas las variables automáticas, independientemente del conjunto anterior. Esto en `printd(123)` el primer `printd` recibe el argumento `n = 123`. Pasa `de 12` a un segundo `impreso`, que a su vez pasa de `1` a un tercero. El tercer nivel `impreso` imprime `1` y, a continuación, vuelve al segundo nivel. Esa `impresión imprime 2` y luego vuelve al primer nivel. Ese imprime `3` y termina.

Otro buen ejemplo de recursividad es quicksort, un algoritmo de clasificación desarrollado por C.A.R. Hoare en 1962. Dado un matriz, se elige un elemento y los demás se dividen en dos subconjuntos: los menores que el elemento de partición y los mayores o iguales que él. A continuación, el mismo proceso se aplica de forma recursiva a los dos subconjuntos. Cuando un subconjunto tiene menos de dos elementos, no necesita ninguna ordenación; Esto detiene la recursividad.

Nuestra versión de quicksort no es la más rápida posible, pero es una de las más sencillas. Usamos el elemento central de cada subarray para la partición.

```
/* qsort: ordenar v[izquierda]... v[derecha] en orden
creciente */ void qsort(int v[], int izquierda, int
derecha)
{
    int i, último;
    void swap(int v[], int i, int j);
```

```
    if (izquierda >= derecha) /* no hacer nada si la
        matriz contiene */ retorno;   /* menos de dos
        elementos */
    intercambio(v, izquierda, (izquierda + derecha)/2); /*
    mover partición elem */ último = izquierda;/* a v[0] */
    for (i = izquierda + 1; i <= derecha; i++) /*
        partición */ if (v[i] < v[izquierda])
            permuta(v, ++último, i);
    swap(v, left, last);/* restaurar la partición elem */
    qsort(v, left, last-1);
    qsort(v, último+1, derecha);
}
```

Hemos movido la operación de intercambio a un `intercambio` de funciones independiente porque ocurre tres veces en `qsort`.

```
/* swap: intercambia v[i] y v[j] */ void
swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j]       =
    temperatura;
}
```

La biblioteca estándar incluye una versión de `qsort` que puede ordenar objetos de cualquier tipo.

Es posible que la recursividad no proporcione ningún ahorro en el almacenamiento, ya que en algún lugar se debe mantener una pila de los valores que se están procesando. Tampoco será más rápido. Pero el código recursivo es más compacto y, a menudo, mucho más fácil de escribir y entender que el equivalente no recursivo. La recursividad es especialmente conveniente para estructuras de datos definidas recursivamente como árboles, veremos un buen ejemplo en la Sección 6.6.

Ejercicio 4-12. Adaptar las ideas de `printd` para escribir una versión recursiva de `itoa`; es decir, convertir un número entero en una cadena llamando a una rutina recursiva.

Ejercicio 4-13. Escriba una versión recursiva de la función `reverse(s)`, que invierte la cadena `s` en su lugar.

4.11 El preprocesador C

C proporciona ciertas facilidades de lenguaje por medio de un preprocesador, que es conceptualmente un primer paso separado en la compilación. Las dos características más utilizadas son `#include`, para incluir el contenido de un archivo durante la compilación, y `#define`, para reemplazar un token por una secuencia arbitraria de caracteres. Otras características descritas en esta sección incluyen la compilación condicional y las macros con argumentos.

4.11.1 Inclusión de archivos

La inclusión de archivos facilita el manejo de colecciones de #defines `y declaraciones (entre otras cosas). Cualquier línea de origen del formulario`

```
#include "nombre de archivo"
```

o

```
#include <nombre de archivo>
```

se reemplaza por el contenido del nombre de archivo del archivo. Si el nombre del *archivo* está entre comillas, la búsqueda del archivo generalmente comienza donde se encontró el programa

fuente; si no se encuentra allí, o si el nombre es

Encerrada en < y >, la búsqueda sigue una regla definida por la implementación para encontrar el archivo. Un archivo incluido puede contener `#include` líneas.

A menudo hay varias líneas de #include al principio de un archivo de código fuente, para incluir `instrucciones de #define` comunes y declaraciones `extern`, o para acceder a las declaraciones de prototipo de función para funciones de biblioteca desde encabezados como `<stdio.h>`. (Estrictamente hablando, no es necesario que sean archivos; los detalles de cómo se accede a los encabezados dependen de la implementación).

`#include` es la forma preferida de vincular las declaraciones para un programa grande. Garantiza que todos los archivos fuente se proporcionarán con las mismas definiciones y declaraciones de variables y, por lo tanto, elimina un tipo de error particularmente desagradable. Naturalmente, cuando se cambia un archivo incluido, todos los archivos que dependen de él deben volver a compilarse.

4.11.2 Sustitución de macros

Una definición tiene la forma

```
#define texto de reemplazo de nombre
```

Requiere una sustitución de macro del tipo más simple: las apariciones posteriores del nombre del token se reemplazarán por el texto de *reemplazo*. El nombre de una `#define` tiene la misma forma que el nombre de una variable; el texto de reemplazo es arbitrario. Normalmente, el texto de reemplazo es el resto de la línea, pero una definición larga puede continuar en varias líneas colocando un `\` al final de cada línea que se va a continuar. El ámbito de un nombre definido con `#define` es desde su punto de definición hasta el final del archivo de código fuente que se está compilando. Una definición puede utilizar definiciones anteriores. Las sustituciones solo se realizan para tokens y no tienen lugar dentro de cadenas entrecomilladas. Por ejemplo, si `YES` es un nombre definido, no habría ninguna sustitución en `printf("YES")` o en `YESMAN`.

Cualquier nombre puede definirse con cualquier texto de reemplazo. Por ejemplo

```
#define para siempre para (;;)  /* bucle infinito */
```

define una nueva palabra, `para siempre`, para un bucle infinito.

También es posible definir macros con argumentos, por lo que el texto de reemplazo puede ser diferente para diferentes llamadas de la macro. Por ejemplo, defina una macro llamada `max`:

```
#define máx. (A, B) (A) > (B) ? (A) : (B))
```

Aunque parece una llamada de función, el uso de `max` se expande en código en línea. Cada aparición de un parámetro formal (aquí `A` o `B`) será reemplazada por el argumento real correspondiente.
Por lo tanto, la línea

```
x = máx(p+q, r+s);
```

será reemplazado por la línea

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Siempre y cuando los argumentos se traten de manera consistente, esta macro servirá para cualquier tipo de datos; no hay necesidad de diferentes tipos de `max` para diferentes tipos de datos, como lo habría con las funciones.

Si examina la expansión de `max`, notará algunas trampas. Las expresiones se evalúan dos veces; Esto es malo si implican efectos secundarios como operadores de incremento o entrada y salida. Por ejemplo

```
max(i++, j++) /* INCORRECTO */
```

incrementará el más grande dos veces. También se debe tener cierto cuidado con los paréntesis para asegurarse de que se conserve el orden de evaluación; Considere lo que sucede cuando la macro

```
#define cuadrado(x) x * x /* INCORRECTO */
```

se invoca como `cuadrado(z+1)`.

No obstante, las macros son valiosas. Un ejemplo práctico proviene de `<stdio.h>`, en el que `getchar` y `putchar` a menudo se definen como macros para evitar la sobrecarga en tiempo de ejecución de una llamada de función por carácter procesado. Las funciones `de <ctype.h>` también se suelen implementar como macros.

Los nombres pueden ser indefinidos con `#undef`, generalmente para garantizar que una rutina sea realmente una función, no una macro:

```
#undef getchar

int getchar(void) { ... }
```

Los parámetros formales no se reemplazan dentro de las cadenas entrecomilladas. Sin embargo, si el nombre de un parámetro está precedido por un `#` en el texto de reemplazo, la combinación se expandirá en una cadena entrecomillada con el parámetro reemplazado por el argumento real. Esto se puede combinar con la concatenación de cadenas para crear, por ejemplo, una macro de impresión de depuración:

```
#define dprint(expr)    printf(#expr " = %g\n", expr)
```

Cuando se invoca esto, como en

```
dprint (x/y)
```

La macro se expande a

```
printf("x/y" " = &g\n", x/y);
```

y las cadenas están concatenadas, por lo que el efecto es

```
printf("x/y = &g\n", x/y);
```

Dentro del argumento real, cada `"` se reemplaza por `\"` y cada `\` por `\\`, por lo que el resultado es una constante de cadena legal.

El operador de preprocesador `##` proporciona una forma de concatenar argumentos reales durante la expansión de macros. Si un parámetro en el texto de reemplazo es adyacente a un `##`, el parámetro se reemplaza por el argumento real, se eliminan el `##` y el espacio en blanco circundante y se vuelve a escanear el resultado. Por ejemplo, la macro `paste` concatena sus dos argumentos:

```
#define pegar (anverso, reverso) anverso ## reverso
```

Por lo tanto , `Paste(name, 1)` crea el token `name1`.

Las reglas para los usos anidados de `##` son arcanas; se pueden encontrar más detalles en el Apéndice A.

Ejercicio 4-14. Defina una macro `swap(t,x,y)` que intercambie dos argumentos de tipo `t`. (La estructura de bloques ayudará).

4.11.3 Inclusión condicional

Es posible controlar el preprocesamiento en sí mismo con instrucciones condicionales que se evalúan durante el preprocesamiento. Esto proporciona una manera de incluir código de forma selectiva, en función del valor de las condiciones evaluadas durante la compilación.

La línea `#if` evalúa una expresión entera constante (que puede no incluir `constantes sizeof`, casts o `enum`). Si la expresión es distinta de cero, se incluyen las líneas siguientes hasta un `#endif` o `#elif` o `#else`. (La instrucción del preprocesador `#elif` es como `si fuera` así). La expresión `defined`(*name*) en un `#if` es 1 si el *nombre* ha sido definido y 0 en caso contrario.

Por ejemplo, para asegurarse de que el contenido de un archivo `hdr.h` se incluye solo una vez, el contenido del archivo se rodea con un condicional como este:

```
#if !defined (HDR)
#define HDR

/* Contenido de HDR.h Ir aquí */

#endif
```

La primera inclusión de `hdr.h` define el nombre `HDR`; las inclusiones posteriores encontrarán el nombre
definido y saltar a la `#endif`. Se puede usar un estilo similar para evitar incluir archivos varias veces. Si este estilo se utiliza de forma coherente, cada encabezado puede incluir a su vez cualquier otro encabezado del que dependa, sin que el usuario del encabezado tenga que lidiar con la interdependencia.

Esta secuencia prueba el nombre `SYSTEM` para decidir qué versión de un encabezado incluir:

```
#if SISTEMA == SYSV
    #define SISTEMA DE
#elif HDR "SYSV.H" == BSD
    #define SISTEMA DE
#elif HDR "BSD.H" == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Las líneas `#ifdef` y `#ifndef` son formularios especializados que comprueban si un nombre está definido. El primer ejemplo de `#if` anterior podría haber sido escrito

```
#ifndef HDR
#define HDR

/* Contenido de HDR.h Ir aquí */

#endif
```

Capítulo 5 - Punteros y matrices

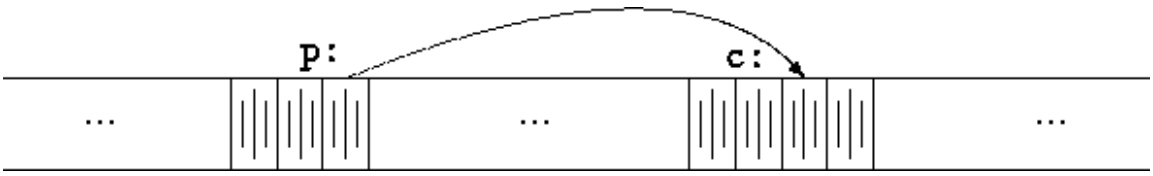
Un puntero es una variable que contiene la dirección de una variable. Los punteros se usan mucho en C, en parte porque a veces son la única forma de expresar un cálculo, y en parte porque generalmente conducen a un código más compacto y eficiente que el que se puede obtener de otras maneras. Los punteros y las matrices están estrechamente relacionados; Este capítulo también explora esta relación y muestra cómo explotarla.

Los punteros se han agrupado con la `declaración goto` como una forma maravillosa de crear programas imposibles de entender. Esto es ciertamente cierto cuando se usan descuidadamente, y es fácil crear punteros que apuntan a algún lugar inesperado. Sin embargo, con disciplina, los indicadores también se pueden usar para lograr claridad y simplicidad. Este es el aspecto que trataremos de ilustrar.

El principal cambio en ANSI C es hacer explícitas las reglas sobre cómo se pueden manipular los punteros, en efecto exigiendo lo que los buenos programadores ya practican y los buenos compiladores ya aplican. Además, el tipo `void *` (puntero a `void`) reemplaza `a char *` como el tipo adecuado para un puntero genérico.

5.1 Punteros y direcciones

Comencemos con una imagen simplificada de cómo se organiza la memoria. Una máquina típica tiene una matriz de celdas de memoria numeradas o direccionadas consecutivamente que se pueden manipular individualmente o en grupos contiguos. Una situación común es que cualquier byte puede ser un `char`, un par de celdas de un byte se puede tratar como un `entero corto` y cuatro bytes adyacentes forman un `largo`. Un puntero es un grupo de celdas (a menudo dos o cuatro) que pueden contener una dirección. Entonces, si `c` es un `carácter` y `p` es un puntero que apunta a él, podríamos representar la situación de esta manera:



El operador unario `&` da la dirección de un objeto, por lo que la declaración

```
p = &c;
```

asigna la dirección de `c` a la variable `p`, y se dice que p "apunta a" `c`. El operador `&` solo se aplica a los objetos en la memoria: variables y elementos de matriz. No se puede aplicar a expresiones, constantes o variables `de registro`.

El operador unario `*` es el operador de *direccionamiento indirecto* o *de desreferenciación*; cuando se aplica a un puntero, accede al objeto al que apunta el puntero. Supongamos que `x` e `y` son números enteros e `ip` es un puntero a `int`. Esta secuencia artificial muestra cómo declarar un puntero y cómo usar `&` y `*`:

```
int x = 1, y = 2, z[10];
int *ip;          /* ip es un puntero a int */
```

```
ip = &x;          /* ip ahora apunta a x */
y = *ip;          /* y ahora es 1 */
*IP = 0;          /* x ahora es 0 */
ip = &z[0];       /* ip ahora apunta a z[0] */
```

La declaración de `x`, `y` y `z` es lo que hemos visto todo el tiempo. La declaración del puntero `ip`,

```
int *ip;
```

Dice que la expresión `*ip` es un `int`. La sintaxis de la declaración de una variable imita la sintaxis de las expresiones en las que podría aparecer la variable. Este razonamiento también se aplica a las declaraciones de funciones. Por ejemplo

```
doble *dp, atof(char *);
```

dice que en una expresión `*dp` y `atof(s)` tienen valores de `double`, y que el argumento de `atof` es un puntero a `char`.

También debe tener en cuenta la implicación de que un puntero está restringido a apuntar a un tipo particular de objeto: cada puntero apunta a un tipo de datos específico. (Hay una excepción: un "puntero a `void`" se usa para contener cualquier tipo de puntero, pero no se puede desreferenciar a sí mismo. Volveremos sobre ello en la sección nº5.11.)

Si `ip` apunta al entero `x`, entonces `*ip` puede ocurrir en cualquier contexto donde `x` podría, por lo que

```
*IP = *IP + 10;
```

Incrementa `*IP` en 10.

Los operadores unarios `*` y `&` se unen más estrechamente que los operadores aritméticos, por lo que la asignación

```
y = *ip + 1
```

toma cualquier IP a la que apunte, suma 1 y asigna el resultado a `Y`, mientras que

```
*IP += 1
```

incrementa lo que apunta IP, al igual que

```
++*IP
```

y

```
(*ip)++
```

Los paréntesis son necesarios en este último ejemplo; sin ellos, la expresión incrementaría `ip` en lugar de lo que apunta, porque los operadores unarios como `*` y `++` se asocian de derecha a izquierda.

Por último, dado que los punteros son variables, se pueden utilizar sin desreferenciar. Por ejemplo, si
`iq` es otro puntero a `int`,

```
IQ = IP
```

Copia el contenido de IP en `IQ`, lo que hace que `IQ` apunte a cualquier `IP` a la que apunte.

5.2 Punteros y argumentos de función

Dado que C pasa argumentos a las funciones por valor, no hay una forma directa de que la función llamada modifique una variable en la función que llama. Por ejemplo, una rutina de ordenación podría intercambiar dos argumentos fuera de orden con una función llamada `swap`. No basta con escribir

```
permuta(a, b);
```

donde la función de `intercambio` se define como

```
void swap(int x, int y) /* INCORRECTO */
{
    int temp;

    temperatu
    ra = x; x
    = y;
    y = temperatura;
}
```

Debido a la llamada por valor, el `intercambio` no puede afectar a los argumentos `a` y `b` en la rutina que lo llamó. La función anterior intercambia *copias* de `a` y `b`.

La forma de obtener el efecto deseado es que el programa que llama pase *punteros* a los valores que se van a cambiar:

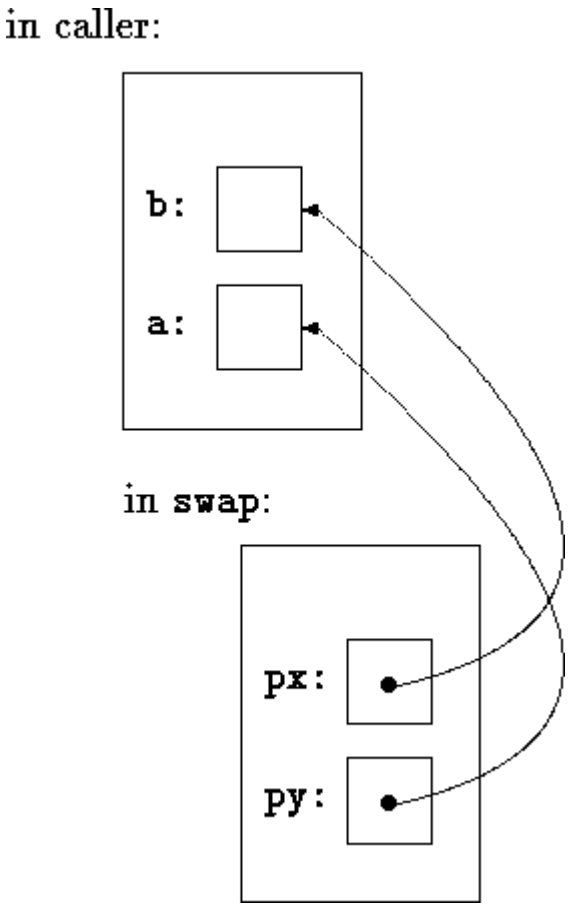
```
permuta(&a, &b);
```

Dado que el operador `&` produce la dirección de una variable, `&a` es un puntero a `a`. En el propio intercambio, los parámetros se declaran como punteros y se accede a ellos indirectamente a los operandos.

```
void swap(int *px, int *py) /* intercambio *px y *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temperatura;
}
```

Pictóricamente:



Los argumentos de puntero permiten que una función tenga acceso a los objetos de la función que la llamó y los cambie. Como ejemplo, considere una función `getint` que realiza la conversión de entrada de formato libre dividiendo una secuencia de caracteres en valores enteros, un entero por llamada. `getint` tiene que devolver el valor que encontró y también señalar el final del archivo cuando no hay más entradas. Estos valores deben devolverse por rutas de acceso separadas, ya que independientemente del valor que se use para `EOF`, también podría ser el valor de un entero de entrada.

Una solución es hacer que `getint` devuelva el estado de fin de archivo como su valor de función, mientras se usa un argumento de puntero para almacenar el entero convertido en la función de llamada. Este es también el esquema utilizado por `scanf` ; véase la Sección 7.4.

El siguiente bucle rellena una matriz con enteros mediante llamadas a `getint`:

```
int n, matriz[TAMAÑO], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

Cada llamada establece `array[n]` en el siguiente entero que se encuentra en la entrada e incrementa `n`. Tenga en cuenta que es esencial pasar la dirección de `array[n]` a `getint`. De lo contrario, no hay forma de `que getint` comunique el entero convertido al autor de la llamada.

Nuestra versión de `getint` devuelve `EOF` para el final del archivo, cero si la siguiente entrada no es un número y un valor positivo si la entrada contiene un número válido.

```
#include <tipo.h>
```

```
int getch(vacío);
void ungetch(int);

/* getint: obtener el siguiente entero de la
entrada en *pn */ int getint(int *pn)
{
    int c, signo;

    while (isspace(c = getch()))    /* Saltar espacio en blanco */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* no es un número */
        devuelve 0;
    }
    signo = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= signo;
    si (c != EOF)
        ungetch(c);
    Retorno C;
}
```

A lo largo de `getint`, `*pn` se utiliza como una variable `int` ordinaria. También hemos usado `getch` y `ungetch` (descritos en la Sección 4.3) para que el carácter adicional que debe leerse se pueda empujar de nuevo a la entrada.

Ejercicio 5-1. Tal como está escrito, `getint` trata un `+` o `-` no seguido de un dígito como una representación válida de cero. Corríjalo para empujar dicho carácter de nuevo en la entrada.

Ejercicio 5-2. Escribe `getfloat`, el análogo de punto flotante de `getint`. ¿Qué tipo `getfloat` return como su valor de función?

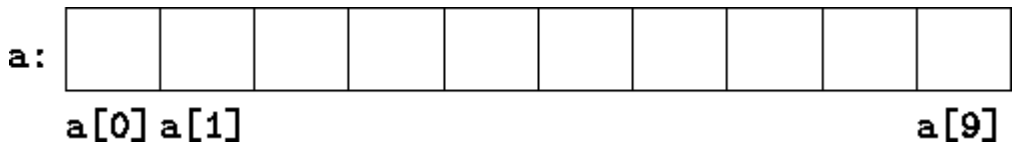
5.3 Punteros y matrices

En C, hay una fuerte relación entre punteros y matrices, lo suficientemente fuerte como para que los punteros y las matrices se discutan simultáneamente. Cualquier operación que se pueda lograr mediante subíndices de matriz también se puede realizar con punteros. La versión del puntero será en general más rápida pero, al menos para los no iniciados, algo más difícil de entender.

La declaración

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

Define una matriz de tamaño 10, es decir, un bloque de 10 objetos consecutivos denominados `A[0]`, `A[1]`, ...,`a[9]`.



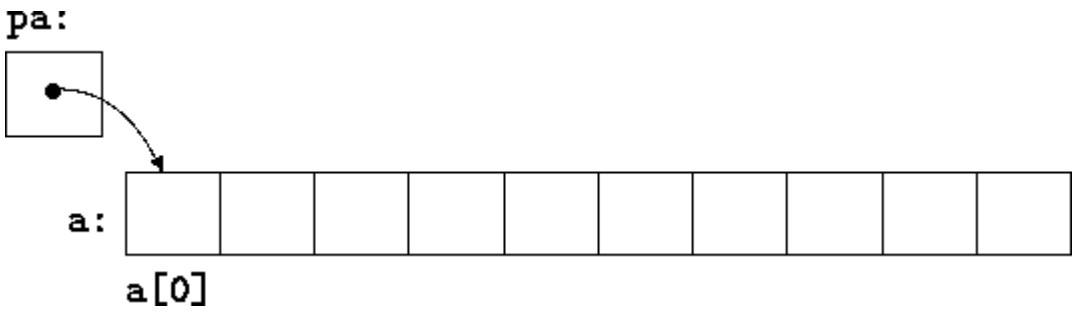
La notación `a[i]` se refiere al i-ésimo elemento de la matriz. Si `pa` es un puntero a un entero, se declara como

```
int *pa;
```

A continuación, la asignación

```
pa = &a[0];
```

Establece `pa` para que apunte al elemento cero de `a`; es decir, `pa` contiene la dirección de `a[0]`.



Ahora la tarea

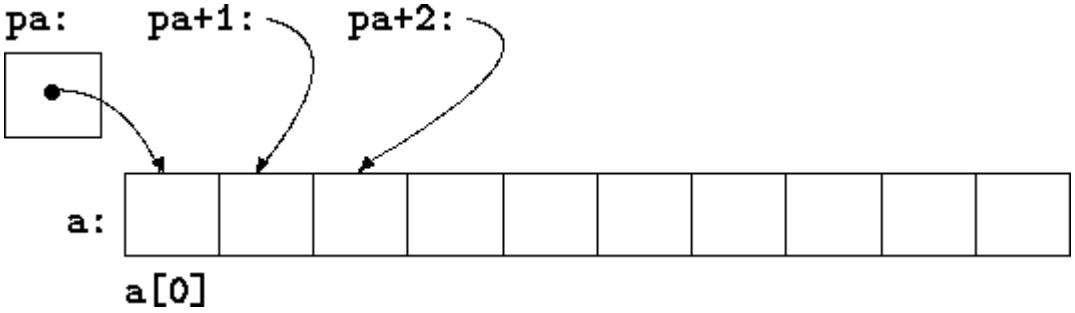
```
x = *pa;
```

copiará el contenido de `a[0]` en `x`.

Si `pa` apunta a un elemento particular de una matriz, entonces, por definición, `pa+1` apunta al siguiente elemento, `pa+i` señala `i` elementos después de `pa`, y `pa-i` señala `i` elementos antes. Por lo tanto, si `pa` señala `a[0]`,

```
*(PA+1)
```

se refiere al contenido de `a[1]`, `pa+i` es la dirección de `a[i]` y `*(pa+i)` es el contenido de `a[i]`.



Estas observaciones son verdaderas independientemente del tipo o tamaño de las variables de la matriz `a`. El significado de "agregar 1 a un puntero", y por extensión, toda la aritmética de punteros, es que `pa+1` apunta al siguiente objeto, y `pa+i` apunta al i-ésimo objeto más allá de `pa`.

La correspondencia entre la indexación y la aritmética de punteros es muy estrecha. Por definición, el valor de una variable o expresión de tipo array es la dirección del elemento cero de la matriz.
Por lo tanto, después de la asignación

```
pa = &a[0];
```

`pa` y `a` tienen valores idénticos. Dado que el nombre de una matriz es un sinónimo de la ubicación del elemento inicial, la asignación `pa=&a[0]` también se puede escribir como

```
pa = a;
```

Bastante más sorprendente, a primera vista, es el hecho de que una referencia a `a[i]` también puede escribirse como
`*(a+i)`. Al evaluar `a[i]`, C lo convierte en `*(a+i)` inmediatamente; las dos formas son equivalentes. Aplicando el operador `&` a ambas partes de esta equivalencia, se deduce que `&a[i]` y `a+i` también son idénticos: `a+i` es la dirección del i-ésimo elemento más allá `de a`. Como la otra cara de esta moneda, si `pa` es un puntero, las expresiones podrían usarlo con un subíndice; `pa[i]` es idéntico a
`*(pa+i)`. En resumen, una expresión de matriz e índice es equivalente a una escrita como puntero y desplazamiento.

Hay una diferencia entre un nombre de matriz y un puntero que debe tenerse en cuenta. Un puntero es una variable, por lo que `pa=a` y `pa++` son legales. Pero un nombre de matriz no es una variable; Construcciones como `A=PA` y `A++` son ilegales.

Cuando se pasa un nombre de matriz a una función, lo que se pasa es la ubicación del elemento inicial. Dentro de la función llamada, este argumento es una variable local, por lo que un parámetro de nombre de matriz es un puntero, es decir, una variable que contiene una dirección. Podemos usar este hecho para escribir otra versión de `strlen`, que calcula la longitud de una cadena.

```
/* strlen: longitud devuelta de la cadena
s */ int strlen(char *s)
{
    int n;

    para (n = 0; *s != '\0', s++)
        n++;
    retorno n;
}
```

Dado que `s` es un puntero, incrementarlo es perfectamente legal; `S++` no tiene ningún efecto en la cadena de caracteres de la función que llamó a `strlen`, sino que simplemente incrementa `la copia privada de strlen` del puntero. Eso significa que llamadas como

```
strlen("hola, mundo");/* constante de cadena */
strlen(matriz);                 /* matriz de
caracteres[100]; */
strlen(ptr);              /* char *ptr; */
```

todo el trabajo.

Como parámetros formales en una definición de función,

```
char s[];
```

y

```
char *s;
```

son equivalentes; Preferimos este último porque dice más explícitamente que la variable es un puntero. Cuando se pasa un nombre de matriz a una función, la función puede hacerlo a su

conveniencia

cree que se le ha entregado una matriz o un puntero, y manipularlo en consecuencia. Incluso puede usar ambas notaciones si le parece apropiado y claro.

Es posible pasar parte de una matriz a una función, pasando un puntero al principio de la submatriz. Por ejemplo, si `a` es una matriz,

```
f(&a[2])
```

y

```
f(a+2)
```

Ambos pasan a la función `f` la dirección del subarray que comienza en `a[2]`. Dentro de `f`, la declaración del parámetro puede leer:

```
f(int arr[]) { ... }
```

o

```
f(int *arr) { ... }
```

Por lo tanto, en lo `que respecta a f`, el hecho de que el parámetro se refiera a parte de una matriz más grande no tiene ninguna consecuencia.

Si uno está seguro de que los elementos existen, también es posible indexar hacia atrás en una matriz; `p[-1]`, `p[-2]`, etc., son sintácticamente legales y se refieren a los elementos que preceden inmediatamente a `p[0]`. Por supuesto, es ilegal hacer referencia a objetos que no están dentro de los límites de la matriz.

5.4 Aritmética de direcciones

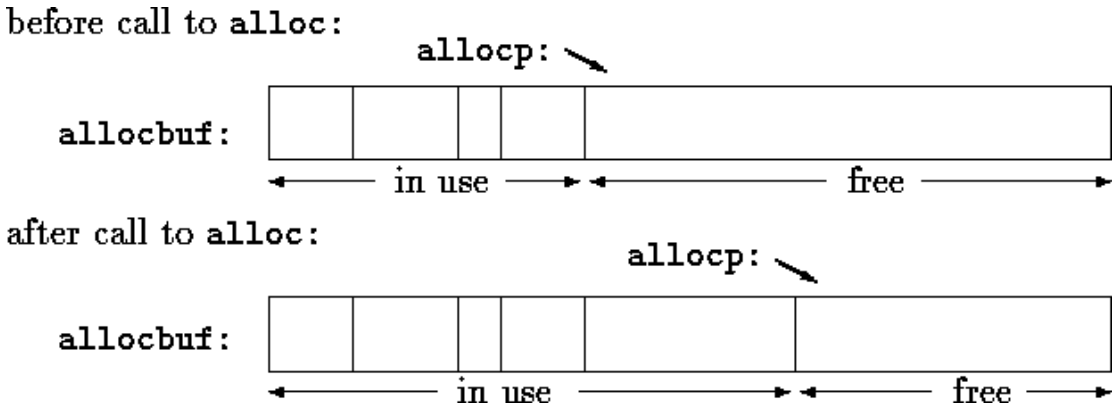
Si `p` es un puntero a algún elemento de una matriz, entonces `p++` incrementa `p` para que apunte al siguiente elemento, y `p+=i` lo incrementa para señalar `i` elementos más allá de donde lo hace actualmente. Estas y otras construcciones similares son las formas simples de la aritmética de punteros o direcciones.

C es consistente y regular en su enfoque de la aritmética de direcciones; su integración de punteros, matrices y aritmética de direcciones es una de las fortalezas del lenguaje. Vamos a ilustrarlo escribiendo un asignador de almacenamiento rudimentario. Hay dos rutinas. El primero, `alloc(n)`, devuelve un puntero a `n` posiciones de caracteres consecutivos, que puede ser utilizado por el llamador de `alloc` para almacenar caracteres. El segundo, `afree(p)`, libera el almacenamiento así adquirido para que pueda ser reutilizado más tarde. Las rutinas son "rudimentarias" porque las llamadas a `afree` deben hacerse en el orden opuesto a las llamadas hechas en `alloc`. Es decir, el almacenamiento administrado por `alloc` y `afree` es una pila, o el último en entrar, el primero en salir. La biblioteca estándar proporciona funciones análogas llamadas `malloc` y `free` que no tienen tales restricciones; en la Sección 8.7 mostraremos cómo se pueden implementar.

La implementación más sencilla es hacer `que alloc` entregue partes de una matriz de caracteres grande que llamaremos `allocbuf`. Esta matriz es private to `alloc` y `afree`. Dado que se trata de punteros, no de índices de matriz, ninguna otra rutina necesita conocer el nombre de la matriz, que puede ser declarada `estática` en el fichero fuente que contiene `alloc` y `afree`, y por lo tanto ser invisible fuera de ella. En implementaciones prácticas, es posible que la matriz ni siquiera tenga un nombre; En su lugar, podría obtenerse llamando a `malloc` o solicitando al sistema operativo un puntero a algún bloque de almacenamiento sin nombre.

La otra información necesaria es la cantidad de `alocbuf` que se ha utilizado. Usamos un puntero, llamado `allocp`, que apunta al siguiente elemento libre. Cuando `se le pide` a alloc `n` caracteres,

Comprueba si queda suficiente espacio en `Allocbuf`. Si es así, `alloc` devuelve el valor actual de `allocp` (es decir, el comienzo del bloque libre) y, a continuación, lo incrementa en `n` para que apunte a la siguiente área libre. Si no hay espacio, `alloc` devuelve cero. `afree(p)` simplemente establece `allocp` en `p` si `p` está dentro de `allocbuf`.




```
#define ALLOCSIZE 10000 /* tamaño del espacio disponible */

static char allocbuf[ALLOCSIZE]; /* almacenamiento para
alloc */ static char *allocp = allocbuf; /* siguiente
posición libre */

char *alloc(int n)    /* devuelve el puntero a n caracteres */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* se ajusta */
        allocp += n;
        devolver allocp - n; /* Viejo P */
    } else/* no hay suficiente
        espacio */ return 0;
}

void afree(char *p) /* almacenamiento libre señalado por p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

En general, un puntero se puede inicializar como cualquier otra variable, aunque normalmente los únicos valores significativos son cero o una expresión que implique la dirección de datos previamente definidos del tipo apropiado. La declaración

```
carácter estático *allocp = allocbuf;
```

Define `allocp` para que sea un puntero de carácter y lo inicializa para que apunte al principio de `allocbuf`, que es la siguiente posición libre cuando se inicia el programa. Esto también podría haber sido escrito

```
static char *allocp = &allocbuf[0];
```

ya que el nombre de la matriz *es* la dirección del elemento cero.

La prueba

```
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* se ajusta */
```

Comprueba si hay suficiente espacio para satisfacer una solicitud de `n` caracteres. Si lo hay, el nuevo valor de `allocp` sería como máximo uno más allá del final de `allocbuf`. Si la solicitud puede ser satisfecha, `alloc` devuelve un puntero al principio de un bloque de caracteres (observe la declaración de la función en sí). Si no es así, `alloc` debe devolver alguna señal de que no queda espacio. C garantiza que cero nunca es una dirección válida para los datos, por lo que se puede usar un valor de retorno de cero para señalar un evento anormal, en este caso sin espacio.

Los punteros y los enteros no son intercambiables. El cero es la única excepción: la constante cero puede asignarse a un puntero, y un puntero puede compararse con la constante cero. La constante simbólica `NULL` se usa a menudo en lugar de cero, como una mnemotecnia para indicar más claramente que se trata de un valor especial para un puntero. `NULL` se define en `<stdio.h>`. A `partir de ahora`, usaremos NULL.

Pruebas como

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* se ajusta */
```

y

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

Mostrar varias facetas importantes de la aritmética de punteros. En primer lugar, los punteros pueden compararse en determinadas circunstancias. Si `p` y `q` apuntan a miembros de la misma matriz, entonces relaciones como `==`, `!=`, `<`, `>=`, etc., funcionan correctamente. Por ejemplo

```
p < q
```

es verdadero si `p` apunta a un elemento anterior de la matriz que `q` . Cualquier puntero se puede comparar significativamente para la igualdad o la desigualdad con cero. Sin embargo, el comportamiento no está definido para la aritmética o las comparaciones con punteros que no apuntan a miembros de la misma matriz. (Hay una excepción: la dirección del primer elemento más allá del final de una matriz se puede usar en la aritmética de punteros).

En segundo lugar, ya hemos observado que se pueden sumar o restar un puntero y un número entero.
La construcción

```
p + n
```

significa la dirección del `objeto n-ésimo más allá del que p` apunta actualmente. Esto es cierto independientemente del tipo de objeto al que apunte p; `n` se escala de acuerdo con el tamaño de los objetos a los que apunta p, que se determina mediante la declaración de `p`. Si un `int` es de cuatro bytes, por ejemplo, el `int` se escalará en cuatro.

La resta de puntero también es válida: si `p` y `q` apuntan a elementos de la misma matriz, y `p<q`, entonces `q-p+1` es el número de elementos de `p` a `q` inclusive. Este hecho se puede utilizar para escribir otra versión de `strlen`:

```
/* strlen: longitud devuelta de la cadena
s */ int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    retorno p - s;
}
```

En su declaración, `p` se inicializa en `s`, es decir, para señalar el primer carácter de la cadena. En el bucle `while`, cada carácter a su vez se examina hasta que se ve el `'\0'` al final. Dado que `p` apunta a caracteres, `p++` avanza `p` al siguiente carácter cada vez, y `p-s` da el número de caracteres avanzados, es decir, la longitud de la cadena. (El número de caracteres de la cadena podría ser demasiado grande para almacenarlo en un archivo `int`. El encabezado `<stddef.h>` define un tipo `ptrdiff_t` que es lo suficientemente grande como para contener la diferencia con signo de dos valores de puntero. Sin embargo, si fuéramos cautelosos, usaríamos `size_t` para el valor devuelto de `strlen`, para que coincida con la versión estándar de la biblioteca. `size_t` es el tipo entero sin signo devuelto por el operador `sizeof`.

La aritmética de punteros es consistente: si hubiéramos estado tratando con `flotantes, que ocupan más espacio` que `los chars, y si p` fuera un puntero a `float`, `p++` avanzaría al siguiente `float`. Por lo tanto, podríamos escribir otra versión de `alloc` que mantenga `floats en lugar de chars, simplemente cambiando char` por `float` a través de `alloc` y `afree`. Todas las manipulaciones del puntero tienen en cuenta automáticamente el tamaño de los objetos a los que se apunta.

Las operaciones de puntero válidas son la asignación de punteros del mismo tipo, la adición o sustracción de un puntero y un número entero, la resta o comparación de dos punteros con miembros de la misma matriz y la asignación o comparación con cero. Todas las demás operaciones aritméticas de punteros son ilegales. No es legal agregar dos punteros, ni multiplicarlos, dividirlos, desplazarlos ni enmascararlos, ni agregarles `float` o `double`, ni incluso, a excepción de `void *`, asignar un puntero de un tipo a un puntero de otro tipo sin una conversión.

5.5 Punteros de caracteres y funciones

Una *constante de cadena*, escrita como

```
"Soy una cuerda"
```

es una matriz de caracteres. En la representación interna, la matriz termina con el carácter nulo `'\0'` para que los programas puedan encontrar el final. Por lo tanto, la longitud almacenada es uno más que el número de caracteres entre las comillas dobles.

Quizás la ocurrencia más común de constantes de cadena es como argumentos para funciones, como en

```
printf("hola, mundo\n");
```

Cuando una cadena de caracteres como esta aparece en un programa, el acceso a ella se realiza a través de un puntero de caracteres; `printf` recibe un puntero al principio de la matriz de caracteres. Es decir, un puntero tiene acceso a una constante de cadena a su primer elemento.

Las constantes de cadena no tienen por qué ser argumentos de función. Si `pmessage` se declara como

```
char *pmensaje;
```

A continuación, la declaración

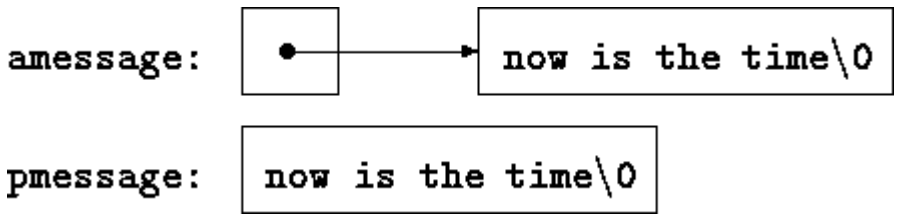
```
pmessage = "ahora es el momento";
```

Asigna a `pmessage` un puntero a la matriz de caracteres. No se trata de una copia de cadena, solo se trata de punteros. C no proporciona ningún operador para procesar una cadena completa de caracteres como una unidad.

Hay una diferencia importante entre estas definiciones:

```
char amessage[] = "Ahora es el momento"; /* un
array */ char *pmessage = "ahora es el momento"; /*
un puntero */
```

`amessage` es una matriz, lo suficientemente grande como para contener la secuencia de caracteres y `'\0'` que la inicializa. Los caracteres individuales dentro de la matriz se pueden cambiar, pero `amessage` siempre se referirá al mismo almacenamiento. Por otro lado, `pmessage` es un puntero, inicializado para apuntar a una constante de cadena; el puntero puede modificarse posteriormente para apuntar a otro lugar, pero el resultado es indefinido si intenta modificar el contenido de la cadena.



Ilustraremos más aspectos de los punteros y matrices estudiando versiones de dos funciones útiles adaptadas de la biblioteca estándar. La primera función es `strcpy(s,t)`, que copia la cadena `t` en la cadena `s`. Sería bueno decir `solo s = t`, pero esto copia el puntero, no los caracteres. Para copiar los caracteres, necesitamos un bucle. La versión de la matriz primero:

```
/* strcpy: copiar de la t a la s; versión del
subíndice de la matriz */ void strcpy(char *s, char
*t)
{
    Int I;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Para contrastar, aquí hay una versión de `strcpy` con punteros:

```
/* strcpy: copiar de la t a la s; Versión del
puntero */ void strcpy(char *s, char *t)
{
    Int I;

    i = 0;
    while ((*s = *t) != '\0') {
        s++;
        T++;
    }
}
```

Debido a que los argumentos se pasan por valor, `strcpy` puede usar los parámetros `s y t` de la forma que desee. Aquí son punteros convenientemente inicializados, que se marchan a lo largo de las matrices un carácter a la vez, hasta que el `'\0'` que termina `en t` se ha copiado en `s`.

En la práctica, `strcpy` no se escribiría como lo mostramos anteriormente. Los programadores de C experimentados preferirían

```
/* strcpy: copiar de la t a la s; Puntero
versión 2 */ void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
```

```
}
```

Esto mueve el incremento de `s` y `t` a la parte de prueba del bucle. El valor de `*t++` es el carácter al que `apuntaba t` antes de que `se incrementara t; el sufijo ++ no cambia t` hasta después de que se haya obtenido este carácter. De la misma manera, el carácter se almacena en la posición `s` anterior antes `de` que se incremente s. Este carácter es también el valor que se compara con `'\0'` para controlar el bucle. El efecto neto es que los caracteres se copian de `t` a `s`, hacia arriba e incluyendo la terminación `'\0'`.

Como abreviatura final, obsérvese que una comparación con `'\0'` es redundante, ya que la cuestión es simplemente si la expresión es cero. Por lo tanto, es probable que la función se escriba como

```
/* strcpy: copiar de la t a la s; Puntero
versión 3 */ void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Aunque esto pueda parecer críptico a primera vista, la conveniencia de la notación es considerable, y el modismo debe dominarse, porque lo verá con frecuencia en los programas C.

El `strcpy de` la biblioteca estándar (`<string.h>`) devuelve la cadena de destino como valor de función.

La segunda rutina que examinaremos es `strcmp(s,t),` que compara las cadenas de caracteres `s` y `t`, y devuelve negativo, cero o positivo si `s` es lexicográficamente menor, igual o mayor que `t`. El valor se obtiene restando los caracteres en la primera posición donde `s` y `t` no están de acuerdo.

```
/* strcmp: devuelve <0 si s<t, 0 si s==t, >0 si s>t */
int strcmp(char *s, char *t)
{
    Int I;

    para (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            devuelve 0;
    return s[i] - t[i];
}
```

La versión de puntero de `strcmp`:

```
/* strcmp: devuelve <0 si s<t, 0 si s==t, >0 si s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            devuelve
    0; devolver *s -
    *t;
}
```

Dado que `++` y `--` son operadores de prefijo o sufijo, otras combinaciones de `*` y `++` y `--` ocurren, aunque con menor frecuencia. Por ejemplo

```
*--p
```

disminuye `p` antes de obtener el carácter al que apunta p. De hecho, el par de expresiones

```
*p++ = val; /* empujar val a la pila */
val = *--p; /* Coloque la parte superior de la pila en Val */
```

son el modismo estándar para empujar y hacer saltar una pila; consulte la Sección 4.3.

El encabezado `<string.h>` contiene declaraciones para las funciones mencionadas en esta sección, además de una variedad de otras funciones de control de cadenas de la biblioteca estándar.

Ejercicio 5-3. Escriba una versión de puntero de la función `strcat` que mostramos en el Capítulo 2:
`strcat(s,t)` copia la cadena `t` al final de `s`.

Ejercicio 5-4. Escriba la función `strend(s,t)`, que devuelve 1 si la cadena `t` aparece al final de la cadena `s`, y cero en caso contrario.

Ejercicio 5-5. Escriba versiones de las funciones de biblioteca `strncpy`, `strncat` y `strncmp`, que operan como máximo con los primeros `n` caracteres de sus cadenas de argumentos. Por ejemplo, `strncpy(s,t,n)` copia como máximo `n` caracteres de `t` a `s`. Las descripciones completas se encuentran en el Apéndice B.

Ejercicio 5-6. Reescriba los programas apropiados de capítulos y ejercicios anteriores con punteros en lugar de indexación de matrices. Algunas buenas posibilidades son `getline` (Capítulos 1 y 4), `atoi`, `itoa` y sus variantes (Capítulos 2, 3 y 4), `reverse` (Capítulo 3) y `strindex` y `getop` (Capítulo 4).

5.6 Matrices de punteros; Punteros a punteros

Dado que los punteros son variables en sí mismas, se pueden almacenar en matrices al igual que otras variables. Ilustrémoslo escribiendo un programa que ordenará un conjunto de líneas de texto en orden alfabético, una versión simplificada de la ordenación del programa UNIX.

En el Capítulo 3, presentamos una función de ordenación de Shell que ordenaría una matriz de números enteros, y en el Capítulo 4 la mejoramos con una ordenación rápida. Los mismos algoritmos funcionarán, excepto que ahora tenemos que lidiar con líneas de texto, que son de diferentes longitudes y que, a diferencia de los números enteros, no se pueden comparar ni mover en una sola operación. Necesitamos una representación de datos que se adapte de manera eficiente y conveniente a las líneas de texto de longitud variable.

Aquí es donde entra la matriz de punteros. Si las líneas que se van a ordenar se almacenan de extremo a extremo en una matriz de caracteres largos, se puede acceder a cada línea mediante un puntero a su primer carácter. Los propios punteros se pueden almacenar en una matriz. Dos líneas se pueden comparar pasando sus punteros a `strcmp`. Cuando se tienen que intercambiar dos líneas desordenadas, se intercambian los punteros de la matriz de punteros, no las propias líneas de texto.

Esto elimina los problemas gemelos de una gestión de almacenamiento complicada y una alta sobrecarga que conllevaría el traslado de las propias líneas.

El proceso de clasificación consta de tres pasos:

leer todas las líneas de
entrada ordenarlas
imprimirlos en orden

Como de costumbre, es mejor dividir el programa en funciones que coincidan con esta división natural, con la rutina principal controlando las otras funciones. Aplacemos el paso de clasificación por un momento y concentrémonos en la estructura de datos y la entrada y salida.

La rutina de entrada tiene que recopilar y guardar los caracteres de cada línea, y construir una matriz de punteros a las líneas. También tendrá que contar el número de líneas de entrada, ya que esa información es necesaria para clasificar e imprimir. Dado que la función de entrada solo puede hacer frente a un número finito de líneas de entrada, puede devolver un recuento ilegal como `-1` si se presenta demasiada entrada.

La rutina de salida solo tiene que imprimir las líneas en el orden en que aparecen en la matriz de punteros.

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

#define MAXLINES 5000/* #lines máx. a clasificar */

char *lineptr[MAXLINES]; /* punteros a líneas de texto

*/ int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* ordenar líneas de
entrada */ main()
{
    int nlines;     /* número de líneas de entrada leídas */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        devuelve 0;
    } else {
        printf("error: entrada demasiado grande
        para ordenar\n"); retorno 1;
    }
}

#define MAXLEN 1000 /* longitud máxima de cualquier
línea de entrada */ int getline(char *, int);
char *alloc(int);

/* readlines: lee las líneas de entrada */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, línea[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || p = alloc(len) == NULL)
            return -1;
        else {
```

```
            línea[len-1] = '\0'; /* borrar nueva línea */
            strcpy(p, línea);
            lineptr[nlines++] = p;
        }
    nlines de retorno;
}

/* writelines: escribir líneas de salida */
void writelines(char *lineptr[], int nlines)
{
    Int I;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

La función `getline` es de la Sección 1.9.

La principal novedad es la declaración de `lineptr`:

```
char *lineptr[MAXLINES]
```

dice que `lineptr` es una matriz de `elementos MAXLINES`, cada uno de cuyos elementos es un puntero a un `char`. Es decir, `lineptr[i]` es un puntero de carácter, y `*lineptr[i]` es el carácter al que apunta, el primer carácter de la i-ésima línea de texto guardada.

