Tema 5: FUNCIONES

PROGRAMACIÓN I

Marcos Novalbos
Elena García Gamella

Índice

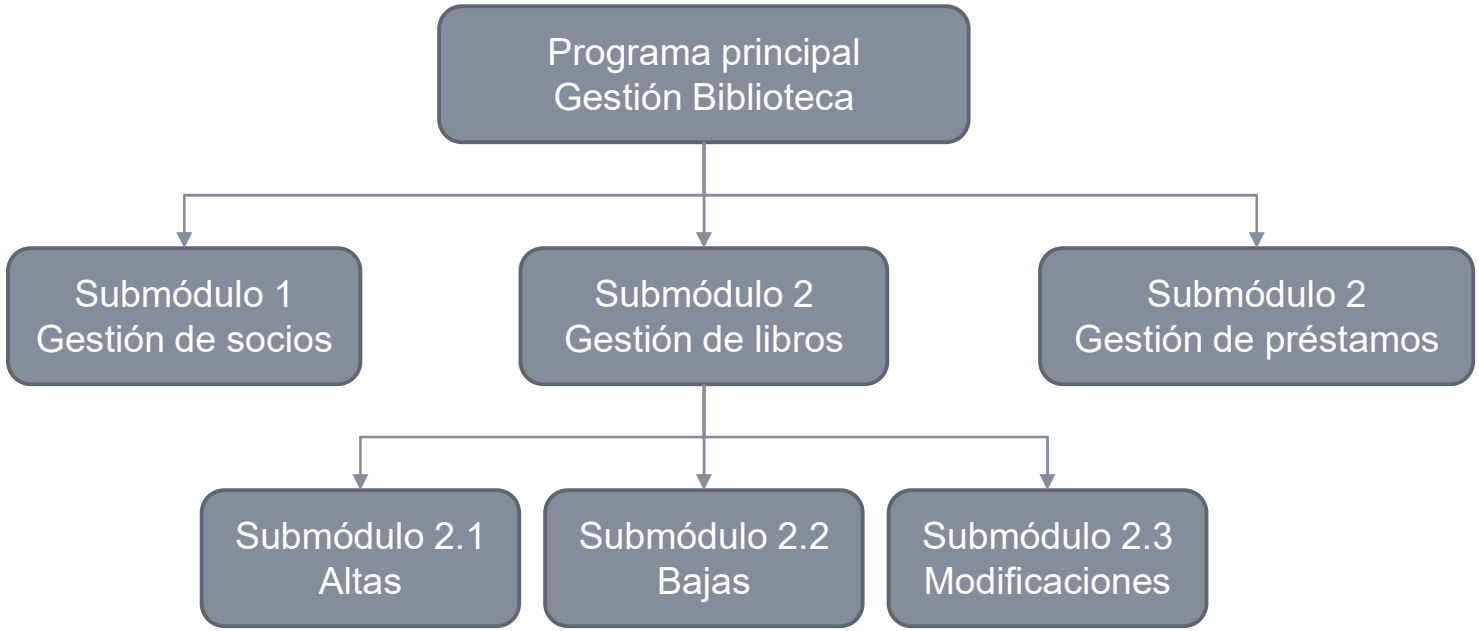
Divide y vencerás. *Julio Cesar*

1. Introducción

- El paradigma del **divide y vencerás** consiste en dividir el problema en trozos más pequeños, resolver esos trozos y luego unir las soluciones para tener la solución del problema completo.
- La programación modular se basa en diseñar un programa dividiéndolo en módulos que resultan de segmentar el problema en funciones lógicas que son perfectamente diferenciadas. Esta división exige la presencia de un módulo denominado módulo de base o principal a objeto de que controle y se relacione con los demás
- Un programa diseñado mediante esta técnica quedará constituido por dos partes claramente diferenciadas:
  - Programa principal
  - Subprogramas

1. Introducción

- Consiste en dividir la APLICACIÓN (Problema completo) en MODULOS (Problemas parciales) coordinados mediante un PROGRAMA PRINCIPAL.

