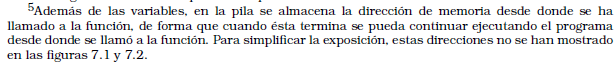
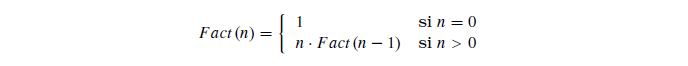
([1], pág. 84)



3.1.7 Definición de recursividad, funciones recursivas

([1], pág. 85) Cada función al comenzar a ejecutarse crea en la pila un espacio para sus variables, por esta razón, no hay ningún problema para que una función llame a otra función. “Es más, tampoco hay ningún problema si una función se llama a sí misma. Se dice entonces que esa función es una **función recursiva**. Sin duda se preguntará ahora: ¿y para qué sirve una función que se llame a sí misma? Pues está claro: para programar algoritmos recursivos. El ejemplo más sencillo es el factorial de un número, que puede definirse mediante un algoritmo recursivo como:



La implantación en C de este algoritmo recursivo es bien sencilla:

long int Factorial(long int n)

{

long int fact;

if(n == 0){

fact = 1;

}else{

fact = n \* Factorial(n-1);

}

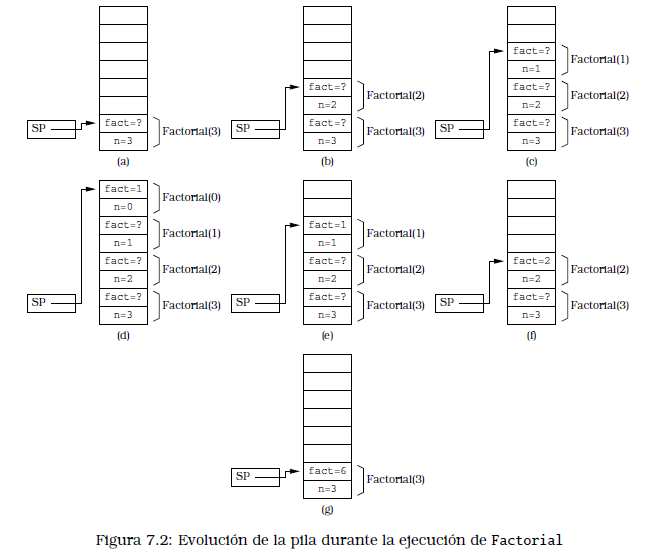
return fact;

}

Como puede observarse, la implantación en C es directa a partir de la definición matemática. Lo único a destacar es que en la línea 8 se llama a la función Factorial, sin que ello implique nada especial. Para el ordenador es lo mismo llamar a una función con nombre distinto que con el mismo nombre de la función que hace la llamada. Para aclarar el proceso, veamos qué ocurre cuando se llama a la función para calcular el factorial de 3, por ejemplo haciendo:

res = Factorial(3);

El programa saltará entonces a la función Factorial, pasándole como argumento un 3. La función creará espacio en la pila para almacenar el argumento n y su variable local fact, que de momento estará sin inicializar. La situación de la pila será la mostrada en la figura 7.2(a). Cuando se ejecute la línea 8 de la función, se evaluará la expresión que consiste en multiplicar n (que vale 3) por el resultado de ejecutar la función Factorial con 2 como argumento. Por tanto, la CPU volverá a llamar a la función Factorial, creándose ahora en la pila otras dos variables para la nueva instancia de la función, una para su argumento, que ahora vale 2, y otra para su variable local, que al igual que antes estará sin inicializar. La situación de la pila será la mostrada en la figura 7.2(b). El proceso se repetirá hasta que se llame a Factorial con un argumento igual a 0, tal como se muestra en las figuras 7.2(c) y 7.2(d). En esta última llamada –Factorial(0)–, se ejecutará la línea 6 y se asignará el resultado 1 a la variable fact de esta instancia de la función Factorial, tal como se ha mostrado en la figura 7.2(d).Este valor se retornará a quien la había llamado, que no era más que otra instancia de ella misma pero con argumento 1. Por tanto, se terminará de ejecutar la línea 8 de esa instancia de la función, obteniéndose como resultado un 1 que se asigna a la variable fact de esta instancia de la función. La situación de la pila en ese momento será la mostrada en la figura 7.2(e). Factorial(1) terminará de ejecutarse, devolviendo un 1 a Factorial(2). El proceso se repetirá, finalizándose la ejecución de la línea 8 de las instancias en curso de Factorial, obteniéndose el valor de la variable fact y retornándola. La evolución de la pila se muestra en las figuras 7.2(f) y 7.2(g).



