Unidad IV – Funciones

Contenido

Funciones: Conceptos

Hasta ahora vimos como hacer algoritmos (programas) utilizando únicamente la función main. Los problemas vistos se resolvían con una secuencia de pasos que comenzaba en la función main y finalizaba en el return del mismo.

Hasta ahora vimos como hacer algoritmos (programas) utilizando la función main (punto de entrada a todo programa en C). Hemos visto que nuestros programas pueden utilizar funciones de la biblioteca standard de C, como por ejemplo la función printf() o scanf() de la biblioteca stdio.h (recuerde otras bibliotecas: stdlib.h, string.h, etc).

El lenguaje C permite también que el programador implemente sus propias funciones. Al usar funciones definidas por el propio programador, los programas pueden estructurarse en partes más pequeñas, más sencillas. Cada una de estas partes debe responder a un único objetivo (por ejemplo, una función para sumar dos valores, otra para multiplicar dos valores). Estas funciones podrán ser luego reutilizadas en diferentes partes del programa.

Resolver un problema pensando en dividirlo en varias funciones, nos obliga a dividir al problema principal (complejo) en un conjunto de problemas más pequeños (y más sencillos). Esta división del problema nos dará varias ventajas:

1. Cada división independiente (función) será más fácil de programar y probar.
2. El problema principal es más sencillo de entender.
3. Podemos reutilizar cada una de las divisiones independientes (funciones).
4. Estas divisiones independientes (funciones) podrían ser programadas por diferentes personas.

```
/*  Funciones:

    - Funciones sin retorno de valor:

            void nombreFuncion( parametros ){
                Instrucciones...
            }

    - Funciones con retorno de valor:

            tipo_dato nombreFuncion( parametros ){
                Instrucciones...
                return expresion
            }
*/

//1. Saludo por pantalla con funcion void
//2. Sumar 2 numeros
```

Funciones: valores de retorno

Escriba funciones para leer un número entero, y un número real; utilícelas en un programa:

```c
#include <stdio.h>

int leerNroEntero()
{
    int valor;

    printf("Ingrese un numero entero: ");
    scanf("%d", &valor);

    return valor;
}

float leerNroReal()
{
    float valor;

    printf("Ingrese un numero real: ");
    scanf("%f", &valor);

    return valor;
}

int main()
{
    int entero;
    float real;

    entero = leerNroEntero();
    real = leerNroReal();

    printf("Entero: %d\nReal: %f\n", entero, real);

    return 0;
}
```

Analicemos un poco el ejercicio anterior. Allí vemos dos funciones: main (como siempre) y leerNroEntero (una nueva función que creamos nosotros). Como vemos en su declaración:

```c
int leerNroEntero()
{
```

Decimos que esta función retorna un entero (int) y que su nombre es *leerNroEntero*. La función espera que se ingrese un número por teclado, lo asigna a la variable *valor*, y retorna la variable *valor* como respuesta a quien la llame. Así entonces cuando en la función main se llama a esta función de la siguiente manera:

```c
entero = leerNroEntero();
```

El valor retornado por *leerNroEntero* se asigna a la variable *edad*. Lo mismo sucede con la siguiente llamada.

Funciones: parámetros

Pero además de retornar valores, una función puede recibir parámetros. Los parámetros son valores que toma la función en el momento en que es llamada. Los parámetros son variables que toman un valor específico en el momento en que se ejecuta el programa (igual que las otras variables que hemos estado viendo. Así, por ejemplo, podemos hacer un programa que calcule la suma de dos números:

```
int suma(int a, int b)
{
    int suma;

    suma = a + b;

    return suma;
}
```

Analicemos un poco la definición de esta función:

```
int suma(int a, int b)
```

Esta función se llama suma, retorna un entero, y recibe dos parámetros enteros: el primer parámetro se llama *a* y el segundo *b*. Para utilizar la función suma debemos especificarle sus dos parámetros, que son los dos números que se deben sumar. Hay varias formas de especificar estos parámetros:

```
suma1 = suma(1, 2);
suma2 = suma(n1, 2);
suma3 = suma(n2, b);
```

El primer ejemplo asigna los valores 1 y 2 a las variables *a* y *b* de la función *suma* respectivamente. La función al finalizar retorna 3 y se lo asigna a la variable *suma1*.

