Punteros

No hay duda de que los **punteros** son uno de los temas más desafiantes en la programación en C. Sin embargo, son lo que hace de C uno de los lenguajes más robustos en la industria informática para crear programas poderosos y eficientes sin igual.

Los punteros son fundamentales para comprender el resto de este libro y, de hecho, el resto de lo que C tiene para ofrecer. A pesar de sus desafíos, tenga en cuenta una cosa: todos los programadores principiantes de C (incluido yo mismo) han luchado con los conceptos de punteros. Puede pensar en los punteros como un rito de iniciación para cualquier programador de C.

Fundamentos de los punteros

Los punteros son estructuras muy potentes que los programadores de C pueden utilizar para trabajar con variables, funciones y estructuras de datos a través de sus direcciones de memoria. Los punteros son variables que, en la mayoría de los casos, contienen una dirección de memoria como valor. En otras palabras, una variable puntero contiene una dirección de memoria que apunta a otra variable. ¿Eh? Puede que esto haya sonado raro, así que analicemos un ejemplo: supongamos que tengo una variable entera llamada `iResult` que contiene el valor 75 con una dirección de memoria de `0x948311`. Ahora, supongamos que tengo una variable puntero llamada `myPointer`, que no contiene un valor de datos, sino que contiene una dirección de memoria de `0x948311`, que, por cierto, es la misma dirección de memoria de mi variable entera `iResult`. Esto significa que mi variable puntero llamada `myPointer` apunta indirectamente al valor 75. Este concepto se conoce como **indirección** y es un concepto esencial de los punteros.

Consejo: Lo creas o no, ya has trabajado con punteros en el Capítulo 6. En concreto, el nombre de una matriz no es más que un puntero al inicio de la matriz.

Declaración e inicialización de variables de puntero

Las variables de puntero deben declararse antes de poder usarse, como se muestra en el siguiente código:

```
int x = 0;
int iAge = 30;
int *ptrAge;
```

Simplemente coloque el operador de indirección (*****) delante del nombre de la variable para declarar un puntero. En el ejemplo anterior declaré tres variables, dos variables enteras y una variable puntero. Para facilitar la lectura, utilizo la convención de nombres `ptr` como prefijo. Esto me ayuda a mí y a otros programadores a identificar esta variable como un puntero.

Consejo: No se requieren convenciones de nombres, como `ptr`. Los nombres de las variables y las convenciones de nombres no importan en C. Simplemente ayudan a identificar el tipo de datos de la variable y, si es posible, el propósito de la variable.

Cuando declaré el puntero `ptrAge`, le estaba diciendo a C que quiero que mi variable puntero apunte indirectamente a un tipo de datos entero. Sin embargo, mi variable puntero todavía no apunta a nada. Para hacer referencia indirecta a un valor a través de un puntero, debe asignar una dirección al puntero, como se muestra aquí:

```
ptrAge = &iAge;
```

Se asigna la dirección de memoria de la variable `iAge` a la variable puntero (`ptrAge`). La indirección en este caso se logra colocando el operador unario (&) delante de la variable `iAge`. Esta declaración le dice a C que quiero asignar la dirección de memoria de `iAge` a mi variable puntero `ptrAge`.

El operador unario (&) suele denominarse operador de "dirección de" porque, en este caso, mi puntero `ptrAge` recibe la "dirección de" `iAge`.

Por el contrario, puedo asignar el contenido de aquello a lo que apunta mi variable de puntero (un valor de datos que no es de puntero), como se muestra a continuación.

```
x = *ptrAge;
```

La variable `x` contendrá ahora el valor entero al que apunta `ptrAge`, en este caso el valor entero 30.

Para tener una mejor idea de cómo funcionan los punteros y la indirección, estudie la Figura 7.1.

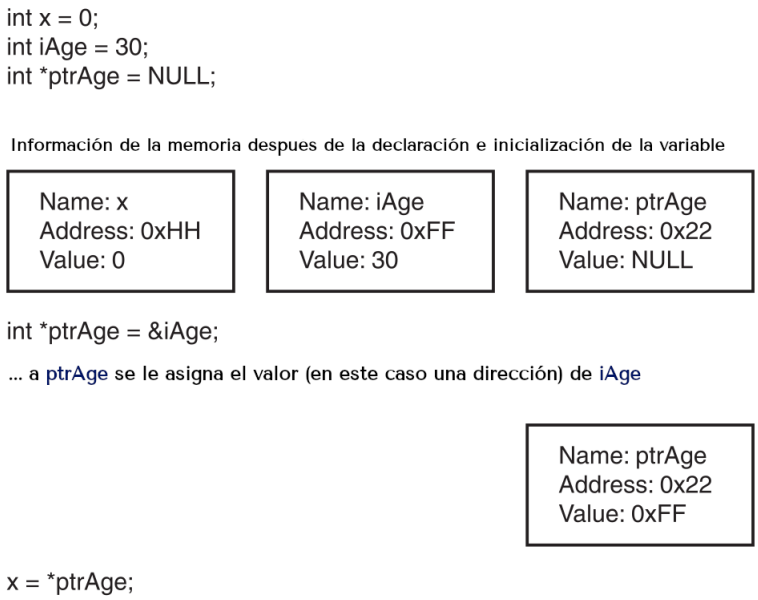

Figura 7.1: Una representación gráfica de la indirección con punteros.

