



**Ingeniería en Desarrollo de Software**  
**Primer Semestre**

Programa de la asignatura:  
**Fundamentos de programación**

**Unidad 6. Funciones**

Clave:  
**TSU    Licenciatura**  
16141102 / 15141102

**Universidad Abierta y a Distancia de México**





### Índice

Unidad 6: Funciones .....	3
Presentación.....	3
Propósitos.....	4
Competencia específica.....	4
6.1. Diseño descendente .....	4
6.2. Definición, declaración e invocación de funciones en C.....	11
6.3. Alcance de las variables .....	17
6.4. Paso de parámetros.....	18
6.4.1. Por valor .....	18
6.4.2. Por referencia .....	20
Cierre de la unidad .....	21
Fuentes de consulta .....	21



### Unidad 6: Funciones

#### Presentación



Hasta esta etapa del curso, has aprendido a utilizar las estructuras de control para diseñar soluciones de problemas simples. También has conocido diferentes formas de representar los datos involucrados en un problema, desde simples hasta estructurados (como arreglos, cadenas y estructuras). Sin embargo, todas las soluciones propuestas constan únicamente de un módulo (función), llamado **principal** (en C, main); lo cual no es conveniente cuando se trata de problemas complejos que requieran de una serie de tareas para lograr su solución, pues ésta sería muy grande y, por lo tanto, difícil de corregir o modificar.

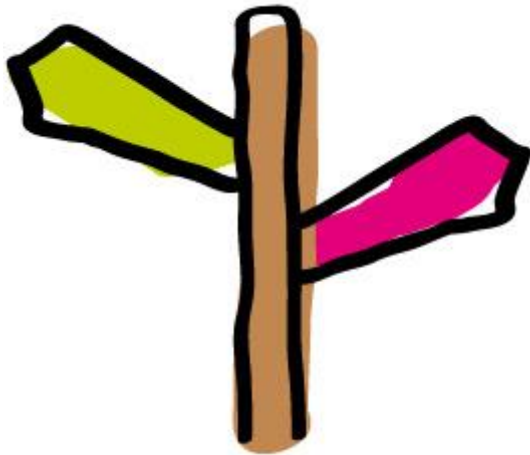
Lo recomendable es crear módulos independientes que realicen cada una de las tareas específicas y que en conjunto implementen la solución del problema. A esta metodología se le llama **diseño modular**, y está inspirado en la estrategia “**divide y vencerás**”.

En esta unidad se explicarán las funciones, que son el medio por el cual se implementan los módulos en lenguaje C. Se presentará la sintaxis para crear una función y la forma en la que se utilizan. Además, se resolverá un problema utilizando el diseño modular para ejemplificar el tema.





### Propósitos



En esta unidad:

- Identificarás los sub-problemas en que se puede dividir un problema.
- Diseñarás algoritmos modulares para solucionar un problema.
- Construirás funciones en lenguaje C que realicen tareas específicas.

### Competencia específica



Implementar funciones para resolver problemas a través del desarrollo de programas modulares escritos en lenguaje C.

#### 6.1. Diseño descendente

La descomposición de un programa en módulos más pequeños se conoce como el método de *divide y vencerás* (*divide and conquer*) o *diseño descendente* (*top-down*). Consiste en dividir un problema en unidades más pequeñas sucesivas hasta que sean directamente ejecutadas por el procesador, en otras palabras, la solución del problema se divide una y otra vez hasta que esté expresada en términos de estructuras de control, cada uno de los niveles o pasos sucesivos se conoce como *refinamiento* (*stepwise*).

La metodología del diseño descendente consiste en efectuar una relación entre las etapas de estructuración de manera que se relacionen entre sí a través de entradas y salidas de datos. Es decir, se descompone el problema en etapas de estructuración jerárquicas, de



forma que se pueda considerar cada estructura desde dos puntos de vista: ¿qué hace? y ¿cómo lo hace?

Con lo anterior podemos decir que: un *módulo* se caracteriza por realizar una tarea específica, posee sus propios datos de entrada – llamados *parámetros de entrada* – y su resultado – llamado *salida o valor de retorno* –. El diseño de cada módulo debe ser independiente de los otros; si es necesario que exista comunicación entre ellos, ésta únicamente podrá realizarse a través de los parámetros de entrada y del valor de retorno. En este sentido, puede ser visto, por otros módulos, como una *caja negra* que hacia el exterior sólo muestra *qué hace* y *qué necesita*, pero no *cómo lo hace*.

La creación de un módulo conlleva dos partes: la *definición del módulo* y la *llamada o invocación (ejecución)*. La primera consta de tres partes:

- Definir los parámetros de entrada;
- Definir el valor de retorno;
- Escribir todas las instrucciones que le permitirán llevar a cabo la tarea, es decir, un algoritmo.

La *llamada o invocación a un módulo* es el proceso de ejecutar el conjunto de instrucciones definidas en el módulo dado un conjunto de entradas específico. La forma general de invocar un módulo es escribiendo su nombre y entre paréntesis los valores de cada uno de los parámetros de entrada, respetando el orden que con el que se definió.

