



UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS



Tecnicatura en diseño
y programación de Videojuegos

UNL VIRTUAL



Introducción a la programación

Unidad Temática Número 4

Funciones

Objetivo: Implementar Funciones en C++

Temas: Subprogramas, Funciones en C++, Variables locales y globales, Sobrecarga de Funciones, Recursividad

Introducción

Cuando vimos la programación estructurada dijimos que todos los problemas computacionales se podían resolver con las tres estructuras: secuencia, iteración (while for) y decisión (if switch), y esto es cierto, pero al crear un programa medianamente complejo el texto en si del programa se vuelve muy complicado de manejar y sobre todo corregir por otras personas. En los casos en que el problema es de mediano a extremadamente complejo se utiliza (además de la programación estructurada) la programación modular, de la cual ya dimos un vistazo en la unidad de Algoritmos.

Como recordaran la idea era dividir el problemas en problemas más simples que resueltos resolvían el problema general.

Estos subproblemas los resolveremos con pequeños programas a Los que llamaremos subprogramas.

Diremos que un **subprograma** es un conjunto de acciones, diseñado generalmente en forma separada y cuyo objetivo es resolver una parte del problema. Estos **subprogramas** pueden ser invocados desde diferentes puntos de un mismo programa y también desde otros **subprogramas**.

La finalidad de los **subprogramas**, es simplificar el diseño, la codificación y la posterior depuración de los programas.

Las ventajas de usar subprogramas

Reducir la complejidad del programa y lograr mayor modularidad.

Permitir y facilitar el trabajo en equipo. Cada diseñador puede atacar diferentes módulos o subprogramas.

Facilitar la prueba de un programa, ya que cada subprograma puede ser probado previamente y en forma independiente.

Optimizar el uso y administración de memoria.

Crear librerías de subprogramas para su posterior reutilización en otros programas.

Cuándo emplear subprogramas?

Es conveniente emplear subprogramas cuando:

Existe un conjunto de operaciones que se utilizan más de una vez en un mismo programa.

Existe un conjunto de operaciones útiles que pueden ser utilizadas por otros programas.

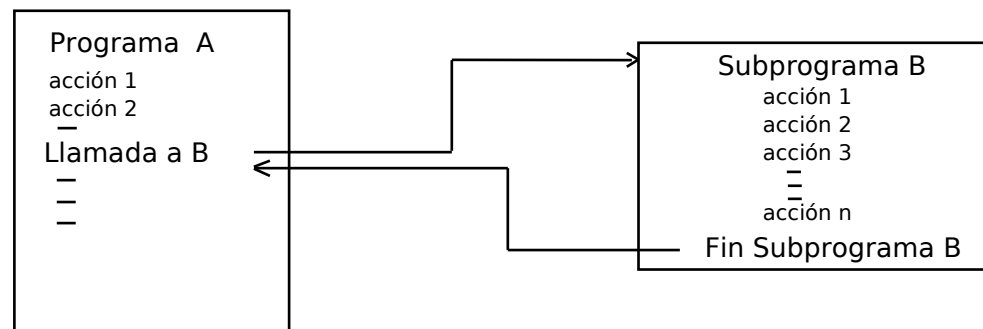
Se desea agrupar procesos para lograr una mayor claridad en el código del programa.

Se pretende crear bibliotecas que permitan lograr mayor productividad en el desarrollo de futuros programas.

Al plantear la solución a un problema que queremos resolver, diseñamos un programa al que llamaremos **programa principal**. Incluirá entre sus acciones una sentencia especial que permite *llamar* al subprograma.

