

10

Introducción a la recursión

Grados en Ingeniería Informática, Ingeniería
del Software e Ingeniería de Computadores

Miguel Gómez-Zamalloa Gil
(adaptadas del original de Luis Hernández Yáñez y Ana Gil)

Facultad de Informática
Universidad Complutense



Índice

Concepto de recursión	2
Algoritmos recursivos	5
Diseño de funciones recursivas	8
Modelo de ejecución	9
Tipos de recursión	37
Recursión simple	38
recursión múltiple	39
Recursión anidada	41
Recursión cruzada	45
Código del subprograma recursivo	46
Parámetros y recursión	51
Ejemplos de algoritmos recursivos	55
Búsqueda binaria	56
Torres de Hanoi	59
Recursión frente a iteración	64
Estructuras de datos recursivas	66



Fundamentos de la programación

Recursión

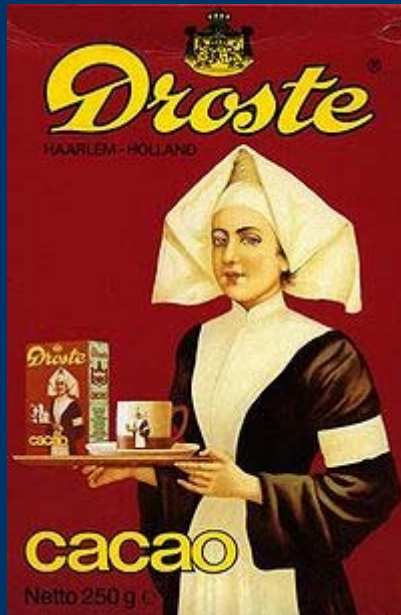


Concepto de recursión

Recursión, recursividad o recurrencia

Algo se define de forma recursiva cuando en la definición aparece lo que se está definiendo:

$$\text{Factorial}(N) = N \times \text{Factorial}(N-1) \quad (N \geq 0)$$



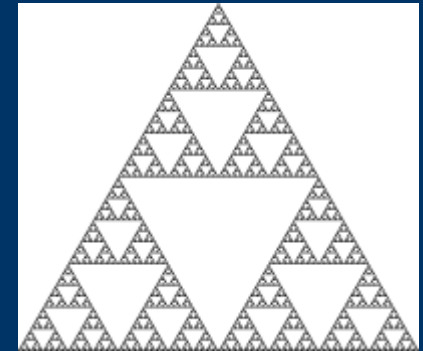
(wikipedia.org)



(http://farm1.staticflickr.com/83/229219543_edf740535b.jpg)

La imagen del paquete aparece dentro del propio paquete, que a su vez contiene otra imagen del paquete... ¡hasta el infinito!

Cada triángulo está formado por otros triángulos más pequeños.



(wikipedia.org)



Las matrioskas rusas



Concepto de recursión

$$\text{Factorial}(N) = N \times \text{Factorial}(N-1)$$

El factorial se define en función de sí mismo

Para calcular el factorial de N hay que calcular antes el de $N-1$

Los programas no pueden manejar la recursión infinita

La definición incluirá uno o más casos base

Caso base: punto final de cálculo (no se usa la definición recursiva)

$$\text{Factorial}(N) \begin{cases} 1 & \text{si } N = 0 & \text{Caso base (o de parada)} \\ N \times \text{Factorial}(N-1) & \text{si } N > 0 & \text{Caso recursivo (inducción)} \end{cases}$$

Cada vez que se aplica el caso recursivo,
el valor de N se va aproximando al valor del caso base (0)



Fundamentos de la programación

Algoritmos recursivos



Algoritmos recursivos

Funciones recursivas

Una función puede implementar un algoritmo recursivo:

La función se llamará a sí misma si no se ha llegado al caso base

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

```
long long int factorial(int n) {  
    long long int resultado;  
    if (n == 0) // Caso base  
        resultado = 1;  
    else  
        resultado = n * factorial(n - 1);  
  
    return resultado;  
}
```



Algoritmos recursivos

```
long long int factorial(int n) {  
    long long int resultado;  
  
    if (n == 0) // Caso base  
        resultado = 1;  
    else  
        resultado = n * factorial(n - 1);  
  
    return resultado;  
}
```

