

Recursividad

La recursividad es una característica de los lenguajes de programación por la cual un procedimiento o función puede invocarse de nuevo a sí mismo como parte de los tratamientos o cálculos que necesite para realizar su tarea. Dicho de otro modo, un módulo recursivo es aquel que entre la lista de las instrucciones que lo forman se encuentra una llamada a sí mismo, directa o indirectamente.

Recursividad

En términos generales, se puede decir que la *recursividad* es la definición de un objeto de forma que lo que se define está incluido en la definición. Por ejemplo, si decimos: “un rebaño está compuesto por una oveja y el resto del rebaño”, estoy haciendo una definición recursiva. De igual forma podemos decir que un número está compuesto por un dígito y el resto del número (número = número + dígito), y que un número puede tener sólo un dígito (número = dígito).

Toda definición recursiva debe estar compuesta de dos partes fundamentales: una parte que utilice el objeto que estamos definiendo (número = número + dígito), llamado caso general o recursivo, y otra que no lo utilice (número = dígito), llamado caso base; esta última será la que nos permita abandonar la recursividad.

Si extrapolamos estas definiciones al concepto de algoritmo, podremos afirmar que un módulo será recursivo si se llama a sí mismo directa o indirectamente. Si esta fuera la única definición, estaríamos así infinitamente, por tanto, debe existir una condición de finalización, esto es, el caso base.

El ejemplo típico de aplicación de la recursividad es el cálculo del factorial de un número entero. Recordemos que el factorial de un número entero ($n!$) se calcula:

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

y que además $0!$ es 1.

La solución al cálculo del factorial, en base al cálculo propuesto, podría ser:

```
#include <iostream>
using namespace std;

int factorial(int );

int main(){
    int n, resultado;

    cout << "Introduce un número: ";
    cin >> n;
    resultado = factorial(n);
    cout << n << "!= " << resultado;
}
```

```
int factorial(int n){
    int i, res;

    res = 1;
    for (i = 1; i <= n; i++)
        res = res * i;
    return(res);
}
```

Vemos que esta solución no es recursiva, se trata de una solución iterativa, es decir, mediante el uso de un bucle.

Sin embargo, si utilizamos los conceptos de caso base y caso general o recursivo podríamos describir el factorial del siguiente modo:

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

Según esta definición resulta muy intuitivo resolver la función factorial empleando recursividad, tal y como se muestra en el siguiente fragmento de código.