No inicializar las variables de puntero puede generar datos no válidos o expresiones no válidas. Las variables de puntero siempre deben inicializarse con la dirección de memoria de otra variable, con `0` o con la palabra clave `NULL`. El siguiente bloque de código demuestra algunas inicializaciones de puntero válidas.

```
int *ptr1;
int *ptr2;
int *ptr3;

ptr1 = &x;
ptr2 = 0;
ptr3 = NULL;
```

Recordar que a las variables de puntero solo se les pueden asignar direcciones de memoria, `0` o el valor `NULL` es el primer paso para aprender a trabajar con punteros. Considere el siguiente ejemplo, en el que intento asignar un valor que no es una dirección a un puntero.

```
#include <stdio.h>

void main()

{
    int x = 5;
    int *iPtr;

    iPtr = 5; // Esto está mal
    iPtr = x; // Esto también está mal
}
```

Puedes ver que intenté asignar el valor entero 5 a mi puntero. Este tipo de asignación provocará un error en tiempo de compilación, como se muestra en la Figura 7.2.

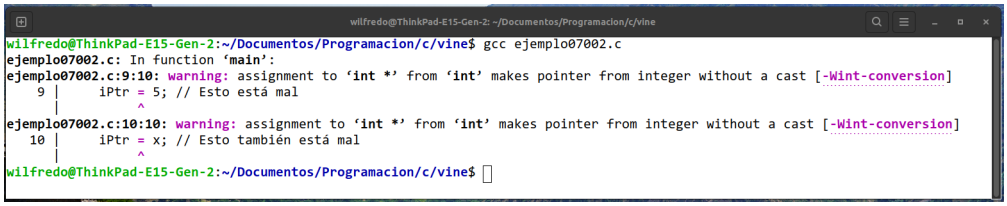

Figura 7.2: Asignación de valores que no son direcciones a punteros.

Precaución: Asignar valores que no sean de dirección, como números o caracteres, a un puntero sin una conversión provocará errores en tiempo de compilación.

Sin embargo, puede asignar valores que no sean de dirección a punteros utilizando un operador de indirección (*), como se muestra a continuación.

```
#include <stdio.h>

void main()
{
    int x = 5;
    int *iPtr;

    iPtr = &x; // A iPtr se le asigna la dirección de x
    *iPtr = 7; // El valor de x se cambia indirectamente a 7
}
```

Este programa asigna la dirección de memoria de la variable `x` a la variable puntero (`iPtr`) y luego asigna indirectamente el valor entero 7 a la variable `x`.

Impresión del contenido de la variable puntero

Para verificar los conceptos de **indirección**, imprima la dirección de memoria de las variables puntero y no puntero utilizando el especificador de conversión %p.

Para demostrar el especificador de conversión %p, estudie el siguiente programa.

```
#include <stdio.h>

void main()
{
    int x = 1;
    int *iPtr;

    iPtr = &x;
    *iPtr = 5;

    printf("\n*iPtr = %p\n&x = %p\n", iPtr, &x);
}
```

Utilizo el especificador de conversión %p para imprimir la dirección de memoria para el puntero y la variable entera. Como se muestra en la Figura 7.3, la variable puntero contiene la misma dirección de memoria (en formato hexadecimal) que la variable entera `x`.

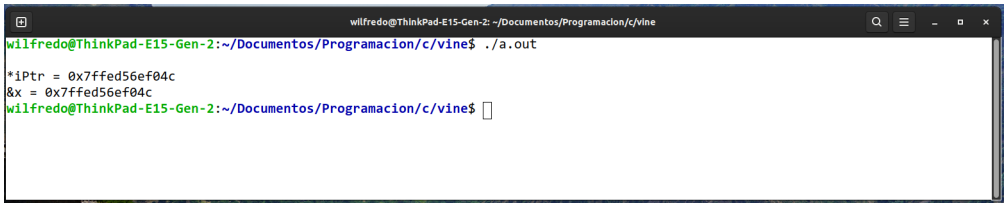

Figura 7.3: Impresión de una dirección de memoria con el especificador de conversión %p..

