Tema 11.5

Descargar estos apuntes en pdf o html

Índice

Recursividad

Definiciones

- **Llamada recursiva**: se produce cuando un método se llama a si mismo para realizar el proceso u obtener el valor de retorno.
- **Recursividad**: la capacidad de un módulo de llamarse a si mismo.
- **Algoritmo recursivo**: un algoritmo resuelto mediante recursividad.

> 📌 **Nota:** Cualquier **algoritmo recursivo** tiene un equivalente **iterativo**.

Ventajas

- Existen problemas cuya solución natural es claramente recursiva con lo que el algoritmo resulta **muy simple y claro frente** a la solución iterativa. Por ejemplo ...
  - El recorrido de colecciones tipo árbol o grafo.
  - Algoritmos que generan árboles en la búsqueda de una solución como el backtracking.

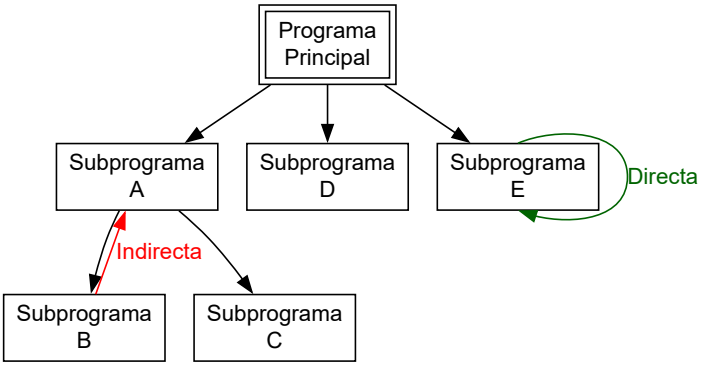
Desventajas

- Generalmente es menos eficiente que la solución iterativa. Aunque se puede aliviar con una técnica denominada **memoization**.
- Consume muchos más recursos.
- Puede llevar al desbordamiento de la pila de llamadas o (**Stack Overflow**).

Tipos de recursividad

- **Recursividad directa**: se produce cuando un subprograma o módulo se llama a si mismo.
- **Recursividad indirecta**: se produce cuando un subprograma o módulo **A** llama a otro **B** que acaba llamando de nuevo a **A**.

> 💀 ✋ Esta última, no deberíamos usarla **en ningún caso**.



Diseño de un algoritmo recursivo

Necesitaremos definir o buscar dos cosas:

1. **El caso general**
   - Deberemos subdividir el problema en **versiones menores de si mismo**.
     - Pueden darse varios casos generales.
     - Todos **tenderán** a un determinado caso base.
   - Deberemos confiar en que la llamada recursiva va a hacer su trabajo y va a solucionar o devolver parte del problema.
2. **Al menos una caso base**
   - Condición de salida de la recursividad o llamada recursiva, donde el caso general ya no se puede subdividir más y donde dejamos de hacer llamadas recursivas.
   - Pueden darse varios casos base.
   - 💀 Si no existe el caso base no saldríamos de la recursividad y se produciría un **Stack Overflow**.

Punto de vista desde el lenguaje C#

- Tendremos una **pila de llamadas** al mismo método.
- En cada llamada se apilan o *'guardan'* y no se modifican los objetos locales y parámetros del método llamador.
- Al recuperar el control, el valor de los objetos apilados es recuperado.
- Es interesante usar el depurador para ver como funcionan este tipo de programas.

Ejemplo: Veámoslo a través de un problema que ya hemos solucionado de forma iterativa, pero que tiene una solución matemática natural recursiva.

Vamos a implementar una función recursiva para calcular el factorial de un número entero. Recordemos que el factorial se simboliza como $n!$, se lee como '*n factorial*', y la definición es: $n! = n \times (n-1) \times (n-2) \times ... \times 1$. Además, no se puede calcular el factorial de números negativos, y el factorial de 0 es 1, de modo que una función bien hecha para cálculo de factoriales debería incluir un control para esos casos.

Pasos

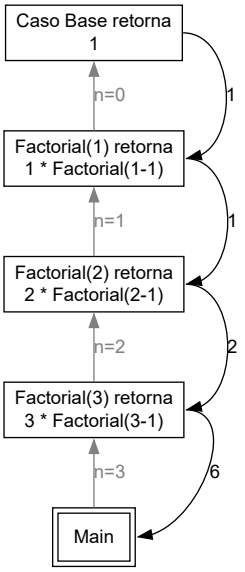
1. La función debería tener un interfaz descriptivo, tal y como lo tiene la iterativa, esto es: `ulong Factorial(ulong n)`
2. Identificaremos el caso base, en este caso es **si n = 0 → !n = 1**

3. Identificaremos el caso general, esto es, cómo descomponer el problema de tal manera que tienda al caso base. En este caso será **n! = n X (n - 1)!**

```
static ulong Factorial(ulong n)
{
    ulong valor;

    if (n == 0)
        valor = 1;                      // Caso Base
    else
        valor = n * Factorial(n - 1);   // Caso General
    return valor;
}
```

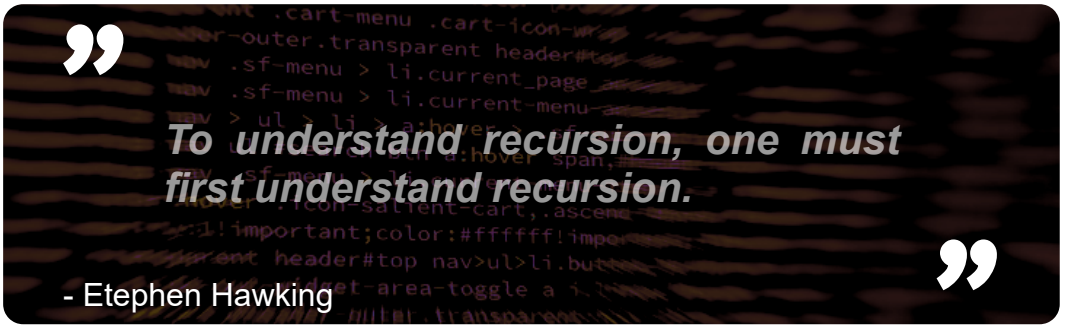
La variable local `valor` y el parámetro formal `n` será diferente en cada llamada y se liberarán de memoria en cuanto se **desapilen** la llamadas a la función.