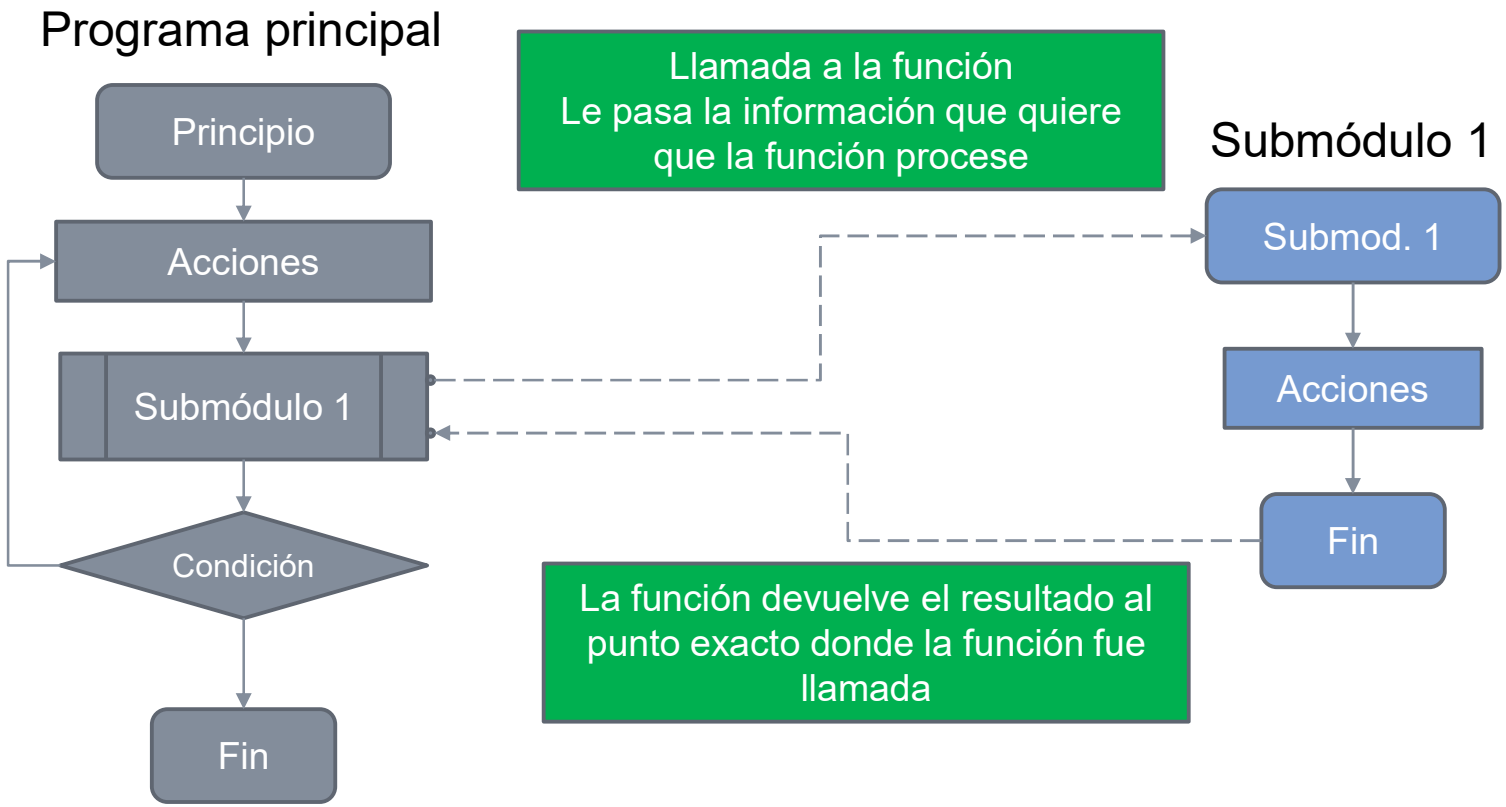


1. Introducción

- **Una función en C es un segmento independiente de código fuente diseñado para hacer una tarea específica y fácilmente reutilizable.**
- El uso de las funciones:
  - Ayuda a dividir un programa grande en componentes más pequeños, y de esta forma permiten el desarrollo modular de los programas.
  - Evita la necesidad de repetir las mismas instrucciones en distintas partes del programa.
  - Permiten transferir un conjunto de datos distintos cada vez que se accede a ellas.

1. Introducción

- Las funciones permitir el desarrollo modular de un programa



2. Forma general de una función. **Definición**

```
tipo_devuelto Nombre_de_la_Funcion (tipo_1
 argumento_1,...,                                    tipo_
 n argumento_n)
{
   instrucción_1;
   instrucción_2;
   ...
   instrucción_n;


   return  expresion_devuelta;
}
```

2. Forma general de una función. **Definición**

- `Tipo_devuelto`
  - Indica el tipo de dato que devuelve la función (tipo de dato en el que se devuelve el resultado de la función).
  - Si no se especifica nada por defecto devuelve un entero.
  - Si la función no devuelve nada se especifica con *void*.
  - Es importante especificar el tipo siempre
- `Nombre_de_la_función`
  - Nombre de la función que será usado para llamar a la función desde cualquier parte del programa.
  - Este nombre tiene las mismas limitaciones que los nombres de los identificadores.

2. Forma general de una función. **Definición**

- `tipo_1 argumento_1, ... , tipo_n argumento n`
  - Es la lista de argumentos que recibe la función, separados por comas
  - Estos argumentos se llaman también parámetros formales
  - Una función puede no tener parámetros, en cuyo caso la lista de parámetros está vacía
  - Una lista de parámetros vacía se puede especificar explícitamente como tal colocando la palabra void entre los paréntesis.
  - Pueden ser cualquier tipo de dato de C: int, double, float, char, ...
- `{`
  - Es la llave que marca el comienzo del cuerpo de la función
- `instrucción_1;... instrucción_n;`
  - Son las instrucciones que componen el cuerpo de la función
  - Se escriben con sangrado de algunos espacios para delimitar más claramente dónde comienza y dónde termina la función

2. Forma general de una función. **Definición**

- `return expresión_devuelta;`
  - Es la última instrucción de la función
  - **Produce que la función termine** y devuelva el resultado en expresión_devuelta a quién la hubiera llamado.
  - Si el tipo_devuelto de la función es void, se termina la función con la instrucción return; (sin expresión_devuelta).
    - Sólo en este caso, se puede omitir la instrucción return, con lo cual la función terminará al llegar a la llave }
  - Puede existir más de una sentencia return en una función, pero NO ES ACONSEJABLE (como mucho se pueden utilizar dos sentencias return en una misma función).
  - La instrucción return permite que la función devuelva un valor por lo que es posible incluir la llamada a una función como operando de una expresión.
- `}`
  - Es la llave que cierra el cuerpo de la función.

2. Forma general de una función. **Definición**.

Ejemplo

```
int areaCuadrado (int altura, int base)
{
  int Area;
  Area = altura * base;
  return (Area);
}
```

3. Declaración de una función. **El prototipo**

- Antes de empezar a utilizar una función se debe declarar -> prototipo
- La sintaxis genérica de la declaración de una función es:

```
tipo_devuelto NombreFunción (tipo_1 argumento 1, ..., tipo_n argumento_ n);
```

- Para que una función pueda usarse en otras partes del programa es necesario colocar al principio del programa el prototipo de la función.
- La misión del prototipo es declarar la función al resto del programa, lo cual permite al compilador comprobar que cada una de las llamadas a la función es correcta. El compilador verifica:
  - Que los argumentos son correctos, tanto en número como en tipo
  - Que el uso del valor devuelto por la función sea acorde con su tipo

3. Declaración de una función. **El prototipo**

- El uso de prototipos en C es opcional, pero MUY RECOMENDABLE pues permite al compilador detectar errores que, de no existir el prototipo, pasarían inadvertidos al programa.