El siguiente programa (y su salida en la Figura 7.4) continúa demostrando los conceptos de indirección y el especificador de conversión %p.

```c
#include <stdio.h>

void main()
{
    int x = 5;
    int y = 10;
    int *iPtr = NULL;

    printf("\niPtr points to: %p\n", iPtr);

    // Asignar la dirección de memoria de y al puntero
    iPtr = &y;
    printf("\niPtr ahora apunta a: %p\n", iPtr);

    // cambia el valor de x al valor de y
    x = *iPtr;
    printf("\nEl valor de x es ahora: %d\n", x);

    // cambia el valor de y a 15
    *iPtr = 15;
    printf("\nEl valor de y es ahora: %d\n", y);
}
```

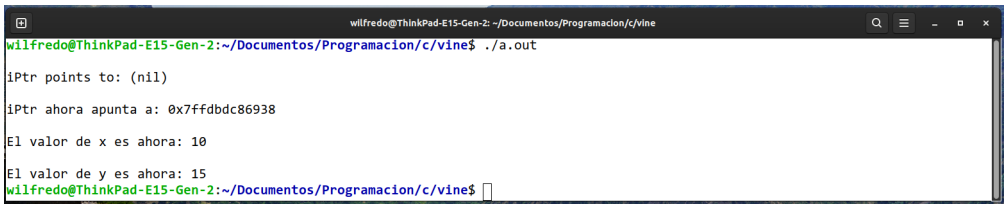

Figura 7.4: Uso de punteros y sentencias de asignación para demostrar la indirección.

Funciones y punteros

Uno de los mayores beneficios de utilizar punteros es la capacidad de pasar argumentos a funciones por referencia. De manera predeterminada, los argumentos se pasan por valor en C, lo que implica hacer una copia del argumento entrante para que la función lo utilice. Según los requisitos de almacenamiento del argumento entrante, este puede no ser el uso más eficiente de la memoria. Para demostrarlo, estudie el siguiente programa.

```c
#include <stdio.h>

int addTwoNumbers(int, int);

void main()
{
    int x = 0;
    int y = 0;

    printf("\nIntroduzca el primer número: ");
    scanf("%d", &x);

    printf("\nIntroduzca el segundo número: ");
    scanf("%d", &y);

    printf("\nEl resultado es %d\n", addTwoNumbers(x, y));
} //end main

int addTwoNumbers(int x, int y)
{
    return x + y;
} //end addTwoNumbers
```

En este programa, paso dos argumentos enteros a mi función `addTwoNumbers` en una función `printf()`. Este tipo de paso de argumentos se llama *paso por valor*. Más específicamente, C reserva espacio de memoria adicional para hacer una copia de las variables `x` e `y`, y las copias de `x` e `y` se envían a la función como argumentos. Pero, ¿qué significa esto? Me vienen a la mente dos preocupaciones importantes.

En primer lugar, pasar argumentos por valor no es el medio de programación más eficiente para programar en C. Hacer copias de dos variables enteras puede no parecer mucho trabajo, pero en el mundo real, los programadores de C deben esforzarse por minimizar el uso de memoria tanto como sea posible. Piense en el diseño de circuitos integrados donde sus recursos de memoria son muy limitados. En estas situaciones de desarrollo, hacer copias de variables para argumentos puede marcar una gran diferencia. Incluso si no está programando circuitos integrados, puede encontrar una degradación del rendimiento al pasar grandes cantidades de datos por valor (piense en matrices o estructuras de datos que contienen grandes cantidades de información, como datos de empleados).

En segundo lugar, cuando C pasa argumentos por valor, no puede modificar el contenido original de los parámetros entrantes. Esto se debe a que C ha hecho una copia de la variable original y, por lo tanto, solo se modifica la copia. Esto puede ser bueno y malo. Por ejemplo, es posible que no desee que la función receptora modifique el contenido original de la variable y, en este caso, se prefiere pasar argumentos por valor. Además, pasar argumentos por valor es una forma en que los programadores pueden implementar el ocultamiento de información, como se analiza en el Capítulo 5, "Programación estructurada".

Para demostrar aún más los conceptos de pasar argumentos por valor, estudie el siguiente programa y su salida que se muestra en la Figura 7.5.

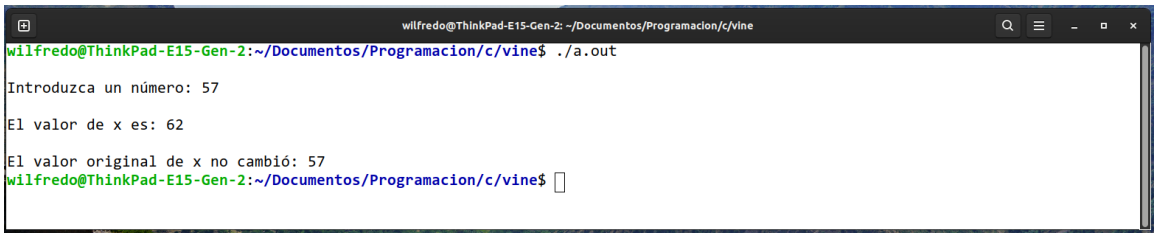

Figura 7.5: Implementación de la ocultación de información mediante el paso de argumentos por valor.

```
#include <stdio.h>

void demoPassByValue(int);

void main()
{
    int x = 0;

    printf("\nIntroduzca un número: ");
    scanf("%d", &x);

    demoPassByValue(x);

    printf("\nEl valor original de x no cambió: %d\n", x);
} //end main

void demoPassByValue(int x)
{
    x += 5;

    printf("\nEl valor de x es: %d\n", x);
} //end demoPassByValue
```

Después de estudiar el código, puedes ver que intento modificar el parámetro entrante incrementándolo en cinco. El argumento parece modificarse cuando imprimo el contenido en la función `printf()` de `demoPassByValue`. Sin embargo, cuando imprimo el contenido de la variable `x` desde la función `main()`, en realidad no se modifica.

