INTRODUCCIÓN

Una función, en el sentido académico de la palabra, es un proceso con entradas y salidas bien definidas. Su implantación práctica debe ir encaminada a la realización de bloques bien definidos en cuanto al proceso que realicen y los insumos y consumos que requiera. En otras palabras, una función es como una receta de cocina: tienes ingredientes (entradas) y tienes un resultado definido (salida).

A lo largo de este curso, ya hemos utilizado funciones sin darnos cuenta. La más fácil de identificar sería la función *main()*. Esta función es el punto de entrada al programa y también el punto de salida, es decir que la función *main()* será la encargada de inicializar el programa y de proporcionarnos el producto final de este. Sin embargo, no es la única función que hemos utilizado, *scanf()*, *printf()* y *getch()* son otras funciones que hemos empleado a lo largo del curso para realizar una acción en particular.

Como podemos darnos cuenta, el lenguaje C está basado en el concepto de funciones. Un programa en C es una colección de una o más funciones, en donde cada una tiene un nombre y un objetivo bien definido. Estas las podemos clasificar en dos grandes grupos: aquellas que podemos encontrar en las bibliotecas como lo son *scanf()*, *printf()* y *getch()*; estas son funciones predefinidas que nos ayudan a realizar tareas recurrentes de una manera eficiente y estandarizada. El otro grupo son aquellas funciones definidas por el desarrollador (es decir nosotros); estas son funciones diseñadas para tareas más específicas y personalizadas.

Los programas sencillos, como los que hemos desarrollado hasta el momento, no necesitan una estructuración elevada, pero conforme se vuelven más complejos, es crucial tener una mejor organización. Es aquí donde entran nuestras propias funciones, las cuales nos permitirán dividir nuestro código en bloques lógicos y fáciles de mantener.

ESTRUCTURA DE LA FUNCIÓN

Una función se compone de dos partes esenciales: el encabezado y el cuerpo. El encabezado es la parte donde definimos la función; este debe incluir el tipo de valor que retornará, el nombre de la función y los argumentos que recibirá. Estos argumentos son parámetros que permiten que la función opere con valores diferentes cada vez que se llama. El encabezado

establece la forma de la función, indicando cómo se debe invocar y qué tipo de datos espera recibir.

El cuerpo de la función es donde se especifica el comportamiento de la función. Al iniciar el cuerpo de la función, es recomendable declarar primero las variables que se utilizarán. Es posible que estas variables sean necesarias para almacenar valores temporales o intermedios que se utilizarán en operaciones dentro de la función. Luego, se escriben las sentencias de ejecución que realizarán las acciones requeridas por la función. Estas instrucciones pueden incluir cálculos, llamadas a otras funciones y cualquier otra operación necesaria para cumplir el propósito de la función.

Al final del cuerpo de la función, es común encontrar una declaración de retorno. Esta declaración se utiliza para devolver un valor de la función a la ubicación desde donde fue llamada. La presencia de una declaración de retorno depende del tipo de función que se defina. En algunas funciones, especialmente aquellas que realizan una tarea específica sin necesidad de devolver un valor, la declaración de retorno puede no ser necesaria. Sin embargo, en funciones que deben devolver un resultado, la declaración de retorno es crucial para proporcionar el valor calculado o el resultado de las operaciones realizadas dentro de la función.

DECLARACIÓN DE UNA FUNCIÓN

Al igual que las variables, las funciones también deben ser declaradas, dicha declaración debe indicar el nombre de la función, sus parámetros (cada uno precedido por el tipo de dato que podrá tener) y el tipo de dato de retorno. Su sintaxis es la siguiente:

tipo-de-dato nombreDeLaFuncion([lista de parametros])

En caso de no tener parametros podemos simplemente agregar la palabra *void* (vacío) o dejar los paréntesis vacíos, como normalmente hacemos con la función *main()*.