$\text{factorial}(5) \rightarrow 5 \times \text{factorial}(4) \rightarrow 5 \times 4 \times \text{factorial}(3)$
 $\rightarrow 5 \times 4 \times 3 \times \text{factorial}(2) \rightarrow 5 \times 4 \times 3 \times 2 \times \text{factorial}(1)$
 $\rightarrow 5 \times 4 \times 3 \times 2 \times 1 \times \text{factorial}(0) \rightarrow 5 \times 4 \times 3 \times 2 \times 1 \times 1 \rightarrow 120$
Caso base

```
D:\FP\Tema11>factorial  
1  
1  
2  
6  
24  
120  
720  
5040  
40320  
362880  
3628800  
39916800  
479001600  
6227020800  
87178291200  
1307674368000  
20922789888000  
355687428096000  
6402373705728000  
121645100408832000
```



Diseño de subprogramas recursivos

Subprogramas recursivos bien diseñados:

- ✓ Caso(s) base: debe haber al menos un caso base de parada
- ✓ Inducción: paso recursivo que provoca una llamada se deberá demostrar correcto
- ✓ Convergencia: cada paso debe acercarse al caso base

Describimos problemas en términos de problemas *más pequeños*

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

- ✓ La función `factorial()` tiene caso base ($N = 0$)
- ✓ Siendo correcta para N es correcta para $N+1$ (*inducción*)
- ✓ Se acerca al caso base ($N-1$ está más cerca de 0 que N)



Fundamentos de la programación

Modelo de ejecución



Modelo de ejecución

```
long long int factorial(int n) {  
    long long int resultado;  
    if (n == 0) // Caso base  
        resultado = 1;  
    else  
        resultado = n * factorial(n - 1);  
    return resultado;  
}
```

