Fundamentos de Programación I

4. Programación modular

Luís Rodríguez Baena (luis.rodriguez@upsam.net)

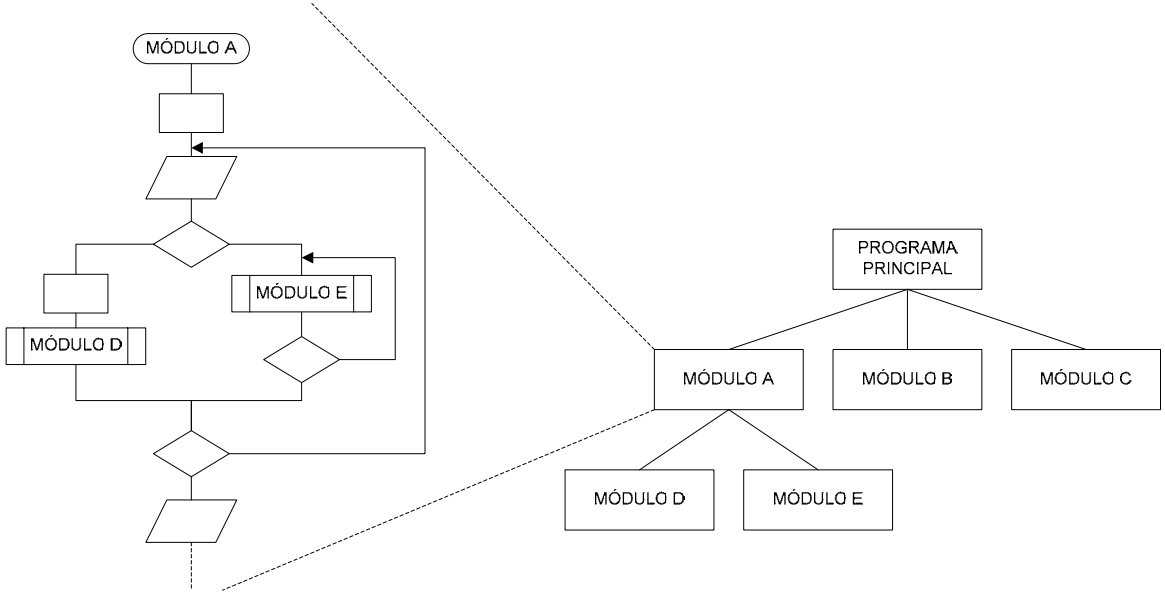
Universidad Pontificia de Salamanca (campus Madrid)
Escuela Superior de Ingeniería y Arquitectura

Introducción a la programación modular

- Es más fácil resolver un problema complejo cuando se divide en partes manejables: técnica de *divide y vencerás*.
  - En un programa monolítico la cantidad de variables utilizadas y caminos que debe seguir el flujo de control hace imposible su correcta comprensión.
    - ✓ Se dificulta la corrección de errores y el mantenimiento posterior del programa.
- La programación modular proporciona un método para plasmar el uso de recursos abstractos y la programación descendente.
- Consiste en descomponer un problema complejo en partes más pequeñas: módulos, subalgoritmos o subprogramas.
- Cada módulo sería un programa normal pensado para ser integrado en una aplicación mayor.

Introducción a la programación modular (II)

- Un programa modular estaría compuesto de:
  - Un programa principal, encargado de coordinar la ejecución.
  - Una serie de módulos que resolverían cada una de la tareas concretas del problema.



Introducción a la programación modular (III)

- Ventajas.
  - Facilidad para aprehender el problema.
  - División del trabajo entre un equipo de programadores.
    - ✓ Si los módulos son independientes, cada programador del equipo de desarrollo puede encargarse de uno.
    - ✓ El jefe del proyecto integrará los distintos módulos en la aplicación principal.
  - Facilidad de mantenimiento y corrección de errores.
    - ✓ Si cada módulo cumple una tarea completa es más fácil detectar donde se produce un error.
    - ✓ Si se necesita realizar una mejora, sólo habrá que modificar un módulo.
  - Reutilización del código.
    - ✓ Un módulo que realice una tarea determinada podrá utilizarse en otro programa que precise de la misma tarea.