Para resolver este problema, se utilizan punteros para pasar argumentos por *referencia*. Más específicamente, se puede pasar la dirección de la variable (argumento) a la función mediante indirección, como se muestra en el siguiente programa y en la Figura 7.6.

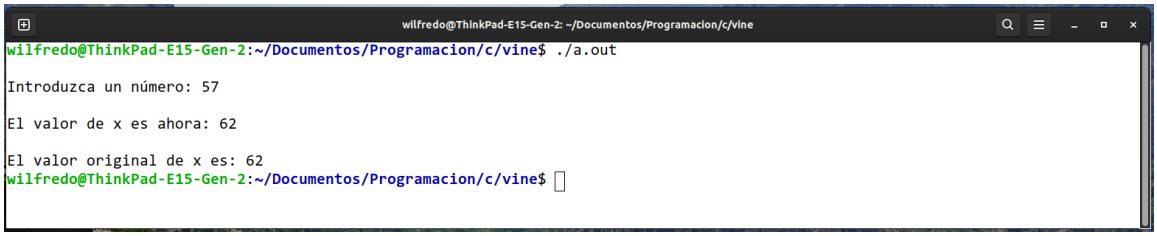

Figura 7.6: Pasar un argumento por referencia usando indirección.

```
#include <stdio.h>

void demoPassByReference(int *);

void main()
{
    int x = 0;

    printf("\nIntroduzca un número: ");
    scanf("%d", &x);

    demoPassByReference(&x);

    printf("\nEl valor original de x es: %d\n", x);
} //end main

void demoPassByReference(int *ptrX)
{
    *ptrX += 5;

    printf("\nEl valor de x es ahora: %d\n", *ptrX);
} //end demoPassByReference
```

Para pasar argumentos por referencia, debe tener en cuenta algunas diferencias sutiles en el programa anterior. La primera diferencia notable está en el prototipo de la función, como se muestra a continuación.

```
void demoPassByReference(int *);
```

Le digo a C que mi función tomará un puntero como argumento colocando el operador de indirección (*****) después del tipo de datos. La siguiente pequeña diferencia está en mi llamada de función, a la que le paso la dirección de memoria de la variable `x` colocando el operador unario (&) delante de la variable, como se muestra a continuación.

```
demoPassByReference(&x);
```

El resto de las actividades de indirección pertinentes se realizan en la implementación de la función, donde le indico al encabezado de la función que espere un parámetro entrante (puntero) que apunte a un valor entero. ¡Esto se conoce como pasar por referencia!

```
void demoPassByReference(int *ptrX)
```

Para modificar el contenido original del argumento, debo utilizar nuevamente el operador de indirección (*****), que le indica a C que quiero acceder al contenido de la posición de memoria contenida en la variable puntero. En concreto, incremento el contenido de la variable original en cinco, como se muestra a continuación.

```
*ptrX += 5;
```

Utilizo el operador de indirección en una función `printf()` para imprimir el contenido del puntero.

```
printf("\nThe value of x is now: %d\n", *ptrX);
```

Precaución: Si olvida colocar el operador de indirección (*****) delante de un puntero en una declaración de impresión que muestra un número con el especificador de conversión %d, C imprimirá una representación numérica de la dirección del puntero.

```
printf("\nThe value of x is now: %d\n", ptrX);  // Esto está mal
printf("\nThe value of x is now: %d\n", *ptrX); // Esto es correcto
```

Hasta ahora, es posible que se haya preguntado por qué es necesario colocar un *ampersand* (también conocido como el operador "dirección de") delante de las variables en las funciones `scanf()`. En términos simples, el operador de dirección proporciona a la función `scanf()` la dirección de memoria en la que C debe escribir los datos ingresados por el usuario.

Pasar arreglos o matrices a funciones

Quizás recuerde del Capítulo 6, "Matrices", que las matrices son agrupaciones de segmentos de memoria contiguos y que el nombre de la matriz en sí es un puntero a la primera ubicación de memoria en el segmento de memoria contiguo. Las matrices y los punteros están estrechamente relacionados en C. De hecho, pasar un nombre de matriz a un puntero asigna la primera ubicación de memoria de la matriz a la variable del puntero.

Para demostrar este concepto, el siguiente programa crea e inicializa una matriz de cinco elementos y declara un puntero que se inicializa al nombre de la matriz. La inicialización de un puntero a un nombre de matriz almacena la primera dirección de la matriz en el puntero, que se muestra en la Figura 7.7.

