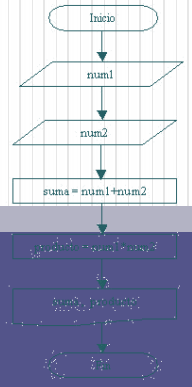
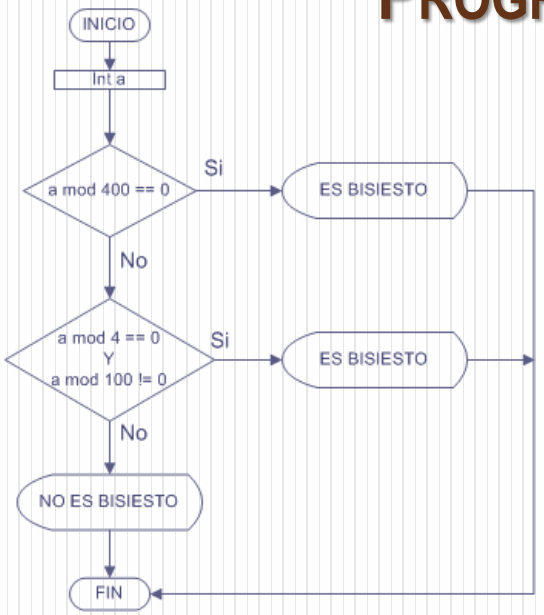


Analista Programador Universitario

Programación Estructurada

Programación Modular



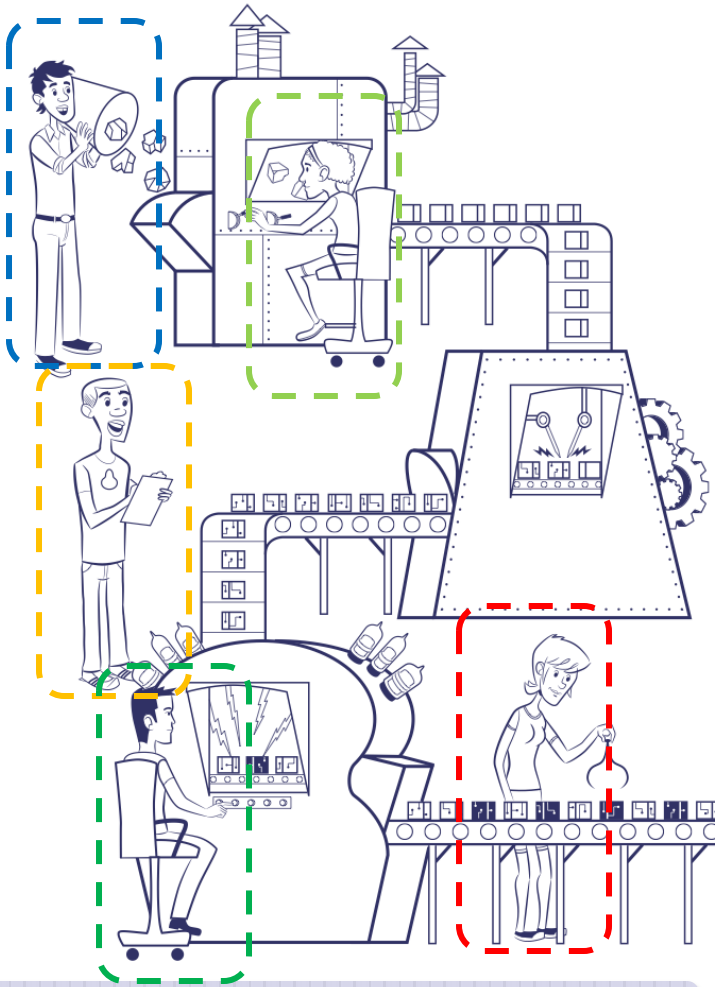
Facultad de Ingeniería
Universidad Nacional de Jujuy

Índice

- Programación Modular
- Funciones
- Procedimientos
- Comunicación entre módulos
  - Pasaje de Parámetros
- Variables Locales y Globales
- Ocultamiento y Protección
- Recursividad

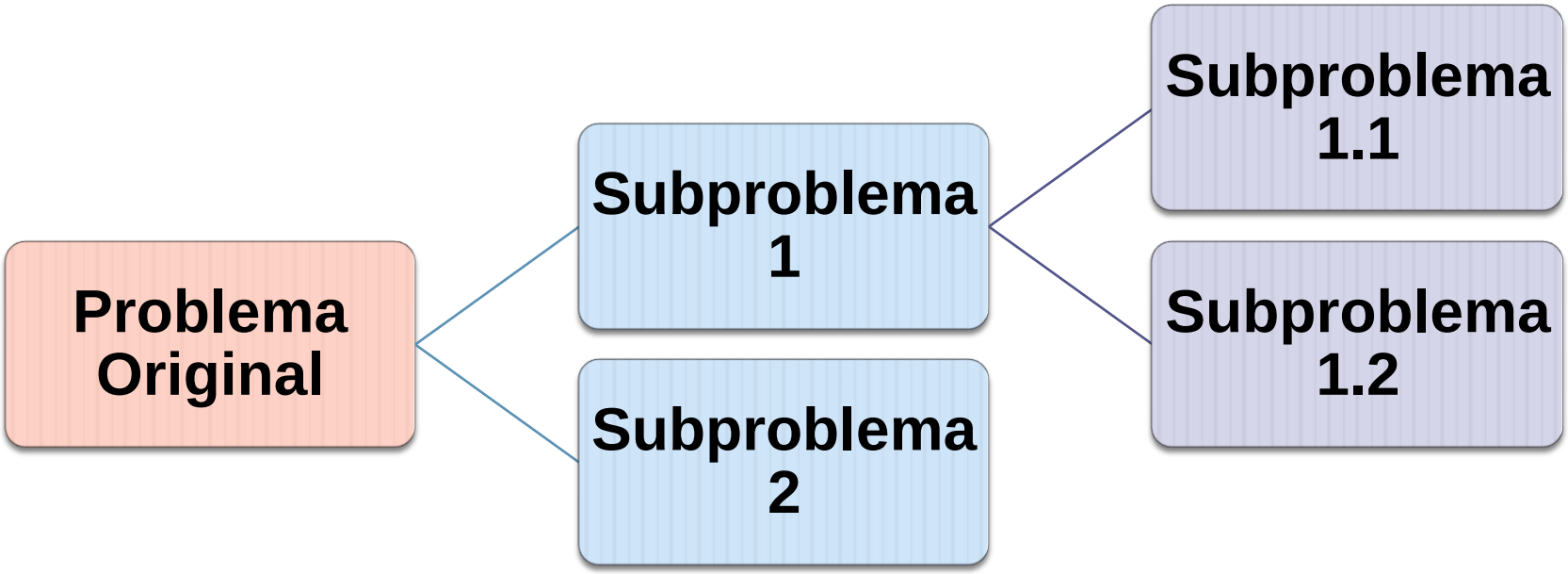
Complejidad de un Problema

- Problemas simples pueden ser tratados por programas con un pequeño número de instrucciones.
- Problemas complejos deben ser tratados por programas con un gran número de instrucciones.
- Para reducir la complejidad de un programa, éste puede dividirse en varias unidades de trabajo.