El segundo ejemplo asigna los valores n1 y 2 a las variables *a* y *b* de la función *suma* respectivamente. Así por ejemplo, si n1 tuviera el valor 3, la variable a tomaría dicho valor. La función al finalizar retorna 5 y se lo asigna a la variable *suma2*.

El tercer ejemplo asigna los valores n1 y n2 a las variables *a* y *b* de la función *suma* respectivamente. Así por ejemplo, si n1 tuviera el valor 3, y n2 el valor 4, las variables a y b tomarían los valores 3 y 4 respectivamente. La función al finalizar retorna 7 y se lo asigna a la variable *suma3*.

El ejemplo completo quedaría así:

```
#include <stdio.h>

int suma(int a, int b)
{
    int suma;

    suma = a + b;

    return suma;
}
```

```c
int main()
{
    int n1 = 3;
    int n2 = 4;
    int suma1, suma2, suma3;

    suma1 = suma(1, 2);
    suma2 = suma(n1, 2);
    suma3 = suma(n1, n2);

    return 0;
}
```

Funciones: uso de void (procedimientos)

Algunas veces necesitamos crear funciones que no retornen ningún valor. Los ejemplos más sencillos de éstos casos son referentes a impresión de datos en pantalla. Supongamos que queremos hacer una función que imprima un mensaje de bienvenida al programa:

```c
void bienvenida()
{
    char aux[100];

    printf("Bienvenido al curso de UNRN\n");
    printf("Presione ENTER para comenzar\n");

    fgets(aux, 100, stdin);

    return;
}
```

El tipo de retorno *void* significa que la función no retorna ningún valor. En otros lenguajes de programación este tipo de funciones que no retornan ningún valor se llaman **procedimientos**. La forma de usar esa función en nuestro programa (función main) es la siguiente:

```c
#include <stdio.h>

void bienvenida()
{
    char aux[100];

    printf("Bienvenido al curso de UNRN\n");
    printf("< Presione ENTER para comenzar >\n");

    fgets(aux, 100, stdin);
}

int main()
{
    bienvenida();

    return 0;
}
```

Hacer los ejercicios del práctico secciones:

- I- Funciones útiles generales.

Funciones: parámetros por referencia y por valor

Algún alumno se preguntará en este momento, ¿que sucede si modifico el valor de la variable dentro de la función? ¿Se modifica también la variable asociada en la llamada a la función? Por ejemnplo:

```c
#include <stdio.h>

void incrementa(int a)
{
    a = a + 1;
}

int main()
{
    int x = 0;

    incrementa(x);
    incrementa(x);
    incrementa(x);

    printf("El valor incrementado 3 veces es: %d\n", x);

    return 0;
}
```

Si ejecutamos este ejemplo veremos que la variable x permanece con su valor asignado originalmente (valor 0) por lo que concluimos que al modificar el valor de un parámetro de una función, su variable equivalente en la llamada no se modifica, y permanece como estaba.

Esto es porque en C el pasaje de parámetros es **por valor**. El pasaje de parámetros **por valor** no cambia el valor de las variables originales al modificar el valor de los parámetros.

En otros lenguajes (por ejemplo en C++) existe el pasaje de parámetros **por referencia**. Cuando los parámetros se pasan por referencia, si cambiamos el valor dentro de la función, también se cambian en las variables de la llamada.

El lenguaje C tiene algo que a primera vista pareciera ser pasaje de parámetros por referencia, pero que en realidad es implementada de otra manera: a través de punteros. Si modificamos el ejercicio anterior para utilizar punteros, entonces si se logrará el resultado deseado.

Notas:

- Para utilizar el puntero en la función utilizar el operador *.
- Para pasar por parámetro el puntero a una variable utilizar el operador &.

```c
#include <stdio.h>

void incrementa(int *a)
{
    *a = *a + 1;
}
```

```c
int main()
{
    int x = 0;

    incrementa(&x);
    incrementa(&x);
    incrementa(&x);

    printf("El valor incrementado 3 veces es: %d\n", x);

    return 0;
}
```

Funciones: vectores, cadenas y matrices por parámetro

Además de tipos de datos primitivos (char, int, float double, etc) las funciones pueden recibir por prámetros vectores, cadenas y matrices. Para esto lo que se debe hacer indicar que se trata de un vector, cadena, o matriz incluyendo los corchetes como se muestra en los siguientes ejemplos.