Existen dos tipos de declaración: la declaración explícita y la declaración implícita. En la declaración explícita, la función se debe declarar antes de la función *main()*. A esto es a lo que se conoce como prototipo de la función. El uso de prototipos le permite al compilador

comprender la estructura y características de las funciones, lo que facilita la verificación de su sintaxis, además, esto nos ayuda a evitar errores.

```
#include <stdio.h>
#include <conio.h>
tipo-de-dato funcionDeEjemplo(void);
main(void) {
   // Bloque de codigo
}
tipo-de-dato funcionDeEjemplo(void) {
   // Bloque de codigo
}
```

En la declaración implícita no hacemos uso del prototipo, esto no es una buena práctica, ya que puede generar errores en nuestro compilador. Además, en este caso C, por defecto, asume que el tipo de dato de retorno es *int*, por lo que no podremos ocupar otro tipo de dato de retorno. Un ejemplo de esto es la función *main()*, por esto es por lo que se le debe agregar *int* antes de "main" y se le debe agregar un dato de retorno de tipo entero. En caso de que la función no tenga un retorno, se le da el valor de cero.

```
#include <stdio.h>
#include <conio.h>
int main(void) {
   // Bloque de codigo
   return 0;
}
```

Es importante aclarar que todas las funciones tienen un carácter global, es decir que ninguna función puede ser definidas dentro de otra función.

A las funciones, también les podemos agregar un atributo llamado calificador, que puede ser *static* o *extern*; este nos dirá el alcance de la función. Una función con el calificador *static* será accesible solamente dentro del archivo de código fuente en el que fue definida. En cambio, si tiene el calificador *extern*, será accesible en todos los archivos de código fuente que componen nuestro programa. El calificador debe ser añadido en el prototipo de la función, antes del tipo de dato de retorno.

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

```
#include <conio.h>
static tipo-de-dato funcionDeEjemplo1(void);
extern tipo-de-dato funcionDeEjemplo2(void);
int main(void) {
   // Bloque de codigo
}
tipo-de-dato funcionDeEjemplo1(void) {
   // Bloque de codigo
}
tipo-de-dato funcionDeEjemplo2(void) {
   // Bloque de codigo
}
```

Por el momento, nosotros no utilizaremos este atributo, pues estamos trabajando con programas de un solo archivo de código fuente. En este caso, al no agregar ningún calificador, el compilador lo interpretará como una función *extern*.

TIPOS DE FUNCIONES

Existen principalmente dos tipos de funciones en C: las funciones sin retorno y las funciones con retorno.

Funciones sin retorno.

En funciones sin retorno usaremos el tipo de datos *void*; Esto significa que la función no devolverá ningún valor cuando se ejecute. Este tipo de funciones nos serán de utilidad cuando necesitemos realizar procesos que no requieran conservar ningún resultado, por ejemplo, realizar una impresión de pantalla, modificar variables globales o realizar operaciones de inicialización. Estas funciones son especialmente útiles para tareas repetitivas que no necesitan devolver información al programa principal.

```
void funcion([parametros])
```

Funciones con retorno.

Este tipo de funciones son imprescindibles en la programación en C, ya que nos permiten ejecutar una secuencia de instrucciones y preservar el resultado final para posteriormente utilizarlo en otro bloque de nuestro código. Al final de nuestra función se debe agregar la

sentencia de retorno usando la palabra reservada *return*, seguida del valor o expresión que queremos devolver. En C, las funciones pueden devolver cualquier tipo de datos válido en el lenguaje, incluidos tipos primitivos como *int*, *float*, *char* y *double*, así como tipos compuestos como matrices, estructuras y uniones. También pueden devolver apuntadores y, en determinados casos, incluso otras funciones.

```
int funcion([parametros])
```

La declaración de retorno finaliza la ejecución de la función y devuelve el control al punto donde se llamó a la función, junto con el valor especificado. Si la función no tiene una declaración de retorno y su tipo de retorno no es nulo, se producirá un error de compilación. En funciones de tipo *void*, la declaración *return* se puede utilizar para finalizar prematuramente la ejecución de la función, aunque no devuelve ningún valor.

Las funciones pueden llamarse a sí mismas, lo que se conoce como recursividad. En estos casos, la declaración de retorno se utiliza para devolver el valor de la función recursiva. Por ejemplo:

```
int factorial(int num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num - 1);
    }
}
```

Comparativa:

Si bien, las funciones sin retorno nos pueden ser de utilidad en algunos casos específicos, tiene el inconveniente de que si luego necesitamos estos datos no podremos acceder a ellos, mientras que las funciones con retorno pueden hacer que el código sea más modular y flexible, permitiendo que los resultados de las operaciones se utilicen en diferentes partes del programa. Esto es particularmente importante en programas grandes y complejos, donde la reutilización de los resultados puede mejorar la eficiencia y claridad del código.