Si ejecutásemos el siguiente código:

```
static void Main()
{
    Console.WriteLine(Factorial(3));
}
```

Se generaría la pila de llamadas del esquema de la derecha.



Usando funciones Lambda (λ)

Para expresar la función factorial anterior con expresiones lambda haríamos algo cómo ...

```
Func<int, int> factorial = n => (n > 0) ? n * factorial(n - 1) : 1;
```

Pero como aún no hemos completado la instrucción donde **definimos** el identificador del delegado factorial **nos dice que aún no existe al usarlo**.

Existen varias soluciones al problema '*elegantes*' desde el punto de vista **funcional y matemático**. Pero una solución muy simple sería definir primero el **identificador** que referencia al delegado asignándole `default`.

Cómo ya está definido el identificador y es una referencia a un delegado. Podremos **reasignarlo** haciendo una clausura dentro del mismo la referencia.

```
Func<int, int> factorial = default;
factorial = n => (n > 0) ? n * factorial(n - 1) : 1;
Console.WriteLine(factorial(3));
```

> 🚀 **Ampliación:**
>
> Fíjate que la implementación en otros lenguajes **débilmente tipados** es '*más simple*'...
>
> 1. **JavaScript**
>
> ```
> let factorial = n => (n > 0) ? n * factorial(n - 1) : 1;
> console.log(factorial(3))
> ```
>
> 2. **Python**
>
> ```
> factorial = lambda n : n * factorial(n - 1) if (n > 0) else 1
> print(factorial(3))
> ```

Ejemplos de recursividad

Ejemplo 1: Vamos ha hacer una función recursiva que calcule la potencia de un numero.

- **Interfaz**: `double Pow(double b, int e)` ... Debería ser igual que el de la iterativa.
- **Caso Base**: $b^0 \rightarrow 1$
- **Caso General**: $b^e \rightarrow b \times b^{e-1}$
- Propuesta de solución

```
static double Pow(double b, int e)
{
    double p;
    if (e <= 0)
        // Caso Base
        p = 1d;
    else
        // Caso General
        p = b * Pow(b, e - 1);
    return p;
}
```

- Propuesta con λ

```
Func<double, int, double> pow = default;
pow = (b, e) => e > 0 ? b * pow(b, e - 1) : 1d;
```

Ejemplo 2: Vamos a implementar una función que dado un número decimal de entrada, me devuelva una cadena con su representación en binario (base 2). En este ejemplo es un poco más difícil de ver el caso base y general.

- **Interfaz**: `string Binario(int valorDecimal)`
- **Caso Base**: Habrá dos casos base que son los valores que no necesitan conversión por ser menores que 2 que que són los dos posibles cocientes que concatenaré en primer lugar ...
  i. valorDecimal = 0 → cadenaBinaria = "0"
  ii. valorDecimal = 1 → cadenaBinaria = "1"
- **Caso General**: Binario(valorDecimal / 2) + "Resto de dividir valorDecimal por 2"

> 📌 **Nota:** Como he concatenar los restos de del último al primero, por eso voy concatenando los restos por la derecha.

- Propuesta de solución

```
static string Binario(int valorDecimal)
{
    string cadenaBinaria;
    switch (valorDecimal)
    {
        case 0:
            cadenaBinaria = "0";
            break;
        case 1:
            cadenaBinaria = "1";
            break;
        default:
            cadenaBinaria = Binario(valorDecimal / 2) + $"{valorDecimal % 2}";
            break;
    }
    return cadenaBinaria;
}
```

- Propuesta con λ

```
Func<int, string> binario = default;
binario = d => d switch
{
    0 => "0",
    1 => "1",
    _ => binario(d / 2) + $"{d % 2}",
};
Console.WriteLine(binario(44)); // Mostrará "101100"

// Algo más ofuscada podría ser la siguiente expresión.
binario = d => d > 1 ? binario(d / 2) + $"{d % 2}" : (d == 0 ? "0" : "1");
```

Ejemplo 3: Dada una cadena de entrada, implementar una función recursiva que me devuelva su inversa.

- **Interfaz**: `string Invierte(string t)`
- **Caso Base**: Si la longitud de la cadena de entrada es 1 su inversa es ella misma.
- **Caso General**: Invierte(t menos la primera letra) + primera letra de t
- Propuesta de solución

```
static string Invierte(string t)
{
    string invertida;
    if (t.Length > 1)
        invertida = Invierte(t.Substring(1, t.Length - 1)) + t[0];
    else
        invertida = t;
    return invertida;
}
```

- Propuesta con λ

```
Func<string, string> invierte = default;
invierte = t => t.Length > 1 ? invierte(t[1..]) + t[0] : t;
```

> 📌 **Nota**: `t[1..]` es un '*syntactic sugar*' de `t.Substring(1, t.Length - 1)`

> 📣 **Resumen:**
>
> Existen problemas típicos mucho más complejos a resolver mediante recursividad pero quedan fuera de lo que pretende este tema que es una mera introducción al concepto.