Criterios de descomposición modular

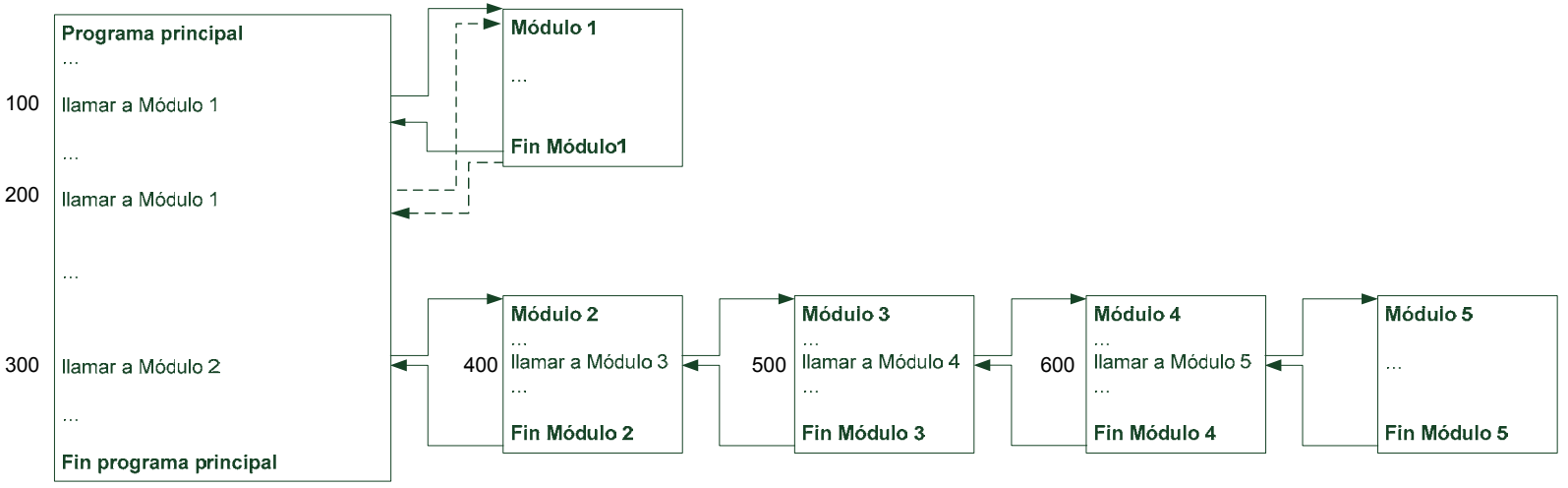
- Es necesario un compromiso entre el tamaño de los módulos y la complejidad de la aplicación.
  - Si un programa se descompone en demasiadas unidades, decrece la efectividad.
    - ✓ Cuando el número de módulos se incrementa, decrece el esfuerzo para realizarlos, pero aumenta el esfuerzo de integración y la carga en memoria.
- Algunos criterios de descomposición (no válidos).
  - Descomposición por tamaño (50 líneas por módulo).
  - Complejidad del módulo: niveles de anidamiento (menos de 7 niveles).
- Independencia funcional.
  - Un módulo debe realizar una única tarea y comunicarse lo menos posible con el resto de módulos.

Criterios de descomposición modular (II)

- Un módulo se debe dividir hasta que se consiga un nivel mínimo aceptable de independencia funcional.
- La independencia funcional se puede medir según dos criterios:
  - Cohesión.
    - ✓ Mide la relación entre las partes internas de un módulo.
    - ✓ Todas deben estar encaminadas a realizar una única función.
  - Acoplamiento.
    - ✓ Mide la relación del módulo con el resto de los módulos.
      - Debe comunicarse lo menos posible.
      - Pocas veces se conseguirá un acoplamiento nulo.
- Un módulo debe tener mucha cohesión y poco acoplamiento.

Llamadas a módulos

- Un programa modular contará con un programa principal y uno o varios módulos.
- El programa principal llama o invoca a los módulos, cediendo a éstos el control del flujo del programa.



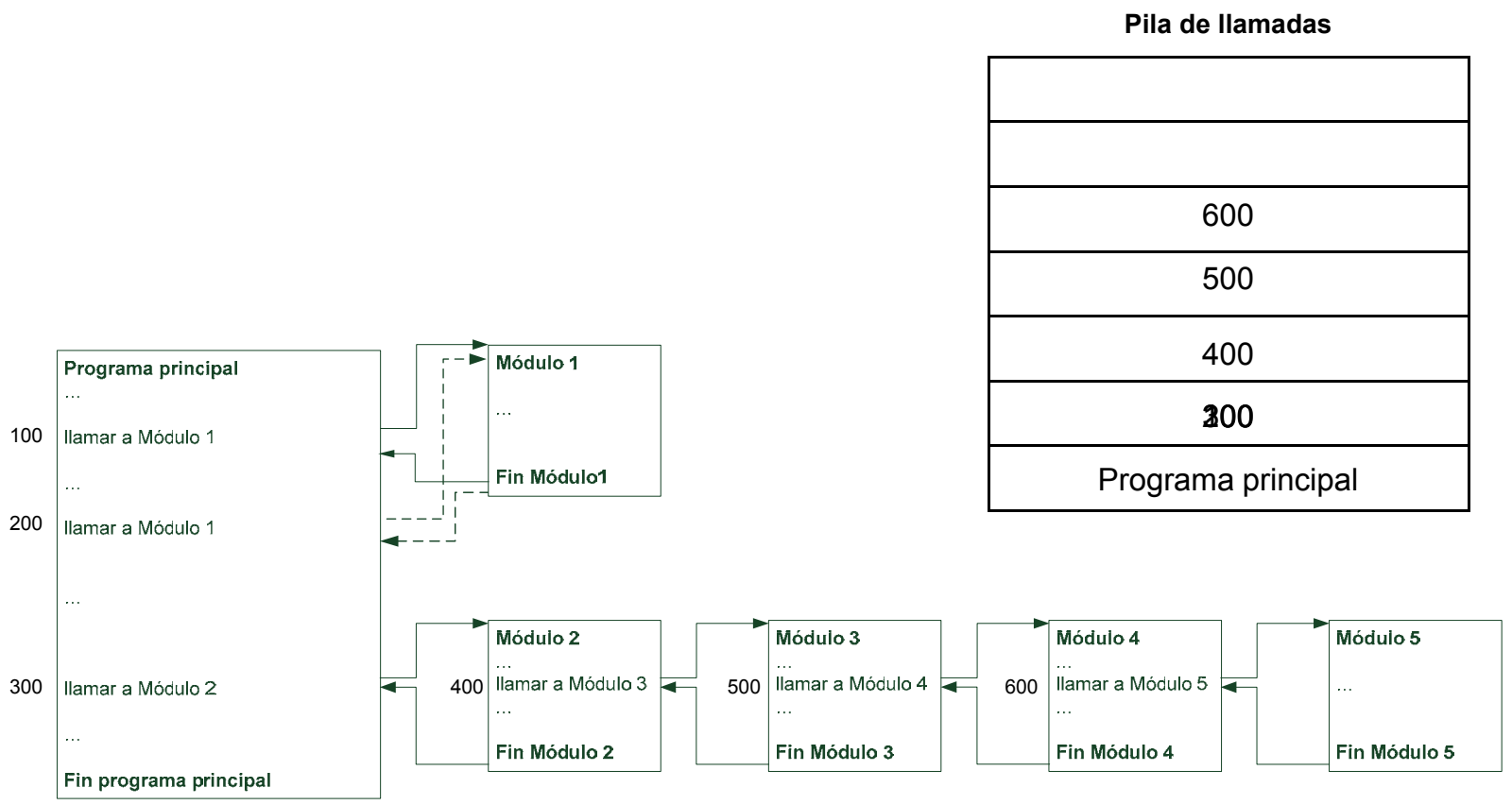
Llamadas a módulos (II)

