Programación Estructurada

La programación estructurada es un concepto arraigado en la historia de la programación informática que permite a los programadores dividir los problemas en componentes pequeños y fáciles de entender que, con el tiempo, formarán un sistema completo. En este capítulo, le mostraré cómo utilizar conceptos de programación estructurada, como el diseño descendente, y técnicas de programación, como la creación de sus propias funciones, para generar código eficiente y reutilizable en sus programas.

Introducción a la Programación Estructurada

La programación estructurada permite a los programadores dividir sistemas complejos en componentes manejables. En C, estos componentes se conocen como *funciones*, que son el núcleo de este capítulo. En esta sección, le brindaré información general sobre técnicas y conceptos comunes de programación estructurada. Después de leer esta sección, estará listo para crear sus propias funciones en C.

La programación estructurada contiene muchos conceptos que van desde la teoría hasta la aplicación.

Muchos de estos conceptos son intuitivos, mientras que otros tardarán un tiempo en asimilarse y arraigarse.

Los conceptos de programación estructurada más relevantes para este texto son los siguientes:

- Diseño de arriba hacia abajo
- Reutilización de código
- Ocultación de información

Diseño de arriba hacia abajo

Común con los lenguajes procedimentales como C, el diseño de *arriba hacia abajo / top-down* permite a los analistas y programadores definir declaraciones detalladas sobre las tareas específicas de un sistema. Los expertos en diseño de arriba hacia abajo sostienen que los humanos tienen capacidades limitadas para realizar múltiples tareas. Aquellos que se destacan en la multitarea y disfrutan del caos que conlleva generalmente no son programadores. Los programadores tienden a trabajar en un solo problema con detalles tediosos. Para demostrar el diseño de arriba hacia abajo, utilizaré un cajero automático como ejemplo.

Supongamos que su jefe, que no es técnico, le pide que programe el software para un nuevo sistema de cajero automático para el Big Money Bank. Probablemente se preguntará por dónde empezar, ya que es una tarea grande llena de complejidades y muchos detalles.

El diseño de arriba hacia abajo puede ayudarlo a diseñar su salida del bosque oscuro y traicionero del diseño de sistemas. Los siguientes pasos demuestran el proceso de diseño de arriba hacia abajo.

1. Divida el problema en componentes pequeños y manejables, comenzando desde arriba. En C, el componente superior es la función main() desde la que se llaman otros componentes.
2. Identifique todos los componentes principales. Para el ejemplo del cajero automático, suponga que hay cuatro componentes principales:
    - Mostrar saldo
    - Depositar fondos
    - Transferir fondos

- Retirar fondos

3. Ahora que ha separado los componentes principales del sistema, puede visualizar el trabajo involucrado. Descomponga un componente principal a la vez y hágalo más manejable y menos complejo.
4. El componente de retiro de fondos se puede dividir en partes más pequeñas, como las siguientes:
   - Obtener el saldo disponible
   - Comparar el saldo disponible con el monto solicitado
   - Actualizar la cuenta del cliente
   - Distribuir los fondos aprobados
   - Rechazar la solicitud
   - Imprimir el recibo
5. Vaya más allá con el proceso de descomposición y divida el componente de "distribución de fondos aprobados" en partes aún más pequeñas:
   - Verificar la existencia de fondos en el cajero automático
   - Iniciar procesos mecánicos
   - Actualizar los registros bancarios

La Figura 5.1 muestra un proceso de muestra para descomponer el sistema de cajeros automáticos.