Dado que `lineptr` es en sí mismo el nombre de una matriz, se puede tratar como un puntero de la misma manera que en nuestros ejemplos anteriores, y `writelines` se puede escribir en su lugar como

```
/* writelines: escribir líneas de salida */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Inicialmente, `*lineptr` apunta a la primera línea; cada elemento la avanza al siguiente puntero de línea mientras `nlines` se cuenta hacia atrás.

Con la entrada y la salida bajo control, podemos proceder a la clasificación. El ordenamiento rápido del Capítulo 4 necesita cambios menores: las declaraciones deben modificarse y la operación de comparación debe realizarse llamando a `strcmp`. El algoritmo sigue siendo el mismo, lo que nos da cierta confianza en que seguirá funcionando.

```
/* qsort: ordenar v[izquierda]... v[derecha] en orden
creciente */ void qsort(char *v[], int izquierda, int
derecha)
{
    int i, último;
    intercambio de vacío(char *v[], int i, int j);

    if (izquierda >= derecha) /* no hacer nada si la
        matriz contiene */ retorno;   /* menos de dos
        elementos */
    intercambio(v, izquierda,
    (izquierda + derecha)/2); último
    = izquierda;
    for (i = izquierda+1; i <=
        derecha; i++) si (strcmp(v[i],
        v[left]) < 0)
            permuta(v,
    ++último, i); swap(v,
```

```
izquierda, última);
qsort(v, izquierda, último-
1);
```

```
    qsort(v, último+1, derecha);
}
```

Del mismo modo, la rutina de intercambio solo necesita cambios triviales:

```
/* swap: intercambia v[i] y v[j] */ void
swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j]       =
    temperatura;
}
```

Dado que cualquier elemento individual de `v` (alias `lineptr`) es un puntero de carácter, `temp` debe ser también, por lo que uno se puede copiar al otro.

Ejercicio 5-7. Vuelva a escribir `las líneas de lectura` para almacenar líneas en una matriz suministrada por `main`, en lugar de llamar a `alloc` para mantener el almacenamiento. ¿Cuánto más rápido es el programa?

5.7 Matrices multidimensionales

C proporciona matrices multidimensionales rectangulares, aunque en la práctica se utilizan mucho menos que las matrices de punteros. En esta sección, mostraremos algunas de sus propiedades.

Considere el problema de la conversión de fechas, de día del mes a día del año y viceversa. Por ejemplo, el 1 de marzo es el día 60 de un año no bisiesto y el día 61 de un año bisiesto. Definamos dos funciones para hacer las conversiones: `day_of_year` convierte el mes y el día en el día del año, y `month_day` convierte el día del año en el mes y el día. Dado que esta última función calcula dos valores, los argumentos month y day serán punteros:

```
month_day(1988, 60, &m, &d)
```

establece `m` en 2 y `d` en 29 (29 de febrero).

Estas funciones necesitan la misma información, una tabla del número de días de cada mes ("treinta días tiene septiembre..."). Dado que el número de días por mes difiere para los años bisiestos y los años no bisiestos, es más fácil separarlos en dos filas de una matriz bidimensional que realizar un seguimiento de lo que sucede en febrero durante el cálculo. La matriz y las funciones para realizar las transformaciones son las siguientes:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: establecer el día del año desde el
mes y el día */ int day_of_year(int año, int mes, int
día)
{
    int i, salto;
    bisio = año%4 == 0 && año%100 != 0 || año%400 == 0;
    para (i = 1; i < mes; i++)
        día += daytab[bisiesto][i];
    día de regreso;
}
```

```
/* month_day: establecer mes, día a partir del día del año */
void month_day(int año, int año, int *pmes, int *pday)
{
    int i, salto;

    bisio = año%4 == 0 && año%100 != 0 || año%400 == 0;
    para (i = 1; día del año > díatab[bisiesto][i]; i++)
        día del año -= díatab[bisiesto][i];
    *pmes = yo;
    *pday = día del año;
}
```

Recuerde que el valor aritmético de una expresión lógica, como el de `leap`, es cero (falso) o uno (verdadero), por lo que se puede usar como un subíndice de la matriz `daytab`.

La matriz `daytab` tiene que ser externa tanto a `day_of_year` como `a month_day`, para que ambos puedan utilizarla. Lo hicimos `char` para ilustrar un uso legítimo de `char` para almacenar pequeños enteros que no son caracteres.

`daytab` es la primera matriz bidimensional con la que nos hemos ocupado. En C, una matriz bidimensional es en realidad una matriz unidimensional, cada uno de cuyos elementos es una matriz. Por lo tanto, los subíndices se escriben como

```
daytab[i][j]     /* [fila][col] */
```

en lugar de

```
daytab[i,j]     /*INCORRECTO*/
```

Aparte de esta distinción notacional, una matriz bidimensional se puede tratar de la misma manera que en otros idiomas. Los elementos se almacenan por filas, por lo que el subíndice o columna situado más a la derecha varía más rápido a medida que se accede a los elementos en orden de almacenamiento.

Una matriz se inicializa mediante una lista de inicializadores entre llaves; cada fila de una matriz bidimensional se inicializa mediante una sublista correspondiente. Comenzamos la matriz `daytab` con una columna de cero para que los números de los meses puedan ir del 1 al 12 natural en lugar del 0 al 11. Dado que el espacio no es escaso aquí, esto es más claro que ajustar los índices.

Si se va a pasar una matriz bidimensional a una función, la declaración del parámetro en la función debe incluir el número de columnas; el número de filas es irrelevante, ya que lo que se pasa es, como antes, un puntero a una matriz de filas, donde cada fila es una matriz de 13 `enteros. En este caso particular, es un puntero a objetos que son matrices de 13 enteros. Por lo tanto, si la matriz daytab` se va a pasar a una función `f`, la declaración de `f` sería:

```
f(int daytab[2][13]) { ... }
```

También podría ser

```
f(int daytab[][13]) { ... }
```

Dado que el número de filas es irrelevante, o podría ser

```
f(int (*daytab)[13]) { ... }
```

que dice que el parámetro es un puntero a una matriz de 13 enteros. Los paréntesis son necesarios ya que los corchetes `[]` tienen mayor prioridad que `*`. Sin paréntesis, la declaración

```
int *daytab[13]
```

es una matriz de 13 punteros a números enteros. De manera más general, solo la primera dimensión (subíndice) de una matriz es libre; todos los demás tienen que ser especificados.

En la sección 5.12 se analizan más a fondo las declaraciones complicadas.

Ejercicio 5-8. No hay comprobación de errores en `day_of_year` ni en `month_day`. Remedie este defecto.

5.8 Inicialización de matrices de punteros

Considere el problema de escribir una función `month_name(n)`, que devuelve un puntero a una cadena de caracteres que contiene el nombre del n-ésimo mes. Esta es una aplicación ideal para una matriz `estática` interna. `month_name` contiene una matriz privada de cadenas de caracteres y devuelve un puntero al correcto cuando se llama. En esta sección se muestra cómo se inicializa esa matriz de nombres.

La sintaxis es similar a la de las inicializaciones anteriores:

```
/* month_name: nombre devuelto del n-ésimo mes
*/ char *month_name(int n)
{
    static char *nombre[] = {
        "Mes ilegal",
        "Enero", "Febrero", "Marzo",
        "Abril", "Mayo", "Junio",
        "Julio", "Agosto", "Septiembre",
        "Octubre", "Noviembre",
        "Diciembre"
    };

    return (n < 1 || n > 12) ? nombre[0] : nombre[n];
}
```

La declaración de `name`, que es una matriz de punteros de caracteres, es la misma que `lineptr` en el ejemplo de ordenación. El inicializador es una lista de cadenas de caracteres; Cada uno se asigna a la posición correspondiente en la matriz. Los caracteres de la i-ésima cadena se colocan en algún lugar, y un puntero a ellos se almacena en `name[i]`. Dado que no se especifica el tamaño del nombre de la matriz , el compilador cuenta los inicializadores y rellena el número correcto.

5.9 Punteros frente a matrices multidimensionales

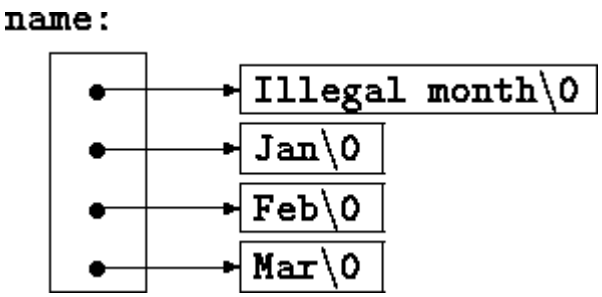
Los recién llegados a C a veces se confunden acerca de la diferencia entre una matriz bidimensional y una matriz de punteros, como name en el ejemplo anterior. Dadas las definiciones

```
int a[10][20];
int *b[10];
```

Entonces, `A[3][4]` y `B[3][4]` son referencias sintácticamente legales a un solo `int`. Pero `a` es una verdadera matriz bidimensional: se han reservado 200 ubicaciones de tamaño entero, y se utiliza el cálculo de subíndice rectangular convencional 20 * *fila* +col para encontrar el elemento `a[fila,col]`. Para `b`, sin embargo, la definición solo asigna 10 punteros y no los inicializa; la inicialización debe hacerse explícitamente, ya sea estáticamente o con código. Suponiendo que cada elemento de `b` apunta a una matriz de veinte elementos, entonces habrá 200 `enteros reservados, más diez celdas para los punteros. La ventaja importante de la matriz de punteros es que las filas de la matriz pueden tener diferentes` longitudes. Es decir, cada elemento de `b` no tiene por qué apuntar a un vector de veinte elementos; algunos pueden apuntar a dos elementos, otros a cincuenta y otros a ninguno.

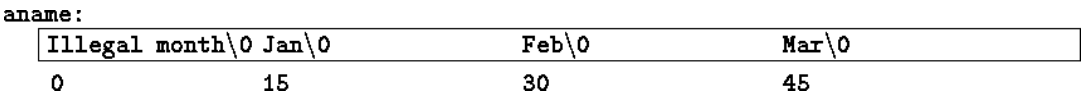
Aunque hemos expresado esta discusión en términos de números enteros, el uso más frecuente de matrices de punteros es almacenar cadenas de caracteres de diversas longitudes, como en la función `month_name`. Compare la declaración y la imagen para una matriz de punteros:

```
char *nombre[] = { "Mes ilegal", "Enero", "Febrero", "Marzo" };
```



con los de una matriz bidimensional:

```
char aname[][15] = { "Mes ilegal", "Enero", "Febrero", "Marzo" };
```



Ejercicio 5-9. Reescriba las rutinas `day_of_year` y `month_day` con punteros en lugar de indexar.

5.10 Argumentos de la línea de comandos

En entornos compatibles con C, hay una manera de pasar argumentos o parámetros de la línea de comandos a un programa cuando comienza a ejecutarse. Cuando `se llama a main`, se llama con dos argumentos. El primero (convencionalmente llamado `argc`, por conteo de argumentos) es el número de argumentos de la línea de comandos con los que se invocó el programa; El segundo (`argv`, para vector de argumentos) es un puntero a una matriz de cadenas de caracteres que contienen los argumentos, uno por cadena. Habitualmente usamos varios niveles de punteros para manipular estas cadenas de caracteres.

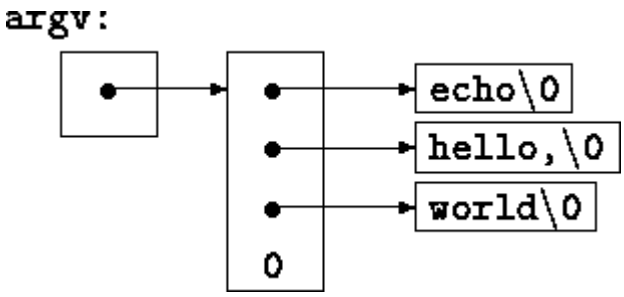
La ilustración más simple es el programa `echo`, que repite sus argumentos de línea de comandos en una sola línea, separada por espacios en blanco. Es decir, el comando

```
eco hola, mundo
```

Imprime la salida

```
Hola mundo
```

Por convención, `argv[0]` es el nombre por el que se invocó el programa, por lo que `argc` es al menos 1. Si `argc` es 1, no hay argumentos de línea de comandos después del nombre del programa. En el ejemplo anterior, `argc` es 3 y `argv[0]`, `argv[1]` y `argv[2]` son `"echo"`, `"hello"` y `"world"` respectivamente. El primer argumento opcional es `argv[1]` y el último es `argv[argc-1]`; además, el estándar requiere que `argv[argc]` sea un puntero nulo.



La primera versión de `echo` trata argv como una matriz de punteros de caracteres:

```
#include <stdio.h>

/* echo argumentos de la línea de comandos; 1ª
versión */ main(int argc, char *argv[])
{
    Int I;

    para (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    devuelve 0;
}
```

Dado que `argv` es un puntero a una matriz de punteros, podemos manipular el puntero en lugar de indexar la matriz. La siguiente variante se basa en el incremento de `argv`, que es un puntero a otro para `char`, mientras `que argc` se cuenta hacia atrás:

```
#include <stdio.h>

/* echo argumentos de la línea de comandos; 2ª
versión */ main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    devuelve 0;
}
```

Dado que `argv` es un puntero al principio de la matriz de cadenas de argumentos, incrementarlo en 1 (`++argv`) hace que apunte al `argv[1] original` en lugar de `argv[0]`. Cada incremento sucesivo lo mueve al siguiente argumento; `*argv` es entonces el puntero a ese argumento. Al mismo tiempo, `argc` se decreta; cuando se convierte en cero, no quedan argumentos para imprimir.

Alternativamente, podríamos escribir la `declaración printf` como

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Esto muestra que el argumento format de `printf` también puede ser una expresión.

Como segundo ejemplo, vamos a hacer algunas mejoras en el programa de búsqueda de patrones de la Sección 4.1. Si recuerdan, conectamos el patrón de búsqueda profundamente en el programa, un arreglo obviamente insatisfactorio. Siguiendo el ejemplo del programa `grep de UNIX`, mejoremos el programa para que el patrón que debe coincidir se especifique mediante el primer argumento en la línea de comandos.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

obtienes línea(char*line, eres máximo);

/* find: imprime líneas que coinciden con el patrón de 1st arg
*/ main(int argc, char *argv[])
{
    línea de
    caracteres[MAXLINE];
    int encontrado = 0;

    si (argc != 2)
        printf("Uso: encontrar
    patrón\n"); más
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                encontrado++;
            }
    devolución encontrada;
}
```

La función de biblioteca estándar `strstr(s,t)` devuelve un puntero a la primera aparición de la cadena `t` en la cadena `s`, o `NULL` si no hay ninguno. Se declara en `<string.h>`.

El modelo ahora se puede elaborar para ilustrar más construcciones de punteros. Supongamos que queremos permitir dos argumentos opcionales. Uno dice "imprime todas las líneas *excepto* las que coinciden con el patrón"; el segundo dice "precede a cada línea impresa por su número de línea".

Una convención común para los programas C en sistemas UNIX es que un argumento que comienza con un signo menos introduce un indicador o parámetro opcional. Si elegimos `-x` (para "excepto") para señalar la inversión, y `-n` ("número") para solicitar la numeración de líneas, entonces el comando

```
find -x -npattern
```

imprimirá cada línea que no coincida con el patrón, precedida por su número de línea.

Los argumentos opcionales deben permitirse en cualquier orden, y el resto del programa debe ser independiente del número de argumentos que presentemos. Además, es conveniente para los usuarios si los argumentos de opción se pueden combinar, como en

```
Buscar patrón -nx
```

Este es el programa:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

obtienes línea(char*line, eres máximo);

/* find: imprime líneas que coinciden con el patrón de
1st arg */ main(int argc, char *argv[])
{
    línea de
    caracteres[MAXLINE];
    línea larga = 0;
    int c, except = 0, number = 0, found = 0;
```

```
while (--argc > 0 && (*++argv)[0] == '-')
```

```
        while (c = *++argv[0])
            switch (c) {
            Caso 'X':
                excepto =
                1; quebrar;
            caso 'n':
                número = 1;
                quebrar;
            predeterminado:
                printf("Buscar: opción ilegal %c\n", c);
                argc = 0;
                encontrado
                = -1;
                quebrar;
            }
    if (argc != 1)
        printf("Uso: find -x -n patrón\n"); más
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(línea, *argv) != NULL) != excepto)
                { if (número)
                    printf("%ld:", lineno);
                printf("%s", línea);
                encontrado++;
            }
        }
    devolución encontrada;
}
```

`argc` se decremente y `argv` se incrementa antes de cada argumento opcional. Al final del bucle, si no hay errores, `argc` indica cuántos argumentos quedan sin procesar y `argv` señala el primero de ellos. Por lo tanto, `argc` debe ser 1 y `*argv` debe apuntar al patrón. Observe que `*++argv` es un puntero a una cadena de argumentos, por lo que `(*++argv)[0]` es su primer carácter. (Una forma válida alternativa sería `**++argv`.) Debido a que `[]` enlaza más estrechamente que `*` y `++`, los paréntesis son necesarios; sin ellos, la expresión se tomaría como `*++(argv[0])`. De hecho, eso es lo que hemos usado en el bucle interno, donde la tarea es caminar a lo largo de una cadena de argumentos específica. En el bucle interno, la expresión `*++argv[0]` incrementa el puntero `argv[0]`!

Es raro que se utilicen expresiones de puntero más complicadas que estas; En estos casos, dividirlos en dos o tres pasos será más intuitivo.

Ejercicio 5-10. Escriba el programa `expr`, que evalúa una expresión polaca inversa desde la línea de comandos, donde cada operador u operando es un argumento independiente. Por ejemplo

```
expr 2 3 4 + *
```

Evalúa 2 * (3 + 4).

Ejercicio 5-11. Modifique el programa `entab` y `detab` (escrito como ejercicios en el Capítulo 1) para aceptar una lista de tabulaciones como argumentos. Utilice la configuración de pestaña predeterminada si no hay argumentos.

Ejercicio 5-12. Extenda `entab` y `detab` para aceptar la abreviatura

```
entab -m +n
```

para significar las paradas de tabulación cada *n* columnas, comenzando en la columna *M*. Elija el comportamiento predeterminado conveniente (para el usuario).

Ejercicio 5-13. Escriba la cola del programa, que imprime las últimas *n* líneas de su entrada. De forma predeterminada, *n*
se establece en 10, digamos, pero se puede cambiar mediante un argumento opcional para que

```
cola -n
```

Imprime las últimas *n* líneas. El programa debe comportarse racionalmente sin importar cuán irrazonable sea la entrada o el valor de *n*. Escriba el programa de manera que haga el mejor uso del almacenamiento disponible; las líneas deben almacenarse como en el programa de clasificación de la Sección 5.6, no en una matriz bidimensional de tamaño fijo.

5.11 Punteros a funciones

En C, una función en sí misma no es una variable, pero es posible definir punteros a funciones, que pueden asignarse, colocarse en matrices, pasarse a funciones, devolver funciones, etc. Ilustraremos esto modificando el procedimiento de clasificación escrito anteriormente en este capítulo para que si se da el argumento opcional `-n` , ordene las líneas de entrada numéricamente en lugar de lexicográficamente.

Una clasificación a menudo consta de tres partes: una comparación que determina el orden de cualquier par de objetos, un intercambio que invierte su orden y un algoritmo de clasificación que realiza comparaciones e intercambios hasta que los objetos están en orden. El algoritmo de clasificación es independiente de las operaciones de comparación e intercambio, por lo que al pasarle diferentes funciones de comparación e intercambio, podemos organizar la clasificación por diferentes criterios. Este es el enfoque adoptado en nuestro nuevo tipo.

La comparación lexicográfica de dos líneas se realiza mediante `strcmp`, como antes; también necesitaremos una rutina `numcmp` que compare dos líneas sobre la base de un valor numérico y devuelva el mismo tipo de indicación de condición que `lo hace strcmp` . Estas funciones se declaran antes de `main` y se pasa un puntero a la apropiada a `qsort`. Hemos escatimado en el procesamiento de errores para los argumentos, a fin de concentrarnos en los problemas principales.

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

#define MAXLINES 5000/* #lines máx. a clasificar */
char *lineptr[MAXLINES]; /* punteros a líneas de texto
*/

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
Int Namkamp (Señor*, Señor*);

/* ordenar líneas de entrada
*/ main(int argc, char
*argv[])
{
    int nlines;/* número de líneas de entrada leídas
                      */ int numérico = 0;/* 1 si
    ordena numérico */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numérico = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void**) lineptr, 0, nlines-1,
          (int (*)(void*,void*))(numérico ? numcmp : strcmp));
```

```
        writelines(lineptr, nlines);
```

```
        devuelve 0;
    } else {
        printf("entrada demasiado grande para
        ordenar\n"); retorno 1;
    }
}
```

En la llamada a `qsort`, `strcmp` y `numcmp` son direcciones de funciones. Dado que se sabe que son funciones, el `&` no es necesario, de la misma manera que no es necesario antes de un nombre de matriz.

Hemos escrito `qsort` para que pueda procesar cualquier tipo de datos, no solo cadenas de caracteres. Como se indica en el prototipo de función, `qsort` espera una matriz de punteros, dos enteros y una función con dos argumentos de puntero. El tipo de puntero genérico `void *` se usa para los argumentos de puntero. Cualquier puntero se puede convertir en `void *` y viceversa sin pérdida de información, por lo que podemos llamar `a qsort` lanzando argumentos a `void *`. El elenco elaborado del argumento de función arroja los argumentos de la función de comparación. Por lo general, no tendrán ningún efecto sobre la representación real, pero aseguran al compilador que todo está bien.

```
/* qsort: ordenar v[izquierda]... v[derecha] en orden
creciente */ void qsort(void *v[], int izquierda, int
derecha,
           int (*comp)(void *, void *))
{
    int i, último;

    void swap(void *v[], int, int);

    if (left >= right)/* no hacer nada si la matriz contiene
        */ return;              /* menos de dos elementos */
    intercambio(v, izquierda,
    (izquierda + derecha)/2); último
    = izquierda;
    for (i = izquierda+1; i <= derecha;
        i++) si ((*comp)(v[i], v[left])
        < 0)
            permuta(v, ++último,
    i); swap(v, izquierda,
    última);
    qsort(v, izquierda, último-1,
    comp); qsort(v, último+1,
    derecha, comp);
}
```

Las declaraciones deben ser estudiadas con cierto cuidado. El cuarto parámetro de `qsort` es

```
int (*comp)(void *, void *)
```

que dice que `comp` es un puntero a una función que tiene dos `argumentos void *` y devuelve un
`int`.

El uso de `comp` en la línea

```
if ((*comp)(v[i], v[left]) < 0)
```

es coherente con la declaración: `comp` es un puntero a una función, `*comp` es la función, y

```
(*comp)(v[i], v[izquierda])
```

es el llamado a ello. Los paréntesis son necesarios para que los componentes se asocien correctamente; sin ellos,

```
int *comp(void *, void *)    /*INCORRECTO*/
```

dice que `comp` es una función que devuelve un puntero a un `int`, que es muy diferente.

Ya hemos mostrado `strcmp`, que compara dos cadenas. Aquí está `numcmp`, que compara dos cadenas en un valor numérico inicial, calculado llamando a `atof`:

```
#include <stdlib.h>

/* numcmp: compara s1 y s2 numéricamente */
int numcmp(char *s1, char *s2)
{
    doble v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        retorno -1;
    de lo contrario,
    si (v1 > v2)
        retorno 1;
    más
        devuelve 0;
}
```

La función de `intercambio`, que intercambia dos punteros, es idéntica a la que presentamos anteriormente en el capítulo, excepto que las declaraciones se cambian a `void *`.

```
void swap(void *v[], int i, int j;)
{
    vacío *temp;

    temp = v[i];
    v[i] = v[j];
    v[j]       =
    temperatura;
}
```

Se puede agregar una variedad de otras opciones al programa de clasificación; Algunos hacen ejercicios desafiantes.

Ejercicio 5-14. Modifique el programa de ordenación para que maneje un `indicador -r`, que indica la ordenación en orden inverso (decreciente). Asegúrese de que `-r` funciona con `-n`.

Ejercicio 5-15. Agregue la opción `-f` para doblar las mayúsculas y minúsculas, de modo que no se hagan distinciones de mayúsculas y minúsculas durante la ordenación; por ejemplo, `a` y `A` se comparan igual.

Ejercicio 5-16. Agregue la `opción -d` ("orden de directorio"), que hace comparaciones solo en letras, números y espacios en blanco. Asegúrese de que funcione junto con `-f`.

Ejercicio 5-17. Agregue una capacidad de búsqueda de campos, de modo que la clasificación se pueda realizar en campos dentro de líneas, cada campo ordenado de acuerdo con un conjunto independiente de opciones. (El índice de este libro se clasificó con `-df` para la categoría de índice y `-n` para los números de página).

5.12 Declaraciones complicadas

C a veces es criticado por la sintaxis de sus declaraciones, particularmente aquellas que involucran punteros a funciones. La sintaxis es un intento de hacer que la declaración y el uso coincidan; Funciona bien para los casos simples, pero puede ser confuso para los más difíciles, porque las declaraciones no se pueden leer de izquierda a derecha y porque los paréntesis se usan en exceso. La diferencia entre

y

```
int
*f()
;               /* f: función que devuelve el puntero a int */
```

```
   int (*pf)();    /* pf: puntero a la función que devuelve int */
```

Ilustra el problema: `*` es un operador de prefijo y tiene menor prioridad que `()`, por lo que los paréntesis son necesarios para forzar la asociación adecuada.

Aunque en la práctica rara vez se producen declaraciones realmente complicadas, es importante saber entenderlas y, si es necesario, crearlas. Una buena manera de sintetizar declaraciones es en pequeños pasos con `typedef`, que se discute en la Sección 6.7. Como alternativa, en esta sección presentaremos un par de programas que convierten de C válido a una descripción de palabras y viceversa. La palabra descripción se lee de izquierda a derecha.

El primero, `dcl`, es el más complejo. Convierte una declaración C en una descripción de palabras, como en estos ejemplos:

```
char **argv
    argv: puntero a char
int (*daytab)[13]
    daytab: puntero a la matriz[13] de
int int *daytab[13]
    daytab: array[13] de puntero a int
void *comp()
    comp: función que devuelve el puntero al
vacío void (*comp)()
    comp: puntero a la función que devuelve
void char (*(*x())[])()
    x: función que devuelve el puntero a la
    matriz[] del puntero a la función que devuelve
    char
char (*(*x[3])())[5]
    x: array[3] de puntero a función devolviendo
    puntero a array[5] de char
```

`dcl` se basa en la gramática que especifica un declarador, que se detalla con precisión en el Apéndice A, Sección 8.5; esta es una forma simplificada:

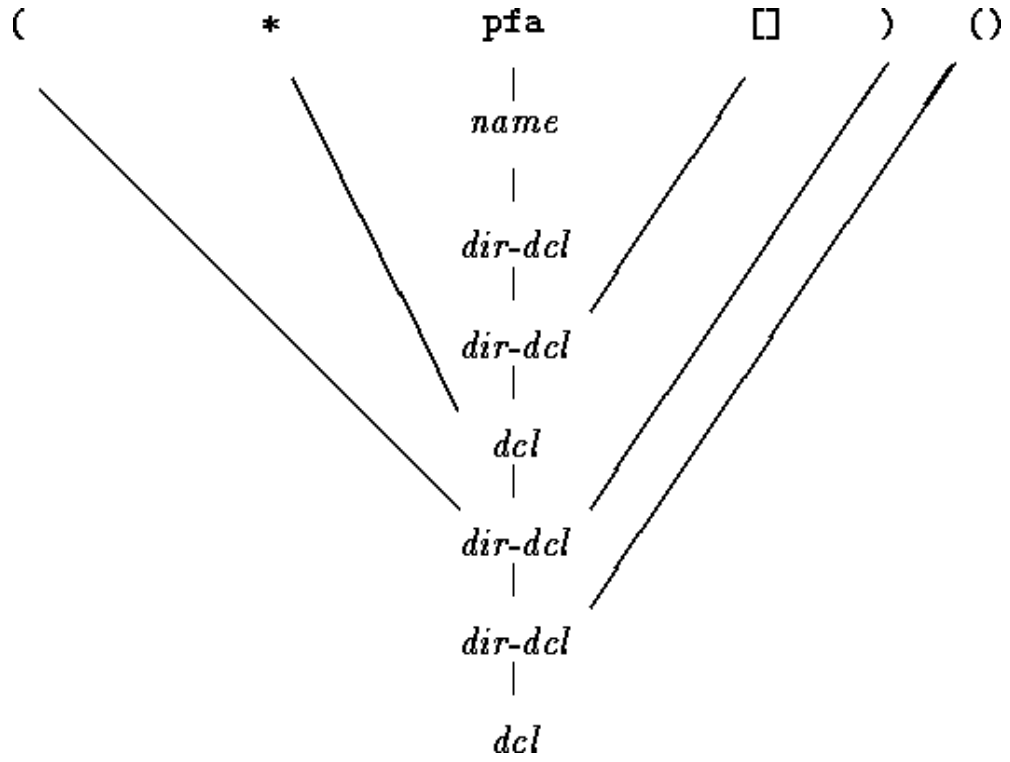
```
Dcl:        Opcional * Direct-DCL
Nombre de Direct-DCL
                  (DCL)
                  directo-dcl()
                  Direct-DCL[tamaño opcional]
```

En palabras, un *dcl* es un *dcl directo*, tal vez precedido por *'s. Un *DCL directo* es un nombre, o un DCL entre paréntesis, o un *DCL directo seguido de* paréntesis, o un *DCL directo* seguido de corchetes con un tamaño opcional.

Esta gramática se puede utilizar para analizar funciones. Por ejemplo, considere este declarador:

```
   (*pfa[])()
```

`PFA` se identificará como un *nombre* y, por lo tanto, como un *DCL directo*. Entonces `pfa[]` también es un *directo-dcl*. Entonces
`*PFA[]` se reconoce como una *DCL*, por lo que `(*PFA[])` es una *DCL directa*. Entonces `(*pfa[])()` es un *dcl directo* y, por lo tanto, un *dcl*. También podemos ilustrar el análisis sintáctico con un árbol como este (donde *direct-dcl* se ha abreviado a *dir-dcl*):



El corazón del programa `dcl` es un par de funciones, `dcl` y `dirdcl`, que analizan una declaración de acuerdo con esta gramática. Debido a que la gramática se define recursivamente, las funciones se llaman entre sí recursivamente a medida que reconocen partes de una declaración; El programa se denomina analizador de descenso recursivo.

```
/* dcl: analizar un declarador
*/ void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; ) /* count *'s */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " puntero a");
}

/* dirdcl: analizar un declarador directo
*/ void dirdcl(void)
{
    tipo int;

    if (tokentype == '(') {/* ( dcl ) */ dcl();

        if (tokentype != ')')
            printf("error: falta )\n");
    } else if (tokentype == NOMBRE) /* nombre de la variable */
        strcpy(nombre, token);
```

```
    más
        printf("error: nombre esperado o (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " función que
        devuelve"); else {
            strcat(fuera, "
            matriz");
            strcat(fuera, ficha);
            strcat(fuera, " de");
        }
}
```

Dado que los programas están destinados a ser ilustrativos, no a prueba de balas, existen restricciones significativas sobre `dcl`. Solo puede manejar un tipo de datos simple line `char` o `int`. No controla los tipos de argumentos en las funciones ni los calificadores como `const`. Los espacios en blanco espurios lo confunden. No hace mucha recuperación de errores, por lo que las declaraciones no válidas también lo confundirán. Estas mejoras se dejan como ejercicios.

Estas son las variables globales y la rutina principal:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#ডিফাইন Mahtken 100

enumeración { NOMBRE, PARÉNTESIS, CORCHETES };

nulo DCL (nulo);
void dirdcl(nulo);

int gettoken(vacío);
int tokentype;/* tipo del último token      */
Cuatro tokens[maxtoken];/* Última cadena de
token */ cuatro llamados [Maxtoken];/*
Identificador llamado */
tipo de datos char[MAXTOKEN]; /* tipo de datos = char, int,
etc. */ char out[1000];

main() /* convertir la declaración en palabras */
{
    while (gettoken() != EOF) {/* 1st token on line */
        strcpy(datatype, token); /* es el tipo de datos
        */ out[0] = '\0';
        dcl();       /* analizar el resto de
        la línea */ if (tokentype != '\n')
            printf("error de sintaxis\n");
        printf("%s: %s %s\n", nombre, salida, tipo de datos);
    }
    devuelve 0;
}
```

La función `gettoken` omite los espacios en blanco y las tabulaciones, luego encuentra el siguiente token en la entrada; un "token" es un nombre, un par de paréntesis, un par de corchetes que tal vez incluyan un número, o cualquier otro carácter individual.

```
int gettoken(void) /* devuelve el siguiente token */
{
    int c, getch(vacío);
    void ungetch(int);
    char *p = ficha;

    while ((c = getch()) == ' ' || c == '\t')
```

```
            ;
        if (c == '(') {
            if ((c = getch()) == ')') {
                strcpy(token, "()");
                return tokentype = PARENS;
            } else {
                ungetch(c);
                return tokentype = '(';
            }
        } else if (c == '[') {
            para (*p++ = c; (*p++ = getch()) != ']'; )
                ;
            *p = '\0';
            return tokentype = PARÉNTESIS;
        } else if (isalpha(c)) {
            for (*p++ = c; isalnum(c = getch()); )
                *p++ = c;
            *p = '\0';
            ungetch(c);
            return tokentype = NOMBRE;
        } de lo contrario
            return tokentype = c;

    }
```

`getch` y `ungetch` se analizan en el capítulo 4.

Ir en la otra dirección es más fácil, sobre todo si no nos preocupamos por generar paréntesis redundantes. El programa `undcl` convierte una descripción de palabras como "x es una función que devuelve un puntero a una matriz de punteros a funciones que devuelven `char`", que expresaremos como

```
    x () * [] * () char
```

Para

```
    char (*(*x())[])()
```

La sintaxis de entrada abreviada nos permite reutilizar la `función gettoken`. `UNDCL` también utiliza las mismas variables externas que `DCL`.

```
    /* undcl: convertir descripciones de palabras en
    declaraciones */ main()
    {
        tipo int;
        Sir Temp[Mastken];

        while (gettoken() != EOF) {
            strcpy(out, token);
            while ((type = gettoken()) != '\n')
                if (type == PARENS || type == BRACKETS)
                    strcat(out, token);
                else if (type == '*') {
                    sprintf(temp, "(*%s)", out);
                    strcpy(out, temp);
                } else if (type == NAME) {
                    sprintf(temp, "%s %s", token, out);
                    strcpy(out, temp);
                } de lo contrario
                    printf("entrada no válida en %s\n", token);
        }
        devuelve 0;
    }
```

Ejercicio 5-18. Haga que `dcl` se recupere de los errores de entrada.

Ejercicio 5-19. Modifique `undcl` para que no agregue paréntesis redundantes a las declaraciones.

Ejercicio 5-20. Expanda `dcl` para controlar declaraciones con tipos de argumentos de función, calificadores como `const`, etc.

Capítulo 6 - Estructuras

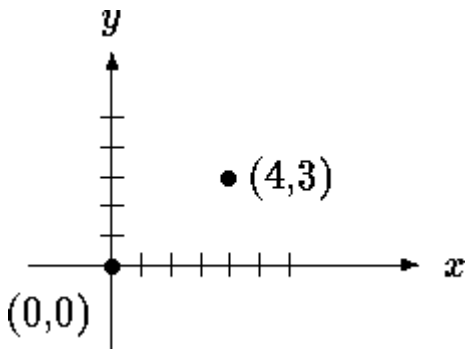
Una estructura es una colección de una o más variables, posiblemente de diferentes tipos, agrupadas bajo un solo nombre para un manejo conveniente. (Las estructuras se llaman "registros" en algunos idiomas, especialmente Pascal). Las estructuras ayudan a organizar datos complicados, particularmente en programas grandes, porque permiten que un grupo de variables relacionadas se traten como una unidad en lugar de como entidades separadas.

Un ejemplo tradicional de estructura es el registro de nómina: un empleado se describe mediante un conjunto de atributos como nombre, dirección, número de seguro social, salario, etc. Algunos de estos, a su vez, podrían ser estructuras: un nombre tiene varios componentes, al igual que una dirección e incluso un salario. Otro ejemplo, más típico de C, proviene de los gráficos: un punto es un par de coordenadas, un rectángulo es un par de puntos, y así sucesivamente.

El principal cambio realizado por el estándar ANSI es definir la asignación de estructuras: las estructuras pueden copiarse y asignarse, pasarse a funciones y ser devueltas por funciones. Esto ha sido compatible con la mayoría de los compiladores durante muchos años, pero las propiedades ahora están definidas con precisión. Ahora también se pueden inicializar estructuras y matrices automáticas.

6.1 Conceptos básicos de estructuras

Vamos a crear algunas estructuras adecuadas para los gráficos. El objeto básico es un punto, que supondremos que tiene una coordenada x y una coordenada y, ambas números enteros.



Los dos componentes se pueden colocar en una estructura declarada de la siguiente manera:

```
struct point
    { int x;
    int y;
};
```

La palabra clave `struct` introduce una declaración de estructura, que es una lista de declaraciones encerradas entre llaves. Un nombre opcional llamado *etiqueta de estructura* puede seguir a la palabra `struct` (como con `point` aquí). La etiqueta nombra este tipo de estructura y se puede utilizar posteriormente como abreviatura para la parte de la declaración entre llaves.

Las variables nombradas en una estructura se denominan *miembros*. Un miembro de estructura o etiqueta y una variable ordinaria (es decir, no miembro) pueden tener el mismo nombre sin conflicto, ya que siempre se pueden distinguir por el contexto. Además, los mismos nombres de miembros pueden aparecer en diferentes estructuras, aunque por una cuestión de estilo normalmente se usarían los mismos nombres solo para objetos estrechamente relacionados.

Una declaración struct define un tipo. La llave derecha que termina la lista de miembros puede ir seguida de una lista de variables, al igual que para cualquier tipo básico. Es decir

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

es sintácticamente análogo a

```
int x, y, z;
```

en el sentido de que cada instrucción declara que `x`, `y` y `z` son variables del tipo nombrado y hace que se reserve espacio para ellas.

Una declaración de estructura que no va seguida de una lista de variables no reserva almacenamiento; simplemente describe una plantilla o forma de una estructura. Sin embargo, si la declaración está etiquetada, la etiqueta se puede utilizar más adelante en las definiciones de instancias de la estructura. Por ejemplo, dada la declaración del `punto` anterior,

```
punto de estructura pt;
```

Define una variable `pt` que es una estructura de tipo `struct point`. Una estructura se puede inicializar siguiendo su definición con una lista de inicializadores, cada uno de los cuales es una expresión constante, para los miembros:

```
struct maxpt = { 320, 200 };
```

Una estructura automática también se puede inicializar por asignación o llamando a una función que devuelva una estructura del tipo correcto.

A un miembro de una estructura determinada se hace referencia en una expresión mediante una construcción de la forma

nombre-estructura.miembro

El operador de miembro de estructura "." conecta el nombre de la estructura y el nombre del miembro. Para imprimir las coordenadas del punto `pt`, por ejemplo,

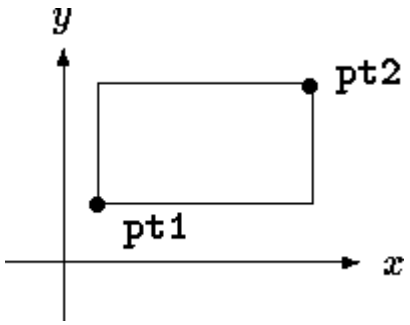
```
printf("%d,%d", pt.x, pt.y);
```

o para calcular la distancia desde el origen (0,0) a `pt`,

```
double dist, sqrt(double);

dist = sqrt((doble)pt.x * pt.x + (doble)pt.y * pt.y);
```

Las estructuras se pueden anidar. Una representación de un rectángulo es un par de puntos que denotan las esquinas diagonalmente opuestas:




```
struct rect {
    Punto de
    estructura PT1;
    Punto de
    estructura PT2;
};
```

La estructura `recta` contiene dos estructuras puntuales. Si declaramos `screen` como

```
Pantalla de estructura recta;
```

entonces

```
pantalla.pt1.x
```

Se refiere a la *coordenada* x del miembro `pt1` de la `pantalla`.

6.2 Estructuras y funciones

Las únicas operaciones legales en una estructura son copiarla o asignarla como una unidad, tomar su dirección con `&` y acceder a sus miembros. La copia y la asignación incluyen el paso de argumentos a las funciones y la devolución de valores de las funciones. Es posible que las estructuras no se comparen. Una estructura puede inicializarse mediante una lista de valores de miembros constantes; Una estructura automática también puede ser inicializada por una asignación.

Investiguemos las estructuras escribiendo algunas funciones para manipular puntos y rectángulos. Hay al menos tres enfoques posibles: pasar componentes por separado, pasar una estructura completa o pasar un puntero a ella. Cada uno tiene sus puntos buenos y sus puntos malos.

La primera función, `makepoint`, tomará dos números enteros y devolverá una estructura de `puntos` :

```
/* makepoint: hacer un punto a partir de los componentes
x e y */ struct point makepoint(int x, int y)
{
    Temperatura del punto de estructura;

    temp.x = x;
    temp.y = y;
    temperatura
    de retorno;
}
```

Observe que no hay ningún conflicto entre el nombre del argumento y el miembro con el mismo nombre; De hecho, la reutilización de los nombres acentúa la relación.

`Makepoint` ahora se puede usar para inicializar cualquier estructura dinámicamente o para proporcionar argumentos de estructura a una función:

Pantalla de estructura recta;

```
punto de estructura medio;
punto de estructura makepoint(int, int);

pantalla.pt1 = punto de
creación(0,0); screen.pt2 = punto de
creación (XMAX, YMAX);
medio = makepoint((screen.pt1.x + screen.pt2.x)/2,
                  (pantalla.pt1.y + pantalla.pt2.y)/2);
```

El siguiente paso es un conjunto de funciones para hacer aritmética sobre puntos. Por ejemplo

```
/* addpoints: añade dos puntos */
struct addpoint(punto de estructura p1, punto de estructura p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    Retorno P1;
}
```

Aquí, tanto los argumentos como el valor devuelto son estructuras. Incrementamos los componentes en `p1` en lugar de usar una variable temporal explícita para enfatizar que los parámetros de estructura se pasan por valor como cualquier otro.

Como otro ejemplo, la función `ptinrect comprueba` si un punto está dentro de un rectángulo, donde hemos adoptado la convención de que un rectángulo incluye sus lados izquierdo e inferior, pero no sus lados superior y derecho:

```
/* ptinrect: devuelve 1 si p en r, 0 si no */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Esto supone que el rectángulo se presenta en una forma estándar donde las coordenadas `pt1` son menores que las coordenadas `pt2`. La siguiente función devuelve un rectángulo garantizado en forma canónica:

```
#define min(a, b) (a) < (b) ? (a) : (b))
#define máx. (a, b) (a) > (b) ? (a) : (b))

/* canonrect: canonicalizar las coordenadas del
rectángulo */ struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    temperatura de retorno;
}
```

Si se va a pasar una estructura grande a una función, generalmente es más eficiente pasar un puntero que copiar toda la estructura. Los punteros de estructura son como punteros a variables ordinarias.
La declaración

```
punto de estructura *pp;
```

Dice que `pp` es un puntero a una estructura de tipo `struct point`. Si `pp` apunta a una estructura de puntos,
`*pp` es la estructura, y `(*pp).x` y `(*pp).y` son los miembros. Para usar `pp`, podríamos escribir, por ejemplo,

```
origen del punto de estructura, *pp;
```

```
pp = &origen;
printf("el origen es (%d,%d)\n", (*pp).x, (*pp).y);
```

Los paréntesis son necesarios en `(*pp).x` porque la prioridad del operador miembro de estructura `.` es mayor que `*`. La expresión `*pp.x` significa `*(pp.x)`, lo cual es ilegal aquí porque `x` no es un puntero.

Los punteros a las estructuras se utilizan con tanta frecuencia que se proporciona una notación alternativa como taquigrafía. Si `p` es un puntero a una estructura, entonces

```
p->miembro-de-estructura
```

se refiere al miembro en particular. Así que podríamos escribir en su lugar

```
printf("el origen es (%d,%d)\n", pp->x, pp->y);
```

Ambos `.` y `->` asociar de izquierda a derecha, por lo que si tenemos

```
struct rect r, *rp = &r;
```

Entonces estas cuatro expresiones son equivalentes:

```
R.pt1.x
RP->pt1.x
(R.pt1).x
(RP->pt1).x
```

Los operadores de estructura `.` y `->`, junto con `()` para las llamadas a funciones y `[]` para los subíndices, se encuentran en la parte superior de la jerarquía de precedencia y, por lo tanto, se unen muy estrechamente. Por ejemplo, dada la declaración

```
struct {
    Carril de
    la posada;
    Cuatro
    *pajitas;
} *p;
```

entonces

```
++p->len
```

incrementa `len`, no `p`, porque el paréntesis implícito es `++(p->len)`. Los paréntesis se pueden usar para modificar el enlace: `(++p)->len` se incrementa `p` antes de acceder a `len`, y `(p++)->len` se incrementa `p` después. (Este último conjunto de paréntesis es innecesario).

De la misma manera, `*p->str` obtiene lo que `str` señala; `*p->str++` incrementa `str` después de acceder a lo que sea que apunte (al igual que `*s++`); `(*p->str)++` incrementa lo que apunta `str`; y `*p++->str` incrementa `p` después de acceder a lo que `apunta` str.

6.3 Matrices de estructuras

Considere la posibilidad de escribir un programa para contar las apariciones de cada palabra clave C. Necesitamos una matriz de cadenas de caracteres para contener los nombres y una matriz de números enteros para los recuentos. Una posibilidad es usar dos matrices paralelas, `keyword` y `keycount`, como en

```
char *palabra
clave[NKEYS]; int
keycount[NKEYS];
```

Pero el hecho mismo de que las matrices sean paralelas sugiere una organización diferente, una matriz de estructuras. Cada palabra clave es un par:

```
char
*palabra; int
cout;
```

y hay una matriz de pares. La declaración de estructura

```
struct key {
    char
    *palabra;
    int count;
} keytab[NKEYS];
```

Declara una clave de tipo de estructura, define una matriz `keytab` de estructuras de este tipo y reserva el almacenamiento para ellas. Cada elemento de la matriz es una estructura. Esto también podría escribirse

```
struct key {
    char
    *palabra;
    int count;
};

tecla struct keytab[NKEYS];
```

Dado que la tecla de estructura contiene un conjunto constante de nombres, es más fácil convertirla en una variable externa e inicializarla de una vez por todas cuando esté definida. La inicialización de la estructura es análoga a las anteriores: la definición va seguida de una lista de inicializadores entre llaves:

```
struct key {
    char
    *palabra;
    int count;
} keytab[] = {
    "auto", 0,
    "descanso", 0,
    "caso", 0,
    "char", 0,
    "const", 0,
    "continuar", 0,
    "predeterminado", 0,
    /* ... */
    "sin firmar", 0,
    "vacío", 0,
    "volátil", 0,
    "mientras", 0
};
```

Los inicializadores se enumeran en pares correspondientes a los miembros de la estructura. Sería más preciso encerrar los inicializadores para cada "fila" o estructura entre llaves, como en

```
{ "auto", 0 },
{ "descanso", 0 },
{ "caso", 0 },
...
```

Pero las llaves internas no son necesarias cuando los inicializadores son variables simples o cadenas de caracteres, y cuando todos están presentes. Como de costumbre, se calculará el número de entradas en la `pestaña de teclas` de la matriz si los inicializadores están presentes y el `[]` se deja vacío.

El programa de conteo de palabras clave comienza con la definición de `keytab`. La rutina principal lee la entrada llamando repetidamente a una función `getword` que obtiene una palabra a la vez. Cada palabra se busca en `keytab` con una versión de la función de búsqueda binaria que escribimos en el capítulo 3. La lista de palabras clave debe ordenarse en orden creciente en la tabla.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
```

```
#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);
```

```
/* cuenta C palabras
clave */ main()
{
    int n;
    palabra char[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count, keytab[n].word);
    devuelve 0;
}

/* binsearch: buscar palabra en la pestaña[0]...
tab[n-1] */ int binsearch(char *word, struct key
tab[], int n)
{
    int cond;
    int bajo, alto, medio;

    bajo = 0;
    alto = n - 1;
    while (bajo <= alto) {
        medio = (bajo+alto) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        más
            retorno medio;
    }
    retorno -1;
}
```

Mostraremos la función `getword` en un momento, por ahora basta con decir que cada llamada a `getword` encuentra una palabra, que se copia en la matriz nombrada como su primer argumento.

La cantidad `NKEYS` es el número de palabras clave en keytab. Aunque podríamos contar esto a mano, es mucho más fácil y seguro hacerlo a máquina, especialmente si la lista está sujeta a cambios. Una posibilidad sería terminar la lista de inicializadores con un puntero nulo y, a continuación, recorrer la `tecla hasta` encontrar el final.