- Es igual a la primera línea de la definición de la función con un ; al final

3. Declaración de una función: el prototipo. Ejemplo

```
int areaCuadrado (int altura, int base); //prototipo

void main (){
   int miarea, a, b ;
   miarea = areaCuadrado (a,b); //invocación a la funcion
}

int areaCuadrado (int altura, int base) // cuerpo de la funcion
{
  int Area;
  Area = altura * base;
  return (Area);
}
```

4. Parámetros formales y reales

- Parámetros formales:
  - Aquellas variables que reciben el valor de los argumentos utilizados en la llamada a la función.
  - A efectos prácticos son las variables que aparecen en la cabecera de la función.
  - Son variables locales:
    - Reconocidas únicamente dentro de la función.

- Parámetros reales:
  - Aquellas variables que se utilizan para hacer referencia a la función.
  - A efectos prácticos son las variables que aparecen en la llamada a la función.

- Los parámetros reales y formales no tienen por qué llamarse igual, pero deben coincidir en tipo y orden.

4. Parámetros formales y reales

```
#include <stdio.h>

int Mul (int a, int b);

int main(void)
{
   int x;
   int y;
   int repuesta;
   printf("introduzca dos numeros: ");
   scanf("%d", &x);
   scanf("%d", &y);

   /* Llamada a la función */

   respuesta = Mul  (x,y);

   printf("La respuesta es %d\n", respuesta);
}

int Mul  (int a, int b)
{
   int res;
   res = a * b;
    return res;
}
```

Prototipo

Parámetros reales

Parámetros formales

Los parámetros reales y formales no tienen por qué llamarse igual, pero deben coincidir en tipo y orden.

5. Instrucción **return**

- La sentencia **`return`** :
  - Produce una salida inmediata de la función, y volver al programa que hizo la llamada.
  - Devuelve un valor concreto y solo uno como resultado de la función.
- Puede existir más de una sentencia return en una función
  - Pero no es aconsejable, como mucho dos sentencias return.
- Como return hace que la función devuelva un valor, es posible incluir la llamada a una función como operando de una expresión.

```
int Min( int a, int b)
{
      if ( a < b) {
            return a;
   } else {
            return b;
      }
}
```

```
int Max( int a, int b)
{
   int res;
      if ( a <  b) {
            res = a;
      } else {
            res = b;
      }
     return res;
}
```

5. Ejemplo 1: función suma serie

- Hacer un programa que pida al usuario dos números enteros (a y b) y que a continuación invoque una función que haga la suma de los términos comprendidos entre dichos números.
- La función recibirá como parámetros los dos números leidos del usuaio, a y b que se corresponden con el valor inicial de la serie y el valor final de la serie.
- La función devolverá como resultado un número entero que será la suma de los términos de la serie comprendidos entre a y b (incluyendo dichos números).

5. Ejemplo 1: función suma serie

```
#include <stdio.h>

int sumaSerie(int num1, int num2);

int main (){
    int a, b;
    int resultado;

    printf("Introduce el valor de a: ");
    scanf("%d", &a);
    printf("Introduce el valor de b: ");
    scanf("%d",&b);
    printf("Valor de a: %d - Valor de b: %d\n", a,b);

    resultado = SumaSerie(a, b);

    printf("El resultado de la suma de los numeros entre %d y %d: es %d\n", a, b, resultado);

}


int SumaSerie(int num1, int num2)
{
    int suma; /* VARIABLE LOCAL que almacena la suma parcial de la serie */

    suma = 0;
    while(num1 <= num2)
    {
        suma += num1;
        num1++;
    }
    return suma;
}
```

6. Paso de argumentos a funciones

- Paso por valor:
  - Este método consiste en copiar el valor del argumento real en el parámetro formal de la función.
  - Los cambios efectuados en los parámetros formales de la función no tienen efecto en los argumentos reales del programa.

- Paso por referencia:
  - En este método, la dirección del argumento real se copia en el parámetro formal de la función.
  - En otras palabras, lo que le pasamos a la función son las direcciones de memoria de los argumentos.
  - Los cambios que se hagan en el parámetro formal afectarán al parámetro real utilizado para realizar la llamada.
  - **Para forzar una llamada por referencia en C se utilizarán punteros.**

- Es posible que una función no tenga argumentos y su prototipo será:

```
tipo_devuelto NombreFuncion (void)
```

6. Paso de argumentos a funciones

```
#include <stdio.h>

int sumaSerie(int num1, int num2);

int main (){
    int a, b;
    int resultado;

    printf("Introduce el valor de a: ");
    scanf("%d", &a);
    printf("Introduce el valor de b: ");
    scanf("%d", &b);
    printf("Valor de a: %d - Valor de b: %d\n", a,b);

    resultado = SumaSerie(a, b);

    printf("El resultado de la suma de los numeros entre %d y %d: es %d\n", a, b, resultado);

}


int SumaSerie(int num1, int num2)
{
    int suma; /* VARIABLE LOCAL que almacena la suma parcial de la serie */

    suma = 0;
    while(num1 <= num2)
    {
        suma += num1;
        num1++;
    }
    return suma;
}
```

En el ejemplo 1 del apartado anterior hemos visto que dentro de la función SumaSerie se incrementa el valor del primer parámetro "num1".

