TEMA 26

PROGRAMACIÓN MODULAR. DISEÑO DE FUNCIONES: RECURSIVIDAD. LIBRERÍAS.

ÍNDICE

1. INTRODUCCIÓN

La resolución de problemas complejos se facilita considerablemente si se dividen en problemas más pequeños (**subproblemas**). La solución a estos subproblemas se realiza con subalgoritmos. Los subalgoritmos (subprogramas) pueden ser de dos tipos: *funciones* y *procedimientos* o subrutinas. Son unidades de programa o módulos que están diseñados para ejecutar alguna tarea específica. Estas funciones y procedimientos se escriben solamente una vez, pero pueden ser referenciados en diferentes puntos de un programa de modo que se puede evitar la duplicación innecesaria del código.

Un método para resolver un problema complejo es dividirlo en subproblemas (problemas más sencillos) y a continuación dividir estos problemas en otros más simples hasta que los problemas pequeños sean fáciles de resolver. Este método de diseñar la solución de un problema principal obteniendo las soluciones de sus subproblemas se conoce como "**diseño descendente**" (**top-down**). Se denomina descendente, ya que, se inicia en la parte superior con un problema general y el diseño específico de las soluciones de los subproblemas. Normalmente, las partes en que se divide un programa deben poder desarrollarse independientemente entre sí.

Normalmente, los pasos diseñados en el primer esbozo de un algoritmo son incompletos e indicarán sólo unos pocos pasos. Tras esta primera descripción, éstos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina "**refinamiento del algoritmo**". Para problemas complejos, se necesitarán con frecuencia diferentes niveles de refinamiento antes de obtener un algoritmo claro, preciso y completo.

Un subprograma puede realizar las mismas acciones que un programa: aceptar datos, realizar algunos cálculos y devolver resultados. Un subprograma, sin embargo, se utiliza por el programa principal para un propósito específico. El subprograma recibe datos desde el programa y le devuelve resultados. Se dice que el programa principal "*llama*" o "*invoca*" al subprograma. El subprograma ejecuta la tarea y, a continuación, devuelve el control al programa. Esto puede suceder en diferentes lugares del programa. Cada vez que se llama a un subprograma, el control retorna al lugar donde fue hecha la llamada. Un subprograma puede llamar, a su vez, a sus propios subprogramas.

Idea de la división del problema

Dado un problema determinado, para obtener su resolución final podemos optar por dividir el problema en partes más pequeñas y sencillas de forma que nos resuelvan aspectos particulares y concretos. Mediante las combinaciones oportunas de los subproblemas resueltos obtendremos como resultado el conjunto del problema total. Así, habremos dividido el problema en módulos que pueden ser analizados, programados y puestos a punto por separado.

Ejemplo: Sistema de seguimiento Horario

Se desea obtener el tiempo de trabajo efectivo de una persona en un día en una determinada empresa en la que se ha establecido un sistema de seguimiento horario en el cual se registran las entradas y salidas que realiza cada trabajador a su puesto de trabajo. Para ello hay una serie de factores que hay que tener en cuenta:

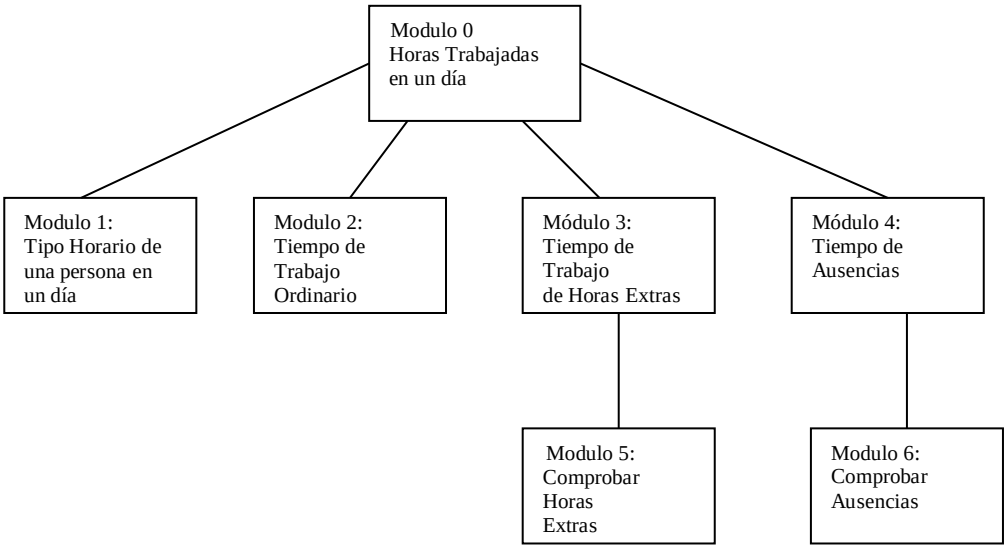
- A cada persona se le asigna un tipo de horario para un rango de fechas determinado, de tal manera que en un día determinado tendrá un tipo de horario concreto.