No existe un método para saber en cuánto módulos se debe dividir un problema, sin embargo es importante tener en cuenta los siguientes principios del diseño modular:

- Las partes altamente relacionadas deben pertenecer a un mismo módulo.
- Las partes no relacionadas deben residir en módulos diferentes.

Para ejemplificar esta forma de resolver problemas y cómo implementarla, se presenta la siguiente situación:

**Problema 6.1:** Realiza el análisis y diseño de un programa que lea las temperaturas promedio mensuales registradas en una ciudad a lo largo de un año y calcule el promedio anual. Además, debe convertir las temperaturas mensuales dadas en grados Celsius a grados Fahrenheit al igual que el promedio.

Empecemos por hacer un bosquejo de los pasos que se tienen que realizar para llegar al resultado deseado.

1. Leer las doce temperaturas promedio mensuales
2. Calcular el promedio anual de las temperaturas
3. Convertir las temperaturas promedio mensuales de Celsius a Fahrenheit





4. Convertir el promedio anual de temperaturas a Fahrenheit
5. Imprimir las temperaturas mensuales en grados Fahrenheit y el promedio anual de las temperaturas, en Celsius y Fahrenheit.

Para registrar las temperaturas mensuales proponemos utilizar un arreglo de tamaño 12. Y de acuerdo con lo anterior, proponemos tres módulos: El primero encargado de leer las temperaturas mensuales dadas en grados Celsius, este módulo necesita el nombre del arreglo donde va a almacenar los datos. Otro módulo encargado de calcular el promedio de las temperaturas, que recibe como parámetros de entrada el arreglo con las temperaturas mensuales y devuelve el promedio en grados Celsius. El último módulo sólo convierte grados Celsius a grados Fahrenheit. Esta información se resume en el diagrama modular que se muestra a continuación.

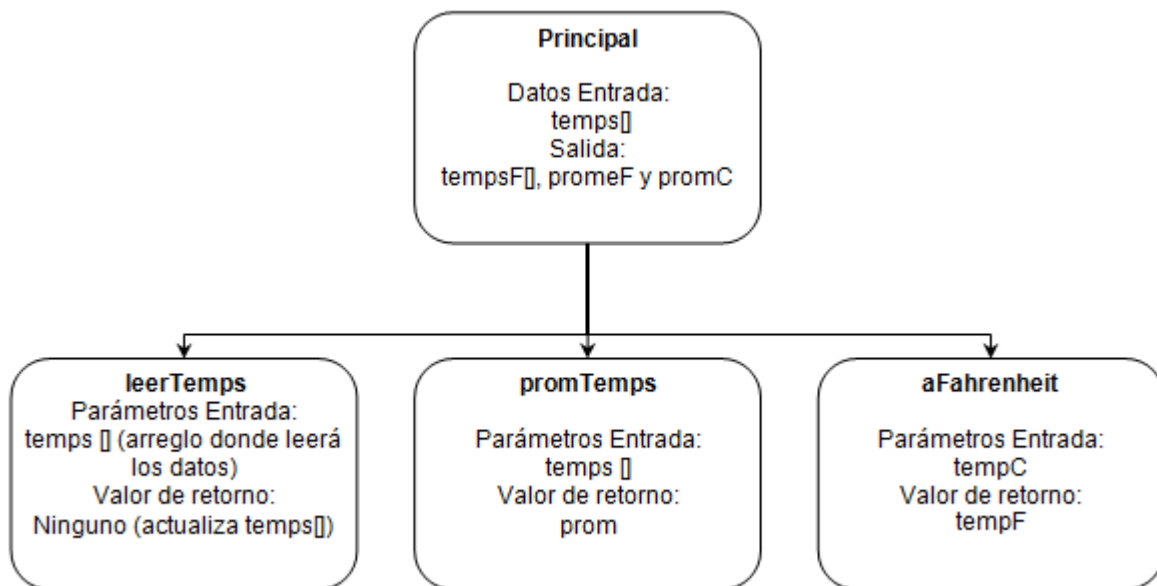


Figura 6.1: Diagrama modular del problema 6.1

Es mediante un diagrama modular como se representan de manera gráfica los módulos que integran la solución del problema. Y una vez que se han definido, el siguiente paso es diseñar el algoritmo de cada uno de ellos.

En primer lugar se muestra el pseudocódigo de los módulos secundarios.