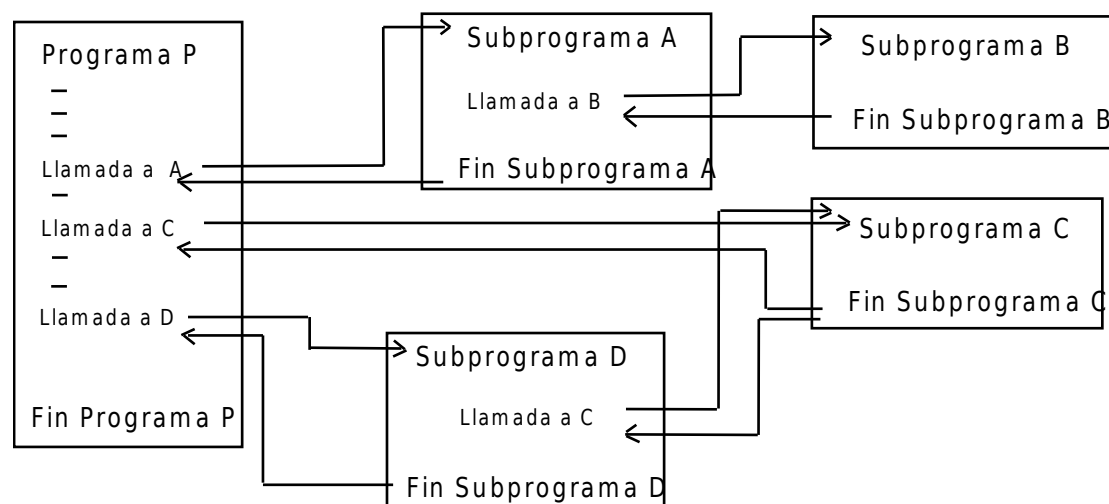
En la etapa de ejecución del programa, al encontrar la llamada al **subprograma**, se transfiere el control de ejecución a éste y comienzan a ejecutarse las acciones previstas en él. Al finalizar la ejecución del **subprograma** y obtenidos los resultados planeados, el control retorna al programa que produjo la llamada, y continúa la ejecución del programa principal.

Observemos lo anterior gráficamente:



En el esquema, **A** es un programa que contiene una acción o *llamada* al **subprograma B**. Cuando el control de ejecución llega a la llamada, comienzan a ejecutarse las acciones descritas en **B**. Al finalizar **B**, el control vuelve al programa principal **A**, para continuar con las acciones restantes. Decimos que **A** es *cliente del subprograma B*.

Este esquema simple: **programa principal - subprograma**, puede adquirir mayor complejidad con la existencia de otros **subprogramas**. El control puede pasar del programa principal a cualquier subprograma, o de un subprograma a otro, pero siempre se retorna al lugar que produjo el llamado.



En el gráfico hemos representado el programa P que contiene 3 llamadas a subprogramas diferentes, A, C, D. A su vez, los subprogramas A y D son clientes de otros subprogramas: el subprograma A llama al B, y el D al C.

Durante la ejecución de P, se encuentra la acción de llamada a A, el control pasa a dicho subprograma y comienzan a ejecutarse las acciones que él describe hasta encontrar la llamada a B; en este momento, el control pasa al subprograma B, se ejecutan sus acciones y al finalizar éste retorna al punto desde donde fue llamado en el subprograma A. Luego se continúan ejecutando las acciones de A, y al finalizar, vuelve el control al punto de llamada en el programa principal P.

Continúa la ejecución de P hasta encontrar la llamada a C, pasando el control a este subprograma, se ejecutan sus acciones y retorna a P en el punto de llamada. Continúa P hasta hallar la llamada a D; pasa el control a D, se ejecutan sus acciones hasta encontrar la invocación al subprograma C; comienzan a ejecutarse las acciones de C, y al terminar el control retorna a D en el mismo lugar donde se llamó a C. Continúa la ejecución de D, para luego retornar al programa principal P.

Observación: nótese que el mismo subprograma C fue llamado desde el programa principal P y desde el subprograma D. En otras palabras: P y D son clientes del subprograma C.

Tipos de Subprogramas

Todos los lenguajes de programación admiten subprogramas. Se los denomina **funciones, procedimientos, subrutinas**. C++ emplea el subprograma **función**.

Funciones en C++

En C++ emplearemos funciones siempre. Todo programa C++ consta de una o más funciones y una de ellas debe llamarse main.

La ejecución de un programa C++ comienza por las acciones planteadas en main.

Si un programa C++ contiene varias funciones estas pueden definirse en cualquier lugar del programa pero en forma independiente una de otra (no incluir una función en otra).

Como vimos en la introducción a subprogramas se puede acceder (llamar) a una función desde cualquier lugar del programa y hacerlo varias veces en un mismo programa. Al llamar a una función el control de ejecución pasa a la función y se llevan a cabo las acciones que la componen; luego el control retorna al punto de llamada.