Vector: haga una función que sume los elementos del vector:

```
int sumaVector(int vec[], int cantElementos)
{
    int i, suma;

    suma = 0;
    for(i = 0; i < cantElementos; i++)
    {
        suma += vec[i];
    }

    return suma;
}
```

Como vemos en la declaración de la función *sumaVector*, el primer parámetro es un vector de enteros. Esto queda determinado por los corchetes [] (se trata de un vector) y por su tipo int (se trata de enteros). Nótese que además se incluye otro parámetro que es la cantidad de elementos que tiene el vector. SIEMPRE que pasamos un vector por parámetro debemos incluír su cantidad de elementos.

Nota: en el caso de los vectores es posible utilizar un define en el programa que especifique esta cantidad, y luego especificar esto en la declaración del vector. Entonces en la función utilizaríamos esta constante N. El siguiente ejemplo ilustra esto:

```
int sumaVector(int vec[N])
{
    int i, suma;

    suma = 0;
    for(i = 0; i < N; i++)
    {
        suma += vec[i];
    }

    return suma;
}
```

Las diferencias de estos dos ejemplos anteriores son:

- en el primero se indica la cantidad de elementos del vector a través de un parámetro, en el segundo se hace con una constante de programa
- en el primero se utiliza la variable que indica la cantidad de elementos para recorrer el vector, en el segundo se utiliza la constante de programa

Vemos que la primera tiene ventajas con respecto a la segunda: podemos utilizar la función para vectores de cualquier tamaño. Mientras que en el segundo caso sólo podremos usar esa función para vectores del mismo tamaño N dentro del mismo programa. Es por esto que preferimos siempre la primer opción.

El ejemplo completo, que muestra también como utilizar la función es el siguiente:

```c
#include <stdio.h>

#define N 5

int sumaVector(int vec[], int cantElementos)
{
    int i, suma;

    suma = 0;
    for(i = 0; i < cantElementos; i++)
    {
        suma += vec[i];
    }

    return suma;
}

int main()
{
    int vector[N] = { 1, 2, 3, 4, 5 };
    int suma;

    suma = sumaVector(vector, N);
    printf("El vector suma: %d\n", suma);

    return 0;
}
```

Cadena de caracteres: como las cadenas de caracteres son vectores, aplica exactamente lo mismo que lo visto para vectores.

Hacer los ejercicios del práctico secciones:

- II- Funciones útiles de vectores
- III- Uso de funciones útiles de vectores

Matrices: haga una función que sume un valor escalar a una matriz, retornando el resultado en una segunda matriz:

```c
#define N 3
#define M 3

void sumaEscalar(int mat[N][M], int escalar, int resultado[N][M])
{
    int fila, col;

    for(fila = 0; fila < N; fila++)
    {
        for(col = 0; col < M; col++)
        {
            resultado[fila][col] = mat[fila][col] + escalar;
        }
    }
}
```

En la declaración de la función observamos varias cosas:

```
void sumaEscalar(int mat[N][M], int escalar, int resultado[N][M])
```

La función es de tipo *procedimiento* ya que no retorna ningún valor (es de tipo void). Esta función se llama *sumaEscalar*, y recibe tres parámetros. El primero y el tercero son matrices enteras de N filas por M columnas. El segundo parámetros es un número entero.

Observe la declaración del parámetro mat: si un parámetro es una matriz SIEMPRE se debe indicar el tamaño de filas y columnas (entre corchetes). Nota: en realidad el tamaño de filas puede o no indicarse (quedando esta declaración optativa así: int mat[][M]), pero a los efectos de este curso y para simplificar al alumno, siempre declararemos ambos atributos simultaneamente.

