Teoría de la Recursividad
Capítulo 3: Variables locales y recursividad. Recursivos de cola e indirectos

---

El objetivo de este capítulo es el de tratar aspectos de la implementación de C# muy relacionados con la programación de procesos recursivos, esto es, de procedimientos o funciones que se llaman a sí mismos.

El autor parte de la suposición de que el lector esta familiarizado con los conocimientos fundamentales de la programación de funciones o procedimientos no recursivos en C#: reglas para su declaración, su implementación y su utilización en el programa principal; conocimientos precisos sobre el mecanismo de traspaso de parámetros; la definición por el programador de "tipos" especiales de variables, en casos necesarios; y otros aspectos relacionados.

De no ser así, es necesario remitir al lector al estudio y práctica de los temas mencionados, cuyo tratamiento no es apropiado en el presente trabajo, por que lo alejaría demasiado de sus objetivos específicos.

Para ello, el lector interesado puede encontrar una gran variedad de libros de texto o manuales de programación, que tratan los temas mencionados con bastante claridad y abundancia de ejemplos. También se recomienda adquirir suficiente práctica en la aplicación de esas nociones a la resolución de ejercicios en máquina, pues la misma suele revelarnos deficiencias o malas interpretaciones de la teoría, que sólo mediante esa experiencia se pueden subsanar.

1. Variables globales y locales

1.1 Concepto de "variable" en programación.

¿Qué es, en definitiva, una variable en C#?

Una variable es un grupo de caracteres que identifica una porción de la memoria de la computadora. Al conjunto de caracteres se le llama "nombre" de la variable. A nombres de variable diferentes, corresponde siempre áreas de memoria diferentes.

Un nombre de variable es un "identificador". También son identificadores otros nombres que determina el programador, como los de Métodos, Clases y otros.

1.2 Variables globales.

Se denomina así, a las variables declaradas a nivel de Clase, y que pueden ser utilizadas por cualquier método, ya sean estáticos o de instancia.

1.3 Variables locales.

Cuando en una clase se declara un método, el mismo, como sabemos, puede tener su propia declaración de variables (declaración "explícita"). Esas son variables LOCALES a este. También son variables LOCALES las utilizadas para el traspaso de los parámetros necesarios en la implementación del método. (Declaraciones "implícitas").

Una variable local puede tener incluso el MISMO NOMBRE que una variable global, pero a pesar de ello, para C# es OTRA variable. Cuando EN UN METODO se menciona a uno de esos nombres de variable declarados en el mismo, C# entiende que se está haciendo referencia a la variable local, aún en el caso de que haya otra con el mismo nombre declarada en la clase.

Por el contrario, cuando la mención se hace fuera de este, la referencia que se hace afecta en todos los casos a la variable global correspondiente. Pudiera pensarse que las variables locales son apenas variables "auxiliares" o de "segunda categoría", que simplemente ayudan de alguna manera a hacer más sencillo o cómodo el trabajo del programador. En lo que sigue, se tratara de mostrar que, por el contrario, las variables locales están sujetas a características de almacenamiento y reglas de empleo muy propias, que las convierten en elementos muy importantes y característicos en la programación recursiva.

1.4 ¿Qué, sucede cuando un programa llama a un método?

El programa principal, compilado y listo para su ejecución, permanece siempre en memoria mientras no se abandone el programa. No obstante, NO SUCEDE ESO con los procedimientos o funciones y sus variables locales.

Para la ejecución de un programa, C# utiliza una parte de la memoria como zona auxiliar, la PILA o STACK, donde va guardando resultados intermedios u otros elementos parciales que necesita ir completando al ejecutar ciertas instrucciones. Luego de utilizarlos, para ahorrar memoria, los borra (funcionamiento de "pila").

[[Una "pila" es similar a un arreglo, y se emplea en los casos en que las ÚLTIMAS informaciones almacenadas son las PRIMERAS que se va a utilizar. Una variable específica, generalmente llamada "tope", señala hasta qué, posición del "arreglo" hay almacenada información útil e indica exactamente cuál será la próxima información a extraer. Después de utilizada cierta información de la pila, para mantener disponible toda la memoria que sea posible, la información ya utilizada se BORRA de la pila. Este borrado es muy rápido: basta correr "hacia atrás" el valor del tope, dejando así "en puerta" otra información, que será, mientras no haya nuevos agregados, la próxima a utilizar. Cualquier información que quede DESPUES de la posición que indica el tope es ignorada, y el posible agregado de nuevas informaciones útiles se realiza sobrescribiendo esa parte de la memoria. Una información borrada de la pila es irrecuperable.]]