Puede existir intercambio de información entre el programa o módulo que llama a la función y ésta. La función devuelve un solo valor (o ninguno) a través de la sentencia **return** y estudiaremos que puede recibir y devolver información a través de sus parámetros o argumentos.

```
//Ejemplo: calcular el promedio entre 3 valores
//enteros que se ingresan como datos de entrada.

#include <iostream.h>
#include<conio.h>
#include<iomanip.h>

float promedio3(int x,int y,int z);

void main( )
{ int d1, d2, d3;
  cout <<"Ingrese el primer dato:" ; cin >> d1;
  cout <<"Ingrese el segundo dato:"; cin >> d2;
  cout <<"Ingrese el tercer dato:" ; cin >> d3;
  float p = promedio3(d1, d2, d3);
  cout <<setprecision(3)<<"El promedio es:" << p;
  getch();
}

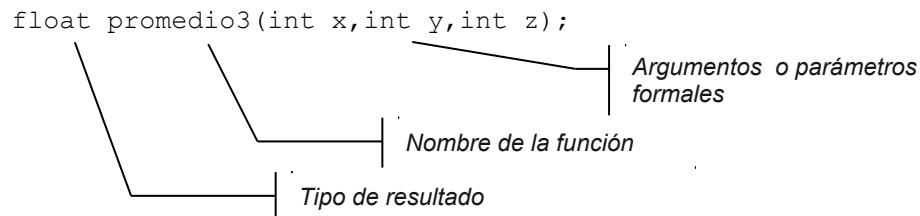
float promedio3(int x,int y,int z)
{ float w=(x+y+z)/3.0 ;
  return (w);
}
```

Diagram annotations:

- Prototipo de la función promedio**: Points to the function declaration `float promedio3(int x,int y,int z);`.
- Llamada a la función promedio3**: Points to the function call `float p = promedio3(d1, d2, d3);` inside the `main` function.
- Definición de función promedio3**: Points to the function definition `float promedio3(int x,int y,int z) { ... }`.

Declarando y definiendo funciones en C++.

C++ exige declarar una función antes de que sea utilizada. Para ello debemos escribir la cabecera de la función; en ella planteamos el tipo de resultado que devuelve la función, su nombre y sus argumentos). Esta cabecera recibe el nombre de **prototipo** de la función. Podemos tomar como ejemplo la función *promedio3* empleada en el recuadro anterior.



Usualmente se escribe el prototipo de la función antes de `main{ }` pero recordemos que en C++ es posible efectuar la declaración de un elemento en cualquier lugar del programa con la condición de hacerlo antes de invocar dicho elemento.

Resultados de una función C++

Si una función devuelve un resultado, se debe especificar su tipo antes del nombre o identificador de la función; y en el cuerpo, debemos emplear una variable o expresión de igual tipo como argumento de la sentencia **return()**.

```
float promedio3(int x,int y,int z)
{ float w=(x+y+z)/3.0 ;
  return w;
}
```

Obsérvese en el ejemplo que el tipo de la función **promedio3** y el tipo de la variable **w** que será retornada coinciden. También puede plantearse:

```
float promedio3(int x,int y,int z)
{
  return((x+y+z)/3.0) ;
}
```

Es posible que una función no devuelva resultados; entonces se especifica el tipo nulo **void** en su prototipo y en la correspondiente definición.

```
void promedio3(int x,int y,int z)
{ float w=(x+y+z)/3.0 ;
  cout << "el promedio es:" << w << endl;
}
```

Intercambio de información desde funciones C++

El empleo de funciones nos permite diseñar rutinas que pueden ser reutilizadas dentro del mismo programa o desde otros programas.

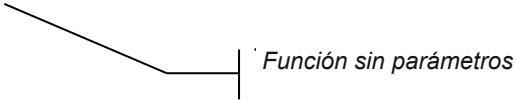
A menudo es necesario enviar información a la función desde el punto de llamada para que complete la tarea asignada. Esto se hace a través de sus parámetros o argumentos.

