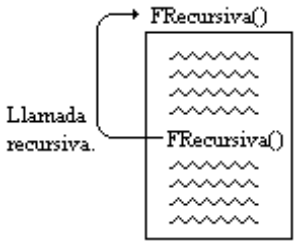
Implementación de Funciones Recursivas

(Ver capítulo 4 del Libro de Cairó y Guardati, 3ª ed., pags. 109 – 136)

La recursión es una potente característica del lenguaje C++ y otros lenguajes como Pascal, Prolog, LISP, Racket, Java, Visual Basic, C#, etc. Básicamente consiste en realizar la ejecución de una función sin que se haya terminado de realizar una o más ejecuciones previas de la misma.

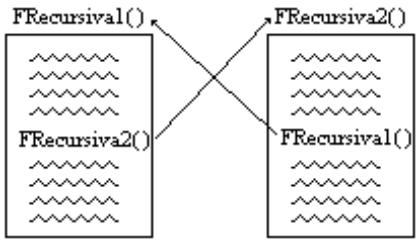
Desde el punto de vista de quién hace la llamada a ejecución de la función recursiva, hay dos tipos de recursión:

a) **Directa**, que es cuando dentro del cuerpo de una función se colocan llamadas para que se invoque a sí misma. Vea la siguiente figura.



De la figura puede notar que la función volverá a iniciar una nueva ejecución, quedando pendiente de finalizar la ejecución en curso. La cantidad de llamadas recursivas que se hagan dejará pendientes de concluir igual cantidad de ejecuciones del procedimiento recursivo.

b) **Indirecta**, que es cuando una función **A** llama a otra función **B**, que realiza nuevas invocaciones a la función **A**. Vea la siguiente figura.



De la figura puede notar que la función **FRecursiva2**, que fue invocada por la función **FRecursiva1**, vuelve a realizar una invocación a esta. La función **FRecursiva1** inicia una nueva ejecución cuando aún está pendiente de finalización la ejecución previa. En la figura se da el ejemplo de la recursión indirecta de dos funciones, pero este tipo de recursión puede involucrar mayor cantidad de funciones.

Cada vez que se realice una llamada recursiva, se volverá a reservar espacio de memoria para contener las variables locales necesarias en la nueva llamada. Aunque las nuevas variable locales tienen los mismos nombres que las variables locales de las ejecuciones previas no concluidas, no se crea ningún conflicto, pues cada conjunto de variables es local de su propio proceso que se encuentra en ejecución.

Muchos problemas se pueden resolver, y de hecho se definen, de manera recursiva. Por ejemplo:

- El factorial de un número **n**: por definición es **n * (n – 1)!**.

- La serie de Fibonacci: el k-ésimo elemento se calcula así: $a_k = a_{k-1} + a_{k-2}$. Exceptuando los dos primeros números de la serie, que son valores fijos.

- Entrar en una sala de belleza o barbería en donde hay un espejo a la izquierda u otro a la derecha de donde nos hemos sentado. Esto genera una serie de imágenes dentro de otras, un efecto que se ve muy interesante.

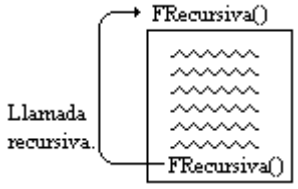
De igual forma, las personas concebimos muchas cosas de manera recursiva. Tiene muchas **ventajas**, a veces se afirma que la recursividad es una forma natural de solución de problemas. Por eso se aconseja utilizarla para mejorar la legibilidad, comprensión del problema y para facilitar la depuración del mismo. Por otro lado, al resolver problemas complejos por métodos recursivos se puede escribe mucho menos código fuente, logrando programas más compactos.

Pero la recursividad también tiene sus **desventajas**. Como ya se dijo anteriormente cada nueva invocación define un nuevo bloque de variables locales que consumirán espacio de memoria. También crece la pila (stack) de direcciones de retorno de subrutina. Estos dos aspectos pueden llevar al agotamiento de la memoria de la computadora. Otra desventaja es el incremento en el tiempo de ejecución debido a la pérdida de tiempo en la realización de la llamada, en la realización de nuevas copias en RAM para variables locales y en el almacenamiento y recuperación de la dirección de retorno de subrutina.

Cuando un programador utiliza recursión debe tener mucho cuidado en establecer mecanismos de control. De lo contrario, una pérdida de control en una función recursiva puede llevar a que el problema nunca se resuelva y/o al agotamiento de la memora de la computadora, lo que llevaría a una terminación fatal del programa. Para evitar el problema de la pérdida de control en las llamadas recursivas ha de implementarse dentro de la función una condición de terminación que sea efectiva, es decir, que con seguridad se sepa que en algún momento será alcanzada. A esto, algunos autores le llaman *caso trivial*, en lenguajes declarativos, o *paso básico*, en lenguajes imperativos. Por el contrario, a la parte de la función que propicia la recursividad le llaman *caso general*, en lenguajes declarativos, y *paso recursivo*, en lenguajes imperativos. Siempre que se analice un problema de forma recursiva será necesario determinar una o más circunstancias para ambas situaciones.

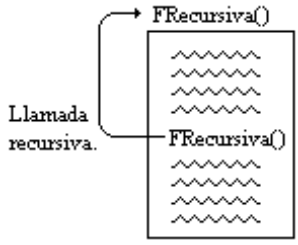
Desde el punto de la ubicación de la llamada recursiva dentro del cuerpo de la función, se conocen dos tipos de recursión:

a) Recursión de **cola**: cuando la llamada recursiva es la última instrucción del cuerpo de la función.



Este tipo de recursión suele ser comparado con instrucciones iterativas, ya que en estas, el bloque se termina por completo antes de iniciar una nueva iteración.

b) Recursión por **posposición**: cuando a continuación de la llamada recursiva existen una o más instrucciones de programa, las cuales se ejecutarán luego de que se regrese de la llamada recursiva.



Usted puede utilizar recursividad para:

a) Resolver problemas que convergen al valor buscado y desplegarlo en la última llamada.

b) Generar la serie de resultados necesarios y desplegarlos en forma inversa, a medida que se retorna para concluir las funciones.

c) Generar la serie de resultados necesarios y desplegarlos a medida que estos van siendo generados.

d) Retornar el valor calculado en la última llamada, hasta obtenerlo en la primera invocación de la función.

e) Obtener un valor en la última llamada recursiva, que será utilizado en las llamadas aún pendientes de concluir para completar operaciones pendientes.

f) Etc.

No hay un límite de llamadas recursivas al momento de ejecutar un programa que utilice esta técnica. La cantidad de llamadas recursivas necesarias para resolver un problema dependerá de los valores iniciales de las variables involucradas, de lo complejo de los cálculos, de qué tan pronto se converge a un resultado, etc. Por otro lado, también ha de tomarse en cuenta la cantidad de memoria de trabajo disponible para las aplicaciones que el sistema operativo concede al usuario que corre el programa.

Ejemplos:

1) Realizar un conteo ascendente, de 1 a 10.

2) Modifique el programa anterior para contar hasta N.

3) Realizar un conteo descendente de 10 a 1.

4) Modifique el programa anterior para iniciar en N y terminar en 1.

5) Sumar todos los enteros contenidos en un intervalo [a, b].

6) Utilizar recursión para obtener el resultado del producto de dos número, a y b. Realizarlo sumando a veces b.

7) Resolver el problema del factorial de un número N.

8) Imprimir los primero N términos de la serie de Fibonacci.

9) Calcular en N-ésimo término de fibonacci.