Tema 4: Programación modular. Recursividad

1 Introducción.

La programación modular es una de las características más significativas del paradigma imperativo. Consiste en dividir el problema a resolver en sub-problemas más pequeños.

Por ejemplo, imaginemos un programa en el que es necesario calcular el factorial de un número entero en cinco ocasiones. El código asociado a dicho cálculo es siempre el mismo por lo que podríamos copiarlo en los cinco sitios pertinentes. Otra solución sería crear un módulo que calcule el factorial de un número entero genérico y utilizarlo para realizar los cinco cálculos tal y como si fuese una operación básica. Esta solución es mucho más compacta puesto que no es necesario duplicar el código asociado al factorial.

La programación modular permite diseñar programas de forma más sencilla y legible. También hace posible la reutilización del código, tanto en el programa que estemos diseñando como para otros programas que realicen cálculos similares. Estos factores hacen que un programa pueda ser escrito por varios programadores aprovechando el rendimiento del grupo sobre el individual (sinergia).

2 Funciones

Una función o subprograma es un conjunto de operaciones que se identifica mediante un único nombre. Dicho identificador sigue las normas de nomenclatura de los datos simples y ha de describir la acción que ejecuta el subprograma. Cualquier referencia al identificador de la función implica la ejecución de las acciones que contiene.

Una función se encarga de resolver un problema parcial. Desde el programa principal podremos llamar a la función por su nombre como si fuese una operación atómica. Como en el caso de las variables, para utilizar una función tiene que estar definida antes. Por tanto, la última función definida en nuestro código ha de ser la principal o *main*.

En el siguiente apartado veremos la estructura de definición de una función, incluyendo la sintaxis para utilizarla, etc…

2.1 Estructura de una función: parámetros de entrada y salida

Una función está compuesta por dos partes: **cabecera** y **cuerpo**. La cabecera de la función define el interfaz de la misma y está compuesta por el nombre de la función, el tipo de salida y los parámetros. El cuerpo de la función es el bloque de instrucciones que define las operaciones que van a realizarse en la función.

La sintaxis de definición de una función es la siguiente:

TipoSalida NombreFuncion(Tipo1 param1, Tipo2 param2, … TipoN paramN)
[BLOQUE DE INSTRUCCIONES DEL CUERPO]

Veamos cada parte por separado:

- **Nombre**: Identificador de la función. Sigue las mismas normas de nomenclatura que los datos simples.
- **Tipo de salida**: Las funciones pueden devolver un dato simple perteneciente a los tipos de datos simples estudiados. Si la función no devuelve valor alguno (tipo vacío o *void*) recibe el nombre de **procedimiento**.
- **Parámetros**: Datos que se le pasan a la función. Podemos especificar cero, uno o varios parámetros.
- **Cuerpo**: Bloque de instrucciones que realiza la función. Mediante la instrucción *return* la función termina y vuelve al mismo sitio desde el que se llamó. Si la función devuelve valor (no es un procedimiento) tendremos que especificar dicho valor de la forma: *return(valor)*.

En el siguiente ejemplo vamos a definir una función que recibe dos números enteros *a* y *b* y devuelve el resultado de la operación *(a+b)*(a-b)*

```
int MiFuncion(int a, int b) {
    int c;
    c = (a+b)*(a-b);
    return(c);
}
```

Como vemos, la función es de tipo entero porque el resultado de la operación va a ser entero. El nombre de la función es *MiFuncion* y recibe dos parámetros enteros llamados *a* y *b*. En el cuerpo de la función se define la variable *c* de tipo entero que nos servirá para almacenar el resultado de la operación. A continuación, calculamos la operación y la almacenamos en *c*. Finalmente devolvemos el valor de c.

La definición de la función no hace nada por sí misma. Es necesario llamarla desde otra función para ejecutarla. En el siguiente fragmento de código solicitamos desde la función *main* dos enteros al usuario y llamamos a la función anterior para realizar el cálculo con los mismos:

```
void main(void) {
    int Num1, Num2, Resul;
    printf("Introduzca el primer numero: "); scanf("%d",&Num1);
```

```
        printf("Introduzca el segundo numero: "); scanf("%d",&Num2);
        Resul = MiFuncion(Num1, Num2);
        printf("Resultado = %d\n",Resul);
        return;
}
```

Como vemos, el nombre de los datos en la llamada no coincide con los definidos en la función. El compilador copia uno a uno y en el mismo orden los parámetros que se le envían sobre los parámetros de la función. Al terminar la función dichas copias se destruyen, por tanto si cambiásemos la variable *a* en la función, la variable *Num1* no cambiaría puesto que, en realidad, son variables distintas.