- Cuando se hace la llamada a un módulo se transfiere el control a la primera línea del módulo llamado.
  - El programa llamador queda en suspenso hasta que termina el módulo llamado.
    - ✓ En algunos lenguajes no estructurados (COBOL) es posible entrar por otra línea.
- El módulo llamado carga en memoria su código y un espacio para sus propias variables y estructuras.
- Cuando el programa llamado termina, transfiere el control a la instrucción siguiente a la llamada.
  - Se pierde la información generada por el módulo a no se que se produzca un intercambio de información entre el programa llamador y el llamado.

Llamadas a módulos (III)

- Durante su ejecución el programa llamado actúa como si se tratara del programa principal.
  - Puede realizar llamadas a otros módulos que funcionan de la misma forma antes expuesta.
  - El posible realizar llamadas a otros módulos: llamadas anidadas.
  - Es posible llamarse a si mismo (recursividad).
- La dirección de la línea donde se realiza la llamada y el estado de las variables del programa llamador se almacena en una estructura de tipo pila: la pila de llamadas (*call stack*).
  - Los datos salen de forma inversa a como han entrado.
  - Cuando acaba un módulo, el flujo de control retorna a la dirección de memoria almacenada en la parte superior de la pila: la ultima que ha entrado.
  - Se retorna en orden inverso al orden de llamada.
  - Cuando termina un programa y la pila de llamadas está vacía, el control retorna el sistema operativo.

Llamadas a módulos (IV)



Tipos de subprogramas

- Procedimientos.
  - Ejecutan una acción que puede o no modificar valores o puede o no necesitar datos de entrada.
    - ✓ ImprimirFactura.
    - ✓ OrdenarLista.
  - Su llamada sustituye a una tarea.
- Funciones.
  - Realizan una operación que siempre tiene como resultado un valor.
    - ✓ Factorial.
    - ✓ Potencia.
    - ✓ Máximo.
  - Su llamada sustituye a un valor.
- Algunos lenguajes hacen esta distinción entre subprogramas.
  - Procedure y Function en Pascal, procedimientos Sub y procedimientos Function en VB.NET
- En algunos lenguajes no hacen esa distinción
  - En C o Java, a todo se le denomina función, aunque algunas no devuelven nada (funciones void).

Funciones

- Realizan una tarea que siempre devuelve un valor asociado a la propia llamada.
  - Cuando se detecta la llamada a una función, el control del programa se transfiera a la función.
  - Cuando la función termina se retorna y la llamada es sustituida por el valor que devuelve.
- En la instrucción de asignación `y ← 1 + sen(45)`...
  - Se evalúa la expresión de izquierda a derecha.
  - Antes de evaluar la expresión, el control se transfiera a la llamada a la función `sen()` con el valor 45.
  - La función se ejecuta, termina y devuelve el valor del seno de 45 (0,7071) que sustituye a la llamada.
  - La expresión evaluada (`1+0.7071`) se asigna a la variable `y`.

Funciones (II)

- Funciones internas (predefinidas o intrínsecas) del lenguaje algorítmico UPSAM 2.0.

Función	Significado	Función en C
abs(x)	Devuelve el valor absoluto de la expresión numérica x	fabs() en el archivo de cabecera maths.h
aleatorio( )	Devuelve un número aleatorio real mayor o igual que 0 y menor que 1	rand() y srand() en al archivo de cabecera stdlib.h
cos(x)	Devuelve el coseno de x	cos() en el archivo de cabecera maths.h
entero(x)	Devuelve el primer valor entero menor que la expresión numérica x	ceil(), redondea al entero mayor más cercano y floor(), redondea al entero menor más cercano, en al archivo de cabecera math.h
exp(x)	Devuelve el valor	exp() en el archivo de cabecera maths.h
log(x)	Devuelve el logaritmo neperiano de x.	log() en el archivo de cabecera maths.h
log10(x)	Devuelve el logaritmo en base 10 de x.	log10() en el archivo de cabecera maths.h
raiz2(x)	Devuelve la raiz cuadrada de x	sqrt() en el archivo de cabecera maths.h
sen(x)	Devuelve el seno de x	sin() en el archivo de cabecera maths.h
tan(x)	Devuelve la tangente de x	tan() en el archivo de cabecera maths.h
trunc(x)	Trunca (elimina los decimales) de la expresión numérica x.	