Cuando se llama a un método, C# lo almacena en la pila, incluyendo las declaraciones de variables que el mismo pueda contener. Gráficamente, se puede decir que C# "hace una copia" del método en la pila, y luego, comienza a ejecutarlo.

Si ese procedimiento invoca a una variable, C# busca su dirección en PRIMER LUGAR en la propia zona del stack donde se copió el método. Si la encuentra, ESA es la dirección que toma,

aunque en la zona de la clase hubiera otra variable que tuviera el mismo nombre. Si no la encuentra allí, entonces sí va a la clase a buscarla.

Debido a eso, en un método también se puede hacer referencia a las variables globales (con tal de que no haya otra con su mismo nombre declarada en este).

Cuando termina la ejecución del método, antes aún de que la ejecución retorne al punto siguiente al llamado del mismo, este SE BORRA todo de la pila, incluyendo las variables locales, tanto implícitas como explícitas. Es por eso que, por el contrario, desde el programa principal NO ES POSIBLE ACCEDER a una variable local a un método.

Con respecto al "tiempo de vida" (tener asignado un espacio específico en la memoria), vemos que las variables globales son, por decirlo así, permanentes. En cambio, las variables locales sólo existen entre el comienzo y el final de la EJECUCIÓN del método que las declara. Su "tiempo de vida" puede ser apenas unos instantes.

1.5 El llamado recursivo.

Como nos encaminamos a trabajar con métodos recursivos, es bueno saber cómo se implementan los mismos en C#, y particularmente qué, papel juegan las variables locales en ese caso. Queda claro que un llamado recursivo es un llamado que se hace al procedimiento o función DESDE sí mismo.

En ese caso, las cosas suceden así:

El PRIMER llamado genera en el STACK la copia correspondiente, con la declaración de las variables locales que contenga. El subprograma comienza a ejecutarse, hasta que llega al punto donde se llama a sí mismo. En ese momento, se genera una NUEVA COPIA (copia 2) del procedimiento, que contiene también las variables locales del mismo. Esta nueva copia se añade a la información que ya estaba en la pila en ese instante, por lo que no afecta para nada a la copia 1, que se mantiene intacta en memoria. La ejecución de la copia 1 se "congela" con los valores que tenía en ese momento cada variable local y comienza a ejecutarse la copia 2.

Cuando se encuentra una referencia a una variable local, a la que se accede es a la declarada EN LA PROPIA COPIA. Por ejemplo, si en esa referencia la variable local "x" cambia de valor, lo que se modifica es el valor de la variable con ese nombre declarada en la SEGUNDA copia. Mientras tanto, la variable local "x" perteneciente a la primera copia permanece INALTERADA.

Si se siguen produciendo nuevos llamados recursivos, sucede lo mismo, y se generarán las copias 3, 4,..., N, que por comodidad llamaremos "niveles". Los N niveles están en ese momento en la pila.

Supongamos que en el nivel N ya no se cumplen las condiciones que originan un nuevo llamado recursivo. Entonces, la ejecución de este nivel continúa hasta el fin del subprograma; la N-ésima copia se borra del stack, y la ejecución vuelve (retorna) al nivel anterior (el N-1), en el punto siguiente al llamado al procedimiento. Con eso, el nivel N-1 se REACTIVA. Los valores de las

variables locales son los mismos que habían quedado “congelados” cuando se produjo el llamado recursivo por parte de ese nivel N-1. A su vez, la copia N-1 completa su ejecución y retorna al nivel anterior (se borra la copia N-1), y así sucesivamente, hasta el retorno al punto en que se llamó por primera vez al proceso recursivo.

En todo el proceso, cuando se retorna a un nivel, se reanuda la ejecución correspondiente al mismo, y las variables locales de esa copia mantienen, en el instante inicial de la reactivación, LOS MISMOS VALORES que tenían cuando se hizo el llamado al siguiente nivel (Desde luego, posteriormente pueden ser modificados por la copia en ejecución).

Esta implementación, por mucho que parezca complicada en una primera lectura, es decisiva para que los procesos recursivos puedan funcionar correctamente. Constituye una manera muy práctica de conservar valores que NO hacen falta en otros niveles, pero que son imprescindibles para que el nivel actual pueda reiniciar su trabajo exactamente en las condiciones que estaba cuando se produjo el llamado.

ESTO ULTIMO, por otra parte, SERA EL CRITERIO BASICO para determinar, al diseñar la solución al problema, qué, variables deben ser locales al procedimiento, y cuáles conviene que sean globales.