Después de inicializar el puntero, puedo acceder a la primera dirección de memoria de la matriz y al primer elemento de la matriz.

```
#include <stdio.h>

void main()
{
    int iArray[5] = {1,2,3,4,5};
    int *iPtr = iArray;

    printf("\nDirección del puntero: %p\n", iPtr);
    printf("Primera dirección de la matriz: %p\n", &iArray[0]);
    printf("\nEl puntero apunta a: %d\n", *iPtr);
    printf("El primer elemento de la matriz contiene: %d\n", iArray[0]);
}
```

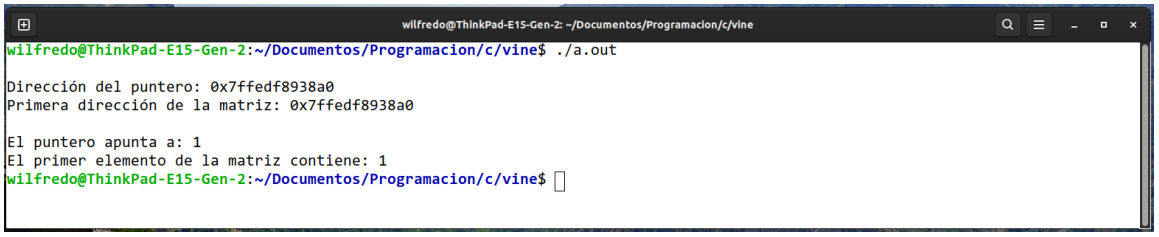

Figura 7.7: Asignación de la primera dirección de una matriz a un puntero.

Sabiendo que el nombre de una matriz contiene una dirección de memoria que apunta al primer elemento de la matriz, se puede suponer que los nombres de matriz se pueden tratar de forma muy similar a un puntero cuando se pasan matrices a funciones. Sin embargo, no es necesario tratar con operadores unarios (&) o de indirección (*) cuando se pasan matrices a funciones. Más importante aún, las matrices que se pasan como argumentos se pasan por referencia automáticamente. Ese es un concepto importante, por lo que lo enunciaré nuevamente. *Las matrices que se pasan como argumentos se pasan por referencia*.

Para pasar una matriz a una función, es necesario definir el prototipo y la definición de la función de modo que esperen recibir una matriz como argumento. El siguiente programa y su salida en la Figura 7.8 demuestran este concepto al pasar una matriz de caracteres a una función que calcula la longitud de la cadena entrante (matriz de caracteres).

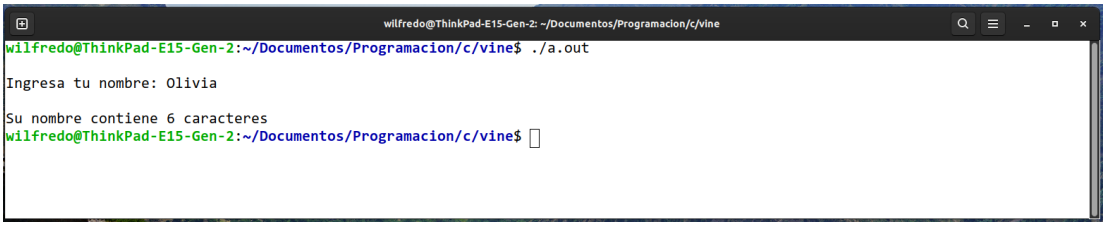

Figura 7.8: Pasar una matriz como argumento.

```
#include <stdio.h>

int nameLength(char []);

void main()
{
    char aName[20] = {'\0'};

    printf("\nIngresa tu nombre: ");
    scanf("%s", aName);

    printf("\nSu nombre contiene ");
    printf("%d caracteres\n", nameLength(aName));
} //end main

int nameLength(char name[])
{
    int x = 0;

    while ( name[x] != '\0' )
        x++;

    return x;
} //end nameLength
```

Puede construir su prototipo de función para recibir una matriz como argumento colocando corchetes vacíos en la lista de argumentos como se muestra nuevamente a continuación.

```
int nameLength(char []);
```

Este prototipo de función le indica a C que espere una matriz como argumento y, más específicamente, la función recibirá la primera dirección de memoria en la matriz. Al llamar a la función, solo necesito pasar el nombre de la matriz como se muestra en la siguiente declaración de impresión.

```
printf("%d characters\n", nameLength(aName));
```

Observe también que en el programa anterior no utilicé la dirección del operador (&) delante del nombre de la matriz en la función `scanf()`. Esto se debe a que un nombre de matriz en C ya contiene una dirección de memoria, que es la dirección del primer elemento de la matriz.

Este programa es una buena demostración de cómo pasar matrices como argumentos, pero no sirve para demostrar que las matrices se pasan por referencia. Para ello, estudie el siguiente programa y su salida en la Figura 7.9, que modifica el contenido de la matriz utilizando técnicas de paso por referencia.