Declaración de funciones

```
tipoDeDato función NombreDeFunción([listaDeParámetrosFormales])
[//Declaraciones locales de tipos de datos, constantes o variables]
inicio
   //Código de la función
   devolver(expresión)
fin_función
```

- Una función siempre devuelve un tipo de dato que habrá que indicar en la cabecera de la función.
- Es necesario un identificador único que identifique la función.
- Es posible que sea necesario pasar información a la función (por ejemplo el 45 de la función seno anterior).
  - Serían los argumentos o parámetros formales con los que realizará las operaciones.
  - De momento por cada argumento se indicará el tipo de dato y el nombre.

Declaración de funciones (II)

- Es posible que sea necesario utilizar datos auxiliares para ejecutar la función.
  - Para ello se utilizarán las declaraciones locales.
  - Las declaraciones de variables se realizarán de la misma forma que en un algoritmo principal.
- Entre las palabras reservadas **`inicio`** y **`fin_función`** se colocará el código de la función.
- Una función siempre devuelve un valor.
  - La expresión de la instrucción **`devolver`**`()`, indicará el valor que devuelve.

Ejemplo 4.1.

- Declarar una función que calcule el factorial de un número entero positivo que se pasará como argumento de la función.
  - ***Análisis del problema***
    - ✓ Se deberá proporcionar a la función un dato sobre el que vamos a sacar el factorial: el argumento de tipo entero n.
    - ✓ Para el cálculo del factorial habrá que acumular las multiplicaciones de todos los números entre 1 y n, por lo que será necesario un acumulador de multiplicaciones (variable fact) que habrá que inicializar al elemento neutro de la multiplicación.
    - ✓ Por último será necesario realizar un bucle en el que una variable vaya tomando los valores entre 2 y n (hay que recordar que factorial de 0 es 1 y factorial de 1 también es 1).
    - ✓ Se utilizará una estructura de tipo desde. El valor de retorno de la función será el del acumulador fact.

```
entero función Factorial(entero : n)
var
   entero: fact, i
inicio
   fact ← 1
   desde i ← 2 hasta n hacer
      fact ← fact * i
   fin_desde
   devolver(fact)
fin_función
```

Llamadas a funciones

- Formato de la llamada:

  NombreFunción([*ListaParámetrosActuales*])

  - La lista de parámetros actuales son los valores reales con los que trabajará la función.
    - ✓ Al realizar la llamada el valor de los parámetros actuales sustituye a los parámetros formales a la hora de ejecutar el código.

```
Factorial(3)   //Realiza la llamada a la función con n = 3
Factorial(a)  //Evalúa el contenido de la variable a
               //y realiza la llamada con n = a
Factorial(a+5)//Evalúa el contenido de la expresión a+5
               // y realiza la llamada con n = a+5
```

  - Una función devuelve un valor, por lo que la llamada se deberá utilizar en una expresión que requiera un valor del tipo de dato devuelto por la función.

```
escribir(Factorial(3)
a ← x * Factorial(y)
b ← Factorial(Factorial(b))
```

Ejemplo 4.2.

El seno de un ángulo x se puede calcular por la siguiente serie:

$$sen(x) = \sum_{n=0}^{\infty} \frac{(-1)}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \pm \frac{x^n}{n!}$$

Cuanto mayor sea el número de términos de la serie, mayor será la precisión del cálculo.

Implemente una función seno que permita calcular el seno de un ángulo x expresado en radianes utilizando dicha serie. El cálculo terminará cuando la diferencia entre dos términos correlativos sea menor que 10^{-3}.

- ***Análisis del problema***
  - ✓ Cada término de la serie se calculará mediante la función factorial declarada en el ejercicio 5.1. En cada término, el exponente y el factorial van tomando valores 1,3,5,7…, por lo que será necesario un contador que, a partir de 1, incremente los valores de dos en dos.
  - ✓ En la serie, el primer término se suma, el segundo se resta, el tercero se suma, etc., es decir, se va alternado la suma y la resta. Para solucionar esto se utiliza una variable lógica (sumar) que cambia de estado en cada iteración. Si la variable es cierta se sumará el término a la serie, en caso contrario se resta.
  - ✓ Además es necesario guardar el último y el penúltimo término para controlar la salida

Ejemplo 4.2. (II)

```
real función Seno(entero: x)
var
   real : suma, último, penúltimo
   entero : i
   lógico : sumar //si es cierto, suma el término, en caso contrario
                  //lo resta
inicio
   suma ← x
   i ← 1
   último ← x
   repetir
      i ← i + 2
      penúltimo ← último
      último ← x ** i / factorial(i)
      //Si sumar es verdad, suma el término x**i/Factorial(i)
      si sumar entonces
         suma ← suma + último
      si_no
         suma ← suma - último
      fin_si
      //En cada iteración se cambia el estado de la variable sumar
      sumar ← no sumar
   hasta_que penúltimo – último < 0.001
   devolver(suma)
fin_función
```

