Unidad 1

Programación modular

Objetivos de aprendizaje: Al finalizar esta unidad el estudiante es capaz de

- Comprender el concepto de función.
- Reconocer cuándo se está llamando a una función.
- Declarar e implementar una función.
- Llamar o invocar a una función.
- Pasar uno o más valores a una función.
- Llamar a una función desde la definición de otra función.
- Implementar funciones recusivamente.

1.1. Una función vista como un sistema

El objeto de estudio de esta unidad es la función, por lo que comenzaremos definiéndola.

Definición 1.1. Una *función* f es una 4-upla:

$$f = (tip_f, nom_f, dir_f, def_f), \tag{1.1}$$

donde

$$\begin{aligned} tip_f &: \text{tipo de } f, \\ nom_f &: \text{nombre de } f, \\ dir_f &: \text{dirección de memoria de } f, \\ def_f &: \text{definición de } f. \end{aligned}$$

El tipo de una función consiste en el tipo de retorno y su lista de parámetros; el nombre de una función es un identificador y la definición de una función está compuesta por el *encabezado* y el *cuerpo* de la misma. Una característica de las funciones es que al ser *llamadas* o *evocadas* las instrucciones que yacen en su cuerpo son ejecutadas según los argumentos pasados. □

Haciendo un abuso de notación por un lado, pero por otro simplificando nuestra forma de expresarnos también nos referiremos a una función f por su nombre, i.e., siguiendo la notación de la definición de arriba: nom_f. De hecho, todo programa en C++ tiene al menos a la función `main`.

Existen dos tipos de funciones: *definidas por el usuario* e *incorporadas*. Estas útimas ya fueron creadas y son parte de los paquetes de compilación, e.g., `sqrt`, `abs`, `cos`, etc. Y las del primer tipo son implementadas por el usuario, i.e., ¡somos nosotros que las creamos! En esta unidad nos enfocaremos en este tipo de funciones.

Veamos a continuación lo que se entiende como *declaración* y *definición* de una función y otros conceptos relacionados a traveś de unos ejemplos.

Ejemplo 1.2. En el programa de la figura 1.1 el usuario implementa la función f dada por expresión (1.1), donde

$$\begin{aligned} tip_f &= \texttt{int (int x, int y)}, \\ nom_f &= \texttt{sumar}, \\ dir_f &= \texttt{0x400936}, \\ def_f &= \texttt{int sumar(int x, int y)\{return x+y;\}}. \end{aligned}$$

```
1 #include <iostream>
2
3 using namespace std;
4
5 // prototipo de la funcion sumar
6 int sumar(int x, int y);
7
8 int main () {
9     cout << "sumar(2,3) = " << sumar(2,3) << endl;
10 }
11
12 // definicion de la funcion sumar
13 int sumar(int x, int y) {
14     return x + y;
15 }
```

Figura 1.1: función `sumar`

Siguiendo nuestro abuso de notación mencionado arriba, nos referiremos a nuestra función f como `sumar`. La memoria de la computadora está compuesta por bits –unidad mínima de almacenamiento de la información– y cada bit tiene asociado una única dirección de memoria: numeral en base 16. Por ello toda dirección de memoria comienza con la secuencia `0x` para indicar que el numeral que sigue –en nuestro caso `400936`– está en la base 16. En la computadora que fue ejecutado este programa el sistema asignó a `sumar` la dirección de arriba; sin embargo, en otra computadora esta dirección puede ser otra. Sobre direcciones de memoria hablaremos con más detalle cuando estudiemos el tema de punteros.

En primer lugar, antes de definir una fución, es una buena práctica introducirla, darla a conocer; por ello, en la línea 6 de dicho programa tenemos la *declaración* de `sumar`. También es lícito declarar `sumar` como: `int sumar(int, int)`; ya que, lo esencial es que el programa sepa la lista de argumentos que la función recibirá (cantidad y tipos de argumentos) y el tipo que retornará. En nuestro caso, declaramos que `sumar` recibirá dos enteros y retornará un entero.

En segundo lugar, al correr el programa, se comienza ejecutando la línea 9 y se termina al finalizar dicha ejecución. En esta línea *llamamos* a `sumar` y le pasamos la lista de argumentos `(2,3)` y esta nos retorna –¡nos responde!– el valor entero de `5`. Es en este intervalo de tiempo –desde el llamado hasta la

respuesta– que el flujo del programa se traslada a las líneas 13-15. Es decir, en este intervalo a las variables locales `x` e `y` de `sumar` se le asignan los valores enteros de `2` y `3`, respectivamente; luego se evalúa la expresión `(x + y)` con estos valores y es este valor entero que se retorna, es esto que responde `sumar`. Finalmente, lo que observamos al ejecutar este programa es `sumar(2,3) = 5`.