### Módulo leerTemps(temp[ ])

#### Inicio

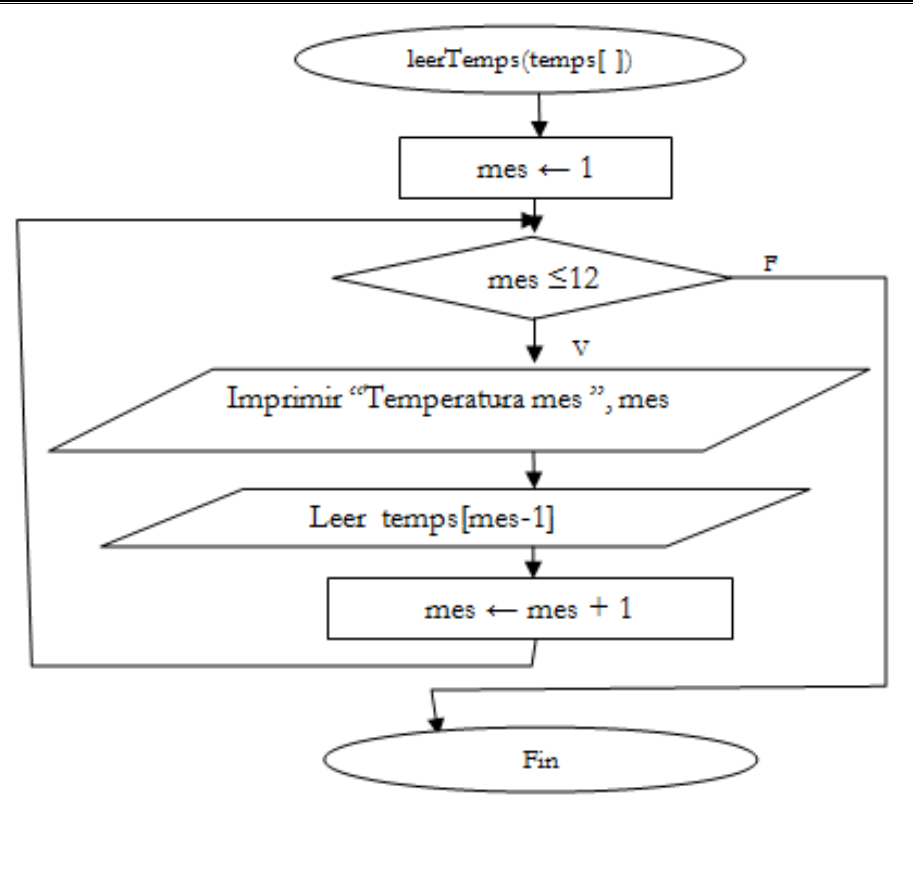
**Desde**  $\text{mes} \leftarrow 1$  **mientras**  $\text{mes} \leq 12$ ,  $\text{mes} \leftarrow \text{mes} + 1$

Imprimir "Proporciona la temperatura del mes", mes

Leer temps[mes-1]