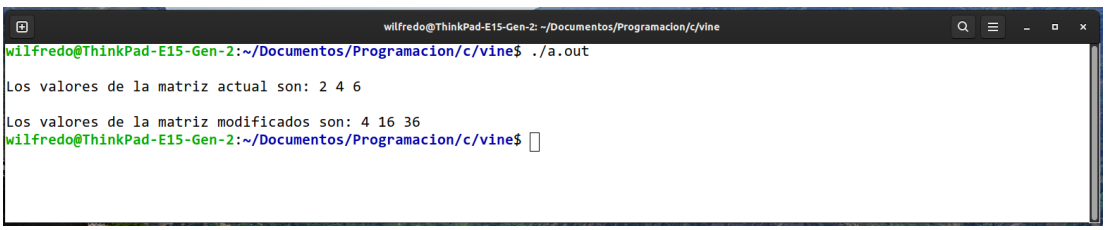

Figura 7.9: Modificación del contenido de una matriz mediante indirección y paso de matrices a funciones.

```
#include <stdio.h>

void squareNumbers(int []);

void main()
{
    int x;
    int iNumbers[3] = {2, 4, 6};

    printf("\nLos valores de la matriz actual son: ");

    for ( x = 0; x < 3; x++ )
        printf("%d ", iNumbers[x]); //print contents of array

    printf("\n");

    squareNumbers(iNumbers);

    printf("\nLos valores de la matriz modificados son: ");

    for ( x = 0; x < 3; x++ )
        printf("%d ", iNumbers[x]); //print modified array contents

    printf("\n");
} //end main

void squareNumbers(int num[])
{
    int x;

    for ( x = 0; x < 3; x++ )
        num[x] = num[x] * num[x]; //modify the array contents
} //end squareNumbers
```

El calificador `const`

Ahora sabe que los argumentos se pueden pasar a funciones de dos maneras: pasar por valor y pasar por referencia. Al pasar argumentos por valor, C hace una copia del argumento para que lo use la función receptora. También conocido como ocultamiento de información, esto evita el cambio directo del contenido del argumento entrante, pero crea una sobrecarga adicional al pasar estructuras grandes a funciones. Sin embargo, pasar argumentos por referencia proporciona a los programadores de C la capacidad de modificar el contenido de los argumentos mediante punteros.

Sin embargo, hay ocasiones en las que querrá la potencia y la velocidad de pasar argumentos por referencia sin el riesgo de seguridad de cambiar el contenido de una variable (argumento). Los programadores de C pueden lograr esto con el calificador `const`.

Quizás recuerde del Capítulo 2, “Tipos de datos primarios”, que el calificador `const` le permite crear variables de solo lectura. También puede usar el calificador `const` junto con punteros para lograr un argumento de solo lectura y, al mismo tiempo, lograr la capacidad de pasar por referencia. Para demostrarlo, el siguiente programa pasa un argumento de tipo entero de solo lectura a una función.

```
#include <stdio.h>

void printArgument(const int *);

void main()
{
    int iNumber = 5;

    printArgument(&iNumber); //pass read-only argument

} //end main

void printArgument(const int *num) //pass by reference, but read-only
{
    printf("\nRead Only Argument is: %d ", *num);
}
```

Si recuerda que las matrices se pasan a las funciones por referencia, debe saber que las implementaciones de funciones pueden alterar el contenido de la matriz original. Para evitar que se altere un argumento de matriz en una función, utilice el calificador `const` como se muestra en el siguiente programa.

```
#include <stdio.h>

void printArray(const int []);

void main()
{
    int iNumbers[3] = {2, 4, 6};

    printArray(iNumbers);
} //end main

void printArray(const int num[]) //pass by reference, but read-only
{
    int x;

    printf("\nArray contents are: ");

    for ( x = 0; x < 3; x++ )
        printf("%d ", num[x]);
}
```

Como se muestra en el programa anterior, puede pasar una matriz a una función como de solo lectura utilizando el calificador `const`. Para ello, debe indicar al prototipo de la función y a la definición de la función que debe esperar un argumento de solo lectura utilizando la palabra clave `const`.

Para demostrar el concepto de solo lectura, considere el siguiente programa, que intenta modificar el argumento de solo lectura en una declaración de asignación desde dentro de la función.

```
#include <stdio.h>

void modifyArray(const int []);

void main()
{
    int iNumbers[3] = {2, 4, 6};

    modifyArray(iNumbers);
} //end main

void modifyArray(const int num[])
{
    int x;

    for ( x = 0; x < 3; x++ )
        num[x] = num[x] * num[x]; //this will not work!
}
```

Observe el resultado en la Figura 7.10. El compilador de C me advierte con un error que estoy intentando modificar una ubicación de solo lectura.

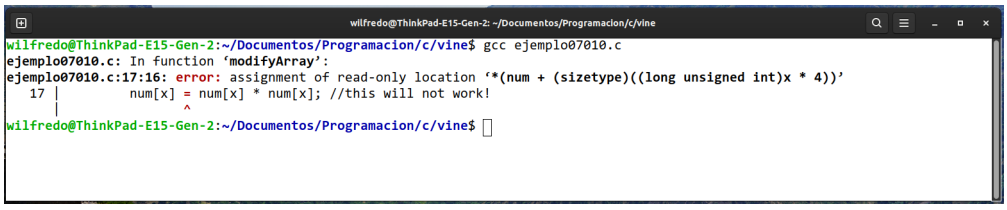

Figura 7.10: Activación de un error de compilación al intentar modificar una ubicación de memoria de solo lectura.