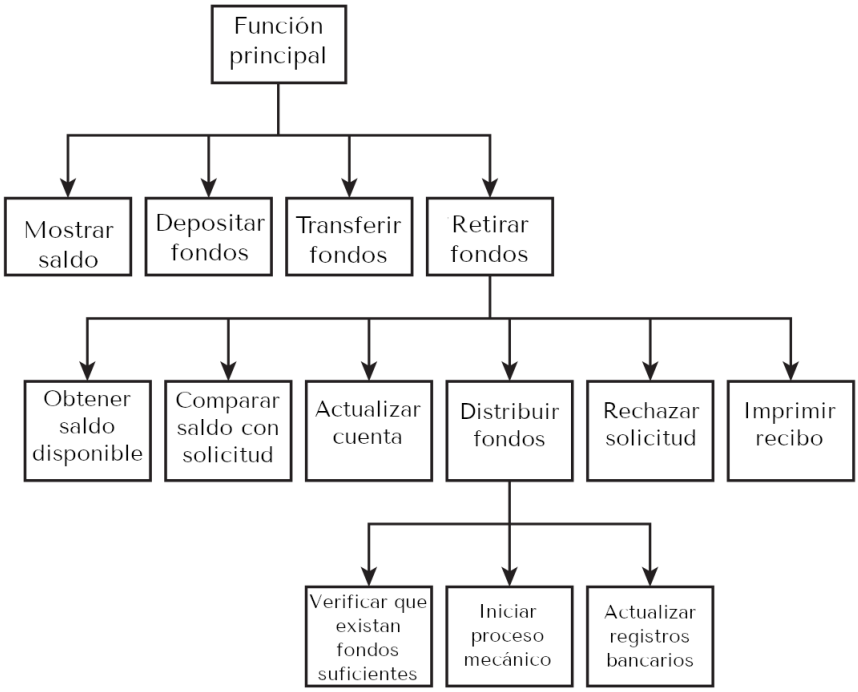

Figura 5.1: Descomposición del sistema de cajeros automáticos mediante diseño descendente.

Con mi sistema de cajeros automáticos descompuesto en componentes manejables, me siento un poco menos ansioso por las próximas tareas de programación. Además, ahora puedo asignarme componentes más pequeños y manejables para comenzar a programar.

Espero que veas lo mucho más fácil que es pensar en implementar un solo componente, como verificar la existencia de fondos en los cajeros automáticos, que la abrumadora tarea de construir un sistema de cajeros automáticos completo. Además, en el nivel descompuesto, varios programadores pueden trabajar en el mismo sistema sin conocer los detalles inmediatos de las tareas de programación de los demás.

Durante tu carrera de programación, estoy seguro de que te enfrentarás a ideas complejas similares que deben implementarse con lenguajes de programación. Si se usa correctamente, el diseño descendente puede ser una herramienta útil para hacer que tus problemas sean más fáciles de entender e implementar.

Reutilización de código

En el mundo del desarrollo de aplicaciones, la reutilización de código se implementa como funciones en C. Específicamente, los programadores crean funciones definidas por el usuario para problemas que generalmente necesitan soluciones de uso frecuente. Para demostrarlo, considere la siguiente lista de componentes y subcomponentes del ejemplo del cajero automático de la sección anterior.

- Obtener el saldo disponible
- Comparar el saldo disponible con el monto solicitado
- Actualizar la cuenta del cliente
- Distribuir los fondos aprobados
- Rechazar la solicitud
- Imprimir el recibo

Dado el sistema del cajero automático, ¿cuántas veces cree que ocurriría el problema de actualización de la cuenta del cliente para un cliente o transacción? Según el sistema del cajero automático, el componente de actualización de la cuenta del cliente se puede llamar varias veces. Un cliente puede realizar muchas transacciones mientras está en un cajero automático. La siguiente lista muestra una posible cantidad de transacciones que un cliente podría realizar en una sola visita a un cajero automático.

- Depositar dinero en una cuenta corriente
- Transferir fondos de una cuenta corriente a una cuenta de ahorros
- Retirar dinero de una cuenta corriente
- Imprimir el saldo

Al menos cuatro ocasiones requieren que acceda al saldo del cliente. Escribir estructuras de código cada vez que necesita acceder al saldo de alguien no tiene sentido, porque puede escribir una función que contenga la lógica y las estructuras para manejar este procedimiento y luego reutilizar esa función cuando sea necesario. Poner todo el código en una función que pueda llamarse repetidamente le ahorrará tiempo de programación inmediatamente y en el futuro si es necesario realizar cambios en la función.

Permítanme hablar de otro ejemplo en el que se utiliza la función `printf()` (con la que ya están familiarizados) que demuestra la reutilización de código. En este ejemplo, un programador ya ha implementado el código y las estructuras necesarias para imprimir texto sin formato en la salida estándar. Simplemente se utiliza la función `printf()` llamando a su nombre y pasándole los caracteres deseados. Como la función `printf()` existe en un módulo o biblioteca, se la puede llamar repetidamente sin conocer los detalles de su implementación o, en otras palabras, cómo se creó. ¡La reutilización de código es realmente el mejor amigo de un programador!