Pero esto es más de lo necesario, ya que el tamaño de la matriz se determina completamente en tiempo de compilación. El tamaño de la matriz es el tamaño de una entrada multiplicado por el número de entradas, por lo que el número de entradas es simplemente

Tamaño de `la tecla TAB` / *Tamaño de* `la clave de estructura`

C proporciona un operador unario en tiempo de compilación denominado `sizeof` que se puede usar para calcular el tamaño de cualquier objeto. Las expresiones

```
Tamañodel objeto
```

y

```
sizeof (nombre del tipo)
```

Produce un número entero igual al tamaño del objeto o tipo especificado en bytes. (Estrictamente, `tamañode`
Genera un valor entero sin signo cuyo tipo, `size_t`, se define en el encabezado `<stddef.h>`.) Un objeto puede ser una variable, una matriz o una estructura. Un nombre de tipo puede ser el nombre de un tipo básico, como `int` o `double`, o un tipo derivado, como una estructura o un puntero.

En nuestro caso, el número de palabras clave es el tamaño de la matriz dividido por el tamaño de un elemento. Este cálculo se utiliza en una `instrucción #define` para establecer el valor de `NKEYS`:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Otra forma de escribir esto es dividir el tamaño de la matriz por el tamaño de un elemento específico:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0])
```

Esto tiene la ventaja de que no es necesario cambiarlo si cambia el tipo.

Un `sizeof` no se puede usar en una `línea #if`, ya que el preprocesador no analiza los nombres de tipo. Pero la expresión en el `#define` no es evaluada por el preprocesador, por lo que el código aquí es legal.

Ahora para la función `getword`. Hemos escrito un texto más general de lo necesario para este programa, pero no es complicado. `getword` obtiene la siguiente "palabra" de la entrada, donde una palabra es una cadena de letras y dígitos que comienzan con una letra, o un solo carácter de espacio que no está en blanco. El valor de la función es el primer carácter de la palabra, o `EOF` para el final del archivo, o el propio carácter si no es alfabético.

```
/* getword: obtener la siguiente palabra o carácter de
la entrada */ int getword(char *word, int lim)
{
    int c, getch(vacío);
    void ungetch(int);
    char *w = palabra;

    while (isspace(c = getch()))
        ;
    si (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        Retorno C;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            quebrar;
        }
    *w = '\0';
    palabra de
    retorno[0];
}
```

`getword` usa el `getch` y `ungetch` que escribimos en el Capítulo 4. Cuando se detiene la recolección de un token alfanumérico, `getword` ha ido demasiado lejos en un carácter. La llamada a `ungetch` empuja ese carácter de nuevo en la entrada para la siguiente llamada. `getword` también usa `isspace` para omitir espacios en blanco, `isalpha` para identificar letras e `isalnum` para identificar letras y números; todos son del encabezado estándar `<ctype.h>`.

Ejercicio 6-1. Nuestra versión de `getword` no maneja correctamente los guiones bajos, las constantes de cadena, los comentarios o las líneas de control del preprocesador. Escribe una versión mejor.

6.4 Punteros a estructuras

Para ilustrar algunas de las consideraciones involucradas con punteros y matrices de estructuras, escribamos de nuevo el programa de conteo de palabras clave, esta vez usando punteros en lugar de índices de matriz.

La declaración externa de `keytab` no necesita cambiar, pero `main` y `binsearch` sí necesitan modificación.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* contar las palabras clave C; versión
del puntero */ main()
{
    palabra char[MAXWORD];
    tecla de estructura *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->cuenta, p-
    >palabra); devuelve 0;
}

/* binsearch: buscar palabra en la pestaña[0]... tab[n-1] */
struct key *binsearch(char *word, struck key *tab, int n)
{
    int cond;
    tecla struct *low = &tab[0];
    clave de estructura *high =
    &tab[n]; tecla de estructura
    *mid;

    while (bajo < alto) {
        medio = bajo + (alto-bajo) / 2;
        if ((cond = strcmp(palabra, medio >palabra))
            < 0) alto = medio;
        else if (cond > 0)
            low = mid + 1;
        más
            retorno medio;
    }
    devuelve NULL;
}
```

Hay varias cosas dignas de mención aquí. En primer lugar, la declaración de `binsearch` debe indicar que devuelve un puntero a `struct` key en lugar de un entero; esto se declara tanto en la función prototype como en `binsearch`. Si `binsearch` encuentra la palabra, devuelve un puntero a ella; si falla, devuelve `NULL`.

En segundo lugar, ahora se accede a los elementos de `keytab` mediante punteros. Esto requiere cambios significativos en `binsearch`.

Los inicializadores para `low` y `high` ahora son punteros al principio y justo después del final de la tabla.

El cálculo del elemento intermedio ya no puede ser simplemente

```
medio = (bajo+alto) / 2 /*INCORRECTO*/
```

porque la adición de punteros es ilegal. Sin embargo, la sustracción es legal, por lo que `el número de elementos es alto-bajo` y, por lo tanto,

```
medio = bajo + (alto-bajo) / 2
```

Establece `el elemento medio` a medio camino entre `el bajo` y `el alto`.

El cambio más importante es ajustar el algoritmo para asegurarse de que no genera un puntero ilegal o intenta acceder a un elemento fuera de la matriz. El problema es que `&tab[- 1]` y `&tab[n]` están fuera de los límites de la pestaña de la matriz. Lo primero es estrictamente ilegal, y es ilegal desreferenciar lo segundo. Sin embargo, la definición del lenguaje garantiza que la aritmética de punteros que implica el primer elemento más allá del final de una matriz (es decir, `&tab[n]`) funcionará correctamente.

En `general` escribimos

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Si `p` es un puntero a una estructura, la aritmética en `p` tiene en cuenta el tamaño de la estructura, por lo que `p++` incrementa `p` en la cantidad correcta para obtener el siguiente elemento de la matriz de estructuras, y la prueba detiene el bucle en el momento adecuado.

Sin embargo, no asumas que el tamaño de una estructura es la suma de los tamaños de sus miembros. Debido a los requisitos de alineación para diferentes objetos, puede haber "agujeros" sin nombre en una estructura. Así, por ejemplo, si un `char` es de un byte y un `int` de cuatro bytes, la estructura

```
struct {
    char c;
    Int I;
};
```

bien podría requerir ocho bytes, no cinco. El operador `sizeof` devuelve el valor adecuado.

Por último, un aparte sobre el formato del programa: cuando una función devuelve un tipo complicado como un puntero de estructura, como en

```
struct key *binsearch(char *word, struct key *tab, int n)
```

El nombre de la función puede ser difícil de ver y de encontrar con un editor de texto. En consecuencia, a veces se utiliza un estilo alternativo:

```
clave de estructura *
binsearch(char *palabra, clave de estructura *tab, int n)
```

Esto es una cuestión de gustos personales; Elija la forma que le guste y manténgala firme.

6.5 Estructuras autorreferenciales

Supongamos que queremos manejar el problema más general de contar las apariciones de *todas las* palabras en alguna entrada. Dado que la lista de palabras no se conoce de antemano, no podemos ordenarla convenientemente y usar una búsqueda binaria. Sin embargo, no podemos hacer una búsqueda lineal de cada palabra a medida que llega, para ver si ya se ha visto; El programa tomaría demasiado tiempo. (Más precisamente, es probable que su tiempo de ejecución crezca cuadráticamente con el número de palabras de entrada). ¿Cómo podemos organizar los datos para copiarlos de manera eficiente con una lista o palabras arbitrarias?

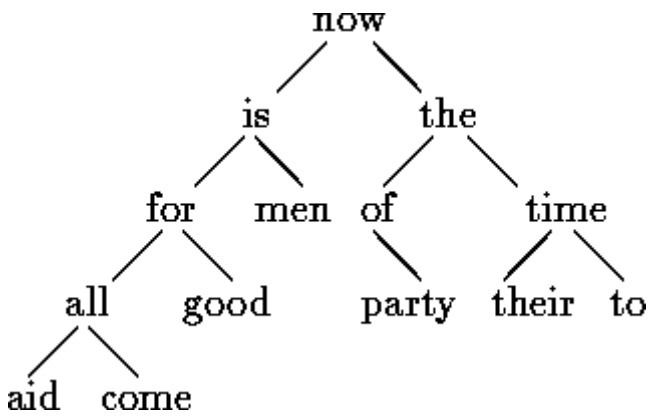
Una solución es mantener ordenado el conjunto de palabras visto hasta el momento en todo momento, colocando cada palabra en su posición correcta en el orden en que llega. Sin embargo, esto no debe hacerse cambiando las palabras en una matriz lineal, ya que eso también lleva demasiado tiempo. En su lugar, usaremos una estructura de datos llamada *árbol binario*.

El árbol contiene un "nodo" por palabra distinta; Cada nodo contiene

- Un puntero al texto de la palabra,
- Un recuento del número de ocurrencias,
- Un puntero al nodo secundario izquierdo,
- Puntero al nodo secundario derecho.

Ningún nodo puede tener más de dos hijos; puede tener solo cero o uno.

Los nodos se mantienen de modo que en cualquier nodo el subárbol izquierdo contiene solo palabras que son lexicográficamente menores que la palabra en el nodo, y el subárbol derecho contiene solo palabras que son mayores. Este es el árbol de la frase "ahora es el momento de que todos los hombres de bien acudan en ayuda de su partido", tal como se construye insertando cada palabra a medida que se encuentra:



Para averiguar si una nueva palabra ya está en el árbol, comience desde la raíz y compare la nueva palabra con la palabra almacenada en ese nodo. Si coinciden, la pregunta se responde afirmativamente. Si el nuevo registro es menor que la palabra del árbol, continúe buscando en el elemento secundario de la izquierda, de lo contrario, en el elemento secundario de la derecha. Si no hay ningún elemento secundario en la dirección requerida, la nueva palabra no está en el árbol y, de hecho, el espacio vacío es el lugar adecuado para agregar la nueva palabra. Este proceso es recursivo, ya que la búsqueda de cualquier nodo utiliza una búsqueda de uno de sus hijos. En consecuencia, las rutinas recursivas para la inserción e impresión serán más naturales.

Volviendo a la descripción de un nodo, lo más conveniente es representarlo como una estructura con cuatro componentes:

```
struct tnode {     /* el nodo del árbol: */
    char *palabra;        /* apunta al texto */
    int count;/* número de ocurrencias      */ struct
    tnode *left;/* left child */
    struct tnode *right; /* niño derecho */
};
```

Esta declaración recursiva de un nodo puede parecer extravagante, pero es correcta. Es ilegal que una estructura contenga una instancia de sí misma, pero

```
 struct tnode *izquierda;
```

declara `que left` es un puntero a un `tnode`, no a un `tnode` en sí.

De vez en cuando, se necesita una variación de las estructuras autorreferenciales: dos estructuras que se refieren la una a la otra. La forma de manejar esto es:

```
struct t {
    ...
    estructura s *p;      /* p señala una s */
};
struct s {
    ...
    estructura t *q;      /* q señala a T */
};
```

El código para todo el programa es sorprendentemente pequeño, dado un puñado de rutinas de apoyo como `getword` que ya hemos escrito. La rutina principal lee palabras con `getword` y las instala en el árbol con `addtree`.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* recuento de frecuencia
de palabras */ main()
{
    struct tnode *raíz;
    palabra
    char[MAXWORD];

    raíz = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(raíz,
    palabra); Huella de árbol (raíz);
    devuelve 0;
}
```

La función `addtree` es recursiva. Una palabra se presenta por `main` al nivel superior (la raíz) del árbol. En cada etapa, esa palabra se compara con la palabra ya almacenada en el nodo y se filtra hasta el subárbol izquierdo o derecho mediante una llamada recursiva a `adtree`. Finalmente, la palabra coincide con algo que ya está en el árbol (en cuyo caso se incrementa el recuento) o se encuentra un puntero nulo, lo que indica que se debe crear un nodo y agregarlo al árbol. Si se crea un nuevo nodo, `addtree` devuelve un puntero a él, que se instala en el nodo principal.

```
struct tnode *talloc(vacío);
```

```
cuatro *strudap(cuatro *);

/* addtree: agrega un nodo con w, en o por debajo
de p */ struct treenode *addtree(struct tnode *p,
char *w)
{
    int cond;

    if (p == NULL) {/* ha llegado una nueva palabra
                        */ p = talloc();/* crear un
        nuevo nodo */
        p->palabra =
        strdup(w); p->
        cuenta = 1;
        p->izquierda = p->derecha = NULO;
    } else if ((cond = strcmp(w, p->palabra)) ==
        0) p->cuenta++;  /* palabra repetida */
    else if (cond < 0)/* menor que en el subárbol
        izquierdo */ p->left = addtree(p->left, w);
    más  /* mayor que en el subárbol derecho */ p->right =
        addtree(p->right, w);
    retorno p;
}
```

El almacenamiento para el nuevo nodo es recuperado por una rutina `talloc`, que devuelve un puntero a un espacio libre adecuado para contener un nodo de árbol, y la nueva palabra es copiada en un espacio oculto por `strdup`. (Discutiremos estas rutinas en un momento). El recuento se inicializa y los dos elementos secundarios se anulan. Esta parte del código se ejecuta solo en las hojas del árbol, cuando se agrega un nuevo nodo. Hemos omitido (imprudentemente) la comprobación de errores en los valores devueltos por `strdup` y `talloc`.

`treeprint` imprime el árbol en orden ordenado; en cada nodo, imprime el subárbol de la izquierda (todas las palabras menores que esta palabra), luego la palabra en sí, luego el subárbol de la derecha (todas las palabras mayores). Si se siente inseguro acerca de cómo funciona la recursividad, simule `la huella del árbol` tal como opera en el árbol que se muestra arriba.

```
/* treeprint: impresión en orden del árbol p
*/ void treeprint (struct tnode *p)
{
    if (p != NULL) {
        Huella de árbol (P->izquierda);
        printf("%4d %s\n", p->cuenta, p-
        >palabra); Huella de árbol (P-
        >derecha);
    }
}
```

Una nota práctica: si el árbol se "desequilibra" porque las palabras no llegan en orden aleatorio, el tiempo de ejecución del programa puede crecer demasiado. En el peor de los casos, si las palabras ya están en orden, este programa hace una costosa simulación de búsqueda lineal. Hay generalizaciones del árbol binario que no sufren de este comportamiento del peor de los casos, pero no las describiremos aquí.

Antes de dejar este ejemplo, también vale la pena hacer una breve digresión sobre un problema relacionado con los asignadores de almacenamiento. Claramente, es deseable que solo haya un asignador de almacenamiento en un programa, aunque asigne diferentes tipos de objetos. Pero si un asignador va a procesar solicitudes de, por ejemplo, punteros a `caracteres y punteros a nodos estructurales, surgen dos preguntas. En primer lugar, ¿cómo cumple el requisito de la mayoría de las máquinas reales de que los objetos de ciertos tipos deben satisfacer las restricciones de alineación (por ejemplo, los números enteros a menudo deben estar ubicados en direcciones pares)?` En segundo lugar, ¿qué declaraciones pueden hacer frente al

hecho de que un asignador debe devolver necesariamente diferentes tipos de punteros?

Por lo general, los requisitos de alineación se pueden satisfacer fácilmente, a costa de un poco de espacio desperdiciado, asegurándose de que el asignador siempre devuelva un puntero que cumpla *con todas las* restricciones de alineación. La `asignación` del Capítulo 5 no garantiza ninguna alineación en particular, por lo que usaremos la función de biblioteca estándar `malloc`, que sí lo hace. En el Capítulo 8 mostraremos una forma de implementar `malloc`.

La cuestión de la declaración de tipos para una función como `malloc` es desconcertante para cualquier lenguaje que se tome en serio su comprobación de tipos. En C, el método adecuado es declarar que `malloc` devuelve un puntero a `void` y, a continuación, forzar explícitamente el puntero al tipo deseado con una conversión. `malloc` y las rutinas relacionadas se declaran en el encabezado estándar `<stdlib.h>`. Por lo tanto, `talloc` se puede escribir como

```
#include <stdlib.h>

/* talloc: hacer un tnode
*/ struct tnode
*talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

`strdup` simplemente copia la cadena dada por su argumento en un lugar seguro, obtenido mediante una llamada a
`Malloc`:

```
Cuatro *Strodap(Cuatro *s)      /* hacer un duplicado de s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 para '\0' */
    if (p != NULL)
        strcpy(p, s);
    retorno p;
}
```

`malloc` devuelve `NULL` si no hay espacio disponible; `strdup` pasa ese valor, dejando el manejo de errores a su llamador.

El almacenamiento obtenido llamando a `malloc` puede ser liberado para su reutilización llamando a `free`; véanse los Capítulos 8 y 7.

Ejercicio 6-2. Escriba un programa que lea un programa C e imprima en orden alfabético cada grupo de nombres de variables que sean idénticos en los primeros 6 caracteres, pero diferentes en algún lugar a partir de entonces. No cuentes palabras dentro de cadenas y comentarios. Haga que 6 sea un parámetro que se pueda establecer desde la línea de comandos.

Ejercicio 6-3. Escriba un correlato que imprima una lista de todas las palabras de un documento y, para cada palabra, una lista de los números de línea en los que aparece. Elimina las palabras irrelevantes como "el"
"y', y así sucesivamente.

Ejercicio 6-4. Escriba un programa que imprima las distintas palabras en su entrada ordenadas en orden decreciente de frecuencia de aparición. Precede a cada palabra por su conteo.

6.6 Búsqueda de tablas

En esta sección escribiremos las entrañas de un paquete de búsqueda de tablas, para ilustrar más aspectos de las estructuras. Este código es típico de lo que se puede encontrar en la

gestión de la tabla de símbolos

rutinas de un procesador de macros o de un compilador. Por ejemplo, considere la instrucción `#define`.
Cuando una línea como

```
#define EN 1
```

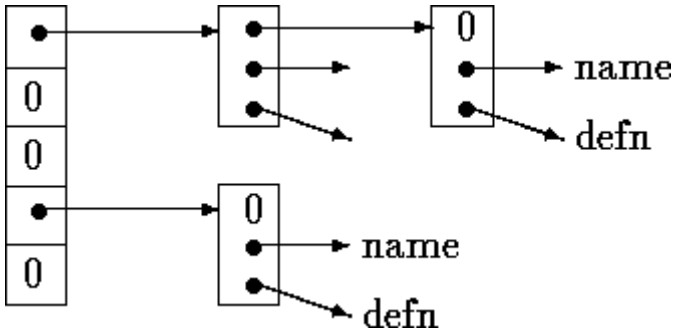
, el nombre `IN` y el texto de reemplazo `1` se almacenan en una tabla. Más tarde, cuando el nombre `IN` aparece en una declaración como

```
estado = EN;
```

debe ser reemplazado por `1`.

Hay dos rutinas que manipulan los nombres y los textos de reemplazo. `install(s,t)` registra el nombre `s` y el texto de reemplazo `t` en una tabla; `s` y `t` son solo cadenas de caracteres. `lookup(s)` busca `s` en la tabla y devuelve un puntero al lugar donde se encontró, o `NULL` si no estaba allí.

El algoritmo es una búsqueda hash: el nombre entrante se convierte en un pequeño número entero no negativo, que luego se usa para indexar en una matriz de punteros. Un elemento de matriz apunta al comienzo de una lista enlazada de bloques que describen los nombres que tienen ese valor hash. Es `NULL` si no se ha establecido ningún nombre con hash en ese valor.



Un bloque de la lista es una estructura que contiene punteros al nombre, al texto de sustitución y al siguiente bloque de la lista. Un puntero siguiente nulo marca el final de la lista.

```
struct nlist {       /* Entrada de tabla: */
    struct nlist *next;/* siguiente entrada en la
    cadena                */ char *nombre;/* nombre
    definido */
    char *defn;           /* texto de reemplazo */
};
```

La matriz de punteros es simplemente

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE]; /* tabla de punteros */
```

La función hash, que se utiliza tanto en `la búsqueda` como en la `instalación`, añade cada valor de carácter de la cadena a una combinación codificada de los anteriores y devuelve el módulo restante del tamaño de la matriz. Esta no es la mejor función hash posible, pero es corta y efectiva.

```
/* hash: valor hash del formulario para
la cadena s */ hash sin signo (char *s)
{
    hashval sin firmar;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
```

```
    return hashval % HASHSIZE;
}
```

La aritmética sin signo garantiza que el valor hash no sea negativo.

El proceso de hashing produce un índice inicial en la matriz `hashtab`; si la cadena se encuentra en cualquier lugar, estará en la lista de bloques que comienzan allí. La búsqueda se realiza mediante `búsqueda`. Si `la búsqueda` encuentra la entrada ya presente, devuelve un puntero a ella; si no, devuelve `NULL`.

```
/* lookup: buscar s en hashtab */ struct
nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;/* encontrado */
    return NULL;/* no encontrado */

}
```

El bucle `for` en la `búsqueda` es el modismo estándar para caminar a lo largo de una lista enlazada:

```
for (ptr = cabeza; ptr != NULL; ptr = ptr->siguiente)
...
```

`install` utiliza `la búsqueda` para determinar si el nombre que se está instalando ya está presente; si es así, la nueva definición reemplazará a la anterior. De lo contrario, se crea una nueva entrada. `install` devuelve `NULL` si por alguna razón no hay espacio para una nueva entrada.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: put (nombre, defn) en hashtab */
struct nlist *install(char *nombre, char *defn)
{
    struct nlist *np;
    hashval sin
    firmar;

    if ((np = lookup(nombre)) == NULL) { /* no
        encontrado */ np = (struct nlist *)
        malloc(sizeof(*np));
        if (np == NULL || (np->nombre = strdup(nombre)) == NULL)
            return NULL;
        hashval = hash(nombre);
        np->next=hashtab[hashval];
        hashtab[hashval] = np;
    } de lo contrario     /* ya está ahí */
        free((void *) np->defn);/*free previous defn */ if
    ((np->defn = strdup(defn)) == NULL)
        devuelve NULL;
    Retorno NP;
}
```

Ejercicio 6-5. Escriba una función `undef` que eliminará un nombre y una definición de la tabla mantenida por lookup e `install`.

Ejercicio 6-6. Implemente una versión simple del procesador `#define` (es decir, sin argumentos) adecuada para su uso con programas C, basada en las rutinas de esta sección. También puede resultarle `útil getch` y `ungetch`.

6.7 Definición de tipo

C proporciona una función denominada typedef para crear nuevos nombres de tipos de datos. Por ejemplo, la declaración

```
typedef int Longitud;
```

hace que el nombre Length sea sinónimo de int. El tipo Length se puede usar en declaraciones, conversiones, etc., exactamente de la misma manera que el tipo int:

```
Longitud len,
maxlen;
Longitud
*longitudes[];
```

Del mismo modo, la declaración

```
typedef char *Cadena;
```

hace que String sea un sinónimo de char * o puntero de carácter, que luego se puede usar en declaraciones y conversiones:

```
Cadena p, lineptr[MAXLINES], alloc(int);
int strcmp(Cadena, Cadena);
p = (cadena) malloc(100);
```

Observe que el tipo que se declara en una definición de tipo aparece en la posición de un nombre de variable, no justo después de la palabra definición de tipo. Sintácticamente, typedef es como las clases de almacenamiento extern, static, etc. Hemos usado nombres en mayúsculas para las definiciones de tipo, para que se destaquen.

Como un ejemplo más complicado, podríamos hacer definiciones de tipo para los nodos de árbol mostrados anteriormente en este capítulo:

```
typedef struct tnode *Treeptr;

typedef struct tnode { /* el nodo del árbol: */
    char *palabra;        /* apunta al texto */
    int count;/* número de ocurrencias      */ struct
    tnode *left;/* left child */
    struct tnode *right; /* niño derecho */
} Nodo de árbol;
```

Esto crea dos nuevas palabras clave de tipo denominadas Treenode (una estructura) y Treeptr (un puntero a la estructura). Entonces el talloc rutinario podría convertirse en

```
Treeptr talloc(vacío)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

Debe enfatizarse que una declaración typedef no crea un nuevo tipo en ningún sentido; simplemente agrega un nuevo nombre para algún tipo existente. Tampoco hay ninguna nueva semántica: las variables declaradas de esta manera tienen exactamente las mismas propiedades que las variables cuyas declaraciones se deletrean explícitamente. En efecto, typedef es como #define, excepto que, dado que es interpretado por el compilador, puede hacer frente a sustituciones textuales que están más allá de las capacidades del preprocesador. Por ejemplo

```
typedef int (*PFI)(char *, char *);
```

crea el tipo PFI, para "puntero a la función (de dos char * argumentos) que devuelve int", que se puede usar en contextos como

```
PFI strcmp, numcmp;
```

en el programa de clasificación del capítulo 5.

Además de las cuestiones puramente estéticas, hay dos razones principales para usar typedefs. La primera es parametrizar un programa frente a problemas de portabilidad. Si se utilizan definiciones de tipos para tipos de datos que

puede depender de la máquina, solo es `necesario cambiar las definiciones de tipos` cuando se mueve el programa. Una situación común es usar nombres de definición de tipo para varias cantidades de enteros y, a continuación, hacer un conjunto adecuado de opciones `de corto`, `int` y `largo` para cada máquina host. Tipos como `size_t` y `ptrdiff_t` de la biblioteca estándar son ejemplos.

El segundo propósito de `typedefs es proporcionar una mejor documentación para un programa: un tipo llamado Treeptr` puede ser más fácil de entender que uno declarado solo como puntero a una estructura complicada.

6.8 Uniones

Una *unión* es una variable que puede contener (en diferentes momentos) objetos de diferentes tipos y tamaños, y el compilador realiza un seguimiento de los requisitos de tamaño y alineación. Las uniones proporcionan una forma de manipular diferentes tipos de datos en una sola área de almacenamiento, sin incrustar ninguna información dependiente de la máquina en el programa. Son análogos a los registros variantes en pascal.

Como ejemplo que se puede encontrar en un administrador de tablas de símbolos del compilador, supongamos que una constante puede ser un `int`, un `float` o un puntero de carácter. El valor de una constante determinada debe almacenarse en una variable del tipo adecuado, pero es más conveniente para la administración de tablas si el valor ocupa la misma cantidad de almacenamiento y se almacena en el mismo lugar, independientemente de su tipo. Este es el propósito de una unión: una sola variable que puede contener legítimamente cualquiera de uno de varios tipos. La sintaxis se basa en estructuras:

```
Unión u_tag {
    int ival;
    flotador
    fval; char
    *sval;
} u;
```

La variable `u` será lo suficientemente grande como para contener el mayor de los tres tipos; el tamaño específico depende de la implementación. Cualquiera de estos tipos se puede asignar a `u` y, a continuación, se puede usar en expresiones, siempre que el uso sea coherente: el tipo recuperado debe ser el tipo almacenado más recientemente. Es responsabilidad del programador realizar un seguimiento de qué tipo está almacenado actualmente en una unión; Los resultados dependen de la implementación si algo se almacena como un tipo y se extrae como otro.

Desde el punto de vista sintáctico, se accede a los miembros de un sindicato como

nombre-de-la-unión`.`*miembro*

o

union-pointer`->member`

al igual que para las estructuras. Si la variable `utype` se usa para realizar un seguimiento del tipo actual almacenado en `u`, entonces se podría ver código como

```
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
    printf("%f\n", u.fval);
```

```
if (utype == STRING)
    printf("%s\n", u.sval);
más
    printf("tipo incorrecto %d en utype\n", utype);
```

Las uniones pueden ocurrir dentro de estructuras y matrices, y viceversa. La notación para acceder a un miembro de una unión en una estructura (o viceversa) es idéntica a la de las estructuras anidadas. Por ejemplo, en la matriz de estructura definida por

```
struct {
    Cuatro*nombr
    ados;
    Banderas
    int; Int
    Utype; Unión
    {
        int ival;
        flotador
        fval; char
        *sval;
    } u;
} symtab[NSYM];
```

El miembro `IVAL` se denomina

```
symtab[i].u.ival
```

y el primer carácter de la cadena `sval` por cualquiera de los

```
*symtab[i].u.sval

symtab[i].u.sval[0]
```

En efecto, una unión es una estructura en la que todos los miembros tienen un desplazamiento cero desde la base, la estructura es lo suficientemente grande como para contener el miembro "más ancho" y la alineación es adecuada para todos los tipos de la unión. Se permiten las mismas operaciones en las uniones que en las estructuras: asignación o copia como unidad, toma de la dirección y acceso a un miembro.

Una unión solo puede inicializarse con un valor del tipo de su primer miembro; por lo tanto, la unión `u`
descrito anteriormente solo se puede inicializar con un valor entero.

El asignador de almacenamiento del Capítulo 8 muestra cómo se puede utilizar una unión para forzar la alineación de una variable en un tipo determinado de límite de almacenamiento.

6.9 Campos de bits

Cuando el espacio de almacenamiento es escaso, puede ser necesario empaquetar varios objetos en una sola palabra de máquina; Un uso común es un conjunto de marcas de un solo bit en aplicaciones como las tablas de símbolos del compilador. Los formatos de datos impuestos externamente, como las interfaces a los dispositivos de hardware, también suelen requerir la capacidad de llegar a partes de una palabra.

Imagine un fragmento de un compilador que manipula una tabla de símbolos. Cada identificador de un programa tiene cierta información asociada, por ejemplo, si es o no una palabra clave, si es externa y/o estática, etc. La forma más compacta de codificar este tipo de información es un conjunto de indicadores de un bit en un solo `carácter` o `int`.

La forma habitual de hacerlo es definir un conjunto de "máscaras" correspondientes a las posiciones de bits relevantes, como en

```
#define PALABRA CLAVE 01
```

```
#define EXTRENAL 02
#define ESTÁTICA 04
```

o

```
enumeración { PALABRA CLAVE = 01, EXTERNO = 02, ESTÁTICO = 04 };
```

Los números deben ser potencias de dos. Entonces, el acceso a los bits se convierte en una cuestión de "jugar con los bits" con los operadores de desplazamiento, enmascaramiento y complemento que se describieron en el Capítulo 2.

Ciertos modismos aparecen con frecuencia:

```
banderas |= EXTERNO | ESTÁTICO;
```

enciende los bits `EXTERNAL` y `STATIC` en `las banderas`, mientras que

```
banderas &= ~(EXTERNO | ESTÁTICO);
```

los apaga, y

```
if ((flags & (EXTERNO | ESTÁTICO)) == 0) ...
```

es verdadero si ambos bits están desactivados.

Aunque estos modismos se dominan fácilmente, como alternativa, C ofrece la capacidad de definir y acceder a campos dentro de una palabra directamente en lugar de mediante operadores lógicos bit a bit. Un *campo de bits*, o *campo* para abreviar, es un conjunto de bits adyacentes dentro de una sola unidad de almacenamiento definida por la implementación que llamaremos una "palabra". Por ejemplo, la tabla de símbolos `#defines anterior podría sustituirse por la definición de tres campos:`

```
struct {
    int sin signo is_keyword :
    1; int is_extern sin signo :
    1; int sin signo is_static :
    1;
} banderas;
```

Esto define una tabla de variables llamada `flags` que contiene tres campos de 1 bit. El número que sigue a los dos puntos representa el ancho del campo en bits. Los campos se declaran `unsigned int` para asegurarse de que son cantidades sin signo.

Se hace referencia a los campos individuales de la misma manera que a otros miembros de la estructura: `flags.is_keyword`, `flags.is_extern`, etc. Los campos se comportan como enteros pequeños y pueden participar en expresiones aritméticas al igual que otros enteros. Por lo tanto, los ejemplos anteriores pueden escribirse de manera más natural como

```
flags.is_extern = flags.is_static = 1;
```

para encender los bits;

```
flags.is_extern = flags.is_static = 0;
```

para apagarlos; y

```
if(flags.is_extern== 0 &&&flags.is_static=0)
    ...
```

para probarlos.

Casi todo lo relacionado con los campos depende de la implementación. El hecho de que un campo pueda superponerse a un límite de palabra se define en la implementación. No es necesario que los campos sean nombres; Los campos sin nombre (solo dos puntos y ancho) se utilizan para el relleno. El ancho especial 0 se puede usar para forzar la alineación en el siguiente límite de palabra.

Los campos se asignan de izquierda a derecha en algunas máquinas y de derecha a izquierda en otras. Esto significa que, aunque los campos son útiles para mantener estructuras de datos definidas internamente, la cuestión de qué extremo es primero debe considerarse cuidadosamente al seleccionar datos definidos externamente; Los programas que dependen de tales cosas no son portátiles. Los campos solo se pueden declarar como `enteros;` Para la portabilidad, especifique `Signed` o `Unsigned` explícitamente. No son matrices y no tienen direcciones, por lo que no se puede aplicar el operador & en ellas.

Capítulo 7 - Entrada y salida

La entrada y la salida no forman parte del propio lenguaje C, por lo que no las hemos destacado en nuestra presentación hasta ahora. Sin embargo, los programas interactúan con su entorno de formas mucho más complicadas que las que hemos mostrado antes. En este capítulo describiremos la biblioteca estándar, un conjunto de funciones que proporcionan entrada y salida, manejo de cadenas, administración de almacenamiento, rutinas matemáticas y una variedad de otros servicios para programas C. Nos concentraremos en la entrada y la salida

El estándar ANSI define estas funciones de biblioteca con precisión, de modo que puedan existir en forma compatible en cualquier sistema en el que exista C. Los programas que limitan sus interacciones del sistema a las instalaciones proporcionadas por la biblioteca estándar se pueden mover de un sistema a otro sin cambios.

Las propiedades de las funciones de la biblioteca se especifican en más de una docena de encabezados; ya hemos visto varios de estos, incluyendo `<stdio.h>`, `<string.h>` y `<ctype.h>`. No presentaremos aquí toda la biblioteca, ya que estamos más interesados en escribir programas en C que la utilicen. La biblioteca se describe en detalle en el Apéndice B.

7.1 Entrada y salida estándar

Como dijimos en el Capítulo 1, la biblioteca implementa un modelo simple de entrada y salida de texto. Un flujo de texto consta de una secuencia de líneas; Cada línea termina con un carácter de nueva línea. Si el sistema no funciona de esa manera, la biblioteca hace lo que sea necesario para que parezca que lo hace. Por ejemplo, la biblioteca puede convertir el retorno de carro y el salto de línea en nueva línea en la entrada y viceversa en la salida.

El mecanismo de entrada más simple es leer un carácter a la vez desde la *entrada estándar*, normalmente el teclado, con `getchar`:

```
int getchar(vacío)
```

`getchar` devuelve el siguiente carácter de entrada cada vez que se llama, o `EOF` cuando encuentra el final del archivo. La constante simbólica `EOF` se define en `<stdio.h>`. El valor suele ser -1, las pruebas de bus deben escribirse en términos de `EOF` para que sean independientes del valor específico.

En muchos entornos, un archivo puede ser sustituido por el teclado utilizando la convención < para la redirección de entrada: si un programa `prog` usa `getchar`, entonces la línea de comandos

```
prog < infile
```

hace que `prog` lea los caracteres de `infile` en su lugar. El cambio de la entrada se realiza de tal manera que `prog` en sí mismo es ajeno al cambio; en particular, la cadena "<infile" no está incluida en los argumentos de la línea de comandos en `argv`. La conmutación de entrada también es invisible si la entrada proviene de otro programa a través de un mecanismo de tubería: en algunos sistemas, la línea de comandos

```
otherprog | Prog
```

Ejecuta los dos programas `otherprog` y `prog`, y canaliza la salida estándar de `otherprog` a la entrada estándar para `prog`.

La función

```
int putchar(int)
```

se utiliza para la salida: `putchar(c)` coloca el carácter `c` en la `salida estándar`, que es por defecto la pantalla. `putchar` devuelve el carácter escrito, o `EOF` es que se produce un error. De nuevo, la salida generalmente se puede dirigir a un archivo con >filename: si `prog` usa `putchar`,

```
prog >outfile
```

escribirá la salida estándar en outfile en su lugar. Si se admiten tuberías,

```
Programa | otroprog
```

Coloca la salida estándar de `prog` en la entrada estándar de `anotherprog`.

La salida producida por `printf` también encuentra su camino a la salida estándar. Las llamadas a `putchar` y
`printf` puede estar intercalado: la salida se produce en el orden en que se realizan las llamadas.

Cada archivo de código fuente que haga referencia a una función de biblioteca de entrada/salida debe contener la línea

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

antes de la primera referencia. Cuando el nombre está entre corchetes < y > se realiza una búsqueda del encabezado en un conjunto estándar de lugares (por ejemplo, en sistemas UNIX, normalmente en el directorio
`/usr/include`).

Muchos programas leen solo un flujo de entrada y escriben solo un flujo de salida; para tales programas, la entrada y salida con `getchar`, `putchar` y `printf` pueden ser completamente adecuadas, y ciertamente son suficientes para comenzar. Esto es particularmente cierto si se utiliza la redirección para conectar la salida de un programa a la entrada del siguiente. Por ejemplo, considere el programa `lower`, que convierte su entrada a minúsculas:

```
#include <stdio.h>
#include <ctype.h>

main() /* lower: convierte la entrada a minúsculas*/
{
    int c

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    devuelve 0;
}
```

La función `tolower` se define en `<ctype.h>`; convierte una letra mayúscula en minúscula y devuelve los demás caracteres sin tocar. Como mencionamos anteriormente, las "funciones" como `getchar` y `putchar` en `<stdio.h>` y `tolower` en `<ctype.h>` suelen ser macros, evitando así la sobrecarga de una llamada a función por carácter. Mostraremos cómo se hace esto en la Sección 8.5. Independientemente de cómo `se implementen las funciones <ctype.h>` en un equipo determinado, los programas que las utilizan están protegidos del conocimiento del conjunto de caracteres.

Ejercicio 7-1. Escriba un programa que convierta las mayúsculas en minúsculas o las minúsculas en mayúsculas, dependiendo del nombre con el que se invoque, como se

encuentra en `argv[0]`.

7.2 Salida formateada - printf

La función de salida `printf` traduce los valores internos a caracteres. Hemos usado `printf` informalmente en capítulos anteriores. La descripción aquí cubre la mayoría de los usos típicos, pero no está completa; para leer la historia completa, véase el Apéndice B.

```
int printf(char *format, arg1, arg2, ...);
```

`printf` convierte, formatea e imprime sus argumentos en la salida estándar bajo el control de la función
`formato`. Devuelve el número de caracteres impresos.

La cadena de formato contiene dos tipos de objetos: caracteres ordinarios, que se copian en el flujo de salida, y especificaciones de conversión, cada una de las cuales provoca la conversión e impresión del siguiente argumento sucesivo en `printf`. Cada especificación de conversión comienza con un % y termina con un carácter de conversión. Entre el % y el carácter de conversión puede haber, en orden:

- Un signo menos, que especifica el ajuste a la izquierda del argumento convertido.
- Número que especifica el ancho de campo mínimo. El argumento convertido se imprimirá en un campo de al menos este ancho. Si es necesario, se rellenará a la izquierda (o a la derecha, si se requiere un ajuste a la izquierda) para compensar el ancho del campo.
- Un punto, que separa el ancho del campo de la precisión.
- Número, la precisión, que especifica el número máximo de caracteres que se imprimirán a partir de una cadena, o el número de dígitos después del separador decimal de un valor de coma flotante, o el número mínimo de dígitos de un entero.
- Una `h` si el número entero se va a imprimir como corto, o `l` (letra ell) si se va a imprimir como `largo`.

Los caracteres de conversión se muestran en la Tabla 7.1. Si el carácter después de % no es una especificación de conversión, el comportamiento es indefinido.

Tabla 7.1 *Conversiones básicas de impresión*

Carácter	Tipo de argumento; Impreso como
`d,i`	`int`; Número decimal
`o`	`int`; Número octal sin signo (sin cero a la izquierda)
`x,x`	`int`; número hexadecimal sin signo (sin 0x o `0X a la izquierda`), usando `abcdef` o `ABCDEF` para 10, ...,15.
`u`	`int`; Número decimal sin signo
`c`	`int`; Un solo carácter
`s`	`char *`; Imprime caracteres de la cadena hasta un `'\0'` o el número de caracteres dado por la precisión.
`f`	`doble`; `[-]`*m.dddddd*, donde el número de *d* viene dado por la precisión (por defecto 6).
`e,E`	`doble`; `[-]`*m.dddddde+/-xx* o `[-]`*m.ddddddE+/-xx*, donde el número de *d* viene dado por la precisión (por defecto 6).
`g,g`	`doble`; use `%e` o `%E` si el exponente es menor que -4 o mayor o igual que la precisión; de lo contrario, use `%f`. Los ceros finales y un punto decimal final no se imprimen.
`p`	`nulo *`; puntero (representación dependiente de la implementación).

%	no se convierte ningún argumento; Imprime un %

Un ancho o precisión se puede especificar como *, en cuyo caso el valor se calcula convirtiendo el siguiente argumento (que debe ser un int). Por ejemplo, para imprimir como máximo el número máximo de caracteres de una cadena s,

```
printf("%.*s", max, s);
```

La mayoría de las conversiones de formato se han ilustrado en capítulos anteriores. Una excepción es la precisión en lo que se refiere a las cadenas. La siguiente tabla muestra el efecto de una variedad de especificaciones en la impresión "hello, world" (12 caracteres). Hemos puesto dos puntos alrededor de cada campo para que pueda ver su extensión.

```
:%s:          :Hola mundo:
:%10s:        :Hola mundo:
:%.10s:       :Hola, wor:
:%-10s:       :Hola mundo:
:%.15s:       :Hola mundo:
:%-15s:       :Hola mundo     :
:%15.10s:     :     Hola, WOR:
:%-15.10s:    :hola, wor      :
```

Una advertencia: printf usa su primer argumento para decidir cuántos argumentos siguen y cuál es su tipo. Se confundirá, y obtendrá respuestas incorrectas, si no hay suficientes argumentos o si son del tipo incorrecto. También debe tener en cuenta la diferencia entre estas dos llamadas:

```
printf(s);          /* FALLA si s contiene % */
printf("%s", s);/* SEGURO */
```

La función sprintf realiza las mismas conversiones que printf , pero almacena la salida en una cadena:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

sprintf formatea los argumentos en arg1, arg2, etc., de acuerdo con el formato como antes, pero coloca el resultado en cadena en lugar de la salida estándar; La cadena debe ser lo suficientemente grande como para recibir el resultado.

Ejercicio 7-2. Escriba un programa que imprima entradas arbitrarias de una manera sensata. Como mínimo, debe imprimir caracteres no gráficos en octal o hexadecimal según la costumbre local, y romper las líneas de texto largas.

7.3 Listas de argumentos de longitud variable

Esta sección contiene una implementación de una versión mínima de printf, para mostrar cómo escribir una función que procesa una lista de argumentos de longitud variable de una manera portátil. Dado que estamos interesados principalmente en el procesamiento de argumentos, minprintf procesará la cadena de formato y los argumentos, pero llamará al printf real para realizar las conversiones de formato.

La declaración correcta para printf es

```
int printf(char *fmt, ...)
```

donde la declaración ... significa que el número y los tipos de estos argumentos pueden variar. La declaración ... solo puede aparecer al final de una lista de argumentos. Nuestro minprintf se declara como

```
void minprintf(char *fmt, ...)
```

ya que no devolveremos el recuento de caracteres que `hace printf` .

La parte complicada es cómo `minprintf` camina a lo largo de la lista de argumentos cuando la lista ni siquiera tiene un nombre. El encabezado estándar `<stdarg.h>` contiene un conjunto de definiciones de macro que definen cómo recorrer una lista de argumentos. La implementación de este encabezado variará de una máquina a otra, pero la interfaz que presenta es uniforme.

El tipo `va_list` se usa para declarar una variable que se referirá a cada argumento a su vez; en `minprintf`, esta variable se llama `ap`, por "argument pointer". La macro `va_start` inicializa `ap` para que apunte al primer argumento sin nombre. Se debe llamar una vez antes de `usar ap`. Debe haber al menos un argumento con nombre; El argumento con nombre final es utilizado por `va_start` para empezar.

Cada llamada de `va_arg` devuelve un argumento y pasa `ap` al siguiente; `va_arg` usa un nombre de tipo para determinar qué tipo devolver y qué tan grande es el paso que se debe dar. Por último, `va_end` hace la limpieza que sea necesaria. Se debe llamar antes de que se devuelva el programa.

Estas propiedades forman la base de nuestra `impresión simplificada`:

```
#include <stdarg.h>

/* minprintf: printf mínimo con lista de argumentos
variables */ void minprintf(char *fmt, ...)
{
    va_list AP; /* señala a su vez cada arg sin nombre
    */ char *p, *sval;
    int ival;
    Doble DVAL;

    va_start (AP, FMT); /* hacer que ap apunte al 1er arg sin
    nombre */ for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continuar;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
            quebrar;
        Caso 'F':
            dval = va_arg(ap, doble);
            printf("%f", dval);
            quebra
        r; Caso
        'S':
            Fore(swal = v_arg(ap, cuatro*); *swal; swal++)
                putchar(*swal);
            quebra
        r;
        predetermi
        nado:
            putchar(*p);
            quebrar;
        }
    }
    va_end(AP); /* limpiar cuando termine */
}
```

Ejercicio 7-3. Revisa `minprintf` para manejar más de las otras instalaciones de `printf`.

7.4 Entrada formateada - Scanf

La función `scanf` es el análogo de entrada de `printf`, proporcionando muchas de las mismas facilidades de conversión en la dirección opuesta.

```
int scanf(char *formato, ...)
```

`scanf` lee los caracteres de la entrada estándar, los interpreta de acuerdo con la especificación en formato y almacena los resultados a través de los argumentos restantes. El argumento format se describe a continuación; Los otros argumentos, *cada uno de los cuales debe ser un puntero*, indican dónde se debe almacenar la entrada convertida correspondiente. Al igual que con `printf`, esta sección es un resumen de las características más útiles, no una lista exhaustiva.

`scanf` se detiene cuando agota su cadena de formato o cuando alguna entrada no coincide con la especificación del control. Devuelve como valor el número de elementos de entrada asignados y coincidentes correctamente. Esto se puede usar para decidir cuántos elementos se encontraron. Al final del archivo, `se devuelve EOF`; tenga en cuenta que esto es diferente de 0, lo que significa que el siguiente carácter de entrada no coincide con la primera especificación en la cadena de formato. La siguiente llamada a `scanf` reanuda la búsqueda inmediatamente después del último carácter ya convertido.

También hay una función `sscanf` que lee de una cadena en lugar de la entrada estándar:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Escanea la `cadena` de acuerdo con el formato en `formato` y almacena los valores resultantes a través de
`arg1`, `arg2`, etc. Estos argumentos deben ser indicadores.

La cadena de formato suele contener especificaciones de conversión, que se utilizan para controlar la conversión de la entrada. La cadena de formato puede contener:

- Espacios en blanco o pestañas, que no se ignoran.
- Caracteres ordinarios (no %), que se espera que coincidan con el siguiente carácter de espacio no blanco de la secuencia de entrada.
- Especificaciones de conversión, que constan del carácter `%`, un carácter de supresión de asignación opcional `*`, un número opcional que especifica un ancho de campo máximo, un `h`, `l` o `L` opcional que indica el ancho del destino y un carácter de conversión.

Una especificación de conversión dirige la conversión del siguiente campo de entrada. Normalmente, el resultado se coloca en la variable señalada por el argumento correspondiente. Sin embargo, si la supresión de la asignación se indica mediante el carácter *, se omite el campo de entrada; No se realiza ninguna cesión. Un campo de entrada se define como una cadena de caracteres que no son espacios en blanco; Se extiende hasta el siguiente carácter de espacio en blanco o hasta que se agote el ancho de campo. Esto implica que `scanf` leerá a través de los límites para encontrar su entrada, ya que las nuevas líneas son espacios en blanco. (Los caracteres de espacio en blanco son espacios en blanco, tabulación, nueva línea, retorno de carro, tabulación vertical y avance de formulario).

El carácter de conversión indica la interpretación del campo de entrada. El argumento correspondiente debe ser un puntero, tal y como requiere la semántica de llamada por valor de C. Los caracteres de conversión se muestran en la tabla 7.2.

Tabla 7.2: *Conversiones básicas de Scanf*

Carácter	Datos de entrada; Tipo de argumento

d	entero decimal; Int*
Yo	entero; int *. El número entero puede estar en octal (0 inicial) o hexadecimal (intercalado 0x o 0X).
o	entero octal (con o sin cero a la izquierda); Int*
u	entero decimal sin signo; unsigned int *
x	entero hexadecimal (con o sin 0x o 0X inicial); Int*
c	Caracteres; char *. Los siguientes caracteres de entrada (por defecto 1) se colocan en el lugar indicado. Se suprime el espacio en blanco de salto normal; para leer el siguiente carácter de espacio que no sea en blanco, use %1s
s	cadena de caracteres (sin comillas); char *, apuntando a una matriz de caracteres lo suficientemente larga para la cadena y una terminación '\0' que se agregará.
e,f,g	número de coma flotante con signo opcional, punto decimal opcional y exponente opcional; flotar*
%	% literal; No se realiza ninguna cesión.

Los caracteres de conversión d, i, o, u y x pueden ir precedidos de h para indicar que aparece un puntero a short en lugar de int en la lista de argumentos, o a l (letra ell) para indicar que aparece un puntero a long en la lista de argumentos.

Como primer ejemplo, la calculadora rudimentaria del Capítulo 4 se puede escribir con scanf para hacer la conversión de entrada:

```
#include <stdio.h>

main() /* calculadora rudimentaria */
{
    suma doble, V;

    suma = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", suma += v);
    devuelve 0;
}
```

Supongamos que queremos leer líneas de entrada que contienen fechas de la forma

```
25 dic 1988
```

La instrucción scanf es

```
int día, año;
char monthname[20];

scanf("%d %s %d", &día, nombre del mes y año);
```

No & se usa con monthname, ya que un nombre de matriz es un puntero.

Los caracteres literales pueden aparecer en la cadena de formato scanf; deben coincidir con los mismos caracteres de la entrada. Por lo tanto, podríamos leer las fechas de la forma mm/dd/aa con la declaración scanf:

```
int día, mes, año;

scanf("%d/%d/%d", &mes, &día, y año);
```

`scanf` ignora los espacios en blanco y las tabulaciones en su cadena de formato. Además, omite los espacios en blanco (espacios en blanco, tabulaciones, nuevas líneas, etc.) mientras busca valores de entrada. Para leer una entrada cuyo formato no es fijo, a menudo es mejor leer una línea a la vez y luego separarla con `scanf`. Por ejemplo, supongamos que queremos leer líneas que pueden contener una fecha en cualquiera de las formas anteriores. Entonces podríamos escribir

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(línea, "%d %s %d", &día, nombredelmes, &año) == 3)
        printf("válido: %s\n", línea); /* Formulario del 25 de
        diciembre de 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line); /* mm/dd/aa forma */
    más
        printf("inválido: %s\n", línea); /* forma inválida */
}
```

Las llamadas a `scanf` se pueden mezclar con llamadas a otras funciones de entrada. La siguiente llamada a cualquier función de entrada comenzará leyendo el primer carácter no leído por `scanf`.