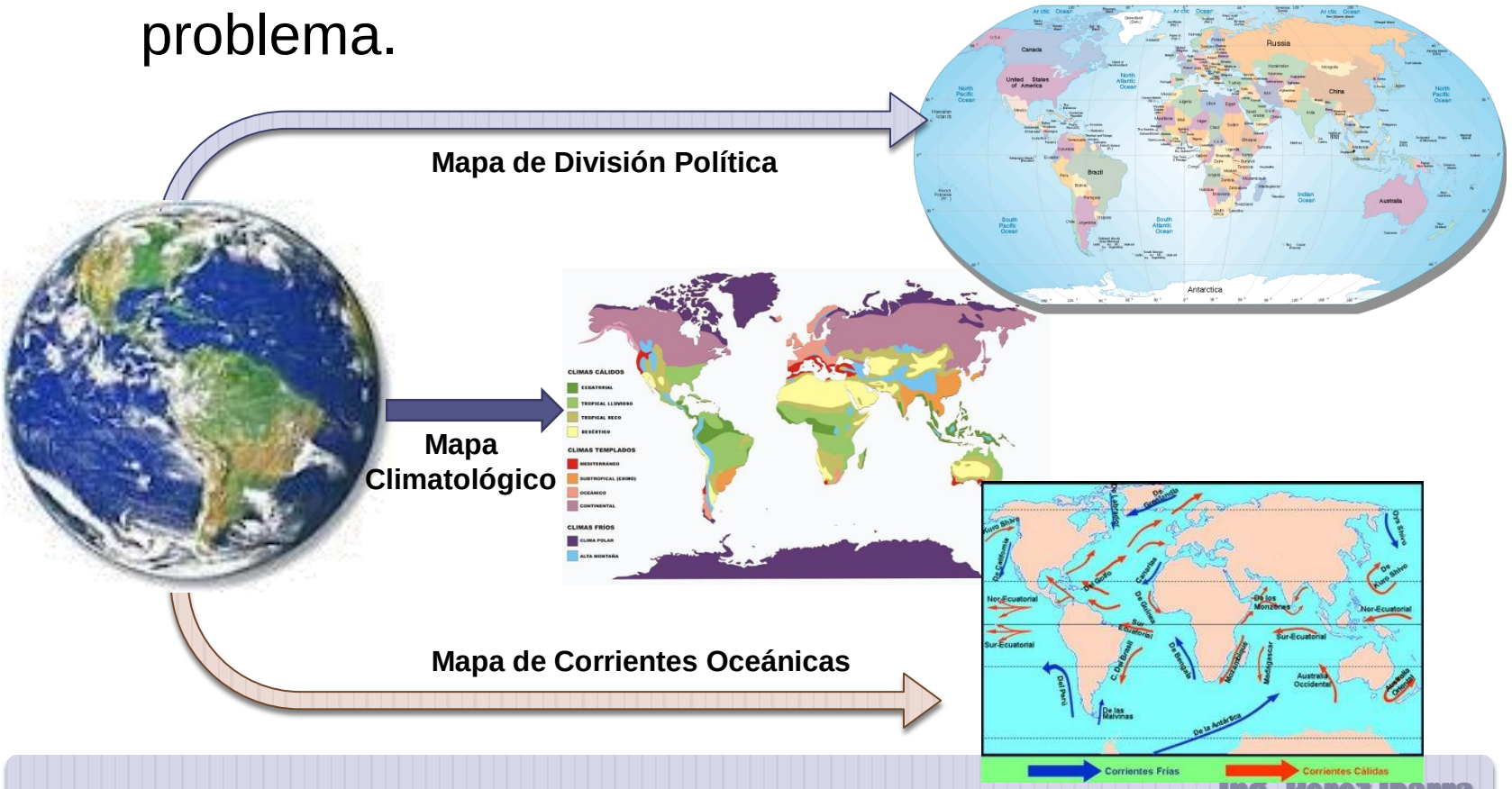
Programación Modular (1)

- Descomposición de problemas
  - Un problema complejo puede dividirse en problemas sencillos e independientes.



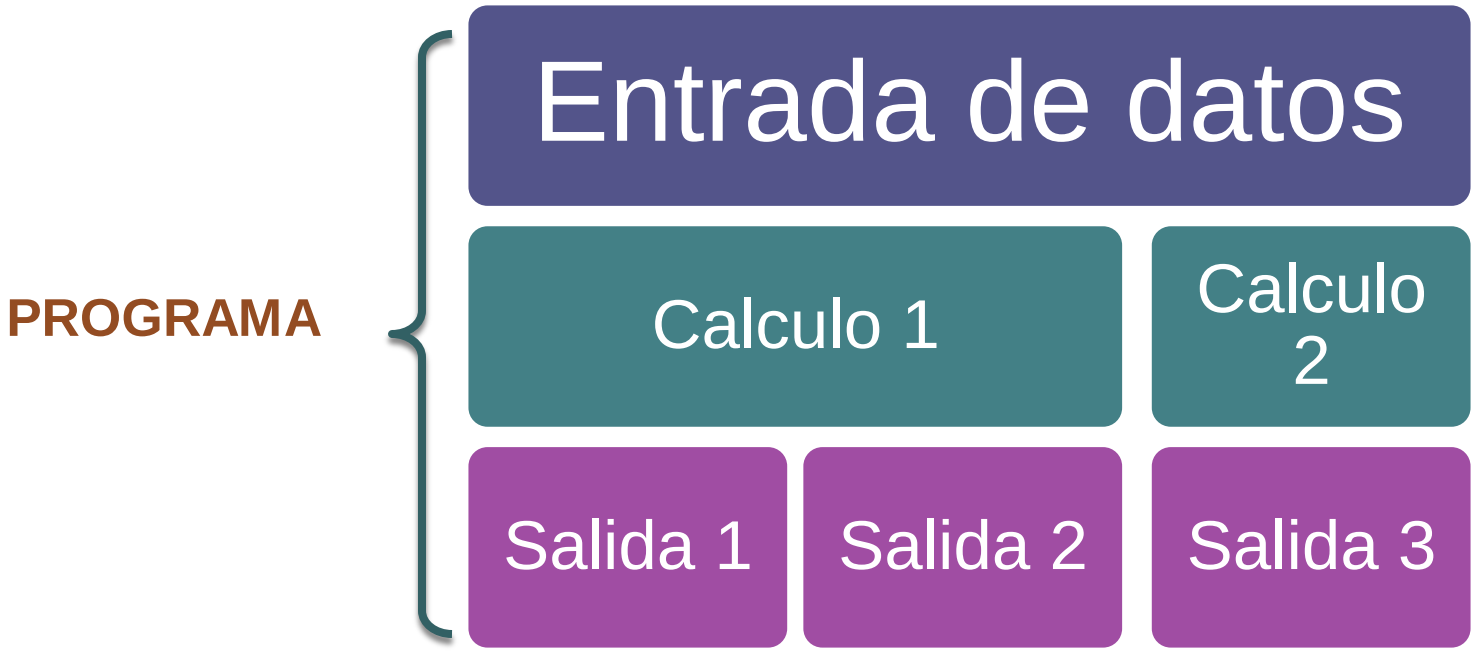
Programación Modular (2)

- Abstracción
  - Permite representar los objetos relevantes del problema.



Programación Modular (3)

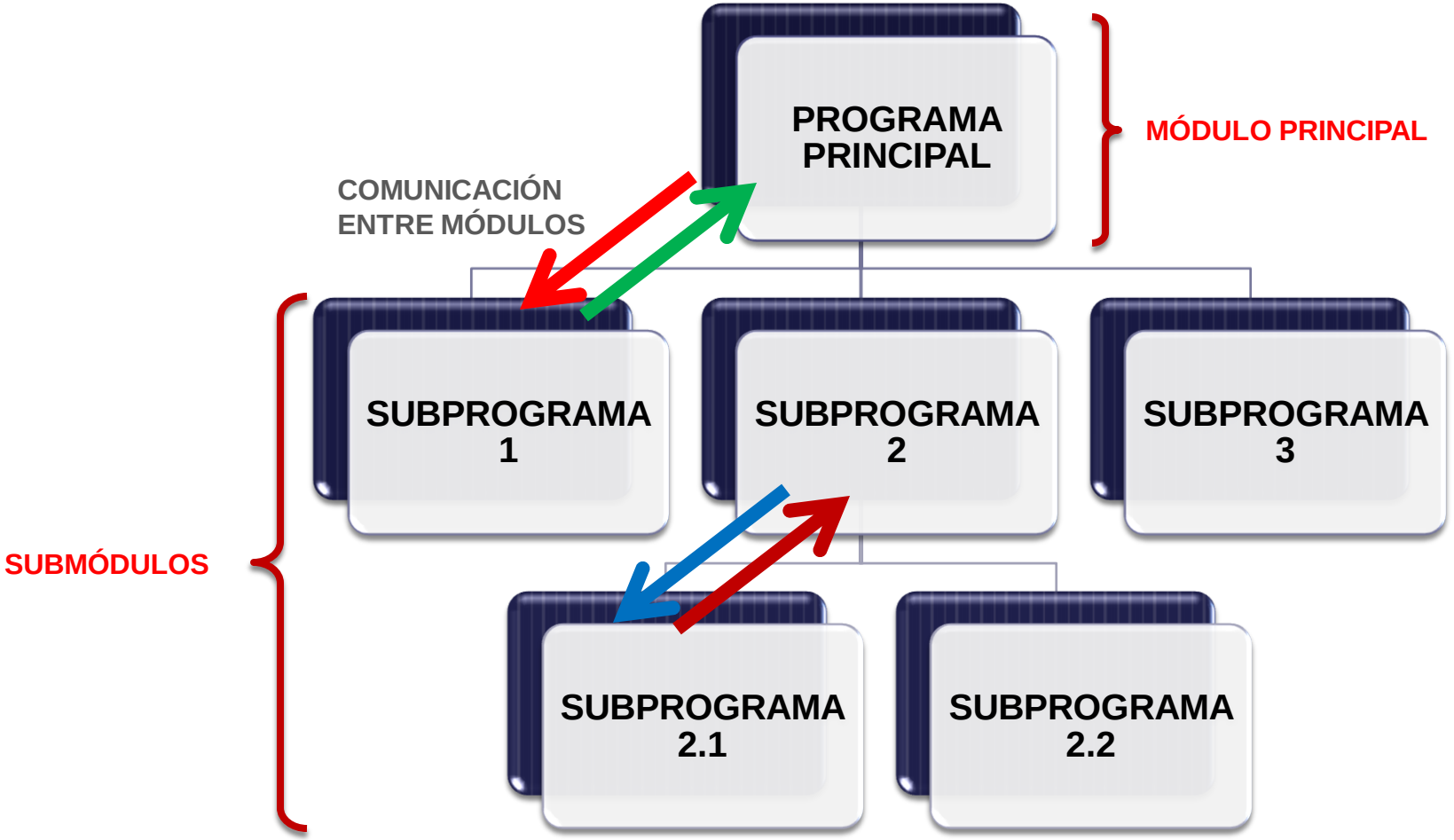
- Módulos
  - Un programa puede estar formado por partes independientes que resuelven subproblemas específicos.



