CAPITULO 2: Tipos, operadores y expresiones

Las variables y las constantes son los objetos de datos básicos que se manipulan en un programa. Las declaraciones muestran las variables que se van a utilizar y establecen el tipo que tienen y algunas veces cuáles son sus valores iniciales. Los operadores especifican lo que se hará con las variables. Las expresiones combinan variables y constantes para producir nuevos valores. El tipo de un objeto determina el conjunto de valores que puede tener y qué operaciones se pueden realizar sobre él. Estos son los temas de este capítulo.

El estándar ANSI ha hecho muchos pequeños cambios y agregados a los tipos básicos y a las expresiones. Ahora hay formas **signed** y **unsigned** de todos los tipos enteros, y notaciones para constantes sin signo y constantes de carácter hexadecimales. Las operaciones de punto flotante pueden hacerse en precisión sencilla; también hay un tipo **long double** para precisión extendida. Las constantes de cadena pueden concatenarse al tiempo de compilación. Las enumeraciones son ya parte del lenguaje, formalizando una característica pendiente por mucho tiempo. Los objetos pueden ser declarados **const**, lo que impide que cambien. Las reglas para conversión automática entre tipos aritméticos se aumentaron para manejar el ahora más rico conjunto de tipos.

2.1 Nombres de variables

Aunque no lo mencionamos en el capítulo 1, existen algunas restricciones en los nombre de las variables y de las constantes simbólicas. Los nombres se componen de letras y dígitos; el primer carácter debe ser una letra. El carácter de subrayado "_" cuenta como una letra; algunas veces es útil para mejorar la legibilidad de nombres largos de variables. Sin embargo, no se debe comenzar los nombres de variables con este carácter, puesto que las rutinas de biblioteca con frecuencia usan tales nombres. Las letras mayúsculas y minúsculas son distintas, de tal manera que **x** y **X** son dos nombres diferentes. La práctica tradicional de C es usar letras minúsculas para nombres de variables, y todo en mayúsculas para constantes simbólicas.

Al menos los primeros 31 caracteres de un nombre interno son significativos. Para nombres de funciones y variables externas el número puede ser menor que 31, puesto que los nombres externos los pueden usar los ensambladores y los cargadores, sobre los que el lenguaje no tiene control. Para nombres externos, el estándar garantiza distinguir sólo para 6 caracteres y sin diferenciar mayúsculas de minúsculas. Las palabras clave como **if, else, int, float**, etc., están reservadas: no se pueden utilizar como nombres de variables. Todas ellas deben escribirse con minúsculas.

Es conveniente elegir nombres que estén relacionados con el propósito de la variable, que no sea probable confundirlos tipográficamente. Nosotros tendemos a utilizar nombres cortos para variables locales, especialmente índices de iteraciones, y nombres más largos para variables externas.

2.2 Tipos y tamaños de datos

Hay unos cuantos tipos de datos básicos en C:

char	un solo byte, capaz de contener un carácter del conjunto de caracteres local.
int	un entero, normalmente del tamaño natural de los enteros en la máquina en la que se ejecuta.
float	punto flotante de precisión normal.
double	punto flotante de doble precisión.

Además, existen algunos calificadores que se aplican a estos tipos básicos. **short** y **long** se aplican a enteros:

```
short int sh;
long int counter;
```

La palabra **int** puede omitirse de tales declaraciones, lo que típicamente se hace.

La intención es que **short** y **long** puedan proporcionar diferentes longitudes de enteros donde sea práctico; **int** será normalmente el tamaño natural para una máquina en particular. A menudo **short** es de 16 bits y **long** de 32; **int** es de 16 o de 32 bits. Cada compilador puede seleccionar libremente los tamaños apropiados para su propio hardware, sujeto sólo a la restricción de que los **short**s e **int**s son, por lo menos de 16 bits, los **long**s son por lo menos de 32 bits y el **short** no es mayor que **int**, el cual a su vez no es mayor que **long**.

El calificador **signed** o **unsigned** puede aplicarse a **char** o a cualquier entero. Los números **unsigned** son siempre positivos o cero y obedecen las leyes de la aritmética módulo 2^n, donde n es el número de bits en el tipo. Así, por ejemplo, si los **char** son de 8 bits, las variables **unsigned char** tienen valores entre 0 y 255, en tanto que las variables **signed char** tienen valores entre −128 y 127 (en una máquina de complemento a dos). El hecho de que los **char**s ordinarios sean con

signo o sin él depende de la máquina, pero los caracteres que se pueden imprimir son siempre positivos.

El tipo **long double** especifica punto flotante de precisión extendida. Igual que con los enteros, los tamaños de objetos de punto flotante se definen en la implantación; **float, double** y **long double** pueden representar uno, dos o tres tamaños distintos.