Una advertencia final: los argumentos de `scanf` y `sscanf` *deben* ser punteros. Con mucho, el error más común es escribir

```
scanf("%d", n);
```

En lugar de

```
scanf("%d", &n);
```

Por lo general, este error no se detecta en tiempo de compilación.

Ejercicio 7-4. Escriba una versión privada de `scanf` análoga a `minprintf` de la sección anterior.

Ejercicio 5-5. Reescriba la calculadora de sufijo del Capítulo 4 para usar `scanf` y/o `sscanf` para hacer la conversión de entrada y número.

7.5 Acceso a archivos

En los ejemplos hasta ahora se ha leído la entrada estándar y se ha escrito la salida estándar, que se definen automáticamente para un programa por el sistema operativo local.

El siguiente paso es escribir un programa que acceda a un archivo que *aún no* esté conectado al programa. Un programa que ilustra la necesidad de tales operaciones es `cat`, que concatena un conjunto de archivos con nombre en la salida estándar. `CAT` se utiliza para imprimir archivos en la pantalla y como un colector de entrada de propósito general para programas que no tienen la capacidad de acceder a los archivos por nombre. Por ejemplo, el comando

```
gato x.c y.c
```

Imprime el contenido de los archivos `x.c` e `y.c` (y nada más) en la salida estándar.

La cuestión es cómo organizar la lectura de los archivos con nombre, es decir, cómo conectar los nombres externos que se le ocurren a un usuario con las instrucciones que leen los datos.

Las reglas son simples. Antes de que pueda ser leído o escrito, un archivo tiene que ser *abierto* por la función de biblioteca `fopen`. `fopen` toma un nombre externo como `x.c` o `y.c`, hace algo de limpieza y

negociación con el sistema operativo (cuyos detalles no tienen por qué preocuparnos), y devuelve un puntero para ser utilizado en lecturas o escrituras posteriores del archivo.

Este puntero, denominado *puntero de archivo*, apunta a una estructura que contiene información sobre el archivo, como la ubicación de un búfer, la posición actual de los caracteres en el búfer, si el archivo se está leyendo o escribiendo, y si se han producido errores o el final del archivo. Los usuarios no necesitan conocer los detalles, ya que las definiciones obtenidas de `<stdio.h>` incluyen una declaración de estructura denominada `FILE`. La única declaración necesaria para un puntero de archivo se ejemplifica mediante

```
ARCHIVO *fp;
Archivo *Phonen (cuatro *nombre, cuatro * modo);
```

Esto dice que `fp` es un puntero a un `FILE`, y `fopen` devuelve un puntero a un `FILE`. Tenga en cuenta que `FILE` es un nombre de tipo, como `int`, no una etiqueta de estructura; se define con una definición de `tipo`. (Detalles de cómo `fopen` se puede implementar en el sistema UNIX se dan en la Sección 8.5.)

La llamada a `fopen` en un programa es

```
FP = phopen(nombre, modo);
```

El primer argumento de `fopen` es una cadena de caracteres que contiene el nombre del archivo. El segundo argumento es el *modo*, también una cadena de caracteres, que indica cómo se pretende utilizar el archivo. Los modos permitidos incluyen lectura (`"r"`), escritura (`"w"`) y anexar (`"a"`). Algunos sistemas distinguen entre archivos de texto y binarios; Para este último, `se debe agregar una "B"` a la cadena de modo.

Si se abre un archivo que no existe para escribirlo o anexarlo, se crea si es posible. Al abrir un archivo existente para escribir, se descarta el contenido antiguo, mientras que al abrirlo para anexarlo se conservan. Intentar leer un archivo que no existe es un error, y también puede haber otras causas de error, como intentar leer un archivo cuando no tiene permiso. Si hay algún error, `fopen` devolverá `NULL`. (El error se puede identificar con mayor precisión; consulte la discusión de las funciones de manejo de errores al final de Sección 1 del Apéndice B.)

Lo siguiente que se necesita es una forma de leer o escribir el archivo una vez que esté abierto. `getc` devuelve el siguiente carácter de un archivo; necesita el puntero del archivo para decirle qué archivo.

```
int getc(ARCHIVO *fp)
```

`getc` devuelve el siguiente carácter de la secuencia a la que se refiere `fp`; devuelve `EOF` para el final del archivo o error.

`putc` es una función de salida:

```
int putc(int c, FICHERO *fp)
```

`putc` escribe el carácter `c` en el fichero `fp` y devuelve el carácter escrito, o EOF si se produce un error. Al igual que `getchar` y `putchar`, `getc` y `putc` pueden ser macros en lugar de funciones.

Cuando se inicia un programa C, el entorno del sistema operativo es responsable de abrir tres archivos y proporcionar punteros para ellos. Estos archivos son la entrada estándar, la salida estándar y el error estándar; Los punteros de archivo correspondientes se denominan `stdin`, `stdout` y `stderr`, y se declaran en `<stdio.h>`. Normalmente, `stdin` se conecta al teclado y

`stdout` y `stderr` están conectados a la pantalla, pero `stdin` y `stdout` pueden ser redirigidos a archivos o canalizaciones como se describe en la Sección 7.1.

`getchar` y `putchar` se pueden definir en términos de `getc`, `putc`, `stdin` y `stdout` de la siguiente manera:

```
#define getchar()getc(stdin)
                     #define
putchar(c)putc((c), stdout)
```

Para la entrada o salida formateada de archivos, se pueden utilizar las funciones `fscanf` y `fprintf`. Estos son idénticos a `scanf` y `printf`, excepto que el primer argumento es un puntero de archivo que especifica el archivo que se va a leer o escribir; la cadena de formato es el segundo argumento.

```
int fscanf(ARCHIVO *fp, char *formato,
...) int fprintf(ARCHIVO *fp, char
*formato, ...)
```

Con estos preliminares fuera del camino, ahora estamos en condiciones de escribir el programa `cat` para concatenar archivos. El diseño es uno que se ha encontrado conveniente para muchos programas. Si hay argumentos de línea de comandos, se interpretan como nombres de archivo y se procesan en orden. Si no hay argumentos, se procesa la entrada estándar.

```
#include <stdio.h>

/* cat: concatenar ficheros, versión 1 */
main(int argc, char *argv[])
{
    ARCHIVO *fp;
    void filecopy(ARCHIVO *, ARCHIVO *)

    if (argc == 1) /* no args; copiar entrada estándar */
        filecopy(stdin, stdout);
    más
       while(--argc > 0)
           if ((fp = fopen(*++argv, "r")) == NULL) {
               printf("cat: can't open %s\n, *argv);
               retorno 1;
           } else {
              filecopy(fp, stdout);
              fclose(fp);
           }
      devuelve 0;
}

 /* filecopy: copiar archivo ifp a archivo ofp
 */ void filecopy(FILE *ifp, FILE *ofp)
 {
     int c;

     while ((c = getc(ifp)) != EOF)
         putc(c, ofp);
 }
```

Los punteros de archivo `stdin` y `stdout` son objetos de tipo `FILE *`. Son constantes, sin embargo,
no variables, por lo que no es posible asignarles.

La función

```
int fclose(ARCHIVO *fp)
```

es el inverso de `fopen`, rompe la conexión entre el puntero del archivo y el nombre externo

establecido por `fopen`, liberando el puntero del archivo para otro archivo. Dado que la mayoría de los sistemas operativos tienen algún límite en el número de archivos que un programa puede tener abiertos

Al mismo tiempo, es una buena idea liberar los punteros de archivo cuando ya no sean necesarios, como hicimos en `Cat`. También hay otra razón para `fclose` en un archivo de salida: vacía el búfer en el que `putc` está recopilando la salida. `fclose` se llama automáticamente para cada archivo abierto cuando un programa termina normalmente. (Puede cerrar `stdin` y `stdout` si no son necesarios. También pueden ser reasignados por la función de biblioteca `freopen`.)

7.6 Manejo de errores: Stderr y Exit

El tratamiento de los errores en `el gato` no es el ideal. El problema es que si no se puede acceder a uno de los archivos por algún motivo, el diagnóstico se imprime al final de la salida concatenada. Eso podría ser aceptable si la salida va a una pantalla, pero no si va a un archivo o a otro programa a través de una canalización.

Para manejar mejor esta situación, se asigna un segundo flujo de salida, llamado `stderr`, a un programa de la misma manera que `stdin` y `stdout` . La salida escrita en `stderr` normalmente aparece en la pantalla incluso si la salida estándar está redirigida.

Revisemos `cat` para escribir sus mensajes de error en el error estándar.

```
#include <stdio.h>

/* cat: concatenar ficheros, versión 2 */
main(int argc, char *argv[])
{
    ARCHIVO *fp;
    void filecopy(ARCHIVO *, ARCHIVO *);
    char *prog = argv[0]; /* nombre del programa para errores */

    if (argc == 1 ) /* no args; copiar entrada estándar */
        filecopy(stdin, stdout);
    más
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: no se puede abrir %s\n",
                        prog, *argv);
                salida(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: error al escribir stdout\n",
        prog); salida(2);
    }
    salida(0);
}
```

El programa señala los errores de dos maneras. En primer lugar, la salida de diagnóstico producida por `fprintf` va a `stderr`, por lo que encuentra su camino a la pantalla en lugar de desaparecer en una tubería o en un archivo de salida. Incluimos el nombre del programa, de `argv[0]`, en el mensaje, por lo que si este programa se usa con otros, se identifica la fuente de un error.

En segundo lugar, el programa utiliza la función de biblioteca estándar `exit`, que finaliza la ejecución del programa cuando se llama. El argumento de `exit` está disponible para cualquier proceso llamado este, por lo que el éxito o el fracaso del programa puede ser probado por otro programa que lo use como un subproceso. Convencionalmente, un valor devuelto de 0 indica que todo está bien; distinto de cero

Los valores suelen indicar situaciones anormales. `exit` llama a `fclose` para cada archivo de salida abierto, para vaciar cualquier salida almacenada en búfer.

Dentro de `main`, `return` *expr* es equivalente a `exit`(*expr*). `exit` tiene la ventaja de que se puede llamar desde otras funciones, y que las llamadas a él se pueden encontrar con un programa de búsqueda de patrones como los del Capítulo 5.

La función `ferror` devuelve un valor distinto de cero si se ha producido un error en la secuencia `fp`.

```
int ferror(ARCHIVO *fp)
```

Aunque los errores de salida son raros, ocurren (por ejemplo, si un disco se llena), por lo que un programa de producción también debe verificarlo.

La función `feof(FILE *)` es análoga a `ferror`; devuelve un valor distinto de cero si se ha producido el final del fichero en el fichero especificado.

```
int feof(ARCHIVO *fp)
```

Por lo general, no nos hemos preocupado por el estado de salida en nuestros pequeños programas ilustrativos, pero cualquier programa serio debe tener cuidado de devolver valores de estado sensatos y útiles.

7.7 Entrada y salida de línea

La biblioteca estándar proporciona una rutina de entrada y salida `fgets` que es similar a la `getline`
función que hemos utilizado en capítulos anteriores:

```
char *fgets(char *line, int maxline, FILE *fp)
```

`fgets` lee la siguiente línea de entrada (incluyendo la nueva línea) del fichero `fp` en la línea de matriz de caracteres; a lo sumo `se leerán los caracteres maxline-1`. La línea resultante termina con `'\0'`. Normalmente, `fgets` devuelve `una línea`; al final del archivo o por error, devuelve `NULL`. (Nuestro `getline` devuelve la longitud de la línea, que es un valor más útil; cero significa el final del archivo).

Para la salida, la función `fputs` escribe una cadena (que no necesita contener una nueva línea) en un archivo:

```
int fputs(char *line, FILE *fp)
```

Devuelve `EOF` si se produce un error y no negativo en caso contrario.

Las funciones `de biblioteca gets` y `puts` son similares a `fgets` y `fputs`, pero funcionan en `stdin`
y `stdout`. De manera confusa, `gets` elimina la terminación `'\n'` y `puts` la agrega.

Para mostrar que no hay nada especial en funciones como `fgets` y `fputs`, aquí están, copiadas de la biblioteca estándar de nuestro sistema:

```
/* fgets: obtener a la mayoría de n
caracteres de iop */ char *fgets(char *s,
int n, FILE *iop)
{
    registro int c;
    registrar char
```

```
*cs;

cs = s;
while (--n > 0 && (c = getc(iop)) != EOF)
```

```
        if ((*cs++ = c) == '\n')
            interrupción;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: poner la cadena s en el
archivo iop */ int fputs(char *s, FILE
*iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

Sin ninguna razón obvia, el estándar especifica diferentes valores de retorno para `ferror` y `fputs`. Es fácil implementar nuestro `getline` desde `fgets`:

```
/* getline: lee una línea, devuelve la
longitud */ int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        devuelve 0;
    más
        return strlen(línea);
}
```

Ejercicio 7-6. Escriba un programa para comparar dos archivos, imprimiendo la primera línea donde difieren.

Ejercicio 7-7. Modifique el programa de búsqueda de patrones del Capítulo 5 para que tome su entrada de un conjunto de archivos con nombre o, si no se nombran archivos como argumentos, de la entrada estándar. ¿Se debe imprimir el nombre del archivo cuando se encuentra una línea coincidente?

Ejercicio 7-8. Escriba un programa para imprimir un conjunto de archivos, comenzando cada nuevo en una nueva página, con un título y un recuento de páginas en ejecución para cada archivo.

7.8 Funciones varias

La biblioteca estándar proporciona una amplia variedad de funciones. Esta sección es una breve sinopsis de los más útiles. En el Apéndice B se pueden encontrar más detalles y muchas otras funciones.

7.8.1 Operaciones de cadena

Ya hemos mencionado las funciones de cadena `strlen`, `strcpy`, `strcat` y `strcmp`, que se encuentran en `<string.h>`. A continuación, `s` y `t` son `char *`, y `c` y `n` son `ints`.

`strcat(s,t)` Concatenar `t` hasta el final de `s`

`strncat(s,t,n)` Concatenar `n` Caracteres de `t` hasta el final de `s`

`strcmp(s,t)` devuelve negativo, cero o positivo para `s < t`, `s == t`, `s > t` `strncmp(s,t,n)` Igual que `STRCMP` pero solo en la primera `n` Caracteres

`strcpy(s,t)` copiar `t` Para `s`

`strncpy(s,t,n)` Copiar a lo sumo `n` Caracteres de `t` Para `s` `strlen(s)` Longitud de retorno de `s`

`strchr(s,c)` devolver el puntero a first `c` en `s`o `NULO` si no está presente

`strrchr(s,c)` devolver el puntero al último `c` en `s`o `NULO` si no está presente

7.8.2 Pruebas y conversión de clases de caracteres

Varias funciones de `<ctype.h>` realizar pruebas de caracteres y conversiones. A continuación, `c` es un `int` que se puede representar como un `char sin signo` o `EOF`. La función devuelve `int`. `isalpha(c)` distinto de cero si `c` es alfabético, 0 si no

`isupper(c)` distinto de cero si `c` está en mayúsculas, 0 si no `islower(c)` distinto de cero si `c` está en minúsculas, 0 si no

`isdigit(c)` distinto de cero si `c` es dígito, 0 si no

`isalnum(c)` distinto de cero si `isalpha(c)` o `isdigit(c)`, 0 si no

`isspace(c)` distinto de cero si `c` está en blanco, tabulación, nueva línea, retorno, saltoform, tabulador vertical

`toupper(c)` return `c` convertido a mayúsculas

`tolower(c)` devuelve `c` convertido a minúsculas

7.8.3 Ungetc

La biblioteca estándar proporciona una versión bastante restringida de la función `ungetch` que escribimos en el Capítulo 4; se llama `ungetc`.

```
int ungetc(int c, FICHERO *fp)
```

empuja el carácter `c` de nuevo al archivo `fp` y devuelve `c` o `EOF` para un error. Solo se garantiza un carácter de retroceso por archivo. `UNGETC` se puede utilizar con cualquiera de las funciones de entrada como `scanf`, `getc` o `getchar`.

7.8.4 Ejecución de comandos

La función `system(char *s)` ejecuta el comando contenido en la cadena de caracteres `s`, luego reanuda la ejecución del programa actual. El contenido de `s` depende en gran medida del sistema operativo local. Como ejemplo trivial, en los sistemas UNIX, la declaración

```
sistema("fecha");
```

Hace que se ejecute la `fecha` del programa; imprime la fecha y la hora del día en la salida estándar. `system` devuelve un estado entero dependiente del sistema del comando ejecutado. En el sistema UNIX, el status return es el valor devuelto por `exit`.

7.8.5 Gestión de almacenamiento

Las funciones `malloc` y `calloc` obtienen bloques de memoria de forma dinámica.

```
void *malloc(size_t n)
```

devuelve un puntero a `n` bytes de almacenamiento sin inicializar, o `NULL` si no se puede satisfacer la solicitud.

```
void *calloc(size_t n, tamaño size_t)
```

devuelve un puntero a suficiente espacio libre para una matriz de `n` objetos del tamaño especificado, o `NULL` si no se puede satisfacer la solicitud. El almacenamiento se inicializa en cero.

El puntero devuelto por `malloc` o `calloc` tiene la alineación adecuada para el objeto en cuestión, pero debe convertirse en el tipo adecuado, como en

```
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

`free(p)` libera el espacio señalado por `p`, donde `p` se obtuvo originalmente mediante una llamada a `malloc` o `calloc`. No hay restricciones en el orden en que se libera el espacio, pero es un error espantoso liberar algo que no se obtiene llamando a `malloc` o `calloc`.

También es un error usar algo después de que se haya liberado. Un fragmento de código típico pero incorrecto es este bucle que libera elementos de una lista:

```
for (p = cabeza; p != NULL; p = p->siguiente) /*
    INCORRECTO */ libre(p);
```

La forma correcta es guardar lo que sea necesario antes de liberar:

```
for (p = cabeza; p != NULL; p = q)
    { q = p->siguiente;
    libre(p);
}
```

La sección 8.7 muestra la implementación de un asignador de almacenamiento como `malloc`, en el que los bloques asignados se pueden liberar en cualquier orden.

7.8.6 Funciones matemáticas

Hay más de veinte funciones matemáticas declaradas en `<math.h>`; estas son algunas de las más utilizadas. Cada uno toma uno o dos argumentos `double` y devuelve un `doble`.

`sin(x)`	seno de *x*, *x* en radianes
`cos(x)`	coseno de *x*, *x* en radianes
`atan2(y,x)`	arcotangente de *y/x*, en radianes
`exp(x)`	Función exponencial *ex*
`registro (x)`	natural (base *e*) logaritmo de *x (x>0)*
`log10(x)`	logaritmo común (base 10) de *x (x>0)*
`pow(x,y)`	*xy*
`sqrt(x)`	raíz cuadrada de *x (x>0)*
`Fabs(x)`	Valor absoluto de *x*

7.8.7 Generación de números aleatorios

La función `rand()` calcula una secuencia de enteros pseudoaleatorios en el rango de cero a `RAND_MAX`, que se define en `<stdlib.h>`. Una forma de producir números aleatorios de coma flotante mayores o iguales que cero pero menores que uno es

```
#define frand() ((doble) rand() / (RAND_MAX+1.0))
```

(Si su biblioteca ya proporciona una función para números aleatorios de punto flotante, es probable que tenga mejores propiedades estadísticas que esta).

La función `srand(unsigned)` establece la semilla para `rand`. La implementación portátil de `rand` y `srand` sugerido por la norma aparece en la Sección 2.7.

Ejercicio 7-9. Funciones como `isupper` se pueden implementar para ahorrar espacio o para ahorrar tiempo. Explora ambas posibilidades.

Capítulo 8 - La interfaz del sistema UNIX

El sistema operativo UNIX proporciona sus servicios a través de un conjunto de llamadas al *sistema*, que son en efecto funciones dentro del sistema operativo que pueden ser llamadas por programas de usuario. En este capítulo se describe cómo utilizar algunas de las llamadas al sistema más importantes de los programas C. Si utiliza UNIX, esto debería ser directamente útil, ya que a veces es necesario emplear llamadas al sistema para obtener la máxima eficiencia, o para acceder a alguna instalación que no está en la biblioteca. Sin embargo, incluso si usa C en un sistema operativo diferente, debería poder obtener información sobre la programación en C al estudiar estos ejemplos; Aunque los detalles varían, se encontrará un código similar en cualquier sistema. Dado que la biblioteca ANSI C se modela en muchos casos en instalaciones UNIX, este código también puede ayudar a su comprensión de la biblioteca.

Este capítulo se divide en tres partes principales: entrada/salida, sistema de archivos y asignación de almacenamiento. Las dos primeras partes asumen una modesta familiaridad con las características externas de los sistemas UNIX.

El Capítulo 7 se refería a una interfaz de entrada/salida que es uniforme en todos los sistemas operativos. En cualquier sistema en particular, las rutinas de la biblioteca estándar deben escribirse en términos de las facilidades proporcionadas por el sistema anfitrión. En las siguientes secciones describiremos las llamadas del sistema UNIX para entrada y salida, y mostraremos cómo se pueden implementar partes de la biblioteca estándar con ellas.

8.1 Descriptores de archivo

En el sistema operativo UNIX, todas las entradas y salidas se realizan mediante la lectura o escritura de archivos, ya que todos los dispositivos periféricos, incluso el teclado y la pantalla, son archivos en el sistema de archivos. Esto significa que una única interfaz homogénea maneja toda la comunicación entre un programa y los dispositivos periféricos.

En el caso más general, antes de leer y escribir un archivo, debe informar al sistema de su intención de hacerlo, un proceso llamado *abrir* el archivo. Si vas a escribir en un archivo también puede ser necesario crearlo o descartar sus contenidos anteriores. El sistema verifica su derecho a hacerlo (¿Existe el archivo? ¿Tiene permiso para acceder a él?) y si todo está bien, devuelve al programa un pequeño número entero no negativo llamado *descriptor de archivo*. Siempre que se deba realizar una entrada o salida en el archivo, se utiliza el descriptor de archivo en lugar del nombre para identificar el archivo. (Un descriptor de archivo es análogo al puntero de archivo utilizado por la biblioteca estándar o al identificador de archivo de MS-DOS). Toda la información sobre un archivo abierto es mantenida por el sistema; El programa de usuario se refiere al archivo solo por el descriptor de archivo.

Dado que la entrada y salida que involucra el teclado y la pantalla es tan común, existen arreglos especiales para que esto sea conveniente. Cuando el intérprete de comandos (el "shell") ejecuta un programa, se abren tres archivos, con los descriptores de archivo 0, 1 y 2, llamados entrada estándar, salida estándar y error estándar. Si un programa lee 0 y escribe 1 y 2, puede realizar entradas y salidas sin preocuparse por abrir archivos.

El usuario de un programa puede redirigir la E/S hacia y desde archivos con < y >:

```
prog <infile >outfile
```

En este caso, el shell cambia las asignaciones predeterminadas para los descriptores de archivo 0 y 1 a los archivos con nombre. Normalmente, el descriptor de archivo 2 permanece adjunto a la pantalla, por lo que los mensajes de error pueden ir allí. Observaciones similares son válidas para la entrada o salida asociada con una tubería. En todos los casos, las asignaciones de archivos son cambiadas por el shell, no por el programa. El programa no sabe de dónde viene su entrada ni a dónde va su salida, siempre y cuando utilice los archivos 0 para la entrada y 1 y 2 para la salida.

8.2 E/S de bajo nivel: lectura y escritura

La entrada y la salida utilizan las llamadas del `sistema de lectura` y `escritura`, a las que se accede desde los programas C a través de dos funciones llamadas `lectura` y `escritura`. Para ambos, el primer argumento es un descriptor de archivo. El segundo argumento es una matriz de caracteres en el programa a la que se van a ir los datos o de donde provenir. El tercer argumento es el número de bytes que se van a transferir.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Cada llamada devuelve un recuento del número de bytes transferidos. En la lectura, el número de bytes devueltos puede ser menor que el número solicitado. Un valor devuelto de cero bytes implica el final del archivo, y `-1` indica un error de algún tipo. Para la escritura, el valor devuelto es el número de bytes escritos; Se ha producido un error si no es igual al número solicitado.

Se puede leer o escribir cualquier número de bytes en una llamada. Los valores más comunes son 1, que significa un carácter a la vez ("sin búfer"), y un número como 1024 o 4096 que corresponde a un tamaño de bloque físico en un dispositivo periférico. Los tamaños más grandes serán más eficientes porque se realizarán menos llamadas al sistema.

Juntando estos hechos, podemos escribir un programa simple para copiar su entrada a su salida, el equivalente al programa de copia de archivos escrito para el Capítulo 1. Este programa copiará cualquier cosa a cualquier cosa, ya que la entrada y la salida se pueden redirigir a cualquier archivo o dispositivo.

```
#include "syscalls.h"

main() /* copiar la entrada a la salida */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    devuelve 0;
}
```

Hemos recopilado prototipos de funciones para las llamadas al sistema en un archivo llamado `syscalls.h` para poder incluirlo en los programas de este capítulo. Sin embargo, este nombre no es estándar.

El parámetro `BUFSIZ` también se define en `syscalls.h`; su valor es un buen tamaño para el sistema local. Si el tamaño del archivo no es un múltiplo de `BUFSIZ`, algunas `lecturas` devolverán un número menor de bytes para ser escritos por `escritura`; la siguiente llamada a `leer` después de eso devolverá cero.

Es instructivo ver cómo se puede usar la `lectura` y la `escritura` para construir rutinas de nivel superior como `getchar`, `putchar`, etc. Por ejemplo, aquí hay una versión de `getchar` que realiza entradas sin búfer, leyendo la entrada estándar un carácter a la vez.

```
#include "syscalls.h"

/* getchar: entrada de un solo carácter sin búfer
*/ int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (carácter sin signo) c : EOF;
}
```

`c` debe ser un `char`, ya que `read` necesita un puntero de carácter. La conversión de `c` a `char sin signo` en la instrucción return elimina cualquier problema de extensión de signo.

La segunda versión de `getchar` ingresa en grandes trozos y reparte los caracteres de uno en uno.

```
#include "syscalls.h"

/* getchar: versión simple almacenada en
búfer */ int getchar(void)
{
    char                 buf
    estático[BUFSIZ];   char
    estático  *bufp  =  buf;
    static int n = 0;

    if (n == 0) { /* el búfer está vacío
        */ n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (carácter sin signo) *bufp++ : EOF;
}
```

Si estas versiones de `getchar` se compilaran con `<stdio.h>` incluido, sería necesario #undef el nombre `getchar` en caso de que se implemente como una macro.

8.3 Abrir, Crear, Cerrar, Desvincular

Aparte de la entrada, salida y error estándar predeterminados, debe abrir archivos explícitamente para leerlos o escribirlos. Hay dos llamadas del sistema para esto, `open` y `creat` [sic].

`open` es bastante parecido al `fopen` discutido en el Capítulo 7, excepto que en lugar de devolver un puntero de archivo, devuelve un descriptor de archivo, que es solo un `int`. `open` devuelve `-1` si se produce algún error.

```
#include <fcntl.h>

int fd;
int open(char *nombre, int flags, int permanentes);

fd = open(nombre, banderas, permanentes);
```

Al igual que con `fopen`, el argumento `name` es una cadena de caracteres que contiene el nombre del archivo. El segundo argumento, `flags`, es un `int` que especifica cómo se va a abrir el archivo; los valores principales son

`O_RDONLY`	abierto solo para lectura
`O_WRONLY`	abierto solo para escribir
`O_RDWR`	Abierto tanto para leer como para escribir

Estas constantes se definen en `<fcntl.h>` en sistemas UNIX de System V y en `<sys/file.h>` en las versiones de Berkeley (BSD).

Para abrir un archivo existente para su lectura,

```
fd = abierto(nombre, O_RDONLY,0);
```

El `argumento perms siempre es cero para los usos de` open `que discutiremos.`

Es un error intentar abrir un archivo que no existe. La creación de llamadas al sistema se proporciona para crear nuevos archivos o para reescribir los antiguos.

```
int creat(char *nombre, int

permanentes); fd = creat(nombre,

permanentes);
```

Devuelve un descriptor de archivo si pudo crear el archivo y `-1` si no. Si el archivo ya existe, `creat` lo truncará a cero, descartando así su contenido anterior; no es un error `crear` un archivo que ya existe.

Si el archivo aún no existe, `creat` lo crea con los permisos especificados por el argumento `perms`. En el sistema de archivos UNIX, hay nueve bits de información de permisos asociados a un archivo que controlan el acceso de lectura, escritura y ejecución para el propietario del archivo, para el grupo del propietario y para todos los demás. Por lo tanto, un número octal de tres dígitos es conveniente para especificar los permisos. Por ejemplo, `0775` especifica el permiso de lectura, escritura y ejecución para el propietario, y el permiso de lectura y ejecución para el grupo y todos los demás.

A modo de ilustración, aquí hay una versión simplificada del programa UNIX `cp`, que copia un archivo a otro. Nuestra versión copia solo un archivo, no permite que el segundo argumento sea un directorio e inventa permisos en lugar de copiarlos.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666/* RW para propietario, grupo, otros

*/ void error(char *, ...);

/* cp: copiar f1 a f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    si (argc != 3)
        error("Uso: cp de a");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: no se puede crear %s, modo
        %03o",
            argv[2], PERMANENTES);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: error de escritura en el fichero
    %s", argv[2]); devuelve 0;
}
```

Este programa crea el archivo de salida con permisos fijos de `0666`. Con la llamada al `sistema stat`, descrita en la Sección 8.6, podemos determinar el modo de un archivo existente y así dar el mismo modo a la copia.

Tenga en cuenta que el error de función se llama con listas de argumentos variables muy parecidas a `printf`. La implementación de error ilustra cómo utilizar otro miembro de la familia `printf`. La función de biblioteca estándar `vprintf` es como `printf`, excepto que la lista de argumentos de la variable se reemplaza por un solo argumento que se ha inicializado llamando a la `macro va_start`. Del mismo modo, `vfprintf` y `vsprintf` coinciden con `fprintf` y `sprintf`.

```
#include <stdio.h>
#include <stdarg.h>

/* error: imprime un mensaje de error y muere
*/ void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    salida(1);
}
```

Hay un límite (a menudo alrededor de 20) en el número de archivos que un programa puede abrir simultáneamente. En consecuencia, cualquier programa que tenga la intención de procesar muchos archivos debe estar preparado para reutilizar los descriptores de archivo. La función `close(int fd)` rompe la conexión entre un descriptor de fichero y un fichero abierto, y libera el descriptor de fichero para usarlo con algún otro fichero; corresponde a `fclose` en la biblioteca estándar, excepto que no hay un búfer para vaciar. La terminación de un programa a través de `la salida` o el retorno del programa principal cierra todos los archivos abiertos.

La función `unlink(char *name)` elimina el nombre del archivo del sistema de archivos. Corresponde a la función de biblioteca estándar `remove`.

Ejercicio 8-1. Reescriba el programa `cat` del Capítulo 7 usando `lectura`, `escritura`, `apertura` y `cierre` en lugar de sus equivalentes de biblioteca estándar. Realizar experimentos para determinar las velocidades relativas de las dos versiones.

8.4 Acceso Aleatorio - Lseek

La entrada y la salida son normalmente secuenciales: cada `lectura` o `escritura` tiene lugar en una posición del archivo justo después de la anterior. Sin embargo, cuando sea necesario, un archivo puede leerse o escribirse en cualquier orden arbitrario. La llamada del sistema `lseek` proporciona una forma de moverse en un archivo sin leer ni escribir ningún dato:

```
lseek largo (int fd, desplazamiento largo, origen int);
```

Establece la posición actual en el archivo cuyo descriptor es `fd` en `offset`, que se toma en relación con la ubicación especificada por Origin. La lectura o escritura subsiguiente comenzará en esa posición. `origin` puede ser 0, 1 o 2 para especificar que el `desplazamiento` se medirá desde el principio, desde la posición actual o desde el final del archivo, respectivamente. Por ejemplo, para anexar a un archivo (el >> de redirección en el shell de UNIX, o `"a"` para `fopen`), busque hasta el final antes de escribir:

```
lseek(fd, 0L, 2);
```

Para volver al principio ("rebobinar"),

```
lseek(fd, 0L, 0);
```

Fíjate en el argumento `0L`; también podría escribirse como `(long) 0` o simplemente como `0` si `lseek` se declara correctamente.

Con `lseek`, es posible tratar los archivos más o menos como matrices, al precio de un acceso más lento. Por ejemplo, la siguiente función lee cualquier número de bytes de cualquier lugar arbitrario de un archivo. Devuelve el número leído, o `-1` en caso de error.

```
#include "syscalls.h"

/*get: leer n bytes desde la posición pos
*/ int get(int fd, long pos, char *buf, int
n)
{
    if (lseek(fd, pos, 0) >= 0) /* get to pos */
        return read(fd, buf, n);
    más
        retorno -1;
}
```

El valor devuelto de `lseek` es un long que proporciona la nueva posición en el archivo, o `-1` si se produce un error. La función de biblioteca estándar `fseek` es similar a `lseek`, excepto que el primer argumento es un `FILE *` y el retorno es distinto de cero si se produjo un error.

8.5 Ejemplo: una implementación de Fopen y Getc

Vamos a ilustrar cómo encajan algunas de estas piezas mostrando una implementación de las rutinas estándar de la biblioteca `fopen` y `getc`.

Recuerde que los archivos de la biblioteca estándar se describen mediante punteros de archivo en lugar de descriptores de archivo. Un puntero de archivo es un puntero a una estructura que contiene varios fragmentos de información sobre el archivo: un puntero a un búfer, por lo que el archivo se puede leer en fragmentos grandes; un recuento del número de caracteres que quedan en el búfer; un puntero a la siguiente posición de carácter en el búfer; el descriptor del archivo; y banderas que describen el modo de lectura/escritura, el estado de error, etc.

La estructura de datos que describe un fichero se encuentra en `<stdio.h>`, que debe incluirse (por `#include`) en cualquier fichero de origen que utilice rutinas de la biblioteca de entrada/salida estándar. También se incluye en las funciones de esa biblioteca. En el siguiente extracto de un `<stdio.h> típico`, los nombres que están destinados a ser utilizados solo por funciones de la biblioteca comienzan con un guión bajo, por lo que es menos probable que colisionen con los nombres del programa de un usuario. Esta convención es utilizada por todas las rutinas de biblioteca estándar.

```
#define NULO      0
#define EF        (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20    /* #files máximo abierto a la
                  vez */

typedef struct _iobuf {
    int CNT;        /* caracteres a la izquierda */
    char *ptr;/* posición del siguiente carácter
                  */ char *base;/* ubicación
    del búfer */
    Int flag;/* modo de acceso a archivos */
    int             fd;/* descriptor de
    fichero */
```

```
} ARCHIVO;
```

```
ARCHIVO EXTERNO _iob[OPEN_MAX];

#define  stdin  (&_iob[0])
#define stdout  (&_iob[1])
#define stderr (&_iob[2])

enum _flags {
     _LEER   = 01,   /* archivo abierto para lectura */
     _WRITE = 02,    /* archivo abierto para escribir */
     _UNBUF = 04,    /* el archivo no está almacenado en búfer */
     _EF     = 010, /* Se ha producido EOF en este archivo */
     _ERRAR  = 020   /* se ha producido un error en este archivo */
};

int _fillbuf(ARCHIVO *);
int _flushbuf(int, ARCHIVO *);

#define  feof(p)  ((p)->flag  &  _EOF)  !=  0)
#define  ferror(p)  ((p)->flag  &  _ERR)  !=  0)
#define fileno(p) ((p)->fd)

#define getc(p)   (--(p)->cnt >= 0 \
               ? (carácter sin signo) *(p)->ptr++ :
_fillbuf(p)) #define putc(x,p) (--(p)->cnt >= 0 \
               ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()getc(stdin) #define
putcher(x), putc((x), stdout)
```

Normalmente, `la macro getc` disminuye el recuento, avanza el puntero y devuelve el carácter. (Recuérdese que una larga `#define` continúa con una barra invertida). Sin embargo, si el recuento es negativo, `getc` llama a la función `_fillbuf` para reponer el búfer, reinicializar el contenido de la estructura y devolver un carácter. Los caracteres se devuelven `sin firmar`, lo que garantiza que todos los caracteres serán positivos.

Aunque no discutiremos ningún detalle, hemos incluido la definición de `putc` para mostrar que funciona de la misma manera que `getc`, llamando a una función `_flushbuf` cuando su búfer está lleno. También hemos incluido macros para acceder al error y al estado de fin de archivo y el descriptor del archivo.

Ahora se puede escribir la función `fopen`. La mayor parte de `fopen` se ocupa de abrir el archivo y colocarlo en el lugar correcto, y de configurar los bits de la bandera para indicar el estado adecuado. `fopen` no asigna ningún espacio de búfer; esto se hace mediante `_fillbuf` cuando el fichero se lee por primera vez.

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMANENTES 0666  /* RW para propietario, grupo, otros */

Archivo *Phonen (cuatro *nombre, cuatro * modo)
{
    int fd;
    ARCHIVO
    *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        devuelve NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
           break;        /* Tragamonedas gratis
                                encontrada */
    if (fp >= _iob + OPEN_MAX)/* no hay ranuras
        libres */ devuelve NULL;
```

```
    if (*mode == 'w')
        fd = creat(nombre, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(nombre, O_WRONLY, 0)) == -
            1) fd = creat(nombre, PERMS);
        lseek(fd, 0L, 2);
    } de lo contrario
        fd = abierto(nombre, O_RDONLY, 0);
    if (fd == -1)/* no pudo acceder al nombre */
        devuelve NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    devolver fp;
}
```

Esta versión de `fopen` no maneja todas las posibilidades de modo de acceso del estándar, aunque agregarlas no requeriría mucho código. En particular, nuestro `fopen` no reconoce la "b" que señala el acceso binario, ya que eso no tiene sentido en los sistemas UNIX, ni el "+" que permite tanto la lectura como la escritura.

La primera llamada a `getc` para un archivo en particular encuentra un conteo de cero, lo que fuerza una llamada de
`_fillbuf`. Si _fillbuf encuentra que el archivo no está abierto para su lectura, devuelve `EOF` inmediatamente. De lo contrario, intenta asignar un búfer (si se va a almacenar en búfer la lectura).

Una vez establecido el búfer, `_fillbuf` llama a `read` para llenarlo, establece el recuento y los punteros, y devuelve el carácter al principio del búfer. Las llamadas posteriores a `_fillbuf` encontrarán un búfer asignado.

```
#include "syscalls.h"

/* _fillbuf: asigna y rellena el búfer de
entrada */ int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag&(_READ|_EOF_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL)/* aún no hay búfer */

        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF;/* no se puede obtener el búfer */

    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->bandera |=
        _EOF; más
            fp->bandera |=
        _ERR; fp->cnt = 0;
        devolver EOF;
    }
    return (carácter sin firmar) *fp->ptr++;
}
```

El único cabo suelto que queda es cómo se inicia todo. El _iob de la matriz debe definirse e inicializarse para `stdin`, `stdout` y `stderr`:

```
ARCHIVO _iob[OPEN_MAX] = { /* stdin, stdout, stderr */
```

```
        { 0, (señor *) 0, (señor *) 0, _reed, 0},
        { 0, (señor *) 0, (señor *) 0, _right, 1},
        { 0, (señor*) 0, (señor*) 0, _right, | _unbuff, 2 }
};
```

La inicialización de la parte de flag de la estructura muestra que `stdin` se va a leer, `stdout` se va a escribir y `stderr` se va a escribir sin búfer.

Ejercicio 8-2. Reescriba `fopen` y `_fillbuf` con campos en lugar de operaciones de bits explícitas. Compare el tamaño del código y la velocidad de ejecución.

Ejercicio 8-3. Diseñe y escriba `_flushbuf`, `fflush` y `fclose`.

Ejercicio 8-4. La función de biblioteca estándar

```
int fseek(ARCHIVO *fp, desplazamiento largo, origen int)
```

es idéntico a `lseek`, excepto que `fp` es un puntero de archivo en lugar de un descriptor de archivo y el valor devuelto es un `estado int`, no una posición. Escribe `fseek`. Asegúrese de que su `fseek` coincida correctamente con el almacenamiento en búfer realizado para las otras funciones de la biblioteca.

8.6 Ejemplo: directorios de listados

A veces se requiere un tipo diferente de interacción con el sistema de archivos: determinar la información *sobre* un archivo, no lo que contiene. Un programa de listado de directorios como el comando `ls de UNIX` es un ejemplo: imprime los nombres de los archivos en un directorio y, opcionalmente, otra información, como tamaños, permisos, etc. El comando dir de MS-DOS es análogo.

Dado que un directorio UNIX es solo un archivo, solo necesita leerlo para recuperar los nombres de archivo. Pero es necesario utilizar una llamada al sistema para acceder a otra información sobre un archivo, como su tamaño. En otros sistemas, puede ser necesaria una llamada al sistema incluso para acceder a los nombres de archivo; este es el caso de MS-DOS, por ejemplo. Lo que queremos es proporcionar acceso a la información de una manera relativamente independiente del sistema, aunque la implementación pueda ser muy dependiente del sistema.

Ilustraremos algo de esto escribiendo un programa llamado `fsize`. `fsize` es una forma especial de `ls` que imprime los tamaños de todos los archivos nombrados en su lista de argumentos de línea de comandos. Si uno de los archivos es un directorio, `fsize` se aplica de forma recursiva a ese directorio. Si no hay ningún argumento, procesa el directorio actual.

Comencemos con una breve revisión de la estructura del sistema de archivos UNIX. Un *directorio* es un archivo que contiene una lista de nombres de archivo y alguna indicación de dónde se encuentran. La "ubicación" es un índice en otra tabla llamada "lista de inodos". El *inodo de* un archivo es donde se guarda toda la información sobre el archivo, excepto su nombre. Una entrada de directorio generalmente consta de solo dos elementos, el nombre de archivo y un número de inodo.

Lamentablemente, el formato y el contenido preciso de un directorio no son los mismos en todas las versiones del sistema. Así que dividiremos la tarea en dos partes para tratar de aislar las partes no portátiles. El nivel externo define una estructura llamada `Dirent` y tres rutinas `opendir`, `readdir` y `closedir` para proporcionar acceso independiente del sistema al nombre y al número de inodo en una entrada de directorio. Escribiremos `fsize` con esta interfaz. A continuación, mostraremos cómo implementarlos en sistemas que utilizan la misma estructura de directorios que la versión 7 y el Sistema V UNIX; Las variantes se dejan como ejercicios.

La estructura `Dirent` contiene el número de inodo y el nombre. La longitud máxima de un componente de nombre de archivo es `NAME_MAX`, que es un valor dependiente del sistema. `opendir` devuelve un puntero a una estructura llamada `DIR`, análoga a `FILE`, que es utilizada por `readdir` y `closedir`. Esta información se recoge en un fichero denominado `dirent.h`.

```
#define NAME_MAX   14 /* componente de nombre de archivo más largo; */
                              /* dependiente del sistema */

typedef struct {/* entrada de directorio portátil
                        */ largo ino;/* número de
    inodo */
    cuatro nombres[name_max+1]; /* nombre + '\0' terminador */
} Diferente;

typedef struct {       /* DIR mínimo: sin almacenamiento en búfer, etc. */
    int fd;               /* descriptor de archivo para el directorio */
    Dirent d;             /* la entrada del directorio */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
vacío cerradoir(DIR *dfd);
```

La `estadística` de llamada al sistema toma un nombre de archivo y devuelve toda la información en el inodo de ese archivo, o `-1` si hay un error. Es decir

```
char *nombre;
struct stat stbuf;
int stat(char *, struct stat *);

stat(nombre, &stbuf);
```

Rellena el `stbuf` `de` la estructura con la información del inodo para el nombre del archivo. La estructura que describe el valor devuelto por `stat` se encuentra en `<sys/stat.h>` y, por lo general, tiene el siguiente aspecto:

```
Estadística de estructura /* información de inodo devuelta por stat */
{
    dev_t     st_dev;      /* dispositivo de
    inodo */ ino_t         st_ino;    /* número
    de inodo */ corto      st_mode;   /* bits de
    modo */
    corto     st_nlink;    /* número de enlaces al
    archivo */ corto       st_uid;    /* ID de usuario
    de los propietarios */
    corto     st_gid;      /* ID del grupo de
    propietarios */ dev_t  st_rdev;   /* para
    archivos especiales */
    off_t     st_size;     /* tamaño del archivo en
    caracteres */ time_t   st_atime;  /* hora de último
    acceso */ time_t       st_mtime;  /* hora de la
    última modificación */ time_t     st_ctime;   /*
    hora creada originalmente */
};
```

La mayoría de estos valores se explican mediante los campos de comentarios. Los tipos como `dev_t` y `ino_t`
se definen en `<sys/types.h>`, que también deben incluirse.

La entrada `st_mode` contiene un conjunto de indicadores que describen el archivo. Las definiciones de los indicadores también se incluyen en `<sys/types.h>`; solo necesitamos la parte que se ocupa del tipo de archivo:

```
#define S_IFMT0160000 /* tipo de archivo:   */
```

```
#define S_IFDIR0040000 /* directorio */
#define S_IFCHR0020000 /* carácter especial
                  */ #define S_IFBLK0060000
                  /* bloque especial */ #define
S_IFREG0010000 /* regular */
/* ... */
```

Ahora estamos listos para escribir el programa `fsize`. Si el modo obtenido de `stat` indica que un archivo no es un directorio, entonces el tamaño está a mano y se puede imprimir directamente. Sin embargo, si el nombre es un directorio, entonces tenemos que procesar ese directorio un archivo a la vez; A su vez, puede contener subdirectorios, por lo que el proceso es recursivo.

La rutina principal se ocupa de los argumentos de la línea de comandos; Entrega cada argumento a la función
`fsize`.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>/* banderas para lectura y escritura
*/ #include <sys/types.h> /* typedefs */
#include <sys/stat.h>/* estructura devuelta por stat */
#include "dirent.h"

void fsize(char *)

/* nombre del archivo de
impresión */ main(int argc,
char **argv)
{
    if (argc == 1) /* default: directorio actual */
        fsize(".");
    más
        while (--argc > 0)
            fsize(*++argv);
    devuelve 0;
}
```

La función `fsize` imprime el tamaño del archivo. Sin embargo, si el archivo es un directorio, `fsize` llama primero `a dirwalk` para controlar todos los archivos que contiene. Observe cómo se utilizan los nombres de los `S_IFMT` y `S_IFDIR` para decidir si el archivo es un directorio. El paréntesis es importante, porque la precedencia de `&` es menor que la de ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: imprime el nombre del fichero "name"
*/ void fsize(char *name)
{
    struct stat stbuf;

    if (stat(nombre, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        devolución;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(nombre, fsize);
    printf("%8ld %s\n", stbuf.st_size, nombre);
}
```

La función `dirwalk` es una rutina general que aplica una función a cada archivo de un directorio. Abre el directorio, recorre los archivos que contiene, llamando a la función en cada uno de ellos, luego cierra el directorio y regresa. Dado que `fsize` llama a `dirwalk` en cada directorio, las dos funciones se llaman entre sí de forma recursiva.

```
#define MAX_PATH 1024

/* dirwalk: aplicar fcn a todos los archivos en
dir */ void dirwalk(char *dir, void (*fcn)(char
*))
{
```

```
        nombrados
        cuatro[max_lesson];
        Diorent *DP;
        DIR *dfd;

        if ((dfd = opendir(dir)) == NULL) {
            fprintf(stderr, "dirwalk: no se puede abrir %s\n",
            dir); devolución;
        }
        while ((dp = readdir(dfd)) != NULL) {
            if (strcmp(dp->name, ".") == 0
                || strcmp(dp->nombre, ".."))
                continuar;    /* Saltar uno mismo y el padre */
            IF(Strollen(dir)+Strollen(DP->name)+2 > CZOF(name))
                Fprintf(Studder, "dirwalk: named %s %s to long\n",
                    Dir, DP->bajo el nombre);
            else {
                Sprintf(nombrado, "%s/%s", dir, dp->);
                (*FCN)(Nombre);
            }
        }
        cerradoIR(DFD);
    }
```

Cada llamada a `readdir` devuelve un puntero a la información del siguiente archivo, o `NULL` cuando no quedan archivos. Cada directorio siempre contiene entradas para sí mismo, llamadas `".",` y su padre, `"..`"Deben omitirse, o el programa se repetirá para siempre.