Programación Modular (4)

- La programación modular
  - es un método de diseño flexible
  - permite dividir un programa en subprogramas
- Subprogramas o módulos
  - pueden analizarse, codificarse y probarse por separado
  - el módulo principal controla el flujo de acciones
  - se clasifican en Procedimientos y Funciones

Programas Modulares (1)



Programas Modulares (2)

- Entradas: se conoce el conjunto de datos con los que trabajará el módulo
- Propósito: se conoce el objetivo del módulo (qué hace)
- Salidas: se conoce el resultado que generará el módulo.

Funciones

- Una función es un módulo o subprograma que toma una lista de valores llamados argumentos o parámetros y devuelve un único valor.
- Las funciones se definen de un tipo de dato (entero, real, carácter, lógico).
- Las funciones pueden ser internas o definidas por el usuario.

Declaración de funciones

```
FUNCIÓN Nombre_función (Parámetros formales): Tipo_de_Función
VARIABLES
   Variables_de_la_Función
INICIO
   ACCIONES
   Nombre_función←resultado_de_la_función
FIN
```

- ✓ *Nombre_función:* especifica el nombre de la función.
- ✓ *Parámetros Formales:* son los valores que recibe la función y que se usarán en el cálculo.
- ✓ *Tipo_de_Función:* la función puede ser entera, real, carácter, lógica.
- ✓ *Variables_de_la_Función:* son las variables de la función, están definidas para la función y desaparecen cuando ésta finaliza su ejecución.
- ✓ *ACCIONES:* sentencias secuenciales, selectivas o repetitivas que implementan la operación.
- ✓ *Nombre_función←resultado_de_la_función:* el resultado del cálculo de la función se asigna al nombre de la función y se retorna al programa que la invocó.

Invocación de funciones

- Un función puede invocarse:

```
Variable←Función(Parámetros_actuales)
```

```
ESCRIBIR 'El resultado es: ', Función(Parámetros_actuales)
```

- Al invocar una función:
  1. A cada parámetro formal se le asigna el valor de su correspondiente parámetro actual.
  2. Se ejecuta el cuerpo de acciones de la función.
  3. Se asigna el resultado a la función y se retorna al punto de llamada.

Procedimientos

- Un procedimiento o subrutina es un subprograma que ejecuta un proceso específico.
- Los procedimientos, a diferencia de las funciones, no tienen asociado un valor.
- Los procedimientos pueden recibir parámetros para llevar acabo su trabajo, e incluso modificar éstos si es necesario.

Declaración de Procedimientos

```
PROCEDIMIENTO Nombre_procedimiento (Parámetros formales)
VARIABLES
   Variables_del_Procedimiento
INICIO
   ACCIONES
FIN
```

- ✓*Nombre_procedimiento:* especifica el nombre del procedimiento.
- ✓*Parámetros Formales:* especifica los valores que recibe el procedimiento, con los que realizará algún procesamiento.
- ✓*Variables_del_Procedimiento:* especifica las variables del procedimiento, éstas sólo están definidas para el procedimiento y desaparecen cuando finaliza su ejecución.
- ✓*ACCIONES:* sentencias secuenciales, selectivas o repetitivas que realizan la operación definida para el procedimiento.

Invocación de Procedimientos

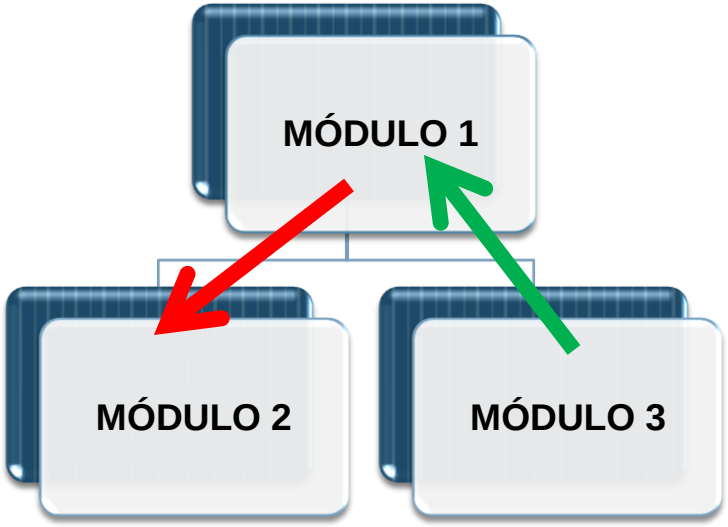
- Un procedimiento puede invocarse:

```
Nombre_Procedimiento(Parámetros_actuales)
```

- Al invocar un procedimiento:
  1. Los parámetros actuales sustituyen a los parámetros formales.
  2. El cuerpo de la declaración del procedimiento sustituye el llamado del procedimiento.
  3. Se ejecutan las acciones escritas por el código resultante.

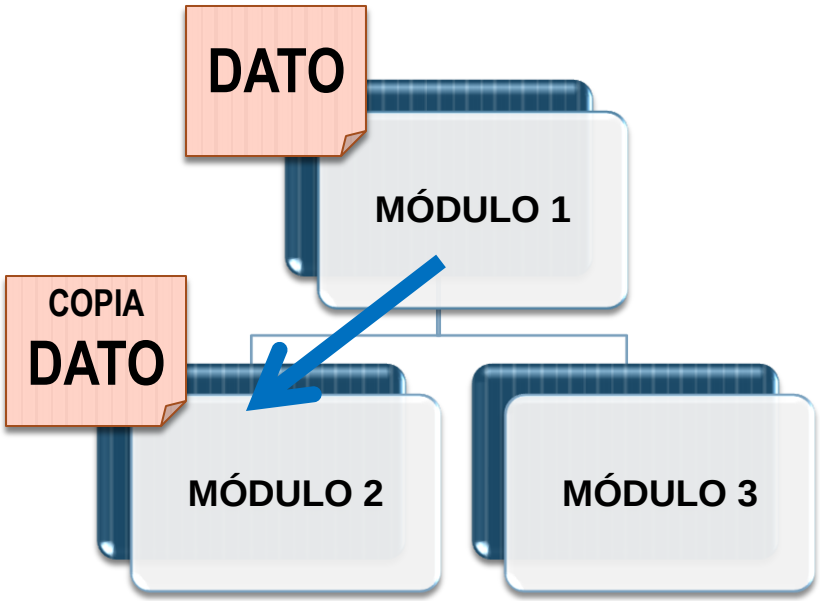
Comunicación entre módulos

- Los datos que usan los módulos de un programa se comunican a éstos cuando son invocados. Esta comunicación se denomina *Pasaje de Parámetros*.