Conviene destacar dos aspectos de este ejemplo:

* Mientras se están evaluando las sucesivas llamadas, existen múltiples instancias de la misma función ejecutándose “a la vez”. El código a ejecutar es el mismo, pero los argumentos son distintos.
* El proceso necesita gran cantidad de memoria para almacenar en la pila los argumentos y las variables locales. Además, las sucesivas llamadas recursivas a la función consumen tiempo de CPU, por lo que el proceso es **ineficiente**.
* Todas las funciones recursivas tienen que tener una condición de salida (en este ejemplo cuando n vale 0) y debe estar garantizado que la llamada recursiva siempre se realiza con distintos argumentos. Si no se cumplen estas condiciones, el algoritmo será infinito y el programa sólo terminará cuando se agote la memoria del ordenador.

7.9.1. Recursividad frente a iteratividad

Como se acaba de mencionar, la ejecución de funciones recursivas es ineficiente. Esto hace que sea casi siempre mejor implantar un algoritmo de forma iterativa que de forma recursiva, siempre y cuando dicho algoritmo exista. Por ejemplo, en el caso del factorial, puede usarse en lugar del algoritmo anterior, el siguiente algoritmo iterativo:



Que puede implantarse en C de la siguiente manera:

long int FactorialIte(long int n)

{

long int fact; /\* valores parciales de factorial \*/

int i; /\* Índice para el cálculo del factorial \*/

fact = 1;

for(i=1; i<=n; i++){

fact = fact \* i;

}

return fact;

}

Para ilustrar este punto, se han ejecutado ambas versiones de la función para calcular el factorial de 13 en un ordenador Pentium 4 a 2.6 GHz y mientras que la versión iterativa tarda en ejecutarse 1,5 μs, la versión recursiva tarda 4 μs.6



No obstante existen algoritmos que son puramente recursivos y no disponen de versiones iterativas. Es por ello que todos los lenguajes de programación modernos soportan la recursividad. Ejemplos de este tipo de algoritmos son la ordenación mediante QuickSort, la evaluación de expresiones matemáticas, la gestión de árboles de búsqueda o la resolución del juego de las torres de Hanoi.

A modo de conclusión, los algoritmos que se puedan implementar fácilmente de manera iterativa o recursiva se deben programar de manera iterativa por eficiencia. Sin embargo, los algoritmos que típicamente se definen de manera recursiva (como los ejemplos anteriores) deben programarse de manera recursiva, porque aunque en algunos casos fuese posible inventarse algún truco que permita programarlos sin recursividad, el código resultante es muy enrevesado, difícil de entender y la posibilidad

de obtener mayor eficiencia disminuye.”