Finalmente, el encabezado de `sumar` es `int sumar(int x, int y)`; nótese que difiere de la declaración de la línea 6 en un punto y como. Luego, el cuerpo de `sumar` es {`return x + y;`}; nótese que el abre llaves delimita el inicio del cuerpo y el cierra llaves el fin del mismo. □

1.2. Creación de una función

En lo sucesivo nos centraremos en la declaración y definición (cabecera y cuerpo) de una función y al flujo del programa durante la ejecución del mismo. A estos procedimientos de declaración y definición nos referiremos como la *creación* o *implementación* de una función. Una declaración de una función es llamada un *prototipo*.

Ejemplo 1.3. En el programa de la figura 1.2 el usuario implementa la función `mcd`. En primer lugar, en la línea 5 declaramos que `mcd` recibe dos enteros y retorna un entero. Luego, al correr el programa, se comienza ejecutando la línea 8 y se termina al ejecutar la línea 10.

```
1 #include <iostream>
2 using namespace std;
3
4 // prototipo de la funcion mcd
5 int mcd(int x, int y);
6
7 int main() {
8     cout << "mcd(2,3) = " << mcd(2,3) << endl;
9     cout << "mcd(2,4) = " << mcd(2,4) << endl;
10    cout << "mcd(6,9) = " << mcd(6,9) << endl;
11 }
12
13 // definicion de la funcion mcd
14 int mcd(int x, int y) {
15     int divisor;
16     for(int n=1; n<=x && n<=y; ++n) {
17         if(x%n == 0 && y%n == 0) divisor = n;
18     }
19     return divisor;
20 }
```

Figura 1.2: función `mcd`

Primero, en la línea 8 se llama a `mcd` y le pasamos la lista de argumentos `(2,3)` y esta nos retorna –¡nos responde!– el valor entero de `1`. Es en este intervalo de tiempo –desde el llamado hasta la respuesta– que el flujo del programa se traslada a las líneas 14-20. Es decir, en este intervalo a las variables locales `x` e `y` de `mcd` se le asignan los valores enteros de `2` y `3`, respectivamente; luego se declara la variable local `divisor` –donde guardaremos los divisores comunes de `x` e `y`–, se ejecuta el bucle `for` y se retorna la expresión `divisor` –que se evaluará como el máximo común divisor de `2` y `3`–, es esto que responde `mcd`. Así, observaremos en la pantalla: `mcd(2,3) = 1`.

Segundo, en la línea 9 se vuelve a evocar a `mcd` y le pasamos la lista `(2,4)` y esta nos retorna `2`, donde una vez más –desde el llamado hasta la respuesta– el flujo del programa se transporta a las líneas 14-20. Así, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 8: `mcd(2,4) = 2`.

Tercero, en la línea 10 se llama a `mcd` por vez tercera y le pasamos la lista `(6,9)` y esta nos retorna `3`, donde más una vez –desde el llamado hasta la respuesta– el flujo del programa se va a las líneas 14-20. Entonces, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 9: `mcd(6,9) = 3`.

Finalmente, al observar la pantalla tendremos la impresión que las tres líneas de texto de arriba se muestran simultáneamente; no obstante, ellas se muestran secuencialmente, una después de otra, pero no lo percibimos por la rapidez en que las instrucciones de las líneas 8-10 del programa se ejecutan. □

Gracias al ejemplo anterior podemos apreciar que cada vez que se llama o invoca a una función primero vemos el nombre de la misma seguido por la lista de argumentos a pasar. Por ejemplo, en el programa anterior se llamó a `mcd` tres veces en total. Además, dicho programa nos muestra la importancia de implementar una función: ¡podemos reutilizar el código que yace en el cuerpo de una función!, basta con llamarlo tantas veces lo necesitemos.

1.3. Llamando a una función desde otra función

Ahora, veamos que es posible llamar a una función dentro del cuerpo de otra; es decir, ¡podemos reutilizar código para crear más código que a su vez podremos volver a reutilizar!