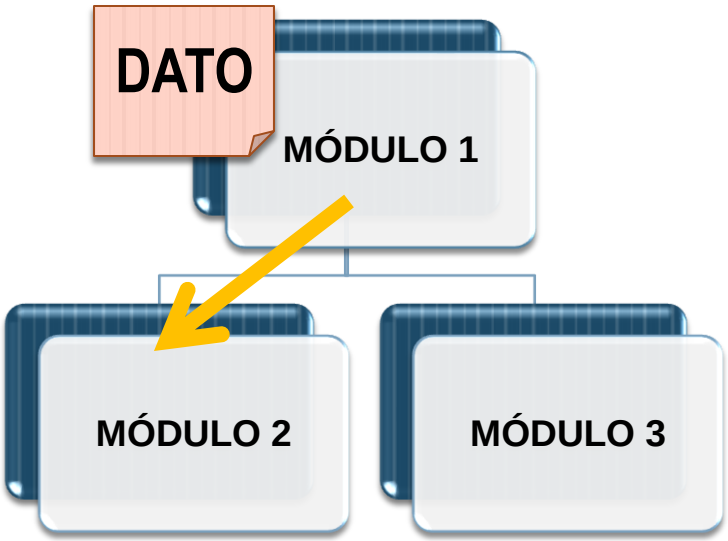
Pasaje de Parámetros (1)

- Por Valor: el módulo trabaja con copias de los datos originales. Estos parámetros se conocen como de entrada (E).



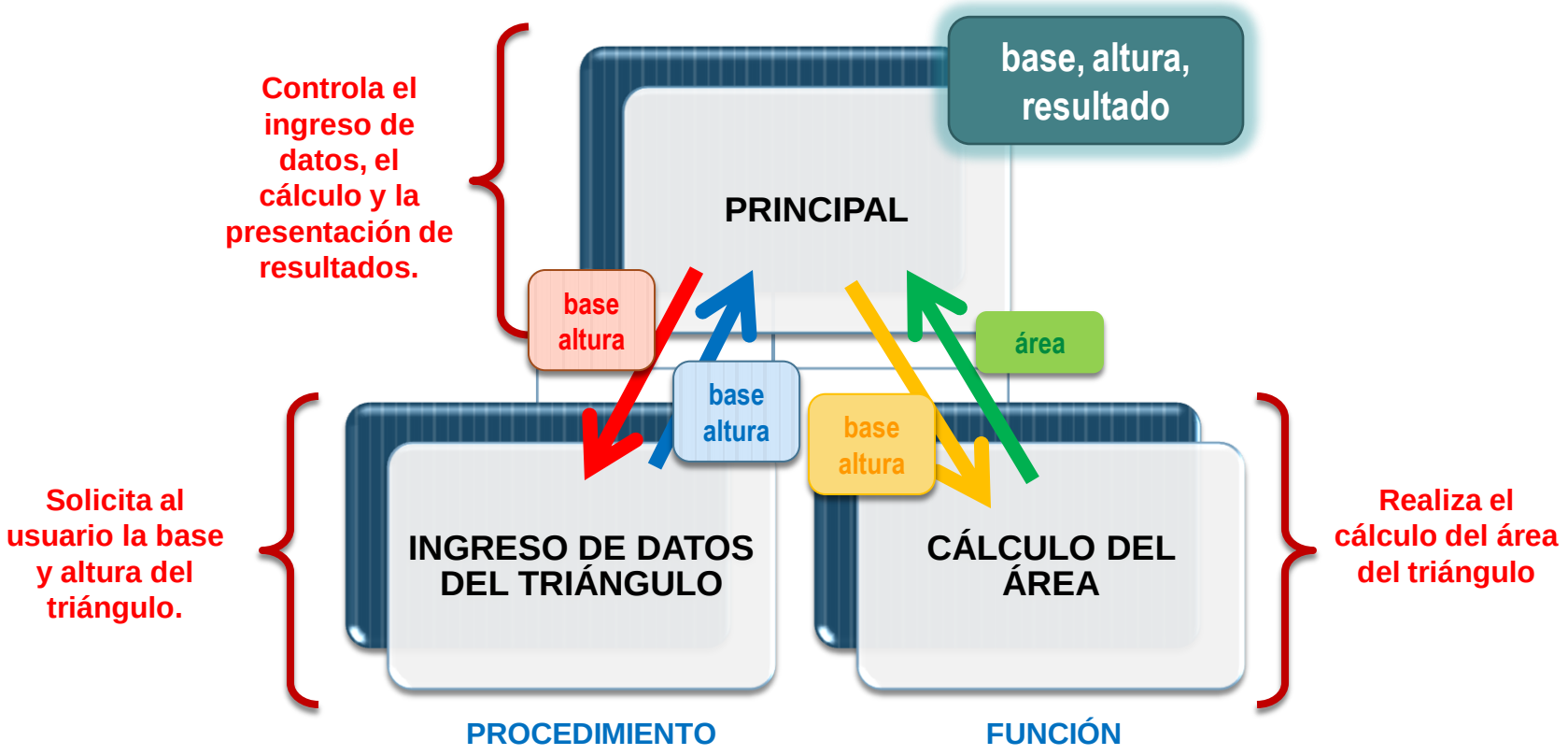
Pasaje de Parámetros (2)

- Por Referencia: el módulo trabaja con los datos originales y cualquier modificación altera los datos del programa que invocó al módulo. Estos parámetros se conocen como de entrada y salida (E/S).



Pasaje de Parámetros. Ej. (1)

- Ejemplo: Diseñe un programa modular que calcule el área de un triángulo.



Pasaje de Parámetros. Ej. (2)

- Ejemplo: Diseñe un programa modular que calcule el área de un triángulo.

```
PROGRAMA calculo_triangulo
VARIABLES
    base, altura, area: real
PROCEDIMIENTO Leer_datos(E/S b: real, E/S h: real)
INICIO
  ESCRIBIR 'Ingrese la base del triángulo:'
  LEER b
  ESCRIBIR 'Ingrese la altura del triángulo:'
  LEER h
FIN
FUNCIÓN Calculo_area(E b:real, E h:real): real
INICIO
   Calculo_area<-b * h / 2
FIN
```

```
INICIO
  Leer_datos(base,altura)
  area<-Calculo_area(base,altura)
  ESCRIBIR 'El área calculada es:', area
FIN
```

Pasaje de Parámetros. Ej. (3)

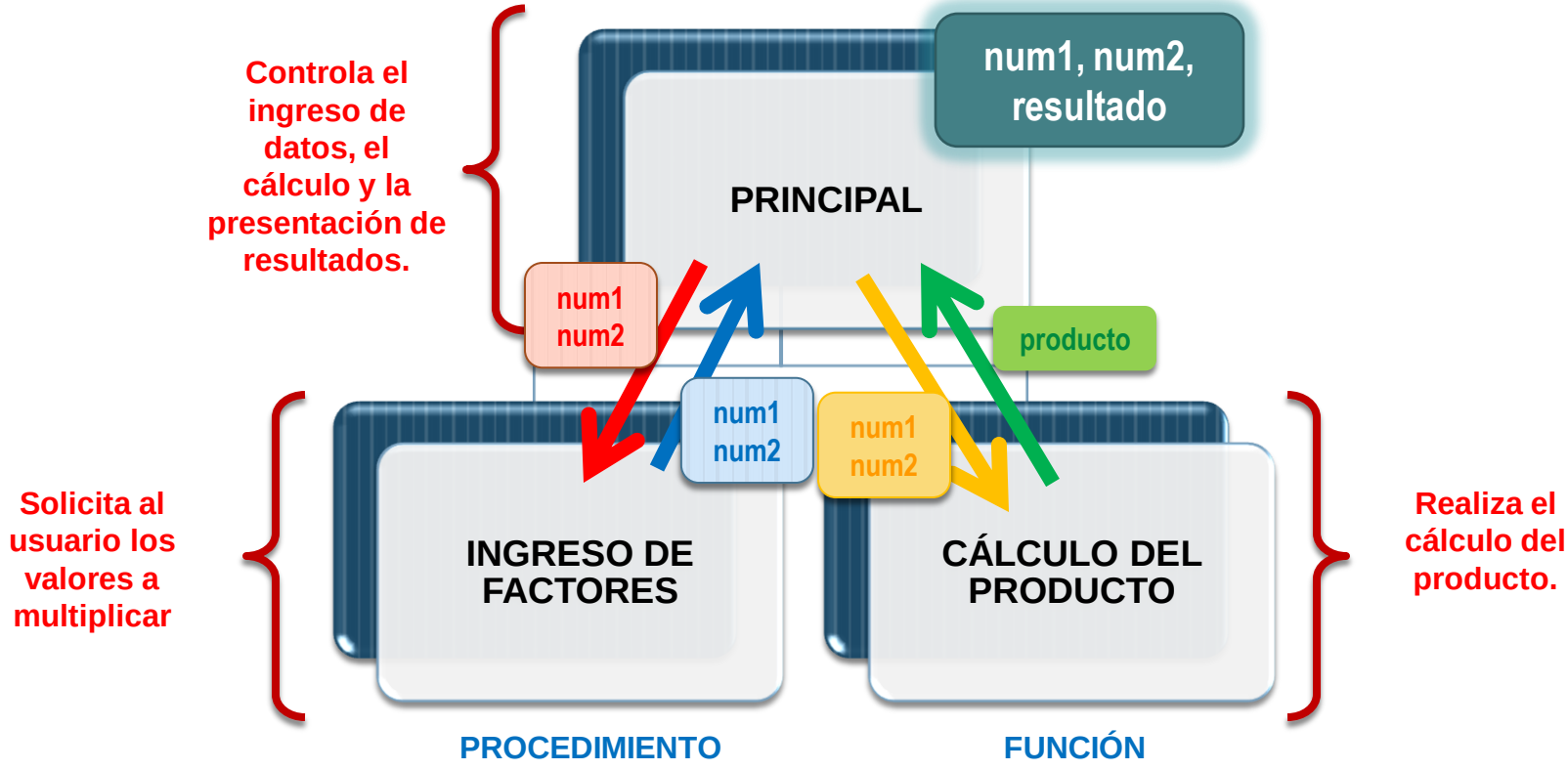
- Ejemplo: Diseñe un programa modular que calcule el área de un triángulo.

```
#include <iostream>
#include <stdlib.h>

using namespace std;

void leer_datos(float &b,float &h);
float calculo_area(float b,float h);

main()
{ float base, altura, area;
  leer_datos(base,altura);
  area=calculo_area(base,altura);
  cout << "El area calculada es: " << area << endl;
  system("pause");
}
```

```
void leer_datos(float &b, float &h)
{
  cout << "Ingrese la base del triangulo: ";
  cin >> b;
  cout << "Ingrese la altura del triangulo: ";
  cin >> h;
}

float calculo_area(float b, float h)
{
  return b * h / 2;
}
```

