5. Recursividad en Pascal

En Pascal, a un procedimiento o función le es permitido no sólo invocar a otro procedimiento o función, sino también invocarse a sí mismo. Una invocación de éste tipo se dice que es *recursiva*.

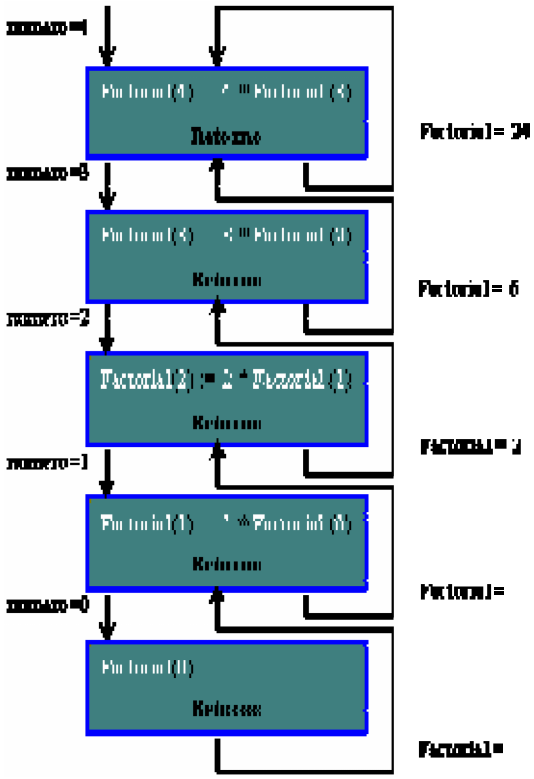
La *función recursiva* más utilizada como ejemplo es la que calcula el factorial de un número entero no negativo, partiendo de las siguientes definiciones :

factorial (0) = 1
factorial (n) = n*factorial(n-1), para n>0

La función, escrita en Pascal, queda de la siguiente manera :

```
function factorial(numero:integer):integer;
begin
  if numero = 0 then
    factorial := 1
  else
    factorial := numero * factorial(numero-1)
end;
```

Si numero = 4, la función realiza los siguientes pasos :



1. factorial(4) := 4 * factorial(3) Se invoca a si misma y crea una segunda variable cuyo nombre es numero y su valor es igual a 3.
2. factorial(3) := 3 * factorial(2) Se invoca a si misma y crea una tercera variable cuyo nombre es numero y su valor es igual a 2.
3. factorial(2) := 2 * factorial(1) Se invoca a si misma y crea una cuarta variable cuyo nombre es numero y su valor es igual a 1.
4. factorial(1) := 1 * factorial(0) Se invoca a si misma y crea una quinta variable cuyo nombre es numero y su valor es igual a 0.
5. Como factorial(0) := 1, con éste valor se regresa a completar la invocación: factorial(1) := 1 * 1 , por lo que factorial(1) := 1
6. Con factorial(1) := 1, se regresa a completar: factorial(2) := 2 * 1, por lo que factorial(2) := 2
7. Con factorial(2) := 2, se regresa a completar : factorial(3) := 3 * 2, por lo que factorial(3) := 6
8. Con factorial(3) := 6, se regresa a completar : factorial(4) := 4 * 6, por lo que factorial(4) := 24 y éste será el valor que la función factorial devolverá al módulo que la haya invocado con un valor de parámetro local igual a 4 .

Un ejemplo de procedimiento recursivo es el siguiente:

Supóngase que una persona se mete a una piscina cuya profundidad es de 5 metros. Su intención es tocar el fondo de la piscina y después salir a la superficie. Tanto en el descenso como en el ascenso se le va indicando la distancia desde la superficie (a cada metro).

```
Program Piscina; Uses Crt;
        Const
          prof_max = 5;
        Var
        profundidad:int
        eger;
        procedure zambullida(Var profun :integer);
        begin
          WriteLn('BAJA 1 PROFUNDIDAD = ',profun);
          profun := profun + 1;
          if profun <=
            prof_max then
            zambullida(profun)
          else
            WriteLn;
          profun := profun - 1;
          WriteLn('SUBE 1 PROFUNDIDAD = ', profun-1)
          end; begin ClrScr;
          profundidad := 1;
          zambullida(profundidad)
        end.
```

5. Recursividad Lenguaje C

La recursividad es la posibilidad de que una función se llame a sí misma, bien directa o indirectamente. Un ejemplo típico es el cálculo del factorial de un número, definido en la forma:

```
N! = N * (N-1)! = N * (N-1) (N-2)! = N * (N-1)*(N-2)*...*2*1
```