Ocultación de información

La ocultación de información es un proceso conceptual mediante el cual los programadores ocultan detalles de implementación en funciones. Las funciones pueden verse como cajas negras. Una caja negra es simplemente un componente, lógico o físico, que realiza una tarea. No se sabe cómo la caja negra realiza (implementa) la tarea; simplemente se sabe que funciona cuando es necesario. La Figura 5.2 representa el concepto de caja negra.

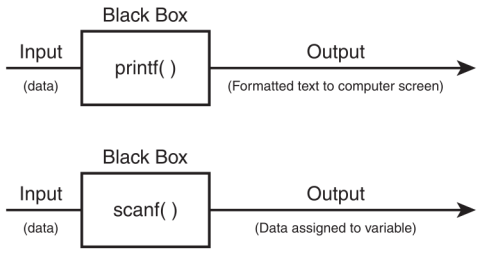

Figura 5.2: Demostración del concepto de caja negra.

Considere los dos dibujos de caja negra en la Figura 5.2. Cada caja negra describe un componente; en este caso, los componentes son `printf()` y `scanf()`. La razón por la que considero que las dos funciones `printf()` y `scanf()` son cajas negras es porque no necesita saber qué hay dentro de ellas (cómo están hechas), solo necesita saber qué toman como entrada y qué devuelven como salida. En otras palabras, comprender cómo usar una función sin saber cómo está construida es un buen ejemplo de ocultamiento de información.

Muchas de las funciones que ha usado hasta ahora demuestran la utilidad del ocultamiento de información.

La Tabla 5.1 enumera funciones de biblioteca más comunes que implementan el ocultamiento de información en la programación estructurada.

Tabla 5.1: Funciones comunes de la biblioteca

Nombre de la biblioteca	Nombre de la función	Descripción
Entrada/salida estándar	`scanf()`	Lee datos del teclado
Entrada/salida estándar	`printf()`	Imprime datos en el monitor de la computadora
Manejo de caracteres	`isdigit()`	Prueba de caracteres decimales
Manejo de caracteres	`islower()`	Prueba de letras minúsculas
Manejo de caracteres	`isupper()`	Prueba de letras mayúsculas
Manejo de caracteres	`tolower()`	Convierte caracteres a minúsculas
Manejo de caracteres	`toupper()`	Convierte caracteres a mayúsculas
Matemáticas	`exp()`	Calcula el exponencial
Matemáticas	`pow()`	Calcula un número elevado a una potencia
Matemáticas	`sqrt()`	Calcula la raíz cuadrada

Si aún te desanima la noción de ocultamiento de información o cajas negras, considera la siguiente pregunta. ¿La mayoría de la gente sabe cómo funciona el motor de un automóvil? Probablemente no, la mayoría de las personas solo se preocupan por saber cómo operar un automóvil. Afortunadamente, los automóviles modernos proporcionan una interfaz desde la cual puede usar el automóvil fácilmente, al tiempo que oculta sus detalles de implementación. En otras palabras, uno podría considerar el motor del automóvil como la caja negra. Solo sabe lo que la caja negra toma como entrada (gasolina) y lo que da como salida (movimiento).

Volviendo a la función `printf()`, ¿qué sabe realmente sobre ella? Sabes que la función `printf()` imprime caracteres que proporcionas en la pantalla de la computadora. Pero, ¿sabes cómo funciona realmente la función `printf()`? Probablemente no, y no necesitas saberlo. Ese es un concepto clave de ocultamiento de información.

En la programación estructurada, se crean componentes que se pueden reutilizar (reutilización de código) y que incluyen una interfaz que otros programadores sabrán cómo usar sin necesidad de entender cómo se construyeron (ocultamiento de información).

Prototipos de funciones

Los *prototipos de funciones* le dicen a C cómo se construirá y utilizará su función. Es una práctica de programación común construir su prototipo de función antes de que se construya la función real. Esa afirmación fue tan importante que vale la pena volver a mencionarla. **Es una práctica de programación común construir el prototipo de la función antes de construir la función real**.