- Cada tipo de horario tendrá uno o varios rangos de trabajo válidos en un día. El trabajo ordinario valido será solamente aquél que se realiza dentro de dichos rangos, pudiendo haber distintas entradas y salidas al puesto de trabajo en cada día.

- Una persona puede realizar horas extras. Estas horas extras deberán haber sido validadas por su superior jerárquico mediante solicitud previa del trabajador, comprobándose, además, que hubo asistencia del trabajador a su puesto de trabajo en ellas.

- De igual manera, existirán ausencias autorizadas del trabajador de su puesto de trabajo. Al igual que las horas extra debe comprobarse que estas ausencias han sido validadas por el superior jerárquico a petición del trabajador.

Esquema de Solución Modular de Sistema de Seguimiento Horario:



2. PROGRAMACIÓN MODULAR

Esta técnica está basada en la metodología denominada "**divide y vencerás**", es decir, en la división del programa en subprogramas; cada uno de ellos ejecuta una tarea independiente, y se codifican por separado unos de otros, hasta conseguir el nivel óptimo en estos subprogramas. Así, siguiendo un método ascendente o descendente se llegará a la descomposición final del problema en módulos con una relación jerárquica.

El programa principal controla todo lo que sucede y los submódulos o subprogramas se ejecutan devolviendo el control al programa principal.

En la solución expuesta en el ejemplo anterior vemos como se desarrolla una red de módulos en el que cada uno intenta resolver problemas particulares. La programación modular se basa así en:

- Un diseño descendente (Top Down) de la red de módulos.
- Un montaje ascendente (Bottom up) de esta red.

De esta forma, un programador puede estar escribiendo el módulo 3 mientras otro está haciendo lo propio con el módulo 5. Vemos, por tanto, que en la elaboración de un programa con criterios modulares aparecen varias fases:

- Diseño descendente de la red de módulos.
- Análisis de cada relación (parámetros que se le envían a cada módulo y parámetros que éste devuelve).
- Diseño de cada módulo.
- Montaje ascendente.

Las *ventajas* más importantes del diseño descendente son: el problema se comprende más fácilmente al dividirse en partes más simples o módulos, las modificaciones en los módulos son más fáciles, y la verificación del problema es más sencilla.

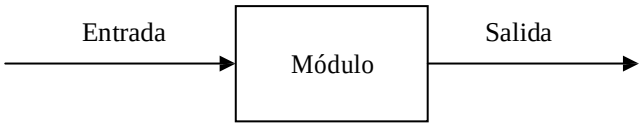
Un **módulo** se distingue por:

- Estar formado por una o varias instrucciones que están físicamente juntas.
- Poderse referenciar por un nombre.
- Poder ser llamado desde diferentes puntos del programa.

Un módulo puede ser: un programa, una función o una subrutina.

2.1. Características de los módulos

a) Un módulo debe tener una estructura de caja negra, de forma que, al facilitarle unos valores de entrada, suministre unas salidas que sean exclusivamente función de los valores de entrada:



Para ello se debe hacer módulos con:

- máxima cohesión (compenetración dentro de un mismo módulo)
- mínimo acoplamiento (comportamiento dependiente entre módulos)

Esto es, se trata de hacer módulos con un máximo de independencia. Para ello, interesará que el número de variables globales sea mínimo.

La salida debe ser función de la entrada pero no de ningún estado interno (es decir, no debe depender del estado anterior).

b) Un módulo debe corresponder a una función lógica perfectamente diferenciada.

c) Los módulos deben de ser pequeños para que sean claros y de poca complejidad.

d) No conviene utilizar demasiados niveles, ya que, entonces sería la propia red la que adquiriría complejidad.

e) Sería deseable tener puntos de entrada y salida únicos.