Por esto es fundamental considerar cuándo utilizar una función nula y cuándo optar por una que devuelva un valor. La decisión dependerá de la naturaleza de la tarea que debe realizar

la función y de cómo se integrará esta tarea en el flujo general del programa. En algunos casos, una función nula puede ser más apropiada debido a su simplicidad y porque cumple su propósito sin necesidad de devolver información. En otros casos, la capacidad de devolver un valor puede ser esencial para la lógica y la funcionalidad del programa.

LLAMADO DE UNA FUNCIÓN

El llamado de una función en C es el proceso mediante el cual se invoca una función, previamente definida en el programa, para su ejecución. A continuación, veremos 3 formas simples de llamar a nuestra función:

La primera manera de llamar a nuestra función es simplemente escribir el nombre de nuestra función con sus parametros, en caso de no tener parametros dejaremos los paréntesis vacíos. Este tipo de llamado es especialmente útil cuando trabajamos con funciones del tipo *void*, sin embargo, podemos utilizarlo para llamar a funciones de cualquier tipo.

```
int main(void) {
   funcionDeEjemplo();
   return 0;
}
void funcionDeEjemplo(void) {
   // Bloque de codigo
}
```

La segunda manera de llamar a nuestra función es asignando el retorno a una variable. Este tipo de llamado solo se puede realizar para funciones con retorno.

```
int main(void) {
   float variableMain;
   variableMain = funcionDeEjemplo();
   return 0;
}
float funcionDeEjemplo(void) {
   float variableFuncion;
   // Bloque de código
   return variableFuncion;
}
```

La tercera manera de llamar a nuestra función es usándola directamente, como si fuera una variable. Al igual que la forma anterior, este tipo de llamado solo se puede realizar para funciones con retorno.

```
int main(void) {
   printf("El valor de la variable de retorno es: %f", funcionDeEjemplo());
   return 0;
}
float funcionDeEjemplo(void) {
   float variableFuncion;
   // Bloque de código
   return variableFuncion;
}
```

TIEMPO DE VIDA DE LOS DATOS

La vida útil de los datos en C se refiere al tiempo durante el cual una variable existe en la memoria y está disponible para su uso. Este concepto es crucial para comprender el comportamiento de las variables y la gestión de la memoria en C. A continuación, analizaremos en detalle la vida útil de las variables globales y locales, así como su uso dentro y fuera de las funciones.

Variables globales.

Las variables globales son aquellas que se declaran fuera de cualquier función, al inicio de un archivo de código fuente. Estas variables tienen un tiempo de vida que abarca desde el inicio de la ejecución del programa hasta su finalización. Es decir que las variables globales son accesibles desde cualquier función dentro del mismo archivo.

```
#include <stdio.h>
#include <conio.h>
float variableGlobal;
int main(void) {
   // Bloque de codigo
}
```

Variables globales estáticas y externas.

Las variables estáticas globales, son declaradas como una variable global, pero se les agrega el calificador *static*. Esto limita a la variable al archivo en el que fue declarada, como ya lo vimos con las funciones.

```
#include <stdio.h>
#include <conio.h>
static float variableGlobal;
int main(void) {
   // Bloque de codigo
}
```

Esta declaración puede ser innecesaria, pues el compilador le asigna, por defecto, este calificador, a no ser que se incluya el atributo *extern*, que nos permitirá acceder a estas variables desde otros archivos de código fuente de nuestro programa:

```
#include <stdio.h>
#include <conio.h>
extern float variableGlobal;
int main(void) {
   // Bloque de codigo
}
```

Variables locales.

Las variables locales son aquellas declaradas dentro de una función. Su vida útil está limitada al tiempo de ejecución de la función en el que están declarados, es decir que estas variables no son accesibles fuera de su función.

```
#include <stdio.h>
#include <conio.h>
int main(void) {
   float variableLocal;
   // Bloque de codigo
}
```

Otra forma en la que podemos utilizar las variables locales es en bloques de código (que los podemos identificar por el uso de llaves) y en argumentos.

```
#include <stdio.h>
#include <conio.h>
int main(void) {
   for(int i = 0; i < k; i++) {
      float variableLocal;
      // Bloque de Codigo
   }
}
```

Sin embargo, esto no es del todo recomendable, pues algunos compiladores no reconocen este tipo de declaración de variables y por lo tanto nos marcará un error.