Llamada recursiva: nueva ejecución de la función

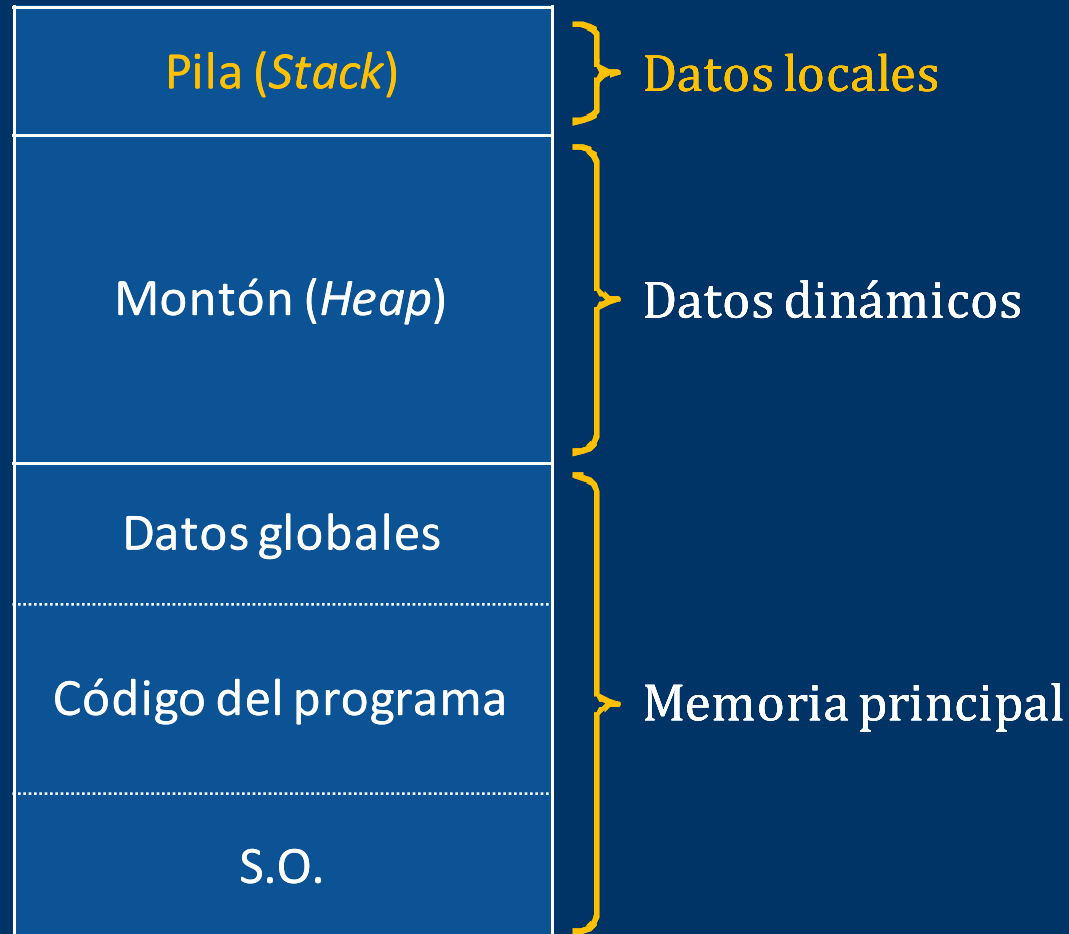
Queda interrumpida la llamada actual

Cada llamada, sus propios parámetros y variables locales (n y resultado en este caso)