Clasificación de los módulos

Según las funciones que puede desarrollar cada módulo, éstos se clasifican en:

- Módulos de tipo raíz, director o principal.
- Módulos de tipo subraíz
- Módulos de entrada (captura de datos).
- Módulos de variación de entradas.
- Módulos de proceso.
- Módulos de creación y de formato de salidas.

2.2. Técnicas de programación modular

Las fases de resolución de un problema con programación modular son las siguientes:

Estudio de las especificaciones del problema. Se suelen utilizar tablas de decisión.
Confección de un ordinograma o tabla de decisión de cada módulo.
Codificación de cada módulo en el lenguaje adecuado.
Pruebas parciales de cada uno de los módulos componentes.
Prueba final de los módulos enlazados.

El diseño de una aplicación con programación modular consiste en realizar una red de módulos. Existirá un módulo raíz, que se denomina principal o director.

Cada módulo sólo puede tener una entrada y una salida que lo enlazan con el módulo principal, incluso habiendo estructuras repetitivas y alternativas dentro de un módulo.

2.3. Criterios de modularización

La división de un programa en módulos debe cumplir los siguientes criterios:

- Cada módulo debe corresponder a una función lógica diferenciada.
- El tamaño de cada módulo es variable. Deben ser pequeños para que sean claros y de poca complejidad. Las normas varían según las situaciones. Como norma general práctica se puede considerar el tamaño máximo de un módulo como una página de listado de impresora.
- Evitar variables externas.
- No utilizar demasiados niveles de modularización para evitar complejidad de la red.
- Estructura de caja negra para cada módulo: la salida debe ser función exclusiva de la entrada.

2.4. Objetivos de la programación modular

Con la programación modular se persigue la consecución de los siguientes objetivos:

Disminuir complejidad. Al dividir un programa en partes, se disminuye la complejidad de cada una de las unidades formadas y en forma más que proporcional.

Disminuir el coste. El coste de la programación disminuye automáticamente como consecuencia de la disminución de la complejidad y por la metodología implícita

Aumentar la claridad. Como consecuencia de la disminución de la complejidad de cada unidad se comprende que cada una de ellas ganará en claridad, resultando más fácil de entender para el propio programador y/o terceras personas.

Aumentar la fiabilidad. La disminución de la complejidad y la metodología asociada, necesariamente comportarán un aumento de la fiabilidad de los programas.

Aumentar el control del proyecto informático. Al resultar módulos acotados, cada uno de ellos puede analizarse, programarse y ponerse a punto por separado, minimizándose los problemas de puesta a punto final.

Facilitar la extensibilidad. La ampliación de un programa construido con técnicas modulares es mucho más simple puesto que se limita a añadir módulos nuevos o a ampliar solamente algunos de los ya existentes, quedando el resto inalterado.

Facilitar las modificaciones y correcciones. Al quedar automáticamente localizadas en un módulo concreto.

Obtener recursos generalizados. La programación modular facilita la elaboración de procedimientos generales que podrán utilizarse en otras aplicaciones posteriores.

3. FUNCIONES

Matemáticamente, una función es una operación que toma uno o más valores llamados **argumentos** y produce un valor denominado **resultado** o **valor** de la función para los argumentos dados. Todos los lenguajes de programación tienen funciones incorporadas o intrínsecas y funciones definidas por el usuario. Así, por ejemplo, la siguiente es una función: $f(x) = 1 - x$, donde f es el nombre de la función y x es el argumento. Ningún valor específico se asocia con x; es un *parámetro formal* utilizado en la definición de la función. Para evaluar f debemos darle un valor real o actual a x, con este valor se puede calcular el resultado. Una función puede tener uno o varios argumentos. Sin embargo, un único valor se asocia con la función para cualquier conjunto dado de argumentos.

Cada lenguaje de programación tiene sus propias funciones incorporadas, que se utilizan escribiendo sus nombres con los argumentos adecuados. Las funciones se evocan utilizando su nombre en una expresión, con los argumentos actuales o reales encerrados entre paréntesis.

Las funciones incorporadas en el sistema se denominan *funciones internas* o intrínsecas y las funciones definidas por el usuario *funciones externas*. Cuando las funciones estándares o internas no permiten realizar el tipo de cálculo deseado es necesario recurrir a las funciones externas que pueden ser definidas por el usuario mediante una *declaración de función.*

3.1. Declaración de funciones

La declaración de una función requiere una serie de pasos que la definen. Una función tiene una constitución similar a un algoritmo; por consiguiente, constará de una *cabecera* con la definición de la función seguida por el *cuerpo* de la función, que serán una serie de acciones o instrucciones cuya ejecución hará que se asigne un valor al nombre de la función. Esto designa el valor particular del resultado que ha de devolverse al programa llamador.