Dicho de otro modo, al llamar a la función se crean dos variables locales llamadas a y b de tipo entero y se les asignan los valores de la llamada: *a=Num1* y *b=Num2*. Una vez terminada la ejecución *a* y *b* se destruyen.

Como vemos, el valor devuelto en *return* se recoge en la llamada sobre el mismo nombre de la función por lo que podemos hacer la asignación como si fuese una operación atómica. Al llamar a la función, ésta se ejecuta completa, devuelve el resultado y pasamos a la siguiente instrucción. Por tanto, hasta que la función no termine no continúa el código principal.

En el siguiente ejemplo hemos escrito un programa completo para calcular el factorial de un número utilizando una función en la que se encapsula la operación del factorial:

```
#include <stdio.h>

int Factorial(int num) {
    int resultado=1;
    while( num > 0 ){
            resultado=resultado * num;
            num = num – 1;
    }
    return(resultado);
}

void main(void) {
    int numero, fact;
    printf("Introduzca el numero:"); scanf("%d",&numero);
    fact = Factorial(numero);
    printf("El factorial de %d es %d\n",numero, fact);
    return;
}
```

Como vemos, en la función se modifica el parámetro para realizar el cálculo del factorial, por tanto, cuando termine el bucle *num* valdrá 0. Sin embargo, en la función

principal la variable *numero* no habrá cambiado puesto que lo que hemos modificado es una copia de su valor.

Hasta ahora, hemos invocado a las funciones desde la función principal. Como podemos imaginar, es posible llamar a una función desde cualquier otra. En el siguiente ejemplo calculamos el máximo entre cinco números utilizando para ello una función llamada *Maximo5*, que a su vez se vale de dos funciones llamadas *Maximo2* y *Maximo3*:

```
int Maximo2(int n1, int n2) {
      if(n1>n2) return(n1);
      else        return(n2);
}

int Maximo3(int n1, int n2, int n3) {
      int soluc;
      soluc=Maximo2(n1,n2);
      soluc=Maximo2(soluc,n3);
      return( soluc );
}

int Maximo5(int n1,int n2,int n3, int n4, int n5) {
      int soluc;
      soluc=Maximo3(n1,n2,n3);
      soluc=Maximo3(n4,n5,soluc);
      return( soluc );
}

void main(void) {
      int a,b,c,d,e,maximo;
      printf("Primer numero:");     scanf("%d",&a);
      printf("Segundo numero:"); scanf("%d",&b);
      printf("Tercer numero:");     scanf("%d",&c);
      printf("Cuarto numero:");     scanf("%d",&d);
      printf("Quinto numero:");     scanf("%d",&e);
      maximo=Maximo5(a,b,c,d,e);
      printf("El maximo es %d\n",maximo);
}
```

La función *Maximo3* utiliza la función *Maximo2* para resolver el problema del máximo entre 3 números. A su vez, la función *Maximo5* utiliza la función *Maximo3* para calcular el máximo de 5 números. Como vemos, para resolver el problema lo hemos descompuesto en problemas más pequeños de fácil solución. Como ejercicio, el lector podría probar a construir la función *Maximo5* sin ninguna función adicional y comprobará con ello que el resultado resulta más complicado.

También podemos escribir las funciones *Maximo3* y *Maximo5* de una forma más compacta sin necesidad de variables locales. El resultado es igualmente efectivo:

```
int Maximo3(int n1, int n2, int n3) {
    return( Maximo2(n1, Maximo2(n2,n3) );
}
int Maximo5(int n1,int n2,int n3, int n4, int n5) {
    return( Maximo2( Maximo2(n1,n2), Maximo3(n3,n4,n5) ) );
}
```

2.2 Variables locales y globales. Ámbito de un identificador.

Entendemos por **ámbito de un identificador** a la zona del código en la que dicho identificador tiene efecto. El ámbito de un identificador marca la zona en la que podemos utilizarlo. Fuera de su ámbito el identificador no existe.