En cada llamada se utiliza la pila del sistema para mantener los datos locales y la dirección de vuelta



La pila del sistema (*stack*)



Memoria dinámica (Tema 9)

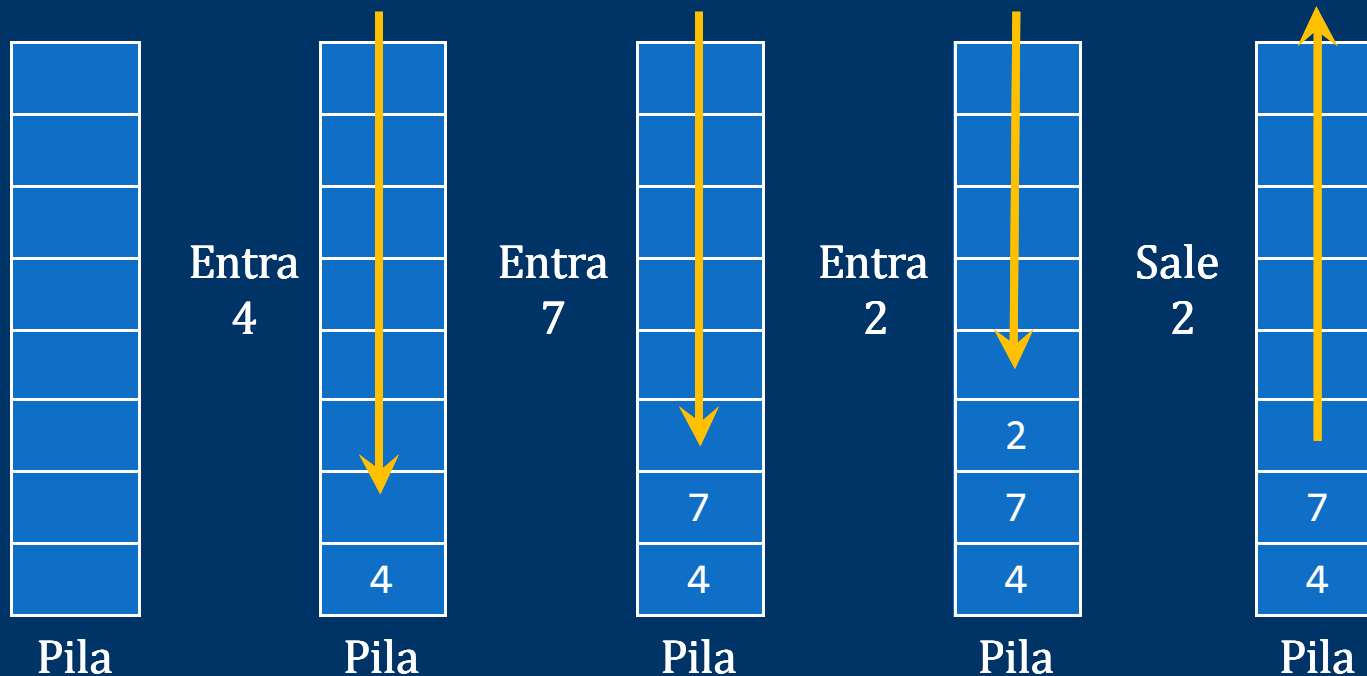


La pila del sistema (*stack*)