Función nombre_función (par1, par2, par3, ...)
<acciones>

par1, par2, par3, ...	son los parámetros formales o argumentos
nombre_función	nombre asociado con la función (será un nombre de identificador válido)
<acciones>	instrucciones que constituyen la definición de la función; deben contener una sola acción de designación que asigne un valor al nombre de la función.

Ejemplo:

La función f(x) = 1 – x

Se definirá por:

```
función F(x)
Inicio
        F    1 – x
Fin
```

Es aconsejable incluir comentarios para describir la función, explicando brevemente lo que hace, lo que representan sus parámetros y cualquier otra información relevante.

En algunos lenguajes de programación que exigen una sección de declaraciones (como Pascal), las funciones se situarán al principio del programa.

Para que las acciones descritas en un subprograma función sean ejecutadas, se necesita que éste sea invocado desde un programa principal o desde otros subprogramas a fin de proporcionarle los argumentos de entrada necesarios para realizar esas acciones.

Los argumentos de la declaración de la función se denominan *parámetros formales*, ficticios o mudos; son nombres de variable, de otras funciones o procedimientos, y que sólo se utilizan dentro del cuerpo de la función. Los argumentos utilizados en la llamada a la función se denominan *parámetros actuales*, que, a su vez, pueden ser constantes, variables, expresiones, valores de funciones o nombres de funciones o procedimientos.

En la declaración de una función debe aparecer el **tipo** de dicha función. Este tipo podrá ser cualquiera de los soportados por el lenguaje de programación utilizado. Si el lenguaje lo permite, el tipo podrá ser definido por el usuario.

Definir el tipo de una función es definir el tipo del valor que retorna la misma.

3.2. Invocación de funciones

Una función se llama *mediante referencia*, de la forma siguiente:

Nombre_función (lista de parámetros actuales)

Nombre_función	es la función que se llama
Lista de parámetros actuales	son constantes, variables, expresiones, valores de funciones, nombres de funciones o procedimientos

Cada vez que se llama a una función desde el algoritmo principal se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función, y se presupone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

Una llamada a la función implica los siguientes pasos:

1. A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual (esta correspondencia se denomina llamada por valor).
2. Se ejecuta el cuerpo de acciones de la función.
3. Se devuelve el valor de la función y se retorna al punto de llamada.

La forma de devolver el valor de la función va a depender del lenguaje utilizado. Existen dos formas que engloban a la mayoría de los lenguajes de programación:

- Utilizando el nombre de la función como si fuese una variable. Por ejemplo:

```
Function Área (R:real):real;
Const Pi=3.141519;
    Begin
    Área:=Pi*R*R;
    End
```

(sintaxis correspondiente a Pascal)

- Utilizando la instrucción *return valor*, donde valor será una constante o una variable del tipo de la función. Por ejemplo:

```
float Área (float R)
{
const float Pi=3.141519;
return Pi*R*R;
}
```

(sintaxis correspondiente a C)

4. ÁMBITO DE LAS VARIABLES

Antes de estudiar con detalle las relaciones entre los parámetros formales y los parámetros actuales hay que estudiar el ámbito de las variables cuando se trabaja con subprogramas. Esto ayudará a entender mejor el comportamiento de los parámetros.

La parte del programa o algoritmo en que una variable se define se conoce como **ámbito**.

Las variables utilizadas en los programas principales y subprogramas se clasifican en dos tipos: variables locales y variables globales.

Una **variable local** es aquella que está declarada y definida dentro de un subprograma, siendo distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. El significado de estas variables se confina al procedimiento en el que está declarada. Cuando otro subprograma utiliza el mismo nombre se refiere a una posición diferente en memoria. Se dice que tales variables son locales al subprograma en el que están declaradas.

Una **variable global** es aquella que está declarada para el programa o algoritmo completo.

El uso de variables locales tiene muchas ventajas. En particular, hace a los subprogramas independientes, con la comunicación entre el programa principal y los subprogramas manipulada estructuralmente a través de la lista de parámetros. Para utilizar un procedimiento sólo necesitamos conocer lo que hace y no tenemos que estar preocupados por su diseño, es decir, cómo está programado.

Esta característica hace posible dividir grandes proyectos en piezas más pequeñas independientes. Cuando diferentes programadores están implicados, pueden trabajar independientemente.