¿Ese incremento se realiza también sobre la variable "a" utilizada como argumento al invocar la función?

6. Paso de argumentos a funciones

```
#include <stdio.h>

int sumaSerie(int num1, int num2);

int main (){
    int a, b;
    int resultado;

    printf("Introduce el valor de a: ");
    scanf("%d", &a);
    printf("Introduce el valor de b: ");
    scanf("%d", &b);
    printf("Valor de a: %d - Valor de b: %d\n", a,b);

    resultado = SumaSerie(a, b);

    printf("El resultado de la suma de los numeros entre %d y %d: es %d\n", a, b, resultado);

}


int SumaSerie(int num1, int num2)
{
    int suma; /* VARIABLE LOCAL que almacena la suma parcial de la serie */

    suma = 0;
    while(num1 <= num2)
    {
        suma += num1;
        num1++;
    }
    return suma;
}
```

En el ejemplo 1 del apartado anterior hemos visto que dentro de la función SumaSerie se incrementa el valor del primer parámetro "num1".

¿Ese incremento se realiza también sobre la variable "a" utilizada como argumento al invocar la función?

¡ La respuesta es no !
Todas las variables de la función, incluyendo los parámetros son locales a la función. El valor de la variable "a" se copió en la variable "num1".
En este caso el paso de parámetros se realizó "por valor"

6. Paso de argumentos a funciones por valor

```
#include <stdio.h>

void modificar (int var);

int main(void)
{
    int a;
    /* Inicializaciones */
  a=1;
  printf ("a = %d antes de llamar a la función modificar \n", a);

  modificar (a);

  printf ("a = %d después de llamar a la función modificar \n", a);
}


void modificar (int  var)
{
    printf ("var = %d dentro de la func. modificar antes de modificar su valor:\n", var);

  var= 9;

  printf ("var = %d dentro de la func. modificar después de modificar su valor: \n", var);
}
```

```
a = 1 antes de llamar a la función modificar
var= 1 dentro de función modificar antes de modificar
var= 9 dentro de función modificar después de var
a = 1 después de llamar a la función modificar
```

6. Paso de argumentos a funciones por referencia

```
#include <stdio.h>

void modificar (int  *var);

int main(void)
{
    int a;

 /* Inicializaciones */
  a=1;
    printf (" a = %d antes de llamar a la función modificar \n", a);

  modificar (&a);

  printf ("a = %d después de llamar a la función modificar \n", a);
}


void modificar (int  *var)
{
    printf ("var = %d dentro de la función modificar antes de modificar \n", *var);

    *var= 9;

    printf ("var= %d dentro de la función modificar después de modificar \n", *var);
}
```

```
a = 1 antes de llamar a la función modificar
var= 1 dentro de función modificar antes de modificar
var= 9 dentro de función modificar después de modificar
a = 9 después de llamar a la función modificar
```

6. Optimización paso de argumentos a funciones por referencia

```
void modificar (int  *var)
{
    printf (“var= %d dentro de la función modificar antes de modificar \n”, *var);
    *var= 9;
    printf (“var= %d dentro de la función modificar después de modificar \n”, *var);
}

void modificar (int  *var)
{
  int varlocal;
  varlocal = *var;

    printf (“variable local = %d dentro de la función modificar antes de modificar \n”, varlocal);
    varlocal= 9;
    printf (“variable local = %d dentro de la función modificar después de modificar \n”, varlocal);
   *var= varlocal;
}
```

El uso de variables locales a la función hace que la ejecución sea más rápida. Esto es debido a que al encontrar un puntero es necesario traducir la dirección (desreferenciar) y acceder a su contenido. Esta operación es más costosa. En el caso modificado se utilizan registros para operar en lugar de posiciones de memoria.

7. Recursividad

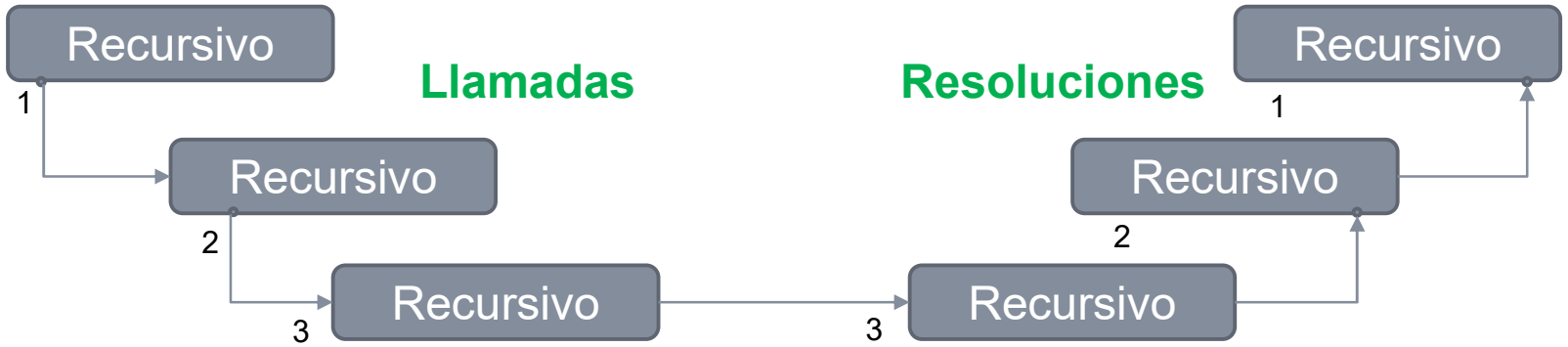
- Es el proceso mediante el cual una función puede llamarse a sí misma.
  - En una función recursiva una sentencia en el cuerpo de la función invoca a la propia función.

- **Las llamadas recursivas se utilizan en problemas iterativos**(repetitivos) en los que cada iteración se determina en función de un resultado anterior.