Los archivos de encabezado *headers* estándar **<limits.h>** y **<float.h>** contienen constantes simbólicas para todos esos tamaños, junto con otras propiedades de la máquina y del compilador, los cuales se discuten en el apéndice B.

Ejercicio 2-1. Escriba un programa para determinar los rangos de variables **char**, **short**, **int** y **long**, tanto **signed** como **unsigned**, imprimiendo los valores apropiados de los *headers* estándar y por cálculo directo. Es más difícil si los calcula: determine los rangos de los varios tipos de punto flotante. □

2.3 Constantes

Una constante entera como 1234 es un **int**. Una constante **long** se escribe con una l (ele) o L terminal, como en 123456789L; un entero demasiado grande para caber dentro de un **int** también será tomado como **long**. Las constantes sin signo se escriben con una u o U, terminal y el sufijo ul o UL indica **unsigned long**.

Las constantes de punto flotante contienen un punto decimal (123.4) o un exponente (1e−2) o ambos; su tipo es **double**, a menos que tengan sufijo. Los sufijos f o F indican una constante **float**; l o L indican un **long double**.

El valor de un entero puede especificarse en forma octal o hexadecimal en lugar de decimal. Un 0 (cero) al principio en una constante entera significa octal; 0x ó 0X al principio significa hexadecimal. Por ejemplo, el decimal 31 puede escribirse como 037 en octal y 0x1f ó OX1F en hexadecimal. Las constantes octales y hexadecimales también pueden ser seguidas por L para convertirlas en **long** y U para hacerlas **unsigned**: OXFUL es una constante **unsigned long** con valor de 15 en decimal.

Una *constante de carácter* es un entero, escrito como un carácter dentro de apóstrofos, tal como 'x'. El valor de una constante de carácter es el valor numérico del carácter en el conjunto de caracteres de la máquina. Por ejemplo, en el conjunto de caracteres ASCII el carácter constante '0' tiene el valor de 48, el cual no está relacionado con el valor numérico 0. Si escribimos '0' en vez 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 carácter participan en operaciones numéricas tal como cualesquier otros enteros, aunque se utilizan más comúnmente en comparaciones con otros caracteres.

Ciertos caracteres pueden ser representados en constante de carácter y de cadena, por medio de secuencias de escape como \n (nueva línea); esas secuencias se ven

como dos caracteres, pero representan sólo uno. Además, un patrón de bits arbitrario de tamaño de un byte puede ser especificado por

'*ooo*'

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

'\x*hh*'

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

```
#define VTAB '\013'     /* tab vertical ASCII */
#define BELL '\007'     /* carácter campana ASCII */
```

o, en hexadecimal,

```
#define VTAB '\xb'      /* tab vertical ASCII */
#define BELL '\x7'      /* carácter campana ASCII */
```

El conjunto completo de secuencias de escape es

\a	carácter de alarma (campana)	\\	diagonal invertida
\b	retroceso	\?	interrogación
\f	avance de hoja	\'	apóstrofo
\n	nueva línea	\"	comillas
\r	regreso de carro	*ooo*	número octal
\t	tabulador horizontal	\x*hh*	número hexadecimal
\v	tabulador vertical		

La constante de carácter '\0' representa el carácter con valor cero, el carácter nulo. '\0' a menudo se escribe en vez de 0 para enfatizar la naturaleza de carácter de algunas expresiones, pero el valor numérico es precisamente 0.

Una *expresión constante* es una expresión que sólo inmiscuye constantes. Tales expresiones pueden ser evaluadas durante la compilación en vez de que se haga en tiempo de ejecución, y por tanto pueden ser utilizadas en cualquier lugar en que pueda encontrarse una constante, como en

```
#define MAXLINE 1000
char line [MAXLINE + 1];
```

o

```
#define LEAP 1    /* en años bisiestos*/
int days [31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

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

```
"Soy una cadena"
```

o

```
""/* la cadena vacía */
```


Las comillas no son parte de la cadena, sólo sirven para delimitarla. Las mismas secuencias de escape utilizadas en constantes de carácter se aplican en cadenas; \" representa el carácter comillas. Las constantes de cadena pueden ser concatenadas en tiempo de compilación:

```
"hola," "mundo"
```

es equivalente a

```
"hola, mundo"
```

Esto es útil para separar cadenas largas entre varias líneas fuente.

Técnicamente, una constante de cadena es un arreglo de caracteres. La representación interna de una cadena tiene un carácter nulo '\0' al final, de modo que el almacenamiento físico requerido es uno más del número de caracteres escritos entre las comillas. Esta representación significa que no hay límite en cuanto a qué tan larga puede ser una cadena, pero los programas deben leer completamente una cadena para determinar su longitud. La función **strlen (s)** de la biblioteca estándar regresa la longitud de su argumento s de tipo cadena de caracteres, excluyendo el '\0' terminal. Aquí está nuestra versión:

```
/* strlen: regresa la longitud de s */
int strlen(char s[ ])
{
    int i;

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

strlen y otras funciones para cadenas están declaradas en el *header* estándar <string.h>.