Hasta ahora, hemos comentado que para utilizar una función o una variable, tiene que estar previamente definida. Por tanto, el ámbito de un identificador comienza a partir de su definición.

En el ejemplo del apartado anterior, si la función *Maximo5* estuviese colocada antes de la función *Maximo3*, la primera no podría llamar a la segunda puesto que el ámbito del identificador *Maximo3* comenzaría a partir de su definición, no antes. Dicho de otro modo, un identificador existe desde el mismo momento en que se define.

Como hemos podido observar en ejemplos anteriores, las variables locales declaradas en una función no tienen efecto fuera de ella. Esto se debe a que el ámbito de un identificador termina cuando finaliza el bloque en el que está enmarcado. Cuando finaliza dicho bloque el identificador deja de existir, por lo que no podremos referirnos a él.

Veamos un ejemplo:

```
#include <stdio.h>
#define DIAS_SEM 7
int  Semanas( int dias ) {
    int resul;
    resul = dias / DIAS_SEM;
    return( resul );
}

void main(void) {
    int i;
    for(i=0 ; i<10;i++){
            int dias, sem;
            printf("Numero de dias: "); scanf("%d","&dias);
            sem=Semanas(dias);
            printf("%d dias equivalen a %d semanas\n", dias, sem);
```

```
        }
        return;
    }
```

Como vemos, en la función principal realizamos bucle que consta de 10 iteraciones. En cada iteración solicitamos un número de días usuario y calculamos de cuántas semanas consta dicho número de días utilizando la función *Semanas*. Vamos a estudiar el rango de cada identificador:

La variable *i* que está definida al principio de la función *main*, tendrá efecto desde su definición hasta el final de dicha función. Dentro del bucle definimos dos variables: *dias* y *sem*. Dichas variables se han definido al principio del bloque asociado al bucle, por tanto, tendrán efecto desde su declaración hasta el final del bucle, pero no fuera de éste. Sin embargo, la variable *i* tendrá efecto tanto fuera como dentro del bucle por estar enmarcada en un bloque superior.

La variable *dias*, perteneciente a la cabecera de la función *Semanas*, tendrá efecto dentro del bloque de la función pero no fuera de éste. Igualmente, la variable *resul*, declarada al principio del bloque de la función tendrá efecto solo dentro del mismo. Por tanto, no tiene sentido utilizar ninguna de estas variables fuera de la función.

Finalmente nos quedan dos identificadores: el de la constante DIAS_SEM y el de la función Semanas. Ambos identificadores se encuentran fuera de todo bloque, por tanto se enmarcan dentro del ámbito de todo el programa. El ámbito de estos identificadores será global, esto es, desde su definición hasta el final del programa.

En general, diremos que un identificador es **local** cuando esté definido dentro de un bloque y **global** cuando esté definido fuera de todo bloque y, por tanto, afecte a todo el código. En ocasiones, es posible declarar una **variable global** que pueda ser utilizada desde cualquier punto del programa (bastaría con definirla al principio del código). En adelante, evitaremos el uso de este tipo de variables puesto que esto no se considera una buena estrategia de programación.

En ocasiones podremos definir dos variables sobre el mismo ámbito y con el mismo nombre, siempre que una de ellas se encuentre en un bloque más interior. Veamos un ejemplo:

```
void main(void) {
    int i;
    for(i=0;i<20;i++){
            int i;
            printf("Introduzca numero");  scanf("%d",&i);
            printf("Numero = %d\n",i);
    }
}
```

Como vemos, hemos utilizado la variable *i* para construir el bucle y dentro de éste hemos utilizado **otra variable** también llamada *i*. Puesto que la segunda se

encuentra en un bloque más interior, el compilador permite duplicar ambos nombres sin que las variables se afecten entre sí. En cualquier caso, al referirnos a *i* perdurará la definición más interna.

En adelante evitaremos este uso de los identificadores puesto que en la mayoría de los casos produce efectos colaterales indeseados.