Los programadores deben pensar en el propósito deseado de la función, cómo recibirá la entrada y cómo y qué devolverá. Para demostrarlo, observe el siguiente prototipo de función.

```
float addTwoNumbers(int, int);
```

Este prototipo de función le dice a C lo siguiente sobre la función:

- **El tipo de datos que devuelve la función**: en este caso, se devuelve un tipo de datos `float`
- **La cantidad de parámetros recibidos**: en este caso, dos
- **Los tipos de datos de los parámetros**: en este caso, ambos parámetros son tipos de datos enteros
- **El orden de los parámetros**

Las implementaciones de funciones y sus prototipos pueden variar. No siempre es necesario enviar entradas como parámetros a las funciones, ni siempre es necesario que las funciones devuelvan valores.

En estos casos, los programadores dicen que las funciones no tienen parámetros y/o no tienen un valor de retorno. Los dos prototipos de función siguientes demuestran el concepto de funciones con la palabra clave void.

```
void printBalance(int);    // prototipo de función
```

La palabra clave `void` en el ejemplo anterior le dice a C que la función printBalance no devolverá un valor. En otras palabras, esta función no tiene un valor de retorno.

```
int createRandomNumber(void);    // prototipo de función
```

La palabra clave `void` en la lista de parámetros de la función `createRandomNumber` le dice a C que esta función no aceptará ningún parámetro, pero devolverá un valor entero. En otras palabras, esta función no tiene parámetros.

Los prototipos de función deben colocarse fuera de la función `main()` y antes de que comience la función `main()`, como se muestra a continuación.

```
#include <stdio.h>

int addTwoNumbers(int, int);    // prototipo de función

main()

{

}
```

No hay límite para la cantidad de prototipos de funciones que puede incluir en su programa C. Considere el siguiente bloque de código, que implementa cuatro prototipos de funciones.

```
#include <stdio.h>

int addTwoNumbers(int, int);         // function prototype
int subtractTwoNumbers(int, int);    // function prototype
int divideTwoNumbers(int, int);      // function prototype
int multiplyTwoNumbers(int, int);    // function prototype

main()
{

}
```

Definiciones de funciones

Le he mostrado cómo los programadores de C crean los planos para funciones definidas por el usuario con prototipos de funciones. En esta sección, le mostraré cómo crear funciones definidas por el usuario utilizando los prototipos de funciones.

Las definiciones de funciones implementan el prototipo de función. De hecho, la primera línea de la definición de función (también conocida como encabezado) se parece al prototipo de función, con pequeñas excepciones. Para demostrarlo, estudie el siguiente bloque de código.

```
#include <stdio.h>

int addTwoNumbers(int, int);    //  function prototype

void main()
{
    printf("Nothing happening in here.");
}

//function definition

int addTwoNumbers(int operand1, int operand2)
```

```
{
    return operand1 + operand2;
}
```

Tengo dos funciones independientes y completas: `main()` y `addTwoNumbers()`. El prototipo de la función y la primera línea de la definición de la función (el encabezado de la función) tienen un parecido sorprendente. La única diferencia es que el encabezado de la función contiene los nombres de las variables reales para los parámetros y el prototipo de la función contiene solo el tipo de datos de la variable. La definición de la función no contiene un punto y coma después del encabezado (a diferencia de su prototipo). De manera similar a la función `main()`, la definición de la función debe incluir una llave de inicio y una de fin.

En C, las funciones pueden devolver un valor a la instrucción que la llama. Para devolver un valor, utilice la palabra clave `return`, que inicia el proceso de valor de retorno. En la siguiente sección, aprenderá a llamar a una función que recibe su valor de retorno.

Consejo

Puede utilizar la palabra clave `return` de una de dos maneras: primero, puede utilizar la palabra clave `return` para pasar un valor o el resultado de una expresión a la instrucción que la llama. Segundo, puede utilizar la palabra clave `return` sin ningún valor o expresión para devolver el control del programa a la instrucción que la llama.