- Una función recursiva se define estableciendo claramente:
  - Caso base:
    - Caso para el cual la función puede proporcionar un resultado de manera directa.
  - Caso recursivo:
    - Caso para el cual la función resuelve el problema específico en función de la resolución del mismo problema en un caso más sencillo (tamaño de problema menor).

7. Recursividad

- Es necesario una sentencia if que en algún momento obligue a la función a salir de las llamadas recursivas.(CUIDADO CON BUCLES INFINITOS).



7. Ejemplo simple de recursividad: El factorial

Ejemplo: Factorial de un natural.

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

Ejemplos simples de recursividad.

A) Cálculo del factorial de un número, por ejemplo, 5.

5! = 5 * 4!

4! = 4 * 3!

3!= 3 * 2!

2! = 2 * 1!

DESCOMPOSICIÓN DEL PROBLEMA

1! = 1 * 0!

SOLUCIÓN CONOCIDA O DIRECTA **0! = 1**

1! = 1 * 0! = 1

2! = 2 * 1! = 2

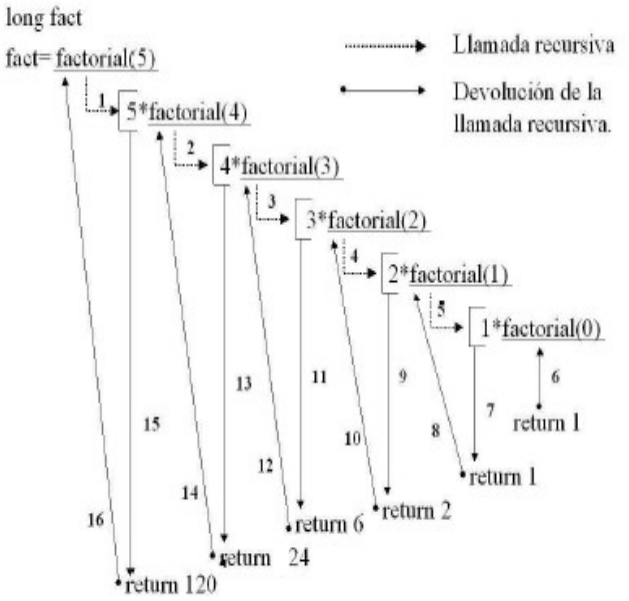
3! = 3 * 2! = 6

4! = 4 * 3! = 24

RESOLUCIÓN DE PROBLEMAS MÁS COMPLEJOS A PARTIR DE OTROS MÁS SIMPLES

5! = 5 * 4! = 120

Ejemplo: Traza del factorial de 5.



7. Ejemplo simple de recursividad: El factorial

```
#include <stdio.h>

int calculofactorial(int N);

int main(void)
{
  int result, numero;

  printf("Intruduce el numero para hallar el factorial:\n");
  scanf("%i", &numero);

  result = calculofactorial (numero);

  printf("El factorial del numero dado es: %i\n", result);

  return 0;
}
```

```
int calculofactorial(int n)
{
   int res;
   if (n == 0)
      res = 1;
   else
      res = n*calculofactorial(n-1);
   return res;

}
```

7. Recursividad y la pila del ordenador

- Ya sabemos que al entrar en una función se crean sus variables locales y sus argumentos y al salir de ésta, estas variables se destruyen.
- Hay que tener en cuenta que, en un programa, una función puede llamar a otras funciones, éstas a su vez pueden llamar a otras y así sucesivamente.
- Por tanto, es necesario disponer de un mecanismo que permita a cada función tener su trozo de memoria para almacenar sus variables.
- Este mecanismo se consigue mediante una zona de memoria, denominada la pila, que se reserva en el ordenador para este propósito y un registro interno en la CPU, denominado SP (stack pointer) que almacena la dirección del último valor que se ha guardado en esta zona de la pila.

7. Recursividad y la pila del ordenador

- La memoria de un ordenador, y siempre que a la hora de ejecutar un programa no se utilice memoria dinámica, queda dividida en dos partes:
  1. la zona donde se almacena el código del programa
  2. la zona donde se guardan los datos: pila (utilizada también para llamadas recursivas).
- Llamada a función
  1. Programa principal llama a una función,
  2. Se crea en la pila un registro de activación, que es un entorno o trozo de memoria que almacena las constantes, variables locales y parámetros formales.
  3. Estos registros se van apilando conforme se llaman sucesivamente desde una función a otras.
  4. Cuando finaliza la ejecución, se va liberando el espacio

7. Recursividad y la pila del ordenador

Ejemplo funcionamiento de la pila

Supongamos una función "main" en la que se definen 3 variables: "a", "b" y "c".
Desde la función "main" se invoca la función "Func1", en la que se definen las variables "v1" y "v2"
Desde "Func1" se invoca la función "Func2" en la que se define la variable "j"