En resumen, el calificador `const` es una solución muy buena para proteger el contenido de los argumentos en un entorno de paso por referencia.

Programa del capítulo: Criptograma

Como se revela en la Figura 7.11, el programa basado en el capítulo Criptograma utiliza muchas de las técnicas que ha aprendido hasta ahora sobre punteros, matrices y funciones. Sin embargo, antes de pasar directamente al código del programa, la siguiente sección le brindará información básica sobre criptogramas y cifrado que lo ayudará a comprender la intención y la aplicación del programa basado en el capítulo.

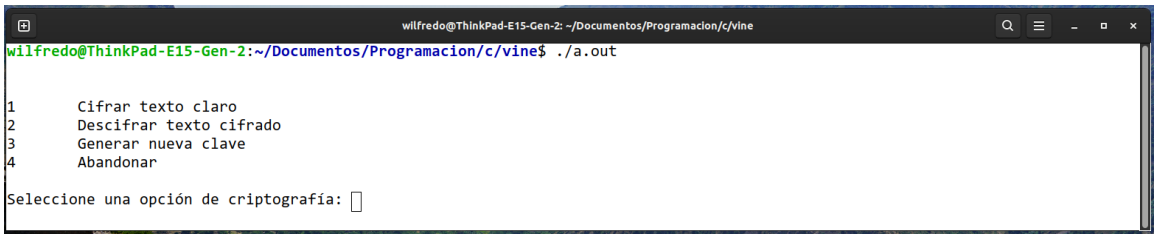

Figura 7.11: El programa basado en capítulos, Cryptogram, pasa matrices a funciones para cifrar y descifrar una palabra.

Introducción al cifrado

El cifrado es un subconjunto de tecnologías y ciencias que se incluyen en el ámbito de la criptografía. Al igual que el mundo de la programación informática, la criptografía y el cifrado tienen muchas palabras clave, definiciones y técnicas especializadas. Es prudente enumerar algunas de las definiciones más comunes en esta sección antes de continuar.

- *Criptografía*, el arte y la ciencia de proteger u ocultar mensajes.
- *Texto cifrado*, un mensaje que se oculta mediante la aplicación de un algoritmo de cifrado.
- *Texto claro*, texto simple o un mensaje legible por humanos.
- *Criptograma*, un mensaje cifrado o protegido.
- *Criptógrafo*, una persona o especialista que practica el cifrado o la protección de mensajes.
- *Cifrado*, el proceso mediante el cual el texto claro se convierte en texto cifrado.
- *Descifrado*, el proceso de convertir el texto cifrado en texto claro; generalmente implica conocer una clave o fórmula.
- *Clave*, la fórmula utilizada para descifrar un mensaje cifrado.

Las técnicas de cifrado se han aplicado durante cientos de años, aunque no fue hasta la llegada de Internet y la era informática que el cifrado ganó un nivel de atención pública sin precedentes.

Ya sea para proteger la información de su tarjeta de crédito con compras en "punto com" (*dot com*) o para mantener sus datos personales en su PC de casa seguros y protegidos, la informática genera un nuevo nivel de ansiedad para todos.

Afortunadamente, hay una serie de "buenos tipos" que intentan descubrir la combinación adecuada de informática, matemáticas y criptografía para construir sistemas seguros para datos privados y confidenciales. Estos "buenos tipos" son generalmente empresas informáticas y profesionales de la informática que intentan aliviar nuestra ansiedad mediante la promesa de datos intangibles o al menos ilegibles mediante el cifrado.

De manera simplificada, el cifrado utiliza muchas técnicas para convertir mensajes legibles por humanos, conocidos como texto claro, en un mensaje ilegible u oscuro llamado texto cifrado.

Los mensajes cifrados generalmente se bloquean y desbloquean con la misma clave o algoritmo. Las claves que se utilizan para bloquear y desbloquear mensajes seguros, o criptogramas, pueden almacenarse en el propio mensaje cifrado o utilizarse junto con fuentes externas, como números de cuenta y contraseñas.

El primer paso para cifrar cualquier mensaje es crear un algoritmo de cifrado. Un algoritmo de cifrado simplificado que se analiza en esta sección es la técnica o algoritmo denominado desplazamiento por n

(*shift by n*), que cambia la forma o el significado de un mensaje. El algoritmo de desplazamiento por n básicamente dice que hay que mover cada carácter hacia arriba o hacia abajo en una escala en una cierta cantidad de incrementos. Por ejemplo, puedo cifrar el siguiente mensaje desplazando cada carácter dos letras.

```
MeetMeAtSeven
```

Desplazando cada carácter dos letras se obtiene el siguiente resultado.

```
OggvOgCvUgxgp
```

La clave del algoritmo *shift by n* es el número utilizado para realizar el cambio (la n en *shift by n*). Sin esta clave, es difícil descifrar o descifrar el mensaje cifrado. ¡Es realmente muy sencillo! Por supuesto, no se trata de un algoritmo de cifrado que la CIA utilizaría para pasar datos a sus agentes y recibirlos, pero ya entiendes la idea.