[[Cabe agregar que todo lo explicado para los llamados recursivos, es valido también para el caso de que un subprograma llame a OTRO subprograma. La “mecánica” es exactamente igual.]]

1.6 Un ejemplo clásico.

A continuación, ilustraremos las ideas anteriores con un ejemplo sencillo y muy utilizado para explicar el “funcionamiento” de la recursividad, la función factorial:

En el capítulo anterior habíamos visto como la misma puede definirse “por recurrencia”, y que la programación recursiva de la función no es más que la “traducción” a C# de las reglas contenidas en aquella.

Recordemos que, para evaluar la función en un número dado, utilizando su definición recurrente, debíamos acudir reiteradamente al MISMO conjunto de reglas, hasta completar la expresión de cálculo necesaria.

La idea de la programación recursiva es totalmente similar:

En lugar de programar el cálculo (o el proceso) de UNA SOLA VEZ (lo que a veces resulta muy largo y muchas veces es sencillamente imposible) se programa el CONJUNTO de todas las reglas que nos llevan al objetivo deseado y que debemos APLICAR REITERADAMENTE hasta llegar al final buscado.

Es muy sencillo emplear la función así definida en el programa principal: la cláusula

```
int result = Fact(4); //  (*)
```

situará en la variable `result` el valor correcto: 24.

Lo que caracteriza a un procedimiento o función recursivos es la existencia de una invocación a si mismo, dentro del código. En este ejemplo, esto sucede en la parte donde se retorna el resultado de un cálculo en el que interviene Fact.

Hay en ello, no obstante, una sutileza: Fact(n-1), es interpretado en como un llamado a la función de ese nombre, en este caso ella misma (Recursivamente).

Si en el aspecto de programación todo ha resultado muy sencillo en este ejemplo, no podemos decir lo mismo del proceso que se desencadena en la computadora luego del llamado (*). Trataremos de detallarlo, para ilustrar los aspectos referidos en los puntos anteriores de este capítulo. Es recomendable mantener a la vista una copia de `Fact`, pues ayudara a la comprensión del proceso.

La invocación `Fact(4)` en el programa principal, hace que se produzca en el stack una copia del código de la función, con sus correspondientes variables locales. Esquemáticamente, representaremos los aspectos esenciales de esta "copia", así:

STACK:

Variables locales	Cálculo inconcluso
Nivel 1:	
n = 4	
return... (en espera)	4*Fact(3); (en espera)
	\
	nuevo llamado
////////////////////// tope //////////////////////////	

Se genera un nuevo llamado y nueva copia:

Nivel 1:	
n = 4	
return ... (en espera)	4*Fact(3); (en espera)
----------	----------
Nivel 2:	
n = 3	
return ...	3*Fact(2); (en espera)
////////////////// tope //////////////////////////////	

El proceso de llamados se continúa, hasta que n llega a tomar valor 0. En ese momento, en el stack tenemos:
STACK:

```
Variables locales              Cálculo inconcluso
---------------------          ---------------------
Nivel 1:
        n = 4
        return ... (en espera)        4*Fact(3);  (en espera)
-------------------------------------------------------------------
Nivel 2:
        n = 3
        return ...                    3*Fact(2);  (en espera)
-------------------------------------------------------------------
Nivel 3:
        n = 2
        return ...                    2*Fact(1);  (en espera)
-------------------------------------------------------------------
Nivel 4:
        n = 1
        return ...                    1*Fact(0);  (en espera)
-------------------------------------------------------------------
Nivel 5:
        n = 0    <--- Para n = 0, la asignación es directa;
        return 1
                ------> Concluye ejecución, y retorna al nivel 4 con el valor 1
        ////////////////// tope ////////////////////////////
```

Con n = 0, no se produce un nuevo llamado, y FACT retorna - con el resultado 1 - al punto de interrupción en el nivel anterior. Se borra el nivel 5.
A partir de aquí, la secesión de estados de la pila es la siguiente:

```
Nivel 1:
        n = 4
        return ... (en espera)        4*Fact(3);  (en espera)
-------------------------------------------------------------------
Nivel 2:
        n = 3
        return ...                    3*Fact(2);  (en espera)
-------------------------------------------------------------------
Nivel 3:
        n = 2
        return ...                    2*Fact(1);  (en espera)
-------------------------------------------------------------------
Nivel 4:
        n = 1
        return 1                      1*1;  (se efectúa)
                ------>   concluye ejecución, y retorna al nivel 3 con el resultado 1
        ////////////////////// tope ////////////////////////
```