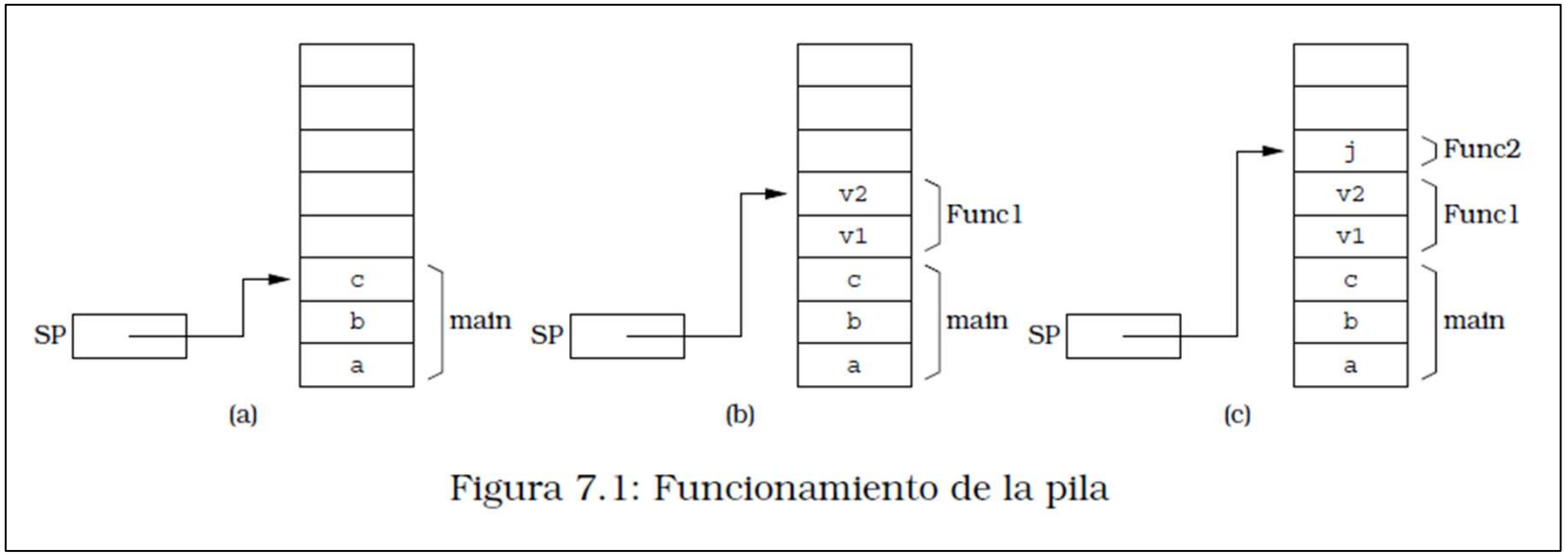


Figura 7.1: Funcionamiento de la pila

7. Recursividad y la pila del ordenador

Ejemplo funcionamiento de la pila

- La figura anterior muestra en primer lugar el estado de la pila cuando empieza a ejecutarse la función **main**, suponiendo que ésta usa tres variables locales denominadas a, b y c. Desde la flecha hacia arriba la memoria está libre. Figura 7.1 (a).
- **Si se llama a una función denominada Func1**, en la que se defunen dos variables, v1 y v2, el estado de la pila durante la ejecución de esta función es el mostrado en la figura 7.1(b). Dichas variables se han “depositado” en el tope de la pila, incrementando el registro SP para que apunte al nuevo límite.
- De este modo la CPU, a través del registro SP puede acceder a las variables de la función. De hecho, las funciones sólo pueden acceder a las variables almacenadas en su trozo de pila (registro de activación).
- **Si la función Func1 llama a la función Func2** y ésta define una variable denominada j, la nueva variable se “depositará” también en el tope de la pila, incrementándose el registro SP de nuevo, tal como se muestra en la figura 7.1(c).
- Importante: mientras se está ejecutando la función Func2 la CPU sólo puede acceder a la variable j. Cuando termine de ejecutarse la función Func2 se “extrae” la variable j de la pila, decrementándose el registro SP para apuntar a las variables de la función Func1. De este modo, la pila queda nuevamente en la situación mostrada en la figura 7.1(b).
- Cuando retorne (finalice) la función Func1 se extraerán las variables v1 y v2 de la pila, volviendo a la situación original mostrada en la figura 7.1(a).
- Destacar que denominar a esta zona de memoria "la pila" viene del hecho de que su funcionamiento, tal como se ha visto, es similar a la de una pila de papeles en donde se añaden nuevas hojas por arriba (cuando se llama a una función) y se retiran las hojas también por arriba (cuando las funciones retornan).

7. Recursividad y la pila del ordenador

- La mecánica de la recursividad, es decir, que una función pueda llamarse a sí misma, está basada en la "pila".

- Cuando un módulo recursivo se está ejecutando se crea en la memoria del ordenador "una pila de recursión" donde se almacenan los valores de los parámetros y de las variables locales del módulo.

- Si el módulo recursivo es una función también se guarda en la pila el valor que devuelve dicha función.

7. Recursividad y la pila del ordenador: factorial de 3

Ejemplo de la evolución de la pila para la ejecución del Factorial de 3:

- La implementación en C es directa a partir de la definición matemática. Lo único a destacar es que en el "else" se llama de nuevo a la función Factorial sin que ello implique nada especial. Para el ordenador es lo mismo llamar a una función con nombre distinto que con el mismo nombre de la función que hace la llamada.
- Para aclarar el proceso, veamos qué ocurre cuando se llama a la función para calcular el factorial de 3, por ejemplo haciendo desde la función main:

```
int calculofactorial(int n)
{
   int fact;
   if (n == 0)
      fact = 1;
   else
      fact = n*calculofactorial(n-1);
   return fact;

}
```

res = calculofactorial (3);

7. Recursividad y la pila del ordenador: factorial de 3

Ejemplo de la evolución de la pila para la ejecución del Factorial de 3:

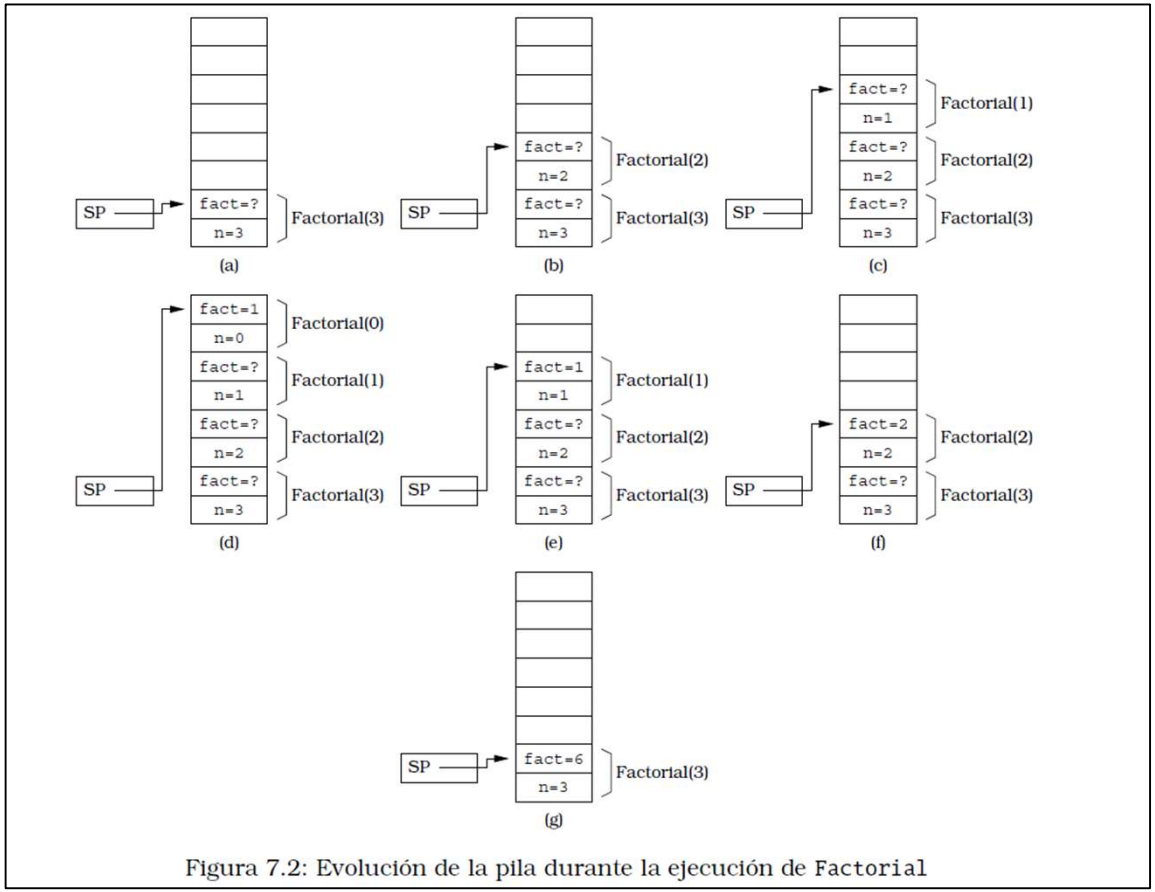


Figura 7.2: Evolución de la pila durante la ejecución de `Factorial`

7. Recursividad y la pila del ordenador: factorial de 3

Ejemplo de la evolución de la pila para la ejecución del Factorial de 3

- Cuando se invoque la función **calculofactorial(3)**, el programa saltará a ejecutar dicha función, pasándole como argumento un 3. La función creará espacio en la pila para almacenar el argumento n y su variable local fact, que de momento estará sin inicializar. La situación de la pila será la mostrada en la figura 7.2(a).

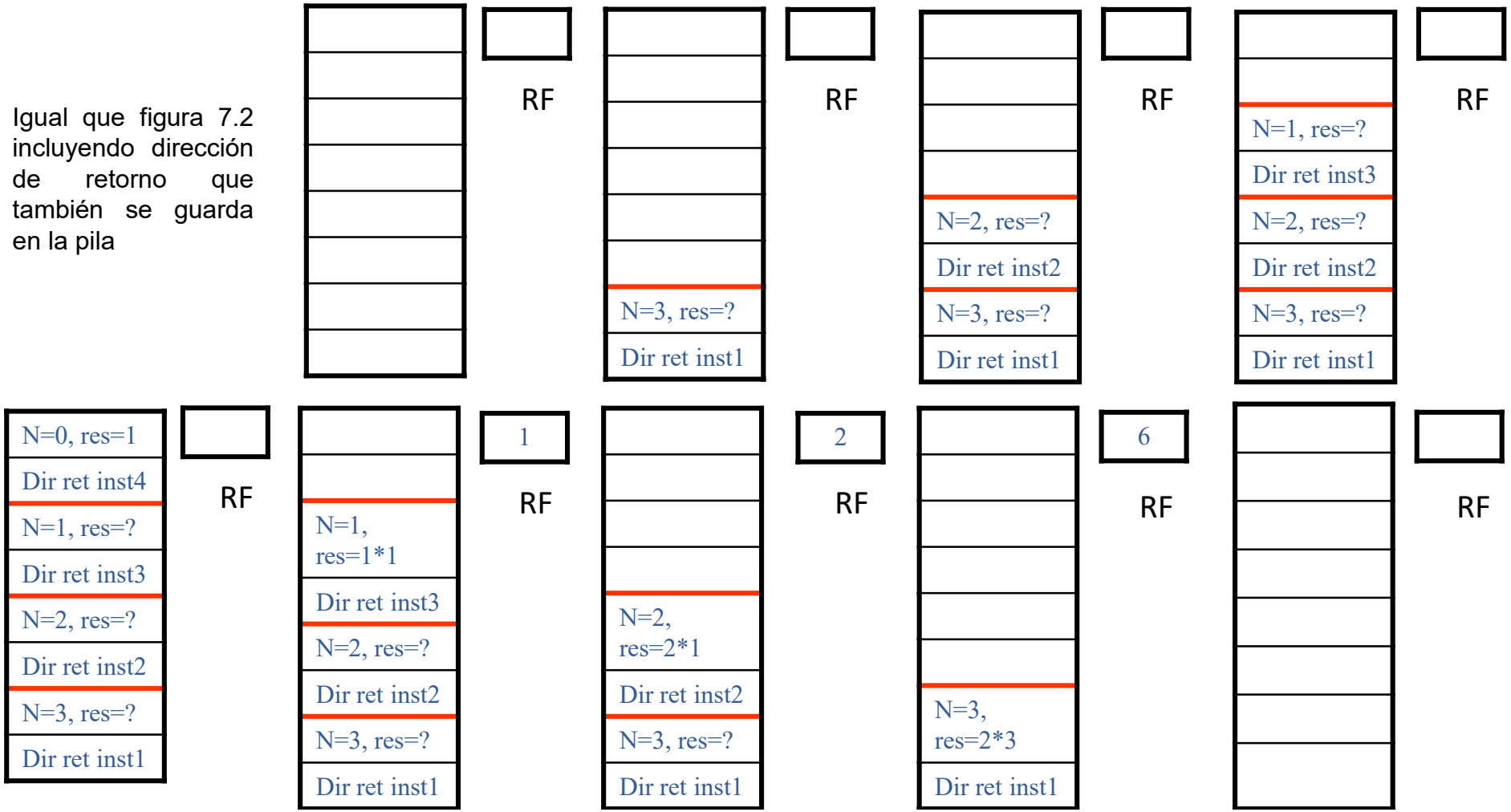
- Cuando se ejecute la línea del "else" de la función, se evaluará la expresión que consiste en multiplicar n (que vale 3) por el resultado de ejecutar la función calculofactorial con 2 como argumento. Por tanto, la CPU volverá a llamar a dicha función, creándose ahora en la pila otras dos variables para la nueva instancia de la función, una para su argumento, que ahora vale 2, y otra para su variable local, que al igual que antes estará sin inicializar. La situación de la pila será la mostrada en la figura 7.2(b).