Ejemplo 1.4. En el programa de la figura 1.3 el usuario implementa la funciones `mcm` y `mcd`, donde esta última es la misma del ejemplo anterior. En primer lugar, en las líneas 5-6 declaramos que ambas son del mismo tipo, i.e., reciben dos enteros y retornan un entero. Luego, al correr el programa, se comienza ejecutando la línea 9 y se termina al ejecutar la línea 10.

```
1 #include <iostream>
2 using namespace std;
3
4 // prototipos de las funciones
5 int mcd(int x, int y);
6 int mcm(int x, int y);
7
8 int main() {
9     cout << "mcm(2,3) = " << mcm(2,3) << endl;
10    cout << "mcm(4,6) = " << mcm(4,6) << endl;
11 }
12
13 // definicion de la funcion mcd
14 int mcm(int x, int y) {
15     return (x*y) / mcd(x,y);
16 }
17
18 // definicion de la funcion mcd
19 int mcd(int x, int y) {
20     int divisor;
21     for(int n=1; n<=x && n<=y; ++n) {
22         if(x%n == 0 && y%n == 0) divisor = n;
23     }
24     return divisor;
25 }
```

Figura 1.3: funciones `mcm` y `mcd`

Primero, en la línea 9 se llama a `mcm` y le pasamos la lista de argumentos `(2,3)` y esta nos retorna el valor entero de `6`. Es en este intervalo de tiempo –desde el llamado hasta la respuesta– que el flujo del programa se traslada a las líneas 14-16. Es decir, en este intervalo a las variables locales `x` e `y` de `mcm` se le asignan los valores enteros de `2` y `3`, respectivamente; luego se evoca a nuestra vieja función `mcd` del ejemplo anterior –la cual nos responde con el valor entero de `1` y desde el llamado hasta la respuesta de esta el flujo se transporta a las líneas 19-25– y se retorna la expresión `(x*y) / mcd(x,y)` –que se evaluará como el mínimo común múltiplo de `2` y `3`–, es esto que responde `mcm`. Así, observaremos en la pantalla: `mcm(2,3) = 6`.

Segundo, en la línea 10 se evoca por vez segunda a `mcm` y le pasamos la lista `(4,6)` y esta nos retorna `12`, donde una vez más –desde el llamado hasta la respuesta– el flujo del programa se transporta a las líneas 14-16. Así, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 9: `mcm(4,6) = 12`.

Así, durante la ejecución de este programa se llamó a la función `mcd` desde la función `mcm`, lo cual está codificado en la definición de `mcm`. □

1.4. Definición recursiva de una función

Del ejemplo anterior se observa que podemos definir una función llamando a otra función. De hecho, en los ejemplos anteriores, desde la función `main` llamamos a otras funciones. En particular, podemos definir una función llamándose a sí misma; sin embargo, en su definición debemos codificar un estado basal; en otras palabras, podemos definir una función de manera recursiva y es a esta forma de definición que de ahora en adelante la llamaremos *definición recursiva* de una función.

Ejemplo 1.5. En el programa de la figura 1.4 el usuario implementa recursivamente la función `factorial1`. En primer lugar, en la línea 5 declaramos el tipo de la misma: recibe un entero y retorna un entero.

```
1 #include <iostream>
2 using namespace std;
3
4 // prototipo de la funcion factorial
5 int factorial1(int n);
6
7 int main () {
8     cout << "factorial1(1) = " << factorial1(1) << endl;
9     cout << "factorial1(2) = " << factorial1(2) << endl;
10    cout << "factorial1(3) = " << factorial1(3) << endl;
11 }
12
13 // definicion de la funcion sumar
14 int factorial1(int n) {
15     if(n == 1)
16         return 1;
17     else
18         return n * factorial1(n-1);
19 }
```

Figura 1.4: función `factorial1`

Primero, en la línea 8 se llama a `factorial1` y le pasamos el valor entero `1`, i.e., la lista de argumentos `(1)` y esta nos retorna el valor entero `1`.

Es en este intervalo de tiempo –desde el llamado hasta la respuesta– que el flujo del programa se traslada a la línea 14. Es decir, en este intervalo a la variable local `n` de `factorial1` se le asigna el valor entero `1`; luego, como la expresión `(n == 1)` se evalúa como true, se retorna el valor entero `1`, es esto que responde `factorial1`. Así, observaremos en la pantalla: `factorial1(1) = 1`.

Segundo, en la línea 9 se vuelve a evocar a `factorial1` y le pasamos el valor entero `1` y esta nos retorna `2`, donde una vez más –desde el llamado hasta la respuesta– el flujo del programa se transporta a la línea 14. Es decir, en este intervalo a la variable local `n` de `fatorial1` se le asigna el valor entero `2`; luego, como la expresión `(n == 1)` se evalúa como false, se evalúa la expresión `(n * factorial1(n-1))` –y aquí se vuelve a llamar a `factorial(1)`, i.e., el flujo del programa se transporta más una vez a la línea 14–, y es este valor entero –a saber `2`– que responde `factorial1`. Así, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 8: `factorial1(2) = 2`.