Luego:

Nivel 1:
n = 4
return ... (en espera) 4*Fact(3); (en espera)

Nivel 2:
n = 3
return ... 3*Fact(2); (en espera)

Nivel 3:
n = 2
return 2 2*1; (se efectúa)
-------> concluye ejecución y retorna al nivel 2 con el resultado 2
//////////////// tope ////////////////////////////

Un instante después:
Nivel 1:
n = 4
return ... (en espera) 4*Fact(3); (en espera)

Nivel 2:
n = 3
return 6 3*2; (se efectúa)
-------> concluye ejecución y retorna al nivel 1 con el resultado 6
//////////////// tope ////////////////////////////

Por último:
Nivel 1:
n = 4
return 24 4*6 (se efectúa)
-------> concluye ejecución y retorna al programa principal con el resultado 24
//////////////////// tope ////////////////////////

Hasta aquí, una visión de lo que sucede en la computadora cuando se ejecuta un proceso recursivo y el papel que juegan las variables locales.

2. Recursivos de cola e indirectos.

Si prestamos atención al análisis anterior de cómo funcionaba la función factorial nos damos cuenta de una particularidad (que es característica de la recursividad) y es que el nivel k-ésimo no devuelve valor hasta que el nivel siguiente no haya terminado y así sucesivamente hasta llegar a un nivel que ejecutaría un fragmento de código sencillo y comenzarían los retornos.

A estos casos típicos de recursividad, que es directa (que no hace tanteos), se le denominan recursivos de cola.

Para aprender a usar la recursividad hay que tener en cuenta dos leyes fundamentales, todo método recursivo debe tener:

1. Al menos un caso base o bootstrap que especifica una acción para la situación mas simple, que generalmente corresponde a un juego de datos pequeños.
2. Un paso inductivo que relaciona el problema a ser resuelto con una versión más simple que tiende a acercarse al caso base.

Para el factorial el caso base es $n \leq 1$ `return 1;` y el paso inductivo relaciona a n! con n*(n-1)!, que como n es positiva se puede apreciar que n-1 se acerca al caso base.

Otra operación recursiva es la potencia: a^n. La siguiente es una manera no recursiva basada en multiplicar a, n veces:

```
public static int Pow(int n, int n)
{
      int t=1;
      for( int i=1; i<=n; i++)
            t*=a;
      return t;
}
```

Para n = 0 el ciclo no se ejecuta y devuelve t que es igual 1 ($a^0 = 1$). Para $n \geq 1$, t va tomando valores de a, a^2, a^3,..., a^n.

Nota: Hay que enfatizar que cuando un método no es recursivo de dice que es iterativo.

La definición recursiva de a^n es:

$$a^n = \begin{cases} 1 & n = 0 \\ a * a^{(n-1)} & n > 0 \end{cases}$$

El caso base es $a^0 = 1$, y el paso inductivo es $a * a^{(n-1)}$ que tiende al caso base pues como n>0, n-1 tiende a 0.

```
public static int Pow(int a, int n)
{
      return (n==0)? 1 : a*Pow(a,n-1);
}
```

Hay algo que realmente no viene bien con ninguna de las 2 versiones:

¿Realmente usted calcula a^{16} haciendo 15 multiplicaciones?

Probablemente no, usted seguro vera que $a^{16} = (a^8)^2$ y que $a^8 = (a^4)^2$ y así sucesivamente. Esto realmente nos muestra que para calcular a^{16} hacen falta solo unas cuantas multiplicaciones:

```
public static int Pow(int a, int n)
{
      if (n==0) return 1;
      if (n % 2==0)             // si n es par
      {
            int t=Pow(a, n/2);  // a^n = (a(n/2))^2
            return t*t;
      }
      return a*Pow(a,n-1);      // si n es impar
                                // a^n = a*a^(n-1)
}
```

Para valores grandes de n, esta versión hace solamente $\text{Log}_2(n)$ multiplicaciones, sin embargo la anterior realiza n-1.
Ejemplo: n = 64 $\text{Log}_2(64)=6$. Comparando 6 con 63, se ahorran varias operaciones.

Otro problema bueno para discutir es el algoritmo de Euclides para hallar el máximo común divisor (MCD) de dos números enteros (m, n):

Si m = n, el MCD es m o n. En caso contrario remplace el mayor por la diferencia de los dos números y repita el proceso.