Hasta este último nivel, el código es independiente de cómo se formatean los directorios. El siguiente paso es presentar versiones mínimas de `opendir`, `readdir` y `closedir` para un sistema específico. Las siguientes rutinas son para sistemas UNIX versión 7 y System V; Utilizan la información del directorio en el encabezado `<sys/dir.h>`, que tiene el siguiente aspecto:

```
#ifndef DIRSIZ
#define DIRSIZ 14
#endif
struct directo {  /* entrada en el directorio */
    ino_t d_ino;           /* número de inodo */
    char d_name[DIRSIZ]; /* el nombre largo no tiene '\0' */
};
```

Algunas versiones del sistema permiten nombres mucho más largos y tienen una estructura de directorios más complicada.

El tipo `ino_t` es una `definición de tipo` que describe el índice en la lista de inodos. Resulta que no está `firmado` en los sistemas que usamos regularmente, pero este no es el tipo de información para incrustar en un programa; podría ser diferente en un sistema diferente, por lo que la definición de `tipo` es mejor. Un conjunto completo de tipos de "sistema" se encuentra en `<sys/types.h>`.

`Opendir` abre el directorio, verifica que el archivo es un directorio (esta vez mediante la llamada del sistema `fstat`, que es como `stat` excepto que se aplica a un descriptor de archivo), asigna una estructura de directorio y registra la información:

```
usted es un estado (usted fd, struct state*);

/* opendir: abre un directorio para llamadas
readdir */ DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
```

```
        DIR *dp;

        if ((fd = open(dirname, O_RDONLY, 0)) == -1
         || fstat(fd, &stbuf) == -1
         || (stbuf.st_mode y S_IFMT) != S_IFDIR
         || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
             return NULL;
        dp->fd = fd;
        dp de
        retorno;
    }
```

`closedir` cierra el archivo de directorio y libera el espacio:

```
    /* closedir: cerrar directorio abierto por opendir
    */ void closedir(DIR *dp)
    {
        if (dp) {
            cerrar(dp-
            >fd); Libre
            (DP);
        }
    }
```

Por último, `readdir` utiliza `read` para leer cada entrada del directorio. Si una ranura de directorio no está actualmente en uso (porque se ha eliminado un archivo), el número de inodo es cero y esta posición se omite. De lo contrario, el número y el nombre del inodo se colocan en una `estructura estática` y se devuelve al usuario un puntero a ella. Cada llamada sobrescribe la información de la anterior.

```
    #include <sys/dir.h>   /* estructura de directorios local */

    /* readdir: lee las entradas del directorio en
    secuencia */ Dirent *readdir(DIR *dp)
    {
        struct dirbuf directo; /* estructura de directorios
        local */ static Dirent d;/* regreso: estructura
        portátil */

        while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
                        == tamañode(dirbuf)) {
            if (dirbuf.d_ino == 0) /* ranura no en uso */
                continuar;
            d.ino = dirbuf.d_ino;
            strncpy(d.name, dirbuf.d_name, DIRSIZ);
            d.name[DIRSIZ] = '\0'; /* asegurar la
            terminación */ devolución &d;
        }
        devuelve NULL;
    }
```

Aunque el programa `fsize` es bastante especializado, ilustra un par de ideas importantes. En primer lugar, muchos programas no son "programas del sistema"; simplemente utilizan información que es mantenida por el sistema operativo. Para estos programas, es crucial que la representación de la información aparezca solo en los encabezados estándar, y que los programas incluyan esos encabezados en lugar de incrustar las declaraciones en sí mismos. La segunda observación es que, con cuidado, es posible crear una interfaz para objetos dependientes del sistema que sea a su vez relativamente independiente del sistema. Las funciones de la biblioteca estándar son buenos ejemplos.

Ejercicio 8-5. Modifique el `programa fsize` para imprimir el resto de la información contenida en la entrada del inodo.

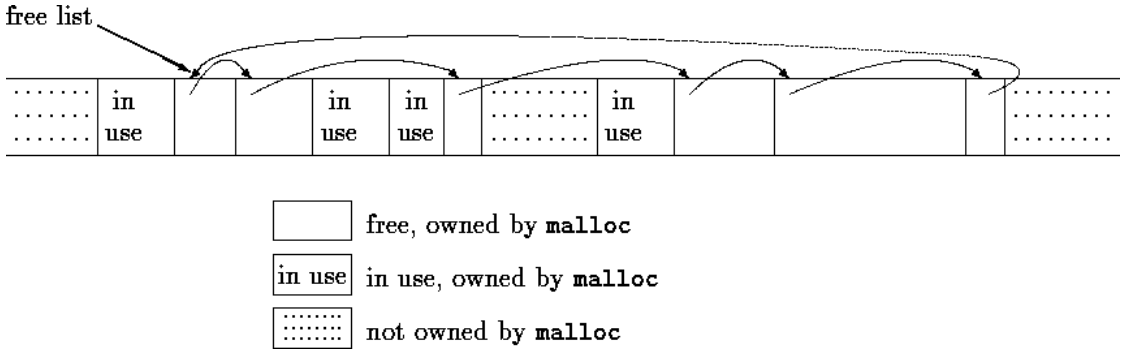
8.7 Ejemplo: un asignador de almacenamiento

En el Capítulo 5, presentamos un asignador de almacenamiento orientado a pilas limitadas. La

versión que ahora escribiremos no tiene restricciones. Las llamadas a `malloc` y `free` pueden ocurrir en cualquier orden; `Llamadas` a malloc

en el sistema operativo para obtener más memoria según sea necesario. Estas rutinas ilustran algunas de las consideraciones involucradas en la escritura de código dependiente de la máquina de una manera relativamente independiente de la máquina, y también muestran una aplicación en la vida real de estructuras, uniones y definición de `tipos`.

En lugar de asignar desde una matriz de tamaño fijo compilada, `malloc` solicitará espacio del sistema operativo según sea necesario. Dado que otras actividades del programa también pueden solicitar espacio sin llamar a este asignador, es posible que el espacio que `administra malloc` no sea contiguo. Por lo tanto, su almacenamiento gratuito se mantiene como una lista de bloques libres. Cada bloque contiene un tamaño, un puntero al siguiente bloque y el propio espacio. Los bloques se mantienen en orden creciente de dirección de almacenamiento, y el último bloque (dirección más alta) apunta al primero.



Cuando se realiza una solicitud, la lista gratuita se escanea hasta que se encuentra un bloque lo suficientemente grande. Este algoritmo se llama "primer ajuste", en contraste con "mejor ajuste", que busca el bloque más pequeño que satisfaga la solicitud. Si el bloque tiene exactamente el tamaño solicitado, se desvincula de la lista y se devuelve al usuario. Si el bloque es demasiado grande, se divide y se devuelve la cantidad adecuada al usuario mientras el residuo permanece en la lista libre. Si no se encuentra un bloque lo suficientemente grande, el sistema operativo obtiene otro trozo grande y se vincula a la lista libre.

La liberación también provoca una búsqueda en la lista libre, para encontrar el lugar adecuado para insertar el bloque que se está liberando. Si el bloque que se libera es adyacente a un bloque libre a ambos lados, se fusiona con él en un solo bloque más grande, por lo que el almacenamiento no se fragmenta demasiado. Determinar la adyacencia es fácil porque la lista libre se mantiene en orden decreciente de dirección.

Un problema, al que aludimos en el Capítulo 5, es asegurarse de que el almacenamiento devuelto por `malloc` esté alineado correctamente para los objetos que se almacenarán en él. Aunque las máquinas varían, para cada máquina hay un tipo más restrictivo: si el tipo más restrictivo se puede almacenar en una dirección particular, todos los demás tipos también pueden serlo. En algunas máquinas, el tipo más restrictivo es un `doble`; en otras, `int` o `long` es suficiente.

Un bloque libre contiene un puntero al siguiente bloque de la cadena, un registro del tamaño del bloque y luego el espacio libre en sí; la información de control al principio se llama "encabezado". Para simplificar la alineación, todos los bloques son múltiplos del tamaño del encabezado y el encabezado está alineado correctamente. Esto se logra mediante una unión que contiene la estructura de encabezado deseada y una instancia del tipo de alineación más restrictivo, que hemos hecho arbitrariamente larga:

```
typedef long Align;/* para la alineación con el límite
```

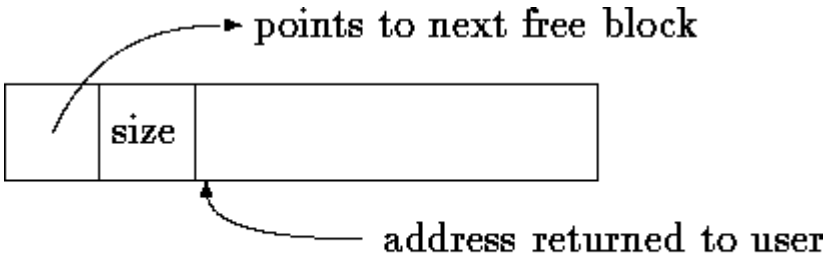
```
largo                    */ cabecera de unión {/* cabecera de
bloque */
```

```
    struct {
        encabezado de la unión *ptr; /* siguiente bloque si
        está en la lista libre */ tamaño sin signo;/*
        tamaño de este bloque */
    } s;
    Alinear x;          /* Alineación forzada de bloques */
};

encabezado de unión typedef Encabezado;
```

El campo `Alinear` nunca se utiliza; simplemente obliga a que cada encabezado se alinee en el límite del peor de los casos.

En `malloc`, el tamaño solicitado en caracteres se redondea al número adecuado de unidades del tamaño del encabezado; el bloque que se asignará contiene una unidad más, para el encabezado en sí, y este es el valor registrado en el campo de tamaño del encabezado. El puntero devuelto por `malloc` apunta al espacio libre, no al encabezado en sí. El usuario puede hacer cualquier cosa con el espacio solicitado, pero si se escribe algo fuera del espacio asignado, es probable que la lista se codifique.



A block returned by **malloc**

El campo size es necesario porque los bloques controlados por `malloc` no necesitan ser contiguos: no es posible calcular tamaños mediante aritmética de punteros.

La `base variable` se utiliza para empezar. Si `freep` es `NULL`, como lo es en la primera llamada de `malloc`, entonces se crea una lista libre degenerada; contiene un bloque de tamaño cero, y apunta a sí mismo. En cualquier caso, a continuación, se realiza una búsqueda en la lista gratuita. La búsqueda de un bloque libre de tamaño adecuado comienza en el punto (`freep`) donde se encontró el último bloque; Esta estrategia ayuda a mantener la lista homogénea. Si se encuentra un bloque demasiado grande, el extremo de la cola se devuelve al usuario; De esta manera, el encabezado del original solo necesita que se le ajuste su tamaño. En todos los casos, el puntero devuelto al usuario apunta al espacio libre dentro del bloque, que comienza una unidad más allá del encabezado.

```
static Header base;/* lista vacía para empezar */ static
Header *freep = NULL;            /* inicio de la lista
gratuita */

/* malloc: asignador de almacenamiento de propósito
general */ void *malloc(nbytes sin signo)
{
    Encabezado *p, *prevp;
    Encabezado *moreroce(sin
    firmar); nunits sin signo;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
    if ((prevp = freep) == NULL) {/* aún no hay lista libre
    */
        base.s.ptr = freeptr = prevptr = &base;
```

```
    base.s.tamaño = 0;
}
```

```
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) { /* suficientemente
        grande */
            if (p->s.size == nunits) /* exactamente
                */ prevp->s.ptr = p->s.ptr;
            else {/* asignar extremo de cola */ p-
                >s.size -= nunits;
                p += p->s.tamaño;
                p->s.tamaño = nunidades;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep) /* envuelto alrededor de la
            lista libre */ if ((p = morecore(nunits))
            == NULL)
                devuelve NULL;  /* no queda ninguno */
    }
}
```

La función `morecore` obtiene almacenamiento del sistema operativo. Los detalles de cómo lo hace varían de un sistema a otro. Ya que pedir memoria al sistema es una operación comparativamente cara. no queremos hacer eso en cada llamada a `malloc`, por lo que `morecore` solicita al menos `unidades NALLOC`; este bloque más grande se cortará según sea necesario. Después de establecer el campo de tamaño, `morecore` inserta la memoria adicional en la arena llamando a `free`.

La llamada al sistema UNIX `sbrk(n)` devuelve un puntero a `n` bytes más de almacenamiento. `sbrk` devuelve `-1` si no había espacio, aunque `NULL` podría haber sido un mejor diseño. El `-1` debe convertirse en `char *` para que se pueda comparar con el valor devuelto. Una vez más, las conversiones hacen que la función sea relativamente inmune a los detalles de la representación del puntero en diferentes máquinas. Sin embargo, todavía hay una suposición de que los punteros a diferentes bloques devueltos por `sbrk` se pueden comparar significativamente. Esto no está garantizado por el estándar, que permite comparaciones de punteros solo dentro de una matriz. Por lo tanto, esta versión de `malloc` es portátil solo entre máquinas para las que la comparación general de punteros es significativa.

```
#define NALLOC 1024    /* #units mínimo a solicitar */

/* morecore: pedir al sistema más memoria */
encabezado estático *morecore(nu sin signo)
{
    char *cp, *sbrk(int);
    Encabezado *arriba;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Encabezado));
    if (cp == (char *) -1)/* sin espacio en
        absoluto */ devuelve NULL;
    arriba = (Encabezado
    *) cp; up->s.tamaño =
    nu; libre((nulo
    *)(arriba+1));
    devolver gratis;
}
```

`La libertad` en sí misma es lo último. Escanea la lista libre, comenzando por `freep`, buscando el lugar para insertar el bloque libre. Esto es entre dos bloques existentes o al final de la lista. En cualquier caso, si el bloque que se libera es adyacente a cualquiera de los vecinos, se combinan los bloques adyacentes. Los únicos problemas son mantener los punteros apuntando a las cosas correctas y los tamaños correctos.

```
/* gratis: poner bloquear ap en la lista
```

```
libre */ void free (void *ap)
{
```

```
    Encabezado *bp, *p;

    bp = (Encabezado *)ap - 1;/* punto para bloquear el
    encabezado */ for (p = freep; !( BP > P & < P->S.Ptr); p
    = p->s.ptr)
         Si (p >= p->s.ptr && (bp > p || p-< p->s.ptr))
             quebrar; /* Bloque liberado al inicio o al final de la arena */

    if (bp + bp->size == p->s.ptr) {/* se unen a la parte
        superior nbr */ bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } de lo contrario
        bp->s.ptr = p->s.ptr;
    if (p + p->tamaño == bp) {/* se unen a NBR inferior */ p-
        >s.tamaño += bp->s.tamaño;
        p->s.ptr = bp->s.ptr;
    } de lo contrario
        p->s.ptr = bp;
    libre = p;
}
```

Aunque la asignación de almacenamiento depende intrínsecamente de la máquina, el código anterior ilustra cómo las dependencias de la máquina se pueden controlar y limitar a una parte muy pequeña del programa. El uso de `typedef` y `union` identifica la alineación (dado que `sbrk` proporciona un puntero adecuado). Las conversiones organizan que las conversiones de puntero se hagan explícitas, e incluso se enfrentan a una interfaz del sistema mal diseñada. Aunque los detalles aquí están relacionados con la asignación de almacenamiento, el enfoque general también es aplicable a otras situaciones.

Ejercicio 8-6. La función de biblioteca estándar `calloc(n,size)` devuelve un puntero a `n` objetos de tamaño size, con el almacenamiento inicializado en cero. Escriba `calloc`, llamando a `malloc` o modificándolo.

Ejercicio 8-7. `malloc` acepta una solicitud de tamaño sin verificar su plausibilidad; `free` cree que el bloque que se le pide free contiene un campo de tamaño válido. Mejore estas rutinas para que hagan más dolores con la verificación de errores.

Ejercicio 8-8. Escriba una rutina `bfree(p,n)` que liberará cualquier bloque arbitrario p de `n` caracteres en la lista libre mantenida por `malloc` y `free`. Mediante el uso de `bfree`, un usuario puede agregar una matriz estática o externa a la lista libre en cualquier momento.

Apéndice A - Manual de Referencia

A.1 Introducción

Este manual describe el lenguaje C especificado por el borrador presentado a ANSI el 31 de octubre de 1988, para su aprobación como "Estándar Americano para Sistemas de Información - Lenguaje de programación C, X3.159-1989". El manual es una interpretación de la norma propuesta, no la norma en sí misma, aunque se ha tenido cuidado de que sea una guía fiable del idioma.

En su mayor parte, este documento sigue las líneas generales de la norma, que a su vez sigue la de la primera edición de este libro, aunque la organización difiere en los detalles. Excepto por el cambio de nombre de algunas producciones, y no formalizar las definiciones de los tokens léxicos o el preprocesador, la gramática dada aquí para el lenguaje propiamente dicho es equivalente a la del estándar.

> A lo largo de este manual, el material de comentarios está sangrado y escrito en letra más pequeña, tal como está. La mayoría de las veces, estos comentarios ponen de relieve las formas en que el Estándar C de ANSI difiere del lenguaje definido por la primera edición de este libro, o de los refinamientos introducidos posteriormente en varios compiladores.

A.2 Convenciones léxicas

Un programa consta de una o más *unidades de traducción* almacenadas en archivos. Se traduce en varias fases, que se describen en el párrafo A.12. Las primeras fases realizan transformaciones léxicas de bajo nivel, llevan a cabo directivas introducidas por las líneas que comienzan con el carácter # y realizan la definición y expansión de macros. Cuando se completa el preprocesamiento del párrafo A.12 , el programa se ha reducido a una secuencia de tokens.

A.2.1 Fichas

Hay seis clases de tokens: identificadores, palabras clave, constantes, literales de cadena, operadores y otros separadores. Los espacios en blanco, las tabulaciones horizontales y verticales, las nuevas líneas, los saltos de formulario y los comentarios que se describen a continuación (colectivamente, "espacio en blanco") se ignoran, excepto porque separan los tokens. Se requiere un poco de espacio en blanco para separar los identificadores, las palabras clave y las constantes adyacentes.

Si el flujo de entrada se ha separado en tokens hasta un carácter determinado, el siguiente token es la cadena más larga de caracteres que podría constituir un token.

A.2.2 Comentarios

Los caracteres `/*` introducen un comentario, que termina con los caracteres `*/`. Los comentarios no se anidan y no se producen dentro de una cadena o literales de carácter.

A.2.3 Identificadores

Un identificador es una secuencia de letras y números. El primer carácter debe ser una letra; El guión bajo _ cuenta como una letra. Las letras mayúsculas y minúsculas son diferentes. Los identificadores pueden tener cualquier longitud y, en el caso de los identificadores internos, al menos los primeros 31 caracteres son significativos;

Algunas implementaciones pueden tardar más caracteres. Los identificadores internos incluyen los nombres de macros de preprocesador y todos los demás nombres que no tienen vinculación externa (Par.A.11.2). Los identificadores con vinculación externa son más restringidos: las implementaciones pueden hacer que tan solo los primeros seis caracteres sean significativos y pueden ignorar las distinciones entre mayúsculas y minúsculas.

A.2.4 Palabras clave

Los siguientes identificadores están reservados para el uso como palabras clave y no se pueden utilizar de otra manera:

Automático	doble	Int	Estructura
quebrar	más	largo	interruptor
caso	Enumeración	registro	typedef
carbonizar	Extern	devolución	unión
Const	flotar	corto	Unsigned
continuar	para	fichado	vacío
predeterminado	Goto	tamañode	volátil
hacer	si	estático	mientras

Algunas implementaciones también reservan las palabras `fortran` y `asm`.

Las palabras clave `const`, `signed` y `volatile` son nuevas con el estándar ANSI; `enum` y `void` son nuevos desde la primera edición, pero de uso común; `La entrada`, anteriormente reservada pero nunca utilizada, ya no está reservada.

A.2.5 Constantes

Hay varios tipos de constantes. Cada uno tiene un tipo de datos; En el párrafo A.4.2 se analizan los tipos básicos:

constante:
constante-entera
constante de carácter
constante flotante
constante de enumeración

A.2.5.1 Constantes enteras

Una constante entera que consta de una secuencia de dígitos se considera octal si comienza con 0 (dígito cero), decimal en caso contrario. Las constantes octales no contienen los dígitos `8` o `9`. Una secuencia de dígitos precedida por `0x` o `0X` (dígito cero) se toma como un entero hexadecimal. Los dígitos hexadecimales incluyen `la A` o la `A` a la `f` o `la F` con valores `del 10` al `15`.

Una constante entera puede tener el sufijo U o `U`, para especificar que no tiene signo. También puede tener el sufijo de la letra `l` o `l` para especificar que es largo.

El tipo de una constante entera depende de su forma, valor y sufijo. (Véase Párrafo A.4 para una discusión de los tipos). Si no tiene sufijo y es decimal, tiene el primero de estos tipos en el que se puede representar su valor: `int`, `long int`, `unsigned long int`. Si es sin sufijo, octal o hexadecimal, tiene el primero posible de estos tipos: `int`, `unsigned int`, `long int`, `unsigned long int`. Si tiene el sufijo `u` o `U`, entonces `unsigned int`, `unsigned long int`. Si tiene el sufijo `l` o `L`, entonces `long int`, `unsigned long int`. Si una constante entera tiene el sufijo `UL`, es `long sin signo`.

La elaboración de los tipos de constantes enteras va mucho más allá de la primera edición, que

simplemente causaba que las constantes enteras grandes fueran `largas`. Los sufijos U son nuevos.

A.2.5.2 Constantes de caracteres

Una constante de caracteres es una secuencia de uno o más caracteres encerrados entre comillas simples como en `'x'`. El valor de una constante de caracteres con un solo carácter es el valor numérico del carácter en el juego de caracteres de la máquina en tiempo de ejecución. El valor de una constante de varios caracteres está definido por la implementación.

Las constantes de caracteres no contienen el `carácter '` ni las nuevas líneas; para representarlas y algunos otros caracteres, se pueden utilizar las siguientes secuencias de escape:

Newline	NL (LF)	`\n`	barra invertida	`\`	`\\`
Pestaña horizontal	HT	`\t`	signo de interrogación	`?`	`\?`
Pestaña vertical	VT	`\v`	Cita simple	`'`	`\'`
retroceso	BS	`\b`	comilla doble	`"`	`\"`
Retorno de carro	CR	`\r`	Número octal	`ooo`	`\ooo`
Avance de encofrado	SS	`\f`	Número hexadecimal	`Hh`	`\xhh`
Alerta sonora	BELIO	`\un`			

El escape `\ooo` consta de la barra diagonal inversa seguida de 1, 2 o 3 dígitos octales, que se toman para especificar el valor del carácter deseado. Un ejemplo común de esta construcción es `\0` (no seguido de un dígito), que especifica el carácter NUL. El escape `\xhh` consta de la barra invertida, seguida de `x`, seguida de dígitos hexadecimales, que se toman para especificar el valor del carácter deseado. No hay límite en el número de dígitos, pero el comportamiento es indefinido si el valor del carácter resultante supera el del carácter más grande. En el caso de los caracteres de escape octales o hexadecimales, si la implementación trata el `tipo char` como signado, el valor se extiende por signo como si se convirtiera en el `tipo char`. Si el carácter que sigue a \ no es uno de los especificados, el comportamiento es indefinido.

En algunas implementaciones, hay un conjunto extendido de caracteres que no se pueden representar en el `tipo char`. Una constante en este conjunto extendido se escribe con una L precedente, por ejemplo `L'x'`, y se denomina constante de caracteres anchos. Esta constante tiene el tipo `wchar_t`, un tipo entero definido en el encabezado estándar `<stddef.h>`. Al igual que con las constantes de caracteres ordinarios, se pueden usar escapes hexadecimales; El efecto es indefinido si el valor especificado supera el que se puede representar con `wchar_t`.

> Algunas de estas secuencias de escape son nuevas, en particular la representación de caracteres hexadecimales. Los caracteres extendidos también son nuevos. Los conjuntos de caracteres comúnmente utilizados en las Américas y Europa occidental se pueden codificar para que quepan en el `tipo char`; la intención principal al agregar `wchar_t` era acomodar los idiomas asiáticos.

A.2.5.3 Constantes flotantes

Una constante flotante consta de una parte entera, una parte decimal, una parte fraccionaria, una `e` o `E`, un exponente entero con signo opcional y un sufijo de tipo opcional, uno de `f`, `F`, `l` o `L`. Tanto el número entero como la fracción constan de una secuencia de dígitos. Es posible que falte la parte entera o la parte de la fracción (no ambas); Es posible que falte el punto decimal o la `e` y el exponente (no ambos). El tipo viene determinado por el sufijo; `F` o `f` lo hace `flotar`, `L` o `l` lo hace `largo doble`, de lo contrario es `doble`.

A2.5.4 Constantes de enumeración

Los identificadores declarados como enumeradores (véase Par.A.8.4) son constantes de tipo `int`.

A.2.6 Literales de cadena

Un literal de cadena, también llamado constante de cadena, es una secuencia de caracteres entre comillas dobles como en `"..."`. Una cadena tiene el tipo "array of characters" y la clase de almacenamiento `static` (ver Par.A.3 a continuación) y se inicializa con los caracteres dados. El hecho de que los literales de cadena idénticos sean distintos se define en la implementación, y el comportamiento de un programa que intenta modificar un literal de cadena no está definido.

Los literales de cadena adyacentes se concatenan en una sola cadena. Después de cualquier concatenación, se anexa un byte nulo `\0` a la cadena para que los programas que examinan la cadena puedan encontrar su final. Los literales de cadena no contienen caracteres de nueva línea o comillas dobles; Para representarlos, están disponibles las mismas secuencias de escape que para las constantes de caracteres.

Al igual que con las constantes de caracteres, los literales de cadena en un juego de caracteres extendido se escriben con una L precedente, como en `L"..."`. Los literales de cadena de caracteres anchos tienen el tipo "array of `wchar_t`". La concatenación de literales de cadena ordinarios y anchos no está definida.

> La especificación de que los literales de cadena no necesitan ser distintos, y la prohibición de modificarlos, son nuevos en el estándar ANSI, al igual que la concatenación de literales de cadena adyacentes. Los literales de cadena de caracteres anchos son nuevos.

A.3 Notación sintáctica

En la notación sintáctica utilizada en este manual, las categorías sintácticas se indican con *letra cursiva* y palabras y caracteres literales en estilo `de máquina de escribir`. Las categorías alternativas generalmente se enumeran en líneas separadas; En unos pocos casos, un largo conjunto de alternativas estrechas se presenta en una línea, marcada con la frase "una de". Un símbolo opcional de terminal o no terminal lleva el subíndice "opt", de modo que, por ejemplo,

{ *expresiónopt* }

significa una expresión opcional, encerrada entre llaves. La sintaxis se resume en el párrafo A.13.

> A diferencia de la gramática dada en la primera edición de este libro, la que se da aquí hace explícita la precedencia y asociatividad de los operadores de expresión.

A.4 Significado de los identificadores

Los identificadores, o nombres, se refieren a una variedad de cosas: funciones, etiquetas de estructuras, uniones y enumeraciones, miembros de estructuras o uniones, constantes de enumeración, nombres de definiciones de tipo y objetos. Un objeto, a veces llamado variable, es una ubicación en el almacenamiento, y su interpretación depende de dos atributos principales: su *clase de almacenamiento* y su *tipo*. La clase de almacenamiento determina la duración del almacenamiento asociado con el objeto identificado; El tipo determina el significado de los valores encontrados en el objeto identificado. Un nombre también tiene un ámbito, que es la región del programa en la que se conoce, y un vínculo, que determina si el mismo nombre en otro ámbito se refiere al mismo objeto o función. El alcance y la vinculación se analizan en el párrafo A.11.

A.4.1 Clase de almacenamiento

Hay dos clases de almacenamiento: automático y estático. Varias palabras clave, junto con el contexto de la declaración de un objeto, especifican su clase de almacenamiento. Los objetos automáticos son locales de un bloque (Par.9.3) y se descartan al salir del bloque. Las declaraciones dentro de un bloque crean objetos automáticos si no se menciona ninguna especificación de clase de almacenamiento o si se utiliza el especificador auto. Los objetos declarados son automáticos y se almacenan (si es posible) en registros rápidos de la máquina.

Los objetos estáticos pueden ser locales a un bloque o externos a todos los bloques, pero en ambos casos conservan sus valores a través de la salida y la reentrada de funciones y bloques. Dentro de un bloque, incluido un bloque que proporciona el código para una función, los objetos estáticos se declaran con la palabra clave `static`. Los objetos declarados fuera de todos los bloques, en el mismo nivel que las definiciones de función, son siempre estáticos. Pueden convertirse en locales de una unidad de traducción en particular mediante el uso de la `palabra clave static`, lo que les proporciona *un enlace interno*. Se vuelven globales para un programa completo omitiendo una clase de almacenamiento explícita, o usando la palabra clave `extern`; esto les da *un enlace externo*.

A.4.2 Tipos básicos

Hay varios tipos fundamentales. El encabezado estándar `<limits.h>` descrito en el Apéndice B define los valores más grandes y más pequeños de cada tipo en la implementación local. Los números que figuran en el Apéndice B muestran las magnitudes mínimas aceptables.

Los objetos declarados como caracteres (`char`) son lo suficientemente grandes como para almacenar cualquier miembro del juego de caracteres de ejecución. Si un carácter genuino de ese conjunto se almacena en un `objeto char`, su valor es equivalente al código entero del carácter y no es negativo. Otras cantidades se pueden almacenar en variables `char`, pero el rango de valores disponible, y especialmente si el valor está firmado, depende de la implementación.

Los caracteres sin signo declarados `unsigned char` consumen la misma cantidad de espacio que los caracteres sin formato, pero siempre aparecen como no negativos; los caracteres con signo explícito declarados `como caracteres sin` signo también ocupan el mismo espacio que los caracteres sin formato.

> El tipo char `sin signo` no aparece en la primera edición de este libro, pero es de uso común. `El carácter firmado` es nuevo.

Además de los `tipos char` , hay disponibles `hasta tres tamaños de entero`, `int corto`, `int` y `entero largo`, declarados. Los objetos `int simples` tienen el tamaño natural sugerido por la arquitectura de la máquina anfitriona; los otros tamaños se proporcionan para satisfacer necesidades especiales. Los enteros más largos proporcionan al menos tanto almacenamiento como los más cortos, pero la implementación puede hacer que los enteros simples sean equivalentes a enteros cortos o enteros largos. Todos los `tipos int` representan valores con signo, a menos que se especifique lo contrario.

Los enteros sin signo, declarados con la palabra clave `unsigned`, obedecen a las leyes de la aritmética módulo 2n, donde *n* es el número de bits en la representación y, por lo tanto, la aritmética en cantidades sin signo nunca puede desbordarse. El conjunto de valores no negativos que se pueden almacenar en un objeto con signo es un subconjunto de los valores que se pueden almacenar en el objeto sin signo correspondiente, y la representación de los valores superpuestos es la misma.

Cualquiera de punto flotante de precisión simple (`flotador`), punto flotante de precisión doble (`doble`) y punto flotante de precisión adicional (`doble largo`) pueden ser sinónimos, pero los que están más adelante en la lista son al menos tan precisos como los anteriores.

`El doble largo` es nuevo. La primera edición hizo `que el flotador largo` equivalga al `doble`; la locución ha sido retirada.

Las enumeraciones son tipos únicos que tienen valores enteros; asociado a cada enumeración hay un conjunto de constantes con nombre (Par.A.8.4). Las enumeraciones se comportan como enteros, pero es habitual que un compilador emita una advertencia cuando a un objeto de una enumeración determinada se le asigna algo distinto de una de sus constantes o una expresión de su tipo.

Dado que los objetos de estos tipos se pueden interpretar como números, se denominarán tipos *aritméticos*. Los tipos `char` e `int` de todos los tamaños, cada uno con o sin signo, así como los tipos de enumeración, se denominarán colectivamente tipos enteros. Los tipos `float`, `double` y `long double` se denominarán *tipos flotantes.*

El tipo `void` especifica un conjunto vacío de valores. Se utiliza como el tipo devuelto por las funciones que no generan ningún valor.

A.4.3 Tipos derivados

Además de los tipos básicos, hay una clase conceptualmente infinita de tipos derivados construidos a partir de los tipos fundamentales de las siguientes maneras:

matrices de objetos de un tipo dado;
funciones que devuelven objetos de un
tipo dado; *punteros* a objetos de un tipo
determinado;
estructuras que contienen una secuencia de objetos de varios tipos;
uniones capaces de contener cualquiera de varios objetos de varios tipos. En

general, estos métodos de construcción de objetos se pueden aplicar de forma

recursiva.

A.4.4 Calificadores de tipo

El tipo de un objeto puede tener calificadores adicionales. Al declarar un objeto `const` se anuncia que su valor no se cambiará; al declararlo `volatile`, se anuncia que tiene propiedades especiales relevantes para la optimización. Ninguno de los calificadores afecta al intervalo de valores ni a las propiedades aritméticas del objeto. Los clasificatorios se discuten en el párrafo A.8.2.

A.5 Objetos y valores L

Un *objeto* es una región de almacenamiento con nombre; un *lvalue* es una expresión que se refiere a un objeto. Un ejemplo obvio de una expresión lvalue es un identificador con el tipo y la clase de almacenamiento adecuados. Hay operadores que producen lvalues, si `E` es una expresión de tipo puntero, `entonces *E` es una expresión lvalue que hace referencia al objeto al que `apunta E` . El nombre "lvalue" proviene de la expresión de asignación `E1 = E2` en la que el operando izquierdo `E1` debe ser una expresión lvalue. La discusión de cada operador especifica si espera operandos lvalue y si produce un lvalue.

A.6 Conversiones

Algunos operadores pueden, en función de sus operandos, provocar la conversión del valor de un operando de un tipo a otro. En esta sección se explica el resultado que se puede esperar de tales

Conversiones. El párrafo 6.5 resume las conversiones demandadas por la mayoría de los operadores ordinarios; se complementará según lo requiera la discusión de cada operador.

A.6.1 Promoción Integral

Un carácter, un entero corto o un campo de bits entero, todos con signo o no, o un objeto de tipo de enumeración, se pueden usar en una expresión siempre que se pueda usar un número entero. Si un `int` puede representar todos los valores del tipo original, el valor se convierte en `int`; de lo contrario, el valor se convierte en `unsigned int`. A este proceso se le llama *promoción integral*.

A.6.2 Conversiones Integrales

Cualquier entero se convierte en un tipo sin signo determinado mediante la búsqueda del valor no negativo más pequeño que sea congruente con ese entero, módulo uno más que el valor más grande que se puede representar en el tipo sin signo. En una representación de complemento a dos, esto es equivalente al truncamiento a la izquierda si el patrón de bits del tipo sin signo es más estrecho, y a los valores sin signo de relleno a cero y a los valores con signo que se extienden si el tipo sin signo es más ancho.

Cuando un entero se convierte en un tipo con signo, el valor no cambia si se puede representar en el nuevo tipo y se define en la implementación de otro modo.

A.6.3 Entero y flotante

Cuando un valor de tipo flotante se convierte en tipo entero, se descarta la parte fraccionaria; Si el valor resultante no se puede representar en el tipo entero, el comportamiento es indefinido. En concreto, no se especifica el resultado de convertir valores flotantes negativos en tipos enteros sin signo.

Cuando un valor de tipo entero se convierte en flotante y el valor está en el intervalo representable pero no es exactamente representable, el resultado puede ser el siguiente valor representable superior o el siguiente valor inferior representable. Si el resultado está fuera del intervalo, el comportamiento es indefinido.

A.6.4 Tipos flotantes

Cuando un valor flotante menos preciso se convierte en un tipo flotante igual o más preciso, el valor no cambia. Cuando un valor flotante más preciso se convierte en un tipo flotante menos preciso y el valor está dentro del intervalo representable, el resultado puede ser el siguiente valor representable superior o el siguiente valor representable inferior. Si el resultado está fuera del intervalo, el comportamiento es indefinido.

A.6.5 Conversiones aritméticas

Muchos operadores provocan conversiones y producen tipos de resultados de forma similar. El efecto es llevar los operandos a un tipo común, que también es el tipo del resultado. Este patrón se denomina *conversiones aritméticas habituales*.

- En primer lugar, si alguno de los operandos es `long double`, el otro se convierte en `long double`.
- De lo contrario, si alguno de los operandos es `double`, el otro se convierte en `double`.
- De lo contrario, si alguno de los operandos es `float`, el otro se convierte en `float`.
- De lo contrario, las promociones integrales se realizan en ambos operandos; luego, si alguno de los operandos es `unsigned long int`, el otro se convierte en `unsigned long int`.

- De lo contrario, si un operando es `long int` y el otro es `unsigned int`, el efecto depende de si un `long int` puede representar todos los valores de un `unsigned int`; si es así, el `operando int sin signo` se convierte en `long int`; si no, ambos se convierten en `unsigned long int`.
- De lo contrario, si un operando es `long int`, el otro se convierte en `long int`.
- De lo contrario, si alguno de los operandos es `int sin signo`, el otro se convierte en `int sin signo`.
- De lo contrario, ambos operandos tienen el tipo `int`.

Aquí hay dos cambios. En primer lugar, la aritmética en operandos flotantes se puede hacer con precisión simple, en lugar de doble; la primera edición especificaba que toda la aritmética flotante era de doble precisión. En segundo lugar, los tipos sin signo más cortos, cuando se combinan con un tipo con signo más grande, no propagan la propiedad sin signo al tipo de resultado; En la primera edición, siempre dominaron los sin firmar. Las nuevas reglas son un poco más complicadas, pero reducen un poco las sorpresas que pueden ocurrir cuando una cantidad sin firmar se encuentra con la firmada. Es posible que se produzcan resultados inesperados cuando se compara una expresión sin signo con una expresión con signo del mismo tamaño.

A.6.6 Punteros y enteros

Una expresión de tipo entero puede ser añadida o sustraída de un puntero; en tal caso, la expresión integral se convierte como se especifica en la discusión del operador de suma (Par.A.7.7).

Dos punteros a objetos del mismo tipo, en la misma matriz, pueden ser sustraídos; el resultado se convierte en un número entero como se especifica en la discusión del operador de resta (Par.A.7.7).

Una expresión constante entera con valor 0, o una expresión de este tipo convertida al tipo `void *`, se puede convertir, mediante una conversión, por asignación o por comparación, en un puntero de cualquier tipo. Esto genera un puntero nulo que es igual a otro puntero nulo del mismo tipo, pero no igual a cualquier puntero a una función u objeto.

Se permiten otras conversiones que implican punteros, pero tienen aspectos definidos por la implementación. Deben especificarse mediante un operador de conversión de tipos explícito, o convertir (Pars.A.7.5 y A.8.8).

Un puntero se puede convertir en un tipo entero lo suficientemente grande como para contenerlo; El tamaño requerido depende de la implementación. La función de mapeo también depende de la implementación.

Un puntero a un tipo se puede convertir en un puntero a otro tipo. El puntero resultante puede provocar excepciones de direccionamiento si el puntero de asunto no hace referencia a un objeto alineado adecuadamente en el almacenamiento. Se garantiza que un puntero a un objeto se puede convertir en un puntero a un objeto cuyo tipo requiere una alineación de almacenamiento menos o igual de estricta y viceversa sin cambios; La noción de "alineación" depende de la implementación, pero los objetos de los `tipos char` tienen requisitos de alineación menos estrictos. Como se describe en el párrafo A.6.8, un puntero también se puede convertir a tipo `void *` y viceversa sin cambios.

Un puntero se puede convertir en otro puntero cuyo tipo sea el mismo, excepto por la adición o eliminación de calificadores (Pars.A.4.4,A.8.2) del tipo de objeto al que se refiere el puntero. Si se agregan calificadores, el nuevo puntero es equivalente al anterior, excepto por las restricciones implícitas en los nuevos calificadores. Si se quitan los calificadores, las operaciones en el objeto subyacente permanecen sujetas a los calificadores de su declaración real.

Por último, un puntero a una función se puede convertir en un puntero a otro tipo de función. Llamar a la función especificada por el puntero convertido depende de la implementación; Sin embargo, si el puntero convertido se vuelve a convertir a su tipo original, el resultado es idéntico al puntero original.

A.6.7 Vacío

El valor (inexistente) de un objeto `void` no se puede usar de ninguna manera, y no se puede aplicar la conversión explícita ni implícita a ningún tipo no void. Dado que una expresión void denota un valor inexistente, dicha expresión sólo puede utilizarse cuando el valor no es necesario, por ejemplo, como una instrucción de expresión (Par.A.9.2) o como operando izquierdo de un operador de coma (Par.A.7.18).

Una conversión puede convertir una expresión a tipo `void`. Por ejemplo, una conversión void documenta el descarte del valor de una llamada de función utilizada como instrucción de expresión.

> `El vacío` no apareció en la primera edición de este libro, pero se ha vuelto común desde entonces.

A.6.8 Punteros al vacío

Cualquier puntero a un objeto se puede convertir a tipo `void *` sin pérdida de información. Si el resultado se vuelve a convertir al tipo de puntero original, se recupera el puntero original. A diferencia de las conversiones de puntero a puntero discutidas en el párrafo A.6.6, que generalmente requieren una conversión explícita, los punteros se pueden asignar a y desde punteros de tipo `void *` y se pueden comparar con ellos.

> Esta interpretación de los punteros `void *` es nueva; anteriormente, los punteros `char *` desempeñaban el papel de punteros genéricos. El estándar ANSI bendice específicamente el encuentro de `punteros void *` con punteros de objeto en asignaciones y relacionales, al tiempo que requiere conversiones explícitas para otras mezclas de punteros.

A.7 Expresiones

La precedencia de los operadores de expresión es la misma que el orden de las subsecciones principales de esta sección, con la precedencia más alta en primer lugar. Así, por ejemplo, las expresiones denominadas operandos de `+` (Par.A.7.7) son aquellas expresiones definidas en Pars.A.7.1-A.7.6. Dentro de cada subsección, los operadores tienen la misma prioridad. La asociatividad a la izquierda o a la derecha se especifica en cada subsección para los operadores que se analizan en ella. La gramática dada en el párrafo 13 incorporala precedencia y asociatividad de los operadores.

La precedencia y asociatividad de los operadores está completamente especificada, pero el orden de evaluación de las expresiones es, con ciertas excepciones, indefinido, incluso si las subexpresiones implican efectos secundarios. Es decir, a menos que la definición del operador garantice que sus operandos se evalúan en un orden determinado, la implementación es libre de evaluar operandos en cualquier orden, o incluso de intercalar su evaluación. Sin embargo, cada operador combina los valores producidos por sus operandos de una manera compatible con el análisis de la expresión en la que aparece.

> Esta regla revoca la libertad anterior para reordenar expresiones con operadores que son matemáticamente conmutativos y asociativos, pero que pueden fallar en ser computacionalmente asociativos. El cambio afecta solo a los cálculos de punto flotante cerca de los límites de su precisión y a las situaciones en las que es posible un desbordamiento.

El lenguaje no define el control de desbordamiento, comprobación de división y otras excepciones en la evaluación de expresiones. La mayoría de las implementaciones existentes de C omiten el desbordamiento en la evaluación de expresiones y asignaciones integrales con signo, pero este comportamiento no está garantizado. Tratamiento de

la división por 0 y todas las excepciones de punto flotante varía entre las implementaciones; A veces es ajustable por una función de biblioteca no estándar.

A.7.1 Conversión de puntero

Si el tipo de una expresión o subexpresión es "matriz de *T*", para algún tipo *T*, entonces el valor de la expresión es un puntero al primer objeto de la matriz, y el tipo de la expresión se altera a "puntero a *T*". Esta conversión no tiene lugar si la expresión está en el operando del operador `unario &`, o de `++`, `--`, `sizeof`, o como el operando izquierdo de un operador de asignación o el . operador. De manera similar, una expresión de tipo "función que devuelve *T*", excepto cuando se usa como operando del `operador &`, se convierte en "puntero a función que devuelve *T*".

A.7.2 Expresiones primarias

Las expresiones principales son identificadores, constantes, cadenas o expresiones entre paréntesis.

identificador de
expresión primaria
Cadena
constante
(expresión)

Un identificador es una expresión primaria, siempre que se haya declarado adecuadamente como se explica a continuación. Su tipo se especifica mediante su declaración. Un identificador es un lvalue si hace referencia a un objeto (Par.A.5) y si su tipo es aritmético, estructura, unión o puntero.

Una constante es una expresión primaria. Su tipo depende de su forma, como se discutió en el párrafo A.2.5.

Un literal de cadena es una expresión principal. Su tipo es originalmente "array of `char`" (para cadenas de caracteres anchos, "array of `wchar_t`"), pero siguiendo la regla dada en Par.A.7.1, esto generalmente se modifica a "pointer to `char`" (`wchar_t`) y el resultado es un puntero al primer carácter de la cadena. La conversión tampoco se produce en determinados inicializadores; véase el párrafo A.8.7.

Una expresión entre paréntesis es una expresión primaria cuyo tipo y valor son idénticos a los de la expresión sin adornos. La prioridad de los paréntesis no afecta a si la expresión es un lvalue.

A.7.3 Expresiones de sufijo

Los operadores de las expresiones de sufijo se agrupan de izquierda a derecha.

expresión-postfijo:
expresión-primaria
expresión-sufijo[expresión]
expresión-de-sufijo(expresión-argumento-listopt) expresión-de-sufijo.identificador
postfix-expression->identifier postfix-expression++
expresión-de-sufijo--

lista-de-expresión-argumento:
expresión-asignación
lista-de-expresión-asignación , *expresión-asignación*

A.7.3.1 Referencias de matrices

Una expresión de sufijo seguida de una expresión entre corchetes es una expresión de sufijo que denota una referencia a una matriz con subíndice. Una de las dos expresiones debe tener el tipo "puntero a *T*", donde *T* es algún tipo, y la otra debe tener un tipo entero; el tipo de la expresión del subíndice es *T*. La expresión `E1[E2]` es idéntica (por definición) a `*((E1)+(E2))`. Véase el párrafo A.8.6.2 para un análisis más detallado.

A.7.3.2 Llamadas a funciones

Una llamada a una función es una expresión de sufijo, denominada designador de función, seguida de paréntesis que contienen una lista de expresiones de asignación posiblemente vacía y separada por comas (Par.A7.17), que constituyen los argumentos de la función. Si la expresión de sufijo consta de un identificador para el que no existe ninguna declaración en el ámbito actual, el identificador se declara implícitamente como si fuera la declaración

```
extern int identifier();
```

se había dado en el bloque más interno que contenía la llamada a la función. La expresión de sufijo (después de una posible declaración explícita y generación de puntero, Par.A7.1) debe ser de tipo "puntero a la función que devuelve *T*", para algún tipo *T*, y el valor de la llamada a la función tiene el tipo *T*.

> En la primera edición, el tipo estaba restringido a "function", y se requería un operador * explícito para llamar a través de punteros a funciones. El estándar ANSI bendice la práctica de algunos compiladores existentes al permitir la misma sintaxis para las llamadas a funciones y a funciones especificadas por punteros. La sintaxis anterior todavía se puede usar.

El término *argumento* se utiliza para una expresión pasada por una llamada de función; el término *parámetro* se utiliza para un objeto de entrada (o su identificador) recibido por una definición de función, o descrito en una declaración de función. Los términos "argumento real (parámetro)" y "argumento formal (parámetro)" respectivamente se utilizan a veces para la misma distinción.

Al prepararse para la llamada a una función, se hace una copia de cada argumento; Todos los pasos de argumentos son estrictamente por valor. Una función puede cambiar los valores de sus objetos parámetros, que son copias de las expresiones de argumentos, pero estos cambios no pueden afectar a los valores de los argumentos. Sin embargo, es posible pasar un puntero en el entendimiento de que la función puede cambiar el valor del objeto al que apunta el puntero.

Hay dos estilos en los que se pueden declarar funciones. En el nuevo estilo, los tipos de parámetros son explícitos y forman parte del tipo de la función; Dicha declaración también se denomina prototipo de función. En el estilo antiguo, no se especifican los tipos de parámetros. La declaración de función se emite en los párrafos A.8.6.3 y A.10.1.

Si la declaración de función en el ámbito de una llamada es de estilo antiguo, la promoción de argumentos predeterminada se aplica a cada argumento de la siguiente manera: la promoción integral (Par.A.6.1) se realiza en cada argumento de tipo entero y cada argumento `float` se convierte en `double`. El efecto de la llamada es indefinido si el número de argumentos no coincide con el número de parámetros de la definición de la función, o si el tipo de un argumento después de la promoción no coincide con el del parámetro correspondiente. La concordancia de tipo depende de si la definición de la función es

De estilo nuevo o de estilo antiguo. Si es de estilo antiguo, entonces la comparación es entre el tipo promocionado de los argumentos de la llamada, y el tipo promocionado del parámetro, si la definición es de estilo nuevo, el tipo promocionado del argumento debe ser el del parámetro mismo, sin promoción.

Si la declaración de función en el ámbito de una llamada es de estilo nuevo, los argumentos se convierten, como si fuera por asignación, a los tipos de los parámetros correspondientes del prototipo de la función. El número de argumentos debe ser el mismo que el número de parámetros descritos explícitamente, a menos que la lista de parámetros de la declaración termine con la notación de puntos suspensivos `(, ...)`. En ese caso, el número de argumentos debe ser igual o superior al número de parámetros; Los argumentos finales más allá de los parámetros con tipo explícito sufren la promoción de argumentos predeterminada, como se describe en el párrafo anterior. Si la definición de la función es de estilo antiguo, entonces el tipo de cada parámetro en la definición, después de que el tipo del parámetro de definición haya pasado por la promoción de argumentos.