Pasaje de Parámetros. Ej. (4)

- Ejemplo: Diseñe un programa modular que calcule el producto de 2 números mediante sumas sucesivas.



Pasaje de Parámetros. Ej. (5)

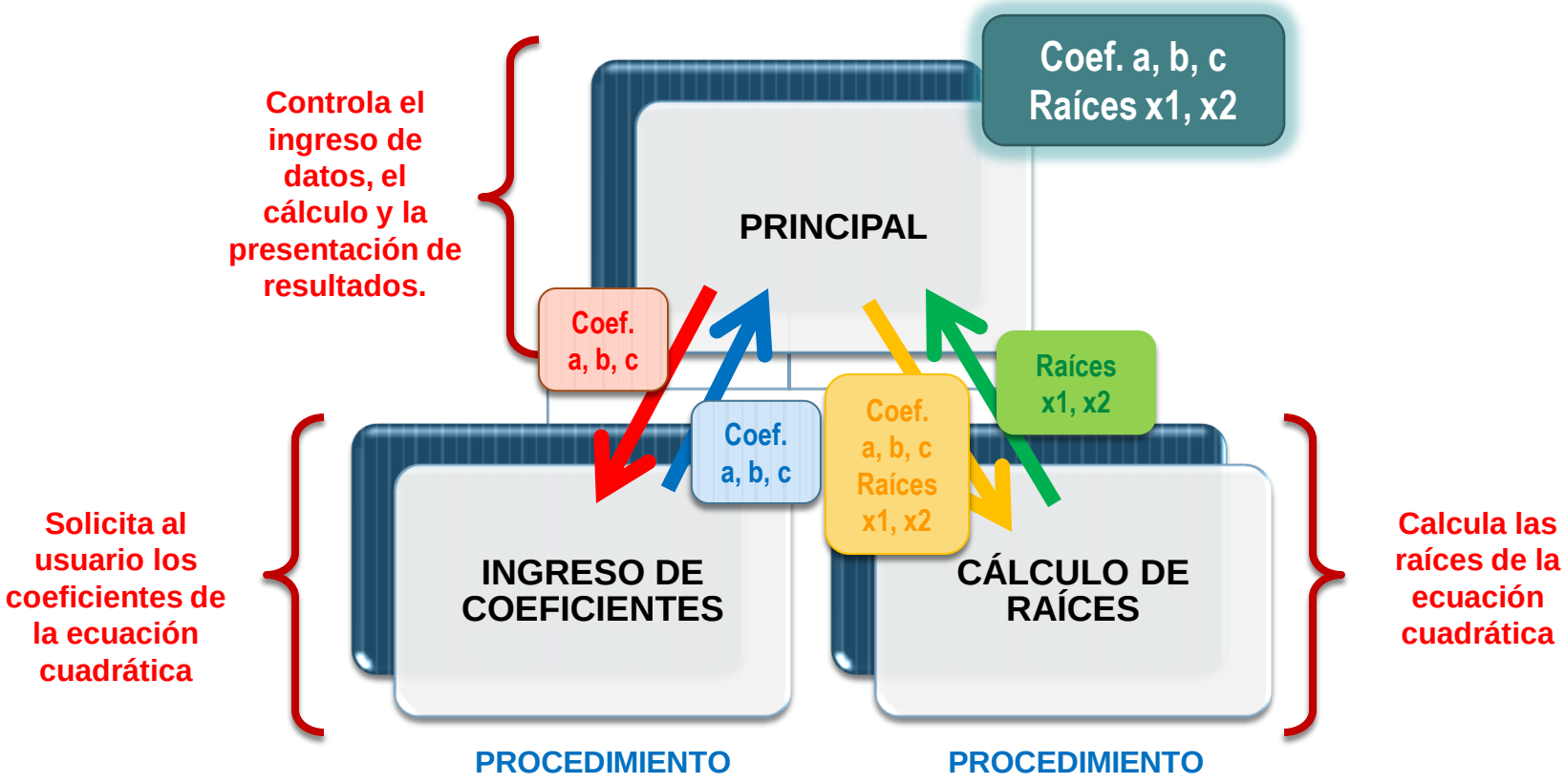
- Ejemplo: Diseñe un programa modular que calcule el producto de 2 números mediante sumas sucesivas.

```
#include <iostream>
#include <stdlib.h>

using namespace std;

void leer_datos(int &a,int &b);
int producto_sumas(int a,int b);

main()
{ int factor1, factor2, producto;
  leer_datos(factor1,factor2);
  producto=producto_sumas(factor1,factor2);
  cout << "El producto es: " << producto << endl;
  system("pause");
}
```

```
void leer_datos(int &a, int &b)
{
  cout << "Ingrese primer factor: ";
  cin >> a;
  cout << "Ingrese segundo factor: ";
  cin >> b;
}

int producto_sumas(int a, int b)
{ int i,p;
  p=0;
  for(i=1;i<=b;i++)
      p=p+a;
  return p;
}
```

Pasaje de Parámetros. Ej. (4)

- Ejemplo: Diseñe un programa modular que calcule las raíces de una ecuación cuadrática.



Pasaje de Parámetros. Ej. (5)

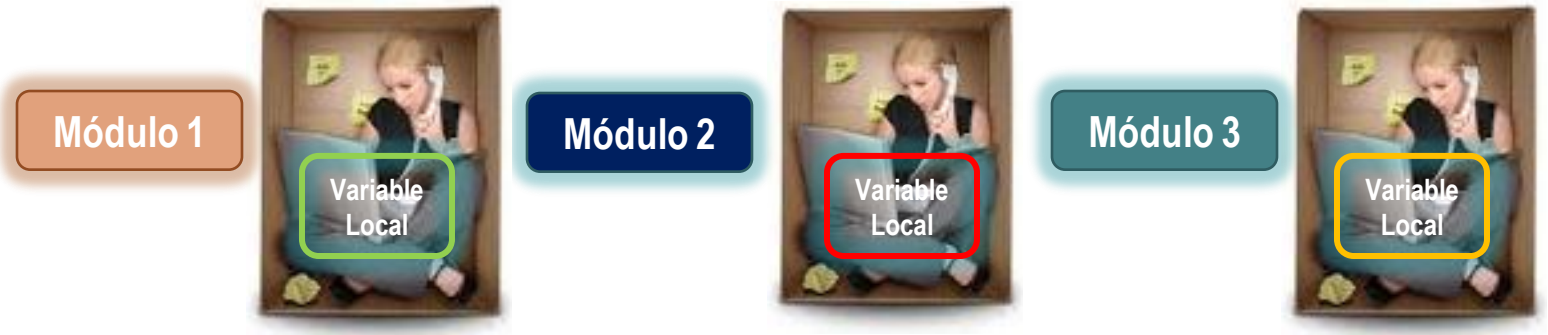
- Ejemplo: Diseñe un programa modular que calcule las raíces de una ecuación cuadrática.

```
#include <iostream>
#include <stdlib.h>
#include <math.h>

using namespace std;

void leer_datos(float &a,float &b,float &c);
void raices(float a,float b,float c,float &r1,float &r2);

main()
{ float ca, cb, cc, x1, x2;
  leer_datos(ca,cb,cc);
  raices(ca,cb,cc,x1,x2);
  cout << "Raiz 1: " << x1 << endl;
  cout << "Raiz 2: " << x2 << endl;
  system("pause");
}
```

```
void leer_datos(float &a, float &b, float &c)
{
  cout << "Ingrese coef. cuadratico: ";
  cin >> a;
  cout << "Ingrese coef. lineal: ";
  cin >> b;
  cout << "Ingrese coef. indep.: ";
  cin >> c;
}

void raices(float a,float b,float c,float &r1,
float &r2)
{
 r1=(-b+pow(pow(b,2)-4*a*c,0.5))/(2*a);
 r2=(-b-pow(pow(b,2)-4*a*c,0.5))/(2*a);
}
```