2.3 Paso de parámetros por valor y por referencia

Hasta ahora las funciones que hemos descrito son capaces de aceptar un conjunto de parámetros como entrada y devolver un único valor. Los parámetros de entrada son copiados sobre variables locales a la función, lo que hace que las variables que se introducen en la llamada no puedan ser modificadas por la función. Por ejemplo:

```
void Funcion(int param) {
    param=param+1;
    return;
}
void main(void) {
    int variable = 5;
    Funcion(variable);
    printf("variable = %d\n",variable);
    return;
}
```

Como vemos, tenemos una función que recibe un parámetro entero. Desde la función principal llamamos a dicha función pasándole una variable con el valor 5. Dentro de la función se le suma una unidad a la variable *param*, que es una copia de *variable*, por lo que *param* pasará a valer 6 pero *variable* seguirá valiendo 5.

Este procedimiento de paso de parámetros recibe el nombre de **paso de parámetros por valor**.

Alternativamente podemos hacer que las variables que se envían a la función puedan modificar su valor dentro de ésta. Para ello, en vez de especificar a la función el valor de las variables, se especifica su dirección de memoria. Esto es lo que se conoce como **paso de parámetros por referencia**.

Una variable está compuesta, como sabemos por su nombre, su tipo y su valor o contenido. Además, una variable se ubica dentro de una **dirección de memoria** que le asigna el sistema cuando se crea. Dicha dirección es un valor numérico que localiza de manera unívoca a una variable dentro de la memoria asignada a nuestro programa. Para extraer la dirección de memoria asociada a una variable utilizaremos el operador **&** antepuesto a su nombre. Para extraer el **contenido** de una dirección de memoria utilizaremos el operador ***** antepuesto a su nombre. Una variable declarada utilizando el operador ***** se denomina **puntero**. Un puntero es una variable que almacena una dirección de memoria independientemente de lo que ésta contenga.

Para conseguir que una función pueda modificar los parámetros originales utilizaremos punteros, o sea, en vez de enviar el contenido de la variable, enviaremos su dirección de memoria. En la cabecera de la función tendremos que especificar que lo que recibe no son variables sino punteros a dichas variables. Para ello, al especificar dichos parámetros utilizaremos el operador *. Dentro de la función, puesto que lo que nos interesa es el contenido, utilizaremos también el operador * para modificar los parámetros. En la llamada a la función utilizaremos el operador & para extraer la dirección de las variables.

En el siguiente ejemplo hemos definido una función llamada SumaUno que modifica el valor de la variable de entrada sumándole una unidad:

```
void SumaUno(int *numero) {
    (*numero) = (*numero) + 1;
    return;
}
void main(void) {
    int mivariable = 10;
    printf("Mi variable vale %d\n",mivariable);
    SumaUno( &mivariable );
    printf("Mi variable vale %d\n",mivariable);
    return;
}
```

Como vemos, en la declaración de la función especificamos que el parámetro de entrada será una dirección de memoria asociada a una variable de tipo entero (puntero a entero). Dentro de la función modificamos el contenido de la variable utilizando el operador *. Como podemos observar, hemos utilizado paréntesis en las operaciones para no confundir el operador contenido * con la multiplicación.

Desde la función principal llamamos a la función *SumaUno* enviando la dirección de la variable entera *mivariable*. Por tanto, antes de la llamada a la función la variable *mivariable* contendrá el valor 10 y después contendrá el valor 11.

Veamos otro ejemplo en el que enviamos dos variables enteras a una función con el propósito de intercambiar su valor (swap):

```
void Intercambiar( int *var1, int *var2) {
    int aux;
    aux= (*var1);
    (*var1) = (*var2);
    (*var2) = aux;
    return;
}
void main(void) {
    int a=5, b=7;
    printf("a vale %d y b vale %d\n",a,b);
    Intercambiar( &a, &b );
```

```
        printf("a vale %d y b vale %d\n",a,b);
        return;
    }
```

Como vemos, la función recibe los dos parámetros por referencia y a continuación, utilizando una variable auxiliar intercambia sus valores. En la función principal antes de la llamada *a* vale 5 y *b* vale 7. Después de la llamada *a* valdrá 7 y *b* valdrá 5.

Hasta ahora hemos estudiado cómo pasar tanto por valor como por referencia tipos de datos simples a una función. En el caso de los tipos de datos estructurados tales como vectores o matrices el procedimiento cambia. En el caso especial de las cadenas de caracteres, que es el único tipo de datos estructurado que conocemos hasta el momento, el paso de parámetros siempre es por referencia y no es necesario especificar ningún operador. Esto es así porque las cadenas de caracteres se definen de forma indirecta con punteros.