#### Fin\_Desde

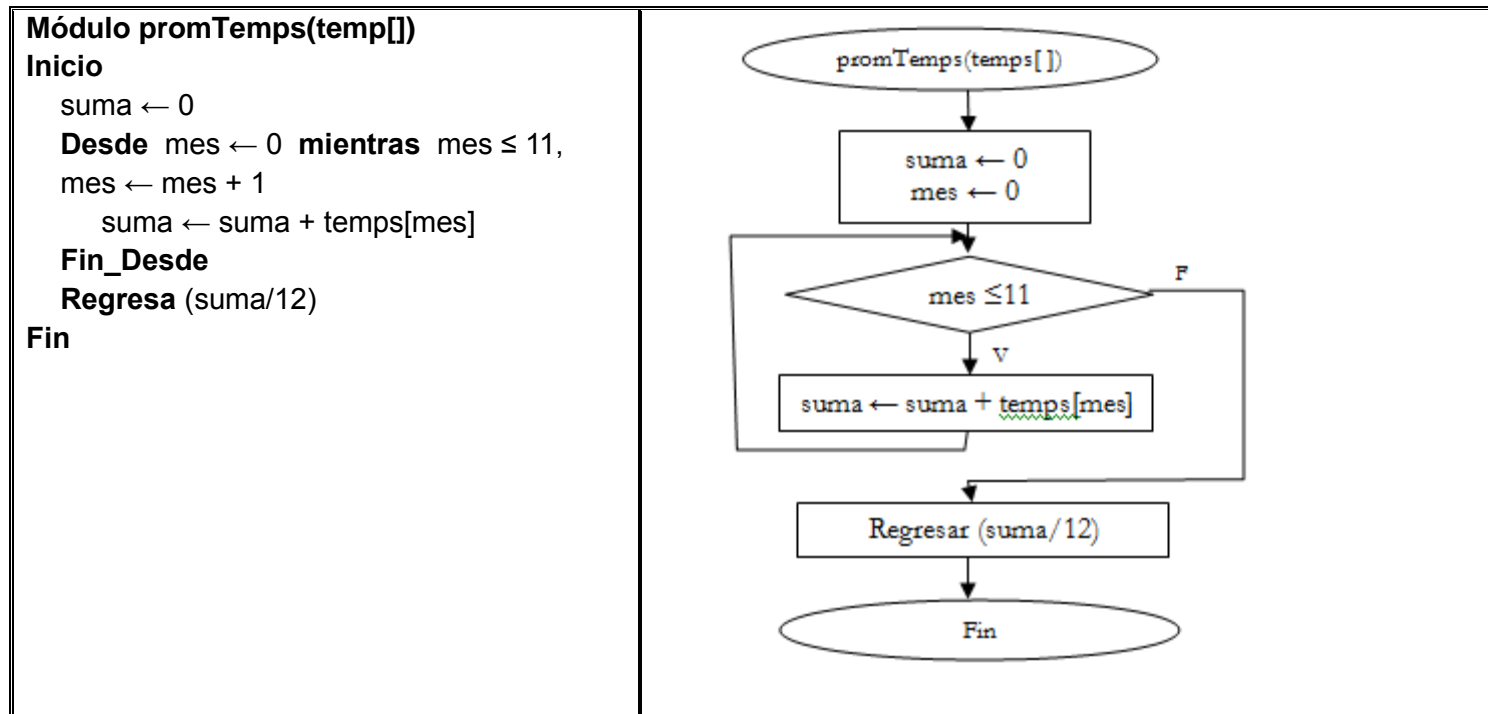
#### Fin



Algoritmo 6.1: Algoritmo del módulo leerTemps(temp[])



Observa que el ciclo del módulo *leeTemps* se inicia con *mes* igual a 1 y termina cuando *mes* es 13, se propone así porque se pide la temperatura de los mes 1, 2, 3,.. 12, así que la variable *mes* guardara el número de mes correspondiente a cada lectura, sin embargo para almacenar la temperatura en la posición del arreglo indicada se resta uno (*temps[mes-1]*), así se guarda desde la posición 0 y hasta la posición 11.



**Algoritmo 6.2:** Algoritmo del módulo promTemps(temp[])

En contraste con el ciclo del módulo *leeTemp*, en este módulo el contador del ciclo se inicia en 0 y termina en 11 (aunque el valor que tiene al finalizar el ciclo es 12) pues se utiliza para ir sumando cada uno de los elementos del arreglo, así que la variable *mes* indicará cada una de las posiciones del arreglo.





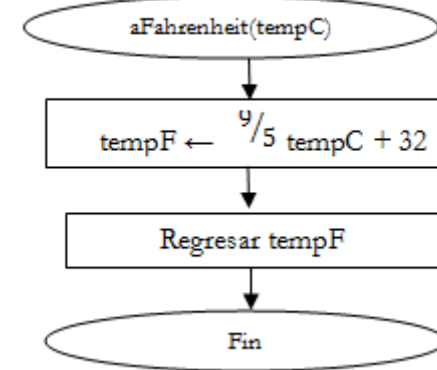
### Módulo aFahrenheit(tempC)

**Inicio**

$$\text{tempF} = \frac{9}{5} \text{tempC} + 32$$

**Regresa** tempF

**Fin**



**Algoritmo 6.2:** Algoritmos de módulos secundarios del problema 6.1 (pseudocódigo)

A partir de los módulos secundarios presentados se puede plantear el módulo principal.

### Módulo Principal

**Inicio**

*/\* Lectura de las temperaturas, invocando al módulo leerTemps \*/*

Imprimir "Ingresa los promedios de temperaturas mensuales"

leerTemps(temps[])

*/\* Cálculo del promedio utilizando el módulo promTemps \*/*

promC ← promTemps(temps[])

*/\* Conversión del promedio a grados Fahrenheit, invocando al módulo aFahrenheit \*/*

promF ← aFahrenheit(promC)

*/\* Conversión de las temperaturas mensuales a grados Fahrenheit \*/*



```
Desde mes  $\leftarrow$  0 mientras mes  $\leq$  11, mes  $\leftarrow$  mes + 1
    tempsF[mes]  $\leftarrow$  aFahrenheit(temps[mes])
Fin_Desde

/* Impresión de temperaturas mensuales en Fahrenheit */
Desde mes  $\leftarrow$  1 mientras mes  $\leq$  12, mes  $\leftarrow$  mes + 1
    Imprimir "Temperatura en Fahrenheit del mes", mes, " es: ", tempF[mes-1]
Fin_Desde

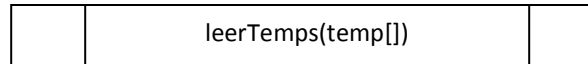
/* Impresión del promedio en grados Celsius y grados Fahrenheit */
Imprimir " El promedio en grados Celsius es: ", promC
Imprimir "El promedio en grados Fahrenheit es: ", promF
```

**Fin**

**Algoritmo 6.4:** Algoritmo del módulo principal del problema 6.1 (pseudocódigo)



Por esta ocasión se deja como ejercicio al lector que realice las representaciones en diagrama de flujo del módulo principal, considerando que el símbolo que se utiliza para llamar a módulos que no devuelven ningún valor, como es el caso de *leerTemps* se muestra a continuación:



Esto no es caso, del resto de los módulos secundarios que sí devuelven un valor, así que se utiliza el símbolo de proceso (rectángulo).