Procedimientos

- Su llamada sustituye a una acción.
- Módulo que realiza una acción específica que puede devolver 0, 1 o n valores.
  - Dentro del procedimiento se pueden modificar algunos de los valores del programa llamador pero no devuelve ningún valor asociado a la llamada.
- Declaración.

```
procedimiento NombreDeProcedimiento([listaDeParámetrosFormales])
[//Declaraciones locales de tipos de datos, constantes o variables]
inicio
   //Código del procedimiento
fin_procedimiento
```

  - *NombreProcedimiento* es un identificador único que referencia el procedimiento.
  - La *listaDeParámetrosFormales* es una lista con los argumentos con los que se ejecutará el procedimiento.
    - ✓ Algunos podrán devolver valores al programa llamador.
  - Si se necesita, se pueden declarar tipos de datos y variables auxiliares que se precisen dentro de las declaraciones locales.
  - El código del procedimiento se realizará de la misma forma que en un programa principal.

Ejemplo 4.3.

Diseñe un módulo que permita escribir por pantalla un número n que se pasará como argumento al revés.

- ***Análisis del problema***
  - ✓ El módulo deseado simplemente realiza una acción (escribir un número al revés) y no necesita devolver o modificar ningún dato del programa principal. Por lo tanto el tipo de módulo apropiado para realizarlos será un procedimiento.
  - ✓ Para poder escribir un número al revés, tenemos que utilizar los operadores **div** y **mod**. El resto de dividir un número entre 10 será el último dígito, las unidades. Si modificamos el número para eliminarle las unidades (n **div** 10) el siguiente resto nos daría las decenas, etc. El proceso se tiene que repetir hasta que el número sea menor que 10.

```
procedimiento EscribirNúmeroAlRevés(entero : n)
inicio
   mientras n >= 10 hacer
      escribir(n mod 10)
      n ← n div 10
   fin_mientras
fin_procedimiento
```

Llamadas a procedimientos

- Formato de la llamada:

  `[`**`llamar_a`**`]`*`NombreProcedimiento`*`([`*`listaDeParámetrosActuales`*`])`

  - Algunos lenguajes precisan de una orden especial para realizar la llamada (`call` o similar).
- Proceso de la llamada.
  - La llamada transfiere el control a la primera línea del procedimiento.
  - Se carga el procedimiento en memoria, se crea las variables locales que se precisen y se sustituyen los parámetros formales por los actuales.
  - Se ejecuta el código del procedimiento.
  - Al finalizar el control vuelve al programa llamador.
    - ✓ En algunos casos, si se han modificado los valores de los parámetros actuales, ese cambio se refleja en el programa llamador.
- Al sustituir una acción la llamada a un procedimiento sustituye a una instrucción.
  - La llamada `EscribirNúmeroAlreves(2345)` produce el mismo efecto que insertar en ese punto las instrucciones del procedimiento haciendo que n sea igual a 2345.

Visibilidad o ámbito de las variables

- No todos los datos se pueden utilizar desde todos los puntos del algoritmo.
  - El ámbito o visibilidad de una variable sería el lugar donde puede ser utilizadas.
- Según su ámbito las declaraciones de un algoritmo pueden ser:
  - Declaraciones globales.
    - ✓ Afectan a todo el programa.
    - ✓ Se almacenan en una zona de memoria común, accesible desde cualquier punto del programa y que se mantiene mientras dura el programa.
    - ✓ Son las declaraciones del programa principal.
  - Declaraciones locales.
    - ✓ Sólo se pueden utilizar en el módulo (procedimiento o función) donde han sido declaradas.
    - ✓ Se almacenan en una zona de memoria que se crea cuando se llama al módulo, que sólo es accesible desde ese módulo y que se destruye cuando termina la ejecución del módulo.

Visibilidad o ámbito de las variables (II)

- Conflictos en los nombres de variables.
  - En un algoritmo pueden coexistir variables distintas con el mismo nombre, siempre que tengan un ámbito distinto.
    - ✓ Cada módulo almacena las variables en lugares de memorias distintos entre sí y distintos del programa principal.
  - Resolución de conflictos.
    - ✓ La prioridad la tiene la declaración más local.
- ¿Por qué son necesarias las declaraciones locales?
  - En la gran mayoría de las ocasiones es posible utilizar variables globales.
  - Si sólo se utilizan variables globales, el programa que utiliza el procedimiento está obligado a utilizar los mismos nombres de variables que el procedimiento.
  - Un algoritmo así diseñado tendría mucho acoplamiento.
    - ✓ Depende mucho de la información del resto de los componentes del algoritmo.
    - ✓ Pierde independencia funcional.

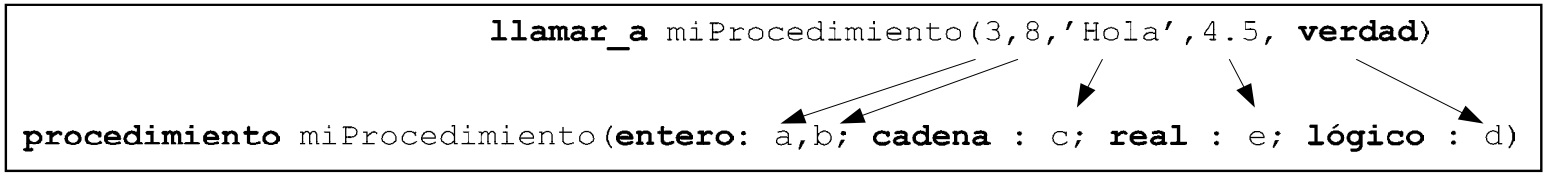
Visibilidad o ámbito de las variables (III)

```
...
var
   entero: n, fact,i
inicio
   leer(n)
   escribir(factorial(n))
   ...
entero función Factorial()
inicio
   fact ← 1
   desde i ← 2 hasta n hacer
      fact ← fact * i
   fin_desde
   devolver(fact)
fin_función
```

- Esta función Factorial tendría mucho acoplamiento.
  - Necesitaría que siempre el programa factorial utilizara las variables enteras `n` , `i` y `factorial`.