> Estas reglas son especialmente complicadas porque deben atender a una mezcla de funciones de estilo antiguo y nuevo. Las mezclas deben evitarse si es posible.

El orden de evaluación de los argumentos es inespecificado; Tenga en cuenta que los distintos compiladores difieren. Sin embargo, los argumentos y el designador de función se evalúan completamente, incluidos todos los efectos secundarios, antes de introducir la función. Se permiten llamadas recursivas a cualquier función.

A.7.3.3 Referencias de estructura

Una expresión de sufijo seguida de un punto seguido de un identificador es una expresión de sufijo. La primera expresión de operando debe ser una estructura o una unión, y el identificador debe nombrar un miembro de la estructura o unión. El valor es el miembro con nombre de la estructura o unión, y su tipo es el tipo del miembro. La expresión es un lvalue si la primera expresión es un lvalue y si el tipo de la segunda expresión no es un tipo de matriz.

Una expresión de sufijo seguida de una flecha (creada a partir de `-` y `>`) seguida de un identificador es una expresión de sufijo. La primera expresión de operando debe ser un puntero a una estructura o unión, y el identificador debe nombrar a un miembro de la estructura o unión. El resultado hace referencia al miembro con nombre de la estructura o unión a la que apunta la expresión de puntero, y el tipo es el tipo del miembro; El resultado es un Lvalue si el tipo no es un tipo de matriz.

Por lo tanto, la expresión `E1->MOS` es la misma que `(*E1).MOS`. Las estructuras y uniones se discuten en el párrafo A.8.3.

> En la primera edición de este libro, ya era la regla que un nombre de miembro en una expresión de este tipo tenía que pertenecer a la estructura o unión mencionada en la expresión de sufijo; Sin embargo, una nota admitió que esta regla no se aplicó firmemente. Los compiladores recientes, y ANSI, lo aplican.

A.7.3.4 Incremento de sufijo

Una expresión de sufijo seguida de un `operador ++` o `--` es una expresión de sufijo. El valor de la expresión es el valor del operando. Una vez anotado el valor, el operando se incrementa `++` o se disminuye `--` en 1. El operando debe ser un lvalue; consulte la discusión de los operadores aditivos (Par.A.7.7) y la asignación (Par.A.7.17) para obtener más restricciones sobre el operando y detalles de la operación. El resultado no es un lvalue.

A.7.4 Operadores unarios

Las expresiones con operadores unarios se agrupan de derecha a izquierda.

expresión-unaria:
Expresión de sufijo
`++`*expresión unaria*
`--`*expresión unaria*
operador unario cast-expression
`sizeof` *unary-expression*
`sizeof`(*nombre-tipo*)

operador unario: uno de los
`& * + - ~ !`

A.7.4.1 Operadores de incremento de prefijos

Una expresión unaria seguida de un `operador ++` o `--` es una expresión unaria. El operando se incrementa `++` o se disminuye `--` en 1. El valor de la expresión es el valor después del incremento (decrementación). El operando debe ser un lvalue; consulte la discusión de los operadores aditivos (Par.A.7.7) y la asignación (Par.A.7.17) para obtener más restricciones sobre los operandos y detalles de la operación. El resultado no es un lvalue.

A.7.4.2 Operador de dirección

El operador unario `&` toma la dirección de su operando. El operando debe ser un valor l que no haga referencia ni a un campo de bits ni a un objeto declarado como `registro`, o debe ser de tipo función. El resultado es un puntero al objeto o función al que hace referencia el lvalue. Si el tipo del operando es *T*, el tipo del resultado es "puntero a *T*".

A.7.4.3 Operador de direccionamiento indirecto

El operador `unario *` denota direccionamiento indirecto y devuelve el objeto o la función a la que apunta su operando. Es un lvalue si el operando es un puntero a un objeto de tipo aritmético, estructura, unión o puntero. Si el tipo de la expresión es "puntero a *T*", el tipo del resultado es *T*.

A.7.4.4 Operador Unary Plus

El operando del operador `unario +` debe tener tipo aritmético y el resultado es el valor del operando. Un operando integral se somete a una promoción integral. El tipo del resultado es el tipo del operando promocionado.

El unario `+` es nuevo con el estándar ANSI. Se añadió para la simetría con el unario `-`.

A.7.4.5 Operador menos unario

El operando del operador `unario -` debe tener tipo aritmético y el resultado es el negativo de su operando. Un operando integral se somete a una promoción integral. El negativo de una cantidad sin signo se calcula restando el valor promocionado del valor más grande del tipo promocionado y sumando uno; Pero menos cero es cero. El tipo del resultado es el tipo del operando promocionado.

A.7.4.6 Operador de complemento

El operando del operador `~` debe tener tipo entero, y el resultado es el complemento uno de su operando. Se realizan las promociones integrales. Si el operando no tiene signo, el resultado se calcula restando el valor del valor más grande del tipo promocionado. Si el operando está firmado, el resultado se calcula convirtiendo el operando promocionado en el tipo sin signo correspondiente, aplicando `~` y volviendo a convertirlo en el tipo con signo. El tipo del resultado es el tipo del operando promocionado.

A.7.4.7 Operador de negación lógica

El operando de la clase `!` El operador debe tener un tipo aritmético o ser un puntero, y el resultado es 1 si el valor de su operando es igual a 0 y 0 en caso contrario. El tipo del resultado es `int`.

A.7.4.8 Tamañodel operador

El operador `sizeof` produce el número de bytes necesarios para almacenar un objeto del tipo de su operando. El operando es una expresión, que no se evalúa, o un nombre de tipo entre paréntesis. Cuando `sizeof` se aplica a un `char`, el resultado es 1; cuando se aplica a una matriz, el resultado es el número total de bytes en la matriz. Cuando se aplica a una estructura o unión, el resultado es el número de bytes del objeto, incluido el relleno necesario para hacer que el mosaico del objeto sea una matriz: el tamaño de una matriz de *n* elementos es *n* veces el tamaño de un elemento. El operador no se puede aplicar a un operando de tipo función, o de tipo incompleto, o a un campo de bits. El resultado es una constante integral sin signo; El tipo concreto está definido por la implementación. El encabezado estándar `<stddef.h>` (véase el apéndice B) define este tipo como `size_t`.

A.7.5 Moldes

Una expresión unaria precedida por el nombre entre paréntesis de un tipo provoca la conversión del valor de la expresión al tipo con nombre.

expresión-cast:
Expresión unaria
(nombre-tipo) expresión-conversión

A esta construcción se le llama *yeso*. Los nombres se describen en el párrafo A.8.8. Los efectos de las conversiones se describen en Par.A.6. Una expresión con una conversión no es un lvalue.

A.7.6 Operadores multiplicativos

Los operadores multiplicativos `*`, `/` y `%` se agrupan de izquierda a derecha.

expresión-multiplicativa:
expresión-multiplicativa `*` *expresión-casta*
expresión-multiplicativa `/` *expresión-casta*
expresión-multiplicativa `%` *expresión-casta*

Los operandos de `*` y `/` deben tener tipo aritmético; los operandos de `%` deben tener tipo entero. Las conversiones aritméticas habituales se realizan en los operandos y predicen el tipo de resultado.

El operador `binario *` denota multiplicación.

El operador binario / produce el cociente, y el operador % el resto, de la división del primer operando por el segundo; si el segundo operando es 0, el resultado es indefinido. De lo contrario, siempre es cierto que (a/b)*b + a%b es igual a a. Si ambos operandos no son negativos, entonces el resto es no negativo y menor que el divisor, si no, solo se garantiza que el valor absoluto del resto es menor que el valor absoluto del divisor.

A.7.7 Operadores aditivos

Los operadores aditivos + y – se agrupan de izquierda a derecha. Si los operandos tienen tipo aritmético, se realizan las conversiones aritméticas habituales. Hay algunas posibilidades de tipo adicionales para cada operador.

expresión-aditiva:
expresión-multiplicativa
expresión-aditiva + *expresión-multiplicativa*
expresión-aditiva – *expresión-multiplicativa*

El resultado del operador + es la suma de los operandos. Se puede agregar un puntero a un objeto en una matriz y un valor de cualquier tipo entero. Este último se convierte en un desplazamiento de dirección multiplicándolo por el tamaño del objeto al que apunta el puntero. La suma es un puntero del mismo tipo que el puntero original y apunta a otro objeto de la misma matriz, desplazado adecuadamente del objeto original. Por lo tanto, si P es un puntero a un objeto en una matriz, la expresión P+1 es un puntero al siguiente objeto en la matriz. Si el puntero de suma apunta fuera de los límites de la matriz, excepto en la primera ubicación más allá del extremo superior, el resultado es indefinido.

> La provisión de punteros justo más allá del final de una matriz es nueva. Legitima un modismo común para recorrer en bucle los elementos de una matriz.

El resultado del operador – es la diferencia de los operandos. Un valor de cualquier tipo entero se puede restar de un puntero y, a continuación, se aplican las mismas conversiones y condiciones que para la suma.

Si se restan dos punteros a objetos del mismo tipo, el resultado es un valor entero con signo que representa el desplazamiento entre los objetos a los que apunta; los punteros a objetos sucesivos difieren en 1. El tipo de resultado se define como ptrdiff_t en la cabecera estándar
<stddef.h>. El valor no está definido a menos que los punteros apunten a objetos dentro de la misma matriz; sin embargo, si P apunta al último miembro de una matriz, entonces (P+1)–P tiene el valor 1.

A.7.8 Operadores de turno

Los operadores de turno << y >> agrupan de izquierda a derecha. Para ambos operadores, cada operando debe ser integral, y está sujeto a las promociones integrales. El tipo del resultado es el del operando izquierdo promocionado. El resultado es indefinido si el operando derecho es negativo, o mayor o igual que el número de bits del tipo de la expresión izquierda.

expresión-desplazamiento:
expresión-aditiva
shift-expression << *expresión-aditiva*
expresión-shift-expression >>
expresión-aditiva

El valor de E1<<E2 es E1 (interpretado como un patrón de bits) bits E2 desplazados a la izquierda; en ausencia de desbordamiento, esto es equivalente a la multiplicación por 2E2. El

valor de `E1>>E2` es `E1` desplazado a la derecha

`Posiciones de bits E2`. El desplazamiento a la derecha es equivalente a la división por 2E2 si `E1` no tiene signo o tiene un valor no negativo; de lo contrario, el resultado está definido por la implementación.

A.7.9 Operadores relacionales

Los operadores relacionales se agrupan de izquierda a derecha, pero este hecho no es útil; `a<b<c` se analiza como
`(a<b)<c`, y se evalúa como 0 o 1.

expresión-relacional:
expresión-desplazamiento
expresión-relacional `<` *expresión-desplazamiento expresión-relacional* `>`
expresión-desplazamiento-expresión-relacional `<=` *expresión-desplazamiento-expresión-relacional* `>=`
expresión-desplazamiento

Los operadores `<` (menor), `>` (mayor), `<=` (menor o igual) y `>=` (mayor o igual) producen 0 si la relación especificada es falsa y 1 si es verdadera. El tipo del resultado es `int`. Las conversiones aritméticas habituales se realizan en operandos aritméticos. Se pueden comparar punteros a objetos del mismo tipo (omitiendo cualquier calificador); El resultado depende de las ubicaciones relativas en el espacio de direcciones de los objetos a los que se apunta. La comparación de punteros solo se define para partes del mismo objeto; Si dos punteros apuntan al mismo objeto simple, se comparan igual; Si los punteros son a miembros de la misma estructura, los punteros a objetos declarados más adelante en la estructura se comparan más arriba; Si los punteros se refieren a miembros de una matriz, la comparación es equivalente a la comparación de los subíndices correspondientes. Si `P` apunta al último miembro de una matriz, entonces `P+1` se compara más alto que `P`, aunque `P+1` apunte fuera de la matriz. De lo contrario, la comparación de punteros no está definida.

> Estas reglas liberalizan ligeramente las restricciones establecidas en la primera edición, al permitir la comparación de punteros con diferentes miembros de una estructura o unión. También legalizan la comparación con un puntero justo al final de una matriz.

A.7.10 Operadores de igualdad

igualdad-expresión:
expresión-relacional
expresión-igualdad `==` *expresión-relacional*
expresión-igualdad `!=` *expresión-relacional*

Los operadores `==` (igual a) y `!=` (no igual a) son análogos a los operadores relacionales, excepto por su precedencia inferior. (Por lo tanto, `a<b == c<d` es 1 siempre que `a<b` y `c<d` tengan el mismo valor de verdad.)

Los operadores de igualdad siguen las mismas reglas que los operadores relacionales, pero permiten posibilidades adicionales: un puntero puede compararse con una expresión integral constante con valor 0, o con un puntero a `void`. Véase el párrafo A.6.6.

A.7.11 Operador AND bit a bit

Expresión AND:
igualdad-expresión
Y-expresión `e` *igualdad-expresión*

Se realizan las conversiones aritméticas habituales; el resultado es la función AND bit a bit de los operandos. El operador solo se aplica a los operandos enteros.

A.7.12 Operador OR exclusivo de Bitwise

expresión-OR exclusiva:
Expresión AND
expresión-OR exclusiva `^` *Expresión-AND*

Se realizan las conversiones aritméticas habituales; el resultado es la función OR exclusiva bit a bit de los operandos. El operador solo se aplica a los operandos enteros.

A.7.13 Operador OR inclusivo bit a bit

expresión-OR inclusiva:
expresión-OR exclusiva
expresión-OR-inclusiva `|` *expresión-OR exclusiva*

Se realizan las conversiones aritméticas habituales; el resultado es la función OR inclusiva bit a bit de los operandos. El operador solo se aplica a los operandos enteros.

A.7.14 Operador lógico AND

expresión-AND-lógica:
expresión-OR inclusiva
expresión-AND-lógica `&&` *expresión-OR-inclusiva*

El `operador &&` se agrupa de izquierda a derecha. Devuelve 1 si sus dos operandos se comparan de manera desigual con cero, 0 en caso contrario. A diferencia `de &`, `&&` garantiza la evaluación de izquierda a derecha: se evalúa el primer operando, incluidos todos los efectos secundarios; si es igual a 0, el valor de la expresión es 0. De lo contrario, se evalúa el operando derecho y, si es igual a 0, el valor de la expresión es 0, de lo contrario, 1.

No es necesario que los operandos tengan el mismo tipo, pero cada uno debe tener un tipo aritmético o ser un puntero. El resultado es `int`.

A.7.15 Operador OR lógico

expresión-OR-lógica:
expresión-lógica AND
expresión-OR lógica `||` *expresión-lógica AND*

El `||` Grupos de operadores de izquierda a derecha. Devuelve 1 si alguno de sus operandos se compara desigual con cero y 0 en caso contrario. A diferencia de `|`, `||` Garantiza la evaluación de izquierda a derecha: se evalúa el primer operando, incluidos todos los efectos secundarios; si es desigual a 0, el valor de la expresión es 1. De lo contrario, se evalúa el operando derecho y, si no es igual que 0, el valor de la expresión es 1, de lo contrario, 0.

No es necesario que los operandos tengan el mismo tipo, pero cada uno debe tener un tipo aritmético o ser un puntero. El resultado es `int`.

A.7.16 Operador condicional

expresión-condicional:
expresión-o-lógica
expresión-lógica-OR `?` *expresión* `:` *expresión-condicional*

Se evalúa la primera expresión, incluyendo todos los efectos secundarios; si se compara desigual a 0, el resultado es el valor de la segunda expresión, de lo contrario, el de la tercera expresión. Solo se evalúa uno de los operandos segundo y tercero. Si el segundo y el tercer operando son aritméticos, se realizan las conversiones aritméticas habituales para llevarlos a un tipo común, y ese tipo es el tipo del resultado. Si ambos son `void`, o estructuras o uniones del mismo tipo, o punteros a objetos del mismo tipo, el resultado tiene el tipo común. Si uno es un puntero y el otro la constante 0, el 0 se convierte en el tipo de puntero y el resultado tiene ese tipo. Si uno es un puntero a `void` y el otro es otro puntero, el otro puntero se convierte en un puntero a `void`, y ese es el tipo de resultado.

En la comparación de tipos para punteros, cualquier calificador de tipo (Par.A.8.2) en el tipo al que apunta el puntero es insignificante, pero el tipo de resultado hereda calificadores de ambos brazos del condicional.

A.7.17 Expresiones de asignación

Hay varios operadores de asignación; todos agrupados de derecha a izquierda.

expresión-asignación:
expresión-condicional
expresión-unaria asignación-operador de asignación-expresión

asignado-operador: uno de los
`= *= /= %= += -= <<= >>= &= ^= |=`

Todos requieren un lvalue como operando izquierdo, y el lvalue debe ser modificable: no debe ser una matriz, y no debe tener un tipo incompleto, ni ser una función. Además, su tipo no debe estar calificado con `const`; si es una estructura o unión, no debe tener ningún miembro o, recursivamente, submiembro calificado con `const`. El tipo de una expresión de asignación es el de su operando izquierdo, y el valor es el valor almacenado en el operando izquierdo después de que se haya realizado la asignación.

En la asignación simple con `=`, el valor de la expresión reemplaza al del objeto al que hace referencia el lvalue. Debe cumplirse una de las siguientes condiciones: ambos operandos tienen tipo aritmético, en cuyo caso el operando derecho se convierte en el tipo de la izquierda mediante la asignación; o ambos operandos son estructuras o uniones del mismo tipo; o un operando es un puntero y el otro es un puntero a `void`, o el operando izquierdo es un puntero y el operando derecho es una expresión constante con valor 0; o ambos operandos son punteros a funciones u objetos cuyos tipos son los mismos, excepto por la posible ausencia de `const` o `volatile` en el operando derecho.

Una expresión de la forma `E1` *`op= E2`* es equivalente a `E1 = E1` *`op`* `(E2)`, excepto que `E1` se evalúa solo una vez.

A.7.18 Operador de coma

Expresión:
expresión-asignación
expresión `,` *asignación-expresión*

Un par de expresiones separadas por una coma se evalúa de izquierda a derecha y se descarta el valor de la expresión izquierda. El tipo y el valor del resultado son el tipo y el valor del operando derecho. Todos los efectos secundarios de la evaluación del operando izquierdo se completan antes de comenzar la evaluación del operando derecho. En contextos en los que se da un significado especial a la coma, por ejemplo, en listas de argumentos de función (Par.A.7.3.2) y listas de inicializadores (Par.A.8.7), la unidad sintáctica requerida es una expresión de asignación, por lo que el operador de coma aparece solo en una agrupación entre paréntesis, por ejemplo,

```
f(a, (t=3, t+2), c)
```

tiene tres argumentos, el segundo de los cuales tiene el valor 5.

A.7.19 Expresiones constantes

Desde el punto de vista sintáctico, una expresión constante es una expresión restringida a un subconjunto de operadores:

expresión-constante:
expresión-condicional

Las expresiones que se evalúan como una constante son necesarias en varios contextos: después de mayúsculas y minúsculas, como límites de matriz y longitudes de campo de bits, como valor de una constante de enumeración, en inicializadores y en determinadas expresiones de preprocesador.

Las expresiones constantes no pueden contener asignaciones, operadores de incremento o decremento, llamadas a funciones u operadores de coma, excepto en un operando de `sizeof`. Si se requiere que la expresión constante sea integral, sus operandos deben constar de constantes enteras, de enumeración, de caracteres y flotantes; Las conversiones deben especificar un tipo entero y las constantes flotantes deben convertirse en enteros. Esto descarta necesariamente las operaciones de matrices, direccionamiento indirecto, dirección de y miembro de estructura. (Sin embargo, cualquier operando está permitido para `tamañode`.)

Se permite más latitud para las expresiones constantes de los inicializadores; los operandos pueden ser cualquier tipo de constante, y el `operador unario &` se puede aplicar a objetos externos o estáticos, y a matrices externas y estáticas con subíndices con una expresión constante. El operador `unario &` también se puede aplicar implícitamente mediante la aparición de matrices y funciones sin subíndice. Los inicializadores deben evaluarse como una constante o como la dirección de un objeto externo o estático declarado previamente más o menos una constante.

Se permite menos latitud para las expresiones constantes integrales después de `#if`; `No` se permiten las expresiones sizeof, las constantes de enumeración ni las conversiones. Véase el párrafo A.12.5.

A.8 Declaraciones

Las declaraciones especifican la interpretación que se da a cada identificador; no necesariamente reservan el almacenamiento asociado con el identificador. Las declaraciones que reservan el almacenamiento se denominan *definiciones*.
Las declaraciones tienen la forma

Declaración:
especificadores-de-declaración init-declarator-listopt;

Los declaradores de la lista init-declarator contienen los identificadores que se declaran; Los

especificadores de declaración constan de una secuencia de especificadores de tipo y clase de almacenamiento.

especificadores-declaración:
especificador-de-clase-de-almacenamiento
declaracion-especificador-sopt tipo-
especificador declaracion-especificadoropt
type-qualifier declaration-specifiersopt

lista-de-declarante-de-inicio:
declarador-de-inicio
lista-de-declarador-de-inicio , *declarador-de-inicio*

declarador de inicio:
Declarador
declarador = *inicializador*

Los declarantes se discutirán más adelante (Par.A.8.5); contienen los nombres que se declaran. Una declaración debe tener al menos un declarador, o su especificador de tipo debe declarar una etiqueta de estructura, una etiqueta de unión o los miembros de una enumeración; No se permiten declaraciones vacías.

A.8.1 Especificadores de clase de almacenamiento

Los especificadores de clase de almacenamiento son:

Especificador de clase de almacenamiento:
`registro`
`automáti`
`co de`
`tipo`
`externo`
`estático`

El significado de las clases de almacenamiento se discutió en el párrafo A.4.4.

Los especificadores `auto` y `register` proporcionan a los objetos declarados una clase de almacenamiento automático y solo se pueden utilizar dentro de las funciones. Estas declaraciones también sirven como definiciones y hacen que se reserve el almacenamiento. Una `declaración de registro` es equivalente a una declaración `automática`, pero sugiere que se tendrá acceso frecuente a los objetos declarados. Solo unos pocos objetos se colocan realmente en los registros, y solo ciertos tipos son elegibles; Las restricciones dependen de la implementación. Sin embargo, si un objeto se declara `register`, el operador `unario &` no se puede aplicar a él, explícita o implícitamente.

> La regla de que es ilegal calcular la dirección de un objeto declarado `registrado`, pero que en realidad se considera `automático`, es nueva.

El especificador static proporciona a los objetos declarados una clase de almacenamiento estático y se puede utilizar dentro o fuera de las funciones. Dentro de una función, este especificador hace que se asigne almacenamiento y sirve como definición; para su efecto fuera de una función, véase el párrafo A.11.2.

Una declaración con `extern`, usada dentro de una función, especifica que el almacenamiento de los objetos declarados se define en otro lugar; para sus efectos fuera de una función, véase Par.A.11.2.

El especificador `typedef` no reserva almacenamiento y se denomina especificador de clase de almacenamiento solo por conveniencia sintáctica; se discute en Par.A.8.9.

A lo sumo, se puede proporcionar un especificador de clase de almacenamiento en una declaración. Si no se da ninguno, se usan estas reglas: los objetos declarados dentro de una función se toman como `auto`; las funciones declaradas dentro de una función se toman como `extern`; los objetos y funciones declarados fuera de una función se toman como `estáticos`, con enlace externo. Véase Pars. A.10-A.11.

A.8.2 Especificadores de tipo

Los especificadores de tipo son

Especificador de tipo:
`Void`
`Char`
`Corto`
`Int`
`Largo`
`Flotad`
`or`
`Doble`
`Firmad`
`o`
`Unsigned`
struct-or-union-specifier
enum-specifier
nombre-de-tip-def

A lo sumo, una de las palabras `long` o `short` puede especificarse junto con `int`; el significado es el mismo si no se menciona int. La palabra `long` puede especificarse junto con `double`. A lo sumo, se puede especificar uno de `signat` o `unsigned` junto con `int` o cualquiera de sus `variedades cortas` o `largas`, o con `char`. Cualquiera de los dos puede aparecer solo, en cuyo caso se entiende int. El especificador signed es útil para forzar a los objetos char a llevar un signo; es permisible pero redundante con otros tipos enteros.

De lo contrario, como máximo se puede proporcionar un especificador de tipo en una declaración. Si falta el especificador de tipo en una declaración, se considera que es `int`.

Los tipos también se pueden calificar para indicar propiedades especiales de los objetos que se declaran.

calificador de tipo:
`Const`
`volátil`

Los calificadores de tipo pueden aparecer con cualquier especificador de tipo. Un `objeto const` puede inicializarse, pero no asignarse posteriormente. No hay ninguna semántica dependiente de la implementación para los objetos volátiles.

> Las propiedades `const` y `volatile` son nuevas con el estándar ANSI. El propósito de `const` es anunciar objetos que se pueden colocar en la memoria de solo lectura y, tal vez, aumentar las oportunidades de optimización. El propósito de `volatile` es forzar una implementación para suprimir la optimización que, de otro modo, podría producirse. Por ejemplo, para una máquina con entrada/salida asignada a memoria, un puntero a un registro de dispositivo podría declararse como un puntero a `volatile`, para evitar que el compilador quite referencias aparentemente redundantes a través del puntero. Excepto que debe diagnosticar intentos explícitos de cambiar objetos `const`, un compilador puede omitir estos calificadores.

A.8.3 Estructura y Declaraciones Sindicales

Una estructura es un objeto que consta de una secuencia de miembros con nombre de varios tipos. Una unión es un objeto que contiene, en diferentes momentos, cualquiera de varios miembros de varios tipos. Los especificadores de estructura y unión tienen el mismo formato.

struct-or-union-especificador:
struct-or-union identifieropt `{` *struct-declaration-list* `}`
Identificador de estructura o unión

estructura-o-unión:
`Unión`
`de`
`estruc`
`turas`

Una lista-declaración-estructura es una secuencia de declaraciones para los miembros de la estructura o unión:

lista-de-declaración-estructura:
Declaración de estructura
struct-declaration-list declaración de estructura

declaración-estructura: *specifier-qualifier-list struct-declarator-*

list`;` *lista-calificador-especificador*:
especificador-de-tipo especificador-
calificador-listaopt calificador de
tipo especificador-calificador-listaopt

lista-declarante-destructiva:
Declarador de estructura
lista-de-declarador-estructura `,` *declarador-estructura*

Por lo general, un declarador de estructura es simplemente un declarador para un miembro de una estructura o unión. Un miembro de estructura también puede constar de un número especificado de bits. Un miembro de este tipo también se denomina *campo de bits*; su longitud se separa del declarador del nombre del campo mediante dos puntos.

struct-declarator:
Declarador declaradoreight `:` *expresión-constante*

Un especificador de tipo de la forma

identificador de estructura o unión `{` *lista-de-declaración-estructura* `}`

Declara que el identificador es la *etiqueta* de la estructura o unión especificada por la lista. Una declaración posterior en el mismo ámbito o en uno interno puede hacer referencia al mismo tipo mediante la etiqueta en un especificador sin la lista:

Identificador de estructura o unión

Si aparece un especificador con una etiqueta pero sin una lista cuando la etiqueta no está declarada, *se especifica un tipo incompleto*. Los objetos con una estructura o tipo de unión incompletos se pueden mencionar en contextos en los que su tamaño no es necesario, por ejemplo, en declaraciones (no definiciones), para especificar un puntero o para crear una definición de `tipo`, pero no de otro modo. El tipo se completa cuando aparece un especificador posterior con esa etiqueta y contiene una lista de declaraciones. Aun

En los especificadores con una lista, la estructura o el tipo de unión que se declara está incompleto dentro de la lista y solo se completa cuando } termina el especificador.

Una estructura no puede contener un miembro de tipo incompleto. Por lo tanto, es imposible declarar una estructura o unión que contenga una instancia de sí misma. Sin embargo, además de dar un nombre a la estructura o tipo de unión, las etiquetas permiten la definición de estructuras autorreferenciales; Una estructura o unión puede contener un puntero a una instancia de sí misma, ya que se pueden declarar punteros a tipos incompletos.

Una regla muy especial se aplica a las declaraciones de la forma

identificador de estructura o unión;

que declaran una estructura o unión, pero no tienen lista de declaraciones ni declarantes. Incluso si el identificador es una etiqueta de estructura o unión ya declarada en un ámbito externo (Par.A.11.1), esta declaración convierte al identificador en la etiqueta de una nueva estructura o unión con tipo incompleto en el ámbito actual.

> Este recóndito es nuevo con ANSI. Está pensado para tratar con estructuras mutuamente recursivas declaradas en un ámbito interno, pero cuyas etiquetas ya podrían estar declaradas en el ámbito externo.

Un especificador de estructura o unión con una lista pero sin etiqueta crea un tipo único; sólo se puede hacer referencia directa a ella en la declaración de la que forma parte.

Los nombres de los miembros y las etiquetas no entran en conflicto entre sí ni con las variables ordinarias. Un nombre de miembro no puede aparecer dos veces en la misma estructura o unión, pero el mismo nombre de miembro se puede utilizar en diferentes estructuras o uniones.

> En la primera edición de este libro, los nombres de los miembros de la estructura y del sindicato no estaban asociados con su padre. Sin embargo, esta asociación se hizo común en los compiladores mucho antes del estándar ANSI.

Un miembro que no es de campo de una estructura o unión puede tener cualquier tipo de objeto. Un miembro de campo (que no necesita tener un declarador y, por lo tanto, puede no tener nombre) tiene el tipo `int`, `unsigned int` o `signed int`, y se interpreta como un objeto de tipo entero de la longitud especificada en bits; el hecho de que un `campo int` se trate como firmado depende de la implementación. Los miembros de campo adyacentes de las estructuras se empaquetan en unidades de almacenamiento dependientes de la implementación en una dirección dependiente de la implementación. Cuando un campo que sigue a otro campo no cabe en una unidad de almacenamiento parcialmente llena, puede dividirse entre unidades o la unidad puede rellenarse. Un campo sin nombre con ancho 0 fuerza este relleno, de modo que el siguiente campo comenzará en el borde de la siguiente unidad de asignación.

> El estándar ANSI hace que los campos dependan aún más de la implementación que en la primera edición. Es aconsejable leer las reglas del lenguaje para almacenar campos de bits como "dependientes de la implementación" sin calificación. Las estructuras con campos de bits se pueden utilizar como una forma portátil de intentar reducir el almacenamiento requerido para una estructura (con el costo probable de aumentar el espacio de instrucción y el tiempo necesarios para acceder a los campos), o como una forma no portátil de describir un diseño de almacenamiento conocido en el nivel de bits. En el segundo caso, es necesario comprender las reglas de la implementación local.

Los miembros de una estructura tienen direcciones crecientes en el orden de sus declaraciones. Un miembro que no es de campo de una estructura se alinea en un límite de direccionamiento en función de su tipo; Por lo tanto, puede haber agujeros sin nombre en una estructura. Si un puntero a una estructura se convierte al tipo de un puntero a su primer miembro, el resultado hace referencia al primer miembro.

Una unión puede considerarse como una estructura cuyos miembros comienzan en el desplazamiento 0 y cuyo tamaño es suficiente para contener cualquiera de sus miembros. A lo sumo, uno de los miembros puede almacenarse en una unión en cualquier momento. Si un puntero a una unión se convierte al tipo de un puntero a un miembro, el resultado hace referencia a ese miembro.

Un ejemplo sencillo de una declaración de estructura es

```
struct tnode {
    char tword[20];
    int count;
    struct tnode
    *izquierda; struct
    tnode *right;
}
```

que contiene una matriz de 20 caracteres, un entero y dos punteros a estructuras similares. Una vez que se ha dado esta declaración, la declaración

```
struct tnode s, *sp;
```

declara `que s` es una estructura del tipo dado y `sp` que es un puntero a una estructura del tipo dado. Con estas declaraciones, la expresión

```
sp->count
```

se refiere al campo de recuento de la estructura a la que apunta `sp` ;

```
s.izquierda
```

se refiere al puntero izquierdo del subárbol de la estructura `s`, y

```
s.right->tword[0]
```

Se refiere al primer carácter del miembro `tword` del subárbol derecho de `S`.

En general, un miembro de un sindicato no puede ser inspeccionado a menos que el valor de la unión se haya asignado utilizando el mismo miembro. Sin embargo, una garantía especial simplifica el uso de las uniones: si una unión contiene varias estructuras que comparten una secuencia inicial común, y la unión contiene actualmente una de estas estructuras, se permite hacer referencia a la parte inicial común de cualquiera de las estructuras contenidas. Por ejemplo, el siguiente es un fragmento legal:

```
unión {
    struct {
        tipo int;
    } n;
    struct {
        tipo int;
        int intnode;
    } ni;
    struct {
        tipo int;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOTADOR;
u.nf.floatnode = 3.14;
...
if (tipo u.n. == FLOTADOR)
    ... sin(u.nf.floatnode) ...
```

A.8.4 Enumeraciones

Las enumeraciones son tipos únicos con valores que abarcan un conjunto de constantes con nombre denominadas enumeradores. La forma de un especificador de enumeración toma prestada de la de estructuras y uniones.

especificador de enumeración:
`enum` *identifieropt* `{` *enumerator-list* `}` `enum` *identifier*

lista-enumeradora:
Enumerador
lista-enumeradora `,` *enumerador*

Enumerador:
identificador
identificador `=` *expresión-constante*

Los identificadores de una lista de enumeradores se declaran como constantes de tipo `int` y pueden aparecer siempre que se requieran constantes. Si no aparece ninguna enumeración con `=`, los valores de las constantes correspondientes comienzan en 0 y aumentan en 1 a medida que la declaración se lee de izquierda a derecha. Un enumerador con `=` proporciona al identificador asociado el valor especificado; los identificadores posteriores continúan la progresión desde el valor asignado.

Los nombres de enumerador del mismo ámbito deben ser distintos entre sí y de los nombres de variables ordinarias, pero no es necesario que los valores sean distintos.

La función del identificador en el especificador de enumeración es análoga a la de la etiqueta de estructura en un especificador de estructura; Nombra una enumeración determinada. Las reglas para los especificadores de enumeración con y sin etiquetas y listas son las mismas que las de los especificadores de estructura o unión, excepto que no existen tipos de enumeración incompletos; La etiqueta de un especificador de enumeración sin una lista de enumeradores debe hacer referencia a un especificador dentro del ámbito con una lista.

> Las enumeraciones son nuevas desde la primera edición de este libro, pero han sido parte del lenguaje desde hace algunos años.

A.8.5 Declarantes

Los declaradores tienen la sintaxis:

Declarante:
Pointeropt de declaración directa

declarante directo:
identificador
`(`*declarante*`)`
declarador-directo `[` *expresión-constante:opt* `]` *declarador-directo* `(` *lista-de-tipos-de-parámetros* `)`
declarador-directo `(` *identificador-listaopt* `)`

Puntero:
`*` *tipo-calificador-listaopt*
`*` *puntero type-qualifier-listopt*

lista-calificadora-tipo:
calificador de tipo
type-qualifier-list type-qualifier

La estructura de los declaradores se asemeja a la de las expresiones de direccionamiento indirecto, función y matriz; La agrupación es la misma.

A.8.6 Significado de los declarantes

Aparece una lista de declaradores después de una secuencia de especificadores de tipo y clase de almacenamiento. Cada declarador declara un identificador principal único, el que aparece como la primera alternativa de la producción para el *declarador directo*. Los especificadores de clase de almacenamiento se aplican directamente a este identificador, pero su tipo depende de la forma de su declarador. Un declarador se lee como una afirmación que cuando su identificador aparece en una expresión de la misma forma que el declarador, produce un objeto del tipo especificado.

Teniendo en cuenta solo las partes de tipo de los especificadores de declaración (párrafo A.8.2) y un declarador en particular, una declaración tiene la forma "T D", donde `T` es un tipo y `D` es un declarador. El tipo atribuido al identificador en las distintas formas de declarador se describe inductivamente utilizando esta notación.

En una declaración `T D` donde `D` es un identificador no adorado, el tipo del identificador es `T`.

En una declaración `T D`, donde `D` tiene la forma

```
( D1 )
```

entonces el tipo del identificador en `D1` es el mismo que el de `D`. Los paréntesis no alteran el tipo, pero pueden cambiar el enlace de declaradores complejos.

A.8.6.1 Declaradores de puntero

En una declaración `T D`, donde `D` tiene la forma

`*` *type-qualifier-listopt* `D1`

y el tipo del identificador en la declaración `T D1` es "modificador de tipo `T`", el tipo del identificador de `D` es "tip-modifier type-qualifier-list puntero a `T`". Los calificadores que siguen a `*` se aplican al propio puntero, en lugar de al objeto al que apunta el puntero.

Por ejemplo, considere la declaración

```
int *ap[];
```

Aquí, `ap[]` desempeña el papel de `D1`; una declaración "int `ap[]`" (abajo) le daría a `ap` el tipo "array of int", la lista de calificadores de tipo está vacía y el modificador de tipo es "array of". Por lo tanto, la declaración real le da `a ap` el tipo "array a punteros a `int`".

Como otros ejemplos, las declaraciones

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

Declare un entero `i` y un puntero a un entero `pi`. El valor del puntero constante `cpi` no se puede cambiar, siempre apuntará a la misma ubicación, aunque el valor al que se refiere puede estar alterado. El entero `ci` es constante y no se puede cambiar (aunque se puede inicializar, como aquí). El tipo de `pci` es "puntero a `const int`", y `pci` en sí mismo puede cambiarse para apuntar a otro lugar, pero el valor al que apunta no puede ser alterado asignándolo a través de `pci`.

A.8.6.2 Declaradores de matrices

En una declaración `T D`, donde `D` tiene la forma

`D1 [`*expresión-constante]*

y el tipo del identificador en la declaración `T D1` es "modificador de tipo `T`", el tipo del identificador de `D` es "matriz modificadora de tipo de `T`". Si la expresión-constante está presente, debe tener un tipo entero y un valor mayor que 0. Si falta la expresión constante que especifica el límite, la matriz tiene un tipo incompleto.

Una matriz se puede construir a partir de un tipo aritmético, de un puntero, de una estructura o unión, o de otra matriz (para generar una matriz multidimensional). Cualquier tipo a partir del cual se construya una matriz debe estar completo; No debe ser una matriz de estructura de tipo incompleto. Esto implica que para una matriz multidimensional, es posible que solo falte la primera dimensión. El tipo de un objeto de tipo aray incompleto se completa con otra declaración completa para el objeto (Par.A.10.2), o inicializándolo (Par.A.8.7). Por ejemplo

```
flotador fa[17], *afp[17];
```

Declara una matriz de `números flotantes y una matriz de punteros a` números flotantes. Además

```
static int x3d[3][5][7];
```

declara una matriz tridimensional estática de enteros, con rango 3 `X` 5 `X` 7. En detalle `completo, x3d` es una matriz de tres elementos: cada elemento es una matriz de cinco matrices; cada una de las últimas matrices es una matriz de siete números enteros. Cualquiera de las expresiones `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` puede aparecer razonablemente en una expresión. Los tres primeros tienen el tipo "array", el último tiene el tipo `int`. Más específicamente, `x3d[i][j]` es una matriz de 7 enteros, y `x3d[i]` es una matriz de 5 matrices de 7 enteros.

La operación de subíndice de matriz se define de modo que `E1[E2]` sea idéntico a `*(E1+E2)`. Por lo tanto, a pesar de su apariencia asimétrica, el subíndice es una operación conmutativa. Debido a las reglas de conversión que se aplican a `+` y a las matrices (Pars.A6.6, A.7.1, A.7.7), si `E1` es una matriz y `E2` un número entero, entonces `E1[E2]` se refiere al `E2-ésimo miembro de` E1.

En el ejemplo, `x3d[i][j][k]` es equivalente a `*(x3d[i][j] + k)`. La primera subexpresión `x3d[i][j]` es convertida por Par.A.7.1 al tipo "puntero a matriz de enteros", por Par.A.7.7, la suma implica la multiplicación por el tamaño de un entero. De las reglas se deduce que las matrices se almacenan por filas (el último subíndice varía más rápido) y que el primer subíndice de la declaración ayuda a determinar la cantidad de almacenamiento consumido por una matriz, pero no desempeña ningún otro papel en los cálculos de subíndices.

A.8.6.3 Declaradores de funciones

En una declaración de función de nuevo estilo, `T D`, donde `D` tiene la forma

`D1` (*lista-de-tipos-de-parámetros*)

y el tipo del identificador en la declaración `T D1` es "modificador de tipo `T`", el tipo del identificador de `D` es "función modificadora de tipo con argumentos *parameter-type-list* que devuelve `T`."

La sintaxis de los parámetros es

lista-de-tipos-de-parámetros:
lista-de-parámetros
lista-de-parámetros `, ...`

lista-de-parámetros:
declaración-parámetro
lista-de-parámetros `,` *declaración-de-parámetros*

declaración de parámetros:
declaradores-especificadores-declaración
declarador especificadores-declaración-
abstractoopt

En la declaración de nuevo estilo, la lista de parámetros especifica los tipos de los parámetros. Como caso especial, el declarador de una función de nuevo estilo sin parámetros tiene una lista de parámetros que consiste únicamente en la palabra clave `void`. Si la lista de parámetros termina con puntos suspensivos "`, ...`", entonces la función puede aceptar más argumentos que el número de parámetros descritos explícitamente, véase Par.A.7.3.2.

Los tipos de parámetros que son matrices o funciones se modifican a punteros, de acuerdo con las reglas para conversiones de parámetros; véase el párrafo A.10.1. El único especificador de clase de almacenamiento permitido en la declaración de un parámetro es `register`, y este especificador se omite a menos que el declarador de función encabece una definición de función. Del mismo modo, si los declaradores de las declaraciones de parámetros contienen identificadores y el declarador de función no encabeza una definición de función, los identificadores salen del ámbito inmediatamente. Los declarantes abstractos, que no mencionan los identificadores, se analizan en el párrafo A.8.8.

En una declaración de función de estilo antiguo, `T D`, donde `D` tiene la forma

`D1`(*identificador-lista)*

y el tipo del identificador en la declaración `T D1` es "modificador de tipo `T`", el tipo del identificador de `D` es "función modificadora de tipo de argumentos no especificados que devuelven `T`". Los parámetros (si están presentes) tienen la forma

lista-de-identificadores:
identificador
lista-de-identificadores `,` *identificador*

En el declarador de estilo antiguo, la lista de identificadores debe estar ausente a menos que el declarador se utilice en el encabezado de una definición de función (Par.A.10.1). La declaración no proporciona información sobre los tipos de los parámetros.

Por ejemplo, la declaración

```
int f(), *fpi(), (*pfi)();
```

Declara una función `f` que devuelve un entero, una función `fpi` que devuelve un puntero a un entero y un puntero `pfi` a una función que devuelve un entero. En ninguno de ellos se especifican los tipos de parámetros; Son a la antigua.

En la declaración de nuevo estilo

```
int strcpy(char *dest, const char *fuente), rand(vacío);
```

`strcpy` es una función que devuelve `int`, con dos argumentos, el primero un puntero de caracteres y el segundo un puntero a caracteres constantes. Los nombres de los parámetros son, en efecto, comentarios. La segunda función `rand` no toma ningún argumento y devuelve `int`.

> Los declaradores de funciones con prototipos de parámetros son, con diferencia, el cambio de lenguaje más importante introducido por el estándar ANSI. Ofrecen una ventaja sobre los declaradores de "estilo antiguo" de la primera edición al proporcionar detección de errores y coerción de argumentos a través de llamadas a funciones, pero a un costo: agitación y confusión durante su introducción, y la necesidad de acomodar ambas formas. Se requería cierta fealdad sintáctica en aras de la compatibilidad, a saber, `void` como marcador explícito de funciones de nuevo estilo sin parámetros.
>
> La notación de puntos suspensivos "`, ...`" para funciones variádicas también es nuevo, y, junto con las macros en el encabezado estándar `<stdarg.h>`, formaliza un mecanismo que estaba oficialmente prohibido pero no oficialmente aprobado en la primera edición.
>
> Estas notaciones fueron adaptadas del lenguaje C++.

A.8.7 Inicialización

Cuando se declara un objeto, su declarador de inicialización puede especificar un valor inicial para el identificador que se está declarando. El inicializador va precedido de `=` y es una expresión o una lista de inicializadores anidados entre llaves. Una lista puede terminar con una coma, una sutileza para un formato ordenado.

Inicializador:
  expresión-asignación
  `{` *lista-de-inicializador* `}`
  `{` *lista-de-inicializador* `,` `}`

lista-de-inicializador:
  Inicializador
  lista-de-inicializadores `,` *inicializador*

Todas las expresiones del inicializador de un objeto estático o una matriz deben ser expresiones constantes, tal como se describe en el párrafo A.7.19. Las expresiones del inicializador para un objeto `o matriz auto` o `register` también deben ser expresiones constantes si el inicializador es una lista entre llaves. Sin embargo, si el inicializador de un objeto automático es una sola expresión, no es necesario que sea una expresión constante, sino que simplemente debe tener el tipo adecuado para la asignación al objeto.

> La primera edición no aprobaba la inicialización de estructuras, uniones o matrices automáticas. El estándar ANSI lo permite, pero solo mediante construcciones constantes, a menos que el inicializador se pueda expresar mediante una expresión simple.

Un objeto estático no inicializado explícitamente se inicializa como si a él (o a sus miembros) se le asignara la constante 0. El valor inicial de un objeto automático no inicializado explícitamente es indefinido.

El inicializador de un puntero o un objeto de tipo aritmético es una sola expresión, quizás entre llaves. La expresión se asigna al objeto.

El inicializador de una estructura es una expresión del mismo tipo o una lista de inicializadores entre llaves para sus miembros en orden. Los miembros de campo de bits sin nombre se omiten y no se inicializan. Si hay menos inicializadores en la lista que miembros de la estructura, los miembros finales se inicializan con 0. Es posible que no haya más inicializadores que miembros. Los miembros de campo de bits sin nombre se ignoran y no se inicializan.

El inicializador de una matriz es una lista de inicializadores entre llaves para sus miembros. Si la matriz tiene un tamaño desconocido, el número de inicializadores determina el tamaño de la matriz y su tipo se completa. Si la matriz tiene un tamaño fijo, el número de inicializadores no puede exceder el número de miembros de la matriz; Si hay menos, los miembros finales se inicializan con 0.

Como caso especial, una matriz de caracteres puede ser inicializada por un literal de cadena; los caracteres sucesivos de la cadena inicializan los miembros sucesivos de la matriz. Del mismo modo, un literal de caracteres anchos (Par.A.2.6) puede inicializar una matriz de tipo `wchar_t`. Si la matriz tiene un tamaño desconocido, el número de caracteres de la cadena, incluido el carácter nulo de terminación, determina su tamaño; Si su tamaño es fijo, el número de caracteres de la cadena, sin contar el carácter nulo de terminación, no debe superar el tamaño de la matriz.

El inicializador de una unión es una sola expresión del mismo tipo o un inicializador entre llaves para el primer miembro de la unión.

> La primera edición no permitía la inicialización de uniones. La regla del "primer miembro" es torpe, pero es difícil de generalizar sin una nueva sintaxis. Además de permitir que las uniones se inicialicen explícitamente al menos de una manera primitiva, esta regla ANSI define la semántica de las uniones estáticas no inicializadas explícitamente.

Un *agregado* es una estructura o matriz. Si un agregado contiene miembros de tipo agregado, las reglas de inicialización se aplican de forma recursiva. Las llaves se pueden elidar en la inicialización de la siguiente manera: si el inicializador de un miembro de agregado que a su vez es un agregado comienza con una llave izquierda, la lista sucesiva de inicializadores separados por comas inicializa los miembros del subagregado; Es erróneo que haya más inicializadores que miembros. Sin embargo, si el inicializador de un subagregado no comienza con una llave izquierda, solo se tienen en cuenta los elementos suficientes de la lista para los miembros del subagregado; Los miembros restantes se dejan para inicializar el siguiente miembro del agregado del que forma parte el subagregado.

Por ejemplo

```
int x[] = { 1, 3, 5 };
```

Declara e inicializa `x` como una matriz de 1 dimensión con tres miembros, ya que no se especificó ningún tamaño y hay tres inicializadores.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

es una inicialización completamente entre corchetes: 1, 3 y 5 inicializan la primera fila de la matriz `y[0]`, es decir, `y[0][0]`, `y[0][1]` e `y[0][2]`. Del mismo modo, las dos líneas siguientes inicializan `y[1]` e `y[2]`. El inicializador termina antes de tiempo y, por lo tanto, los elementos de `y[3]` se inicializan con 0. Precisamente el mismo efecto podría haberse logrado

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

El inicializador de `y` comienza con una llave izquierda, pero el `de y[0]` no; por lo tanto, se utilizan tres elementos de la lista. Del mismo modo, los tres siguientes se toman sucesivamente para `y[1]` y para `y[2]`. Además

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

Inicializa la primera columna de `Y` (considerada como una matriz bidimensional) y deja el

resto 0. Finalmente

```
char msg[] = "Error de sintaxis en la línea %s\n";
```

muestra una matriz de caracteres cuyos miembros se inicializan con una cadena; Su tamaño incluye el carácter nulo de terminación.