Variables locales estáticas y automáticas.

Las variables locales las podemos dividir en dos categorías principales, de acuerdo con sus calificadores: variables estáticas y variables automáticas.

Las variables estáticas locales son declaradas dentro de una función, sin embargo, conservan su valor entre llamadas a la función. Por ejemplo:

```
#include <stdio.h>
#include <conio.h>
void funcionDeEjemplo(void);
int main(void) {
   funcionDeEjemplo();
   funcionDeEjemplo();
   funcionDeEjemplo();
   return 0;
}
void funcionDeEjemplo(void) {
   static int variable-estatica-local = 0;
   variable-estatica-local++;
   printf("Variable estatica en la funcion: %d", variable-estatica-local);
}
```

En este caso, nuestra *variable-estatica-local* conserva su valor de un llamado a otro, es decir que la primera vez que la función es llamada, nuestra variable será inicializada en 0 y posteriormente su valor será aumentado en una unidad, dando como resultado 1. Sin embargo, en el segundo llamado, la variable no se volverá a inicializar, sino que conservará el valor del llamado anterior, por lo que, al inicializar la función, la variable tendrá el valor

de 1 y aumentará su valor en una unidad, dándonos así 2 como resultado final. Y en el tercer llamado nos dará 3 como resultado final y así sucesivamente.

En cambio, una variable automática borrará los datos al finalizar la función, por lo que cada vez que llamemos a la función, la variable volverá a inicializarse en 0 y al aumentar recibirá el valor de 1. Para este tipo de variables no es necesario especificarla, pues el compilador, por omisión, toma todas las variables como automáticas.

ARGUMENTOS Y PARAMETROS

En programación, particularmente en el contexto de funciones, los términos "argumento" y "parámetro" son fundamentales para comprender cómo se pasan los datos entre funciones.

- Parámetro: Es una variable declarada en la definición de una función. Actúa como un marcador de posición para los valores que serán pasados a la función cuando se llame. Por ejemplo, en la función *int suma(int num1, int num2)*, *num1* y *num2* son parámetros.
- Argumento: Es el valor real que se pasa a la función cuando se la invoca. Es el dato concreto que se asigna a un parámetro. Siguiendo el ejemplo anterior, si llamamos a *suma(3, 5)*, *3* y *5* son argumentos.

Paso de argumentos por valor.

Pasar argumentos por valor es un método común en funciones de C. Al pasar un argumento por valor, el valor de la variable original se copia al parámetro correspondiente de la función. Esto significa que cualquier modificación realizada al parámetro dentro de la función no afectará la variable original. Por ejemplo:

```
#include <stdio.h>
#include <conio.h>
void incrementar(int parametro);
int main(void) {
   int argumento = 5;
   incrementar(argumento);
   printf("Despues de llamar a la funcion: %d", argumento);
   return 0;
}
```

```
void incrementar(int parametro) {
   parametro ++;
   printf("Dentro de la funcion: %d", parametro);
}
```

En este caso, las operaciones que se realizan dentro de la función *incrementar()* solo tienen efecto dentro de esa función. Es decir que únicamente la *parametro* será afectada, por lo que la impresión dentro de la función *incrementar()* será *6*, pero en la función *main()*, *argumento* conservará su valor, por lo que la impresión será *5*.

Paso de argumentos por referencia (utilizando apuntadores).

Pasar por referencia implica pasar la dirección de memoria de una variable (un apuntador) a la función en lugar del valor real. Esto permite que la función acceda y modifique directamente la variable original. Por ejemplo:

```
#include <stdio.h>
#include <conio.h>
void incrementar(int *parametro);
int main(void) {
   int argumento = 5;
   incrementar(&argumento);
   printf("Despues de llamar a la funcion: %d", argumento);
   return 0;
}
void incrementar(int *parametro) {
   (*parametro)++;
   printf("Dentro de la funcion: %d", *parametro);
}
```

En este caso, las operaciones que se realizan dentro de la función *incrementar()* no solo tienen efecto dentro de la función, sino que afectan directamente a la variable *argumento* en la función *main()*. Así que, tanto la impresión dentro de la función *incrementar()*, como en la función *main()*, será *6*.