Se debe ser cuidadoso al distinguir entre una constante de carácter y una cadena que contiene un sólo carácter: 'x' no es lo mismo que "x". El primero es un entero, utilizado para producir el valor numérico de la letra *x* en el conjunto de caracteres de la máquina. El último es un arreglo de caracteres que contiene un carácter (la letra *x*) y un '\0'.

Existe otra clase de constante, la *constante de enumeración*. Una enumeración es una lista de valores enteros constantes, como en

```
enum boolean {NO, YES};
```

El primer nombre en un enum tiene valor 0, el siguiente 1, y así sucesivamente, a menos que sean especificados valores explícitos. Si no todos los valores son es-

pecificados, los valores no especificados continúan la progresión a partir del último valor que sí lo fue, como en el segundo de esos ejemplos:

```
enum escapes { BELL = '\a', RETROCESO = '\b', TAB = '\t',
               NVALIN = '\n', VTAB = '\v', RETURN = '\r'};

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

Los nombres que están en enumeraciones diferentes deben ser distintos. Los valores no necesitan ser distintos dentro de la misma enumeración.

Las enumeraciones proporcionan una manera conveniente de asociar valores constantes con nombres, una alternativa a #define con la ventaja de que los valores pueden ser generados para uno. Aunque las variables de tipos enum pueden ser declaradas, los compiladores no necesitan revisar que lo que se va a almacenar en tal variable es un valor válido para la enumeración. No obstante, las variables de enumeración ofrecen la oportunidad de revisarlas y tal cosa es a menudo mejor que #define. Además, un depurador puede ser capaz de imprimir los valores de variables de enumeración en su forma simbólica.

2.4 Declaraciones

Todas las variables deben ser declaradas antes de su uso, aunque ciertas declaraciones pueden ser hechas en forma implícita por el contexto. Una declaración especifica un tipo, y contiene una lista de una o más variables de ese tipo, como en

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

Las variables pueden ser distribuidas entre las declaraciones en cualquier forma; la lista de arriba podría igualmente ser escrita como

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

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

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


```
char esc = '\\';
int   i = 0;
int   limit = MAXLINE+1;
float eps = 1.0e-5;
```

Si la variable en cuestión no es automática, la inicialización es efectuada sólo una vez, conceptualmente antes de que el programa inicie su ejecución, y el inicializador debe ser una expresión constante. Una variable automática explícitamente inicializada es inicializada cada vez que se entra a la función o bloque en que se encuentra; el inicializador puede ser cualquier expresión. Las variables estáticas y externas son inicializadas en cero por omisión. Las variables automáticas para las que no hay un inicializador explícito tienen valores indefinidos (esto es, basura).

El calificador const puede aplicarse a la declaración de cualquier variable para especificar que su valor no será cambiado. Para un arreglo, el calificador const indica que los elementos no serán alterados.

```
const double e = 2.71828182845905;
const char msg[] = "precaución: ";
```

La declaración const también se puede utilizar con argumentos de tipo arreglo, para indicar que la función no cambia ese arreglo:

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

Si se efectúa un intento de cambiar un const, el resultado está definido por la implantación.

2.5 Operadores aritméticos

Los operadores aritméticos binarios son +, −, *, /, y el operador módulo %. La división entera trunca cualquier parte fraccionaria. La expresión

```
x % y
```

produce el residuo cuando x es dividido entre y, por lo que es cero cuando y divide a x exactamente. Por ejemplo, un año es bisiesto si es divisible entre 4 pero no entre 100, excepto aquellos años que son divisibles entre 400, que *sí son* bisiestos. Por lo tanto

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

El operador % no puede aplicarse a operandos float o double. La dirección de truncamiento para / y el signo del resultado de % son dependientes de la máquina para operandos negativos, así como la acción que se toma en caso de sobreflujo o subflujo.

Los operadores binarios + y − tienen la misma precedencia, la cual es menor que la precedencia de *, /, y %, que a su vez es menor que + y − unarios. Los operadores aritméticos se asocian de izquierda a derecha.

La tabla 2-1 que se encuentra al final de este capítulo, resume la precedencia y asociatividad para todos los operadores.

2.6 Operadores de relación y lógicos

Los operadores de relación son

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

Todos ellos tienen la misma precedencia. Precisamente bajo ellos en precedencia están los operadores de igualdad:

```
==   !=
```

Los operadores de relación tienen precedencia inferior que los operadores aritméticos, así que una expresión como `i < lim-1` se toma como `i < (lim-1)`, como se esperaría.

Más interesantes son los operadores lógicos && y ||. Las expresiones conectadas por && o || son evaluadas de izquierda a derecha, y la evaluación se detiene tan pronto como se conoce el resultado verdadero o falso. La mayoría de los programas en C descansan sobre esas propiedades. Por ejemplo, aquí está un ciclo de la función de entrada getline que escribimos en el capítulo 1:

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

Antes de leer un nuevo carácter es necesario verificar que hay espacio para almacenarlo en el arreglo s, así que la prueba `i < lim-1` *debe* hacerse primero. Además, si esta prueba falla, no debemos seguir y leer otro carácter.

De manera semejante, sería desafortunado si c fuese probada contra EOF antes de que se llame a getchar; por lo tanto, la llamada y la asignación deben ocurrir antes de que se pruebe el carácter c.

La precedencia de && es más alta que la de ||, y ambas son menores que los operadores de relación y de asignación, así que expresiones como

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

no requieren de paréntesis adicionales. Pero puesto que la precedencia de != es superior que la asignación, los paréntesis se necesitan en

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

para obtener el resultado deseado de asignación a c y después comparación con '\n'.

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

El operador unario de negación ! convierte a un operando que no es cero en 0, y a un operando cero en 1. Un uso común de ! es en construcciones como

```
if (!válido)
```

en lugar de

```
if (válido == 0)
```

Es difícil generalizar acerca de cuál es la mejor. Construcciones como !válido se leen en forma agradable ("si no es válido"), pero otras más complicadas pueden ser difíciles de entender.

Ejercicio 2-2. Escriba un ciclo equivalente a la iteración for anterior sin usar && o ||. □

2.7 Conversiones de tipo

Cuando un operador tiene operandos de tipos diferentes, éstos se convierten a un tipo común de acuerdo con un reducido número de reglas. En general, las únicas conversiones automáticas son aquellas que convierten un operando "angosto" en uno "amplio" sin pérdida de información, tal como convertir un entero a punto flotante en una expresión como f + i. Las expresiones que no tienen sentido, como utilizar un float como subíndice, no son permitidas. Las expresiones que podrían perder información, como asignar un tipo mayor a uno más corto, o un tipo de punto flotante a un entero, pueden producir una advertencia, pero no son ilegales.

Un char sólo es un entero pequeño, por lo que los char se pueden utilizar libremente en expresiones aritméticas. Esto permite una flexibilidad considerable en ciertas clases de transformación de caracteres. Una es ejemplificada con esta ingenua implantación de la función atoi, que convierte una cadena de dígitos en su equivalente numérico.

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

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

Tal como se discutió en el capítulo 1, la expresión

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

da el valor numérico del carácter almacenado en s[i], debido a que los valores de '0', '1', etc., forman una secuencia ascendente contigua.

Otro ejemplo de conversión de **char** a **int** es la función lower, que convierte un carácter sencillo a minúscula *para el conjunto de caracteres ASCII*. Si el carácter no es una letra mayúscula, lower lo regresa sin cambio.

```
/* lower:  convierte c a minúscula; solamente ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Esto funciona para ASCII debido a que las correspondientes letras mayúsculas y minúsculas están a una distancia fija como valores numéricos y cada alfabeto es contiguo —no hay sino letras entre A y Z. Sin embargo, esta última observación no es cierta para el conjunto de caracteres EBCDIC, así que este código podría convertir algo más que sólo letras en EBCDIC.

El *header* estándar <ctype.h>, que se describe en el apéndice B, define una familia de funciones que proporcionan pruebas y conversiones independientes de los juegos de caracteres. Por ejemplo, la función **tolower(c)** regresa el valor de la letra minúscula de c si c es una mayúscula, de modo que **tolower** es un reemplazo transportable para la función lower mostrada antes. De modo semejante, la prueba.

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

puede reemplazarse por

```
isdigit(c)
```

Nosotros utilizaremos las funciones de <ctype.h> en adelante.

Existe un sutil punto acerca de la conversión de caracteres a enteros. El lenguaje no especifica si las variables de tipo **char** son valores con o sin signo. Cuando un **char** se convierte a **int**, ¿puede producir alguna vez un entero negativo? La respuesta varía de una máquina a otra, reflejando diferencias en la arquitectura. En algunas máquinas un **char** cuyo bit más a la izquierda es 1 se convertirá a un entero negativo ("extensión de signo"). En otras, un **char** es promovido a un **int** agregando ceros del lado izquierdo, así que siempre es positivo.

La definición de C garantiza que ningún carácter que esté en el conjunto estándar de caracteres de impresión de la máquina será negativo, de modo que esos caracteres siempre serán cantidades positivas en las expresiones. Pero hay patrones arbitrarios de bits almacenados en variables de tipo carácter que pueden

aparecer como negativos en algunas máquinas, aunque sean positivos en otras. Por transportabilidad, se debe especificar **signed o unsigned** si se van a almacenar datos que no son caracteres en variables tipo **char**.

