



MODULO 3

Funciones

Breve descripción

En este documento se explica qué son las funciones, cómo se definen, para qué se utilizan y cómo se utilizan.

Versión 1.1

Martin Jerman

Martin.jerman@inspt.utn.edu.ar



Introducción

Hasta este momento, hemos trabajado programas desarrollados dentro de un bloque de código llamado "main" o función main. ¿Pero, no se preguntaron por qué tiene esa forma y por qué es así? Bien, llegó el momento de entender por qué main es main y por qué tiene esa forma.

Las funciones

Las funciones son bloques de código con un nombre asociado que realizan una tarea en particular. En C todo se construye con funciones, (main es la función principal del programa).

Una función se escribe una vez y luego se invoca, es decir, se ordena que se ejecute, todas las veces que sea necesario.

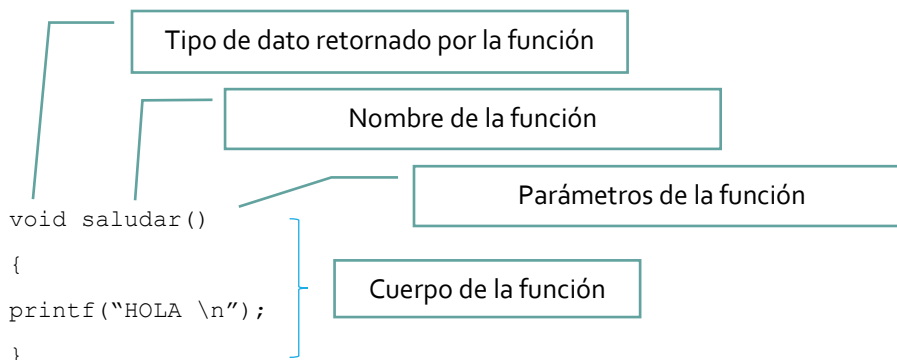
Algunas funciones requieren que se les pase uno o más datos para poder llevar a cabo su tarea; otras, no. Algunas retornan (es decir, devuelven) un dato, y otras no. En general, trataremos de que una función cumpla un único objetivo. Si al enunciar lo que debe hacer una función nos encontramos con que enumeramos tareas simples que son desarrolladas completamente por ella, lo más probable es que la función este mal planteada.

Funciones que no retornan valores

Por ejemplo, si una función tuviera que ingresar una secuencia de N números, calcular el mayor y emitirlo, no estaría bien diseñada, porque es poco probable que muchas veces se necesite realizar esa tarea compleja. Sin embargo, es cierto que muchas veces, se requiere ingresar una serie de números y almacenarlos, y que muchas veces se necesita obtener el valor mayor de un grupo de números ingresados previamente.

Entonces, lo correcto es escribir una función que permita ingresar y almacenar una secuencia de N números, escribir otra función que reciba una secuencia de N números y obtenga el mayor, y escribir una tercera función, que llame a las otras dos en el orden correcto.

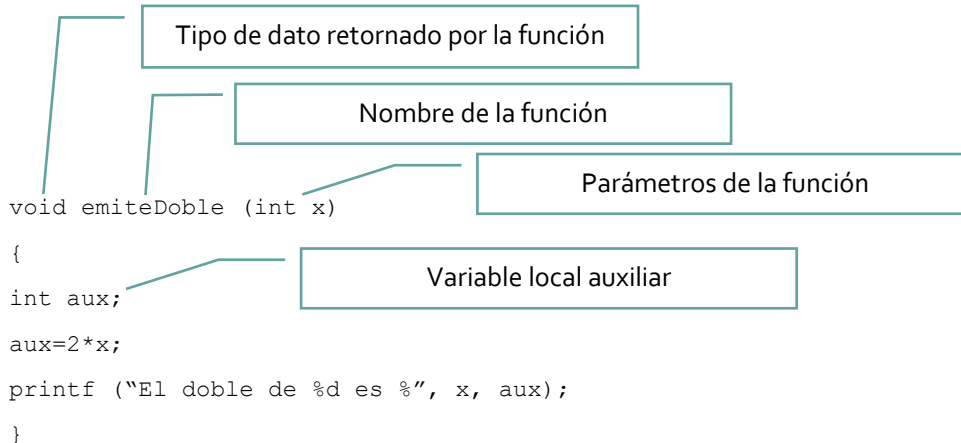
Ejemplo: definición de función que emite un saludo por pantalla.



Ejemplo de invocación a `saludar ()`:

```
int main() {
    saludar ();
    saludar ();
    printf("invocamos una vez mas\n");
    saludar ();
    system("pause");
    return 0;
}
```

Ahora veremos una función que recibe un dato entero y emite el doble de este por pantalla:



El parámetro `x` es una variable entera local, que se inicializa con el valor del argumento de llamada, `aux` es otra variable local pero no inicializada. Las variables locales solo existen mientras se está ejecutando la función. Estas se generan en la pila o stack.

Las variables locales solo existen mientras se está ejecutando la función.