Veamos un ejemplo de función para pasar a mayúsculas una frase:

```
    void Mayusculas( char cadena[] ) {
        int i;
        int lon=strlen(cadena);
        for(i=0;i<lon;i++){
                if(cadena[i]>='a' && cadena[i]<='z'){
                        cadena[i]=cadena[i]-'a'+'A';
                }
        }
        return;
    }
    void main(void) {
        char cad[255];
        printf("Introduzca cadena: ");
        scanf("%s",cad);
        Mayusculas(cad);
        printf("La cadena en mayusculas es: %s",cad);
        return;
    }
```

En la función *Mayusculas* se recorre la cadena carácter a carácter comprobando si es una letra minúscula (valor ASCII entre 'a' y 'z'). Si es así se traslada al rango de las mayúsculas (valor ASCII entre 'A' y 'Z'). Si le pasamos, por ejemplo, la cadena "HolaMundo" el resultado tras aplicar la función sería "HOLAMUNDO".

Como vemos, en el caso de las cadenas de caracteres siempre efectuamos el paso de parámetros por referencia sin necesidad de especificarlo con ningún operador específico. Además en la cabecera de la función no es necesario especificar el tamaño de la cadena.

Llegados a este punto podremos entender por qué en la función scanf es necesario el uso del operador & en todos los tipos de datos simples excepto en cadenas.

2.4 Recursividad

Tal como se ha comentado anteriormente, es posible realizar una llamada desde una función a otra para resolver un problema específico. Además, existe la posibilidad de que una función se llame a sí misma. Cuando una función se invoca a sí misma en alguna parte de su código decimos que es **recursiva**.

```
void EjemploRecursivo(int n){
    printf("Antes de la llamada n=%d\n",n);
    if(n>0){
            EjemploRecursivo(n-1);
    }
    printf("Despues de la llamada n=%d\n",n);
    return;
}
```

Cada llamada crea una nueva instancia de la función. Ante una llamada, la instancia actual se pausa y continúa cuando la llamada finalice. La recursividad por tanto no es como un bucle donde cada llamada hace que la función empiece desde el principio. Cuando hacemos una llamada recursiva se crea otra función (distinta a la actual aunque con la misma forma). En el ejemplo anterior, si hacemos EjemploRecursivo(10); se crearán 11 funciones que imprimirán 22 mensajes por pantalla (todos los mensajes "antes de la llamada" saldrían en orden y al principio mientras que los mensajes "Despues de la llamada" saldrían en segundo lugar y en orden inverso).

En muchas ocasiones, la definición de un problema nos obliga a utilizar una estructura recursiva para solucionarlo. Por ejemplo, la definición del factorial de un número se puede ver de la siguiente forma:

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

Como vemos, la operación factorial se puede definir en base a sí misma. De forma general podemos decir que el factorial de un número es dicho número multiplicado por el factorial del número menos una unidad (por ejemplo, el factorial de 5 es igual a 5 multiplicado por el factorial de 4). Este es el caso general o **caso recursivo**.