Las expresiones de relación como i > j y las expresiones lógicas conectadas por && y || están definidas para tener un valor de 1 siendo verdaderas, y 0 al ser falsas. De este modo, la asignación

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

hace 1 a d si c es un dígito, y 0 si no lo es. Sin embargo, las funciones como isdigit pueden regresar cualquier valor diferente de cero como verdadero. En la parte de validación de **if, while, for**, etc., "verdadero" es sólo "diferente de cero", por lo que esto no hace diferencia.

Las conversiones aritméticas implícitas trabajan como se espera. En general, si un operador como + o * que toma dos operandos (operador binario) tiene operandos de diferentes tipos, el tipo "menor" es *promovido* al tipo "superior" antes de que la operación proceda. El resultado es el del tipo mayor. La sección 6 del apéndice A establece las reglas de conversión en forma precisa. Si no hay operandos **unsigned**, sin embargo, el siguiente conjunto informal de reglas bastará:

Si cualquier operando es **long double**, conviértase el otro a **long double**.

De otra manera, si cualquier operando es **double**, conviértase el otro a **double**.

De otra manera, si cualquier operando es **float**, conviértase el otro a **float**.

De otra manera, conviértase **char** y **short** a **int**.

Después, si cualquier operando es **long**, conviértase el otro a **long**.

Nótese que los **float** que están en una expresión no se convierten automáticamente a **double**; esto es un cambio de la definición original. En general, las funciones matemáticas como las de <math.h> utilizarán doble precisión. La razón principal para usar **float** es ahorrar espacio de almacenamiento en arreglos grandes o, con menor frecuencia, ahorrar tiempo en máquinas en donde la aritmética de doble precisión es particularmente costosa.

Las reglas de conversión son más complicadas cuando hay operandos **unsigned**. El problema es que las comparaciones de valores con signo y sin signo son dependientes de la máquina, debido a que dependen de los tamaños de los varios tipos de enteros. Por ejemplo, supóngase que **int** es de 16 bits y **long** de 32. Entonces -1L < 1U, debido a que 1U, que es un **int**, es promovido a **signed long**. Pero -1L > 1UL, debido a que -1L es promovido a **unsigned long** y así parece ser un gran número positivo.

Las conversiones también tienen lugar en las asignaciones; el valor del lado derecho es convertido al tipo de la izquierda, el cual es el tipo del resultado.

Un carácter es convertido a un entero, tenga o no extensión de signo, como se describió anteriormente.

Los enteros más largos son convertidos a cortos o a char desechando el exceso de bits de más alto orden. Así en

```
int  i;
char c;

i = c;
c = i;
```

el valor de c no cambia. Esto es verdadero ya sea que se inmiscuya o no la extensión de signo. Sin embargo, el invertir el orden de las asignaciones podría producir pérdida de información.

Si x es float e i es int, entonces x = i e i = x producirán conversiones; de float a int provoca el truncamiento de cualquier parte fraccionaria. Cuando double se convierte a float, el que se redondee o trunque el valor es dependiente de la implantación.

Puesto que un argumento de la llamada a una función es una expresión, también suceden conversiones de tipo cuando se pasan argumentos a funciones. En ausencia del prototipo de una función, char y short pasan a ser int, y float se hace doble. Esta es la razón por la que hemos declarado los argumentos a funciones como int y double, aun cuando la función se llama con char y float.

Finalmente, la conversión explícita de tipo puede ser forzada ("coaccionada") en cualquier expresión, con un operador unario llamado *cast*. En la construcción

(*nombre-de-tipo*) *expresión*

la *expresión* es convertida al tipo nombrado, por las reglas de conversión anteriores. El significado preciso de un *cast* es como si la *expresión* fuera asignada a una variable del tipo especificado, que se utiliza entonces en lugar de la construcción completa. Por ejemplo, la rutina de biblioteca sqrt espera un argumento de doble precisión y producirá resultados sin sentido si maneja inadvertidamente algo diferente. (sqrt está declarado en <math.h>.) Así, si n es un entero, podemos usar

```
sqrt((double) n)
```

para convertir el valor de n a doble antes de pasarlo a sqrt. Nótese que la conversión forzosa produce el *valor* de n en el tipo apropiado; n en sí no se altera. El operador *cast* tiene la misma alta precedencia que otros operadores unarios, como se resume en la tabla del final de este capítulo.

Si un prototipo de función declara argumentos, como debe ser normalmente, la declaración produce conversión forzada automática de los argumentos cuando la función es llamada. Así, dado el prototipo de la función sqrt:

```
double sqrt(double);
```

la llamada

```
raíz2 = sqrt(2);
```

obliga al entero 2 a ser el valor double 2.0 sin necesidad de ningún *cast*.