Intercambio de información: paso de parámetros

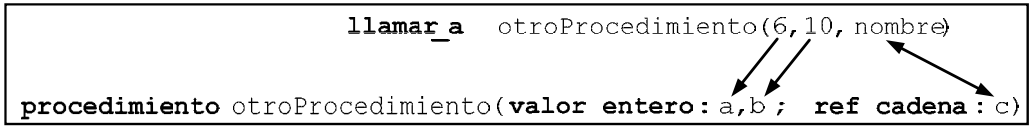
- Los argumentos de un procedimiento o función se utilizan para intercambiar información entre el módulo y el programa llamador.
  - La lista de parámetros actuales sustituye a la lista de parámetros formales.
    - ✓ La lista de parámetros actuales está compuesta de una lista de variables o expresiones separadas por comas.
    - ✓ La lista de parámetros actuales está compuesta por grupos de argumentos del mismo tipo.
      - Los argumentos se separan por comas.
      - Los grupos por punto y coma.
- La sustitución se realiza por su posición.



- Los parámetros de ambas listas deben coincidir en número posición y tipo de dato.

Intercambio de información: paso de parámetros (II)

- En algunos casos el intercambio de información se realizará sólo desde el programa llamador al módulo.
  - Se trata de argumentos de entrada.
  - Se señalan utilizando una **`E`** o la palabra **`valor`** antes del tipo del grupo de argumentos.
- Otras veces es preciso realizar un intercambio bidireccional.
  - Se trata de argumentos de entrada y salida.
  - Se señalan utilizando una **`E/S`** o la palabra **`ref`** antes del tipo del grupo de argumentos.



Paso de argumentos

- Por cada tipo de argumento los parámetros formales se definen:

  {**E**|**S**|**E/S**|**valor**|**ref**} *tipoDato* : listaIdentificadores ; …

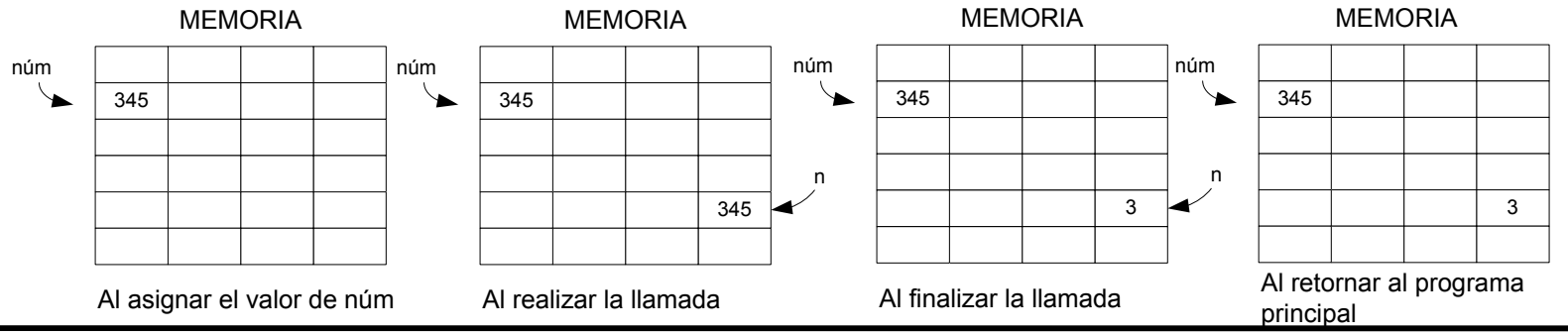
  - **E**, **S** o **E/S** (**valor** o **ref**) define el tipo de paso de argumentos.
  - *tipoDato* es un dato ya declarado (estándar o definido por el usuario).
  - *listaIdentificadores* es el nombre de los argumentos separados por comas.
  - Puede haber distintos tipos de argumentos, cada tipo se separa por punto y coma.
- Según se trate de argumentos de entrada o de entrada salida los argumentos se pasarán:
  - Por valor.
  - Por referencia.

Paso por valor

- Argumentos de entrada.
- Modo de funcionamiento:
  - Al realizar la llamada se evalúan los valores de la lista de parámetros actuales.
  - Se envía una copia de dichos valores al subprograma.
  - La copia de los parámetros actuales sustituye a los parámetros formales por posición.
  - Los parámetros formales actúan como variables locales.
    - ✓ Se reserva una zona de la memoria para almacenar el valor de la copia que se ha pasado y se la identifica con el nombre del parámetro.
  - Las modificaciones que se realicen en el subprograma se hacen sobre la copia.
  - Se realiza una copia del valor de la variable en otra variable que tiene el nombre del argumento.
  - Al finalizar el subprograma la referencia a esa zona de la memoria local desaparece.
    - ✓ El valor del parámetro actual en el programa llamador permanece inalterado puesto que lo que se ha modificado es la copia.
- Cómo lo que se manda es una copia de un valor, los parámetros actuales pueden ser expresiones.