Con el ejemplo anterior resaltan indudablemente varias ventajas que proporciona un diseño modular:

1. *Es posible reutilizar código*, ésta indudablemente es una de las más importantes ya que no es necesario escribir el mismo código cada vez que deseamos realizar una tarea semejante. Por ejemplo, el módulo *aFahrenheit* se utiliza 13 veces (12 en el ciclo y una para convertir el promedio) y sólo bastó definirlo una vez.
2. *Fácil detección y corrección de errores*, dado que el problema fue dividido en pequeños módulos cada uno responsable de una tarea, si en algún momento existiera un error en la solución global, basta identificar cuál de las tareas es la que no se está resolviendo adecuadamente y corregir únicamente aquellos módulos involucrados.
3. *Fácil modificación o extensión*, si se requiere modificar una tarea del problema o agregar una nueva, no será necesario rediseñar todo el algoritmo, basta con hacer las modificaciones en los módulos pertinente.

*Un problema se vuelve más sencillo de solucionar si pensamos de manera modular.*

Una vez que se ha ilustrado como realizar una solución modular, lo siguiente es explicar cómo se codifica cada uno de los módulos, tema que se abordará en las siguientes secciones.

## 6.2. Definición, declaración e invocación de funciones en C

En el caso particular de C, un módulo se implementa como una función, recuerda que en la unidad 3 se explicó que un programa en C está integrado de varias funciones, donde una función se define como una porción de código (un conjunto de instrucciones agrupadas por separado) enfocado a realizar una tarea en específico, resaltando la función principal *main*.



Al igual que la función principal, cada una de las funciones existe de manera independiente, tiene sus propias variables, instrucciones y un nombre que las distingue de las otras.

Además de los parámetros de entrada (también llamados *argumentos*) que recibe y el tipo de dato que regresa.

La forma general para *definir una función* es:

```
<tipo de dato retorno><identificador de la función>( <parámetros de entrada>
{
    <instrucciones de la función>
    return<expresión>;
}
```

El <tipo de dato retorno> indica el tipo de dato que la función devuelve (puede ser cualquiera de los tipos básicos), para lo cual se utiliza la palabra reservada `return`. Cuando la función no va a devolver ningún dato se especifica mediante el tipo `void` y no debe incluir la palabra reservada `return`. El <identificador de la función> es el nombre que le vamos a dar a la función y mediante el cual vamos a hacer referencia a ella. Se deben seguir las mismas reglas que los identificadores de las variables y se recomienda que sea nemónicos. Enseguida del nombre y entre paréntesis va la lista de parámetros, que consiste de una lista de declaraciones de variables locales que van a contener los datos de entrada para la función, se debe especificar explícitamente el tipo y nombre para cada uno, separados por comas, aun cuando sean del mismo tipo: `tipo1 param1, tipo2 param2, ..., tipoN paramN`. Por último, las instrucciones del cuerpo de la función van entre llaves.

La primera línea de la definición de una función se conoce como *encabezado de la función* (en inglés *header*).

Para ilustrar esto, a continuación se muestra la codificación del módulo *aFahrenheit* definido en la sección anterior.

---

```
float aFahrenheit(float tempC)
{
    return ((9.0/5.0)*tempC+32);
}
```

---

**Programa 6.1:** Codificación del módulo *aFahrenheit*



La *llamada o invocación a una función* se realiza cuando se requiere que se ejecuten las instrucciones del cuerpo con valores de entrada determinados. Para invocar a una función se tiene que escribir el nombre seguido de los valores que deben coincidir con el orden, número y tipo de los parámetros de entrada dados en la definición de la función y deben estar encerrados entre paréntesis.

La sintaxis general para invocar una función ya sea predefinida o definida por el programador, es la siguiente:

<identificador de la función>( <lista de valores>);

Los parámetros de entrada de una función son valores, por lo que pueden estar determinados por: valores constantes (8, 'a', 5.2, "cadena constante") o una variable (lo que realmente se pasa a la función es el valor almacenado) o una expresión (el parámetro de entrada real es el resultado de la evaluación). Por ejemplo, la llamada a la función que definimos se remarcar en la siguiente instrucción:

```
promF = aFahrenheit(promC);
```

Al igual que las variables una función debe de ser declarada antes de utilizarse, es decir, antes de invocarla. La declaración de una función se realiza escribiendo el *prototipo de la función*. El prototipo de una función coincide con el encabezado de la misma terminando con punto y coma (;) El prototipo de una función sólo indica al compilador que existe y cómo es, más no lo que hace por lo tanto se debe definir después. Por ejemplo, el prototipo de la función antes definida sería:

```
float aFahrenheit(float tempC);
```

Cabe señalar que en el prototipo de las funciones se puede omitir los identificadores de los parámetros, más no los tipos.

Para ilustrar todo lo anterior, a continuación se muestra la codificación del algoritmo modular diseñado en la sección anterior.

