Lenguaje de Programación C

1 El primer ejemplo

Este capítulo presenta el código fuente de un programa en C muy simple y lo utiliza para explicar algunas características del lenguaje. Si ya conoce los puntos básicos de C presentados en este capítulo, puede hojearlo o salteárselo.

Presentamos ejemplos de código fuente en C (excepto comentarios) utilizando una fuente de ancho fijo, ya que así es como se ven cuando los edita en un editor como GNU Emacs.

1.1 Ejemplo: Fibonacci recursivo

Para presentar las características más básicas de C, veamos el código de una función matemática simple que realiza cálculos con números enteros. Esta función calcula el número n de la serie de Fibonacci, en la que cada número es la suma de los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```
int
fib (int n)
{
    if (n <= 2) /* Esto evita la recursión infinita. */
        return 1;
    else
        return fib (n - 1) + fib (n - 2);
}
```

Este programa muy simple ilustra varias características de C:

- Una definición de función, cuyas primeras dos líneas constituyen el encabezado de la función. Consulte la Sección 22.1 [Definiciones de funciones], página 132.
- Un parámetro de función `n`, al que se hace referencia como la variable `n` dentro del cuerpo de la función. Consulte la Sección [Variables de parámetros de función]. Una definición de función utiliza parámetros para hacer referencia a los valores de los argumentos proporcionados en una llamada a esa función.
- Aritmética. Los programas de C suman con '+' y restan con '-'. Consulte [Aritmética].
- Comparaciones numéricas. El operador '<=' prueba "menor o igual que". Consulte [Comparaciones numéricas].
- Constantes enteras escritas en base 10. Consulte [Constantes enteras].
- Una llamada a una función. La llamada a una función `fib (n - 1)` llama a la función `fib`, pasando como argumento el valor `n - 1`. Consulte [Llamadas a funciones].
- Un comentario, que comienza con '`/*`' y termina con '`*/`'. El comentario no tiene ningún efecto sobre la ejecución del programa. Su propósito es proporcionar explicaciones a las personas que leen el código fuente. Incluir comentarios en el código es tremendamente importante: proporcionan información de fondo para que otros puedan entender el código más rápidamente. Consulte [Comentarios].

- En este manual, presentamos el texto de los comentarios en la tipografía de ancho variable utilizada para el texto de los capítulos, no en la tipografía de ancho fijo utilizada para el resto del código. Esto es para que los comentarios sean más fáciles de leer. Esta distinción de tipografía no existe en un archivo real de código fuente de C.
- Dos tipos de instrucciones, la instrucción `return` y la instrucción `if...else`. Consulte [Instrucciones].
- Recursión. La función `fib` se llama a sí misma; esto se denomina *llamada recursiva*. Estas son válidas en C y bastante comunes.
- La función `fib` no sería útil si no retornara. Por lo tanto, las definiciones recursivas, para ser de alguna utilidad, deben evitar la *recursión infinita*.

  Esta definición de función evita la recursión infinita al manejar especialmente el caso en el que n es dos o menos. Por lo tanto, la profundidad máxima de las llamadas recursivas es menor que `n`.

1.1.1 Encabezado de la función

En nuestro ejemplo, las dos primeras líneas de la definición de la función son el *encabezado / header*. Su propósito es indicar el nombre de la función y decir cómo se llama:

```
int
fib (int n)
```

dice que la función retorna un entero (tipo `int`), su nombre es `fib` y toma un argumento llamado `n` que también es un entero. (Los tipos de datos se explicarán más adelante, en [Tipos primitivos].

1.1.2 Cuerpo de la función

El resto de la definición de la función se llama `cuerpo de la función / function body`. Como todo cuerpo de función, este comienza con '{', termina con '}' y contiene cero o más instrucciones y declaraciones. Las sentencias especifican acciones a realizar, mientras que las declaraciones definen nombres de variables, funciones, etc. Cada sentencia y cada declaración termina con un punto y coma (';').

Las sentencias y declaraciones a menudo contienen expresiones; una expresión es una construcción cuya ejecución produce un valor de algún tipo de datos, pero también puede realizar acciones a través de "efectos secundarios" que alteran la ejecución posterior. Una sentencia, por el contrario, no tiene un valor; afecta la ejecución posterior del programa solo a través de las acciones que realiza.

Este cuerpo de función no contiene declaraciones, y solo una sentencia, pero esa es una sentencia compleja porque contiene sentencias anidadas. Esta función usa dos tipos de sentencias:

`return`

La sentencia `return` hace que la función regrese inmediatamente. Se ve así: `return value;` Su significado es calcular el valor de la expresión y salir de la función, haciendo que devuelva cualquier valor que haya producido la expresión. Por ejemplo, `return 1;` devuelve el entero 1 de la función, y `return fib (n - 1) + fib (n - 2);` devuelve un valor calculado al realizar dos llamadas de función como se especifica y sumar sus resultados.

`if...else`

La sentencia `if...else` es una condicional. Cada vez que se ejecuta, elige una de sus dos subdeclaraciones para ejecutar e ignora la otra. Se ve así:

```
if (condición)
    sentencia-if-true
else
    sentencia-if-false
```

Su significado es calcular la expresión *condición* y, si es "verdadera", ejecutar la `sentencia-if-true`. De lo contrario, ejecutar la `sentencia-if-false`. Consulte [Sentencia if-else].

Dentro de la sentencia `if...else`, *condición* es simplemente una expresión. Se considera "verdadera" si su valor es distinto de cero. (Una operación de comparación, como `n <= 2`, produce el valor **1** si es "verdadera" y **0** si es "falsa". Consulte [Comparaciones numéricas]). Por lo tanto,

```
if (n <= 2)
    return 1;
else
    return fib (n - 1) + fib (n - 2);
```

primero prueba si el valor de `n` es menor o igual a 2. Si es así, la expresión `n <= 2` tiene el valor 1. Por lo tanto, la ejecución continúa con la instrucción

```
return 1;
```

De lo contrario, la ejecución continúa con esta instrucción:

```
return fib (n - 1) + fib (n - 2);
```

Cada una de estas instrucciones finaliza la ejecución de la función y proporciona un valor para que esta retorne. Consulte [Instrucción return].

El cálculo de `fib` utilizando números enteros ordinarios en C funciona solo para `n < 47`, porque el valor de `fib (47)` es demasiado grande para caber en el tipo `int`. La operación de suma que intenta sumar `fib (46)` y `fib (45)` no puede entregar el resultado correcto. Esta ocurrencia se llama *desbordamiento de entero*.

El desbordamiento puede manifestarse de varias maneras, pero una cosa que no puede suceder es que se produzca el valor correcto, ya que no cabe en el espacio para el valor. Consulte [Desbordamiento de enteros].

Consulte [Funciones], para obtener una explicación completa sobre las funciones.

1.2 La pila y el desbordamiento de pila

La recursión tiene un inconveniente: existen límites en cuanto a la cantidad de niveles anidados de llamadas a funciones que puede realizar un programa. En C, cada llamada a función asigna un bloque de memoria que utiliza hasta que la llamada retorna. C asigna estos bloques consecutivamente dentro de una gran área de memoria conocida como la pila, por lo que nos referimos a los bloques como *marcos de pila / stack frames*.

El tamaño de la pila es limitado; si el programa intenta usar demasiado, eso hace que el programa falle porque la pila está llena. Esto se llama *desbordamiento de pila / stack ovetflow*.

El desbordamiento de pila en GNU/Linux generalmente se manifiesta como la señal denominada `SIGSEGV`, también conocida como "error de segmentación". Por defecto, esta señal finaliza el programa inmediatamente, en lugar de dejar que el programa intente recuperarse o alcanzar un punto final esperado. (En este caso, solemos decir que el programa "se bloquea"). Consulte [Señales].

Es inconveniente observar un bloqueo al pasar un argumento demasiado grande a Fibonacci recursivo, porque el programa se ejecutaría durante mucho tiempo antes de bloquearse. Este algoritmo es simple pero ridículamente lento: al calcular `fib(n)`, la cantidad de llamadas (recursivas) `fib(1)` o `fib(2)` que realiza es igual al resultado final.

Sin embargo, puede observar un desbordamiento de pila muy rápidamente si usa esta función en su lugar:

```
int
fill_stack (int n)
{
    if (n <= 1) /* Esto limita la profundidad de la recursión. */
       return 1;
    else
       return fill_stack (n - 1);
}
```

En gNewSense GNU/Linux en el Lemote Yeeloong, sin optimización y utilizando la configuración predeterminada, un experimento demostró que hay suficiente espacio en la pila para realizar 261906 llamadas anidadas a esa función. Una más y la pila se desborda y el programa se bloquea. En otra plataforma, con una configuración diferente o con una función diferente, el límite puede ser mayor o menor.

1.3 Ejemplo: Fibonacci iterativo

A continuación se muestra un algoritmo mucho más rápido para calcular la misma serie de Fibonacci. Es más rápido por dos razones. En primer lugar, utiliza *iteración* (es decir, repetición o bucle) en lugar de recursión, por lo que no lleva tiempo realizar una gran cantidad de llamadas a funciones. Pero principalmente, es más rápido porque la cantidad de repeticiones es pequeña: solo n.