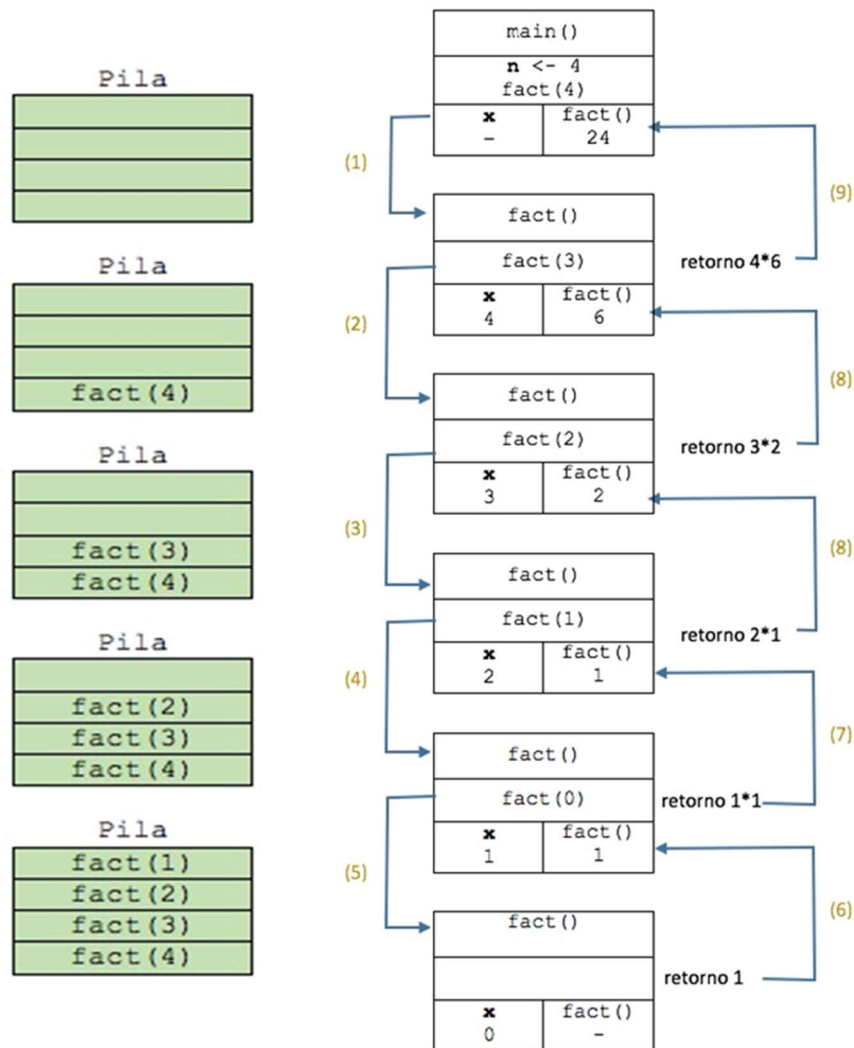
```
int fact (int x){
    int func;

    [1]  if (x == 0)
    [2]      func = 1;                /* caso base */
        else
    [3]      func = x * fact(x-1);    /* caso recursivo */

    return (func);
}
```

En la función anterior, en el caso de que el argumento utilizado en la llamada sea 0, ésta devuelve 1, y caso contrario se calcula un producto que involucra a la variable x y una nueva llamada a la función cuyo argumento es menor en una unidad (x-1).

El funcionamiento de un módulo recursiva se realiza almacenando las llamadas pendientes, con sus argumentos, en la **pila en tiempo de ejecución**. Vamos a ilustrar el proceso suponiendo que desde `main` se realiza la llamada `fact(4)`. En la figura siguiente se muestra cómo funcionaría si desde el `main`, se llama a `fact(4)`. El seguimiento (*traza*) de un algoritmo es el mejor método para entender y comprobar su funcionamiento.



La recursividad es una técnica muy útil para resolver problemas, pero en ocasiones se cometen errores difíciles de detectar, como el que se muestra en el siguiente ejemplo.

```
int mal (int n){
    int malo;

    [1] if (n == 0)
    [2]     malo = 0;
    else
    [3]     malo = mal(n / 3 + 1) + n - 1;

    return (malo);
}
```

Este es un ejemplo de recursividad infinita. Imaginemos que llamamos a la función mal con $n=1$. En este caso, se ejecutará la instrucción [3]. Cuando se calcula el valor de la expresión $(n / 3 + 1)$, resulta que es igual a 1 con lo que se efectúa una llamada a mal(1) nuevamente y así indefinidamente. Así, el computador hará llamadas repetidas a mal(1) tratando de hallar su valor. Finalmente, el sistema agotará la memoria, y el programa abortará. No funcionará para ningún caso, ya que sea cual sea el valor de n , a medida que la ejecución avance, llegará un momento en el que n valdrá 1 y no se alcanzará el caso base, por tanto, no finalizará. Por ejemplo, mal(2) llama a mal(1). Así, tampoco mal(2) puede evaluarse. Además, mal(3), mal(4) y mal(5) hacen llamadas a mal(2). Y como mal(2) no es evaluable, ninguno de esos valores lo es. De hecho, este programa no sirve para ningún valor n , excepto para 0.

Estas consideraciones llevan a las dos primeras reglas fundamentales de la recursión:

1. *Casos base*. Siempre se deben tener algunos casos base que se puedan resolver sin recursión.
2. *Caso recursivo*. Para los casos que deben resolverse recursivamente, la llamada recursiva siempre debe tender a un caso base.