Paso por valor (II)

```
algoritmo PasoPorValor
var
   entero : núm
inicio
   núm ←345
   EscribirNúmeroAlRevés(núm)
   …
Fin
Procedimiento   EscribirNúmeroAlRevés(valor entero: n)
inicio
   mientras n >= 10 hacer
      escribir(n mod 10)
      n ← n div 10
   fin_mientras
fin_procedimiento
```



Ejemplo 4.4.

Diseñe una función lógica que indique si una fecha es válida. Se pasará a la función tres números enteros con el día, el mes y el año.

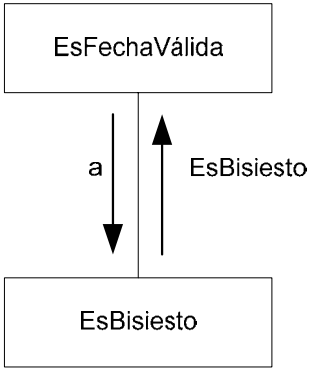
- ***Análisis del problema***

  Habrá que comprobar si el mes está entre 1 y 12 y el día entre 1 y 31. Si esto es cierto habrá que comprobar que si el mes es de 30 días, el día no sea mayor que 30. Si se trata de febrero habrá que comprobar si es un año bisiesto y si es así comprobar que el día no sea mayor que 29 o, en caso contrario, mayor que 28.

  La solución propuesta utilizará un *switch* o interruptor para comprobar la fecha. Se trata de una variable lógica (`válida`) que, al final del proceso, servirá para saber si la fecha es correcta. Inicialmente se le dará un valor verdadero y en el caso de que se produzca algún error tomará el valor falso.

  La solución propuesta implementará también una función que indique si un número corresponde a un año bisiesto.

Ejemplo 4.4. (II)




```
lógico función EsBisiesto(valor entero: a)
inicio
   devolver((a mod 4 = 0) y ((a mod 100 <> 0) o (a mod 400 = 0)))
fin_función
lógico función EsFechaVálida(valor entero: año, mes, día)
var
   lógico : válida
inicio
   válida ← verdad
   si (dia > 31) o (día > 1) o (día > 12) o (día < 1) entonces
      válida ← falso
   si_no
      según_sea mes hacer
          4, 6, 9, 11 : válida ← día <= 30
          2 : si EsBisiesto(año) entonces
                 válida ← día <= 29
              si_no
                 válida ← dia <= 28
              fin_si
      fin_según
   fin_si
   devolver(válida)
fin_función
```

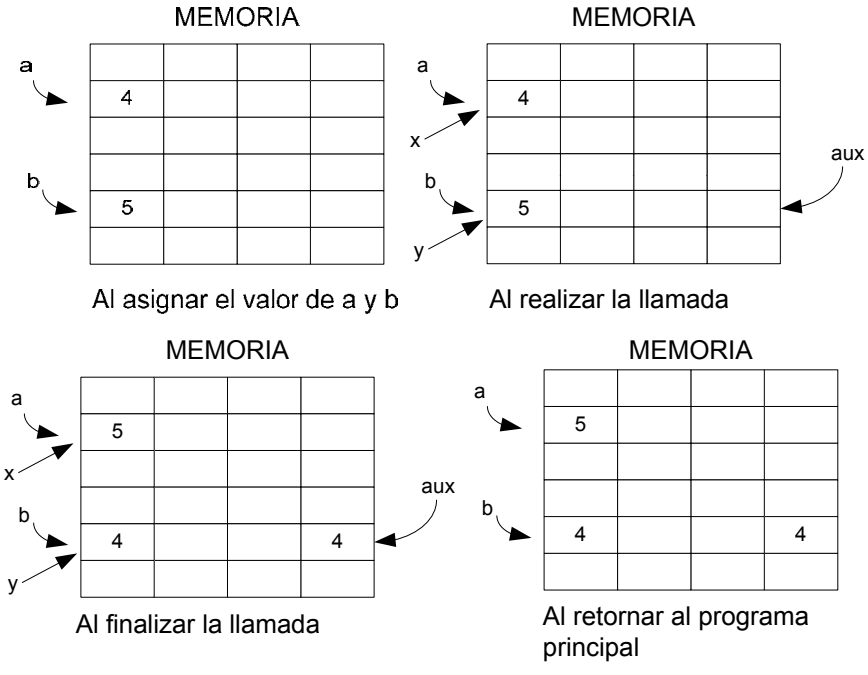
Paso por referencia

- También se llama paso por variable.
- Argumentos de entrada o entrada/salida.
- No se pasa una copia del valor, sino la referencia a la zona de la memoria donde se almacena el parámetro actual.
  - En este caso el parámetro actual y el formal hacen referencia a la misma zona de memoria.
- Durante la ejecución del programa se referencia esa zona de memoria con el nombre de los parámetros formales.
- Cualquier modificación que se haga en esa zona de memoria afectará a la misma región de memoria donde se almacenan los parámetros actuales.
- Como no se pasa un valor sino una referencia a una zona de memoria, es necesario pasar algo que se almacene en memoria:
  - Se pasan variables, no expresiones.