Sin embargo, a veces no es necesario que una función devuelva ningún valor. Por ejemplo, el siguiente programa crea una función simplemente para comparar los valores de dos números.

```
//definición de función

int compareTwoNumbers(int num1, int num2)
{
    if (num1 < num2)
        printf("\n%d is less than %d\n", num1, num2);
    else if (num1 == num2)
           printf("\n%d is equal to %d\n", num1, num2);
         else
           printf("\n%d is greater than %d\n", num1, num2);
}
```

Observe que en la definición de función anterior la función `compareTwoNumbers()` no devuelve un valor. Para demostrar con más detalle el proceso de creación de funciones, estudie el siguiente programa que crea un encabezado de informe.

```
//definición de función

void printReportHeader()
{
    printf("\nColumn1\tColumn2\tColumn3\tColumn4\n");
}
```

Para crear un programa que implemente múltiples definiciones de funciones, cree cada definición de función como se indica en cada prototipo de función. El siguiente programa implementa la función `main()`, que no hace nada importante, y luego crea dos funciones para realizar operaciones matemáticas básicas y devolver un resultado.

```
#include <stdio.h>

int addTwoNumbers(int, int);    //  function prototype
int subtractTwoNumbers(int, int);    //  function prototype

void main()
{
    printf("Nothing happening in here.");
}

//function definition
int addTwoNumbers(int num1, int num2)
{
    return num1 + num2;
}
```

```
//function definition
int subtractTwoNumbers(int num1, int num2)
{
    return num1 - num2;
}
```

Llamadas a funciones

Ahora es el momento de poner a funcionar sus funciones con llamadas a funciones. Hasta este punto, es posible que se haya preguntado cómo usted o su programa utilizarán estas funciones. Trabaja con sus funciones definidas por el usuario de la misma manera que trabaja con otras funciones de la biblioteca C, como `printf()` y `scanf()`.

Usando la función `addTwoNumbers()` de la sección anterior, incluyo una única llamada a función en mi función `main()` como se muestra a continuación.

```
#include <stdio.h>

int addTwoNumbers(int, int);    //function prototype

void main()
{
    int iResult;

    iResult = addTwoNumbers(5, 5); //function call
}

// definición de función
int addTwoNumbers(int operand1, int operand2)
{
    return operand1 + operand2;
}
```

`addTwoNumbers(5, 5)` llama a la función y le pasa dos parámetros enteros. Cuando C encuentra una llamada a una función, redirige el control del programa a la definición de la función. Si la definición de la función devuelve un valor, toda la declaración de llamada a la función se reemplaza por el valor de retorno.

En otras palabras, toda la declaración `addTwoNumbers(5, 5)` se reemplaza por el número 10. En el programa anterior, el valor devuelto de 10 se asigna a la variable entera `iResult`.

Las llamadas a funciones también se pueden colocar en otras funciones. Para demostrarlo, estudie el siguiente bloque de código que utiliza la misma llamada a la función `addTwoNumbers()` dentro de una función `printf()`.

```
#include <stdio.h>

int addTwoNumbers(int, int);    // function prototype

void main()
{
    printf("\nThe result is %d", addTwoNumbers(5, 5));
}

//function definition
int addTwoNumbers(int operand1, int operand2)
{
    return operand1 + operand2;
}
```

En la llamada de función anterior, codifiqué dos números como parámetros. Puedes ser más dinámico con las llamadas de función al pasar variables como parámetros, como se muestra a continuación.

```
#include <stdio.h>

int addTwoNumbers(int, int);    // function prototype
```

```c
void main()
{
    int num1, num2;

    printf("\nEntre el primer número: ");
    scanf("%d", &num1);

    printf("\nEntre el segundo número: ");
    scanf("%d", &num2);
    printf("\nEl resultado es %d\n", addTwoNumbers(num1, num2));
}

//function definition
int addTwoNumbers(int operand1, int operand2)

{
    return operand1 + operand2;
}
```

La salida del programa anterior se muestra en la Figura 5.3.