([1], pág. 122)

9.4.1. Retorno de más de un valor por parte de una función

“Como se ha visto anteriormente una función sólo puede devolver un valor mediante la sentencia return. Sin embargo en muchas ocasiones es deseable que una función devuelva más de un valor. En estos casos la técnica usada consiste en pasarle a la función la dirección de las variables donde queremos que nos devuelva sus resultados; de modo que la función pueda modificar directamente el valor de dichas variables. Por ejemplo, si queremos realizar una función que devuelva la suma y el producto de dos valores, dicha función se escribiría:

void SumaProd(double \*psuma , double \*pprod , double dato1 , double dato2)

{

\*psuma = dato1 + dato2;

\*pprod = dato1 \* dato2;

}

y la manera de llamarla para que devuelva la suma y el producto de 2 y 3 en las variables sum y prod sería:

double sum;

double prod;

SumaProd(&sum, &prod , 2.0, 3.0);

Por convenio, en estos casos se suelen colocar en la lista de argumentos las variables en las que devuelve la función sus resultados en primer lugar.

Si la función sólo devolviese un valor, típicamente se habría declarado de tipo double y se devolvería el resultado mediante un return. Sin embargo, al querer realizar dos cálculos dentro de la función nos vemos obligados a utilizar punteros y declaramos la función como void (puestos a usar punteros devolvemos los dos resultados por medio de punteros). Nótese que para lamar a una función que trabaje con punteros, en el main no se declaran punteros sino variables normales y luego se pasan sus direcciones (con el operador &) a la función.

([1], pág. 122)

**9.5. Punteros y vectores**

Una de las aplicaciones más frecuentes de los punteros es el manejo de vectores y cadenas de caracteres. Como se recordará todos los elementos de un vector se almacenan en posiciones de memoria consecutivas y por lo tanto basta conocer la posición de memoria del primer elemento para poder recorrer todo el vector con un puntero. El siguiente ejemplo inicializa un vector de double por el método “normal” y luego escribe su contenido con la ayuda de un puntero.

/\*

Programa de punteros y vectores.

Descripción: Este programa inicializa un vector y

luego escribe sus valores.

Versión: 1.0

\*/

#include <stdio.h>

#define N 10

int main(void)

{

double a[N]; /\* vector \*/

int i; /\* contador \*/

double \*pd; /\* puntero a double \*/

/\*\*\* Inicialización \*\*\*/

for(i=0; i<N; i++) {

a[i] = 7.8\*i;

}

/\*\*\* Imprimir valores \*\*\*/

pd = &a[0]; /\* Apunto al primer elemento del vector \*/

for(i=0; i<N; i++) {

printf(" %f\n", \*pd); /\* \*pd es de tipo double \*/

pd++; /\* pd pasa a apuntar al siguiente elemento \*/

}

return 0;

}

Hay que tener en cuenta que cuando termina el programa el puntero pd queda apuntado a un lugar no válido, ya que queda fuera del vector a. Supongamos ahora que queremos copiar el vector a en el vector b de manera que sea su inverso; es decir, si a vale [1, 2, 3, 4] queremos que b valga [4, 3, 2, 1]. Generar

b a partir de a es muy sencillo utilizando dos punteros pa y pb, uno creciente y otro decreciente:

pa=&a[0]; /\* pa apunta al primer elemento del vector a \*/

pb=&b[N-1]; /\* pb apunta al último elemento del vector b \*/

for(i=0; i<N; i++) {

\*pb=\*pa; /\* copio un elemento \*/

pa++; /\* avanzo un elemento \*/

pb--; /\* retrocedo un elemento \*/

}

Con mucha frecuencia se utilizan varios punteros sobre el mismo vector para copiar datos o hacer manipulaciones sin tener que andar con la complicación de manejar varios índices. Los ejemplos más típicos son las transformaciones de cadenas de caracteres.

([1], pág. 123)

9.5.1. Equivalencia de punteros y vectores

Cuando se define un vector, tal como se ha visto en el ejemplo anterior:

”

REFERENCIAS

[1] Muñoz Frías, J.D., & Palacios Hielscher, R. (2006). FUNDAMENTOS DE PROGRAMACIÓN UTILIZANDO EL LENGUAJE C (Núm. de edición no disponible). PUBLICACIONES DE LA UNIVERSIDAD PONTIFICIA COMILLAS MADRID.