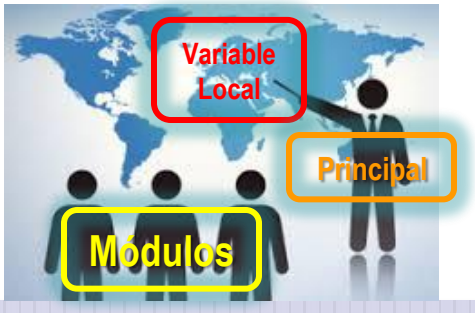
Variables Locales

- Una variable local es aquella que está declarada y definida dentro de un subprograma.
- El significado de las variables locales se confina al subprograma que las contiene.
- Una variable local no puede ser accedida por otros módulos del programa.



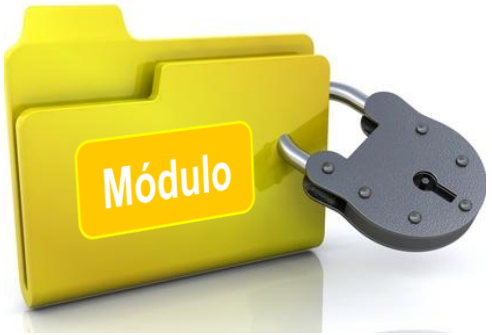
Variables Globales

- Una variable global es aquella que está declarada y definida en el programa principal.
- Las variables globales son conocidas por todos los módulos del programa.
- Una variable global puede ser accedida por cualquier módulo del programa.
  - ¡Cuidado! La manipulación de variables globales en los módulos puede ocasionar modificaciones accidentales y generar errores en el programa.



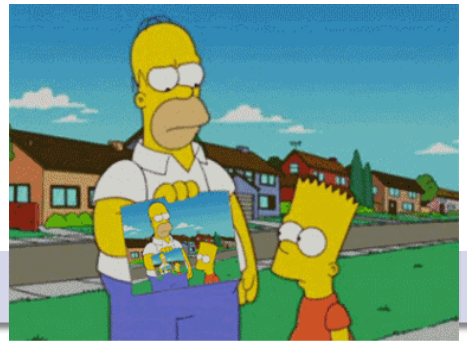
Ocultamiento y Protección

- *Data Hidding* significa que los datos relevantes para un módulo deben ocultarse a otros módulos.
- Esto evita que en el programa principal se declaren datos que sólo son relevantes para algún módulo en particular y, además, se protege la integridad de los datos.

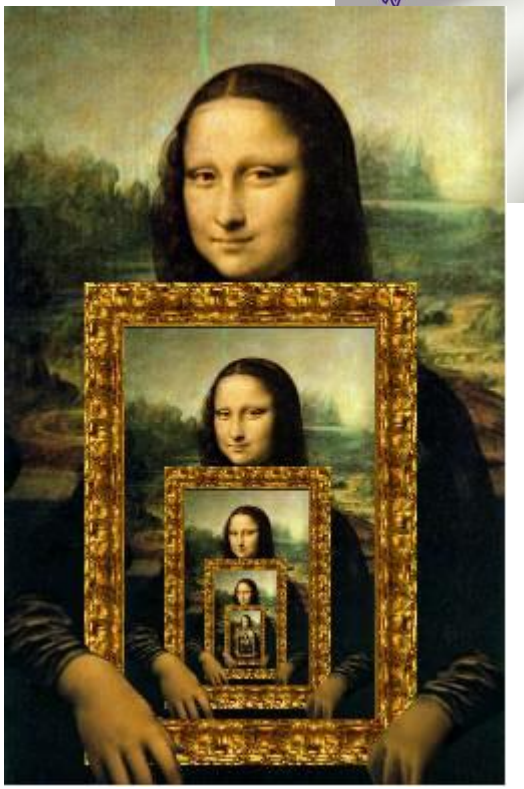


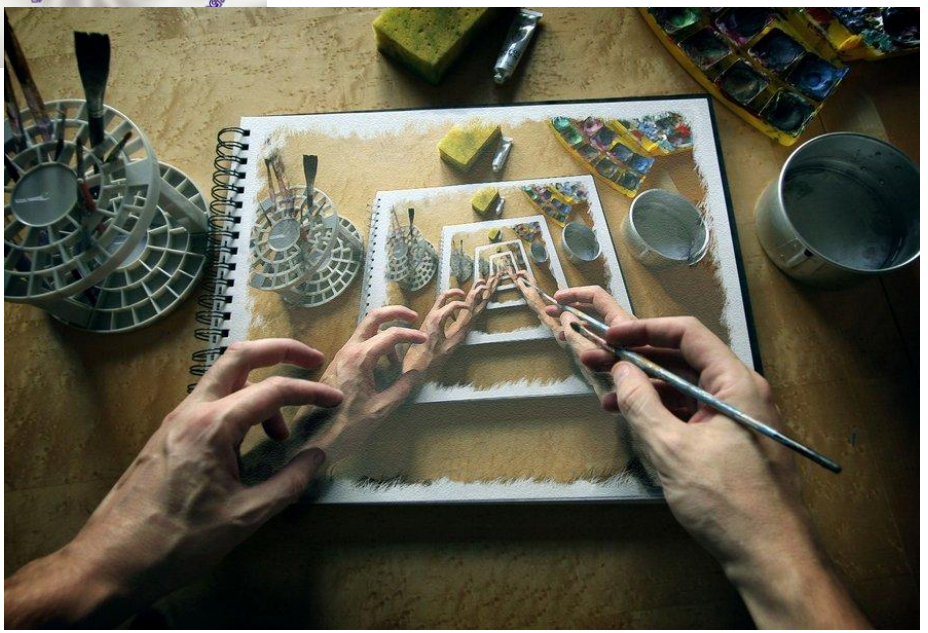
Recursividad

- La recursividad consiste en definir un concepto en términos del propio concepto.
- Una definición recursiva es válida si la referencia a sí misma es relativamente más sencilla que el caso considerado.
- La recursividad expresa un concepto complejo en función de las formas más simples del mismo concepto.



Recursividad





Razonamiento Recursivo (1)

- Partes del razonamiento recursivo:
  - Caso Base: indica el problema o caso más simple cuya resolución es directa.
  - Regla Recursiva de Construcción: plantea versiones más simples del problema original cuyas soluciones parciales permiten resolver el problema principal.

Razonamiento Recursivo (2)

- Consideraciones
  1. la división sucesiva del problema original en uno o varios problemas más pequeños, del mismo tipo que el inicial;
  2. la resolución de los problemas más sencillos, y
  3. la construcción de las soluciones de los problemas complejos a partir de las soluciones de los problemas más sencillos.

Algoritmo Recursivo (1)

- Características
  1. el algoritmo debe contener una llamada a sí mismo,
  2. el problema planteado puede resolverse atacando el mismo problema pero de tamaño menor,
  3. la reducción del tamaño del problema permite alcanzar el caso base, que tiene solución directa.