```
int
fib (int n)
{
   int last = 1;     /* El valor inicial es fib (1). */
   int prev = 0;     /* El valor inicial controla fib (2). */
   int i;

   for (i = 1; i < n; ++i)
      /* Si n es 1 o menor, el bucle se ejecuta cero veces, */
      /* ya que i < n es falso la primera vez. */
      {
         /* Ahora last es fib (i)
            y prev es fib (i  1). */
         /* Calcula fib (i + 1). */
         int next = prev + last;
         /* Desplaza los valores hacia abajo. */
         prev = last;
         last = next;
         /* Ahora last es fib (i + 1)
            y prev es fib (i).
            Pero eso no será así por mucho tiempo,
```

```
            porque estamos a punto de incrementar i. */
      }
   return last;
}
```

Esta definición calcula `fib (n)` en un tiempo proporcional a `n`. Los comentarios en la definición explican cómo funciona: avanza a través de la serie, siempre mantiene los dos últimos valores en `last` y `prev`, y los suma para obtener el siguiente valor.

Estas son las características adicionales de C que utiliza esta definición:

Bloques internos

El cuerpo de la función también cuenta como un bloque, por lo que puede contener declaraciones y declaraciones. Consulte [Bloques].]

Declaraciones de variables locales

Este cuerpo de función contiene declaraciones y declaraciones. Hay tres declaraciones directamente en el cuerpo de la función, así como una cuarta declaración en un bloque interno. Cada una comienza con `int` porque declara una variable cuyo tipo es entero. Una declaración puede declarar varias variables, pero cada una de estas declaraciones es simple y declara solo una variable. Las variables declaradas dentro de un bloque (ya sea un cuerpo de función o un bloque interno) son `variables locales`. Estas variables existen solo dentro de ese bloque; sus nombres no se definen fuera del bloque y al salir del bloque se desasigna su almacenamiento. Este ejemplo declara cuatro variables locales: `last`, `prev`, `i` y `next`. La declaración de variable local más básica se ve así: `type variablename;` Por ejemplo, `int i;` declara la variable local `i` como un entero. Consulte [Declaraciones de variables].

Inicializadores

Cuando declara una variable, también puede especificar su valor inicial, de esta manera: `type variablename = value;` Por ejemplo, `int last = 1;` declara la variable local `last` como un entero (tipo `int`) y la inicia con el valor 1. Consulte [Inicializadores].

Asignación

`Asignación`: un tipo específico de expresión, escrita con el operador '=', que almacena un nuevo valor en una variable u otro lugar. Por lo tanto, `variable = value` es una expresión que calcula `value` y almacena el valor en variable. Consulte [Expresiones de asignación].

Declaraciones de expresión

Una declaración de expresión es una expresión seguida de un punto y coma. Esto calcula el valor de la expresión y luego ignora el valor. Una declaración de expresión es útil cuando la expresión cambia algunos datos o tiene otros efectos secundarios, por ejemplo, con llamadas de función o con asignaciones como en este ejemplo. Consulte [Declaración de expresión]. El uso de una expresión sin efectos secundarios en una declaración de expresión no tiene sentido, excepto en casos muy especiales. Por ejemplo, la declaración de expresión `x;` examinaría el valor de `x` y lo ignoraría. Eso no es útil.

Operador de incremento

El operador de incremento es '++'. `++i` es una expresión que es la abreviatura de `i = i + 1`. Consulte [Incremento/Decremento].

Declaraciones for

Una declaración `for` es una forma clara de ejecutar una declaración repetidamente: un bucle (*loop*) (consulte [Declaraciones de bucle]). Específicamente,