La biblioteca estándar incluye una implantación transportable de un generador de números pseudoaleatorios, y una función para inicializar la semilla; lo primero ilustra un *cast*:

```
unsigned long int next = 1;

/* rand:  regresa un entero pseudoaleatorio en 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand:  fija la semilla para rand( ) */
void srand(unsigned int seed)
{
    next = seed;
}
```

Ejercicio 2-3. Escriba la función htoi(s), que convierte una cadena de dígitos hexadecimales (incluyendo 0x ó 0X en forma optativa) 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

El lenguaje C proporciona dos operadores poco comunes para incrementar y decrementar variables. El operador de aumento ++ agrega 1 a su operando, en tanto que el operador de disminución -- le resta 1. Hemos usado frecuentemente ++ para incrementar variables, como en

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

El aspecto poco común es que ++ y -- pueden ser utilizado como prefijos (antes de la variable, como en ++n), o como postfijos (después de la variable: n++). En ambos casos, el efecto es incrementar n. Pero la expresión ++n incrementa a n *antes* de que su valor se utilice, en tanto que n++ incrementa a

n *después* de que su valor se ha empleado. Esto significa que en un contexto donde el valor está siendo utilizado, y no sólo el efecto, ++n y n++ son diferentes. Si n es 5, entonces

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

asigna 5 a x, pero

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

hace que x sea 6. En ambos casos, n se hace 6. Los operadores de incremento y decremento sólo pueden aplicarse a variables; una expresión como (i+j)++ es ilegal.

Dentro de un contexto en donde no se desea ningún valor, sino sólo el efecto de incremento, como en

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

prefijos y postfijos son iguales. Pero existen situaciones en donde se requiere específicamente uno u otro. Por ejemplo, considérese la función squeeze(s,c), que elimina todas las ocurrencias del carácter c de una cadena s.

```
/* squeeze:  borra todas las c de s */
void squeeze(char s[], int c)
{
    int i, j;

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

Cada vez que se encuentra un valor diferente de c, éste se copia en la posición actual j, y sólo entonces j es incrementada para prepararla para el siguiente carácter. Esto es exactamente equivalente a

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

Otro ejemplo de construcción semejante viene de la función getline que escribimos en el capítulo 1, en donde podemos reemplazar

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


por algo más compacto como

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

Como un tercer ejemplo, considérese que 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 almacenar la combinación. Como la hemos escrito, strcat no regresa un valor; la versión de la biblioteca estándar regresa un apuntador a la cadena resultante.

```
/* strcat: concatena t al final de s; s debe ser suficientemente grande */
void strcat(char s[], char t[])
{
    int i, j;

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

Como cada carácter se copia de t a s, el ++ postfijo se aplica tanto a i como a j para estar seguros de que ambos están en posición para la siguiente iteración.

Ejercicio 2-4. Escriba una versión alterna de squeeze(s1,s2) que borre cada carácter de s1 que coincida con cualquier carácter de la *cadena* s2. □

Ejercicio 2-5. Escriba la función any(s1,s2), que regresa la primera posición de la cadena s1 en donde se encuentre cualquier carácter de la cadena s2, o −1 si s1 no contiene caracteres de s2. (La función de biblioteca estándar strpbrk hace el mismo trabajo pero regresa un apuntador a la posición encontrada.) □

2.9 Operadores para manejo de bits

El lenguaje C proporciona seis operadores para manejo de bits; sólo pueden ser aplicados a operandos integrales, esto es, char, short, int, y long, con o sin signo.

&	AND de bits
|	OR inclusivo de bits
^	OR exclusivo de bits
<<	corrimiento a la izquierda
>>	corrimiento a la derecha
~	complemento a uno (unario)

El operador AND de bits & a menudo es usado para enmascarar algún conjunto de bits; por ejemplo,

```
n = n & 0177;
```

hace cero todos los bits de n, menos los 7 de menor orden.

El operador OR de bits | es empleado para encender bits:

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

fija en uno a todos los bits de x que son uno en SET_ON.

El operador OR exclusivo ^ pone un uno en cada posición en donde sus operandos tienen bits diferentes, y cero en donde son iguales.

Se deben distinguir los operadores de bits & y | de los operadores lógicos && y ||, que implican evaluación de izquierda a derecha de un valor de verdad. Por ejemplo, si x es 1 y y es 2, entonces x & y es cero en tanto que x && y es uno.