La función ***factorial***, escrita de forma recursiva, sería como sigue:

```
unsigned long factorial(unsigned long numero)
{
     if ( numero == 1 || numero == 0 )
          return 1;
     else
          return numero*factorial(numero-1);
}
```

Supóngase la llamada a esta función para N=4, es decir **factorial(4)**. Cuando se llame por primera vez a la función, la variable **numero** valdrá 4, y por tanto devolverá el valor de ***4*factorial(3);*** pero **factorial(3)** devolverá ***3*factorial(2)**; **factorial(2)** a su vez es ***2*factorial(1)** y dado que **factorial(1)** es igual a 1 (es importante considerar que sin éste u otro caso particular, llamado ***caso base***, la función recursiva no terminaría nunca de llamarse a sí misma), el resultado final será 4*(3*(2*1)).

Por lo general la recursividad no ahorra memoria, pues ha de mantenerse una pila[7] con los valores que están siendo procesados. Tampoco es más rápida, sino más bien todo lo contrario, pero el código recursivo es más compacto y a menudo más sencillo de escribir y comprender.

Gestión dinámica de la memoria

Según lo visto hasta ahora, *la reserva o asignación de memoria* para vectores y matrices se hace de forma automática con la declaración de dichas variables, asignando suficiente memoria para resolver el problema de tamaño máximo, dejando el resto sin usar para problemas más pequeños. Así, si en una función encargada de realizar un producto de matrices, éstas se dimensionan para un tamaño máximo (100, 100), con dicha función se podrá calcular cualquier producto de un tamaño igual o inferior, pero aun en el caso de que el producto sea por ejemplo de tamaño (3, 3), la memoria reservada corresponderá al tamaño máximo (100, 100). Es muy útil el poder reservar más o menos memoria *en tiempo de ejecución*, según el tamaño del caso concreto que se vaya a resolver. A esto se llama ***reserva o gestión dinámica de memoria.***

Existen en C dos funciones que reservan la cantidad de memoria deseada en tiempo de ejecución. Dichas funciones devuelven –es decir, tienen como valor de retorno– un puntero a la primera posición de la zona de memoria reservada. Estas funciones se llaman **malloc()** y **calloc()**, y sus declaraciones, que están en la librería **stdlib.h**, son como sigue:

```
void *malloc(int n_bytes)
void *calloc(int n_datos, int tamaño_dato)
```

La función **malloc()** busca en la memoria el espacio requerido, lo reserva y devuelve un puntero al primer elemento de la zona reservada. La función **calloc()** necesita dos argumentos: el nº

---

[7] Una *pila* es un tipo especial de estructura de datos que se estudiará en INFORMÁTICA 2.

de celdas de memoria deseadas y el tamaño en bytes de cada celda; se devuelve un puntero a la primera celda de memoria. *La función **calloc()** tiene una propiedad adicional: inicializa todos los bloques a cero*.

Existe también una función llamada **free()** que deja libre la memoria reservada por **malloc()** o **calloc()** y que ya no se va a utilizar. Esta función usa como argumento el puntero devuelto por **calloc()** o **malloc()**. La memoria no se libera por defecto, sino que el programador tiene que liberarla explícitamente con la función **free()**. El prototipo de esta función es el siguiente:

```
void free(void *)
```

A continuación se presenta un ejemplo de gestión dinámica de memoria para el producto de matriz por vector {y}=[a]{x}. *Hay que tener en cuenta que reservando memoria por separado para cada fila de la matriz, no se garantiza que las filas estén contiguas en la memoria*. Por otra parte, de esta forma se pueden considerar filas de distinto tamaño. El nombre de la matriz se declara como *puntero a vector de punteros*, y los nombres de los vectores como *punteros* (recuérdese la figura 6.1). Supóngase que **N** es una constante simbólica predefinida con el número de filas.

```
// declaraciones
double **a, *x, *y;
void prod(int , double **, double *, double *);
...
// reserva de memoria para la matriz a
a = calloc(N, sizeof(double *));
for (i=0; i<N; i++)
      a[i]=calloc(N, sizeof(double));
...
// reserva de memoria para los vectores x e y
x = calloc(N, sizeof(double);
y = calloc(N, sizeof(double);
...
prod(N, a, x, y);
...

// definicion de la funcion prod()
void prod(int N, double **mat, double *x, double *y)
{...}
```

Con gestión dinámica de memoria es más fácil utilizar matrices definidas como vectores de punteros que matrices auténticas (también éstas podrían utilizarse con memoria dinámica: bastaría reservar memoria para las **N** filas a la vez y asignar convenientemente los valores al vector de punteros **a[i]**). El ejemplo anterior quedaría del siguiente modo:

```
// declaraciones
double **a, *x, *y;
void prod(int , double **, double *, double *);
...
// reserva de memoria para el vector de punteros a[]
a = calloc(N, sizeof(double *));
// reserva de memoria para toda la matriz a[][]
a[0] = calloc(N*N, sizeof(double));
// asignación de valor para los elementos del vector de punteros a[]
for (i=1; i<N; i++)
     a[i] = a[i-1]+N;
// el resto del programa sería idéntico

...
```

LAS LIBRERÍAS DEL LENGUAJE C

A continuación se incluyen en forma de tabla algunas de las funciones de librería más utilizadas en el lenguaje C.