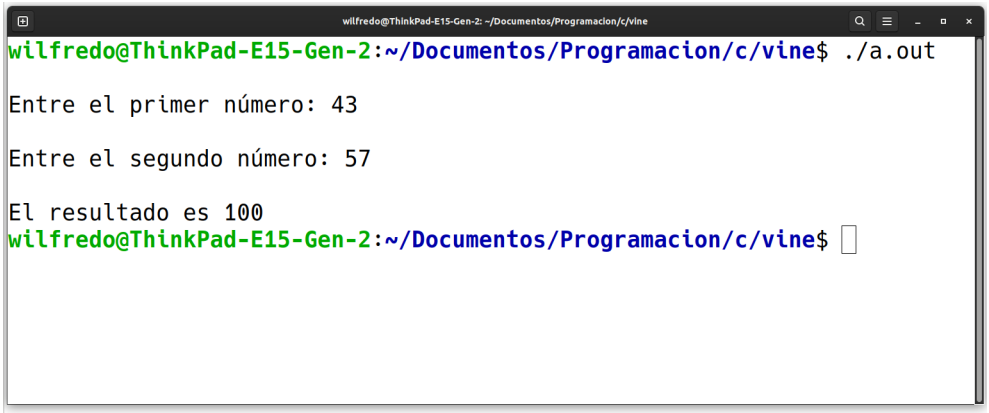

Figura 5.3: Pasar variables como parámetros a funciones definidas por el usuario.

A continuación se muestra la función `printReportHeader()` que imprime un encabezado de informe utilizando la secuencia de escape
`t` para imprimir una tabulación entre palabras.

```c
#include <stdio.h>

void printReportHeader();    // function prototype

void main()
{
    printReportHeader();
}

//function definition
void printReportHeader()
{
    printf("\nColumn1\tColumn2\tColumn3\tColumn4\n");
}
```

Llamar a una función que no requiere parámetros o que no devuelve ningún valor es tan sencillo como llamar a su nombre con paréntesis vacíos.

Precaución

Si no se utilizan paréntesis en llamadas de función que no tienen parámetros, pueden producirse errores de compilación u operaciones de programa no válidas. Considere las dos llamadas de función siguientes.

```c
printReportHeader;    // Llamada de función incorrecta
printReportHeader();  // Llamada de función correcta
```

La primera llamada de función no provocará un error de compilación, pero no podrá ejecutar la llamada de función a `printReportHeader`. Sin embargo, la segunda llamada de función contiene los paréntesis vacíos y llamará correctamente a `printReportHeader()`.

Ámbito de Variable

El *ámbito de variable* identifica y determina la duración de vida de cualquier variable en cualquier lenguaje de programación. Cuando una variable pierde su ámbito, significa que se pierde su valor de datos. Analizaré dos tipos comunes de ámbitos de variables en C, local y global, para que comprenda mejor la importancia del ámbito de las variables.

Ámbito local

Sin saberlo, ha estado utilizando variables de ámbito local desde el Capítulo 2, "Tipos de datos primarios". Las variables locales se definen en funciones, como la función `main()`, y pierden su ámbito cada vez que se ejecuta la función, como se muestra en el siguiente programa:

```
#include <stdio.h>
void main()
{
    int num1;

    printf("\nEntre un número: ");
    scanf("%d", &num1);
    printf("\nUsted entró %d\n ", num1);
}
```

Cada vez que se ejecuta el programa anterior, C asigna espacio de memoria para la variable entera `num1` con su declaración de variable. Los datos almacenados en la variable se pierden cuando finaliza la función `main()`.

Debido a que las variables de ámbito local están vinculadas a sus funciones de origen, puede reutilizar los nombres de las variables en otras funciones sin correr el riesgo de sobrescribir los datos. Para demostrarlo, revise el siguiente código de programa y su salida en la Figura 5.4.

```
#include <stdio.h>

int getSecondNumber();    //function prototype

void main()
{
    int num1;
    printf("\nEntre un número: ");
    scanf("%d", &num1);
    printf("\nUsted entró %d y %d\n ", num1, getSecondNumber());
}

//function definition
int getSecondNumber ()
{
    int num1;

    printf("\nEntre un segundo número: ");
    scanf("%d", &num1);

    return num1;
}
```

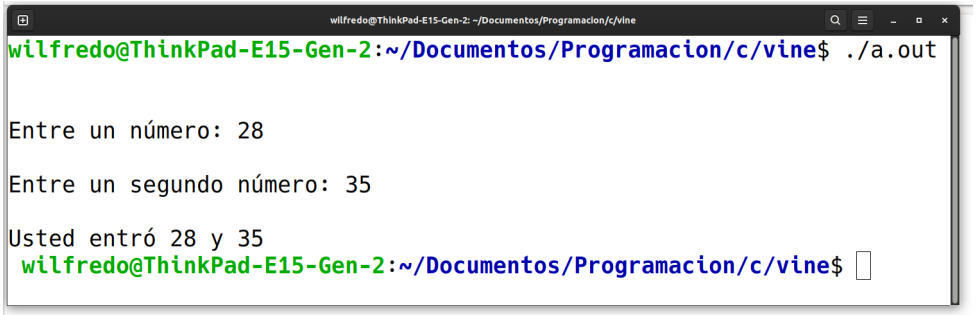

Figura 5.4: Uso del mismo nombre de variable de ámbito local en diferentes funciones.