Los operadores de corrimiento << y >> realizan corrimientos a la izquierda y a la derecha de su operando que está a la izquierda, el número de posiciones de bits dado por el operando de la derecha, el cual debe ser positivo. Así x << 2 desplaza el valor de x a la izquierda dos posiciones, llenando los bits vacantes con cero; esto es equivalente a una multiplicación por 4. El correr a la derecha una cantidad *unsigned* siempre llena los bits vacantes con cero. El correr a la derecha una cantidad signada llenará con bits de signo ("corrimiento aritmético") en algunas máquinas y con bits 0 ("corrimiento lógico") en otras.

El operador unario ~ da el complemento a uno de un entero; esto es, convierte cada bit 1 en un bit 0 y viceversa. Por ejemplo,

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

fija los últimos seis bits de x en cero. Nótese que x & ~077 es independiente de la longitud de la palabra, y por lo tanto, es preferible a, por ejemplo, x & 0177700, que supone que x es una cantidad de 16 bits. La forma transportable no involucra un costo extra, puesto que ~077 es una expresión constante que puede ser evaluada en tiempo de compilación.

Como ilustración de algunos de los operadores de bits, considere la función getbits(x,p,n) que regresa el campo de n bits de x (ajustado a la derecha) que principia en la posición p. Se supone que la posición del bit 0 está en el borde derecho y que n y p son valores positivos adecuados. Por ejemplo, getbits(x,4,3) regresa los tres bits que están en la posición 4, 3 y 2, ajustados a la derecha.

```
/* getbits:  obtiene n bits desde la posición p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

La expresión x >> (p+1-n) mueve el campo deseado al borde derecho de la palabra. ~0 es todos los bits en 1; corriendo n bits hacia la izquierda con ~0<<n coloca ceros en los n bits más a la derecha; complementado con ~ hace una máscara de unos en los n bits más a la derecha.

Ejercicio 2-6. Escriba una función setbits(x,p,n,y) que regresa x con los n bits que principian en la posición p iguales a los n bits más a la derecha de y, dejando los otros bits sin cambio. □

Ejercicio 2-7. Escriba una función invert(x,p,n) que regresa x con los n bits que principian en la posición p invertidos (esto es, 1 cambiado a 0 y viceversa), dejando los otros sin cambio. □

Ejercicio 2-8. Escriba una función rightrot(x,n) que regresa el valor del entero x rotado a la derecha n posiciones de bits. □

2.10 Operadores de asignación y expresiones

Las expresiones tales como

```
i = i + 2
```

en las que la variable del lado izquierdo se repite inmediatamente en el derecho, pueden ser escritas en la forma compacta

```
i += 2
```

El operador += se llama *operador de asignación*.

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

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

Si $expr_1$ y $expr_2$ son expresiones, entonces

$expr_1$ *op*= $expr_2$

es equivalente a

$expr_1$ = ($expr_1$) *op* ($expr_2$)

exceptuando que $expr_1$ se calcula sólo una vez. Nótense los paréntesis alrededor de $expr_2$:

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

significa

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

y no

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

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

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

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Declarar al argumento x como unsigned asegura que cuando se corre a la derecha, los bits vacantes se llenarán con ceros, no con bits de signo, sin importar la máquina en la que se ejecute el programa.

Muy aparte de su concisión, los operadores de asignación tienen la ventaja de que corresponden mejor con la forma en que la gente piensa. Decimos "suma 2 a i" o "incrementa i en 2", no "toma i, agrégale 2, después pon el resultado de nuevo en i". Así la expresión i + = 2 es preferible a i = i + 2. Además, para una expresión complicada como

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

el operador de asignación hace al código más fácil de entender, puesto que el lector no tiene que verificar arduamente que dos expresiones muy largas son en realidad iguales, o preguntarse por qué no lo son, y un operador de asignación puede incluso ayudar al compilador a producir código más eficiente.

Ya hemos visto que la proposición de asignación tiene un valor y puede estar dentro de expresiones; el ejemplo más común es

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

Los otros operadores de asignación (+ =, − =, etc.) también pueden estar dentro de expresiones, aunque esto es menos frecuente.

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

Ejercicio 2-9. En un sistema de números de complemento a dos, x &= (x−1) borra el bit 1 de más a la derecha en x. Explique el porqué. Utilice esta observación para escribir una versión más rápida de bitcount. □

2.11 Expresiones condicionales

Las proposiciones