Veamos un ejemplo de uso:

```
int main()
{
    int a=5, b=8;
    emitedoble(5); // línea 2
    emitedoble(8);
    printf("invocamos una vez mas");
    emitedoble(a+b+1);
    return 0;
}
```

Cuando se ejecuta la línea 2 y se invoca a `emiteDoble`, el control del programa "sale" de la función `main` y pasa a la función `emiteDoble`. Lo mismo ocurre en cada llamado a `emiteDoble`. En cada caso, al entrar el control del programa a la función, las variables de la función que estaba ejecutando (en este caso las variables `a` y `b` de `main`) se guardan en la pila del programa para cuando el control vuelva a aquella función y se crean las variables locales de `emiteDoble`, cada una con su respectivo valor que se le asigno o pasó por parámetro (en este caso `x` y `aux`). Esta forma de pasarle valores a las variables locales de una función se llama **pasaje por valor**.

Al terminar la ejecución de la función invocada, el control regresa a sentencia siguiente de la invocación con la restauración de los valores de las variables de aquella función (en este caso, de `main`). Esto es posible porque se almacena, también en la pila o stack, la dirección correspondiente a la siguiente instrucción, para que pueda continuarse la ejecución de `main`.

Como se observa, en ningún caso se altera el contenido de las variables definidas en `main`, llamadas variables estáticas. Este modo de pasar los argumentos a una función se denomina **pasaje por valor**. La variable local parámetro, que corresponde al argumento de la función, se inicializa con una copia del valor del argumento correspondiente.

Otro ejemplo: Esta es una función que recibe dos enteros llamados `m` y `n` y emite por pantalla `m%n` si `m >= n`, o `n%m` en caso contrario (recordar que el operador `%` devuelve el resto entero de división entera).



```
void resto (int m, int n)
{
    // comparar a m con n para intercambiarlos en caso de que m<n
    // aux es la variable local auxiliar para el intercambio
    int aux;
    if(m<n)
    {
        aux=m;
        m=n;
        n=aux;
    }
    printf ("El resto de la división entre %d y %d es %d\n", m, n, m%n);
}
```

Ejemplo de invocaciones a la función `resto`:

```
main()
{
    int a, b, c;
    a=100, b=27, c=39;
    resto(a,b); //m corresponde a a y n a b
    resto (b,a); //m corresponde a b y n a a
    resto(23,c); //m corresponde a 23 y n a c
    resto (200, 49); //m corresponde a 200 y n a 49
    resto(209+b-c, 25+a); //m corresponde a 209+b-c, y n a 25+a
    system ("pause");
    return 0;
}
```

Observar que la función `resto` puede recibir como argumento tanto variables, (de las cuáles hará copia de sus contenidos en sus variables locales) como valores inmediatos, (como por ejemplo 23, el cual se almacenará en el parámetro correspondiente). ¿Cómo lograr que las funciones alteren el contenido de las variables correspondientes a los argumentos de llamada?

Supongamos que queremos una función que altere el contenido de una variable entera, duplicándolo. Para resolver esta cuestión, se pasará por valor la copia de la dirección de la variable. Entonces, la función en cuestión (llamémosla `duplicar`), recibirá un puntero a la variable, conteniendo la dirección de la variable. Es decir, que `main` podría ser, por ejemplo:

```
int main()
{
    int s=4, h=18;
    printf(" s vale %d\n", s);
    //Le pasaremos a duplicar la dirección de s para que modifique su contenido
    duplicar(&s);
    printf("ahora s vale %d\n", s);
    printf(" h vale %d\n", h);
    //Ahora le pasaremos a duplicar la direccion de h para que modifique su
    contenido
    duplicar(&h);
    printf("ahora h vale %d\n", h);
    system("pause");
    return 0;
}
```

El código de `duplicar` puede ser, por ejemplo, (observar en la declaración que `duplicar` recibe un puntero):



```
void duplicar (int *p)
{
    *p=(*p) *2;
}
```

p es &s cuando se invoca duplicar (&s)

p es &h cuando se invoca duplicar (&h)

Es decir, los parámetros de la función son variables locales, que solo existen mientras se ejecuta la función. Pero como se trata de un puntero, mediante ese puntero se puede alterar el valor de la variable de llamada. Observar que en este caso, carece de sentido invocar a la función pasándole un valor inmediato (como `duplicar(100)`, por ejemplo), ya que la función exige por su diseño que se le pase la dirección de una variable para modificar su contenido. Veamos un ejemplo:

```
int main()
{
    float f;
    leer(&f);
    printf ("La variable f contiene %f\n");
    system("pause");
    return 0;
}
```

La función puede ser:

```
void leer(float *q)
{
    printf("Ingrese un valor real\n");
    scanf("%f", q); // se puede usar q para ingresar el real, ya que es la dirección
    en donde debe almacenarse el valor entrante
    system ("pause");
    return 0;
}
```

Cuando la función tiene por parámetro un **puntero**, lo que ocurre es que se le **pasa por valor** la dirección de memoria de la variable (en este caso `q`). Con lo cual, la variable puntero local (en este caso `float *q`) de `leer` apunta a una dirección de memoria fuera de su alcance, por lo cual el valor que se trabajara en `leer` en realidad está fuera de la función. Este pasaje de parámetro se lo conoce como **pasaje por referencia**. Es decir que la función recibe por parámetro una referencia a la variable original, con lo cual tiene acceso a ella para modificarla.