Se guardan los datos locales y la dirección de vuelta

Es una estructura de tipo *pila*: lista LIFO (*last-in first-out*)

El último que entra es el primero que sale:



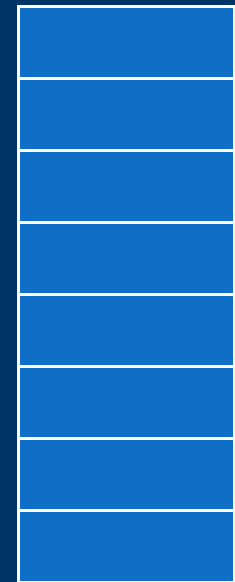
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...
```

Llamada a función:
Se guarda la dirección de vuelta



Pila



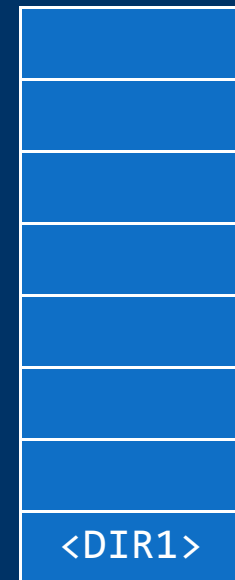
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...  
}
```

← Entrada en la función:
Se alojan los datos locales



Pila



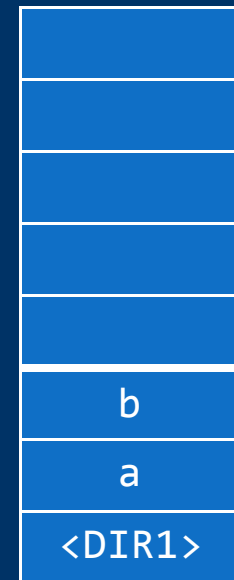
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...  
}
```

Llamada a función:
Se guarda la dirección de vuelta



Pila

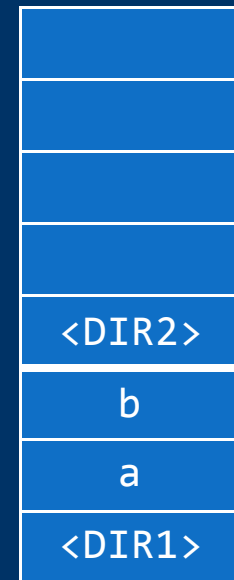


La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) { ← Entrada en la función:  
    ...                Se alojan los datos locales  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...
```



Pila



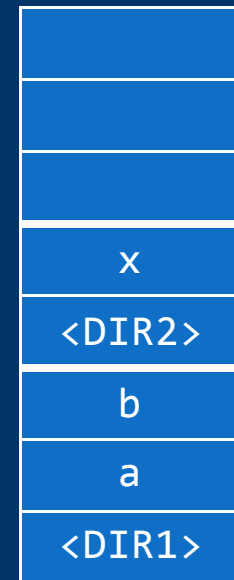
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...  
}
```

Vuelta de la función:
Se eliminan los datos locales



Pila



La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    cout << funcA(4);
    ...
}
```