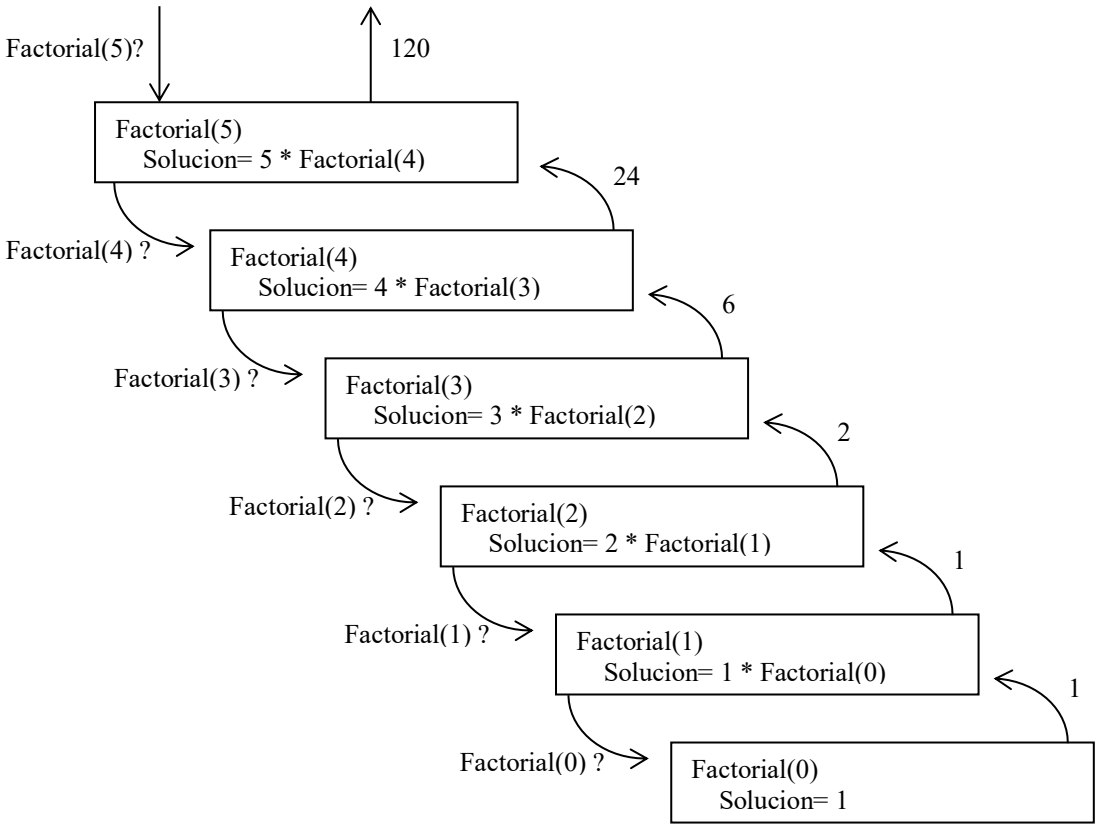
Si la definición solamente consta del caso recursivo sería infinita. Es por ello que necesitamos un caso especial no recursivo que se resuelva directamente. Esto es lo que se llama el **caso base**, que en este caso se produce cuando el número es 0 y se resuelve directamente por la regla 0!=1.

En lenguaje C podemos escribir la función factorial recursiva de la siguiente forma:

```
int Factorial( int num ) {
    int solucion;
    if( num == 0 ){
        solucion=1;
    }else{
        solucion = num * Factorial( num - 1 );
    }
    return (solucion);
}
```

Como vemos, la función *Factorial* recibe un número entero llamado *num* y devuelve otro número entero que será la solución, esto es *num!*. Dentro de la función discriminamos si estamos o no en el caso base (num == 0).

Si estamos en el caso base devolvemos directamente 1, que es el factorial de 0. Si no, la solución será multiplicar num por el resultado del factorial de num-1. Cada vez que llamamos a la función factorial se crea una nueva instancia de la misma independiente de la actual. Hasta que no se resuelva dicha instancia no podremos continuar.



Veamos una traza de la función para resolver el factorial del número 5:

La función se ejecuta en primer lugar para calcular el factorial de *5*. Puesto que resulta distinto de cero, la función calcula *5*Factorial(4)*. Dicho cálculo no puede resolverse directamente, puesto que es necesario llamar a la función de nuevo con el valor *4*. La nueva instancia de la función calculará *4*Factorial(3)*, que a su vez quedará pendiente hasta que termine *Factorial(3)*. La nueva instancia de factorial calculará *3*Factorial(2)*. *Factorial(2)* se resolverá como *2*Factorial(1)* y *Factorial(1)* como *1*Factorial(0)*. Llegados a este punto la última instancia no necesita hacer ninguna llamada puesto que ha llegado al caso base y devuelve directamente el valor *1*. En este momento volvemos a donde se llamó con el valor resuelto: *Factorial(1)=1*1=1*. Este valor se retorna a su vez a donde se llamó resolviendo *Factorial(2)=2*1=2*. Continuamos el cálculo en cascada resolviendo *Factorial(3)=3*2= 6*, *Factorial(4)=4*6=24*, y finalmente *Factorial(5) =5*24 =120*. Este ya es el resultado final que será devuelto a donde se llamó.

Como vemos, al efectuar la llamada recursiva la función queda pendiente hasta que ésta se resuelva, por tanto, el procedimiento realiza las peticiones hacia delante y va recogiendo los resultados hacia atrás.