Para más detalles sobre este punto, puedes mirar [este link](#).

Funciones que retornan un valor

Ahora se diseñará una función que reciba tres valores enteros y retorne el promedio de estos. El valor devuelto por la función es `float`. La función (llamémosla `promedio`) podrá invocarse de estos modos:

```
int main ()
{
    int a=4,b=10,c=8;
    float r;

    r = promedio(a,b,c);
}
```



```

printf("El promedio entre %d, %d y %d es %f \n",a,b,c,r);
r= promedio(100, 34, a);
printf("El promedio entre %d,%d y %d es %f \n",100,34,a,r);
printf(("El promedio entre %d, %d y %d es %f \n",70,30,b,promedio(70,30,b));
r = promedio(34,56,a) + promedio(a,b,c) - 100;
printf("El resultado de promedio(34,56,a) + promedio(a,b,c) - 100
es %f\n",r);
return 0;
}

```

Observar que la invocación a `promedio` se utiliza como si fuera una expresión `float`, y debe, o bien ser almacenada en una variable, o ser usada en una expresión aritmética, o para construir una expresión lógica, o directamente en un `printf`. Nada de esto era posible con las funciones que devolvían `void`. La función `promedio` puede diseñarse así:

```

float promedio (int x, int y, int z)
{
    float aux=(x+y+z)/3;
    return aux;
}

```

Otro ejemplo: otra forma de ingresar un valor desde teclado, si diseñamos una función del siguiente modo:

```

int lee()
{
    int aux;
    printf("Ingrese un valor entero");
    scanf("%d", &aux);
    return aux;
}

```

La variable `aux` es local; sólo existe mientras se ejecuta `lee()`. La dirección de `aux` corresponde a alguna posición dentro de la zona de pila. El valor almacenado en `aux` se retorna. La invocación puede ser:

```

int main()
{
    int a,b;
    a=lee();
    printf("a vale %d\n", a);
    b=lee();
    printf("b vale %d\n", b);
    a=lee();
    printf("Ahora a vale %d\n", a);
    return 0;
}

```

¿Dónde se ubican las funciones?

Hasta este momento, las funciones que utilizamos como `scanf()` o `printf()` están definidas en archivos separados llamados bibliotecas que incluimos con la instrucción de precompilador



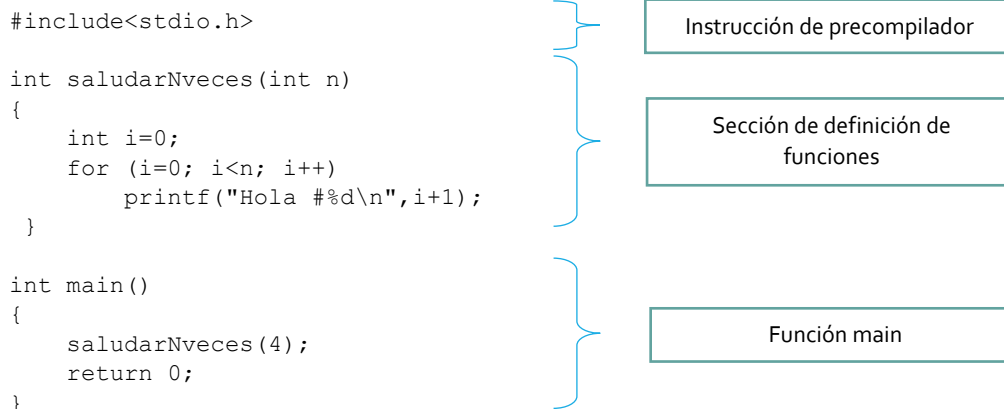
#include. Dado que el armado de bibliotecas no entra en este curso, las funciones que definiremos las colocaremos en el mismo archivo de nuestro código fuente.

Para hacerlo hay dos formas:

1. Definirlas antes de main
2. Prototiparlas antes de main y definir las después

Definirlas antes de main

En este caso lo que se hace es colocar toda la definición de la función antes de la función `main`. De este modo, todo nuestro archivo fuente quedaría con la siguiente forma:

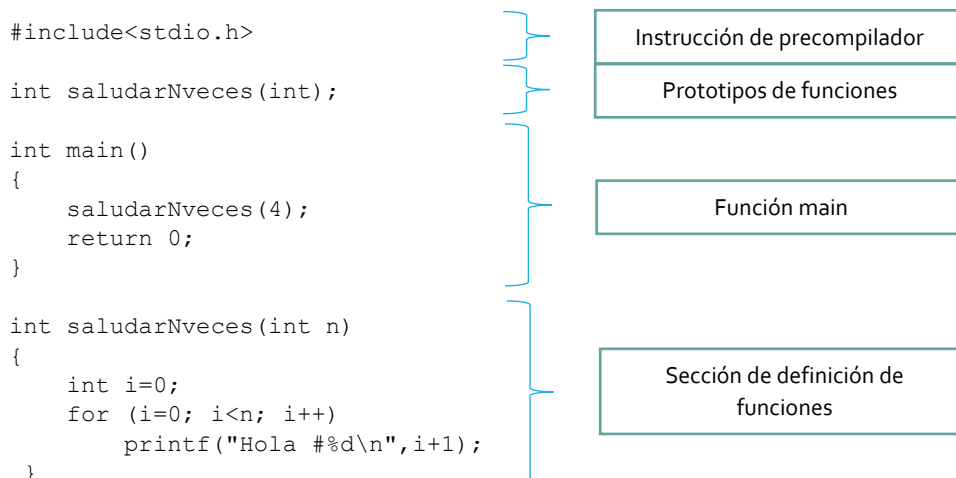


Parecería que éste es el caso que más se utiliza, pero no. El problema de definir las funciones en esta forma es que el `main` queda al final de toda una larga lista de funciones, lo que hace que entender el programa sea difícil. Sería ideal poder tener el `main` en la parte superior del código fuente para entender lo que el programa ejecuta y luego el desarrollo detallado de cada función. Es por ello por lo que existe otra forma de definir funciones: a través de prototipos.

Prototiparlas antes de main y definir las después

Este es el caso más usado. Básicamente lo que se hace es presentar al principio los prototipos (o encabezados) de las funciones que se van a desarrollar después del `main`. Esto sirve para decirle al compilador que dichas funciones están definidas al final y no al principio.

Veamos la estructura:





Con esta forma, cuando se tienen muchas funciones desarrolladas, queda en la parte superior del código fuente la lista de funciones que se desarrollaron junto al `main` que describe lo que el programa realiza.

Pasaje por valor vs. Pasaje por referencia

Hasta este momento hemos visto que cada variable que se pasa por parámetro en el llamado a la función, la función en realidad recibe una copia el valor de dicha variable. Con lo cual, si la función modifica dicho valor, el valor la variable fuera de la función no se ve afectada. Este es el efecto del pasaje por valor. Esta forma de pasaje de parámetros tiene una limitación: que, si mi función necesita devolver más de un resultado, no es posible dado que la única forma que una función puede devolver resultados es a través de la instrucción `return`.

Entonces, ¿cómo hacer que una función retorne más de un valor?

La idea es hacer que los parámetros que son de entrada también sean de salida.

¿Y cómo es que logro hacer que esos parámetros también sean de salida?

Pues, pasando las variables **por referencia**.

El pasaje de variables por referencia consiste en pasar no el valor de la variable a la función, sino la dirección de memoria de dicha variable. De esta manera como la función recibe una dirección de memoria, cuando se quiera referir al contenido del espacio de memoria al que apunta, se obtendrá el valor de la variable.

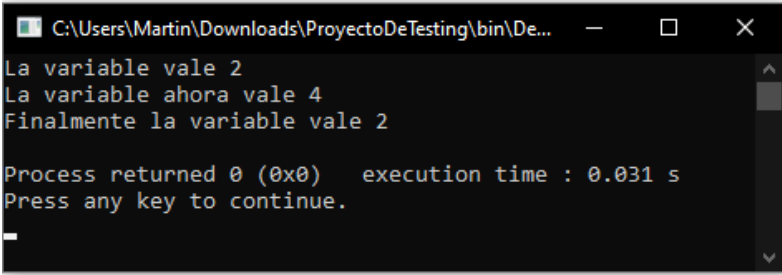
¿Y cuál es el efecto de esto? Pues dado que la función trabaja con la dirección de memoria de la variable, el impacto de modificar el valor contenido en dicho espacio de memoria sí modifica el valor del parámetro que fue pasado.

Veamos un ejemplo donde se evidencia en **pasaje por valor**:

```
void f1(int x){
    x+=2;
    printf("La variable ahora vale %d\n",x);
}

int main(){
    int y=2;
    printf("La variable vale %d\n",y);
    f1(y);
    printf("Finalmente la variable vale %d\n",y);
    return 0;
}
```

La salida de este programa es:



```
C:\Users\Martin\Downloads\ProyectoDeTesting\bin\De...
La variable vale 2
La variable ahora vale 4
Finalmente la variable vale 2

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Como se puede ver, el valor de la variable pasada a la función `f1` que se modifico dentro de dicha función, no tuvo impacto en la variable en `main`.

Ahora bien, veamos el ejemplo modificado con el **pasaje por referencia**:



```
void f1(int* x){
    *x+=2;
    printf("La variable ahora vale %d\n",*x);
}

int main(){
    int y=2;
    printf("La variable vale %d\n",y);
    f1(&y);
    printf("Finalmente la variable vale %d\n",y);
    return 0;
}
```

La salida de este programa es la siguiente:

En el pasaje por referencia la clave esta en las **variables del tipo puntero**. ¿Cuáles son estas variables? Son aquellas que tienen el operador de indirección * luego del tipo de dato. En el ejemplo, la declaración:

```
int* x
```

Es la declaración de una variable puntero a entero llamada x. Esto significa que x no contendrá un entero sino una dirección de memoria cuyo contenido es un entero.

Ahora bien, dado que la variable por si misma es una dirección de memoria, si la operamos directamente estaríamos operando dicha dirección de memoria, y el valor apuntado. Por lo cual, si se quiere trabajar con el valor apuntado por la variable, es necesario utilizar el operador de indirección *. En el ejemplo:

```
*x+=2;
```

Finalmente, nos queda entender cómo es que podemos pasar una variable ya definida en nuestra función `main` a esta nueva función. Para ello se utiliza el operador de dirección & (ampersand). Con esto, la llamada de a la función `f1` recibirá una dirección de memoria de un entero de la siguiente manera:

```
f1(&y);
```

Veamos un ejemplo más:

```
void resetVariables(int* x, int* y, float* z){
    *x=1;
    *y=2;
    *z=2.2;
}
```



```
int main() {  
    int j=33,k=25;  
    float iva=21.0;  
    printf("Valores iniciales: %d-%d-%2.2f\n",j,k,iva);  
    resetVariables(&j,&k,&iva);  
    printf("Luego del reset: %d-%d-%2.2f\n",j,k,iva);  
    return 0;  
}
```

La salida de este programa es la siguiente:

```
C:\Users\Martin\Downloads\ProyectoDeTesting\bin\...  
Valores iniciales: 33-25-21.00  
Luego del reset: 1-2-2.20  
  
Process returned 0 (0x0)   execution time : 0.060 s  
Press any key to continue.  
_
```

Características introducidas por las funciones

Ahora que entendemos de funciones, es importante repasar algunas características que éstas introducen a la programación estructurada que son muy relevantes para el diseño de las soluciones.

Modularidad

El aporte más importante que hizo el diseño estructurado fue la idea de que, para resolver un problema complejo de desarrollo de software, conviene **separarlo en partes más pequeñas**, que se puedan diseñar, desarrollar, probar y modificar, de manera sencilla y lo más independientemente posible del resto de la aplicación.

Esas partes, cuando se quiere usar un nombre genérico, habitualmente se denominan módulos. De allí que otro nombre para la programación estructurada, luego caído en desuso, fue "programación modular".

El diseño estructurado, al plantear la separación del sistema en módulos, se basó en las propias funciones del sistema. Esto es, **los módulos de la programación estructurada serían los procedimientos y funciones**. Por lo tanto, al modularizar, lo que hacíamos era tomar nuestra solución del problema, y separarla en partes. Por lo tanto, en programación estructurada modularizamos la solución, el "cómo" del desarrollo.

Cohesión

La cohesión tiene que ver con que **cada módulo del sistema se refiera a un único proceso o entidad**. A mayor cohesión, mejor: el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener.

En el diseño estructurado, se logra **alta cohesión cuando cada módulo** (función o procedimiento) **realiza una única tarea trabajando sobre una sola estructura de datos**. Una prueba que se suele hacer a los módulos funcionales para ver si son cohesivos es analizar que puedan describirse con una



oración simple, con un solo verbo activo. Si hay más de un verbo activo en la descripción del procedimiento o función, deberíamos analizar su partición en más de un módulo, y volver a hacer la prueba.

Acoplamiento

El acoplamiento mide el grado de relacionamiento de un módulo con los demás. A menor acoplamiento, mejor: el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener.

En el diseño estructurado, se logra bajo acoplamiento reduciendo las interacciones entre procedimientos y funciones, reduciendo la cantidad y complejidad de los parámetros y disminuyendo al mínimo los parámetros por referencia y los efectos colaterales.

El efecto colateral

En Informática, se dice que una función o expresión tiene **efecto colateral**, o **efecto de lado** o **efecto secundario** si ésta, además de retornar un valor, modifica el estado de su entorno.

Por ejemplo, una función puede modificar una variable global o estática, modificar uno de sus argumentos (pasaje por referencia), escribir datos a la pantalla o a un archivo, o leer datos de otras funciones que tienen efecto secundario.

Los efectos secundarios frecuentemente hacen que el comportamiento de un programa sea más difícil de predecir. De hecho, la programación imperativa generalmente emplea funciones con efecto secundario para hacer que los programas funcionen, mientras que la programación funcional en cambio se caracteriza por minimizar estos efectos.

Funciones recursivas

Una función se dice recursiva si durante su ejecución se invoca directa o indirectamente a sí misma. Esta invocación depende al menos de una condición que actúa como condición de corte que provoca la finalización de la recursión. Un algoritmo recursivo consta de:

- Al menos un caso trivial o base, es decir, que no vuelva a invocarse y que se ejecuta cuando se cumple cierta condición, y
- El caso general que es el que vuelve a invocar al algoritmo con un caso más pequeño del mismo.

Los lenguajes que soportan recursividad dan al programador una herramienta poderosa para resolver ciertos tipos de problemas reduciendo la complejidad u ocultando los detalles del problema. La recursión es un medio particularmente poderoso en las definiciones matemáticas. Para apreciar mejor cómo es una llamada recursiva, estudiemos la descripción matemática de factorial de un número entero no negativo:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1) * (n-2) * \dots * 1 = n * (n-1)! & \text{si } n > 0 \end{cases}$$

¡Por supuesto, no podemos hacer la multiplicación aún, puesto que no sabemos el valor de 3!

$$\begin{aligned} 3! &= 3 * 2! \\ 2! &= 2 * 1! \\ 1! &= 1 * 0! \\ 0! &= 1 \end{aligned}$$



Podemos ahora completar los cálculos

$1! = 1 * 1 = 1$
 $2! = 2 * 1 = 2$
 $3! = 3 * 2 = 6$
 $4! = 4 * 6 = 24$

Con este detalle de cómo se calcula el factorial, podemos definir la función en C de la siguiente manera:

```
long factorial (int x)
{
    if (x == 0)
        return 1;
    else
        return x * factorial (x - 1);
}
```

Es importante observar que:

1. la existencia del **caso base** (en este caso cuando x es igual a cero) dado que si este no existe o la condición sea inalcanzable (que nunca se dé), se entrará en un **ciclo infinito** a nivel llamadas de funciones. Estos errores pueden ser de programación o de definición del problema/función, dado que podemos ignorar condiciones necesarias para que funcione.
2. La definición de la llamada recursiva a la función que no es el mismo que a la función original, sino que *"es el mismo problema, pero más pequeño"*. Es decir, los parámetros que se le pasan a la función son diferentes que van acercando la forma de la función al caso base.

Las funciones recursivas se basan en resolver el mismo problema haciéndolo más pequeño hasta llegar al caso base

Diseño de Algoritmos Recursivos

Para que una función o procedimiento recursivo funcione se debe cumplir que:

- Existe una salida no recursiva del procedimiento o función y funciona correctamente en ese caso.
- Cada llamada al procedimiento o función se refiere a un caso más pequeño del mismo.
- Funciona correctamente todo el procedimiento o función.

Para poder construir cualquier rutina recursiva teniendo en cuenta lo anterior, podemos usar el siguiente método:

1. Primero, obtener una definición exacta del problema.
2. A continuación, determinar el tamaño del problema completo a resolver. Así se determinarán los valores de los parámetros en la llamada inicial al procedimiento o función.
3. Resolver el caso base en el que problema puede expresarse no recursivamente. Esto asegurará que se cumple el punto 1 de la prueba anterior.
4. Por último, resolver correctamente el caso general, en términos de un caso más pequeño del mismo problema (una llamada recursiva).

Cuando el problema tiene una definición formal, posiblemente matemática, como el ejemplo del cálculo del factorial, el algoritmo deriva directamente y es fácilmente implementable en otros casos debemos encontrar la solución más conveniente.



Cómo funcionan los Algoritmos Recursivos

Para entender cómo funciona la recursividad es necesario que tengamos presente las reglas y los tipos de pasaje de parámetros provistos por C.

Si un procedimiento o función p invoca a otro q , durante la ejecución de q se reservarán locaciones de memoria para todas las variables locales de q y para los parámetros pasados por valor. Al terminar la ejecución de q este espacio es desocupado. Ahora bien, si durante la ejecución de q se produce una llamada a sí mismo, tendremos una segunda "instancia" de q en ejecución, la primera instancia se suspende hasta que la instancia recursiva termine. Antes de iniciarse la ejecución recursiva de q , se ocupan nuevas locaciones de memoria para las variables locales y parámetros por valor de q . Cualquier referencia a estas variables accederá a estas locaciones. Las locaciones reservadas durante la ejecución inicial de q son inaccesibles para la 2da. instancia de q .

Cuando la 2da. instancia de q termine, el control vuelve a la primera instancia de q , exactamente al lugar siguiente a la llamada recursiva. Cualquier referencia a las variables locales o parámetros por valor de q accederá a las locaciones reservadas inicialmente, inaccesibles para la segunda instancia de q . Como vemos, **no se conoce la cantidad de memoria que va a utilizarse** al ejecutar un procedimiento o función recursivo sino que se produce una **asignación dinámica de memoria**, es decir, a medida que se producen instancias nuevas, se van "apilando" las que quedan pendientes: se ponen al menos tres elementos en la pila: una para la dirección de vuelta, otra para el/los parámetro/s formal/es y otra para el identificador de la función que en esencia es un parámetro pasado por variable. Cuando la última instancia de la recursión - elige el caso base - se cumple, se "des apila" esa instancia y el control vuelve a la instancia anterior; así hasta "des apilar" todas las instancias. Esta "pila" que se va generando en memoria es importante tenerla en cuenta por lo que debemos asegurarnos de que el algoritmo recursivo no sea divergente. Vamos a ver cómo funciona la recursión en algunos problemas vistos a través de una traza dónde iremos guardando los valores de los parámetros formales.

Volvamos al ejemplo del factorial:

```
long factorial (int x)
{
    if (x == 0)
        return 1;
    else
        return x * factorial (x - 1);
}
```

Si la llamada inicial es `factorial(5)`. La primera instancia se coloca en la pila, luego a medida que se va invocando la función, vamos apilando el valor del parámetro formal que se envía: Como $x=0$, el algoritmo va por la rama de la condición verdadera llegando al caso base obteniendo `factorial(0) = 1`.

0	Instancia 6
1	Instancia 5
2	Instancia 4
3	Instancia 3
4	Instancia 2
5	Instancia 1
x	

Llegado a ese punto, se empieza a des apilar la pila desde la instancia 6 hasta la instancia 1, reemplazando los valores que fuimos obteniendo:

1	Des apilando Instancia 6	0	factorial(0)	1
2	Des apilando Instancia 5	1	1 * factorial(0)	1 * 1 = 1
3	Des apilando Instancia 4	2	2 * factorial(1)	2 * 1 = 2
4	Des apilando Instancia 3	3	3 * factorial(2)	3 * 2 = 6
5	Des apilando Instancia 2	4	4 * factorial(3)	4 * 6 = 24



6 Des apilando Instancia 1

5 ~~5 * factorial(4)~~ 5 * 24 = 120
x x * factorial(x-1)

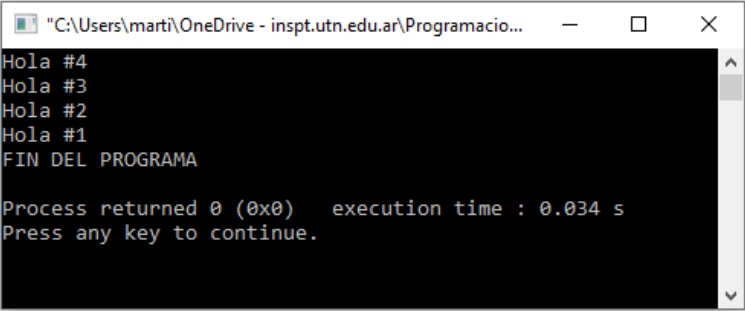
Un último ejemplo: veamos la función `saludarNveces()` la cual hace uso de la recursividad para llevar la cuenta de las veces que se saluda:

```
int saludarNveces(int n)
{
    if (n == 0)
        return 1;
    else
    {
        printf("Hola #%d\n", n);
        saludarNveces(n-1);
    }
}
```

Y ahora utilicémosla en un programa:

```
int main()
{
    saludarNveces(4);
    printf("FIN DEL PROGRAMA\n");
    return 0;
}
```

En este ejemplo, obtenemos la siguiente salida



```
"C:\Users\marti\OneDrive - inspt.utn.edu.ar\Programacio...
Hola #4
Hola #3
Hola #2
Hola #1
FIN DEL PROGRAMA

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

¿Pero qué ocurre si cambiamos el orden de las instrucciones al momento de llamar a la función recursivamente?

```
int saludarNveces(int n)
{
    if (n == 0)
        return 1;
    else
    {
        saludarNveces(n-1);
        printf("Hola #%d\n", n);
    }
}
```



```
"C:\Users\marti\OneDrive - inspt.utn.edu.ar\Progra...
Hola #1
Hola #2
Hola #3
Hola #4
FIN DEL PROGRAMA

Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

Esto ocurre porque en el segundo caso llamamos a la recursividad antes de mostrar los mensajes por pantalla, con lo cual la impresión de los mensajes también se apila y por ende, **se muestra en el orden inverso**.

Un último ejemplo: desarrollar la función `potencia(base,exponente)` que devuelva el resultado de $base^{exponente}$. La función nos quedaría así:

```
int potencia (int base, int exponente){
    if (exponente==0)
        return 1;
    else
        return base * potencia (base, exponente - 1);
}
```

La función `potencia` se puede traducir a “multiplicaciones sucesivas”. Por tanto, la idea es multiplicar la base tantas veces como `exponente` indique. Entonces la lógica de la función es usar el `exponente` como un “*contador*” en las sucesivas llamadas recursivas que irán cargando a la pila la multiplicación de la base hasta llegar al 1 (que es neutral para la multiplicación).

Esta otra estrategia de recursividad es interesante dado que se usa un parámetro para el control de la recursión, pero se utiliza otro para el cálculo que la función realiza.

¿Por qué escribir programas recursivos?

Tenemos argumentos por los cuales elegir la recursividad:

- Son más cercanos a la descripción matemática.
- Generalmente más fáciles de analizar.
- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

Pero entonces, resta saber cuándo es que conviene la recursividad.

¿Cuándo usar recursividad?

- Para simplificar el código.
- Cuando la estructura de datos es recursiva ejemplo: listas, árboles.

¿Cuándo NO usar recursividad?

- Cuando los métodos usen vectores largos.
- Cuando el método cambia de manera impredecible de campos.
- Cuando las iteraciones sean la mejor opción.

Clasificaciones recursión directa vs recursión indirecta

Cuando una función incluye una llamada a sí misma se conoce como **recursión directa**. Este es el caso de la función factorial y los ejemplos antes vistos. Ahora bien, cuando una función llama a otra



función y esta invoca a la función anterior, se conoce como **recursión indirecta**. Veamos el siguiente ejemplo:

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

int par(int n) {
    if (n == 0) return 1;
    return impar(n-1);
}

int impar(int n) {
    if (n == 0) return 0;
    return par(n-1);
}

int par(int n);
int impar(int n);
int main() {
    int x;
    printf( "Introduzca un entero:\n " );
    scanf( "%d", &x );
    if (par(x)==1)
        printf( "\n %d Es par\n", x);
    else
        printf( "\n %d Es impar\n", x);
    system("Pause");
    return 0;
}
```

Como se ve en el ejemplo, las funciones par e impar se llaman mutuamente dependiendo del valor de n. entonces decimos que las funciones par e impar son **indirectamente recursivas**.