En el prototipo y en la definición de la función planteamos los **parámetros formales o de diseño**, y cuando invocamos a la función utilizamos **parámetros actuales o de llamada**. Si una función no requiere parámetros de entrada se la define con el tipo void entre paréntesis (o directamente con paréntesis vacíos) y se la invoca con paréntesis vacíos.

```
void funcion_nula( );

void main()
{.....
  funcion_nula( );
  .....
}

void funcion_nula( )
{ ..... }
```



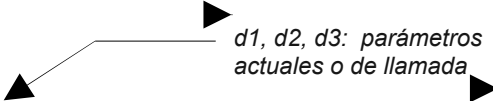
Función sin parámetros

El pasaje o intercambio de información entre parámetros puede hacerse por **valor** o por **referencia**.

Pasaje de parámetros por valor

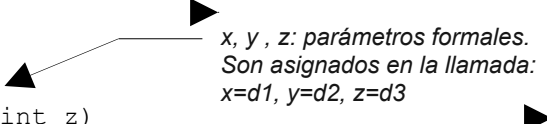
En este caso, al producirse la llamada a la función los parámetros formales son asignados en forma correspondiente con los valores contenidos en los parámetros actuales o de llamada.

```
.....
int main( )
{ .....
  float p = promedio3(d1, d2, d3);
  cout << "Datos:"<<d1<<" "<<d2<<" "<<d3;
  cout << "Promedio:"<<p;
  .....
  return 0;}
```



d1, d2, d3: parámetros actuales o de llamada

```
float promedio3(int x,int y,int z)
{ float w=(x+y+z)/3.0 ;
  return w; }
```



x, y, z: parámetros formales.
Son asignados en la llamada:
x=d1, y=d2, z=d3

Al finalizar las acciones de la función **promedio3()** se devuelve el control a la función principal **main()** retornándose el valor obtenido en **w**. Si en el ejemplo anterior los datos asignados a d1, d2, d3 son 10, 20, 45, la salida a través de los flujos **cout** será:

Datos: 10 20 45
Promedio: 25.00

Pasaje de parámetros por referencia

La referencia consiste en utilizar como parámetro formal una referencia a la posición de memoria del parámetro actual o de llamada. Esto puede hacerse a través del operador referencia `&` o empleando punteros (que veremos más adelante en otra unidad). Con el operador referencia es posible definir alias de una variable y emplear los parámetros actuales o de llamada para obtener resultados.

C++ emplea el operador `&` para realizar esta referencia. Observemos el siguiente ejemplo:

```
int m=10;
int &q = m; // q es definido como alias de m
q++;        // se incrementa q en 1 y también m
cout << m; // se obtiene 11 como salida
```

Es decir que la expresión `&q` permite hacer referencia a la variable `m` a través de otro nombre (`q` es el alias de `m`). Veamos que ocurre en nuestro ejemplo de código para el cálculo del promedio al usar alias para pasaje por referencia:

```
.....
int main( )
{ int d1,d2,d3; float prom=0;
  .....
  cout << "prom antes de llamar a promedio3:"<<prom<<endl;
  promedio3(d1, d2, d3, prom);
  cout << "Datos:"<<d1<<" "<<d2<<" "<<d3;
  cout << "prom despues de llamar a promedio3:"<<prom;
  .....
  return 0;
}
void promedio3(int x,int y,int z,float &p)
{ p=(x+y+z)/3.0 ; }
```

Parámetros actuales o de llamada

Parámetros formales `x, y, z, p` alias de `d1, d2, d3` respectivamente

Modificación del parámetro formal `p` y consecuente modificación de `prom`

Obsérvese que el valor correspondiente a la variable `prom` es **25**, pues al asignarse a su alias `p` el calculo del promedio en el cuerpo de la función se ha modificado su valor original:

```
prom antes de llamar a promedio3: 0
Datos: 10 20 45
prom despues de llamar a promedio3: 25
```

El uso de alias a través del operador `&` permiten a una función devolver otros resultados además del correspondiente a la sentencia `return()`.

Parámetros por defecto

Es posible proponer, en el prototipo de la función, parámetros formales inicializados con valores. Estos valores serán asumidos por defecto en el cuerpo de la función **si no se indican parámetros actuales** para tales argumentos.

```
float promedio3(int x,int y,int z=10)
.....
void main( )
{
    .....
    float p=promedio3(d1, d2);
    .....
    float q=promedio3(d1,d2,d3);
}
```

Parámetro formal con valor 10 por defecto

Llamada a la función con solo 2 parámetros actuales (el tercero se asumirá por defecto)

Llamada a la función con 3 parámetros actuales. Aquí no se empleará el valor por defecto en la función.