El ejemplo completo, que muestra también como utilizar la función es el siguiente:

```c
#include <stdio.h>

#define N 3
#define M 3

void sumaEscalar(int mat[N][M], int escalar, int resultado[N][M])
{
    int fila, col;

    for(fila = 0; fila < N; fila++)
    {
        for(col = 0; col < M; col++)
        {
            resultado[fila][col] = mat[fila][col] + escalar;
        }
    }
}

void mostrarMatriz(int mat[N][M])
{
    int fila, col;

    for(fila = 0; fila < N; fila++)
    {
        for(col = 0; col < M; col++)
        {
            printf(" %d ", mat[fila][col]);
        }
        printf("\n");
    }
    printf("\n");
}


int main()
{
    int a = 5;
    int m1[N][M] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 },
    };
```

```
    int m2[N][M];

    sumaEscalar(m1, a, m2);
    mostrarMatriz(m2);

    return 0;
}
```

En el ejemplo se ve también como implementar una función para imprimir una matriz.

En este ejemplo se ve también que, en el caso de matrices, al modificar el valor del parámetro dentro de la función, SI se modifica la matriz original (es el efecto explicado en pasaje de parámetros por referencia). Pero recuerde que C no tiene pasaje de parámetros por referencia, sino que este efecto se debe a que las matrices se pasan por parámetro como punteros (como ya se mencionó, esto no se verá en este capítulo).

Esta última observación (al modificar el valor del parámetro también se modifica la variable original) aplica tanto para matrices como para vectors (y por lo tanto cadena de caracteres).

Funciones: pasaje de parámetros referencia / valor – síntesis

El pasaje de parámetros de datos nativos (char, int, float, double, etc) es siempre por valor: la variable original mantiene su valor aunque el parámetro se modifique en la función.

El pasaje de parámetros de vectores, cadenas y matrices es siempre por puntero: al modificar el valor de una posición en el vector, cadena o matriz, se modifica la original.

Funciones: ejemplos

Queda para el alumno hacer algunos ejemplos de funciones indicados a continuación y que se encuentran en el práctico:

1. hallar el valor máximo de un vector
2. hallar el valor máximo de una matriz
3. hallar el valor mínimo de un vector
4. hallar el valor mínimo de una matriz
5. contar la cantidad de ocurrencias de un número dado en un vector
6. contar la cantidad de ocurrencias de un número dado en una matriz
7. sumar dos vectores en un tercer vector resultado
8. sumar dos matrices en una tercer matriz resultado
9. multiplicar dos matrices en una matriz resultado

Hacer los ejercicios del práctico secciones:

- IV- Funciones útiles de matrices
- V- Uso de funciones útiles de matrices

Alcance de una variable

El alcance de una variable determina el ámbito en el que está puede ser llamada. Así tenemos variables locales y globales.

Hasta ahora todas las variables que vimos son **locales**: todas están declaradas dentro de una función (por ejemplo, dentro de la función main, o de la función sumarVector, etc). Decimos que el ámbito de estas funciones es el ámbito de la función: estas variables pueden ser utilizadas únicamente dentro de la función en donde se declaran. Diferentes funciones que declaren variables del mismo nombre hacen referencia a variables diferentes. Estas funciones *viven* únicamente dentro de la función en donde se declaran.

Pero también podemos tener variables que puedan ser usadas dentro de todo un programa, o dentro de una serie de funciones (a partir de que estas variables son declaradas). Estas variables se llaman variables globales. La forma de usar variables globales se ilustra en el siguiente ejemplo:

```
#include <stdio.h>

int var;

void incrementar()
{
    var = var + 1;
}

int main()
{
    var = 3;
    printf("var = %d\n", var);

    incrementar();
    printf("var = %d\n", var);

    incrementar();
    printf("var = %d\n", var);

    return 0;
}
```

En el ejemplo se observa que la variable *var* se declara fuera de todas las funciones, no pertenece a ninguna, sino que es global. Esta variable se usa entonces en la función *incrementar* y en la función *main*.

Preferimos las variables locales frente a las variables globales. Las variables globales se prestan a confusión, y hace que el código de algunas funciones dependan del código de otras funciones. Esto suele conducir a defectos en los programas. En el curso vamos a evitar el uso de variables globales.