Un ejemplo más avanzado

Analicemos el siguiente código de ejemplo:

```
#include<stdio.h>
int menu();
void calcularSumas();
void calcularMultiplicaciones();
void error(int);
void salir();

int main()
{
    int opcion;
    opcion = menu();
    while (opcion != 3){
        switch(opcion){
            case 1:
                calcularSumas(); break;
            case 2:
                calcularMultiplicaciones(); break;
            default:
                error(1);
        }
        opcion = menu();
    }
    salir();
    return 0;
}

int menu(){
    int opcion;
    printf("Ingrese una opcion:\n");
    printf("1- Calcular sumas\n");
    printf("2- Calcular multiplicaciones\n");
    printf("3- Salir\n");
    scanf("%d",&opcion);
    printf("\n");
    return opcion;
}

void calcularSumas(){
    int a,b;
    printf("Ingrese un valor:");
    scanf("%d",&a);
    printf("Ingrese otro valor:");
    scanf("%d",&b);
    printf("El resultado es %d\n\n",a+b);
}

void calcularMultiplicaciones(){
    printf("*** Funcion no implementada!\n\n");
}

void error(int n){
    printf("*** ERROR - ");
    switch(n){
        case 1:
            printf("Opcion invalida!"); break;
        case 2:
            printf("Otro error!"); break;
    }
    printf(" ***\n\n");
}
```



```
}  
  
void salir(){  
    printf("Gracias por utilizar este programa.\n");  
}
```

En este último ejemplo vemos un programa completamente funcional en el que se implementa un menú de opciones, las cuales no todas están implementadas y además se controlan algunos errores. Primero, observemos la función `menu()`. Esta función no recibe parámetros, pero si devuelve uno, la opción elegida. Esta función es muy típica para el manejo de menús dado que hace lo que se pretende de ella: mostrar un menú y devolver la opción elegida. Luego la opción elegida es utilizada dentro de `main` para saber que hacer según lo que se implementa en el `switch`. Es importante destacar que por cada opción válida de la función `menu()` debe haber una entrada valida en el `switch`.

Ahora observemos la opción de salida. El “salir” del programa es una opción válida, pero dado que utilizamos el valor de dicha opción en la condición del `while`, no podemos ejecutar la función de `salir()` como una opción dentro del `switch`. Por ello `salir()` se ejecuta al final del programa. Otra forma de implementar el ciclo seria utilizando una variable bandera llamada `salida` que valga siempre cero hasta que se selecciona la opción salir. Dicha función cambiaria a la variable `salida` a 1 y con ello se saldría del ciclo `while`.

Hasta aquí, vemos que en las primeras 25 líneas de código tenemos:

- Todos los prototipos de las funciones
- Una función `main` simple, compacta, que explica lo que hace el programa

De esta forma queda en el final del código, todo el detalle de qué hace y cómo funciona cada función. Pero en este caso, a pesar de que todas las funciones están implementadas, no todas están desarrolladas. La función `calcularSumas()` que responde a la opción 1 si está desarrollada porque ejecuta lo que se pretende, sumar. Pero ese no es el caso de la función `calcularMultiplicaciones()` que en lugar de ejecutar algo, simplemente muestra un mensaje de aviso que aun no fue desarrollada. Esto es totalmente valido al momento de desarrollar software. De hecho, es una metodología de desarrollo llamada top-down: ir desde lo general (desde el `main`) hasta lo particular (cada función). Esta forma de encarar la resolución de un programa es muy útil dado que siempre se puede delegar, en una función a desarrollar más tarde, determinadas acciones.

Finalmente, el manejo de errores. Es verdad que no se manejan todos los errores posibles, pero sí se plantea como manejar uno de ellos: la opción invalida. Aquí mostramos una de ellas, a través de la sentencia `default` del `switch`. Esta sentencia nos queda perfecto dado que el `switch` ingresa si el valor de la variable a evaluar no es válido. Entonces implementamos una función `error()` la cual podría implementar todos los mensajes de errores que el programa podría tener. En este caso solamente implementamos el error número 1 para el ingreso de opción invalida. Pero en un desarrollo más grande se pueden implementar varios `códigos de error` y para cada uno, un mensaje o acciones diferentes.

Claro que esto es un ejemplo simple para ver la forma que tiene en código fuente completo, la estructura y cómo está organizado. Es un ejemplo útil para usar de base para desarrollos más grandes.