Una variable local a un subprograma no tiene ningún significado en otros subprogramas. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros subprogramas, es decir, no pueden utilizar este valor. A veces, también es necesario que una variable tenga el mismo nombre en diferentes subprogramas.

Por el contrario, las variables globales tienen la ventaja de compartir información de diferentes subprogramas sin una correspondiente entrada en la lista de parámetros.

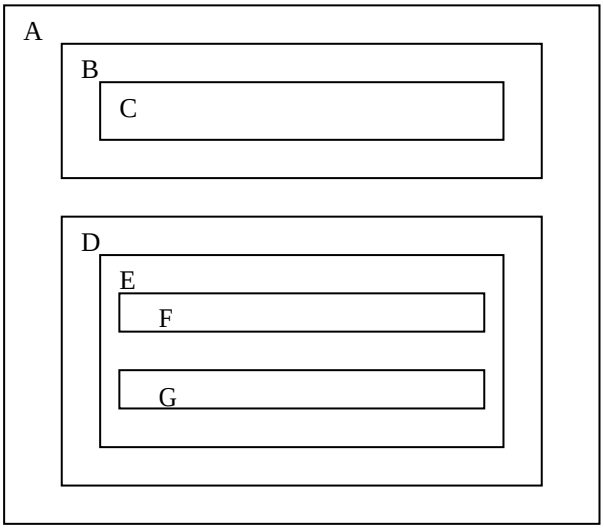
En un programa sencillo con un subprograma, cada variable u otro identificador es o bien local al procedimiento o global al programa completo. Sin embargo, si el programa incluye procedimientos que llaman a otros procedimientos (procedimientos anidados), entonces la noción de global y local es algo más complicado de entender.

El ámbito de un identificador (variables, constantes, procedimientos) es la parte del programa donde se conoce el identificador. Si un procedimiento está definido localmente a otro procedimiento, tendrá significado sólo dentro del ámbito de ese procedimiento. A las variables les sucede lo mismo; si están definidas localmente dentro de un procedimiento, su significado o uso se confina a cualquier función o procedimiento que pertenezca a esa definición.

Los lenguajes que admiten variables locales y globales suelen tener la posibilidad explícita de definir dichas variables como tales en el cuerpo del programa, o, lo que es lo mismo, definir su ámbito de actuación.

Las variables definidas en un ámbito son accesibles sólo en él, es decir, en todos los procedimientos interiores.

Ejemplo:



Las variables definidas en	*Son accesibles desde*
A	A, B, C, D, E, F, G
B	B, C
C	C
D	D, E, F, G
E	E, F, G
F	F
G	G

5. PASO DE PARÁMETROS

Cuando un programa llama a un subprograma, la información se comunica a través de la lista de parámetros y se establece una correspondencia automática entre los parámetros formales y los actuales. Los parámetros actuales son sustituidos o utilizados en lugar de los parámetros formales.

La **declaración de un subprograma** se hace de la forma:

```
Subprograma nombre (F1, F2, ..., Fn)
.
.
.
fin
```

Y la **llamada al subprograma** con:

```
Llamar_a nombre (A1, A2, ..., An)
```

donde F1, F2, ..., Fn son los parámetros formales y A1, A2, ..., An los parámetros actuales o reales.

Existen dos **métodos para establecer la correspondencia de parámetros**:

1. *Correspondencia posicional*. Se establece emparejando los parámetros reales y formales según su posición en las listas.
2. *Correspondencia por el nombre explícito*, también llamado método de paso de parámetros por nombre. En las llamadas se indica explícitamente la correspondencia entre parámetros reales y formales. Se utiliza en Ada.

Por lo general, la mayoría de los lenguajes usan exclusivamente la correspondencia posicional.
Los parámetros pueden ser clasificados como:

- *Entradas*. Proporcionan valores desde el programa que llama y se utilizan dentro de un procedimiento. En las funciones las entradas son los argumentos en el sentido tradicional.

- *Salidas*. Producen los resultados del subprograma. En el caso de una función, éste devuelve un valor calculado por dicha función, mientras que con procedimientos pueden calcularse cero, una o varias salidas.

- *Entradas/Salidas*. El mismo parámetro se utiliza para mandar argumentos a un subprograma y para devolver resultados.

Existen diferentes métodos para el paso de los parámetros a los subprogramas. Es preciso conocer el método adoptado por cada lenguaje, ya que la elección puede afectar a la semántica del lenguaje.