A.8.8 Nombres de tipo

En varios contextos (para especificar conversiones de tipos explícitamente con una conversión, para declarar tipos de parámetros en declaradores de funciones y como argumento de `sizeof`) es necesario proporcionar el nombre de un tipo de datos. Esto se logra mediante un *nombre* de tipo, que es sintácticamente una declaración para un objeto de ese tipo omitiendo el nombre del objeto.

nombre-tipo:
especificador-calificador-lista-abstracto-declaradoropt

declarador de resúmenes:
puntero
direct-abstract-declarator pointeropt

declarador-abstracto-directo:
(*abstracto-declarativo*)
direct-abstract-declaratoropt [*expresión-constante]*
direct-abstract-declaratoropt (*parámetro-tipo-listopt*)

Es posible identificar de forma única la ubicación en el declarador abstracto donde aparecería el identificador si la construcción fuera un declarador en una declaración. El tipo con nombre es el mismo que el tipo del identificador hipotético. Por ejemplo

```
int
int *
tú *[3]
tú (*)[]
tú*()
int (*[])(void)
```

nombre respectivamente los tipos "entero", "puntero a entero", "matriz de 3 punteros a enteros". "puntero a un número no especificado de enteros", "función de parámetros no especificados que devuelve puntero a entero" y "matriz, de tamaño no especificado, de punteros a funciones sin parámetros, cada una devolviendo un número entero."

A.8.9 Definición de tipo

Las declaraciones cuyo especificador de clase de almacenamiento es `typedef` no declaran objetos, sino que definen identificadores que denominan tipos. Estos identificadores se denominan nombres de definición de tipo.

nombre-de-tipo:
identificador

Una `declaración typedef` atribuye un tipo a cada nombre entre sus declaradores de la manera habitual (consulte Par.A.8.6). A partir de entonces, cada uno de estos nombres de typedef es sintácticamente equivalente a una palabra clave de especificador de tipo para el tipo asociado.

Por ejemplo, después de

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complejo;
```

Las construcciones

```
Bloque b;
externo Blockptr
bp;
Complejo z, *zp;
```

son declaraciones legales. El tipo de `b` es `largo`, el de `bp` es "puntero a `largo`" y el de `z` es la estructura especificada; `zp` es un puntero a dicha estructura.

`typedef` no introduce nuevos tipos, solo sinónimos de tipos que podrían especificarse de otra manera. En el ejemplo, `b` tiene el mismo tipo que cualquier `objeto largo`.

Los nombres de definición de tipo se pueden volver a declarar en un ámbito interno, pero se debe proporcionar un conjunto no vacío de especificadores de tipo. Por ejemplo

```
externo Blockno;
```

no vuelve a declarar `Blockno`, sino

```
external int Blockno;
```

hace.

A.8.10 Equivalencia de tipos

Dos listas de especificadores de tipo son equivalentes si contienen el mismo conjunto de especificadores de tipo, teniendo en cuenta que algunos especificadores pueden estar implícitos en otros (por ejemplo, `long` solo implica `long int`). Las estructuras, uniones y enumeraciones con etiquetas diferentes son distintas, y una unión, estructura o enumeración sin etiquetas especifica un tipo único.

Dos tipos son iguales si sus declaradores abstractos (Par.A.8.8), después de expandir cualquier `tipo de definición de tipo` y eliminar cualquier especificador de parámetros de función, son iguales hasta la equivalencia de las listas de especificadores de tipo. Los tamaños de las matrices y los tipos de parámetros de función son significativos.

A.9 Declaraciones

A excepción de lo descrito, las instrucciones se ejecutan en secuencia. Las instrucciones se ejecutan por su efecto y no tienen valores. Se dividen en varios grupos.

Declaración:
instrucción-
etiquetada sentencia-
expresión-
declaración
compuesta
declaración de
selección
declaración de
iteración
declaración de salto

A.9.1 Declaraciones etiquetadas

Las instrucciones pueden llevar prefijos de etiqueta.

Declaración-etiquetada:
identifier : *declaración*
`case` *constant-expression* : *instrucción*
`default` : *instrucción*

Una etiqueta que consta de un identificador declara el identificador. El único uso de una etiqueta de identificador es como destino de `goto`. El ámbito del identificador es la función actual. Dado que las etiquetas tienen su propio espacio de nombres, no interfieren con otros identificadores y no se pueden volver a declarar. Véase el párrafo A.11.1.

Las etiquetas de mayúsculas y minúsculas y las etiquetas predeterminadas se utilizan con la `instrucción switch` (Par.A.9.4). La expresión constante de `case` debe tener tipo entero.

Las etiquetas en sí mismas no alteran el flujo de control.

A.9.2 Instrucción de expresión

La mayoría de las instrucciones son instrucciones de expresión, que tienen la forma

declaración-expresión:
expresión;

La mayoría de las sentencias de expresión son asignaciones o llamadas a funciones. Todos los efectos secundarios de la expresión se completan antes de que se ejecute la siguiente instrucción. Si falta la expresión, la construcción se denomina instrucción nula; A menudo se usa para proporcionar un cuerpo vacío a una instrucción de iteración para colocar una etiqueta.

A.9.3 Declaración compuesta

Para que se puedan usar varias sentencias donde se espera una, se proporciona la sentencia compuesta (también llamada "bloque"). El cuerpo de una definición de función es una instrucción compuesta.

enunciado compuesto:
`{` *declaración-listopt* declaración-listaopt `}`

declaration-list:
declaración
declaración de lista

lista-de-declaraciones:
declaración
declaración statement-list

Si un identificador de la lista-declaración estaba en el ámbito fuera del bloque, la declaración externa se suspende dentro del bloque (véase el párrafo A.11.1), después de lo cual reanuda su vigor. Un identificador solo se puede declarar una vez en el mismo bloque. Estas reglas se aplican a los identificadores en el mismo espacio de nombres (Par.A.11); Los identificadores en diferentes espacios de nombres se tratan como distintos.

La inicialización de objetos automáticos se realiza cada vez que se introduce el bloque en la parte superior y continúa en el orden de los declaradores. Si se ejecuta un salto en el bloque, estas inicializaciones no se realizan. La inicialización de `objetos estáticos` se realiza solo una vez, antes de que el programa comience a ejecutarse.

A.9.4 Declaraciones de selección

Las instrucciones de selección eligen uno de varios flujos de control.

selección-declaración:
`Sentencia if` (*expresión*)
`if` (*expresión*) *sentencia* `else` *sentencia*
`Instrucción switch` (*expresión*)

En ambas formas de la `instrucción if`, se evalúa la expresión, que debe tener un tipo aritmético o de puntero, incluidos todos los efectos secundarios, y si se compara desigual con 0, se ejecuta la primera subdeclaración. En la segunda forma, la segunda subinstrucción se ejecuta si la expresión es 0. La ambigüedad else se resuelve conectando un `else` con el último else-less `encontrado si se` encuentra en el mismo nivel de anidamiento de bloques.

La instrucción `switch` hace que el control se transfiera a una de varias instrucciones en función del valor de una expresión, que debe tener un tipo entero. La subinstrucción controlada por un `modificador` suele ser compuesta. Cualquier enunciado dentro de la subsentencia puede etiquetarse con una o más `etiquetas de caso` (Par.A.9.1). La expresión de control se somete a la promoción integral (Par.A.6.1) y las constantes de caso se convierten al tipo promocionado. Es posible que dos de estas constantes de caso asociadas con el mismo modificador tengan el mismo valor después de la conversión. También puede haber como máximo una `etiqueta predeterminada` asociada a un modificador. Los conmutadores pueden estar anidados; Una `etiqueta case` o `predeterminada` está asociada con el modificador más pequeño que la contiene.

Cuando se ejecuta la instrucción switch, se evalúa su expresión, incluidos todos los efectos secundarios, y se compara con cada constante de caso. Si una de las constantes de mayúsculas y minúsculas es igual al valor de la expresión, el control pasa a la instrucción de la etiqueta de mayúsculas y `minúsculas` coincidentes. Si ninguna constante de mayúsculas y minúsculas coincide con la expresión, y si hay una `etiqueta predeterminada`, el control pasa a la instrucción etiquetada. Si no coincide con ningún caso, y si no hay ningún `valor predeterminado`, no se ejecuta ninguna de las subinstrucciones del swtich.

> En la primera edición de este libro, se requería que la expresión de control de `switch` y las constantes de caso tuvieran `el tipo` int.

A.9.5 Instrucciones de iteración

Las instrucciones de iteración especifican bucles.

sentencia-iteración:
`while` (*expresión*) *instrucción*
do instrucción `while` (*expresión*);
`para` (*expressionopt; expresión; expressionopt*)

En las sentencias `while` y `do`, la subinstrucción se ejecuta repetidamente siempre que el valor de la expresión siga siendo desigual a 0; la expresión debe tener tipo aritmético o de puntero. Con `while`, la prueba, incluidos todos los efectos secundarios de la expresión, se produce antes de cada ejecución de la instrucción; con `do`, la prueba sigue a cada iteración.

En la instrucción `for`, la primera expresión se evalúa una vez y, por lo tanto, especifica la inicialización del bucle. No hay restricción en su tipo. La segunda expresión debe tener tipo aritmético o de puntero; Se evalúa antes de cada iteración y, si es igual a 0, se termina `el for`. La tercera expresión se evalúa después de cada iteración y, por lo tanto, especifica una reinicialización para el bucle. No hay restricción en su tipo. Los efectos secundarios de cada expresión se completan inmediatamente después de su evaluación. Si la subinstrucción no contiene `continue`, una instrucción

`para` (*expresión1; expresión2; Expresión 3*)

es equivalente a

```
expresión1;
while (expresión2) {
    declaración
    expresión3;
}
```

Se puede descartar cualquiera de las tres expresiones. Una segunda expresión que falta hace que la prueba implícita sea equivalente a probar un elemento distinto de cero.

A.9.6 Instrucciones de salto

Las instrucciones de salto transfieren el control incondicionalmente.

salto-declaración:
`goto`
identificador;
`continuar;`
`interrumpir;`
`return` *expressionopt;*

En la instrucción `goto`, el identificador debe ser una etiqueta (Par.A.9.1) ubicada en la función actual. Las transferencias de control a la instrucción etiquetada.

Una `instrucción continue` solo puede aparecer dentro de una instrucción de iteración. Hace que el control pase a la parte de continuación de bucle de la instrucción más pequeña que contenga dicha instrucción. Más precisamente, dentro de cada una de las declaraciones

```
mientras que (...) {
  ...
Contin:;
}
```

```
...
Contin:;
} mientras (...);
```

```
hacer {  para (...) {
  ...
Contin:;
}
```

Un `continue` no contenido en una instrucción de iteración más pequeña es lo mismo que `goto contin`.

Una `instrucción break` puede aparecer solo en una instrucción de iteración o en una `instrucción switch`, y finaliza la ejecución de la instrucción más pequeña que la contiene; el control pasa a la instrucción que sigue a la instrucción terminada.

Una función regresa a su llamador mediante la `instrucción return`. Cuando `return` va seguido de una expresión, el valor se devuelve al llamador de la función. La expresión se convierte, como por asignación, en el tipo devuelto por la función en la que aparece.

Fluir desde el final de una función es equivalente a un retorno sin expresión. En cualquier caso, el valor devuelto es undefined.

A.10 Declaraciones Externas

La unidad de entrada proporcionada al compilador de C se denomina unidad de traducción; Consta de una secuencia de declaraciones externas, que son declaraciones o definiciones de funciones.

Unidad de traducción:
declaración-externa
declaración-externa-de-la-unidad de traducción

declaración externa:
declaración de definición
de función

El ámbito de las declaraciones externas persiste hasta el final de la unidad de traducción en la que se declaran, del mismo modo que el efecto de las declaraciones dentro de los bloques persiste hasta el final del bloque. La sintaxis de las declaraciones externas es la misma que la de todas las declaraciones, excepto que solo en este nivel se puede proporcionar el código de las funciones.

A.10.1 Definiciones de funciones

Las definiciones de función tienen la forma

definición de función:
declarador-de-declaraciónopt declarador declaración-listaopt declaración-declaración-compuesto

Los únicos especificadores de clase de almacenamiento permitidos entre los especificadores de declaración son `extern` o
`estático`; véase el párrafo A.11.2 para la distinción entre ellos.

Una función puede devolver un tipo aritmético, una estructura, una unión, un puntero o `un vacío`, pero no una función o una matriz. El declarador de una declaración de función debe especificar explícitamente que el identificador declarado tiene el tipo de función; es decir, debe contener una de las formas (véase el párrafo A.8.6.3).

declarador directo (*lista-de-tipos-de-parámetros*)
declarador-directo (*identificador-listopt*)

donde el declarador directo es un identificador o un identificador entre paréntesis. En particular, no debe lograr el tipo de función por medio de una definición de `tipo`.

En la primera forma, la definición es una función de nuevo estilo, y sus parámetros, junto con sus tipos, se declaran en su lista de tipos de parámetros; la lista de declaraciones que sigue al declarador de la función debe estar ausente. A menos que la lista de tipos de parámetros conste únicamente de `void`, lo que muestra que la función no toma parámetros, cada declarador de la lista de tipos de parámetros debe contener un identificador. Si la lista de tipos de parámetros termina con ", `...`" entonces la función puede ser llamada con más argumentos que parámetros; el `mecanismo de macro va_arg` definido en el encabezado estándar `<stdarg.h>` y descrito en el Apéndice B debe ser usado para referirse a los argumentos adicionales. Las funciones variádicas deben tener al menos un parámetro con nombre.

En la segunda forma, la definición es de estilo antiguo: la lista de identificadores nombra los parámetros, mientras que la lista de atributos de declaración les atribuye tipos. Si no se proporciona ninguna declaración para un parámetro, se considera que su tipo es `int`. La lista de declaraciones debe declarar solo los parámetros nombrados en la lista, no se permite la inicialización y el único especificador de clase de almacenamiento posible es `register`.

En ambos estilos de definición de función, se entiende que los parámetros se declaran justo después del comienzo de la declaración compuesta que constituye el cuerpo de la función y, por lo tanto, los mismos identificadores no deben volver a declararse allí (aunque pueden, como otros identificadores, volver a declararse en bloques internos). Si se declara que un parámetro tiene el tipo "array of *type*", la declaración se ajusta para que diga "pointer to *type*"; de manera similar, si se declara que un parámetro tiene el tipo "function returning *type*", la declaración se ajusta para que diga "pointer to function *returning type*". Durante la llamada a una función, los argumentos se convierten según sea necesario y se asignan a los parámetros; véase el párrafo A.7.3.2.

> Las definiciones de funciones de nuevo estilo son nuevas con el estándar ANSI. También hay un pequeño cambio en los detalles de la promoción; La primera edición especificaba que las declaraciones de los parámetros float se ajustaban para que dijeran `double`. La diferencia se hace notable cuando se genera un puntero a un parámetro dentro de una función.

Un ejemplo completo de una definición de función de nuevo estilo es

```
int max (int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    Volver (M > C) ? m : c;
}
```

Aquí int es el especificador de declaración; `max(int a, int b, int c)` es el declarador de la función, y `{ ... }` es el bloque que da el código para la función. La definición antigua correspondiente sería la siguiente:

```
int máx(A, B, C)
int a, b, c;
{
    /* ... */
}
```

donde ahora `int max(a, b, c)` es el declarador, e `int a, b, c;` es la lista de declaración de los parámetros.

A.10.2 Declaraciones Externas

Las declaraciones externas especifican las características de los objetos, funciones y otros identificadores. El término "externo" se refiere a su ubicación fuera de las funciones, y no está directamente relacionado con

La clase de almacenamiento de un objeto declarado externamente puede dejarse vacía o especificarse como `extern` o `static`.

Pueden existir varias declaraciones externas para el mismo identificador dentro de la misma unidad de traducción si coinciden en el tipo y la vinculación, y si hay como máximo una definición para el identificador.

Se considera que dos declaraciones para un objeto o función coinciden en tipo según la regla discutida en el párrafo A.8.10. Además, si las declaraciones difieren porque un tipo es un tipo de estructura, unión o enumeración incompleto (Par.A.8.3) y el otro es el tipo completado correspondiente con la misma etiqueta, se considera que los tipos coinciden. Además, si un tipo es un tipo de matriz incompleta (Par.A.8.6.2) y el otro es un tipo de matriz completa, los tipos, si son idénticos, también se consideran concordantes. Por último, si un tipo especifica una función de estilo antiguo y el otro una función de estilo nuevo idéntica, con declaraciones de parámetros, se considera que los tipos coinciden.

Si el primer declarador externo para una función u objeto incluye el especificador estático, el identificador tiene *vinculación interna*; de lo contrario, tiene *vinculación externa*. La vinculación se analiza en el párrafo 11.2.

Una declaración externa para un objeto es una definición si tiene un inicializador. Una declaración de objeto externo que no tiene un inicializador y que no contiene el especificador `extern` es una *definición provisional*. Si una definición para un objeto aparece en una unidad de traducción, las definiciones tentativas se tratan simplemente como declaraciones redundantes. Si no aparece ninguna definición para el objeto en la unidad de traducción, todas sus definiciones provisionales se convierten en una sola definición con el inicializador 0.

Cada objeto debe tener exactamente una definición. En el caso de los objetos con vinculación interna, esta regla se aplica por separado a cada unidad de traducción, ya que los objetos enlazados internamente son exclusivos de una unidad de conversión. En el caso de los objetos con vinculación externa, se aplica a todo el programa.

> Aunque la regla de una definición está formulada de manera algo diferente en la primera edición de este libro, en realidad es idéntica a la que se menciona aquí. Algunas implementaciones lo relajan generalizando la noción de definición tentativa. En la formulación alternativa, que es usual en los sistemas UNIX y reconocida como una extensión común por el estándar, todas las definiciones tentativas para un objeto vinculado externamente, a través de todas las unidades de traducción del programa, se consideran juntas en lugar de en cada unidad de traducción por separado. Si aparece una definición en algún lugar del programa, las definiciones tentativas se convierten simplemente en declaraciones, pero si no aparece ninguna definición, todas sus definiciones tentativas se convierten en una definición con inicializador 0.

A.11 Ámbito de aplicación y vinculación

No es necesario compilar todo un programa a la vez: el texto fuente puede guardarse en varios archivos que contienen unidades de traducción, y las rutinas precompiladas pueden cargarse desde bibliotecas. La comunicación entre las funciones de un programa puede llevarse a cabo tanto a través de llamadas como mediante la manipulación de datos externos.

Por lo tanto, hay dos tipos de alcance a considerar: en primer lugar, el *alcance léxico* de un identificador, que es la región del texto del programa dentro de la cual se entienden las características del identificador; y en segundo lugar, el alcance asociado con objetos y funciones con vinculación externa, que determina las conexiones entre identificadores en unidades de traducción compiladas por separado.

A.11.1 Ámbito léxico

Los identificadores se dividen en varios espacios de nombres que no interfieren entre sí; el mismo identificador se puede utilizar para diferentes propósitos, incluso en el mismo ámbito, si los usos se realizan en diferentes espacios de nombres. Estas clases son: objetos, funciones, nombres de definición de tipos y `constantes de enumeración`; etiquetas; etiquetas de estructuras o uniones, y enumeraciones; y miembros de cada estructura o unión individualmente.

> Estas reglas difieren en varios aspectos de las descritas en la primera edición de este manual. Anteriormente, las etiquetas no tenían su propio espacio de nombres; Las etiquetas de estructuras y uniones tenían cada una un espacio separado, y en algunas implementaciones las etiquetas de enumeración también lo hacían; Poner diferentes tipos de etiquetas en el mismo espacio es una nueva restricción. La diferencia más importante con respecto a la primera edición es que cada estructura o unión crea un espacio de nombres separado para sus miembros, de modo que el mismo nombre puede aparecer en varias estructuras diferentes. Esta regla ha sido una práctica común durante varios años.

El ámbito léxico de un identificador de objeto o función en una declaración externa comienza al final de su declarador y persiste hasta el final de la unidad de traducción en la que aparece. El ámbito de un parámetro de una definición de función comienza al principio del bloque que define la función y persiste a través de la función; El ámbito de un parámetro en una declaración de función finaliza al final del declarador. El ámbito de un identificador declarado en el encabezado de un bloque comienza al final de su declarador y persiste hasta el final del bloque. El ámbito de una etiqueta es el conjunto de la función en la que aparece. El ámbito de una etiqueta de estructura, unión o enumeración, o una constante de enumeración, comienza en su aparición en un especificador de tipo y persiste hasta el final de una unidad de traducción (para las declaraciones en el nivel externo) o hasta el final del bloque (para las declaraciones dentro de una función).

Si se declara explícitamente un identificador en la cabecera de un bloque, incluido el bloque que constituye una función, cualquier declaración del identificador fuera del bloque se suspende hasta el final del bloque.

A.11.2 Ligamiento

Dentro de una unidad de traducción, todas las declaraciones del mismo identificador de objeto o función con enlace interno hacen referencia a lo mismo, y el objeto o función es único para esa unidad de traducción. Todas las declaraciones para el mismo identificador de objeto o función con vinculación externa hacen referencia a lo mismo, y el objeto o función es compartido por todo el programa.

Como se discutió en el párrafo A.10.2, la primera declaración externa de un identificador proporciona al identificador un enlace interno si se utiliza el especificador estático, y un enlace externo en caso contrario. Si una declaración para un identificador dentro de un bloque no incluye el `especificador extern`, el identificador no tiene vínculo y es único para la función. Si incluye `extern`, y una declaración externa for está activa en el ámbito que rodea al bloque, entonces el identificador tiene el mismo enlace que la declaración externa y hace referencia al mismo objeto o función; pero si no hay ninguna declaración externa visible, su enlace es externo.

A.12 Preprocesamiento

Un preprocesador realiza la sustitución de macros, la compilación condicional y la inclusión de archivos con nombre. Las líneas que comienzan con `#`, quizás precedidas por un espacio en blanco, se comunican con este preprocesador. La sintaxis de estas líneas es independiente del resto del lenguaje; Pueden aparecer en cualquier lugar y tener un efecto que dura (independientemente del alcance) hasta el final de la unidad de traducción. Los límites de línea son significativos; cada línea se analiza individualmente (ver bus Par.A.12.2 para saber cómo unir líneas). Para el preprocesador, un token es cualquier token de lenguaje, o una secuencia de caracteres que da un nombre de archivo como en la `directiva #include` (Par.A.12.4); además,

Cualquier carácter que no esté definido de otro modo se toma como un token. Sin embargo, el efecto de los espacios en blanco distintos del espacio y la tabulación horizontal no está definido dentro de las líneas del preprocesador.

El preprocesamiento en sí mismo tiene lugar en varias fases lógicamente sucesivas que, en una implementación particular, pueden condensarse.

1. En primer lugar, las secuencias de trígrafo descritas en el párrafo A.12.1 se sustituyen por sus equivalentes. En caso de que el entorno del sistema operativo lo requiera, se introducen caracteres de nueva línea entre las líneas del archivo fuente.
2. Cada aparición de un carácter de barra invertida `\` seguido de una nueva línea se elimina, esto empalma las líneas (Par.A.12.2).
3. El programa se divide en tokens separados por caracteres de espacio en blanco; los comentarios se reemplazan por un solo espacio. A continuación, se obedecen las directivas de preprocesamiento y las macros (Pars.A.12.3-A.12.10) se amplían.
4. Secuencias de escape en constantes de caracteres y literales de cadena (Pars. A.2.5.2, A.2.6) se sustituyen por sus equivalentes; A continuación, se concatenan los literales de cadena adyacentes.
5. El resultado se traduce, luego se vincula con otros programas y bibliotecas, recopilando los programas y datos necesarios, y conectando funciones externas y referencias de objetos a sus definiciones.

A.12.1 Secuencias de trígrafo

El conjunto de caracteres de los programas fuente C está contenido en ASCII de siete bits, pero es un superconjunto del conjunto de códigos invariantes ISO 646-1983. Con el fin de permitir que los programas se representen en el conjunto reducido, todas las apariciones de las siguientes secuencias de trígrafos se reemplazan por el carácter único correspondiente. Esta sustitución se produce antes de cualquier otro tratamiento.

```
??=  #          ??(  [          ??<  {
??/  \          ??)  ]          ??>  }
??'  ^          ??!  |          ??-  ~
```

No se producen otros reemplazos de este tipo.

Las secuencias de trígrafos son nuevas con el estándar ANSI.

A.12.2 Empalme de línea

Las líneas que terminan con el carácter de barra diagonal inversa `\` se pliegan eliminando la barra diagonal inversa y el siguiente carácter de nueva línea. Esto ocurre antes de la división en tokens.

A.12.3 Definición y expansión de macros

Una línea de control de la forma

`# definir` *la secuencia de tokens del identificador*

hace que el preprocesador reemplace las instancias posteriores del identificador con la secuencia de tokens dada; se descartan los espacios en blanco iniciales y finales alrededor de la secuencia de tokens. Un segundo `#define` para el mismo identificador es erróneo a menos que la segunda secuencia de tokens sea idéntica a la primera, donde todas las separaciones de espacios en blanco se consideran equivalentes.

Una línea de la forma

`# definir` *la* secuencia de tokens del *identificador (*lista de identificadores*)*

donde no hay espacio entre el primer identificador y el (, es una definición de macro con parámetros dados por la lista de identificadores. Al igual que con la primera forma, se descarta el espacio en blanco inicial y final alrededor de la secuencia de tokens, y la macro solo se puede redefinir con una definición en la que el número y la ortografía de los parámetros, y la secuencia de tokens, sean idénticos.

Una línea de control de la forma

`# identificador` *undef*

hace que se olvide la definición de preprocesador del identificador. No es erróneo aplicar `#undef` a un identificador desconocido.

Cuando se ha definido una macro en la segunda forma, las instancias textuales subsiguientes del identificador de macro seguidas de un espacio en blanco opcional y, a continuación, de (, una secuencia de tokens separados por comas y un) constituyen una llamada de la macro. Los argumentos de la llamada son las secuencias de tokens separadas por comas; Las comas que se entrecomen o protegen mediante paréntesis anidados no separan los argumentos. Durante la recopilación, los argumentos no se expanden en macros. El número de argumentos de la llamada debe coincidir con el número de parámetros de la definición. Una vez aislados los argumentos, se eliminan los espacios en blanco iniciales y finales. A continuación, la secuencia de tokens resultante de cada argumento se sustituye por cada aparición sin comillas del identificador del parámetro correspondiente en la secuencia de tokens de reemplazo de la macro. A menos que el parámetro de la secuencia de reemplazo esté precedido por `#`, o precedido o seguido por `##`, los tokens de argumento se examinan en busca de llamadas a macros y se expanden según sea necesario, justo antes de la inserción.

Dos operadores especiales influyen en el proceso de sustitución. En primer lugar, si una aparición de un parámetro en la secuencia de tokens de reemplazo está inmediatamente precedida por `#`, las comillas de cadena (`"`) se colocan alrededor del parámetro correspondiente y, a continuación, tanto # como el identificador del parámetro se reemplazan por el argumento citado. Se inserta un `carácter \` antes de cada carácter `"` o `\` que aparece alrededor o dentro de un literal de cadena o constante de caracteres en el argumento.

En segundo lugar, si la secuencia de tokens de definición para cualquier tipo de macro contiene un `operador ##`, justo después de reemplazar los parámetros, cada `##` se elimina, junto con cualquier espacio en blanco a ambos lados, para concatenar los tokens adyacentes y formar un nuevo token. El efecto es indefinido si se producen tokens no válidos o si el resultado depende del orden de procesamiento de los `operadores ##`. Además, es posible que `##` no aparezca al principio o al final de una secuencia de tokens de reemplazo.

En ambos tipos de macros, la secuencia de tokens de reemplazo se vuelve a examinar repetidamente en busca de identificadores más definidos. Sin embargo, una vez que se ha reemplazado un identificador determinado en una expansión determinada, no se reemplaza si vuelve a aparecer durante el nuevo escaneo; en su lugar, se deja sin cambios.

Incluso si el valor final de una expansión de macro comienza con `#`, no se considera una directiva de preprocesamiento.

> Los detalles del proceso de macroexpansión se describen con mayor precisión en la norma ANSI que en la primera edición. El cambio más importante es la adición de los operadores # y ##, que hacen que la cita y la concatenación sean admisibles. Algunas de las nuevas reglas, especialmente las que involucran la concatenación, son extrañas. (Vea el ejemplo a continuación).

Por ejemplo, esta función se puede usar para "constantes de manifiesto", como en

```
#define TABSIZE 100
int table[TABSIZE];
```

La definición

```
#define ABSDIFF(a, b) (a)>(b) ? a)-b) : b)-a))
```

Define una macro para devolver el valor absoluto de la diferencia entre sus argumentos. A diferencia de una función que hace lo mismo, los argumentos y el valor devuelto pueden tener cualquier tipo aritmético o incluso ser punteros. Además, los argumentos, que pueden tener efectos secundarios, se evalúan dos veces, una para la prueba y otra para producir el valor.

Dada la definición

```
#define tempfile(dir)    #dir "%s"
```

La llamada a la macro `tempfile(/usr/tmp)` produce

```
"/usr/tmp" "%s"
```

que posteriormente se catenará en una sola cadena. Después

```
#define gato(x, y)      x ## y
```

La llamada `cat(var, 123)` produce `var123`. Sin embargo, la llamada `cat(cat(1,2),3)` es indefinida: la presencia de `##` impide que se expandan los argumentos de la llamada externa. Por lo tanto, produce la cadena de token

```
gato ( 1 , 2 )3
```

y `)3` (la catenación del último token del primer argumento con el primer token del segundo) no es un token legal. Si se introduce un segundo nivel de definición de macro,

```
#define xcat(x, y)      gato (x,y)
```

las cosas funcionan de manera más fluida; `xcat(xcat(1, 2), 3)` produce `123`, porque la expansión de `xcat` en sí misma no involucra el `operador ##`.

Del mismo modo, `ABSDIFF(ABSDIFF(a,b),c)` produce el resultado esperado y completamente expandido.

A.12.4 Inclusión de

archivos Una línea de

control con el formato `#`

`include <nombre de`

`archivo>`

provoca el reemplazo de esa línea por todo el contenido del nombre de archivo del archivo. Los caracteres del nombre *de archivo* no deben incluir `>` ni una nueva línea, y el efecto es indefinido si contiene cualquiera de `los términos ", ', \` o `/*`. El archivo con nombre se busca en una secuencia de lugares definidos por la implementación.

Del mismo modo, una línea de control de la forma

`# incluir "`*nombre de archivo*`"`

Busca primero en asociación con el archivo fuente original (una frase deliberadamente dependiente de la implementación), y si esa búsqueda falla, entonces como en la primera forma. El efecto de usar `'`, `\` o
`/*` en el nombre del archivo permanece sin definir, pero `>` está permitido.

Por último, una directiva de la forma

`# incluir` *secuencia de tokens*

No coincidir con una de las formas anteriores se interpreta expandiendo la secuencia de tokens como para el texto normal; una de las dos formas con `<... >` o `"..."` debe resultar y, a continuación, se trata como se ha descrito anteriormente.

`#include` archivos pueden estar anidados.

A.12.5 Compilación condicional

Las partes de un programa se pueden compilar condicionalmente, de acuerdo con la siguiente sintaxis esquemática.

preprocessor-conditional:
texto de línea if elif-parts else-partopt `#endif`

Línea if:
`# si` *expresión-constante*
`# identificador ifdef`
`# identificador` *ifndef*

Piezas ELIF:
texto de
línea elif
elif-
partsopt

Elif-line:
`#` *expresión-constante elif*

else-part:
else-line texto

else-line:
`#else`

Cada una de las directivas (if-line, elif-line, else-line y `#endif`) aparece sola en una línea. Las expresiones constantes de `#if` líneas #elif y siguientes se evalúan en orden hasta que se encuentra una expresión con un valor distinto de cero; se descarta el texto que sigue a una línea con un valor cero. El texto que sigue a la línea de directiva correcta se trata normalmente. "Texto" aquí se refiere a cualquier material, incluidas las líneas de preprocesador, que no sea parte de la estructura condicional; Puede estar vacío. Una vez que `se ha encontrado una línea #if` o `#elif` correcta y se ha procesado su texto, se descartan las líneas `#elif` y `#else` siguientes, junto con su texto. Si todas las expresiones son cero y hay un `#else`, el texto que sigue al `#else` se trata normalmente. El texto controlado por los brazos inactivos del condicional se omite, excepto para comprobar el anidamiento de los condicionales.

La expresión constante en `#if` y `#elif` está sujeta a la sustitución de macros ordinarias. Además, cualquier expresión de la forma

`Identificador` *definido*

o

`definido` (*identificador*)

se reemplazan, antes de buscar macros, por `1L` si el identificador está definido en el preprocesador, y por `0L` si no lo está. Los identificadores que quedan después de la expansión de macros se reemplazan por `0L`. Por último, se considera que cada constante entera tiene el sufijo `L`, por lo que toda la aritmética se considera larga o larga sin signo.

La expresión constante resultante (Par.A.7.19) está restringida: debe ser integral y no puede contener `sizeof`, una conversión o una constante de enumeración.

Las líneas de control

`Identificador` *de #ifdef*
`Identificador de #ifndef`

son equivalentes a

`# si se define` *el identificador*
`# if !` *identificador definido*

respectivamente.

> `#elif` es nuevo desde la primera edición, aunque ha estado disponible en algunos preprocesadores. El `El` operador de preprocesador definido también es nuevo.

A.12.6 Control de línea

Para el beneficio de otros preprocesadores que generan programas C, una línea en una de las formas

`#` *constante* `de línea`
`"`*nombre de archivo*`" #`
`constante de línea`

Hace que el compilador crea, con fines de diagnóstico de errores, que el número de línea de la siguiente línea de origen viene dado por la constante entera decimal y que el identificador asigna un nombre al archivo de entrada actual. Si el nombre de archivo citado está ausente, el nombre recordado no cambia. Las macros de la línea se expanden antes de que se interprete.

A.12.7 Generación de errores

Una línea de preprocesador de la forma

`# error` *token-sequenceopt*

Hace que el preprocesador escriba un mensaje de diagnóstico que incluya la secuencia de tokens.

A.12.8 Pragmas

Una línea de control de la forma

```
# pragma token-sequenceopt
```

hace que el preprocesador realice una acción dependiente de la implementación. Un pragma no reconocido se ignora.

A.12.9 Directiva nula

Una línea de control de la forma

```
#
```

no tiene ningún efecto.

A.12.10 Nombres predefinidos

Varios identificadores están predefinidos y se expanden para producir información especial. Ellos, y también el operador de expansión del preprocesador `definido`, no pueden ser indefinidos o redefinidos.

`__LINE` Una constante decimal que contiene el número de línea de origen actual.

`__FILE` Literal de cadena que contiene el nombre del archivo que se está compilando.

`__DATE` Un literal de cadena que contiene la fecha de compilación, con el formato `"Mmmm dd aaaa"`

`__TIME` Un literal de cadena que contiene la hora de la compilación, con el formato `"hh:mm:ss"`

`__STDC` La constante `1`. Se pretende que este identificador se defina como `1` solo en implementaciones conformes con el estándar.

`#error` y `#pragma` son nuevos con el estándar ANSI; las macros de preprocesador predefinidas son nuevas, pero algunas de ellas han estado disponibles en algunas implementaciones.

A.13 Gramática

A continuación se presenta una recapitulación de la gramática que se dio a lo largo de la parte anterior de este apéndice. Tiene exactamente el mismo contenido, pero está en un orden diferente.

La gramática tiene símbolos terminales indefinidos *entero-constante*, *caracteres-constantes*, *constantes flotantes*, *identificadores*, *cadenas* y *enumeraciones-constantes*; las palabras y símbolos al estilo de la máquina de escribir son terminales dados literalmente. Esta gramática se puede transformar mecánicamente en una entrada aceptable para un analizador-generador automático. Además de añadir cualquier marca sintáctica que se utilice para indicar alternativas en las producciones, es necesario ampliar las construcciones "una de", y (dependiendo de las reglas del analizador-generador) duplicar cada producción con un *símbolo de opción*, una vez con el símbolo y otra sin él. Con un cambio adicional, a saber, eliminar el *identificador* typedef-name*:* de producción y hacer *que typedef-name* sea un símbolo terminal, esta gramática es aceptable para el generador de analizadores YACC. Solo tiene un conflicto, generado por la `ambigüedad` del "si-no".

Unidad de traducción:
declaración-externa
declaración-externa-de-la-unidad de traducción

declaración externa:
declaración de definición
de función

definición de función:
declarador-de-declaraciónopt declarador declaración-listaopt declaración-declaración-compuesto

Declaración:
especificadores-de-declaración init-declarator-listopt;

lista-de-declaraciones:
declaración
declaración de lista

especificadores-declaración:
especificador-de-clase-de-almacenamiento
declaracion-especificador-sopt tipo-especificador declaracion-especificadoropt
type-qualifier declaration-specifiersopt

storage-class especificador: uno de los
`registro automático de tipo externo estático`

especificador de tipo: uno de los
`void char corto int long float doble firmado sin signo` *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: uno de los
`Const volátil`

struct-or-union-especificador:
struct-or-union identifieropt `{` *struct-declaration-list* `}`
Identificador de estructura o unión

struct-or-union: una de las siguientes opciones:
`Unión de estructuras`

lista-de-declaración-estructura:
Declaración de estructura
struct-declaration-list declaración de estructura

lista-de-declarante-de-inicio:
declarador-de-inicio
lista-de-declarante-de-inicio, *declarante-de-inicio*

declarador de inicio:
Declarador
declarador = *inicializador*

declaración-estructura:
lista-calificadora-de-estructura-lista-declaradora;

lista-calificador-especificador:
especificador-de-tipo especificador-
calificador-listaopt calificador de
tipo especificador-calificador-listaopt

lista-declarante-destructiva:
Declarador de estructura
lista-de-declarador-estructura , *declarador-estructura*

struct-declarator:
Declarador
declaratoropt : *expresión-constante*

especificador de enumeración:
`enum` *identifieropt* `{` *enumerator-list*
`}` `enum` *identifier*

lista-enumeradora:
Enumerador
lista-enumeradora , *enumerador*

Enumerador:
identificador
identificador = *expresión-constante*

Declarante:
Pointeropt de declaración directa

declarante directo:
identificador
(*declarante*)
declarador-directo `[` *expresión-*
constante:opt `]` *declarador-directo* (
lista-de-tipos-de-parámetros)
declarador-directo (*identificador-listaopt*
)

Puntero:
* *tipo-calificador-listaopt*
* *puntero type-qualifier-listopt*

lista-calificadora-tipo:
calificador de tipo
type-qualifier-list type-qualifier

lista-de-tipos-de-parámetros:
lista-de-parámetros
lista-de-parámetros
, ...

lista-de-parámetros:
declaración-parámetro
lista-de-parámetros , *declaración-de-parámetros*

declaración de parámetros:
declaradores-especificadores-declaración
declarador especificadores-declaración-
abstractoopt

lista-de-identificadores:
identificador
lista-de-identificadores , *identificador*

Inicializador:
expresión-asignación
{ *lista-de-inicializador* }
{ *lista-de-inicializador* , }

lista-de-inicializador:
Inicializador
lista-de-inicializadores , *inicializador*

nombre-tipo:
especificador-calificador-lista-abstracto-declaradoropt

declarador de resúmenes:
puntero
direct-abstract-declarator pointeropt

declarador-abstracto-directo:
(*abstracto-declarativo*)
direct-abstract-declaratoropt [*expresión-constante]*
direct-abstract-declaratoropt (*parámetro-tipo-listopt*)

nombre-de-tipo:
identificador

Declaración:
instrucción-
etiquetada sentencia-
expresión-
declaración
compuesta
declaración de
selección
declaración de
iteración
declaración de salto

Declaración-etiquetada:
identifier : *declaración*
`case` *constant-expression* : *instrucción*
`default` : *instrucción*

expresión-declaración:
expressionopt;

enunciado compuesto:
{ *declaración-listopt declaración-listaopt* }

lista-de-declaraciones:
declaración
declaración statement-list

selección-declaración:
`Sentencia if` (*expresión*)
`if` (*expresión*) *sentencia* `else` *sentencia*
`Instrucción switch` (*expresión*)

sentencia-iteración:
`while` (*expresión*) *instrucción*
do instrucción `while` (*expresión*);
`para` (*expressionopt; expresión; expressionopt*)

salto-declaración:
`goto`
identificador;
`continuar;`
`interrumpir;`
`return` *expressionopt;*

Expresión:
expresión-asignación
expresión `,` *asignación-expresión*

expresión-asignación:
expresión-condicional
expresión-unaria asignación-operador de asignación-expresión

asignado-operador: uno de los
`= *= /= %= += -= <<= >>= &= ^= |=`

expresión-condicional:
expresión-o-lógica
expresión-lógica-OR `?` *expresión* `:` *expresión-condicional*

expresión-constante:
expresión-condicional

expresión-OR-lógica:
expresión-lógica AND
expresión-OR lógica `||` *expresión-lógica AND*

expresión-AND-lógica:
expresión-OR inclusiva
expresión-AND-lógica `&&` *expresión-OR-inclusiva*

expresión-OR inclusiva:
expresión-OR exclusiva
expresión-OR-inclusiva `|` *expresión-OR exclusiva*

expresión-OR exclusiva:
Expresión AND
expresión-OR exclusiva `^` *Expresión-AND*

Expresión AND:
igualdad-expresión
Y-expresión `e` *igualdad-expresión*

igualdad-expresión:
expresión-relacional
expresión-igualdad `==` *expresión-relacional*
expresión-igualdad `!=` *expresión-relacional*

expresión-relacional:
expresión-desplazamiento
expresión-relacional `<` *expresión-*
desplazamiento expresión-relacional `>`
expresión-desplazamiento-expresión-
relacional `<=` *expresión-*
desplazamiento-expresión-relacional `>=`
expresión-desplazamiento

expresión-desplazamiento:
expresión-aditiva
shift-expression `<<` *expresión-aditiva*
expresión-shift-expression `>>`
expresión-aditiva

expresión-aditiva:
expresión-multiplicativa
expresión-aditiva `+` *expresión-multiplicativa*
expresión-aditiva `-` *expresión-multiplicativa*

expresión-multiplicativa:
expresión-multiplicativa `*` *expresión-casta*
expresión-multiplicativa `/` *expresión-casta*
expresión-multiplicativa `%` *expresión-casta*

expresión-cast:
Expresión unaria
(*nombre-tipo*) *expresión-conversión*

expresión-unaria:
Expresión de sufijo
`++expresión unaria`
--expresión unaria
operador unario cast-
expression `sizeof` *unary-*
expression `sizeof` (*nombre-*
tipo)

operador unario: uno de los
`& * + - ~ !`

postfix-expresión:
expresión-primaria

expresión-sufijo[*expresión*]
expresión-de-sufijo(*expresión-argumento-listopt*)
expresión-de-sufijo.*identificador*
postfix-expression->+identifier postfix-expression++
expresión-de-sufijo--

expresión-primaria:
Cadena
constante de
identificador
(*expresión*)

lista-de-expresión-argumento:
expresión-asignación
lista-de-expresión-asignación , *expresión-asignación*

Constante:
constante-entera
constante de carácter
constante flotante
constante de enumeración

La siguiente gramática para el preprocesador resume la estructura de las líneas de control, pero no es adecuada para el análisis mecanizado. Incluye el texto del símbolo, que significa texto de programa ordinario, líneas de control de preprocesador no condicionales o instrucciones condicionales completas del preprocesador.

Línea de control:
definir *la secuencia de tokens del identificador*
define *identifier*(*identifier*, ... , *identifier*) *token-sequence*
identificador *undef*
include <nombre de archivo>
include "*nombre de archivo*"
constante de línea "*nombre de archivo*"
constante de línea
error *token-sequenceopt*
pragma *token-sequenceopt*
#
preprocesador-condicional

preprocessor-conditional:
texto de línea if elif-parts else-partopt #endif

Línea if:
si *expresión-constante*
identificador ifdef
identificador *ifndef*

elif-parts:
texto de línea elif-

elif-
partsopt

Elif-line:
`#` *expresión-constante elif*

else-part:
texto de otra
línea

else-line:
`#else`

Apéndice B - Biblioteca estándar

Este apéndice es un resumen de la biblioteca definida por el estándar ANSI. La biblioteca estándar no forma parte del lenguaje C propiamente dicho, pero un entorno que admita C estándar proporcionará las declaraciones de funciones y las definiciones de tipos y macros de esta biblioteca. Hemos omitido algunas funciones que son de utilidad limitada o fácilmente sintetizables a partir de otras; hemos omitido caracteres multibyte; y hemos omitido la discusión de los problemas de localización; es decir, propiedades que dependen del idioma, la nacionalidad o la cultura local.

Las funciones, tipos y macros de la biblioteca estándar se declaran en *cabeceras* estándar:

```
<assert.h> <float.h>    <matemáticas.h>     <stdarg.h> <stdlib.h>
<tipo.h>    <limits.h> <setjmp.h> <stddef.h> <string.h>
<errno.h>   <locale.h> <signal.h> <stdio.h>     <tiempo.h>
```

Se puede acceder a un encabezado mediante

```
#include <encabezado>
```

Los encabezados se pueden incluir en cualquier orden y en cualquier número de veces. Un encabezado debe incluirse fuera de cualquier declaración o definición externa y antes de cualquier uso de cualquier cosa que declare. Un encabezado no tiene por qué ser un archivo de origen.

Los identificadores externos que comienzan con un carácter de subrayado están reservados para su uso por la biblioteca, al igual que todos los demás identificadores que comienzan con un carácter de subrayado y una letra mayúscula u otro carácter de subrayado.

B.1 Entrada y salida: <stdio.h>

Las funciones, tipos y macros de entrada y salida definidas en `<stdio.h>` representan casi un tercio de la biblioteca.

Un *flujo* es un origen o destino de datos que pueden estar asociados con un disco u otro periférico. La biblioteca soporta flujos de texto y flujos binarios, aunque en algunos sistemas, especialmente UNIX, estos son idénticos. Un flujo de texto es una secuencia de líneas; Cada línea tiene cero o más caracteres y termina con `'\n'`. Es posible que un entorno necesite convertir una secuencia de texto a o desde alguna otra representación (como la asignación `'\n'` al retorno de carro y al salto de línea). Un flujo binario es una secuencia de bytes sin procesar que registran datos internos, con la propiedad de que si se escriben y luego se vuelven a leer en el mismo sistema, se compararán igual.

Una transmisión se conecta a un archivo o dispositivo *al abrirlo*; la conexión se interrumpe al *cerrar* la transmisión. Al abrir un archivo, se devuelve un puntero a un objeto de tipo `FILE`, que registra la información necesaria para controlar la secuencia. Usaremos "file pointer" y "stream" indistintamente cuando no haya ambigüedad.

Cuando un programa comienza a ejecutarse, las tres secuencias `stdin`, `stdout` y `stderr` ya están abiertas.

B.1.1 Operaciones de archivo

Las siguientes funciones se ocupan de las operaciones en los archivos. El tipo `size_t` es el tipo entero sin signo producido por el operador `sizeof`.

`ARCHIVO *fopen(const char *nombre de archivo, const char *modo)`

`fopen` abre el archivo con nombre y devuelve una secuencia, o `NULL` si el intento falla. Los valores válidos para `mode` incluyen:

`"R"` Abrir archivo de texto para lectura

`"w"` crea un archivo de texto para escribir; descarta el contenido

anterior si se agrega alguna `"a"` ; abre o crea un archivo de texto

para escribir al final del archivo `"r+"` abre el archivo de texto

para actualizar (es decir, leer y escribir)

`"w+"` crea un archivo de texto para la actualización, descarte el contenido anterior si lo hay

`Anexar "A+"` ; abrir o crear un archivo de texto para actualizarlo, escribiéndolo al final

El modo de actualización permite leer y escribir el mismo archivo; `Se debe llamar a fflush` o a una función de posicionamiento de archivos entre una lectura y una escritura o viceversa. Si el modo incluye `b` después de la letra inicial, como en `"rb"` o `"w+b"`, eso indica un archivo binario. Los nombres de archivo están limitados a `FILENAME_MAX` caracteres. A lo sumo, `FOPEN_MAX` archivos pueden estar abiertos a la vez.

`ARCHIVO *freopen(const char *nombre de archivo, const char *modo, ARCHIVO *flujo)`

`freopen` abre el archivo con el modo especificado y asocia la secuencia con él. Devuelve `stream`, o `NULL` si se produce un error. `freopen` se usa normalmente para cambiar los archivos asociados con `stdin`, `stdout` o `stderr`.

`int fflush(ARCHIVO *flujo)`

En un flujo de salida, `fflush` hace que se escriban los datos almacenados en búfer pero no escritos; en un flujo de entrada, el efecto es indefinido. Devuelve `EOF` para un error de escritura y cero en caso contrario. `fflush(NULL)` vacía todos los flujos de salida.

`int fclose(ARCHIVO *flujo)`

`fclose` vacía los datos no escritos para la `secuencia`, descarta cualquier entrada almacenada en búfer no leída, libera cualquier búfer asignado automáticamente y, a continuación, cierra la secuencia. Devuelve `EOF` si se ha producido algún error y cero en caso contrario.

`int remove(const char *nombre de archivo)`

`remove` elimina el archivo con nombre, de modo que se producirá un error en un intento posterior de abrirlo. Devuelve un valor distinto de cero si se produce un error en el intento.

`Cambiar el nombre de int (const cuatro *oldname, const four *newname)`

`rename` cambia el nombre de un archivo; devuelve un valor distinto de cero si el intento falla.

`ARCHIVO *tmpfile(vacío)`

`tmpfile` crea un archivo temporal de modo `"WB+"` que se eliminará automáticamente cuando se cierre o cuando el programa finalice normalmente. `tmpdevuelve` una secuencia, o `NULL` si no pudo crear el archivo.

`char *tmpnam(char s[L_tmpnam])`

`tmpnam(NULL)` crea una cadena que no es el nombre de un archivo existente y devuelve un puntero a una matriz estática interna. `tmpnam(s)` almacena la cadena en `s` y la devuelve como el valor de la función; `s` debe tener espacio para al menos `L_tmpnam` caracteres. `tmpnam` genera un nombre diferente cada vez que se le llama; a lo sumo, `se garantizan TMP_MAX` nombres diferentes durante la ejecución del programa. Tenga en cuenta que `tmpnam` crea un nombre, no un archivo.

`int setBoof(archivo *stream, cuatro *BUF, modo int, tamaño Size_T)`

`setvbuf` controla el almacenamiento en búfer para el flujo; debe llamarse antes de leer, escribir o cualquier otra operación. Un `modo` de `_IOFBF` provoca un

almacenamiento en búfer completo, `_IOLBF` almacenamiento en búfer de línea de archivos de texto y `_IONBF` ningún almacenamiento en búfer. Si `buf` no es `NULL`, se utilizará como búfer, de lo contrario, se asignará un búfer. `size` determina el tamaño del búfer. `setvbuf` devuelve un valor distinto de cero para cualquier error.