Hacer los ejercicios del práctico secciones:

- VI- Alcance de variables

Recursión

¿Pero qué sucedería si una función se llamara a si misma? La respuesta es recursión. Esta es una técnica de programación utilizada para resolver problemas del tipo factorial, series, etc.

Pero para que una función se pueda llamar así misma es necesario que cumpla dos características:

1. debe existir un caso, llamado caso base, por el que el procedimiento no se llama así mismo y finaliza (evitando llamarse así mismo infinitas veces);
2. cada vez que el procedimiento se llame a si mismo (directa o indirectamente), debe estar más cerca del criterio base.

Veamos en un único programa varios ejemplos de funciones recursivas. En el siguiente ejemplo se muestra como implementar una función que muestre una cuenta regresiva, tanto en forma recursiva como en forma iterativa. También se ilustra como hallar un número factorial de forma recursiva y una sumatoria de manera recursiva. Y por último se ejemplifica el uso de la recursión para hallar series, mostrando la serie de Fibonacci:

```c
#include <stdio.h>

/* Funcion que calcula e imprime una cuenta regresiva de forma recursiva */
void cuentaRegresivaRecursiva(int count)
{
    printf("%d\t", count);

    /* Siempre tienen que tener una condicion de fin,
       que tiene relacion a la variable recibida. */
    if (count != 0)
    {
        /* Vuelve a llamarse a si misma, por eso se dice que es recursiva */
        cuentaRegresivaRecursiva(count - 1);
    }
}

/* Funcion que calcula e imprime una cuenta regresiva de forma iterativa */
void cuentaRegresivaIterativa(int count)
{
    int i;

    /* Iterador for, por eso se dice que es iterativa. */
    for(i = count; i > 0; i--)
    {
        printf("%d\t", i);
    }
}

int factorial(int n)
{
    if (n == 1) /* caso base */
    {
        return(1);
    }

    return(n * factorial(n - 1)); /* recursion */
}

int sumatoria(int n)
{
    if( n == 1 )
    {
```

```
        return 1;
    }

    return n + sumatoria(n - 1);
}

int fibonacci(int n)
{
    if (n == 0) /* primer caso base */
    {
       return(0);
    }

    if ( n==1 ) /* segundo caso base */
    {
       return(1);
    }

    return(fibonacci(n-1) + fibonacci(n-2));
}

int main()
{
    int sum, fac, fib, nro;

    printf("Ingrese Numero: \n");
    scanf("%d", &nro);

    cuentaRegresivaRecursiva(nro);

    printf("\n\n");

    sum = sumatoria(nro);
    fac = factorial(nro);
    fib = fibonacci(nro);

    printf("Sumatoria(%d): %d\n", nro, sum);
    printf("Factorial(%d): %d\n", nro, fac);
    printf("Fibonacci(%d): %d \n", nro, fib);

    return(0);
}
```

Cambie el ejercicio de manera que se obtenga un único resultado dada una operación elegida por el operador. De esta manera, el operador debe ingresar dos números: primero la operación, y luego el número base para la operación:

- Si el operador ingresa 0 – salir del programa
- Si el operador ingresa 1 – Sumatoria
- Si el operador ingresa 2 – Factorial
- Si el operador ingresa 3 – Fibonacci

Uso de parámetros en la función main

La función main recibe dos parámetros del sistema operativo al ser ejecutado. El primero es un entero que indica la cantidad de parámetros recibidos. El segundo es un vector de punteros a cadenas de caracteres con cada uno de los parámetros recibidos. El siguiente ejemplo ilustra esto:

```c
#include <stdio.h>

int main(int n, char *argv[]) {
    printf("Cantidad de parametros: %d\n", n);

    for(int i = 0; i < n; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

Al ejecutar este programa con distinta combinación de parámetros se obtendrán distintos resultados. Ejemplos:

- main.exe -x -param2

  Cantidad de parametros: 3
  argv[0] = main.exe
  argv[1] = -x
  argv[2] = -param2

- main.exe -param1 -param2 param3 param4 a b c

  Cantidad de parametros: 8
  argv[0] = main.exe
  argv[1] = -param1
  argv[2] = -param2
  argv[3] = param3
  argv[4] = param4
  argv[5] = a
  argv[6] = b
  argv[7] = c