Vuelta de la función:
Se obtiene la dirección de vuelta



Pila



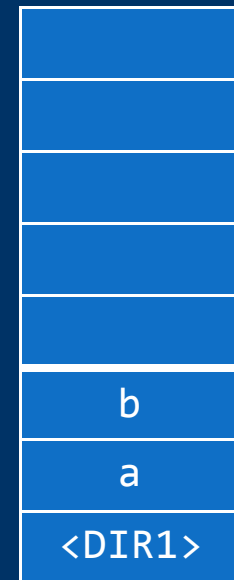
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...  
}
```

La ejecución continúa en esa dirección



Pila

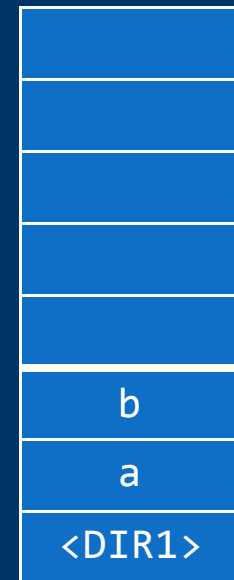


La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b; ← Vuelta de la función:  
              Se eliminan los datos locales  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...  
}
```



Pila



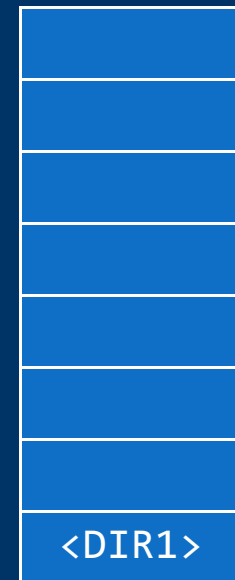
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...  
}
```

← Vuelta de la función:
Se obtiene la dirección de vuelta



Pila



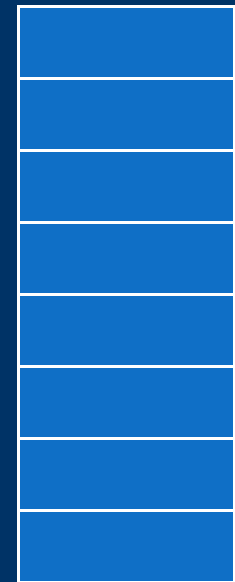
La pila y las llamadas a subprogramas

Llamada a subprograma:

Se alojan en la pila sus datos locales y la dirección de vuelta

```
...  
int funcB(int x) {  
    ...  
    return x;  
}  
int funcA(int a) {  
    int b;  
    ...  
<DIR2>    b = funcB(a);  
    ...  
    return b;  
}  
int main() {  
    ...  
<DIR1>    cout << funcA(4);  
    ...
```

La ejecución continúa
en esa dirección

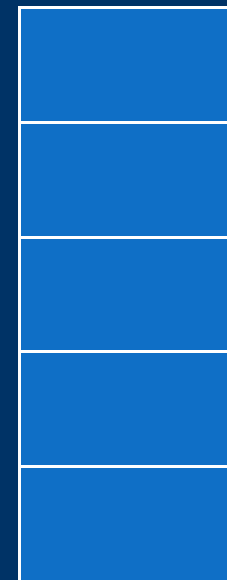
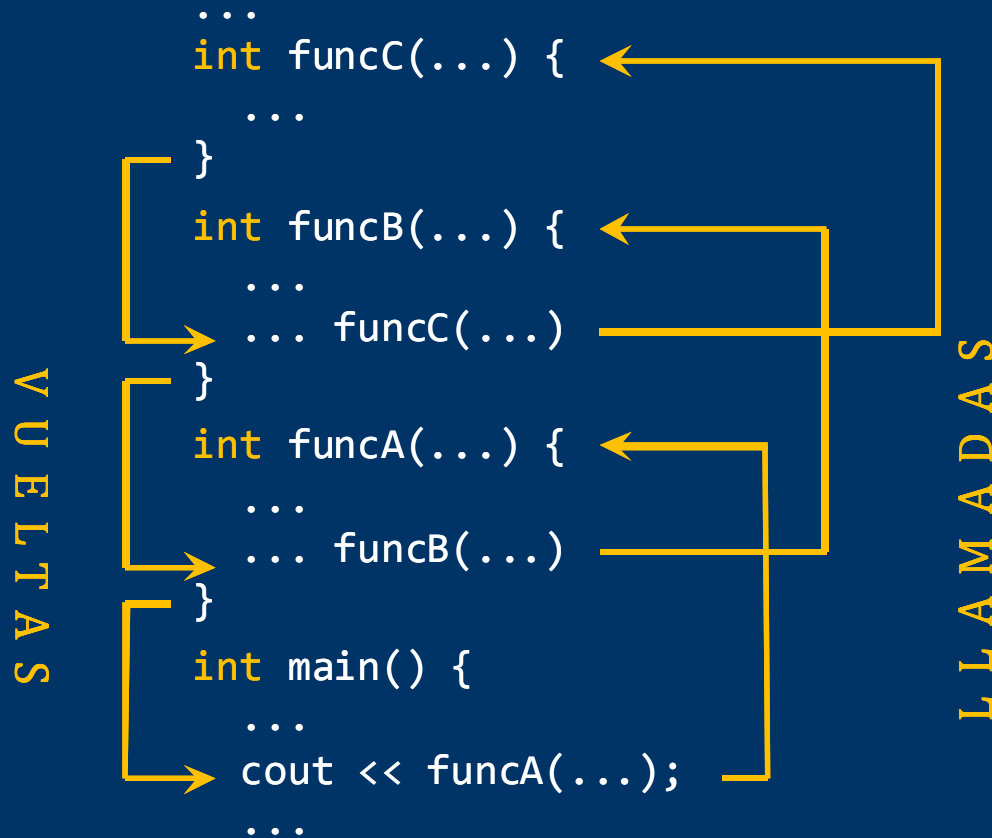


Pila



La pila y las llamadas a subprogramas

Mecanismo tipo *LIFO*: adecuado para anidamiento de llamadas
Los subprogramas terminan en orden contrario a como se llaman



Pila



Ejecución de la función factorial()