Algoritmo Recursivo (2)

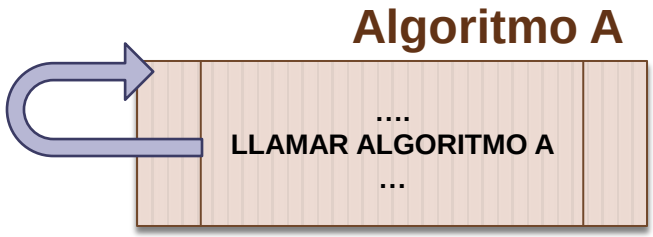
- Partes del algoritmo recursivo:
  - iterativa o no recursiva
  - condición de terminación (caso base)
  - recursiva (que reduce el tamaño del problema hasta alcanzar el caso base).
- La parte recursiva y la condición de terminación son obligatorias.
- El caso base siempre debe alcanzarse, sino el algoritmo se invoca indefinidamente.

Ventajas y Desventajas

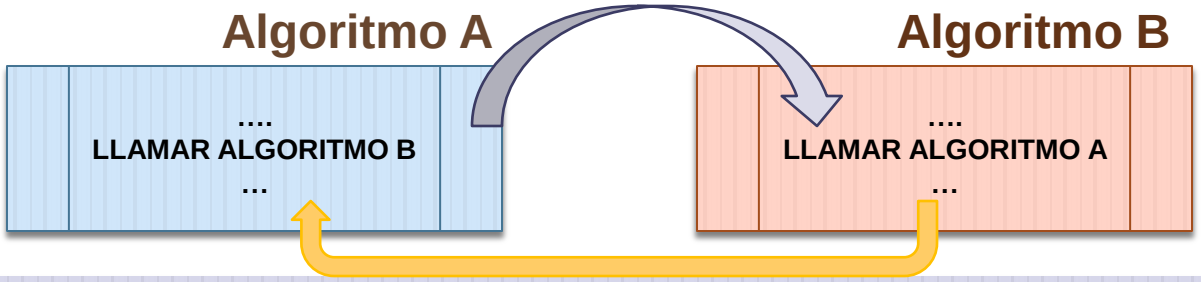
- Ventajas
  - Fácil comprensión
  - Fácil comprobación
  - Solución sencilla a problemas de naturaleza recursiva (versiones iterativas complicadas).
- Desventajas
  - Las soluciones recursivas, en general, son menos eficientes que las iterativas (consumo de memoria)

Tipos de Recursividad

- Recursividad Directa (simple): un algoritmo se invoca a sí mismo una o más veces directamente.



- Recursividad Indirecta (mutua): un algoritmo *A* invoca a otro algoritmo *B* y éste a su vez invoca al algoritmo *A*.



Procedimientos Recursivos

- Un procedimiento *P* es recursivo si:
  1. incluye un cierto criterio, llamado caso base, por el que el procedimiento no se llama a sí mismo.
  2. cada vez que el procedimiento se llame a sí mismo (directa o indirectamente), debe estar más cerca del caso base.

Funciones Recursivas

- Un función *F* es recursiva si:
  1. existen ciertos argumentos, llamados valores base, para los que la función no se refiere a sí misma.
  2. cada vez que la función se refiera a sí misma, el argumento de la función debe acercarse más al valor base.

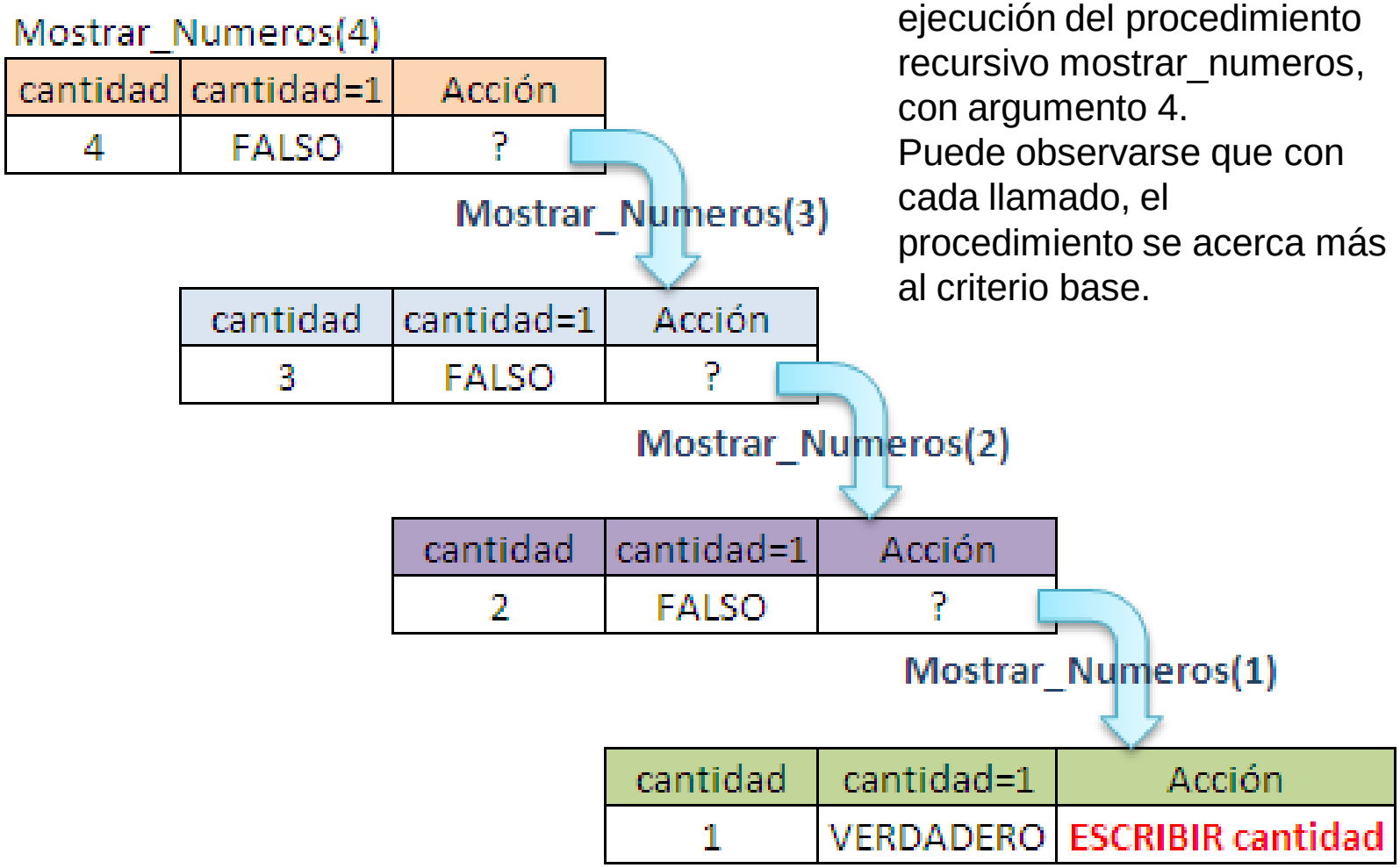
Ejemplo 1 (1)

```
PROCEDIMIENTO Mostrar_Numeros(E cantidad: entero)
INICIO
    SI cantidad=1 ENTONCES
       ESCRIBIR cantidad
    SINO
       Mostrar_Numeros(cantidad-1)
       ESCRIBIR cantidad
    FINSI
FIN
```

```
void mostrar_numeros (int cantidad)
{
  if (cantidad==1)
     cout << cantidad << endl;
  else
   { mostrar_numeros(cantidad-1);
     cout << cantidad << endl; }
}
```