Podemos expresar el procedimiento de una forma más algebraica que posiblemente aclare el orden de ejecución de las funciones:

```
Factorial(5) =
5 * (Factorial(4)) =
5 * ( 4 * (Factorial(3))) =
5 * ( 4 * ( 3 * (Factorial(2)))) =
5 * ( 4 * ( 3 * ( 2 * (Factorial(1))))) =
5 * ( 4 * ( 3 * ( 2 * ( 1 * (Factorial(0)))))) =
5 * ( 4 * ( 3 * ( 2 * ( 1 * 1)))) =
5 * ( 4 * ( 3 * ( 2 * 1))) =
5 * ( 4 * ( 3 * 2)) =
5 * ( 4 * 6) =
5 * 24 =
120
```

Como vemos, los cálculos quedan pendientes hasta que se resuelvan todas las llamadas. A este procedimiento de cálculo hacia atrás se le denomina **backward**.

Veamos otro ejemplo de función recursiva similar al anterior. A partir de dos números a y b, queremos calcular el exponente a^b. Este procedimiento se puede ver desde un punto de vista recursivo de la siguiente manera:

$$a^b = \begin{cases} 1 & si \quad b = 0 \\ a\left(a^{b-1}\right) & si \quad b > 0 \end{cases}$$

En el caso recursivo, podemos calcular *a* elevado a *b* como *a* por *a* elevado a *b-1*. El caso base es cuando *b* valga *0*, en cuyo caso el resultado es *1* valga lo que valga *a*.

La función recursiva sería:

```
int Exponente( int a, int b ) {
    if( b == 0 ){
        return (1);
    }else{
        return( a * Exponente( a , b-1 ));
    }
}
```

Como vemos, se trata de una función muy similar al anterior. En este caso hemos compactado un poco más el código sin necesidad de utilizar una variable local.

En ocasiones también es posible tener varios casos base. Por ejemplo, la serie Fibonacci se define recursivamente de la forma:

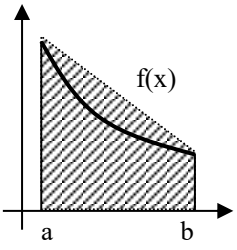
$$Fib(n) = \begin{cases} 0 & si \quad n = 1 \\ 1 & si \quad n = 2 \\ Fib(n-1) + Fib(n-2) & si \quad n > 2 \end{cases}$$

Como vemos, el valor de la serie para n=1 o n=2 es constante. El resto de valores se calculan recursivamente. La función recursiva que resuelve este problema sería:

```
int Fibonacci( int n ) {
    if(n<=2){
        return(n-1);
    }else{
        return( Fibonacci(n-1) + Fibonacci(n-2) );
    }
}
```

Veamos un caso un poco más complicado, en el que vamos a calcular la integral aproximada de una función *f(x)* en un cierto intervalo [*a,b*]. Si el intervalo es suficientemente pequeño, podemos aproximar la integral de la función utilizando la regla del trapecio:

$$S(a,b) = \frac{f(a) + f(b)}{2}(b-a)$$



Si el intervalo es grande, el error de aproximación será muy alto. Por tanto, marcaremos un tamaño mínimo de intervalo al que llamaremos T, por debajo del cual la estimación será segura. Si el intervalo es mayor al umbral, lo dividiremos en dos partes y aplicaremos la ecuación a cada parte. El valor de la integral será la suma de la integral de cada parte. Si el intervalo es menor al umbral realizaremos la aproximación directamente.

Este procedimiento puede verse recursivamente de la siguiente forma:

$$S(a,b)=\begin{cases}\dfrac{f(a)+f(b)}{2}(b-a) & si \quad (b-a)<T\\ S\left(a,\dfrac{a+b}{2}\right)+S\left(\dfrac{a+b}{2},b\right) & si \quad (b-a)\geq T\end{cases}$$