---

```
/* Programa: promedioTemp.c
 * Descripción: Calcula el promedio de las temperaturas promedio
 * mensuales registrada a lo largo de un año*/

/* Biblioteca */
#include<stdio.h>
#include<stdlib.h>

/* Variables globales */
```





```
int meses = 12;

/* Prototipos de las funciones */
/* Función que lee las temperaturas promedio mensuales registradas en un año*/
void leerTemps( float temps[]);

/* Función que calcula el promedio de las temperaturas promedio mensuales
registradas en un año*/
float promTemps( float temps[]);

/* Función que convierte de grados Celsius a grados Fahrenheit */
float aFahrenheit(float tempC);

/* Función principal */
main()
{
    /* Declaración de variables locales a main */
    float temps[12], tempsF[12], promF, promC;
    int mes;

    /* Lectura de las temperaturas, invocando a leerTemps*/
    printf("Ingresa los promedios de temperaturas mensuales\n");
    leerTemps(temps);

    /* Cálculo del promedio utilizando la función promTemps */
    promC = promTemps(temps);

    /* Conversión del promedio a grados Fahrenheit, invocando al
módulo aFahrenheit */
    promF = aFahrenheit(promC);

    /* Conversión de las temperaturas promedio mensuales a grados
Fahrenheit, invocando al módulo aFahrenheit */
    for(mes = 0; mes<=11; mes++)
        tempsF[mes] = aFahrenheit(temps[mes]);

    /* Impresión de temperaturas promedio mensuales en grados
Fahrenheit*/
    for(mes = 1; mes<=12; mes++)
        printf("\n La temperatura en grados Fahrenheit del mes %d es %.2f: ", mes,
tempsF[mes-1]);
}
```



---

```
    /* Impresión del promedio */
    printf("\n\n El promedio anual en grados Celsius es: %.2f ", promC);
    printf("\n El promedio anual en grados Fahrenheit es: %.2f ", promF);

    system("pause");

} /* fin main */

/* Definición de funciones */

void leerTemps (float temps[])
{
    /* Definición de variables locales a leerTemps */
    int mes;

    for(mes = 1; mes<=12; mes++)
    {
        printf("\n Ingresa la temperatura promedio del mes %d: ", mes);
        scanf("%f", &temps[mes-1]);
    }
} /* fin leerTemps */

float promTemps (float temps[])
{
    /* Definición de variables locales a promTemps */
    int mes;
    float suma=0;

    for(mes = 0; mes<=11; mes++)
        suma = suma + temps[mes];

    return (suma/12);
} /* fin leerTemps */

float aFahrenheit(float tempC)
{
    return ((9.0/5.0)*tempC+32);
} /* fin aFahrenheit */
```

**Programa 6.1:** promedioTemp.c



```
C:\Users\Lilian\Documents\Respaldo2oct\IP\IP92\IP101\promTemperaturas.exe
Ingresa los promedios de temperaturas mensuales
Ingresa la temperatura promedio del mes 1: 25
Ingresa la temperatura promedio del mes 2: 24
Ingresa la temperatura promedio del mes 3: 25
Ingresa la temperatura promedio del mes 4: 24
Ingresa la temperatura promedio del mes 5: 23
Ingresa la temperatura promedio del mes 6: 22
Ingresa la temperatura promedio del mes 7: 21
Ingresa la temperatura promedio del mes 8: 24
Ingresa la temperatura promedio del mes 9: 25
Ingresa la temperatura promedio del mes 10: 20
Ingresa la temperatura promedio del mes 11: 18
Ingresa la temperatura promedio del mes 12: 19

La temperatura en grados Fahrenheit del mes 1 es 77.00:
La temperatura en grados Fahrenheit del mes 2 es 75.20:
La temperatura en grados Fahrenheit del mes 3 es 77.00:
La temperatura en grados Fahrenheit del mes 4 es 75.20:
La temperatura en grados Fahrenheit del mes 5 es 73.40:
La temperatura en grados Fahrenheit del mes 6 es 71.60:
La temperatura en grados Fahrenheit del mes 7 es 69.80:
La temperatura en grados Fahrenheit del mes 8 es 75.20:
La temperatura en grados Fahrenheit del mes 9 es 77.00:
La temperatura en grados Fahrenheit del mes 10 es 68.00:
La temperatura en grados Fahrenheit del mes 11 es 64.40:
La temperatura en grados Fahrenheit del mes 12 es 66.20:

El promedio anual en grados Celsius es: 22.50
El promedio anual en grados Fahrenheit es: 72.50
Presione una tecla para continuar . . . _
```

Figura 6.2: Ejecución del programa promedioTemp.c



### 6.3. Alcance de las variables

El *alcance* de las variables es la parte del programa dentro de la cual se pueden utilizar. Cuando son declaradas fuera del cuerpo de cualquier función se denominan *variables globales* y pueden ser utilizadas en cualquier punto del programa a partir del lugar donde fueron declaradas, en cambio cuando son declaradas dentro del cuerpo de alguna función se denominan *variables locales* a ésta, es decir sólo dentro de esa función pueden ser utilizadas.

Las variables locales que tienen el mismo nombre pero fueron declaradas en diferentes funciones, no tienen relación, son espacios de memoria totalmente independientes uno de otro. Podemos decir que, son como dos personas diferentes que tienen el mismo nombre. Por otro lado las variables que se ponen como argumentos en la declaración de una función se consideran locales a estas. Para ejemplificar lo anterior, se muestra el siguiente programa, en el cual se distinguen con diferentes colores el alcance de las variables.