```
if (a > b)
    z = a;
else
    z = b;
```


calculan en z el máximo de a y b. La *expresión condicional*, escrita con el operador ternario "?:", proporciona una forma alternativa para escribir ésta y otras construcciones semejantes. En la expresión

$expr_1$? $expr_2$: $expr_3$

la expresión $expr_1$ es evaluada primero. Si es diferente de cero (verdadero), entonces la expresión $expr_2$ es evaluada, y ése es el valor de la expresión condicional. De otra forma, $expr_3$ se evalúa, y ése es el valor. Sólo uno de entre $expr_2$ y $expr_3$, se evalúa. Así, para hacer z el máximo de a y b,

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

Se debe notar que la expresión condicional es en sí una expresión, y se puede utilizar en cualquier lugar donde otra expresión pueda. Si $expr_2$ y $expr_3$, son de tipos diferentes, el tipo del resultado se determina por las reglas de conversión discutidas anteriormente en este capítulo. Por ejemplo, si f es un float y n es un int, entonces la expresión

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

es de tipo float sea n positivo o no.

Los paréntesis no son necesarios alrededor de la primera expresión de una expresión condicional, puesto que la precedencia de ?: es muy baja, sólo arriba de la asignación. De cualquier modo son recomendables, puesto que hacen más fácil de ver la parte de condición de la expresión.

La expresión condicional frecuentemente lleva a un código conciso. Por ejemplo, este ciclo imprime n elementos de un arreglo, 10 por línea, con cada columna separada por un blanco, y con cada línea (incluida la última) terminada por una nueva línea.

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

Se imprime un carácter nueva línea después de cada diez elementos, y después del n-ésimo. Todos los otros elementos son seguidos por un espacio en blanco. Esto podría verse oscuro, pero es más compacto que el if-else equivalente. Otro buen ejemplo es

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

Ejercicio 2-10. Reescriba la función lower, que convierte letras mayúsculas e minúsculas, con una expresión condicional en vez de un if-else. □

2.12 Precedencia y orden de evaluación

La tabla 2-1 resume las reglas de precedencia y asociatividad de todos los operadores, incluyendo aquellos que aún no se han tratado. Los operadores que están en la misma línea tienen la misma precedencia; los renglones están en orden

de precedencia decreciente, así, por ejemplo, *,/, y % tienen todos la misma precedencia, la cual es más alta que la de + y − binarios. El "operador" ( ) se refiere a la llamada a una función. Los operadores −> y . son utilizados para tener acceso a miembros de estructuras; serán cubiertos en el capítulo 6, junto con sizeof (tamaño de un objeto). En el capítulo 5 se discuten * (indirección a través de un apuntador) y & (dirección de un objeto), y en el capítulo 3 se trata al operador coma.

Nótese que la precedencia de los operadores de bits &, ^, y | está debajo de == y !=. Esto implica que las expresiones de prueba de bits como

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

deben ser completamente colocadas entre paréntesis para dar los resultados apropiados.

TABLA 2-1. PRECEDENCIA Y ASOCIATIVIDAD DE OPERADORES

OPERADORES	ASOCIATIVIDAD
( ) [ ] -> .	izquierda a derecha
! ~ ++ -- + - * & (*tipo*) sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= /= %= &= ^= |= <<= >>=	derecha a izquierda
,	izquierda a derecha

Los +, −, y * unarios, tienen mayor precedencia que las formas binarias.

Como muchos lenguajes, C no especifica el orden en el cual los operandos de un operador serán evaluados. (Las excepciones son &&, ||, ?: y ',') Por ejemplo, en proposiciones como

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

f puede ser evaluada antes de g o viceversa; de este modo si f o g alteran una variable de la que la otra depende, x puede depender del orden de evaluación. Se pueden almacenar resultados intermedios en variables temporales para asegurar una secuencia particular.

De manera semejante, el orden en el que se evalúan los argumentos de una función no está especificado, de modo que la proposición

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

puede producir resultados diferentes con distintos compiladores, dependiendo de si n es incrementada antes de que se llame a power. La solución, por supuesto, es escribir

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

Las llamadas a funciones, proposiciones de asignación anidadas, y los operadores de incremento y decremento provocan "efectos colaterales" —alguna variable es modificada como producto de la evaluación de una expresión. En cualquier expresión que involucra efectos colaterales, pueden existir sutiles dependencias del orden en que las variables involucradas en la expresión se actualizan. La infortunada situación es tipificada por la proposición

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

La pregunta es si el subíndice es el viejo o el nuevo valor de i. Los compiladores pueden interpretar esto en formas diferentes, y generar diferentes respuestas dependiendo de su interpretación. El estándar deja intencionalmente sin especificación la mayoría de tales aspectos. Cuando hay efectos colaterales (asignación a variables) dentro de una expresión, se deja a la prudencia del compilador, puesto que el mejor orden depende grandemente de la arquitectura de la máquina. (El estándar sí especifica que todos los efectos colaterales sobre argumentos sucedan antes de que la función sea llamada, pero eso podría no ayudar en la llamada a printf mostrada anteriormente.)

La moraleja es que escribir un código que dependa del orden de evaluación es una mala práctica de programación en cualquier lenguaje. Naturalmente, es necesario conocer qué cosas evitar, pero si no sabe *cómo* se hacen las cosas en varias máquinas, no debe intentar aprovechar una implantación en particular.