Paso por valor. Se utiliza en muchos lenguajes de programación; por ejemplo, Modula-2 y Pascal. La razón de su popularidad es la analogía con los argumentos de una función, donde los valores se proporcionan en el orden de cálculo de resultados. Los parámetros se tratan como variables locales y los valores iniciales se proporcionan copiando los valores de los correspondientes argumentos.

Los parámetros formales (locales a la función) reciben como valores iniciales los valores de los parámetros actuales y con ellos se ejecutan las acciones descritas en el subprograma.

Aunque el paso por valor es sencillo, tiene una limitación importante: no existe ninguna otra conexión con los parámetros actuales, entonces los cambios que se produzcan por efecto del subprograma no repercuten en los argumentos originales y, por consiguiente, no se pueden pasar valores de retorno al punto de llamada: todos los parámetros serán sólo de entrada. Los parámetros actuales no pueden modificarse por el subprograma. Cualquier cambio realizado en los valores de los parámetros formales durante la ejecución del subprograma se destruye cuando se termina.

La llamada por valor no devuelve información al programa que llama.

Paso por referencia. En numerosas ocasiones se requiere que ciertos parámetros sirvan de salida, es decir, que se devuelvan los resultados a la unidad o programa que llama. La unidad que llama pasa a la unidad llamada la dirección del parámetro actual (que está en el ámbito de la unidad llamante). Una referencia al correspondiente parámetro formal se trata como una referencia a la posición de memoria, cuya dirección se ha pasado. Entonces una variable pasada como parámetro será compartida, puede ser modificada directamente por el subprograma. Si el parámetro actual es una expresión, el subprograma recibe la dirección de la posición temporal que contiene el valor de la expresión.

El área de almacenamiento (direcciones de memoria) se utiliza para pasar información de entrada y/o salida; en ambas direcciones. Los parámetros son de entrada/salida y se denominan parámetros variables.

Ambos métodos de paso de parámetros se aplican tanto a la llamada de funciones como de procedimientos. Una función tiene la posibilidad de devolver los valores al programa principal de dos formas: como valor de la función y por medio de argumentos gobernados por la llamada de referencia. Un procedimiento sólo puede devolver valores por el método de devolución de resultados.

La diferencia entre paso por valor y paso por referencia se puede simplificar diciendo que, en el primer caso, los parámetros formales ocupan una zona de memoria distinta a la de los parámetros actuales reales. En el caso de paso por referencia, ambos parámetros ocupan la misma posición de memoria.

Cuando se declara una función hay que indicar si los parámetros formales van a soportar pasos por valor o pasos por referencia, no pudiéndose cambiar esta circunstancia durante la ejecución del programa.

La forma (sintaxis) de indicar si un parámetro soportará pasos por valor o por referencia dependerá del lenguaje de programación utilizado. En general, para pasos por valor, los parámetros formales se definen como variables comunes, sin variación en la sintaxis. Para pasos por referencia la sintaxis varía según el lenguaje.

6. RECURSIVIDAD

Se habla de recursividad cuando una función o procedimiento se llaman a sí mismos.

La recursividad es posible. De hecho, tener un procedimiento que se llame a sí mismo no es diferente, desde el punto de vista de su codificación, que tener un procedimiento que llame a otro. Cuando un procedimiento llama a otro, en primer lugar, el procedimiento que hace la llamada es suspendido, sus parámetros formales y variables locales son almacenados en la *pila del sistema*. A continuación, se almacena también en la pila la dirección de retorno (posición del código desde la que se ha hecho la llamada al procedimiento y a la cual ha de devolverse el control una vez finalice su ejecución). Finalmente, el control se pasa al código del procedimiento invocado. Cuando este procedimiento se termina de ejecutar, la dirección de retorno y las variables y parámetros formales se recuperan de la pila del sistema, y el control es devuelto al punto del código donde se realizó la llamada.

Esto mismo sucede cuando el procedimiento se llama a sí mismo. En una llamada recursiva, se almacenan nuevas copias de los parámetros formales y las variables locales del procedimiento en la pila. A continuación, el control se pasa nuevamente al inicio del procedimiento.

Si el procedimiento sigue llamándose a sí mismo de manera indefinida, llegará un momento en el que la pila aumente tanto que llegue a invadir alguna otra zona ocupada de la memoria y el sistema se venga abajo. Evidentemente, el aspecto más importante de la recursividad es *saber cuando debe detenerse el procedimiento*.