- El proceso se repetirá hasta que se llame a calculofactorial con un argumento igual a 0, tal como se muestra en las figuras 7.2(c) y 7.2(d). En esta última llamada –calculofactorial(0)–, se ejecutará la línea del "if" y se asignará el resultado 1 a la variable fact, de esta instancia de la función calculofactorial tal como se muestra en la figura 7.2(d).

- Este valor se retornará a quien la había llamado, que no era más que otra instancia de ella misma pero con argumento 1. Por tanto, se terminará de ejecutar la línea del "else" de esa instancia de la función, obteniéndose como resultado un 1 que se asigna a la variable fact de esta instancia de la función. La situación de la pila en ese momento es la mostrada en la figura 7.2(e). calculofactorial(1) terminará de ejecutarse, devolviendo un 1 a calculofactorial(2). El proceso se repetirá, finalizándose la ejecución de la línea del "else" de las instancias en curso de calculofactorial, obteniéndose el valor de la variable fact y retornándola. La evolución de la pila se muestra en las figuras 7.2(f) y 7.2(g).

7. Recursividad y la pila del ordenador: factorial de 3

Igual que figura 7.2 incluyendo dirección de retorno que también se guarda en la pila



7. Recursividad frente a iteratividad

- Todas las funciones recursivas tienen que tener una condición de salida (en este ejemplo cuando n vale 0) y debe estar garantizado que la llamada recursiva siempre se realiza con distintos argumentos. Si no se cumplen estas condiciones, el algoritmo será infinito y el programa sólo terminará cuando se agote la memoria del ordenador.

- El proceso de recursividad necesita gran cantidad de memoria para almacenar en la pila los argumentos y las variables locales. Además, las sucesivas llamadas recursivas a la función consumen tiempo de CPU, por lo que el proceso en general es **ineficiente**.

- Esto hace que **casi siempre sea mejor implantar un algoritmo de forma iterativa que de forma recursiva**, siempre y cuando dicho algoritmo exista. Por ejemplo, en el caso del factorial, puede usarse en lugar del algoritmo anterior, el siguiente algoritmo iterativo.

```
long int FactorialIte(long int n)
{
    long int fact; /* valores parciales de factorial */
    int i; /* Índice para el cálculo del factorial */
    fact = 1;
    for(i=1; i<=n; i++){
        fact = fact * i;
     }
   return fact;
}
```

7. Recursividad. Ejercicios

Ejercicio 1:

Escriba un programa para calcular la serie de Fibonacci de un número n. Dicha serie se obtiene según la fórmula:

$$F(n) = \begin{cases} 0 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ F(n-1) + F(n-2) \text{ si } n > 1 \end{cases}$$

Para ello creará una función que calcule la serie y un programa principal que pedirá al usuario el número n, calculará la serie llamando a la función e imprimirá el resultado. Deberán escribirse dos versiones de la función, una recursiva y otra iterativa.

7. Recursividad. Ejercicios

Ejercicio 2:

La función exponencial, x^y siendo x e y números enteros, puede implementarse de forma recursiva, dado que $x^y = x * x^{y-1}$. Implementa una función recursiva que calcule la función exponencial.

7. Recursividad. Ejercicios

Ejercicio 3:

Mediante la estrategia conocida como "divide y vencerás" nos podemos ahorrar un número de multiplicaciones considerable para calcular la potencia de un número, x^y. Para ello:

- Si **y** es PAR, tenemos que $x^y = x^{y/2} * x^{y/2}$
- Si **y** es IMPAR, tenemos que $x^y = x^{y/2} * x^{y/2} * x$

- Implementa otra función recursiva que calcule la exponencial con este método y añade un contador de llamadas recursivas para comprobar el número de multiplicaciones ahorradas por este método.


U-tad
CENTRO UNIVERSITARIO
DE TECNOLOGÍA Y ARTE DIGITAL