Como la variable `num1` tiene un ámbito local en cada función, no hay problemas con la sobrescritura de datos. En concreto, la variable `num1` en cada función es una dirección de memoria independiente y, por lo tanto, cada una es una variable única.

Ámbito Global

Las variables de ámbito local se pueden reutilizar en otras funciones sin dañar el contenido de las demás. Sin embargo, a veces es posible que desee compartir datos entre funciones. Para respaldar el concepto de compartir datos, puede crear y utilizar *variables globales*.

Las variables globales se crean y definen fuera de cualquier función, incluida la función `main()`. Para mostrar cómo funcionan las variables globales, examine el siguiente programa.

```
#include <stdio.h>

void printLuckyNumber();     //function prototype

int iLuckyNumber;            //global variable

void main()
{
    printf("\nEnter your lucky number: ");
    scanf("%d", &iLuckyNumber);
    printLuckyNumber();
}

//function definition
void printLuckyNumber()
{
    printf("\nYour lucky number is: %d\n", iLuckyNumber);
}
```

La variable `iLuckyNumber` es global porque se crea fuera de cualquier función, incluida la función `main()`. Puedo asignar datos a la variable global en una función y hacer referencia al mismo espacio de memoria en otra función. Sin embargo, no es prudente utilizar las variables globales de forma liberal, ya que pueden ser propensas a errores y desviarse de la protección de los datos. El uso de variables globales permite que todas las funciones de un archivo de programa tengan acceso a los mismos datos, lo que va en contra del concepto de ocultamiento de información.

Programa del capítulo: Trivia

Como se muestra en la Figura 5.5, el juego Trivia utiliza muchos de los conceptos y técnicas de este capítulo.

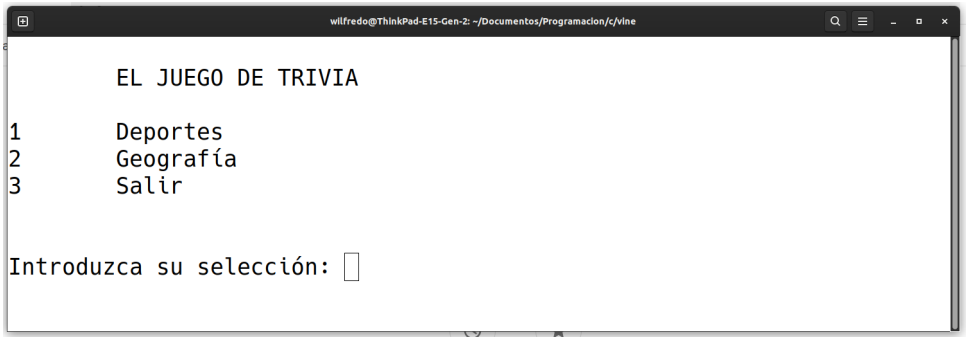

Figura 5.5: Demostración de conceptos basados en el capítulo con el juego Trivia.

El juego Trivia utiliza prototipos de funciones, definiciones de funciones, llamadas de funciones y una variable global para crear un juego simple y divertido. Los jugadores seleccionan una categoría de trivia del menú principal y se les hace una pregunta. El programa responde si la respuesta es correcta o incorrecta.

Cada categoría de trivia se divide en una función que implementa la lógica de preguntas y respuestas. También hay una función definida por el usuario que crea una utilidad de pausa.

A continuación se muestra todo el código necesario para crear el juego Trivia.