Como podemos ver en el ejemplo, esto se hace añadiendo un asterisco (*) al argumento de la función, lo cual sirve para indicar al compilador que se está utilizando un apuntador, y un et (&) antes de la variable que deseamos modificar, esto le indicará al compilador que los

cambios están dirigidos a esa variable. Además, podemos notar que para hacer operaciones con el apuntador debemos ponerlo paréntesis ().

Argumentos constantes.

Los argumentos constantes son aquellos que no se pueden modificar dentro de una función. Pueden ser constantes literales, variables constantes o expresiones constantes. Esto se hace agregando el calificador *const* antes del tipo de dato del argumento.

void incrementar(const int x)

CONCLUSIÓN

En definitiva, las funciones son uno de los pilares fundamentales del lenguaje de programación C. Nos permiten estructurar el código de forma modular, mejorando la legibilidad, facilitando la reutilización y simplificando el mantenimiento del software. Las funciones nos ayudan a dividir un problema complejo en tareas más manejables y específicas, lo cual es esencial para desarrollar programas eficientes y escalables.

Uno de los aspectos más importantes a considerar cuando se trabaja con funciones es su correcta definición y declaración. Es crucial comprender la diferencia entre una declaración de función (prototipo) y su definición. La declaración le informa al compilador sobre el tipo de retorno y los parámetros que tomará una función, mientras que la definición contiene el código real que ejecuta la función. Otro aspecto fundamental es el uso adecuado del alcance de variables, ya que evita conflictos y errores en el código.

Algunos errores más comunes que debemos evitar son:

- **No coincidir prototipos y definiciones:** Asegúrate de que los tipos de retorno, el nombre y los parámetros en la declaración y la definición de la función coincidan exactamente.
- **Olvidar retornar un valor en funciones no void:** Si una función está definida para retornar un valor, siempre debe terminar con una instrucción return que devuelva un valor del tipo adecuado. Esto incluye a la función *main()*, que como ya vimos es una función con retorno del tipo *int*.

- **Pasar parámetros incorrectos:** Verificar que los argumentos pasados a la función sean del tipo correcto y en el orden esperado.
- **Variables locales y globales:** Confundir el uso de variables locales y globales puede generar comportamientos inesperados y difíciles de depurar.

Para evitar estos errores y mejorar la calidad del código, se pueden seguir las siguientes recomendaciones:

- **Identar correctamente** el código dentro de las funciones para mejorar la legibilidad y facilitar la detección de errores. Una buena indentación ayuda a entender la estructura y flujo del programa de manera más clara.
- **Documentar adecuadamente** cada función, incluyendo comentarios sobre su propósito, parámetros, y valor de retorno.
- **Usar nombres descriptivos** tanto para las funciones como para los parámetros, lo que facilita la comprensión del código.
- **Mantener las funciones cortas y específicas**, respetando el principio de responsabilidad única, lo que significa que cada función debe realizar una única tarea claramente definida.
- **Realizar pruebas unitarias** de las funciones para asegurar que se comporten como se espera en diversos escenarios.
- **Aplicar buenas prácticas de programación** como la verificación de valores de retorno y el manejo de errores dentro de las funciones.

BIBLIOGRAFÍA

Santos González, M., Patiño Cortés, I., Carrasco Vallinot, R. (2006). *Fundamentos de Programación.* Alfaomega Grupo Editor, S.A. de C.V.

Peñaloza Romero, E. (2004). *Fundamentos de Programación C/C++.* (Cuarta Edición, Universidad Nacional Autónoma de México - Escuela Nacional de Estudios Profesionales Aragón). Alfa Omega Grupo Editor, S.A. de C.V.

Caballos Sierra, F. J. (2015). *C/C++: Curso de programación.* (Cuarta Edición). RA-MA Editorial, S.A. de C.V.

Gookin, D. (1995). *C para Principiantes.* (Primera Edición). Editorial Limusa, S.A. de C.V.