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

int TAM = 5;

void inicializaA(int A[])
{
    int i;
    for (i=0; i<TAM; i++)
        A[i] = 0;
}

main()
{
    int i;
    int A[] = {1,1,1,1,1};
    printf("Arreglo antes de la llamada a inicializaA: A = [");
    for (i=0; i<TAM; i++)
    {
        if(i< TAM -1)
            printf("%d ", A[i]);
        else
            printf("%d ]\n\n\t", A[i]);
    }
    inicializaA(A);
    printf("Arreglo despues de la llamada a inicializaA: A = [");
    for (i=0; i<TAM; i++)
```

Variable global

Variable local a  
inicializaA

Variable local

Referencia a una  
variable global

Declaración de  
variables locales a main



```
{ if(i < TAM - 1)
    printf("%d ", A[i]);
    else
        printf("%d ]\n\n\t", A[i]);
}
system("pause");
}
```

Programa 6.2: porReferencia.c

Al utilizar variables globales todas las funciones pueden manipularlas, sus valores permanecen mientras el programa está en ejecución. Sin embargo su uso puede promover errores de tipo lógico, ya que al modificar el valor de una variable dentro de una función puede afectar el resultado de otra.

Por ejemplo, supongamos que la función `inicializaA()` modifica el valor de la variable `TAM` que almacena el número de elementos del arreglo `A`, este cambio repercutirá en los ciclos de la función `main`, los cuales imprimen el arreglo `A`. En este caso se producirá un error en la ejecución, pues si el valor es menor a cinco no se imprimirán todos los valores y si es mayor entonces habrá elementos indefinidos. Detectar y corregir este tipo de errores puede ser una tarea nada fácil, por lo que no se recomienda el uso de variables globales, lo cual no ocurre si son constantes.

Las variables locales por otra parte favorecen mucho la reutilización de código y la modularidad, ya que cada función declara y manipula sus propias variables sin depender de lo que ocurra en otras funciones, esto no significa que al utilizar solamente variables locales no sea posible compartir datos entre las diferentes funciones, esto se hace mediante sus datos de entrada y retorno, una posible desventaja es que el valor de las variables locales se pierde cada vez que la función termina.

### 6.4. Paso de parámetros

El paso de parámetros se refiere a la forma en la que se transfieren como parámetro una variable a una función, esencialmente, si se le otorgan o no permisos para modificar los valores originales. Cuando no se le otorgan permisos para que la modifique se dice que es *paso de parámetros por valor*, pues en este caso sólo se transfiere el valor de la variable, el cual se almacena en una variable local de la función que se está llamando. En cambio, cuando la función puede modificar el valor de la variable se dice que es un *paso de parámetro por referencia*, pues en este caso no se pasa sólo el valor sino la dirección de memoria que le corresponde a la variable.

En los siguientes subtemas se explica más a detalle los dos tipos de paso de parámetro.

#### 6.4.1. Por valor





Cuando se realiza una llamada a una función *por valor* y en ésta aparece una variable como uno de los argumentos, en realidad no se está pasando la variable sino una copia del valor que ésta contiene, lo cual implica que si dentro de la función se modifica el argumento esto no se ve reflejado en el programa desde el cual se hizo la llamada, pues son localidades de memoria diferentes (recuerda que en cada llamada a una función se crean nuevas variables y se destruyen una vez finaliza la ejecución).

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

void inicializa(int a)
{
    a = 0;
    printf("\nEl valor de la variable local \"a\" es %d\n\n",a);
}

main()
{
    int a=10;

    /* Llamada a la función inicializa */
    printf("\nEl valor de \"a\" antes de la llamada es %i\n\n", a);
    inicializa(a);
    printf("\nEl valor de \"a\" despues de la llamada es %i\n\n", a);

    system("pause");
}
```

Programa 6.3: porValor.c

La ejecución del programa es:

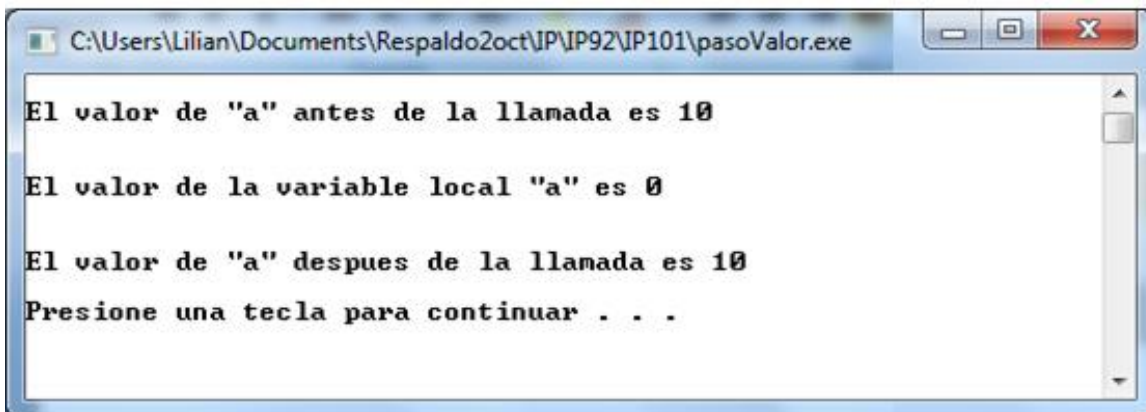


Figura 6.3: Ejecución del programa pasoValor.c

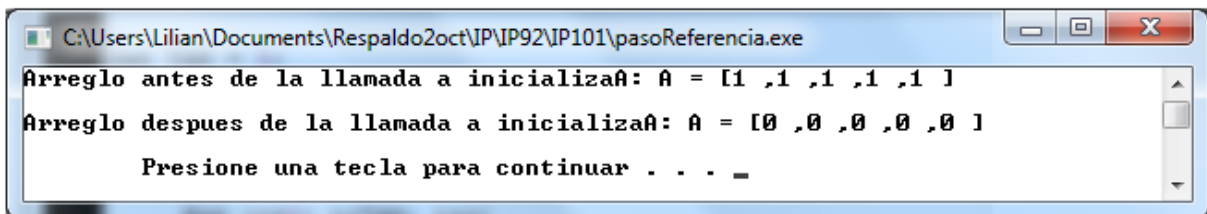
En la ejecución puedes ver que la variable local a **main** no se modifica, esto es porque se pasa una copia del valor que almacena cuando se realiza la llamada a la función



inicializa(a). Este valor se guarda en un espacio de memoria, también llamado *a*, que es una variable local a la función y que existe mientras ésta se ejecuta. Observa que el cambio si se realiza en la variable local de la función inicializa.

### 6.4.2. Por referencia

La *llamada a una función por referencia* sí modifica el valor de la variable, pues lo que realmente se está pasando es la dirección de memoria asignada a la variable para que la función pueda modificar el valor. En C los arreglos siempre se pasan por referencia, ya que el nombre del arreglo en realidad almacena la dirección de memoria donde se encuentra almacenado el primer elemento del arreglo. De esta manera cuando se realiza una llamada a una función y se escribe el identificador de un arreglo como parámetro, se está pasando la dirección. Para ejemplificar lo anterior se muestra la ejecución del programa `pasoReferencia.c` que se presentó en un subtema anterior.



```
C:\Users\Lilian\Documents\Respaldo2oct\IP\IP92\IP101\pasoReferencia.exe
Arreglo antes de la llamada a inicializaA: A = [1 ,1 ,1 ,1 ,1 ]
Arreglo despues de la llamada a inicializaA: A = [0 ,0 ,0 ,0 ,0 ]
Presione una tecla para continuar . . . _
```

Figura 6.4: Ejecución del programa `pasoReferencia.c`

En la ejecución del programa se observa que después de la llamada a la función cambia el estado del arreglo *A*.

Finalmente, cabe mencionar que para realizar la llamada por referencia de una variable de tipo básico en lenguaje C es necesario pasar la dirección de la variable para lo cual se utiliza el operador `&` seguido del nombre de la variable (`&nombre`), como se hace en la función `scanf`, este operador regresa la dirección de memoria que le corresponde a la variable indicada. Por otra parte, para almacenar la dirección de memoria de una variable se utiliza una variable de tipo *apuntador*. Una variable *apuntador* se encarga de almacenar una dirección de memoria, la declaración de una variable *apuntador* es similar a la de cualquier otra variable, con la diferencia que se debe escribir un asterisco entre el tipo de la variable y el identificador. El tema de los *apuntadores* es muy interesante, sin embargo, no es uno de los objetivos de este curso.

Para que apliques lo aprendido en esta unidad y a lo largo del curso se propone la siguiente evidencia de aprendizaje.



### Cierre de la Unidad

¡Felicidades!, aquí concluye el estudio de la Unidad 6. **Funciones**; también de la asignatura de **Fundamentos de programación** y con ella parte de un ciclo de aprendizaje, desarrollo profesional y personal único.

Esperamos que los contenidos y actividades a lo largo de la unidad hayan permitido implementar funciones para resolver problemas a través del desarrollo de programas modulares escritos en lenguaje C. Si requieres profundizar o resolver alguna duda sobre el tema no dudes en consultar a tu docente en línea.

### Fuentes de consulta

- Deitel H, M., & Deitel P, J. *Cómo programar en C/C++*. México: Prentice Hall.
- Joyanes, L., & Zohanero, I. (2005). *Programación en C. Metodología, algoritmos y estructuras de datos*. aspaño: Mc Graw Hill.
- Kernighan, B., & Ritchie, D. (1991). *El lenguaje de programación C*. México: Prentice-Hall Hispanoamericana.
- López, L. (2005). *Programación estructurada en lenguaje C*. México: Alfaomega.
- Pérez, H. (1992). *Física General* (Primera Edición ed.). México: Publicaciones Cultura.