El siguiente programa calcularía el valor de la integral de la función *Func(x)=x²+x+2* en un cierto intervalo [*a,b*] que solicitamos al usuario:

```
float Func( float x ) {
    float solucion;
    solucion = x*x + x + 2.0;
    return( solucion );
}

float Integral(float a, float b, float T) {
    float tam,soluc;
    tam=b-a;
    if( tam < T ){
        soluc = tam * ( Func(a) + Func(b) )/2.0;
    }else{
        soluc = Integral(a,(a+b)/2.0,T) + Integral((a+b)/2.0,b,T);
    }
    return(soluc);
}

void main(void) {
    float min,max,umbral;
    printf("Valor minimo del intervalo : "); scanf("%f",&min);
    printf("Valor maximo del intervalo : "); scanf("%f",&max);
    printf("Tamaño maximo: "); scanf("%f",&umbral);
    printf("Valor de la integral: %f\n", Integral(min,max,umbral));
    return;
}
```

Como se puede comprobar se ha encapsulado la función *Func* en una función de C, puesto que aparece en varios lugares a lo largo del código. En este caso la función posee dos llamadas recursivas, una para cada mitad del intervalo en caso de que éste sea excesivamente grande.

3 Esquema de un programa en C

En este apartado describiremos el esquema general de un programa en C. Antes de esto, nos falta por conocer por completo la cabecera de la función principal.

Hasta ahora hemos utilizado la función principal sin parámetros de entrada ni salida, pero en realidad sí que los tiene. La estructura completa de la función es la siguiente:

```
int main(int argc, char **argv) {
    //acciones
    return(Valor_Salida);
}
```

La función *main* devuelve un valor entero que se especifica, como en todas las funciones, en el *return*. Dicho valor es un código de error que nuestro programa envía al sistema operativo. Si el programa termina correctamente ha de devolver el valor 0. Si se produce un error devolveremos un número entero distinto de cero.

Los parámetros de la función son, respectivamente, el número de argumentos del programa y las cadenas relacionadas con dichos argumentos. Imaginemos que nuestro programa ejecutable se llama Prog.exe y que desde la línea de comandos ejecutamos el programa de la forma:

```
C:\>Prog 50 anabel
```

El sistema operativo recoge estos parámetros y los envía a la cabecera de la función principal del programa. El entero argc simboliza el número de parámetros, que en este caso es 3 (se cuenta el nombre del programa ejecutable). En argv se introducen las cadenas asociadas a cada parámetro en orden. La cadena 0 será "Prog", la 1 será "50" y la 2 será "anabel". Como podemos ver, argv es una lista de cadenas (o doble cadena).

Veamos un ejemplo en el que listamos todos los parámetros entrantes al programa:

```
int main( int argc, char **argv) {
    int i;
    printf("Numero de parametros = %d\n",argc);
    for(i=0; i<argc; i++){
            printf("Parametro %d = %s\n", i, argv[i]);
    }
    return(0);
}
```

Imaginemos que el archivo ejecutable se llama programa.exe. Veamos un ejemplo de ejecución del mismo:

```
C:\>programa Juan Angel Anabel
Numero de parametros = 4
Parametro 0 = C:\PROGRAMA.EXE
Parametro 1 = Juan
Parametro 2 = Angel
Parametro 3 = Anabel
```

Gracias a esto es posible disponer de un programa que acepte parámetros de entrada directamente desde el sistema operativo.

Llegados a este punto, podemos especificar la estructura general de un programa en C:

```
/*        Estructura General de un programa en C
              Fundamentos de la Programación          */

/* Inclusión de las librerías que se van a usar */
#include <libreria1.h>
#include <libreria2.h>
// etc…

/* Definición de constantes y variables globales */
#define Constante1 Valor1
#define Constante2 Valor2
// etc...

/* Definicion de funciones */
TipoFuncion1 Funcion1( Tipo1 param1, Tipo2 param2, …) {
      //Definicion de variables locales de Funcion1
      //Instrucciones asociadas a la Funcion1
}

TipoFuncion2 Funcion2( Tipo1 param1, Tipo2 param2, …) {
      //Definicion de variables locales de Funcion2
      //Instrucciones asociadas a la Funcion2
}
//etc…

/* Función principal */
int  main(int argc, char **argv) {
      //Definicion de variables locales de la funcion principal
      //Instrucciones asociadas a la funcion principal
      //fin del programa
      return( ErrorDeSalida );
}
```

En este estructura tendremos que tener en cuenta que para utilizar una función ha de estar declarada anteriormente, por lo que el orden de definición de las funciones tendrá que realizarse convenientemente.