```
for (i = 1; i < n; ++i)
    body
```

significa comenzar haciendo `i = 1` (establecer `i` en uno) para prepararse para el bucle. El bucle en sí consiste en

- Probando `i < n` y saliendo del bucle si es falso.
- Ejecutando `body`.
- Avanzando el bucle (ejecutando `++i`, que incrementa `i`).

El resultado neto es ejecutar `body` con 1 en `i`, luego con 2 en `i`, y así sucesivamente, deteniéndose justo antes de la repetición donde `i` sería igual a `n`. Si `n` es menor que 1, el bucle ejecutará `body` cero veces.

El cuerpo de la declaración `for` debe ser una y solo una declaración. No puedes escribir dos declaraciones seguidas allí; si lo intentas, solo la primera de ellas será tratada como parte del bucle.

La forma de poner múltiples declaraciones en un lugar así es agruparlas con un bloque, y eso es lo que hacemos en este ejemplo.

2 Un programa completo

Está muy bien escribir una función de Fibonacci, pero no se puede ejecutar por sí sola. Es un programa útil, pero no es un programa completo.

En este capítulo presentamos un programa completo que contiene la función `fib`. Este ejemplo muestra cómo hacer que el programa comience, cómo hacer que finalice, cómo hacer los cálculos y cómo imprimir un resultado.

2.1 Ejemplo de programa completo

A continuación se muestra el programa completo que utiliza la versión recursiva simple de la función `fib` (consulte [Fibonacci recursivo]):

```
#include <stdio.h>

int
fib (int n)
{
   if (n <= 2) /* Esto evita la recursión infinita.. */
      return 1;
   else
      return fib (n - 1) + fib (n - 2);
}

int
main (void)
```

```
{
   printf ("El elemento %d de la serie de Fibonacci es %d\n", 20, fib (20));
   return 0;
}
```

Este programa imprime un mensaje que muestra el valor de `fib (20)`.

Ahora, una explicación de lo que significa ese código.

2.2 Explicación completa del programa

Este programa de ejemplo imprime un mensaje que muestra el valor de `fib (20)` y termina con el código 0 (que indica una ejecución exitosa).

Todo programa en C se inicia ejecutando la función denominada `main`. Por lo tanto, el programa de ejemplo define una función denominada `main` para proporcionar una forma de iniciarlo. Lo que haga esa función es lo que hace el programa. Consulte [La función main].

La función `main` es la primera que se llama cuando se ejecuta el programa, pero no aparece primero en el código de ejemplo. El orden de las definiciones de funciones en el código fuente no afecta el significado del programa.

La llamada inicial a `main` siempre pasa ciertos argumentos, pero `main` no tiene que prestarles atención. Para ignorar esos argumentos, defina `main` con `void` como lista de parámetros. (`void` como lista de parámetros de una función normalmente significa "*llamada sin argumentos*", pero `main` es un caso especial.)

La función `main` devuelve 0 porque esa es la forma convencional en que `main` indica una ejecución exitosa. En cambio, podría devolver un entero positivo para indicar un error, y algunos programas de utilidad tienen convenciones específicas para el significado de ciertos códigos numéricos de error. Vea [Valores de main].

La forma más simple de imprimir texto en C es llamando a la función `printf`, por lo que aquí explicamos muy brevemente lo que hace esa función. Para una explicación completa de `printf` y las otras funciones estándar de E/S, vea "E/S en flujos" en The GNU C Library Reference Manual.

El primer argumento de `printf` es una constante de cadena (vea [Constantes de cadena]) que es una plantilla para la salida. La función `printf` copia la mayor parte de esa cadena directamente como salida, incluido el carácter de nueva línea al final de la cadena, que se escribe como '$\backslash n$'. La salida va al destino de salida estándar del programa, que en el caso habitual es la terminal.

'%' en la plantilla introduce un código que sustituye otro texto en la salida. Específicamente, '%d' significa tomar el siguiente argumento de `printf` y sustituirlo en el texto como un número decimal. (El argumento para '%d' debe ser de tipo `int`; si no lo es, `printf` no funcionará correctamente). Por lo tanto, la salida es una línea que se ve así:

```
El elemento 20 de la serie de Fibonacci es 6765
```

Este programa no contiene una definición para `printf` porque está definido por la biblioteca C, que lo hace disponible en todos los programas C. Sin embargo, cada programa necesita declarar `printf` para que se lo llame correctamente. La línea `#include` se encarga de eso; incluye un archivo de encabezado llamado `stdio.h` en el código del programa. Ese archivo lo proporciona el sistema operativo y contiene declaraciones para las muchas funciones de entrada/salida estándar en la biblioteca C, una de las cuales es `printf`.

No se preocupe por los archivos de encabezado por ahora; los explicaremos más adelante en [Archivos

de encabezado].

El primer argumento de `printf` no tiene que ser una constante de cadena; puede ser cualquier cadena (vea [Cadenas]). Sin embargo, el uso de una constante es el caso más común.

2.3 Programa completo, línea por línea

A continuación se muestra el mismo ejemplo, explicado línea por línea. **Principiantes, ¿les resulta útil o no? ¿Preferirían un diseño diferente para el ejemplo? Por favor, dígaselo a rms@gnu.org**.

```
#include <stdio.h>      /* Incluir declaración de funciones de E/S habituales */
                        /* como printf. */
                        /* La mayoría de los programas las necesitan. */

int                     /* Esta función devuelve un valor entero (int). */
fib (int n)             /* Su nombre es fib; */
                        /* su argumento se llama n. */
{                       /* Inicio del cuerpo de la función. */
/* Esto evita que la recursión sea infinita. */
if (n <= 2)             /* Si n es 1 o 2, */
   return 1;            /* hacer que fib devuelva 1. */
else                    /* de lo contrario, sumar los dos números */
                        /* de Fibonacci anteriores. */
   return fib (n - 1) + fib (n - 2);
}

int                     /* Esta función devuelve un valor entero (int). */
main (void)             /* Comenzar aquí; ignorar argumentos. */
{                       /* Imprimir mensaje con números. */
   printf ("El elemento de la serie de Fibonacci %d es %d\n", 20, fib (20));
   return 0;            /* Finalizar programa, informar éxito. */
}
```

2.4 Compilación del programa de ejemplo

Para ejecutar un programa en C es necesario convertir el código fuente en un archivo *ejecutable*. Esto se denomina *compilar* el programa y el comando para hacerlo usando GNU C es `gcc`.

Este programa de ejemplo consta de un solo archivo fuente. Si llamamos a ese archivo `fib1.c`, el comando completo para compilarlo es el siguiente:

```
gcc -g -O -o fib1 fib1.c
```

Aquí, `-g` indica que se debe generar información de depuración / debugging, `-O` indica que se debe optimizar a nivel básico y `-o` fib1 indica que se debe colocar el programa ejecutable en el archivo `fib1`.

Para ejecutar el programa, utilice su nombre de archivo como un comando de shell. Por ejemplo,

```
./fib1
```

Sin embargo, a menos que esté seguro de que el programa es correcto, es de esperar que deba depurarlo. Utilice este comando,

```
gdb fib1
```

que inicia el depurador de GDB (consulte la Sección “Una sesión de GDB de muestra” en *Depuración*

con GDB) para poder ejecutar y depurar el programa ejecutable `fib1`.

El consejo de Richard Stallman, basado en su experiencia personal, es recurrir al depurador tan pronto como pueda reproducir el problema. No intente evitarlo utilizando otros métodos en su lugar; en ocasiones son atajos, pero por lo general desperdician una cantidad ilimitada de tiempo. Con el depurador, seguramente encontrará el error en un tiempo razonable; en general, realizará su trabajo más rápido. Cuanto antes se ponga serio e inicie el depurador, más pronto es probable que encuentre el error.

Consulte [Compilación], para obtener una introducción a la compilación de programas más complejos que constan de más de un archivo fuente.

3 Almacenamiento y datos

El almacenamiento en los programas de C se compone de unidades llamadas `bytes`. Un byte es la unidad de almacenamiento más pequeña que se puede utilizar de manera óptima.

En casi todas las computadoras, un byte consta de 8 bits. Hay algunas computadoras peculiares (en su mayoría "controladores integrados" para sistemas muy pequeños) en las que un byte es más largo que eso, pero este manual no intenta explicar la peculiaridad de esas computadoras; asumimos que un byte tiene 8 bits.

Cada tipo de datos de C está compuesto por una cierta cantidad de bytes; esa cantidad es el *tamaño* del tipo de datos. Vea [Tamaño de tipo], para obtener más detalles. Los tipos `signed char` y `unsigned char` tienen una longitud de un byte; use esos tipos para operar con los datos byte por byte. Vea [Tipos con y sin signo]. Puede referirse a una serie de bytes consecutivos como un arreglo de elementos `char`; así es como se ve una cadena de caracteres en la memoria. Vea [Constantes de cadena].

4 Más allá de los números enteros

Hasta ahora hemos presentado programas que operan con números enteros. En este capítulo presentaremos ejemplos de manejo de números no enteros y arreglos de números.

4.1 Un ejemplo con números no enteros

A continuación se muestra una función que opera con *números de punto flotante* que no tienen que ser números enteros y los devuelve. El *punto flotante* representa un número como una fracción junto con una potencia de 2. (Para obtener más detalles, consulte [Tipos de datos de punto flotante]). Este ejemplo calcula el promedio de tres números de punto flotante que se le pasan como argumentos:

```
double
prom_de_tres (double a, double b, double c)
{
    return (a + b + c) / 3;
}
```

Los valores de los parámetros `a`, `b` y `c` no tienen por qué ser números enteros, e incluso cuando lo sean, lo más probable es que su promedio no sea un número entero.

`double` es el tipo de datos habitual en C para los cálculos con números de punto flotante.

Para imprimir un `double` con `printf`, debemos utilizar '%f' en lugar de '%d':

```
printf ("El promedio es %f\n", prom_de_tres (1.1, 9.8, 3.62));
```

El código que llama a `printf` debe pasar un `double` para imprimir con '%f' y un `int` para imprimir con '%d'. Si el argumento tiene el tipo incorrecto, `printf` producirá una salida sin sentido.

A continuación se muestra un programa completo que calcula el promedio de tres números específicos e imprime el resultado:

```
double
prom_de_tres (double a, double b, double c)
{
   return (a + b + c) / 3;
}

int
main (void)
{
   printf ("El promedio es %f\n", prom_de_tres (1.1, 9.8, 3.62));
   return 0;
}
```

A partir de ahora no presentaremos ejemplos de llamadas a `main`. En su lugar, le recomendamos que los escriba usted mismo cuando desee probar la ejecución de algún código.

4.2 Un ejemplo con arreglos

Una función para tomar el promedio de tres números es muy específica y limitada. Una función más general tomaría el promedio de cualquier número de números. Eso requiere pasar los números en un arreglo. Un arreglo es un objeto en memoria que contiene una serie de valores del mismo tipo de datos. Este capítulo presenta los conceptos básicos y el uso de arreglos a través de un ejemplo; para obtener una explicación completa, consulte [Arreglos].

A continuación se muestra una definición de función para tomar el promedio de varios números de punto flotante, pasados como tipo `double`. El primer parámetro, `largo`, especifica cuántos números se pasan. El segundo parámetro, `datos_entrada`, es un arreglo que contiene esos números.

```
double
prom_de_doble (int largo, double datos_entrada[])
{
   double suma = 0;
   int i;

   for (i = 0; i < largo; i++)
      suma = suma + datos_entrada[i];

   return suma / largo;
}
```

Esto introduce la expresión para hacer referencia a un elemento de un arreglo: `datos_entrada[i]` significa el elemento en el índice `i` en `datos_entrada`. El índice del elemento puede ser cualquier expresión con un valor entero; en este caso, la expresión es `i`. Consulte la [Acceso a elementos de matriz].

El índice válido más bajo en un arreglo es 0, *no* 1, y el índice válido más alto es uno menos que el número de elementos. (Esto se conoce como *indexación de origen cero* / *zero-origin indexing*).

Este ejemplo también introduce la forma de declarar que un parámetro de la función es un arreglo. Dichas declaraciones se modelan a partir de la sintaxis de un elemento del arreglo. Así como `double foo` declara que `foo` es de tipo `double`, `double datos_entrada[]` declara que cada elemento de `datos_entrada` es de tipo `double`. Por lo tanto, `datos_entrada` en sí tiene el tipo "arreglo de `double`".

Al declarar un parámetro de arreglo, no es necesario decir qué tan largo es arreglo. En este caso, el parámetro `datos_entrada` no tiene información de longitud. Por eso la función necesita otro parámetro, `largo`, para que el invocador proporcione esa información a la función `prom_de_doble`.

4.3 Ejemplo de llamada de arreglo

Para llamar a la función `prom_de_doble` es necesario crear un arreglo y luego pasarlo como argumento. A continuación, se incluye un ejemplo.