Existen problemas que tienen una solución más sencilla mediante llamadas recursivas dada su propia definición. Sin embargo, necesitan mayor cantidad de memoria para su ejecución y son más lentos en su funcionamiento, ya que en cada invocación del módulo recursivo se van ocupando distintas zonas de memoria. Así pues, en algoritmos en que es crítica la velocidad de ejecución se tiende a evitar las soluciones recursivas.

Aunque los ejemplos anteriores trabajan con funciones, eso no quiere decir que la recursividad sólo se pueda aplicar con funciones. El siguiente ejemplo escribe al revés desde un número introducido por teclado hasta el 1. Por ejemplo si se introduce 5, escribirá 5 4 3 2 1.

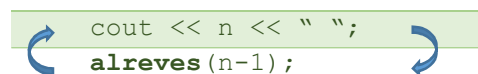
```
#include <iostream>
using namespace std;

void alreves (int n){
    if (n == 1)                // caso base
        cout << n << endl;
    else{                      // caso recursivo
        cout << n << " ";
        alreves (n-1);
    }
}

int main(){
    int num;

    cout << "Introduce un número:";
    cin >> num;
    alreves (num);
    cout << endl;
    return 0;
}
```

Podemos apreciar que sólo realizando un intercambio de orden entre dos instrucciones conseguimos que el orden de escritura de los números sea justamente el inverso. Esto es, cambiando el orden de las instrucciones



```
cout << n << " ";
alreves (n-1);
```

Ante una entrada del número 5, escribirá por pantalla: 1 2 3 4 5.

- ✓ ¿Qué muestra en pantalla este fragmento de código suponiendo que n vale 9?

```
void rec (int n){
    if (n>1){
        if (n%2==0){
            cout << n << ", ";
            rec (n-2);
```

```

    }
    else
        rec(n-1);
}
}

```

Se trata de una rutina recursiva que muestra por pantalla los números pares desde n hasta el 2. Por tanto, escribe 8, 6, 4, 2,

- ✓ ¿Qué valor devuelve esta función suponiendo que n vale 287?

```

int cuenta (int n){
    int cont;
    if (n==0)
        cont =0;
    else
        cont =1+cuenta(n/10);

    return (cont);
}

```

Se trata de una rutina recursiva que cuenta el número de cifras que tiene un número. Por tanto, devuelve 3

- ✓ ¿Qué muestra en pantalla este módulo suponiendo que n vale 1269?

```

void rec (int n){
    if (n>0){
        if ((n%10)%2!=0){
            rec(n/10);
            cout << n%10 <<" ";
        }
        else
            rec(n/10);
    }
}

```

Se trata de una rutina recursiva que escribe en pantalla las cifras impares de un número. Por tanto, escribe 1, 9,

Tipos de recursividad.

La recursividad puede ser de dos tipos:

- *recursividad directa (simple)*: un subprograma se llama a sí mismo una o más veces directamente.

- *recursividad indirecta (mutua)*: un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A.

Anteriormente ya se ha visto algún ejemplo de recursividad directa, veamos ahora un ejemplo de recursividad indirecta, en el que se muestra si un número es par o impar sin realizar ninguna operación matemática, tan solo utilizando dos funciones recursivas, en las que una llama a la otra:

```
#include <iostream>
using namespace std;

bool esPar(int );
bool esImpar(int );

bool esPar(int n) {
    bool par;

    if (n == 0)
        par = true;
    else
        par = esImpar(n-1);
    return(par);
}

bool esImpar(int n) {
    bool impar;

    if (n==0)
        impar = false;
    else
        impar = esPar(n-1);
    return(impar);
}

int main() {
    int num;

    cout << "Introduce un número: ";
    cin >> num;

    if (esPar(num) == true)
        cout << "El número es par" << endl;
    else
        cout << "El número es impar" << endl;
    return 0;
}
```