```
void setbuf(ARCHIVO * flujo, char *buf)
```

Si `buf` es `NULL`, el almacenamiento en búfer está desactivado para la secuencia. De lo contrario, `setbuf` es equivalente a `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

B.1.2 Salida formateada

Las funciones `printf` proporcionan conversión de salida formateada.

```
int fprintf(FILE *stream, const char *format, ...)
```

`fprintf` convierte y escribe la salida en el `flujo` bajo el control de format . El valor devuelto es el número de caracteres escritos, o negativo si se ha producido un error.

La cadena de formato contiene dos tipos de objetos: caracteres ordinarios, que se copian en el flujo de salida, y especificaciones de conversión, cada uno de los cuales provoca la conversión e impresión del siguiente argumento sucesivo en `fprintf`. Cada especificación de conversión comienza con el carácter `%` y termina con un carácter de conversión. Entre el `%` y el carácter de conversión puede haber, en orden:

- Banderas (en cualquier orden), que modifican la especificación:
  - `-`, que especifica el ajuste a la izquierda del argumento convertido en su campo.
  - `+`, que especifica que el número siempre se imprimirá con un signo.
  - *Espacio*: si el primer carácter no es un signo, se prefijará un espacio.
  - `0`: para conversiones numéricas, especifica el relleno del ancho del campo con ceros a la izquierda.
  - `#`, que especifica una forma de salida alternativa. Para `o`, el primer dígito se convertirá en cero. Para `x` o `X`, `0x` o `0X` se prefijarán a un resultado distinto de cero. Para `e`, `E`, `f`, `g` y `G`, la salida siempre tendrá un punto decimal; para `g` y `G`, los ceros finales no se eliminarán.
- Un número que especifica un ancho de campo mínimo. El argumento convertido se imprimirá en un campo de al menos este ancho, y más ancho si es necesario. Si el argumento convertido tiene menos caracteres que el ancho del campo, se rellenará a la izquierda (o a la derecha, si se ha solicitado el ajuste a la izquierda) para compensar el ancho del campo. El carácter de relleno es normalmente el espacio, pero es `0` si está presente la marca de relleno cero.
- Un punto, que separa el ancho del campo de la precisión.
- Un número, la precisión, que especifica el número máximo de caracteres que se imprimirán a partir de una cadena, o el número de dígitos que se imprimirán después del punto decimal para las `conversiones e`, `E` o `f`, o el número de dígitos significativos para la conversión `g` o `G`, o el número de dígitos que se imprimirán para un número entero (`se agregarán 0 iniciales para completar el ancho necesario`).
- Un modificador de longitud `h`, `l` (letra ell) o `L`. "h" indica que el argumento correspondiente se debe imprimir como `corto` o `corto sin signo`; "l" indica que el argumento es `long` o `unsigned long`, "L" indica que el argumento es un `double largo`.

El ancho o la precisión, o ambos, se pueden especificar como `*`, en cuyo caso el valor se calcula convirtiendo los siguientes argumentos, que deben ser `int`.

Los caracteres de conversión y sus significados se muestran en la Tabla B.1. Si el carácter después de % no es un carácter de conversión, el comportamiento es indefinido.

Tabla B.1 *Conversiones de Printf*

Carácter	Tipo de argumento; Impreso como

d,i	int; Notación decimal con signo.
o	int; Notación octal sin signo (sin cero a la izquierda).
x,x	int sin signo; notación hexadecimal sin signo (sin 0x o 0X inicial), usando abcdef para 0x o ABCDEF para 0X.
u	int; Notación decimal sin signo.
c	int; un solo carácter, después de la conversión a char sin signo
s	char *; Los caracteres de la cadena se imprimen hasta que se alcanza un valor '\0' o hasta que se imprime el número de caracteres indicado por la precisión.
f	doble; notación decimal de la forma [-]*mmm.ddd*, donde el número de *d*'s viene dado por la precisión. La precisión predeterminada es 6; Una precisión de 0 suprime el separador decimal.
e,E	doble; notación decimal de la forma [-]*m.dddddde+/-xx* o [-]*m.ddddddE+/-xx*, donde el número de *d* se especifica mediante la precisión. La precisión predeterminada es 6; Una precisión de 0 suprime el separador decimal.
g,g	doble; %e o %E se utiliza si el exponente es menor que -4 o mayor o igual que la precisión; de lo contrario, se utiliza %f. Los ceros finales y un punto decimal final no se imprimen.
p	nulo *; print como puntero (representación dependiente de la implementación).
n	int *; El número de caracteres escritos hasta el momento por esta llamada a printf se *escribe en* el argumento. No se convierte ningún argumento.
%	no se convierte ningún argumento; Imprime un %

`int printf(const char *format, ...)`
printf(...) es equivalente a fprintf(stdout, ...).`int sprintf(char *s, const char *format, ...)`
sprintf es lo mismo que printf, excepto que la salida se escribe en la cadena s, terminada con '\0'. s debe ser lo suficientemente grande como para contener el resultado. El recuento de valores devueltos no incluye '\0'.

`int vprintf(const char *formato, va_list arg) int vfprintf(ARCHIVO *flujo, const char *formato, va_list arg) int vsprintf(char *s, const char *formato, va_list arg)`
Las funciones vprintf, vfprintf y vsprintf son equivalentes a las funciones printf correspondientes, excepto que la lista de argumentos de la variable se reemplaza por arg, que ha sido inicializado por la macro va_start y quizás va_arg llamadas. Véase la discusión de <stdarg.h> en la sección B.7.

B.1.3 Entrada formateada

La función scanf se ocupa de la conversión de entrada formateada.

`int fscanf(FILE *stream, const char *format, ...)`
fscanf lee de la secuencia bajo el control de format y asigna valores convertidos a través de argumentos posteriores, *cada uno de los cuales debe ser un puntero*. Regresa cuando se agota el formato.fscanf devuelve EOF si se produce el final del fichero o un error antes de cualquier conversión; de lo contrario, devuelve el número de elementos de entrada convertidos y asignados.

La cadena de formato suele contener especificaciones de conversión, que se utilizan para la interpretación directa de la entrada. La cadena de formato puede contener:

- Espacios en blanco o pestañas, que no se ignoran.

- Caracteres ordinarios (no %), que se espera que coincidan con el siguiente carácter de espacio no blanco de la secuencia de entrada.
- Especificaciones de conversión, que constan de un `%`, un carácter de supresión de asignación opcional `*`, un número opcional que especifica un ancho de campo máximo, un h, `l` o `L` opcional que indica el ancho del destino y un carácter de conversión.

Una especificación de conversión determina la conversión del siguiente campo de entrada. Normalmente, el resultado se coloca en la variable señalada por el argumento correspondiente. Sin embargo, si la supresión de la asignación se indica con `*`, como en `%*s`, el campo de entrada simplemente se omite; no se realiza ninguna asignación. Un campo de entrada se define como una cadena de caracteres que no son espacios en blanco; Se extiende hasta el siguiente carácter de espacio en blanco o hasta que se agote el ancho de campo, si se especifica. Esto implica que `scanf` leerá a través de los límites de línea para encontrar su entrada, ya que las nuevas líneas son espacios en blanco. (Los caracteres de espacio en blanco son espacios en blanco, tabulación, nueva línea, retorno de carro, tabulación vertical y avance de formulario).

El carácter de conversión indica la interpretación del campo de entrada. El argumento correspondiente debe ser un puntero. Los caracteres de conversión legal se muestran en la Tabla B.2.

Los caracteres `de conversión d`, `i`, `n`, `o`, `u` y `x` pueden ir precedidos de `h` si el argumento es un puntero a `short` en lugar de int, o de `l` (letra ell) si el argumento es un puntero a `long`. Los caracteres `de conversión e`, `f` y `g` pueden ir precedidos de `l` si hay un puntero a `double` en lugar de `float` en la lista de argumentos, y de `L` si es un puntero a un `double largo`.

***Tabla B.2** Conversiones de Scanf*

Carácter	Datos de entrada; Tipo de argumento
`d`	entero decimal; `Int*`
`Yo`	entero; `int*`. El número entero puede estar en octal (0 inicial) o hexadecimal (0x inicial o `0X`).
`o`	entero octal (con o sin cero a la izquierda); `int *`.
`u`	entero decimal sin signo; `unsigned int *`.
`x`	entero hexadecimal (con o sin 0x o `0X inicial`); `int*`.
`c`	Caracteres; `char*`. Los siguientes caracteres de entrada se colocan en la matriz indicada, hasta el número dado por el campo de ancho; El valor predeterminado es 1. No `se agrega '\0'`. En este caso, se suprime la omisión normal de los caracteres de espacio en blanco; Para leer el siguiente carácter que no sea un espacio en blanco, use `%1s`.
`s`	cadena de caracteres de espacio no en blanco (sin comillas); `char *`, apuntando a una matriz de caracteres lo suficientemente grande como para contener la cadena y una terminación `'\0'` que se agregará.
`e,f,g`	número de coma flotante; `float *`. El formato de entrada para `float` es un signo opcional, una cadena de números que posiblemente contenga un punto decimal y un exponente opcional campo que contiene una `E` o `e` seguida de un número entero posiblemente con signo.
`p`	valor del puntero impreso por `printf("%p");`, `nulo *`.

`n`	escribe en el argumento el número de caracteres leídos hasta el momento por esta llamada; `int *`. No se lee ninguna entrada. El recuento de elementos convertidos no se incrementa.
`[...]`	coincide con la cadena no vacía más larga de caracteres de entrada del conjunto entre corchetes; `char *`. Se agrega un `'\0'` . `[]...]` incluye `]` en el conjunto.

`[^...]`	coincide con la cadena no vacía más larga de caracteres de entrada *que no* pertenecen al conjunto entre corchetes; `char *`. Se agrega un `'\0' .[^]...]` incluye `]` en el conjunto.
`%`	% literal; No se realiza ninguna cesión.

`int scanf(const char *formato, ...)`
`scanf(...)` es idéntico a `fscanf (stdin, ...)`.

`int sscanf(const char *s, const char *formato, ...)`
`sscanf(s, ...)` es equivalente a `scanf(...)` excepto que los caracteres de entrada se toman de la cadena `s`.

B.1.4 Funciones de entrada y salida de caracteres

`int fgetc(ARCHIVO *flujo)`
`fgetc` devuelve el siguiente carácter de `stream` como un `char sin signo` (convertido en un `int`), o `EOF` si se produce el final del archivo o el error.

`char *fgets(char *s, int n, FILE *stream)`
`fgets` lee a lo sumo los siguientes `n-1` caracteres en la matriz `s`, deteniéndose si se encuentra una nueva línea; la nueva línea se incluye en la matriz, que termina en `'\0'`. `fgets` devuelve `s`, o `NULL` si se produce el fin del archivo o el error.

`int fputc(int c, ARCHIVO *flujo)`
`fputc` escribe el carácter `c` (convertido en un `unsigend char`) en la `secuencia`. Devuelve el carácter escrito, o `EOF` para el error.

`int fputs(const char *s, FILE *stream)`
`fputs` escribe la cadena `s` (que no necesita contener `\n`) en `el flujo`; devuelve un valor no negativo, o `EOF` para un error.

`int getc(ARCHIVO *flujo)`
`getc` es equivalente a `fgetc` , excepto que si es una macro, puede evaluar `stream` más de una vez.

`int getchar(vacío)`
`getchar` es equivalente a
`getc(stdin). char *gets(char *s)`
`gets` lee la siguiente línea de entrada en la matriz `s`; reemplaza la nueva línea de terminación con `'\0'`. Devuelve `s` o `NULL` si se produce el final del archivo o un error.

`int putc(int c, ARCHIVO *flujo)`
`putc` es equivalente a `fputc` , excepto que si es una macro, puede evaluar el `flujo` más de una vez.

`int putchar(int c)`
`putchar(c)` es equivalente a
`putc(c,stdout). int puts(const char *s)`
`puts` escribe la cadena `s` y una nueva línea en `stdout`. Devuelve `EOF` si se produce un error, no negativo en caso contrario.

`int ungetc(int c, ARCHIVO *flujo)`
`UNGETC` empuja `C` (convertido en un `carácter sin signo`) de vuelta a la `transmisión`, donde se devolverá en la próxima lectura. Solo se garantiza un carácter de retroceso por transmisión. Es posible que `la EOF` no se retraiga. `ungetc` devuelve el carácter empujado hacia atrás, o `EOF` para el error.

B.1.5 Funciones de entrada y salida directa

`size_t fread(void *ptr, tamaño size_t, size_t nobj, FILE *stream)`
`fread` lee desde stream en la matriz `ptr` en la mayoría de los objetos `nobj` de tamaño `size`. `fread` devuelve el número de objetos leídos, que puede ser menor que el número solicitado. `feof` y `ferror` deben usarse para determinar el estado.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)`

`fwrite` escribe, desde la matriz `ptr`, objetos `nobj` de tamaño size en `el flujo`. Devuelve el número de objetos escritos, que es menor que `nobj` en caso de error.

B.1.6 Funciones de posicionamiento de archivos

`int fseek(ARCHIVO *flujo, desplazamiento largo, origen int)`

`fseek` establece la posición del archivo para el `flujo`; una lectura o escritura posterior accederá a los datos que comiencen en la nueva posición. En el caso de un archivo binario, la posición se establece en `caracteres de desplazamiento` desde el `origen`, que pueden ser `SEEK_SET` (principio), `SEEK_CUR` (posición actual) o `SEEK_END` (fin del archivo). Para una secuencia de texto, `offset` debe ser cero o un valor devuelto por `ftell` (en cuyo caso `origin` debe ser `SEEK_SET`). `fseek` devuelve un valor distinto de cero en caso de error.

`long ftell(ARCHIVO *flujo)`

`ftell` devuelve la posición actual del archivo para stream, o `-1` en caso de error.

`void rewind(ARCHIVO *transmisión)`

`rewind(fp)` es equivalente a `fseek(fp, 0L, SEEK_SET); clearerr(fp).`

`int fgetpos(ARCHIVO *flujo, fpos_t *ptr)`

`fgetpos` registra la posición actual en `el flujo` en `*ptr`, para su uso posterior por `fsetpos`. El tipo `fpos_t` es adecuado para registrar dichos valores. `fgetpos` devuelve un valor distinto de cero en caso de error.

`int fsetpos(ARCHIVO *flujo, const fpos_t *ptr)`

`Las posiciones` de fsetpos `fluyen` en la posición registrada por `fgetpos` en `*ptr`.
`fsetpos`
Devuelve un valor distinto de cero en caso de error.

B.1.7 Funciones de error

Muchas de las funciones de la biblioteca establecen indicadores de estado cuando se produce un error o un fin de archivo. Estos indicadores pueden establecerse y probarse explícitamente. Además, la expresión entera `errno` (declarada en `<errno.h>`) puede contener un número de error que proporciona más información sobre el error más reciente.

`void clearerr(ARCHIVO *flujo)`

`clearerr` borra el final del archivo y los indicadores de error para la `secuencia`. `int feof(ARCHIVO *flujo)`

`feof` devuelve un valor distinto de cero si se establece el indicador de fin de archivo para stream .

`int ferror(ARCHIVO *flujo)`

`ferror` devuelve un valor distinto de cero si se establece el indicador de error para stream .

`void perror(const char *s)`

`perror(s)` imprime `s` y un mensaje de error definido por la implementación correspondiente al entero en `errno`, como si fuera por

```
fprintf(stderr, "%s: %s\n", s, "mensaje de error");
```

Véase `strerror` en la sección B.3.

B.2 Pruebas de clase de caracteres: <ctype.h>

El encabezado `<ctype.h>` declara funciones para probar caracteres. Para cada función, la lista de argumentos es un `int`, cuyo valor debe ser `EOF` o representarse como un `char sin signo`, y el valor devuelto es un `int`. Las funciones devuelven un valor distinto de cero (verdadero) si el argumento `c` cumple la condición descrita, y cero si no es así.

`isalnum(c)` `isalpha(c)` o `isdigit(c)` es true

`isalpha(c)`, `isupper(c)` o `islower(c)` es true

Carácter de control iscntrl(c)
isdigit(c) dígito decimal
isgraph(c) carácter de impresión excepto el espacio
islower(c) letra minúscula
isprint(c) carácter de impresión incluyendo espacio
ispunct(c) carácter de impresión excepto el espacio o la letra o el dígito isspace(c) espacio, avance de formulario, nueva línea, retorno de carro, tabulación, tabulación vertical isupper(c) letra mayúscula
Dígito hexadecimal isxdigit(c)

En el juego de caracteres ASCII de siete bits, los caracteres de impresión son 0x20 (' ') a 0x7E ('~'); los caracteres de control son 0 NUL a 0x1F (US) y 0x7F (DEL).

Además, hay dos funciones que convierten las mayúsculas y minúsculas de las letras:

int tolower(c) convertir c a minúsculas
int toupper(c) convertir c a mayúsculas

Si c es una letra mayúscula, tolower(c) devuelve la letra minúscula correspondiente, toupper(c) devuelve la letra mayúscula correspondiente; de lo contrario, devuelve c.

B.3 Funciones de cadena: <string.h>

Hay dos grupos de funciones de cadena definidas en el encabezado <string.h>. Los primeros tienen nombres que comienzan con str, los segundos tienen nombres que comienzan con mem. A excepción de memmove, el comportamiento es indefinido si la copia tiene lugar entre objetos superpuestos. Las funciones de comparación tratan los argumentos como matrices char sin signo.

En la tabla siguiente, las variables s y t son de tipo char *; cs y ct son de tipo const char *; n es de tipo size_t; y c es un int convertido en char.

char *strcpy(s,ct) copia la cadena ct a la cadena s, incluyendo '\0'; devuelve s.
carbonizar *strncpy(s,ct,n) Copie a lo sumo n caracteres de la cadena ct a s; retorne s. Rellene con '\0's si ct tiene menos de n caracteres.
char *strcat(s,ct) concatenar la cadena ct al final de la cadena s; devuelve s.
carbonizar *strncat(s,ct,n) Concatenar a lo sumo n caracteres de la cadena ct a la cadena s, terminar s con '\0'; devuelve s.
int strcmp(cs,ct) Cadena de comparación Cs a la cadena Ct, devuelve <0 si CS<CT, 0 si cs==ct, o >0 si CS>CT.
int strncmp(cs,ct,n) Compare a lo sumo n caracteres de la cadena cs con la cadena ct; devuelva <0 si cs<ct, 0 si cs==ct, o >0 si cs>ct.
char *strchr(cs,c) devuelve el puntero a la primera aparición de c en cs o NULL si no está presente.
char *strrchr(cs,c) devuelve el puntero a la última aparición de c en cs o NULL si no está presente.
size_t strspn(cs,ct)

Devuelve la longitud del prefijo de `CS` que consta de caracteres en `CT`.

`size_t strcspn(cs,ct)`	Devuelve la longitud del prefijo de `CS` que consta de caracteres *que*
`carbonizar *strpbrk (cs, ct)`	*no* están en `CT`. Puntero de retorno a la primera aparición en la cadena `CS` de cualquier cadena de caracteres `ct`, o `NULL` si no está presente.
`char *strstr(cs,ct)`	devuelve el puntero a la primera aparición de la cadena `ct` en `cs`, o `NULL` si no está presente.
`size_t strlen(cs)`	devuelven la longitud de cs.
`char *strerror(n)`	devuelve el puntero a la cadena definida por la implementación correspondiente al error `n`.
`char *strtok(s,ct)`	`strtok` busca tokens delimitados por caracteres de `ct`; ver más abajo.

Una secuencia de llamadas de `strtok(s,ct)` divide `s` en tokens, cada uno delimitado por un carácter de `ct`. La primera llamada en una secuencia tiene un s no `NULL`, encuentra el primer token en `s` que consta de caracteres que no están en `ct`; lo termina sobrescribiendo el siguiente carácter de `s` con `'\0'` y devuelve un puntero al token. Cada llamada subsiguiente, indicada por un `valor NULL` de `s`, devuelve el siguiente token de este tipo, buscando justo después del final del anterior. `strtok` devuelve `NULL` cuando no se encuentra ningún token adicional. La cadena `ct` puede ser diferente en cada llamada.

El `mem...` Las funciones están pensadas para manipular objetos como matrices de caracteres; la intención es una interfaz para rutinas eficientes. En la tabla siguiente, `s` y `t` son de tipo `void *`; `cs` y `ct` son de tipo `const void *`; `n` es de tipo `size_t`; y `c` es un `int` convertido en un `carácter sin signo`.

`vacío *memcpy(s,ct,n)`	Copie `n` caracteres de `ct` a `s` y devuelva `s`.
`vacío *memmove(s,ct,n)`	Igual que `MEMCPY,` excepto que funciona incluso si los objetos se superponen.
`int memcmp(cs,ct,n)`	compara los primeros `n` caracteres de `cs` con `ct`; devuelve como con `strcmp`.
`vacío *memchr(cs,c,n)`	devuelve el puntero a la primera aparición del carácter `c` en `cs`, o `NULL` si no está presente entre los primeros `n` caracteres.
`void *memset(s,c,n)`	coloca el carácter `c` en los primeros `n` caracteres de `s`, devuelve `s`.

B.4 Funciones matemáticas: <math.h>

El encabezado `<math.h>` declara funciones matemáticas y macros.

Las macros `EDOM` y `ERANGE` (que se encuentran en `<errno.h>`) son constantes integrales distintas de cero que se utilizan para señalar errores de dominio y rango para las funciones; `HUGE_VAL` es un valor doble positivo . Se produce un *error de dominio* si un argumento está fuera del dominio sobre el que se define la función. En un error de dominio, `errno` se establece en `EDOM;` el valor devuelto está definido por la implementación. Se produce un *error de rango* si el resultado de la función no se puede representar como un `doble`. Si el resultado se desborda, la función devuelve `HUGE_VAL` con el signo derecho y `errno` se establece en `ERANGE`. Si el resultado se desborda por debajo, la función devuelve cero; si `errno` se establece en `ERANGE` está definido por la implementación.

En la tabla siguiente, `x` e `y` son de tipo `double`, `n` es un `int` y todas las funciones devuelven `doble`. Los ángulos de las funciones trigonométricas se expresan en radianes.

`sin(x)` seno de *x*

`cos(x)` coseno de *x*

`tan(x)` tangente de *x*

`asin(x)` pecado^{-1}(x) en el rango [-pi/2,pi/2], x en [-1,1].

`acos(x)` cos^{-1}(x) en el rango [0,pi], x en [-1,1]. `atan(x)` bronceado^{-1}(x) en el rango [-pi/2,pi/2]. `atan2(y,x)` bronceado^{-1}(y/x) en el rango [-pi,pi]. `sinh(x)` seno hiperbólico de *x*

`cosh(x)` coseno hiperbólico de *x*

`tanh(x)` tangente hiperbólica de *x*

`exp(x)` Función exponencial *ex*

`registro (x)` logaritmo neperiano ln(x), *x>0.*

`log10(x)` base 10 logaritmo log10(x), *x>0.*

`pow(x,y)` *xy*. Se produce un error de dominio si *x=0* e *y<=0*, o si *x<0* e *y no* son un entero.

`sqrt(x)` raíz cuadrada de x, x>=0.

`ceil(x)` entero más pequeño no menos de `x`, como un `doble`. `Piso(x)` El entero más grande no es mayor que `x`, como un `doble`. `Fabs(x)` Valor absoluto |x|

`ldexp(x,n)` x*2n

`frexp(x, int *ip)` divide *x* en una fracción normalizada en el intervalo [1/2,1) que se devuelve, y una potencia de 2, que se almacena en `*exp`. Si *x* es cero, ambas partes del resultado son cero.

`modf(x, doble *ip)` Divide *x* en partes integrales y fraccionarias, cada una con el mismo signo que *x*. Almacena la parte integral en `*ip` y devuelve la parte fraccionaria.

`fmod(x,y)` resto de coma flotante de *x/y*, con el mismo signo que *x*. Si *y* es cero, el resultado es definido por la implementación.

B.5 Funciones de utilidad: <stdlib.h>

El encabezado `<stdlib.h>` declara funciones para la conversión de números, la asignación de almacenamiento y tareas similares. `Doble ATOF(const char *s)`

`atof` convierte `s` en `double`; es equivalente a `strtod(s, (char**)NULL)`.

`int atoi(const char *s)`

convierte a `int`; es equivalente a `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char *s)`

se convierte en `long`; es equivalente a `strtol(s, (char**)NULL, 10)`.

`Strtod doble(const char *s, char **endp)`

`strtod` convierte el prefijo de `s` en `double`, ignorando el espacio en blanco inicial; almacena un puntero a cualquier sufijo no convertido en `*endp` a menos que `endp` sea `NULL`. Si la respuesta se desborda, se devuelve HUGE_VAL con el signo adecuado; si la respuesta se desborda, se devuelve cero. En cualquier caso, `errno` se establece en `ERANGE`.

`Puntal largo (const cuatro *s, cuatro ** ndp, int base)`

`strtol` convierte el prefijo de `s` en `largo`, ignorando el espacio en blanco inicial; almacena un puntero a cualquier sufijo no convertido en `*endp` a menos que `endp` sea `NULL`. Si `la base` está entre 2

y 36, la conversión se realiza asumiendo que la entrada está escrita en esa base. Si la base es cero, la base es 8, 10 o 16; el 0 inicial implica octal y el 0x o 0X hexadecimal. Las letras en ambos casos representan dígitos desde 10 hasta base 1; se permite un 0x o 0X inicial en base 16. Si la respuesta se desborda, se devuelve LONG_MAX o LONG_MIN, dependiendo del signo del resultado, y errno se establece en ERANGE.

unsigned long strtoul(const char *s, char **endp, int base)

strtoul es lo mismo que strtol, excepto que el resultado es unsigned long y el valor de error es ULONG_MAX.

int rand(vacío)

rand devuelve un entero pseudoaleatorio en el intervalo de 0 a RAND_MAX, que es al menos 32767.

void srand(semilla int sin signo)

SRAND utiliza seed como semilla para una nueva secuencia de números pseudoaleatorios. La semilla inicial es 1.

void *calloc(size_t nobj, tamaño size_t)

calloc devuelve un puntero al espacio para una matriz de objetos nobj, cada uno de tamaño size, o

NULL si no se puede satisfacer la solicitud. El espacio se inicializa en cero bytes.

void *malloc(tamaño size_t)

malloc devuelve un puntero al espacio para un objeto de tamaño size, o NULL si no se puede satisfacer la solicitud. El espacio no está inicializado.

void *realloc(void *p, tamaño size_t)

realloc cambia el tamaño del objeto al que apunta p a size. El contenido no se modificará hasta el mínimo de los tamaños antiguo y nuevo. Si el nuevo tamaño es mayor, el nuevo espacio no se inicializa. realloc devuelve un puntero al nuevo espacio, o NULL si la solicitud no se puede satisfacer, en cuyo caso *p no cambia.

Vacío Libre (Vacío *P)

free desasigna el espacio señalado por p; no hace nada si p es NULL. p debe ser un puntero al espacio asignado previamente por calloc, malloc o realloc.

void abort(vacío)

abort hace que el programa termine de forma anormal, como si fuera por

raise (SIGABRT). void exit(estado int)

exit provoca la terminación normal del programa. Se llama a las funciones ATEXIT en orden inverso al registro, se vacían los archivos abiertos, se cierran las secuencias abiertas y se devuelve el control al entorno. La forma en que se devuelve el estado al entorno depende de la implementación, pero cero se toma como finalización correcta. También se pueden utilizar los valores EXIT_SUCCESS y EXIT_FAILURE.

int atexit(void (*fcn)(void))

atexit registra la función fcn a la que se llamará cuando el programa termine normalmente; devuelve un valor distinto de cero si no se puede realizar el registro.

int system(const char *s)

El sistema pasa la cadena s al entorno para su ejecución. Si s es NULL, el sistema devuelve un valor distinto de cero si hay un procesador de comandos. Si s no es NULL, el valor devuelto depende de la implementación.

Cuatro *Getenov (bajo el nombre de Konst Four*)

getenv devuelve la cadena de entorno asociada con name, o NULL si no existe ninguna cadena. Los detalles dependen de la implementación.

```
void *bsearch(const void *key, const void *base,
              size_t n, tamaño size_t,
              int (*cmp)(const void *keyval, const void *datum))
```

bsearch busca base [0]... base[n-1] para un elemento que coincida con *key. La función cmp debe devolver negativo si su primer argumento (la clave de búsqueda) es menor que el segundo (una entrada de tabla), cero si es igual y positivo si es mayor. Los elementos de la base de la matriz deben estar en orden ascendente. bsearch devuelve un puntero a un elemento coincidente, o NULL si no existe ninguno.

```
void qsort(void *base, size_t n, tamaño size_t,
           int (*cmp)(const void *, const void *))
```

`qsort` ordena en orden ascendente una matriz `base[0]... base[n-1]` de objetos de tamaño
`tamaño`. La función de comparación `cmp` es como en `bsearch`. `int abs(int n)`
`abs` devuelve el valor absoluto de su argumento `int`.
`Laboratorios largos (n larga)`
`labs` devuelve el valor absoluto de su argumento `long`.
`div_t div(int num, int denom)`
`div` calcula el cociente y el resto de `num/denom`. Los resultados se almacenan en el archivo
`miembros int quot` y `rem` de una estructura de tipo `div_t`. `ldiv_t ldiv (número largo, denom largo)`
`ldiv` calcula el cociente y el resto de `num/denom`. Los resultados se almacenan en el archivo
`miembros largos quot` y `rem` de una estructura de tipo `ldiv_t`.

B.6 Diagnóstico: <assert.h>

La macro `assert` se utiliza para agregar diagnósticos a los programas:

```
void assert(expresión int)
```

Si *expression* es cero cuando

```
assert(expresión)
```

se ejecuta, la macro `assert` imprimirá en `stderr` un mensaje, como

`Error de aserción:` *expresión*, `nombre de` *archivo de archivo*, `línea` *nnn*

A continuación, llama `a abort` para finalizar la ejecución. El nombre de archivo de origen y el número de línea provienen de las macros de preprocesador `FILE` y `LINE` .

Si `NDEBUG` se define en el momento en que `se incluye <assert.h>`, se omite la macro assert.

B.7 Listas de argumentos de variables: <stdarg.h>

El encabezado `<stdarg.h>` proporciona facilidades para recorrer una lista de argumentos de función de número y tipo desconocidos.

Supongamos que `lastarg` es el último parámetro nombrado de una función `f` con un número variable de argumentos. A continuación, declara dentro `de f` una variable de tipo `va_list` que apuntará a cada argumento a su vez:

```
   va_list AP;
```

`ap` debe inicializarse una vez con la va_start de macro antes de acceder a cualquier argumento sin nombre:

```
va_start(va_list AP, lastarg);
```

A partir de entonces, cada ejecución de la va_arg de macro producirá un valor que tiene el tipo y el valor del siguiente argumento sin nombre, y también modificará `ap` para que el próximo uso de `va_arg` devuelva el siguiente argumento:

Tipo `va_arg(va_list AP,` *tipo*`);`

La macro

```
nulo va_end(va_list AP);
```

Se debe llamar una vez después de que se hayan procesado los argumentos, pero antes `de que se salga` de f.

B.8 Saltos no locales: <setjmp.h>

Las declaraciones de `<setjmp.h>` proporcionan una manera de evitar la llamada a la función normal y la secuencia de retorno, normalmente para permitir un retorno inmediato de una llamada de función profundamente anidada.

```
int setjmp(jmp_buf env)
```

La macro `setjmp` guarda la información de estado en `env` para su uso por `longjmp`. El retorno es cero de una llamada directa de `setjmp`, y distinto de cero de una llamada posterior de `longjmp`. Una llamada a `setjmp` solo puede ocurrir en ciertos contextos, básicamente la prueba de `if`, `switch` y loops, y solo en expresiones relacionales simples.

```
if (setjmp(env) == 0)
    /* Llegar aquí en llamada
directa */ De lo contrario
    /* llegar aquí llamando a longjmp
*/ void longjmp(jmp_buf env, int val)
```

`longjmp` restaura el estado guardado por la llamada más reciente a `setjmp`, utilizando la información guardada en `env`, y la ejecución se reanuda como si la `función setjmp` acabara de ejecutarse y devolviera el valor val distinto de cero. La función que contiene el `setjmp` no debe haber terminado. Los objetos accesibles tienen los valores que tenían en el momento en `que se llamó a` longjmp, excepto que las variables automáticas no `volátiles` en la llamada a `la función setjmp` se vuelven indefinidas si se cambiaron después de la llamada a `setjmp`.

B.9 Señales: <signal.h>

El encabezado `<signal.h>` proporciona facilidades para manejar condiciones excepcionales que surgen durante la ejecución, como una señal de interrupción de una fuente externa o un error en la ejecución.

```
void (*signal(int sig, void (*handler)(int)))(int)
```

`signal` determina cómo se manejarán las señales subsiguientes. Si handler es `SIG_DFL`, se utiliza el comportamiento predeterminado definido por la implementación, si es `SIG_IGN`, se ignora la señal; de lo contrario, se llamará a la función señalada por handler , con el argumento del tipo de señal. Las señales válidas incluyen:

`SIGABRT` terminación anormal, por ejemplo, de `abortar SIGFPE` Error aritmético, por ejemplo, división por cero o desbordamiento `SIGILL` imagen de función ilegal, por ejemplo, instrucción ilegal `SIGINT` atención interactiva, p. ej., interrumpir

`SIGSEGV` Acceso ilegal al almacenamiento, por ejemplo, acceso fuera de los

límites de memoria

`SIGTERM` Solicitud de terminación enviada a este programa

`signal` devuelve el valor anterior de `handler` para la señal específica, o `SIG_ERR` si se produce un error.

Cuando se produce una señal `sig` posteriormente, la señal se restaura a su comportamiento predeterminado; entonces se llama a la función de manejo de señales, como si fuera por `(*handler)(sig)`. Si el controlador vuelve, la ejecución se reanudará donde estaba cuando se produjo la señal.

El estado inicial de las señales está definido por la implementación.

`int raise(int sig)`
`raise` envía la señal `sig` al programa; devuelve un valor distinto de cero si no tiene éxito.

B.10 Funciones de fecha y hora: <time.h>

El encabezado `<time.h>` declara tipos y funciones para manipular la fecha y la hora. Algunas funciones procesan *la hora local*, que puede diferir de la hora del calendario, por ejemplo, debido a la zona horaria. `clock_t` y `time_t` son tipos aritméticos que representan tiempos, y `struct tm` contiene los componentes de una hora de calendario:

`int tm_sec;` segundos después del minuto (0,61) `int tm_min;` minutos después de la hora (0,59) `int tm_hour;` horas desde la medianoche (0,23) `int tm_mday;` Día del mes (1,31)
`int tm_mon;` Meses *desde* Enero (0,11)
`int tm_year;` años desde 1900
`int tm_wday;` días desde el domingo (0,6)
`int tm_yday;` días desde el 1 de enero (0,365)
`int tm_isdst;` Indicador de horario de verano

`tm_isdst` es positivo si el horario de verano está en vigor, cero si no es así y negativo si la información no está disponible.

`clock_t reloj (vacío)`
`clock` devuelve el tiempo de procesador utilizado por el programa desde el comienzo de la ejecución, o `-1` si no está disponible. `clock()/CLK_PER_SEC` es un tiempo en segundos.

`time_t tiempo(time_t *tp)`
`time` devuelve la hora actual del calendario o `-1` si la hora no está disponible. Si `tp` no es `NULL`, el valor devuelto también se asigna a `*tp`. `double difftime(time_t tiempo2, time_t tiempo1)`
`difftime` devuelve `time2-time1` expresado en segundos.

`time_t mktime(struct tm *tp)`
`mktime` convierte la hora local en la estructura `*tp` en hora de calendario en la misma representación utilizada por time. Los componentes tendrán valores en los rangos que se muestran. `mktime` devuelve la hora del calendario o `-1` si no se puede representar.

Las siguientes cuatro funciones devuelven punteros a objetos estáticos que pueden ser sobrescritos por otras llamadas.

`char *asctime(const struct tm *tp)`

asctime</tt< convierte el tiempo en la estructura *tp en una cadena de la forma

```
Sun Jan 3 15:14:13 1988\n\0
```

```
char *ctime(const time_t *tp)
```

ctime convierte la hora del calendario *tp a la hora local; Es equivalente a

```
asctime(localtime(tp))
```

```
struct tm *gmtime(const time_t *tp)
```

gmtime convierte la hora del calendario *tp en Tiempo Universal Coordinado (UTC). Devuelve NULL si UTC no está disponible. El nombre gmtime tiene un significado histórico.

```
struct tm *localtime(const time_t *tp)
```

localtime convierte la hora del calendario *tp en hora local.

```
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
```

strftime da formato a la información de fecha y hora de *tp a s de acuerdo con fmt, que es análogo a un formato printf. Los caracteres ordinarios (incluida la terminación '\0') se copian en s. Cada %c se reemplaza como se describe a continuación, utilizando valores adecuados para el entorno local. No se colocan más de smax caracteres en s. strftime devuelve el número de caracteres, excluyendo '\0', o cero si se produjeron más de smax caracteres.

- %a Nombre abreviado del día de la semana.
- %A Nombre completo del día de la semana.
- %b Nombre abreviado del mes.
- %B Nombre completo del mes.
- %c Representación de fecha y hora local.
- %d Día del mes (01-31).
- %H hora (reloj de 24 horas) (00-23).
- %I hour (reloj de 12 horas) (01-12).
- %j Día del año (001-366).
- %m mes (01-12).
- %M minuto (00-59).
- %p equivalente local de AM o PM.
- %S segundo (00-61).
- %U número de semana del año (domingo como 1er día de la semana) (00-53).
- %w día de la semana (0-6, el domingo es 0).
- %W número de semana del año (lunes como 1er día de la semana) (00-53).
- %x Representación de fecha local.
- %X Representación de la hora local.
- %y año tras siglo (00-99).
- %Y año con siglo.
- %Z Nombre de la zona horaria, si la hay.
- %% %

B.11 Límites definidos por la implementación: <limits.h> y <float.h>

El encabezado <limits.h> define constantes para los tamaños de los tipos enteros. Los valores a continuación son magnitudes mínimas aceptables; Se pueden utilizar valores más grandes.

CHAR_BIT	8	bits en un carácter
CHAR_MAX	UCHAR_MAX *o* SCHAR_MAX	Valor máximo de char
CHAR_MIN	0 *o* SCHAR_MIN	Valor máximo de char
INT_MAX	32767	Valor máximo de int
INT_MIN	-32767	Valor mínimo de int
LONG_MAX	2147483647	Valor máximo de long
LONG_MIN	-2147483647	Valor mínimo de long
SCHAR_MAX	+127	Valor máximo de char firmado
SCHAR_MIN	-127	Valor mínimo de char firmado
SHRT_MAX	+32767	Valor máximo de corto
SHRT_MIN	-32767	Valor mínimo de corto
UCHAR_MAX	255	Valor máximo de unsigned char
UINT_MAX	65535	Valor máximo de unsigned int
ULONG_MAX	4294967295	Valor máximo de unsigned long
USHRT_MAX	65535	Valor máximo de la posición corta sin firmar

Los nombres de la tabla siguiente, un subconjunto de <float.h>, son constantes relacionadas con la aritmética de punto flotante. Cuando se da un valor, representa la magnitud mínima para la cantidad correspondiente. Cada implementación define los valores adecuados.

FLT_RADIX	2	base de exponente, representación, por ejemplo, 2, 16
FLT_ROUNDS		Modo de redondeo de coma flotante para la adición
FLT_DIG	6	Dígitos decimales de precisión
FLT_EPSILON	1E-5	Número más pequeño *x* tal que 1.0+x != 1.0
FLT_MANT_DIG		Número de base FLT_RADIX En Mantissa
FLT_MAX	1E+37	Número máximo de coma flotante
FLT_MAX_EXP		máximo *n* de tal manera que FLT_RADIXn-1 es representable
FLT_MIN	1E-37	Número mínimo de punto flotante normalizado
FLT_MIN_EXP		mínimo *n* de tal manera que 10n es un número normalizado
DBL_DIG	10	Dígitos decimales de precisión
DBL_EPSILON	1E-9	Número más pequeño *x* tal que 1.0+x != 1.0
DBL_MANT_DIG		Número de base FLT_RADIX En Mantissa
DBL_MAX	1E+37	máximo doble Número de coma flotante
DBL_MAX_EXP		máximo *n* de tal manera que FLT_RADIXn-1 es representable
DBL_MIN	1E-37	Normalizado mínimo doble Número de coma flotante
DBL_MIN_EXP		mínimo *n* de tal manera que 10n es un número normalizado

Apéndice C - Resumen de los cambios

Desde la publicación de la primera edición de este libro, la definición del lenguaje C ha sufrido cambios. Casi todos eran extensiones del idioma original, y fueron cuidadosamente diseñados para que siguieran siendo compatibles con la práctica existente; algunas ambigüedades reparadas en la descripción original; y algunos representan modificaciones que cambian la práctica existente. Muchas de las nuevas facilidades fueron anunciadas en los documentos que acompañan a los compiladores disponibles en AT&T, y posteriormente han sido adoptadas por otros proveedores de compiladores de C. Más recientemente, el comité ANSI que estandarizó el lenguaje incorporó la mayoría de los cambios y también introdujo otras modificaciones significativas. Su informe fue en parte participado por algunos compiladores comerciales incluso antes de la publicación del estándar formal C.

Este Apéndice resume las diferencias entre el lenguaje definido por la primera edición de este libro, y el que se espera que sea definido por la norma final. Trata sólo de la lengua en sí, no de su entorno y biblioteca; Aunque estos son una parte importante de la norma, hay poco con lo que comparar, porque la primera edición no intentó prescribir un entorno o una biblioteca.

- El preprocesamiento se define más cuidadosamente en el estándar que en la primera edición, y se amplía: se basa explícitamente en tokens; hay nuevos operadores para la concatenación de tokens (`##`) y la creación de cadenas (`#`); hay nuevas líneas de control como `#elif` y `#pragma`; se permite explícitamente la redeclaración de macros por la misma secuencia de tokens; los parámetros dentro de las cadenas ya no se reemplazan. El empalme de líneas por \ está permitido en todas partes, no solo en cadenas y definiciones de macros. Véase el párrafo A.12.
- La significación mínima de todos los identificadores internos aumentó a 31 caracteres; La significación obligatoria más pequeña de los identificadores con vinculación externa sigue siendo de 6 letras monominúsculas. (Muchas implementaciones proporcionan más).
- Secuencias de trígrafas introducidas por `??` Permiten la representación de caracteres que faltan en algunos conjuntos de caracteres. Se definen los escapes para `#\^[]{}|~` , véase Par.A.12.1. Observe que la introducción de trígrafos puede cambiar el significado de las cadenas que contienen la secuencia `??`.
- Se introducen nuevas palabras clave (`void, const, volatile, signed, enum`). Se retira la `palabra clave` de entrada nacida muerta.
- Se definen nuevas secuencias de escape, para su uso dentro de constantes de caracteres y literales de cadena. El efecto de seguir \ por un carácter que no forma parte de una secuencia de escape aprobada es indefinido. Véase el párrafo A.2.5.2.
- El cambio trivial favorito de todos: `8` y `9` no son dígitos octales.
- El estándar introduce un conjunto más grande de sufijos para hacer explícito el tipo de constantes: `U` o `L` para enteros, `F` o `L` para flotantes. También refina las reglas para el tipo de constantes no fijas (Par.A.2.5).
- Los literales de cadena adyacentes se concatenan.
- Hay una notación para los literales de cadena de caracteres anchos y las constantes de caracteres; véase Par.A.2.6.
- Los caracteres, así como otros tipos, se pueden declarar explícitamente para llevar, o no llevar, un signo mediante las palabras clave `signed` o `unsigned`. Se retira la locución `long float` como sinónimo de `double` , pero `long double` se puede usar para declarar una cantidad flotante de precisión extra.
- Durante algún tiempo, el tipo `unsigned char` ha estado disponible. La norma introduce la
  `signed` para hacer explícita la signedness para `char` y otros objetos integrales.

- El tipo `void` ha estado disponible en la mayoría de las implementaciones durante algunos años. El estándar introduce el uso del `tipo void *` como un tipo de puntero genérico; anteriormente `char *` desempeñaba este papel. Al mismo tiempo, se promulgan reglas explícitas contra la mezcla de punteros y enteros, y punteros de diferentes tipos, sin el uso de conversiones.
- La Norma establece mínimos explícitos en los rangos de los tipos aritméticos y ordena encabezados (`<limits.h>` y `<float.h>`) que indican las características de cada implementación en particular.
- Las enumeraciones son nuevas desde la primera edición de este libro.
- El estándar adopta de C++ la noción de calificador de tipos, por ejemplo, `const` (Párr. A.8.2).
- Las cadenas ya no se pueden modificar y, por lo tanto, se pueden colocar en la memoria de solo lectura.
- Las "conversiones aritméticas usuales" se cambian, esencialmente de "para los enteros, `unsigned` siempre gana; para el punto flotante, siempre use `double`" para "promover al tipo lo suficientemente espacioso y más pequeño". Véase el párrafo A.6.5.
- Los antiguos operadores de asignación como `=+` realmente han desaparecido. Además, los operadores de asignación ahora son tokens únicos; En la primera edición, eran pares y podían estar separados por un espacio en blanco.
- Se revoca la licencia de un compilador para tratar operadores matemáticamente asociativos como computacionalmente asociativos.
- Se introduce un `operador unario +` para la simetría con unario -.
- Un puntero a una función se puede usar como designador de función sin un `* explícito` operador. Véase el párrafo A.7.3.2.
- Las estructuras pueden ser asignadas, pasadas a funciones y devueltas por funciones.
- Se permite aplicar el operador address-of a las matrices, y el resultado es un puntero a la matriz.
- El operador `sizeof`, en la primera edición, producía el tipo `int`; posteriormente, muchas implementaciones lo hicieron `sin signo`. La norma hace que su tipo dependa explícitamente de la implementación, pero requiere que el tipo, `size_t`, se defina en un encabezado estándar (`<stddef.h>`). Un cambio similar se produce en el tipo (`ptrdiff_t`) de la diferencia entre punteros. Véanse los párrafos A.7.4.8 y A.7.7.
- El operador de dirección & no se puede aplicar a un registro declarado de objeto, incluso si la implementación elige no mantener el objeto en un registro.
- El tipo de expresión de desplazamiento es el del operando izquierdo; el operando derecho no puede promover el resultado. Véase el párrafo A.7.8.
- La Norma legaliza la creación de un puntero justo más allá del final de una matriz, y permite la aritmética y las relaciones en él; véase Par.A.7.7.
- La norma introduce (tomando prestada de C++) la noción de una declaración de prototipo de función que incorpora los tipos de los parámetros, e incluye un reconocimiento explícito de las funciones variádicas junto con una forma aprobada de tratarlas. Véase Pars. A.7.3.2, A.8.6.3, B.7. El estilo más antiguo todavía se acepta, con restricciones.
- Las declaraciones vacías, que no tienen declaradores y no declaran al menos una estructura, unión o enumeración, están prohibidas por el estándar. Por otro lado, una declaración con solo una etiqueta de estructura o unión vuelve a declarar esa etiqueta incluso si se declaró en un ámbito externo.
- Se prohíben las declaraciones de datos externas sin especificadores o calificadores (solo un declarador desnudo).
- Algunas implementaciones, cuando se presentan con una `declaración extern` en un bloque interno, exportarían la declaración al resto del archivo. La Norma deja claro que el alcance de dicha declaración es solo el bloque.
- El ámbito de los parámetros se inserta en la instrucción compuesta de una función, de modo que las declaraciones de variables en el nivel superior de la función no pueden ocultar los parámetros.

- Los espacios de nombres de los identificadores son algo diferentes. La Norma coloca todas las etiquetas en un único espacio de nombres y también introduce un espacio de nombres separado para las etiquetas; véase el párrafo A.11.1. Además, los nombres de los miembros están asociados con la estructura o unión de la que forman parte (esta ha sido una práctica común desde hace algún tiempo).
- Las uniones pueden inicializarse; El inicializador hace referencia al primer miembro.
- Las estructuras, uniones y matrices automáticas se pueden inicializar, aunque de forma restringida.
- Las matrices de caracteres con un tamaño explícito se pueden inicializar mediante un literal de cadena con exactamente ese número de caracteres (el `\0` se exprime silenciosamente).
- La expresión de control, y las etiquetas de caso, de un conmutador pueden tener cualquier entero
  tipo.