La única restricción sintáctica para estos parámetros por defecto, es el hecho de que deben figurar al final (a la derecha) de la lista de parámetros formales.

De acuerdo a esto último, el siguiente prototipo de función C++ sería causa de error en una compilación:

```
float promedio3(int x,int y=5,int z) //ERROR!!
```

Variables locales y globales

Analizamos el concepto de *ámbito de variables* en la unidad 2, al estudiar el tema *declaración de variables*. Ampliando esos conceptos podemos afirmar:

El ámbito de una variable lo constituyen las partes del programa en donde dicha variable es reconocida.

El ámbito de una variable puede ser: un bloque, una función, un archivo.

Si una variable se declara fuera de todo bloque en un programa se define como *variable global*, y es reconocida en cualquier parte del programa.

Una variable declarada dentro de un bloque tiene validez solamente en dicho bloque y en otros bloques más internos o anidados. Estas variables reciben el calificativo de variables **locales** o **automáticas**.

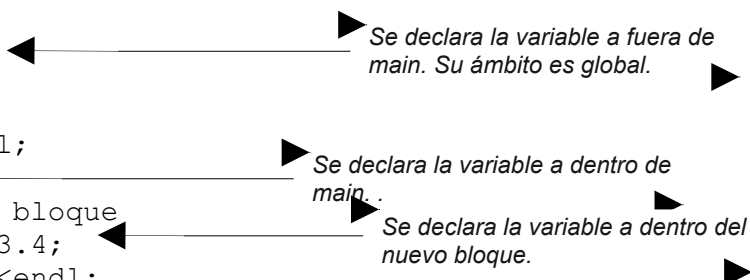
Si se emplea el mismo identificador para una variable local y una global, se anula el acceso a la variable global en el ámbito de validez de la local, pues esta tiene prioridad en su bloque de definición.

Los parámetros formales de las funciones, tienen validez local mientras se ejecuten las acciones de la función.

Finalizado un bloque o una función las variables locales y parámetros formales dejan de existir y su empleo es causa de error.

Observe el siguiente ejemplo que emplea la variable **a** en 3 ámbitos diferentes y la muestra como salida:

```
char a='F';  
int main()  
{  
    cout<<a<<endl;  
    int a=12;  
    { // nuevo bloque  
        float a=3.4;  
        cout<<a<<endl;  
    } // fin del nuevo bloque  
    cout<<a<<endl;  
    return 0;  
}
```



La salida del programa será:

```
F  
12  
3.4
```

No es una buena práctica de programación duplicar identificadores. Resta claridad a la lectura de la lógica del mismo. El ejemplo anterior solo tiene por objeto mostrar las reglas de ámbito de las variables.

En el ejemplo siguiente se muestra un caso donde se pretende emplear una variable en un ámbito en el cual no está definida. Esto producirá un error de compilación.

```
int doble(int x)
{ int a;
  a=2*x;
  return(a);
}
int main()
{cout<<a<<endl;
  int b=25;
  cout<<doble(b)<<endl;
  return 0;
}
```

Se declara la variable a en el ámbito local a la función doble

Causa de error de compilación . El símbolo 'a' no está definido.

Sobrecarga de Funciones

Dos funciones diferentes pueden tener el mismo nombre si el prototipo de sus parámetros es distinto, esto significa que se puede nombrar de la misma manera dos o más funciones si tienen distinta cantidad de parámetros o diferentes tipos de parámetros. Por ejemplo,

```
// Ejemplo de sobrecarga de funciones
#include <iostream>

int dividir (int a, int b)
{ return (a/b); }

float dividir (float a, float b)
{ return (a/b); }

int main ()
{ int x=5,y=2;
  float n=5.0,m=2.0;
  cout << dividir (x,y);
  cout << "\n";
  cout << dividir (n,m);
  return 0;
}
```

La salida del programa será:

2
2.5

En este caso se ha definido dos funciones con el mismo nombre, pero una de ellas acepta parámetros de tipo **int** y la otra acepta parámetros de tipo **float**. El compilador *sabe* cual debe emplear pues los tipos de parámetros son considerados previamente.

Por simplicidad, ambas funciones tienen el mismo código, pero esto no es estrictamente necesario. Se pueden hacer dos funciones con el mismo nombre pero con comportamientos totalmente diferentes.