Finalmente, en la línea 10 se llama a `factorial1(3)` y esta nos retorna `6`, donde una vez más –desde el llamado hasta la respuesta– el flujo del programa se traslada a la línea 14. Es decir, en este intervalo a la variable local `n` de `fatorial1` se le asigna el valor entero `3`; luego, como la expresión `(n == 1)` se evalúa como `false`, se evalúa la expresión `(n * factorial1(n-1))` –y aquí se vuelve a llamar a `factorial(2)`, i.e., el flujo del programa se transporta más una vez a la línea 14–, y es este valor entero –a saber `3*2`– que responde `factorial1`. Así, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 9: `factorial1(3) = 6`. □

Ahora, desde otra perspectiva, produzcamos el mismo resultado del programa anterior, pero a través de una función con distinta implementación, a saber, sin emplear recursividad.

Ejemplo 1.6. En el programa de la figura 1.5 el usuario implementa –utilizando un bucle– la función `factorial2`. En primer lugar, en la línea 5 declaramos el tipo de la misma: recibe un entero y retorna un entero.

```
1 #include <iostream>
2 using namespace std;
3
4 // prototipo de la funcion factorial
5 int factorial2(int n);
6
7 int main () {
8     cout << "factorial2(1) = " << factorial2(1) << endl;
9     cout << "factorial2(2) = " << factorial2(2) << endl;
10    cout << "factorial2(3) = " << factorial2(3) << endl;
11 }
12
13 // definicion de la funcion sumar
14 int factorial2(int n) {
15     int producto = 1;
16     for(int j = 2; j <= n; ++j) {
17         producto *= j;
18     }
19     return producto;
20 }
```

Figura 1.5: función `factorial2`

Primero, en la línea 8 se llama a `factorial2(1)` y esta nos retorna el valor entero `1`. Es en este intervalo de tiempo –desde el llamado hasta la respuesta– que el flujo del programa se traslada a la línea 14. Es decir, en este intervalo a la variable local `n` de `factorial2` se le asigna el valor entero `1`; luego, a la variable local `producto` se le asigna el valor entero `1` y, como la expresión `(j <= n)` se evalúa como false, se retorna el valor entero `1`, es esto que responde `factorial2`. Así, observaremos en la pantalla: `factorial1(1) = 1`.

Segundo, en la línea 9 se evocar a `factorial2(2)` y esta nos retorna `2`, donde una vez más –desde el llamado hasta la respuesta– el flujo del programa se transporta a la línea 14. Es decir, en este intervalo a la variable local `n` de `fatorial2` se le asigna el valor entero `2`; luego, a la variable local `producto` se le asigna el valor entero `1` y, como al salir del bucle `for` la variable `producto` se evalúa como 2, `factorial2` retorna el valor `2`. Así, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 8: `factorial1(2) = 2`.

Finalmente, en la línea 10 se llama a `factorial1(3)` y esta nos retorna `6`, donde una vez más –desde el llamado hasta la respuesta– el flujo del

programa se traslada a la línea 14. Es decir, en este intervalo a la variable local `n` de `fatorial1` se le asigna el valor entero `3`; luego, a la variable local `producto` se le asigna el valor entero `1` y, como al salir del bucle `for` la variable `producto` se evalúa como 6, `factorial2` retorna el valor `6`. Así, observamos en la pantalla que debajo de la línea mostrada gracias a la línea 9: `factorial1(3) = 6`. □

Para concluir, obsérve que en la definición de `factorial2` no se evoca a ella misma, i.e., no es una definición recursiva, a diferencia de la definición de `factorial1`.

1.5. Ejercicios

1.5.1. Nivel bajo de dificultad

Ejercicio 1.1. Cree la función con el siguiente prototipo: `bool es_primo(int n);`[1] que retorne `true` si el valor entero pasado es un primo y `false` caso contrario.

Ejercicio 1.2. La sucesión de Fibonacci $(F_n)_{n\geq 0}$ se define de la siguiente manera: $F_0 = F_1 = 1$ y para todo $n \geq 2$:

$$F_n = F_{n-1} + F_{n-2}.$$

Imprima los primeros 25 números de Fibonacci.

Ejercicio 1.3. Ingrese un entero mayor que 1 desde el teclado y muestre todos los primos menores o iguales al valor ingresado.

Ejercicio 1.4. Ingrese dos vectores de $\mathbb{R}^3$. Luego, los sume, reste y multiplique, tanto escalar como vectorialmente, imprimiendo después todos estos resultados. Para ello se deberá implementar las funciones que realicen dichas operaciones vectoriales.