Función	Tipo	Propósito	lib
abs(i)	int	Devuelve el valor absoluto de **i**	stdlib.h
acos(d)	double	Devuelve el arco coseno de **d**	math.h
asin(d)	double	Devuelve el arco seno de **d**	math.h
atan(d)	double	Devuelve el arco tangente de **d**	math.h
atof(s)	double	Convierte la cadena **s** en un número de doble precisión	stdlib.h
atoi(s)	long	Convierte la cadena **s** en un número entero	stdlib.h
clock()	long	Devuelve la hora del reloj del ordenador. Para pasar a segundos, dividir por la constante CLOCKS_PER_SEC	time.h
cos(d)	double	Devuelve el coseno de **d**	math.h
exit(u)	void	Cerrar todos los archivos y buffers, terminando el programa.	stdlib.h
exp(d)	double	Elevar **e** a la potencia **d** (e=2.77182...)	math.h
fabs(d)	double	Devuelve el valor absoluto de **d**	math.h
fclose(f)	int	Cierra el archivo **f**.	stdio.h
feof(f)	int	Determina si se ha encontrado un fin de archivo.	stdio.h
fgetc(f)	int	Leer un carácter del archivo **f**.	stdio.h
fgets(s,i,f)	char *	Leer una cadena **s**, con **i** caracteres, del archivo **f**	stdio.h
floor(d)	double	Devuelve un valor redondeado por defecto al entero más cercano a **d**.	math.h
fmod(d1,d2)	double	Devuelve el resto de **d1**/**d2** (con el mismo signo de **d1**)	math.h
fopen(s1,s2)	FILE *	Abre un archivo llamado **s1**, para una operación del tipo **s2**. Devuelve el puntero al archivo abierto.	stdio.h
fprintf(f,...)	int	Escribe datos en el archivo **f**.	stdio.h
fputc(c,f)	int	Escribe un carácter en el archivo **f**.	stdio.h
free(p)	void	Libera un bloque de memoria al que apunta **p**.	malloc.h
fscanf(f,...)	int	Lee datos del archivo **f**.	stdio.h
getc(f)	int	Ler un carácter del archivo **f**.	stdio.h
getchar()	int	Lee un carácter desde el dispositivo de entrada estándar.	stdio.h
log(d)	double	Devuelve el logaritmo natural de **d**.	
malloc(n)	void *	Reserva **n** bytes de memoria. Devuelve un puntero al principio del espacio reservado.	malloc.h o stdlib.h
pow(d1,d2)	double	Devuelve **d1** elevado a la potencia **d2**.	
printf(...)	int	Escribe datos en el dispositivo de salida estándar.	stdio.h
rand(void)	int	Devuelve un valor aleatorio positivo.	stdlib.h
sin(d)	double	Devuelve el seno de **d**.	math.h
sqrt(d)	double	Devuelve la raíz cuadrada de **d**.	math.h
strcmp(s1,s2)	int	Compara dos cadenas lexicográficamente.	string.h
strcomp(s1,s2)	int	Compara dos cadenas lexicográficamente, sin considerar mayúsculas o minúsculas.	string.h
strcpy(s1,s2)	char *	Copia la cadena **s2** en la cadena **s1**	string.h
strlen(s1)	int	Devuelve el número de caracteres en la cadena **s**.	string.h
system(s)	int	Pasa la orden **s** al sistema operativo.	stdlib.h
tan(d)	double	Devuelve la tangente de **d**.	math.h
time(p)	long int	Devuelve el número de segundos transcurridos desde de un tiempo base designado (1 de enero de 1970).	time.h
toupper(c)	int	convierte una letra a mayúscula.	stdlib.h o ctype.h

Nota: La columna ***tipo*** se refiere al tipo de la cantidad devuelta por la función. Un asterisco denota
puntero, y los argumentos que aparecen en la tabla tienen el significado siguiente:

c denota un argumento de tipo carácter.

d denota un argumento de doble

precisión. f denota un argumento

archivo.

i denota un argumento entero.

l denota un argumento

entero largo. p denota un

argument puntero.

s denota un argumento cadena.

u denota un argumento entero sin signo.