Obsérvese un caso erróneo de aplicación de sobrecarga en funciones:

```
// Ejemplo erróneo de sobrecarga
int dividir(int,int);

float dividir(int,int);
```

En la primer función se propone una división entera y en la segunda una división entre enteros que arroja un flotante, Pero ambas funciones están sobrecargadas y con el mismo tipo de parámetros. Cuando el programa cliente invoque a división() no podrá discernir a cual de las 2 funciones se refiere la llamada.

Operaciones de entrada y salida en funciones

No es conveniente emplear operaciones de entrada y salida interactiva en funciones. Es mejor operar a través de parámetros y que la entrada y salida la realice el cliente de la función.

Esto es para independizar la función del tipo de entorno en que se ejecutará el programa cliente que la utilizará. Por ejemplo: si en una función incluimos operaciones de salida empleando el modo consola en C++ a través del objeto **cout**, no podremos emplear esta función en un programa C++ que opere en un entorno gráfico (como Windows), donde la entrada y salida se realizan a través de componentes visuales situados en formularios (ventanas).

En cambio, no representa un inconveniente utilizar entrada y salida a dispositivos que almacenan archivos.

Observemos en el ejemplo de abajo la diferencia entre una función que realiza una salida y otra que sólo devuelve el resultado.

La función `volumen_cilindro1()` calcula el volumen de un cilindro y produce una salida con ese resultado

```
void volumen_cilindro1(float radio, float altura)
{   float vol;
    vol= 3.14*radio*radio*altura;
    cout<<"El volumen del cilindro es:"<<vol;
}
```

La función `volumen_cilindro2()` solo devuelve el resultado obtenido al cliente que invoque la función.

```
float volumen_cilindro2(float radio, float altura)
{   float vol;
    vol= 3.14*radio*radio*altura;
    return vo);
}
```

La función `volumen_cilindro2()` puede ser reutilizada en programas cuya entrada y salida se realice a través de componentes visuales de un entorno gráfico. La función `volumen_cilindro1()` solo puede emplearse en programas C++ que operen en modo consola.

Recursividad

La recursividad es una técnica que permite a definir a una función en términos de sí misma. En otras palabras: una función es recursiva cuando se invoca a sí misma.

C++ admite el uso de funciones recursivas; cualquier función C++ puede incluir en su código una invocación a sí misma, a excepción de `main()`.

Para su implementación los compiladores utilizan una pila (stack) de memoria temporal, la cual puede causar una interrupción del programa si se sobrepasa su capacidad (stack overflow).

Como ventaja de esta técnica podemos destacar que permite en algunos casos resolver elegantemente algoritmos complejos. Como desventaja debemos decir que los procedimientos recursivos son menos eficientes –en términos de velocidad de ejecución- que los no recursivos.

Los algoritmos recursivos surgen naturalmente de muchas definiciones que se plantean conceptualmente como recursivas. Obsérvese el caso del factorial de un número: por definición es el producto de dicho número por todos los factores consecutivos y decrecientes a partir de ese número, hasta la unidad:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$.

Por ejemplo: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

Pero el producto $4 \cdot 3 \cdot 2 \cdot 1$ es $4!$

Por lo tanto podemos escribir: $5! = 5 \cdot 4!$

Como vemos en la línea anterior, hemos expresado el factorial en función de sí mismo. Es exactamente lo que podemos plantear algorítmicamente usando funciones C++. La solución recursiva del factorial de un número puede expresarse de la siguiente forma:

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

Obsérvese que en la función recursiva existe una condición ($x==0$) que permite abandonar el proceso recursivo cuando la expresión relacional arroje verdadero; de otro modo el proceso sería infinito.

Condiciones para que una función sea recursiva

Toda función recursiva debe.

- Realizar llamadas a sí misma para efectuar versiones reducidas de la misma tarea.
- Incluir uno o más casos donde la función realice su tarea sin emplear una llamada recursiva, permitiendo detener la secuencia de llamadas (condición de detención o stop)

Analizando el ejemplo del factorial de un número, podemos ver que la expresión **$x * \text{factorial}(x-1)$** corresponde al requisito 1 y la expresión **$x == 0$** al segundo requisito.

Bibliografía:

Ing. Horacio Loyarte. Funciones [UNL-FICH] [2008]