```
{
/* arreglo de valores a promediar. */
double numeros_a_prom[5];
/* El promedio, una vez que lo calculamos. */
double promedio;

/* Completar elementos de numeros_a_promediar. */

numeros_a_prom[0] = 58.7;
numeros_a_prom[1] = 5.1;
numeros_a_prom[2] = 7.7;
numeros_a_prom[3] = 105.2;
numeros_a_prom[4] = -3.14159;

promedio = prom_de_doble (5, numeros_a_prom);

/* . . . Ahora haz uso del promedio. . . */
}
```

Esto muestra nuevamente una expresión de subíndice de matriz, esta vez en el lado izquierdo de una asignación, que almacena un valor en un elemento de un arreglo.

Se muestra también cómo declarar una variable local como un arreglo: `double numeros_a_prom[5];`. Dado que esta declaración asigna el espacio para el arreglo, *necesita saber la longitud del arreglo.* Puede especificar la longitud con cualquier expresión cuyo valor sea un entero, pero en esta declaración la longitud es una constante, el entero 5.

El *nombre del arreglo*, cuando se usa solo como una expresión, representa la dirección de los datos del arreglo, y eso es lo que se pasa a la función `prom_de_doble` en `prom_de_doble (5, numeros_a_prom)`.

Podemos hacer que el código sea más fácil de mantener al evitar la necesidad de escribir 5, la longitud del arreglo, al llamar a `prom_de_doble`. De esa manera, si cambiamos el arreglo para incluir más elementos, no tendremos que cambiar esa llamada. Una forma de hacer esto es con el operador `sizeof`

```
promedio = prom_de_doble ((sizeof (numeros_a_prom)
                           / sizeof (numeros_a_prom[0])),
                           números_a_prom);
```

Esto calcula la cantidad de elementos en `numeros_a_prom` dividiendo su tamaño total por el tamaño de un elemento. Consulte [Tamaño de tipo], para obtener más detalles sobre el uso de `sizeof`.

No mostramos en este ejemplo lo que sucede después de almacenar el resultado de `prom_de_doble` en la variable `promedio`. Es de suponer que seguiría más código que utilizaría ese resultado de alguna manera.

(¿Por qué calcular el promedio y no usarlo?) Pero eso no es parte de este tema.

4.4 Variaciones para el ejemplo de arreglo

El código para llamar a `prom_de_doble` tiene dos declaraciones que comienzan con el mismo tipo de datos:

```
/* Arreglo de valores a promediar. */
double numeros_a_prom[5];

/* El promedio, una vez que lo calculamos. */
double promedio;
```

En C, puedes combinar los dos, de esta manera:

```
double numeros_a_prom[5], promedio;
```

Esto declara `numeros_a_prom` de modo que cada uno de sus elementos sea un `double`, y `promedio` de modo que simplemente sea un `double`.

Sin embargo, si bien *puedes* combinarlos, eso no significa que *debas* hacerlo. Si es útil escribir comentarios sobre las variables, y generalmente lo es, entonces es más claro mantener las declaraciones separadas para que puedas poner un comentario en cada una. Eso también ayuda con el uso de herramientas textuales para encontrar ocurrencias de una variable en archivos fuente.

Establecemos todos los elementos del arreglo `numeros_a_prom` con asignaciones, pero es más conveniente usar un inicializador en la declaración:

```
{
   /* El arreglo de valores a promediar. */
   double numeros_a_prom[]
      = { 58.7, 5.1, 7.7, 105.2, -3.14159 };

   /* El promedio, una vez que lo calculamos. */
   promedio = prom_de_doble ((sizeof (numeros_a_prom)
                              / sizeof (numeros_a_prom[0])),
                             numeros_a_prom);

   /* . . . Ahora haz uso del promedio . . . */
}
```

El inicializador del arrego es una lista de valores separados por comas, delimitada por llaves. Consulte [Inicializadores].

Tenga en cuenta que la declaración no especifica un tamaño para `numeros_a_prom`, por lo que el tamaño se determina a partir del inicializador. Hay cinco valores en el inicializador, por lo que el arreglo `numeros_a_prom` obtiene una longitud de 5. Si agregamos otro elemento al inicializador, `numeros_a_prom` tendrá seis elementos.

Debido a que el código calcula la cantidad de elementos a partir del tamaño de la matriz, utilizando `sizeof`, el programa operará sobre todos los elementos en el inicializador, independientemente de cuántos sean.