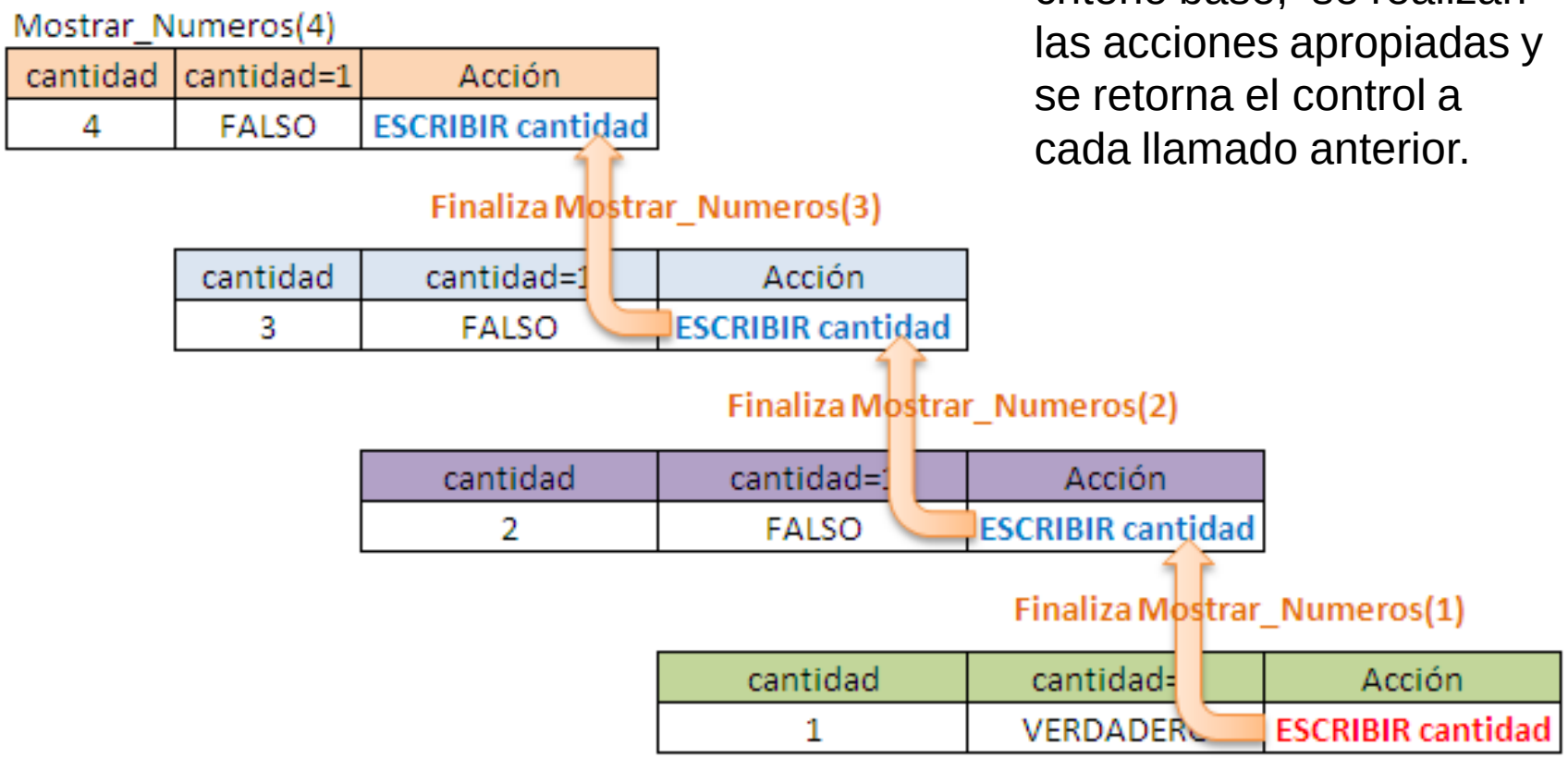
Ejemplo 1 (2)

El ejemplo 1 muestra la ejecución del procedimiento recursivo mostrar_numeros, con argumento 4.
Puede observarse que con cada llamado, el procedimiento se acerca más al criterio base.



Ejemplo 1 (3)

Una vez cumplido el criterio base, se realizan las acciones apropiadas y se retorna el control a cada llamado anterior.

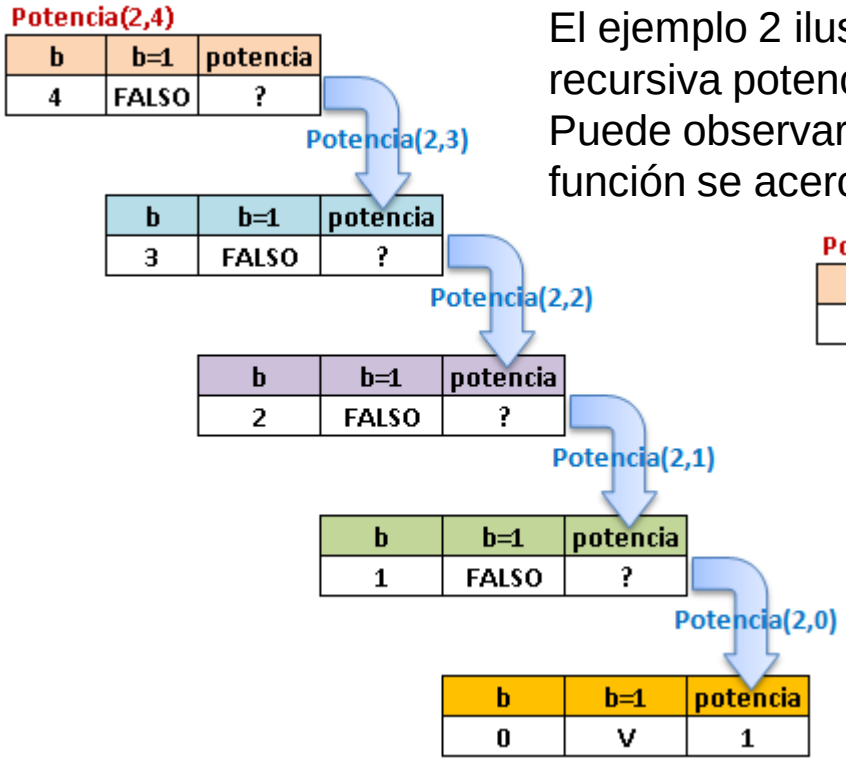


Ejemplo 2 (1)

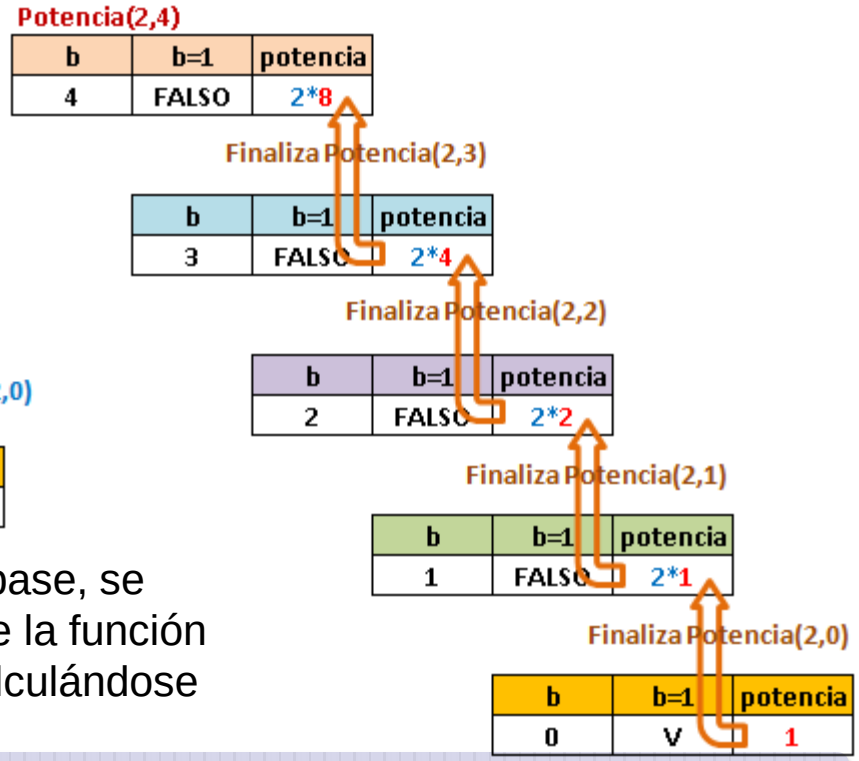
```
FUNCIÓN Potencia(E a:entero, E b:entero): entero
INICIO
        SI b=0 ENTONCES
            Potencia←1
        SINO
            Potencia←a * Potencia(a,b-1)
        FINSI
FIN
```

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

Ejemplo 2 (2)



El ejemplo 2 ilustra la ejecución de la función recursiva potencia, con argumentos 2 y 4. Puede observarse que con cada llamado, la función se acerca más a los valores base.



Una vez que se alcanzan los valores base, se obtiene la solución directa y el valor de la función se retorna a cada llamado anterior, calculándose así el valor final.

Bibliografía

- Sznajdleder, Pablo Augusto. Algoritmos a fondo. Alfaomega. 2012.
- López Román, Leobardo. Programación estructurada y orientada a objetos. Alfaomega. 2011.
- De Giusti, Armando *et al*. Algoritmos, datos y programas, conceptos básicos. Editorial Exacta, 1998.
- Joyanes Aguilar, Luis. Fundamentos de Programación. Mc Graw Hill. 1996.
- Joyanes Aguilar, Luis. Programación en Turbo Pascal. Mc Graw Hill. 1990.