El algoritmo de cifrado es tan bueno como su clave sea segura. Para demostrarlo, imagina que tu casa está cerrada y segura hasta que una persona no autorizada obtiene acceso físico a tu llave. Aunque tienes las mejores cerraduras que se pueden comprar, ya no ofrecen seguridad porque una persona no deseada tiene la llave para abrir tu casa.

Como verás en la siguiente sección, puedes crear tus propios procesos de cifrado simples con algoritmos de cifrado y claves de cifrado utilizando C, el conjunto de caracteres ASCII y el algoritmo *shift by n*.

Creación del programa de criptograma

Si utiliza sus conocimientos de los conceptos básicos de cifrado, podrá crear fácilmente un programa de cifrado en C que utilice el algoritmo de desplazamiento por n para generar una clave y cifrar y descifrar un mensaje.

Como se muestra en la Figura 7.11, se le presenta al usuario una opción para cifrar un mensaje, descifrar un mensaje o generar una nueva clave. Cuando se genera una nueva clave, el algoritmo de cifrado utiliza la nueva clave para desplazar cada letra del mensaje por n, donde n es el número aleatorio generado al seleccionar la opción "generar nueva clave". La misma clave se vuelve a utilizar para descifrar el mensaje.

Si genera una nueva clave después de cifrar un mensaje, es muy posible que no pueda descifrar el mensaje cifrado previamente. Esto demuestra la importancia de conocer la clave de cifrado en ambos extremos del criptograma.

A continuación se muestra todo el código necesario para crear el programa de criptograma.

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

// function prototypes
void encrypt(char [], int);
void decrypt(char [], int);

void main()
{
    char myString[21] = {0};
    int iSelection = 0;
    int iRand;

    srand(time(NULL));
    iRand = (rand() % 4) + 1; // random #, 1-4

    while ( iSelection != 4 )
    {
        printf("\n\n1\tCifrar texto claro\n");
        printf("2\tDescifrar texto cifrado\n");
        printf("3\tGenerar nueva clave\n");
        printf("4\tAbandonar\n");
        printf("\nSeleccione una opción de criptografía: ");
        scanf("%d", &iSelection);
```

```
        switch (iSelection)
        {
            case 1:
                printf("\nIntroduzca una palabra como texto sin cifrar para cifrar: ");
                scanf("%s", myString);
                encrypt(myString, iRand);
                break;
            case 2:
                printf("\nIntroduzca el texto cifrado para descifrarlo: ");
                scanf("%s", myString);
                decrypt(myString, iRand);
                break;
            case 3:
                iRand = (rand() % 4) + 1; // random #, 1-4
                printf("\nNueva clave generada\n");
                break;
        } //end switch
    } //end loop
} //end main

void encrypt(char sMessage[], int random)
{
    int x = 0;

    //encrypt the message by shifting each characters ASCII value
    while ( sMessage[x] )
    {
        sMessage[x] += random;
        x++;
    } //end loop

    x = 0;
    printf("\nEl mensaje cifrado es: ");

    //print the encrypted message
    while ( sMessage[x] )
    {
        printf("%c", sMessage[x]);
        x++;
    } //end loop
} //end encrypt function

void decrypt(char sMessage[], int random)
{
    int x = 0;

    x = 0;

    //decrypt the message by shifting each characters ASCII value
    while ( sMessage[x] )
    {
        sMessage[x] = sMessage[x] - random;
        x++;
    } //end loop

    x = 0;
    printf("\nEl mensaje descifrado es: ");

    //print the decrypted message
    while ( sMessage[x] )
    {
        printf("%c", sMessage[x]);
        x++;
    } //end loop
} //end decrypt function
```

Resumen

- Los punteros son variables que contienen una dirección de memoria que apunta a otra variable.

- Coloque el operador de indirección (*) delante del nombre de la variable para declarar un puntero.
- El operador unario (&) se conoce a menudo como el operador de “dirección de”.
- Las variables de puntero siempre deben inicializarse con la dirección de memoria de otra variable, con 0 o con la palabra clave NULL.
- Puede imprimir la dirección de memoria (*address memory*) de los punteros utilizando el especificador de conversión %p.
- De forma predeterminada, los argumentos se pasan por valor en C, lo que implica hacer una copia del argumento entrante para que lo utilice la función.
- Los punteros se pueden utilizar para pasar argumentos por referencia.
- Pasar un nombre de matriz a un puntero asigna la primera ubicación de memoria de la matriz a la variable de puntero. De manera similar, inicializar un puntero a un nombre de matriz almacena la primera dirección de la matriz en el puntero.
- Puede utilizar el calificador const junto con punteros para lograr un argumento de solo lectura y, al mismo tiempo, lograr la capacidad de pasar por referencia.