Paso por referencia (II)

```
algoritmo PasoPorReferencia
var
   entero : a, b
inicio
   a ← 4
   b ← 5
   Intercambia(a,b)
   …
fin

//Intercambia el contenido de dos variables
procedimiento Intercambia(ref entero : x,y)
var
   entero : aux
inicio
   aux ← x
   x ← y
   y ← aux
fin_procedimiento
```



Ejemplo 4.5.

Diseñe un subprograma que reciba la posición de un punto en coordenadas polares (radio y ángulo) y devuelva su posición en coordenadas cartesianas (x,y). Para pasar de coordenadas cartesianas a polares se utilizan las siguientes fórmulas.

x = radio * cos(a)

y = radio * sen(a)

- ***Análisis del problema***

  El módulo a utilizar debería ser un procedimiento, puesto que deberá devolver más de un valor y es imposible devolver los valores de x y de y en la propia llamada.

  La función recibirá dos argumentos reales que se pasarán por valor para el radio y el ángulo. También tendrá otros dos argumentos reales pasados por referencia para los valores de x e y.

```
procedimiento DeCPolaresACartesianas(valor real : r,a; ref real: x,y)
inicio
   x ← r * cos(a)
   y ← r * sen(a)
fin_procedimiento
```

Efectos laterales

- Un efecto lateral es el efecto producido por la modificación de una variable global o incluso de un argumento directamente en un procedimiento o función.
  - Puede que en algunos casos sea beneficiosos.
    - ✓ Es la única forma en que los procedimientos pueden devolver valores a un programa.
    - ✓ En ocasiones, cuando se pasa una estructura de datos muy grande, utilizar variables globales directamente puede ahorra espacio y tiempo de ejecución.
  - Pero se deben evitar.
- Para disminuir el acoplamiento de los módulos todo intercambio de información se debe realizar por medio de parámetros.
  - Aunque se pueda realizar el intercambio por medio de variables globales, esto aumentará el acoplamiento.
- Norma:
  - No modificar variables globales dentro de los módulos.
  - Si se desea modificar variables globales se debe utilizar el paso por referencia.

Ejercicios

- Diseñe una función que calcule el valor absoluto de un número entero.
- Suponiendo que nuestro lenguaje de programación no dispone de los operadores de división entera y resto, diseñe un módulo que permita obtener la división entera y el resto de dos números enteros positivos mediante restas sucesivas. Justifique la elección del tipo de módulo utilizado.
- Suponiendo que nuestro lenguaje de programación no dispone del operador de exponenciación, diseñe un módulo que devuelva el resultado de x^y, siendo x un número real e y un número entero. Justifique la elección del tipo de módulo utilizado.
- El número Π se puede calcular mediante la serie

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$$

  Calcule el valor del número hasta que la diferencia entre dos términos sea menor a 10^{-15}.

Ejercicios (II)

- El coseno de un ángulo x se puede calcular mediante la serie

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

  Diseñe una función que permita calcular el valor del seno de x hata que la diferencia entre dos términos sea menor que 10^{-3}.
- Diseñe una función que devuelva la suma de divisores de un número pasado como argumento. Por ejemplo, si el argumento pasado es 24, deberá devolver 1+2+3+4+6+8+12+24.
- Diseñe una función lógica que indique si un número es perfecto. Un número es perfecto si es igual a la suma de sus divisores. Por ejemplo, 6 es un número perfecto ya que 6 = 1+2+3.

Ejercicios (III)

- Diseñe una función lógica que indique si dos números son amigos. Dos números son amigos si cada uno es igual a la suma de divisores del otro excepto el propio número. Por ejemplo, 284 y 220 son amigos:

  Divisores de 284 1, 2, 4, 71, 142 y 284.
  Suma de divisores excepto 284 = 220.

  Divisores de 220: 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110 y 220.
  Suma de divisores excepto 220 = 284

  Diseñe además un programa principal que saque por pantalla todas las parejas de números amigos entre 1 y n.

Ejercicios (IV)

- Diseñe una función que devuelva el máximo común divisor por el algoritmo de Euclides.
  - (32 y 6) = 2
  - 1428 y 316
    - ✓ 1428 mod 316 = 164
    - ✓ 316 mod 164 = 152
    - ✓ 164 mod 152 = 2
    - ✓ 152 mod 2 = 0
- Codifique **un procedimiento** permita leer por teclado una serie de caracteres. La lectura terminará cuando el carácter introducido sea '0' (cero). El procedimiento deberá devolver al programa que lo llamó el número de vocales que se introdujeron. Por ejemplo, si la serie de caracteres introducida es 'a', 'x', 'h', 'e', 'a', 'v', '0', el procedimiento deberá devolver el valor 3.
- Dada una fecha expresada en día, mes y año, escriba una función que devuelva el número de días transcurrido desde el comienzo del año.
- Dada una fecha posterior al 1 de enero de 1980, escriba un módulo que devuelva el día de la semana sabiendo que el 1 de enero de 1980 fue martes.

Ejercicios (V)

- Diseñe una función que reciba una hora expresada en horas, minutos y segundos y devuelva el número de segundos totales.
- Diseñe un subprograma que reciba un número de segundos transcurridos y devuelva el número de horas, minutos y segundos que representan.
  - Hh = segTotales div 3600
  - Mm = segTotales mod 3600 div 60
  - Ss = segTotales mod 3600 mod 60
- Diseñe un subprograma que reciba dos horas expresadas en horas, minutos y segundos y devuelva otra hora que represente la suma de ambas.