Un procedimiento recursivo ha de comprobar alguna condición antes de llamarse a sí mismo, así sabrá si aún necesita hacerlo. Esta condición podría ser la comparación de un contador frente a un número determinado de llamadas recursivas, o alguna condición booleana que se hace cierta (o falsa) cuando llega el momento de detener la llamada recursiva y volver hacia atrás.

Cuando se controla de este modo, la recursividad se convierte en una forma realmente potente y elegante de resolver algunos problemas de programación.

6.1. Aplicaciones de la recursividad

Algunos problemas de programación son muy adecuados para un tratamiento recursivo. Tal vez el más simple y mejor conocido sea el **cálculo de factoriales**. El factorial de un número es el producto de dicho número por todos los números menores que él hasta llegar a uno. **Ejemplo**: 5! = 5*4*3*2*1

Analizando el ejemplo, se observa que 5! es lo mismo que 5*4!, a su vez 4! es lo igual que 4*3!, y así sucesivamente. En general, se tiene que n! = n * (n-1)!.

Podemos entonces expresar la definición de función factorial de n como:

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

Es decir, se ha definido la función factorial en función de ella misma. El algoritmo que resolvería el cálculo de la función según esta definición sería:

```
Función Factorial (N     entero)     entero

    Si N > 1 Entonces
            Factorial      N * Factorial (N – 1)
    Sino
            Factorial      1
    Fin Si
```

Como se ve, se construye una función llamada Factorial que incluye una llamada a ella misma con un nuevo valor del parámetro N. Un subprograma que entre sus instrucciones tiene una llamada a sí mismo se dice que es recursivo. La recursión o recursividad es una herramienta especialmente potente en algunas aplicaciones.

Siempre es necesaria una sentencia condicional que indique al código cuándo debe detenerse la recursividad. Sin la comprobación de N > 1 la función seguiría decrementando la variable N por debajo de cero, realizando llamadas recursivas hasta que el sistema se viniese abajo.

6.2. Definiciones recursivas

Los algoritmos recursivos son apropiados principalmente cuando el problema a resolver, o la función a calcular, o la estructura de datos a procesar, están ya definidos en forma recursiva. A continuación se muestra otro enunciado definido en forma recursiva:

Números naturales:

(a) 1 es un número natural.
(b) El siguiente de un número natural es un número natural.

6.3. Recursividad directa e indirecta

Si un programa P contiene una referencia explícita a sí mismo se dice que es directamente recursivo; si P contiene una referencia a otro procedimiento Q que, a su vez contiene una referencia (directa o indirecta) a P, se dice que P es indirectamente recursivo.

```
{Programa directamente recursivo}

Procedimiento P(...)
...
Inicio
...
P(...)
...
Fin P
Inicio {programa principal}
...
P(...)
...
Fin
```

```
{Programa indirectamente recursivo}

Procedimiento P(...);
...
Inicio
...
Q(...)
...
Fin P
Procedimiento Q(...)
....
Inicio
...
P(...)
Fin Q
Inicio {programa principal}
...
P(...)
....
Fin
```

6.4. Características de la recursividad

De lo visto hasta ahora se pueden deducir las siguientes características de la recursividad:

1. La potencia de la recursión reside en la posibilidad de definir un número infinito de objetos mediante un enunciado finito, que en términos informáticos significa describir un número infinito de operaciones mediante un programa (recursivo) finito.

2. El instrumento necesario y suficiente para expresar los programas recursivamente es el procedimiento o subrutina, ya que permite dar un nombre a una instrucción por el cual ésta pueda ser llamada.

3. Para que un procedimiento o función recursiva esté bien definido tiene que cumplir dos características:

   a) debe existir un cierto criterio, llamado criterio base, por el que el procedimiento o función no se llame a sí mismo.

   b) cada vez que el procedimiento o función se llame a sí mismo (directa o indirectamente), debe estar más cerca del criterio base.

7. LIBRERÍAS

Una librería es un archivo en el que se encuentra almacenado el código correspondiente a una serie de funciones y/o procedimientos independientes entre sí. Aunque contienen código no tienen estructura de programa ejecutable y aunque se compilen o linken no son ejecutables de modo independiente, sino que la ejecución de sus funciones o procedimientos se realizará por la llamada de un programa. Las funciones y procedimientos contenidos en una librería se denominan **funciones externas** o **procedimientos externos**.