Una solución iterativa es:

```
public static int MCD(int m, int n)
{
      while(m!=n)
      {
            if (m>n) m-=n;
            else n-=m;
      }
      return m;
}
```

Una solución recursiva con la misma lógica es:

```
public static int MCD(int m, int n)
{
      if (m==n) return m;
      return (m>n)? MCD(m-n,n) : MCD(m,n-m);
}
```

Ambas soluciones son muy ineficientes si la diferencia entre m y n es muy grande pues causa un gran número de restas. Pero sustracciones repetitivas son una división, por lo que un algoritmo más eficiente seria:

```
public static int MCD(int m, int n)
{
      if (m==n) return m;
      return (m>n)? MCD(n, m % n) : MCD(m, n % m);
}
```

Otro caso de recusivo de cola es el siguiente problema:

Dada una lista de palabras cuyo final viene dado por la cadena FIN imprima la lista en sentido inverso al que se introdujo.

```
public static void LeeCadena()
{
      string s = Console.ReadLine();
      if (s!="FIN") LeeCadena();
      Console.WriteLine(s);
}
```

No solamente un método puede ser recursivo en sí mismo sino que puede serlo indirectamente a través de otros. Por ejemplo P1 y P2 pueden ser recursivos si P1 llama a P2 y P2 llama a P1. Otro ejemplo es si mediante varios métodos se crea una recursión indirecta a través de llamados circulares tales como P1->P2->P3->...->PN->P1.

El siguiente ejemplo es un caso típico de esta forma de recursividad y nos muestra una manera de como se calculan algunas funciones trigonométricas en la computadora.

Vamos a calcular el seno y el coseno empleando las identidades:

$$\text{sen } x = 2 \text{ sen}(x/2) \cos(x/2) \quad \text{y} \quad \cos x = \cos^2(x/2) - \text{sen}^2(x/2)$$

Para hacer de este par de identidades la base de nuestros métodos debemos notar lo siguiente:

- Ambos métodos Sin y Cos se llaman a si mismos y son recursivos.
- Cada método llama al otro por lo que son mutuamente un par recursivo.
- Se necesita un caso base para terminar el proceso.

Los primeros 2 puntos son simples observaciones pero el ultimo es un problema a resolver por lo que nos enfocaremos en ello. Como cada llamado al seno o al coseno generan llamados que usan argumentos que son la mitad del anterior, nos da la idea de que este se hace cada vez menor y menor. Si nosotros conociéramos el valor de sen(x) y cos(x) para ángulos pequeños tendríamos nuestro caso base.

Pues para valores muy pequeños de x se cumple que:

$$\text{sen } x \approx x * (1 - x^2 / 6) \quad \text{y} \quad \cos x \approx 1 - x^2 / 2$$

Cada aproximación se puede calcular usando la serie de Taylor.
¿Entonces cuán pequeña tiene que ser x para que se cumpla esta aproximación? Eso depende de que precisión queremos alcanzar en nuestra respuesta. Después de varios análisis tenemos que para $x<0.02$ la aproximación de cos (x) nos da un valor de 8 lugares decimales muy cercanos al valor real. Por lo que usaremos este valor como caso base y lo llamaremos epsilon.

```
static double epsilon=0.02;

public static double Sen(double x)
{
      return (Math.Abs(x)<epsilon)? x*(1-x*x/6) : 2*Sen(x/2)*Cos(x/2);
}

public static double Cos(double x)
{
      if (Math.Abs(x)<epsilon) return 1-x*x/2;
      double c=Cos(x/2);
      double s=Sen(x/2);
      return c*c-s*s;
}
```

Esta manera de definir la recursividad nos hace pensar en algo como:
¿Quién surgió primero, el huevo o la gallina?

Ejercicios Propuestos

1. Escriba un método iterativo para la versión más eficiente de Pow.

2. Las versiones anteriores de Pow trabajan solamente con $n \geq 0$. Escriba un método recursivo lo mas eficiente posible que sea $n \in Z$.

3. La función recursiva más famosa se debe a Wilhelm Ackermann, quien la inventó para probar el poder recursivo de los compiladores.
   Su definición es:

   $A(0, n) = n + 1$
   $A(m, 0) = A(m-1, 1)$
   $A(m, n) = A(m-1, A(m, n-1))$

   Nota: Debe mantener este orden de las operaciones.

   Escriba un método recursivo que implemente esta definición y póngalo en una aplicación de consola para corroborar los valores de la siguiente tabla:

m ↓ \ n →	0	1	2	3	4	5
0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	13	29	61	125	253
4	13					