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

/*****************************************
FUNCTION PROTOTYPES
*****************************************/

int sportsQuestion(void);
int geographyQuestion(void);
void pause(int);
/*****************************************/

/*****************************************
GLOBAL VARIABLE
*****************************************/

int giResponse = 0;
/*****************************************/

void main()
{
    do {
        system("clear");
        printf("\n\tEL JUEGO DE TRIVIA\n\n");
        printf("1\tDeportes\n");
        printf("2\tGeografía\n");
        printf("3\tSalir\n");
        printf("\n\nIntroduzca su selección: ");
        scanf("%d", &giResponse);

        switch(giResponse) {
            case 1:
                if (sportsQuestion() == 4)
                    printf("\nCorrecto!\n");
                else
                    printf("\nIncorrecto\n");
                pause(2);
                break;
            case 2:
                if (geographyQuestion() == 2)
                    printf("\nCorrecto!\n");
                else
                    printf("\nIncorrecto\n");
                pause(2);
                break;
        } //end switch
    } while ( giResponse != 3 );
} //end main function

/*********************************************************
FUNCTION DEFINITION
*********************************************************/

int sportsQuestion(void)
{
    int iAnswer = 0;

    system("clear");
    printf("\tPregunta Deportiva\n");
    printf("\n¿A qué universidad asistió la estrella de la NFL Deion Sanders? ");
    printf("\n\n1\tUniversidad de Miami\n");
    printf("2\tUniversidad Estatal de California\n");
    printf("3\tUniversidad de Indiana\n");
    printf("4\tUniversidad Estatal de Florida\n");
    printf("\nIntroduzca su selección: ");
    scanf("%d", &iAnswer);

    return iAnswer;
} //end sportsQuestion function

/*********************************************************
```

```
FUNCTION DEFINITION
*******************************************************/

int geographyQuestion(void)
{
    int iAnswer = 0;

    system("clear");
    printf("\tPregunta de Geografía\n");
    printf("\n¿Cuál es la capital del estado de Florida? ");
    printf("\n\n1\tPensecola\n");
    printf("2\tTallahassee\n");
    printf("3\tJacksonville\n");
    printf("4\tMiami\n");
    printf("\nIntroduzca su selección: ");
    scanf("%d", &iAnswer);

    return iAnswer;
} //end geographyQuestion function

/*********************************************************
FUNCTION DEFINITION
*********************************************************/

void pause(int inNum)
{
    int iCurrentTime = 0;
    int iElapsedTime = 0;
    iCurrentTime = time(NULL);

    do {
        iElapsedTime = time(NULL);
    } while ( (iElapsedTime - iCurrentTime) < inNum );
} // end pause function
```

Resumen

- La programación estructurada permite a los programadores dividir sistemas complejos en componentes manejables.
- El diseño descendente divide el problema en componentes pequeños y manejables, comenzando desde arriba.
- La reutilización del código se implementa como funciones en C.
- El ocultamiento de información es un proceso conceptual mediante el cual los programadores ocultan detalles de implementación en funciones.
- Los prototipos de función le dicen a C cómo se construirá y utilizará su función.
- Es una práctica de programación común construir su prototipo de función antes de que se construya la función real.
- Los prototipos de función le dicen a C el tipo de datos devueltos por la función, la cantidad de parámetros recibidos, los tipos de datos de los parámetros y el orden de los parámetros.
- Las definiciones de función implementan el prototipo de función.
- En C, las funciones pueden devolver un valor a la declaración que realiza la llamada. Para devolver un valor, use la palabra clave return, que inicia el proceso de valor de retorno.
- Puede usar la palabra clave `return` para pasar un valor o resultado de expresión a la declaración que realiza la llamada o puede usar la palabra clave `return` sin ningún valor o expresión para devolver el control del programa a la declaración que realiza la llamada.
- Si no se utilizan paréntesis en llamadas de función sin parámetros, pueden producirse errores de compilación u operaciones de programa no válidas.

- El ámbito de las variables identifica y determina la duración de cualquier variable en cualquier lenguaje de programación. Cuando una variable pierde su ámbito, se pierde su valor de datos.
- Las variables locales se definen en funciones, como la función main(), y pierden su ámbito cada vez que se ejecuta la función.
- Las variables de ámbito local se pueden reutilizar en otras funciones sin dañar el contenido de las demás.
- Las variables globales se crean y definen fuera de cualquier función, incluida la función `main()`.