Ejercicio 1.5. Un *semiprimo* es un número natural que es el producto de dos (no necesariamente distintos) números primos. Ingrese un entero postivo y muestre si es o no semiprimo, en caso lo sea muestre su descomposición.

[1] A este tipo de función que retorna valores booleanos se le conoce como *predicativa*.

1.5.2. Nivel medio de dificultad

Ejercicio 1.6. Cree una función que simule el lanzamiento de dos dados justos. Luego,

1. ejecute dicha función 100 veces y muestre el resultado de dividir la cantidad de veces que la suma de los dados dio 5 entre 100;
2. ejecute dicha función 1,000 veces y muestre el resultado de dividir la cantidad de veces que la suma de los dados dio 5 entre 1,000;
3. ejecute dicha función 100,000 veces y muestre el resultado de dividir la cantidad de veces que la suma de los dados dio 5 entre 100,000;
4. ¿cuál es la probabilidad de que al tirar los dados la suma de los mismos de 5?

Ejercicio 1.7. Muestre las ocho primeros términos de la sucesión:

$$S(n) := 1^1 - 2^2 + 3^3 - 4^4 + \ldots n^n.$$

Sugerencia: utilice la función incorporada `pow`, puede importarla con `#include<cmath>`.

Ejercicio 1.8. Ingrese el numerador y el denominador de una fracción. Luego, se mmuestra su equivalente irreducible. Por ejemplo, si se ingresa

- 4 y 6, se muestra 2/3;
- -4 y 6, se muestra -2/3;
- -4 y -6, se muestra 2/3.

Ejercicio 1.9. Ingrese un número positivo b y un entero n y devuelva como resultado b^n. Defina una función con el siguiente prototipo: `double potencia(double b, int n);` para dicho fin.

Ejercicio 1.10. Ingrese los coeficientes a, b y c de una ecuación cuadrática

$$ax^2 + bx + c = 0, \qquad \text{donde} \quad a, b, c \in \mathbb{R},$$

y muestre las raíces de dicha ecuación. Para ello implemente tres funciones con los siguientes prototipos: `void reales_diferentes(double a, double b, double delta);` `void reales_iguales(double a, double b);` y `void complejas(double a, double b, double delta);` que mostrarán las raíces en caso el discriminante sea positivo, cero o negativo, respectivamente.

1.5.3. Nivel alto de dificultad

Ejercicio 1.11. Una combinación es una manera de seleccionar a los miembros de una agrupación dada, de tal manera que (a diferencia de las permutaciones) el orden de seleccióon, no importa. Ingrese un entero positivo n y un entero positivo k tal que $k \leq n$. Se debe verificar que los valores asignados a n y k sean correctos; caso contrario, se debe volver a ingresar hasta que se cumpla con lo requerido. Luego, calcule el número de combinaciones de k elementos de n elementos dados. Implemente una función con el siguiente prototipo: `int combinatoria(int n, int k);` para dicho fin. Puede utilizar la siguiente identidad:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Ejercicio 1.12. Ingrese un entero positivo y devuelva el número que resulta de invertir las cifras del valor ingresado. Por ejemplo, al ingresar el número 10230 se muestra 3201.

Ejercicio 1.13. La conjetura fuerte de Golbach es uno de los problemas sin resolver más conocidos en teoría de números. Establece que todo entero par mayor que 2 puede expresarse como la suma de dos primos. Verifique la validez de dicha conjetura imprimiendo sólo una descomposición de n como suma de dos primos para $n = 4, 6, \ldots, 2020$.

Ejercicio 1.14. La función de Ackermann está definida para todo par (m, n) de enteros no negativos como:

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ y } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ y } n > 0. \end{cases}$$

1. Justifique por qué $A(1,1) = 3$.
2. Implemente la función de Ackermann definida arriba; su función debe tener el siguiente prototipo: `int Ackermann (int m, int n);`

Ejercicio 1.15. El juego de dados conocido como craps es muy popular. Simule dicho juego; las reglas son: un jugador tira dos dados justod. Luego, se calcula la suma de los puntos de las dos caras.

1. Primera tirada:
   a) Si la suma es 7, o bien 11, el jugador gana.
   b) Si la suma es 2, 3 o 12 (conocido como "craps"), el jugador pierde.
   c) Si la suma es 4, 5, 6, 8, 9 o 10, entonces dicha suma se convierte en su "tirada".
2. El jugador continúa tirando los dados:
   a) Si la suma es 7, el jugador pierde.
   b) Si la suma es su "tirada", el jugador gana.
   c) En caso contrario, sigue jugando.