Un programa tiene un punto concreto (función o procedimiento principal) donde empieza la ejecución del mismo. En una librería no existe tal punto de inicio, sino que se puede llamar a cualquiera de sus funciones o procedimientos independientemente del lugar en que se haya definido dentro de la librería.

Un programa contiene un segmento de pila, que es un elemento clave, ya que, (junto al de código y al puntero de instrucción) define con precisión el flujo. Una librería compilada carece de estos elementos y sólo puede pedirlos prestados a la aplicación padre.

La utilidad de las librerías es doble. Por una parte, facilitar la reutilización de código y, por otra, liberar al programador de tareas de codificación tediosas o de bajo nivel. También facilitan la portabilidad de software cuando diferentes sistemas utilizan un mismo estándar de codificación.

Cuando un programador necesita utilizar una serie de funciones en distintos programas, no incluirá las funciones que necesite utilizar en todos o parte de ellos. Lo que creará es un fichero externo al programa en el que incluirá dichas funciones. Cuando en uno de los programas necesite utilizar alguna de las funciones sólo tendrá que realizar una llamada a la misma, sin codificar la función en el módulo que esté desarrollando. Lo que sí tendrá que indicar en el programa son las librerías de las funciones a las que vaya a hacer referencia. Estas librerías creadas con este fin por el programador se denominan **librerías de usuario**.

Para realizar tareas de bajo nivel (entrada y salida de datos, utilización de comunicaciones, etc.) el programado, además de conocer el hardware para el que está trabajando, necesitaría escribir un código complicado y largo. Para evitar esto, los fabricantes de compiladores suministran las funciones y procedimientos necesarios para ejecutar estas tareas. Todo este código suele ir suministrado en librerías, con lo que el programador sólo necesita conocer la librería y el nombre de la función correspondiente a ejecutar. Estas librerías se denominan **librerías estándar** y a las funciones contenidas en las mismas **funciones estándar** o **predefinidas**.

Independientemente si las librerías son de usuario o estándares, la utilización de las mismas es similar. Cuando en un programa se necesita utilizar una función de librería sólo será necesario realizar la llamada a la función. En algunos lenguajes, como en C, es necesario incluir ficheros de cabecera donde aparecen prototipos de las funciones, para garantizar la portabilidad a otros sistemas.

Las librerías se utilizarán de forma diferente en función de cómo han sido creadas. Según este criterio, las librerías las podemos agrupar en tres tipos fundamentales:

Librerías en código ASCII. En el programa fuente se indican las librerías a utilizar y su código es unido al programa cuando se compila. Por tanto, el módulo objeto resultante de la compilación tendrá el código correspondiente a la definición de las funciones. El inconveniente de este tipo de librerías es que se incluye la totalidad del código de las mismas, se utilicen todas sus funciones o no.

Librerías precompiladas. Son módulos objeto generados a partir del código fuente de la declaración y definición de las funciones. Estas son utilizadas cuando el programa es linkado. El linker extraerá de estas librerías sólo el código correspondiente a las funciones utilizadas, añadiéndolo al programa ejecutable. Una vez obtenido el programa ejecutable no será necesaria la presencia de las librerías para que el programa funcione. El inconveniente es que se generan programas de gran tamaño, lo que puede suponer una ocupación excesiva de la memoria del ordenador. Este sistema suele ser utilizado para programas que no han de compartir recursos con otras aplicaciones.

Librerías de enlace dinámico. Son ficheros con código ejecutable que mantienen su independencia física del programa principal. La extracción del código correspondiente a una función llamada por el programa principal se realiza en tiempo de ejecución. Es imprescindible la presencia de estas librerías para que el programa funcione adecuadamente. La utilidad principal es evitar programas ejecutables de tamaño excesivo, lo que podría provocar un desbordamiento de la memoria del ordenador. Cuando un programa realiza una llamada a una función de la librería sólo se carga en memoria el código de la función llamada, liberándose el espacio ocupado por ésta cuando se retorna al programa que llamó. Este sistema también es ventajoso cuando muchas aplicaciones utilizan de forma común un amplio grupo de funciones; el código de una función dada sólo aparece una vez en el soporte físico. El inconveniente principal es la pérdida de velocidad en la ejecución.

8. BIBLIOGRAFÍA

Pérez Lobato, J.M.
Metodología de la programación
Alhambra-Longman, 1994

Joyanes Aguilar, L.
Fundamentos de programación
Mc Graw-Hill, 1992

Alfonso Ureña López
Fundamentos de Informática
Ra-ma, 1997