```
long long int factorial(int n) {  
    long long int resultado;  
  
    if (n == 0) // Caso base  
        resultado = 1;  
    else  
        resultado = n * factorial(n - 1);  
  
    return resultado;  
}
```

Datos locales: n y resultado

Ejecución de factorial(5) (obviando direcciones de vuelta)



Ejecución de la función factorial()

factorial(5)

resultado = ?
n = 5

Pila



Ejecución de la función factorial()

factorial(5)
└─> factorial(4)

resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

factorial(5)
 ↳ factorial(4)
 ↳ factorial(3)

resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

factorial(5)
 ↳ factorial(4)
 ↳ factorial(3)
 ↳ factorial(2)

resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

factorial(5)
 ↳ factorial(4)
 ↳ factorial(3)
 ↳ factorial(2)
 ↳ factorial(1)

resultado = ?
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

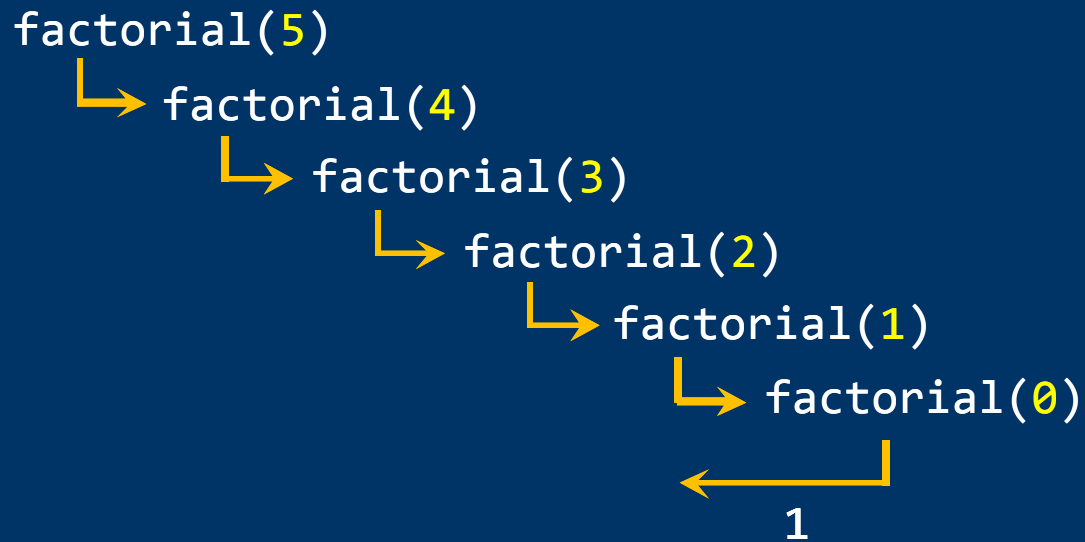
factorial(5)
 ↳ factorial(4)
 ↳ factorial(3)
 ↳ factorial(2)
 ↳ factorial(1)
 ↳ factorial(0)

resultado = 1
n = 0
resultado = ?
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

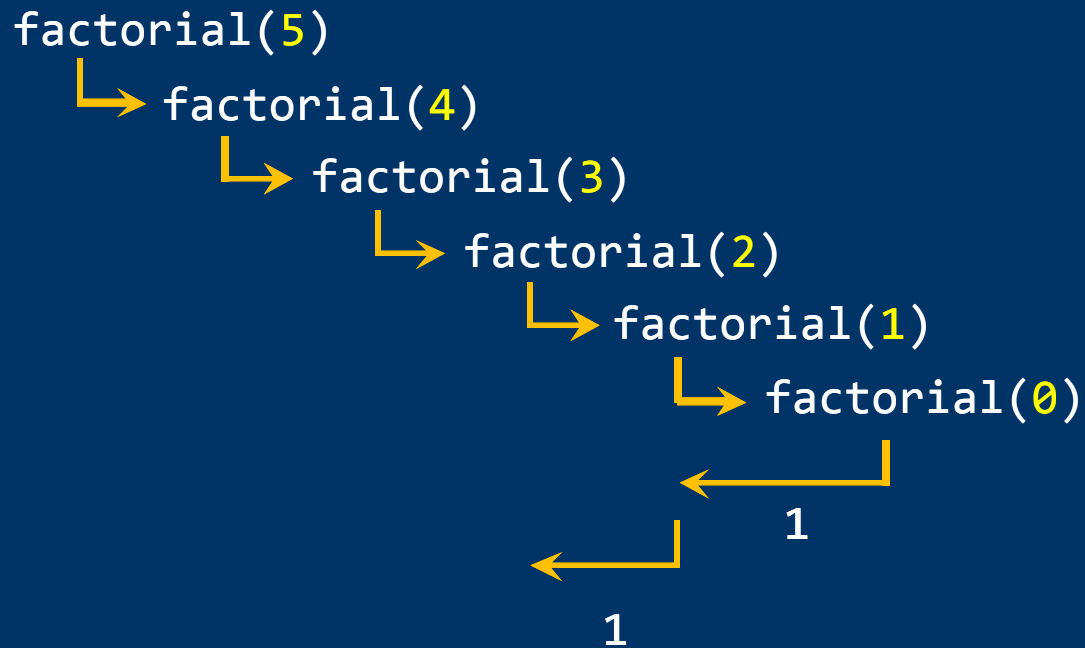


resultado = 1
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

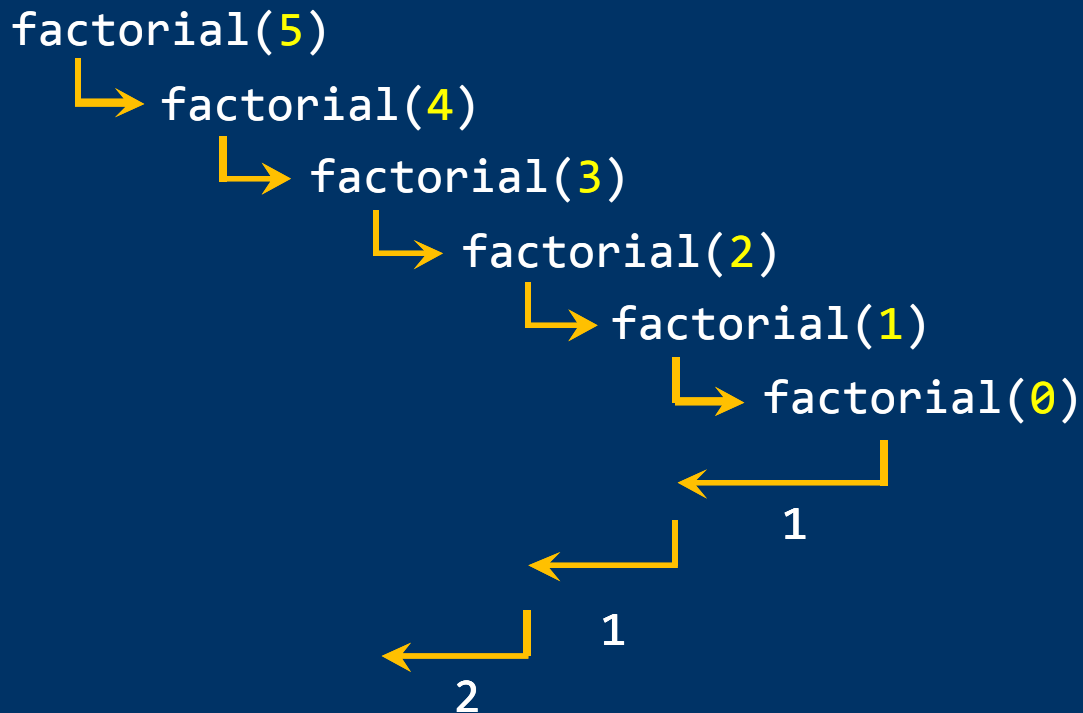


resultado = 2
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

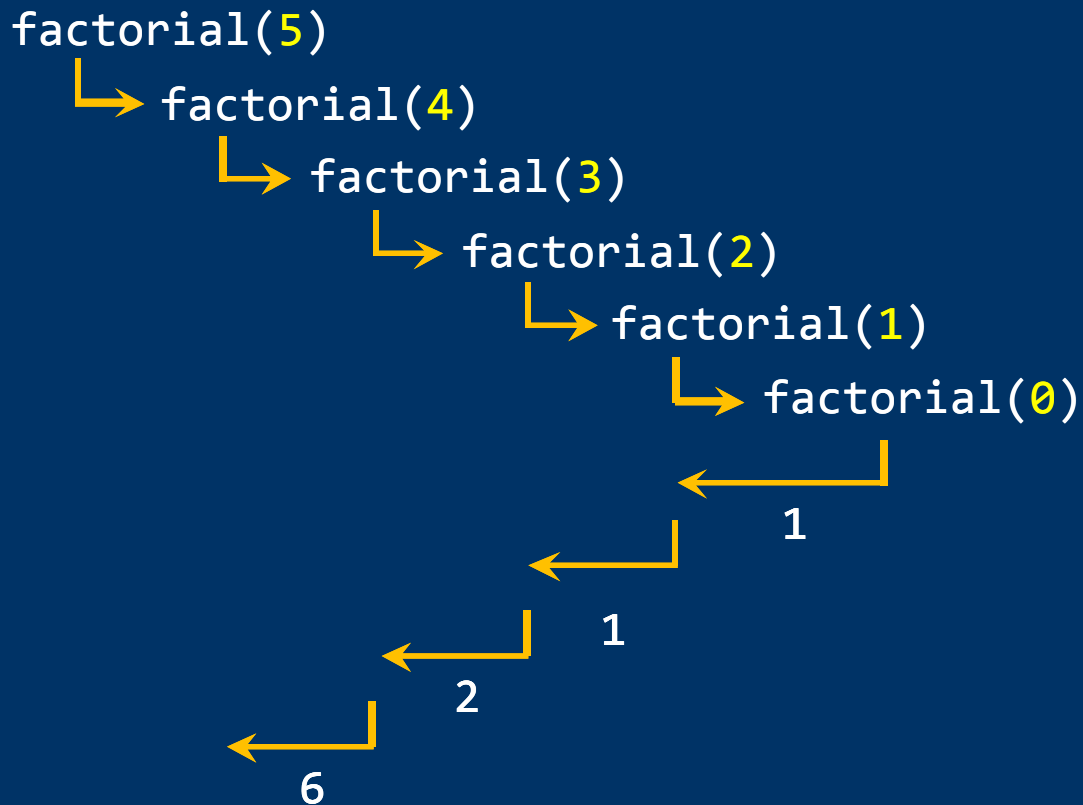


resultado = 6
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

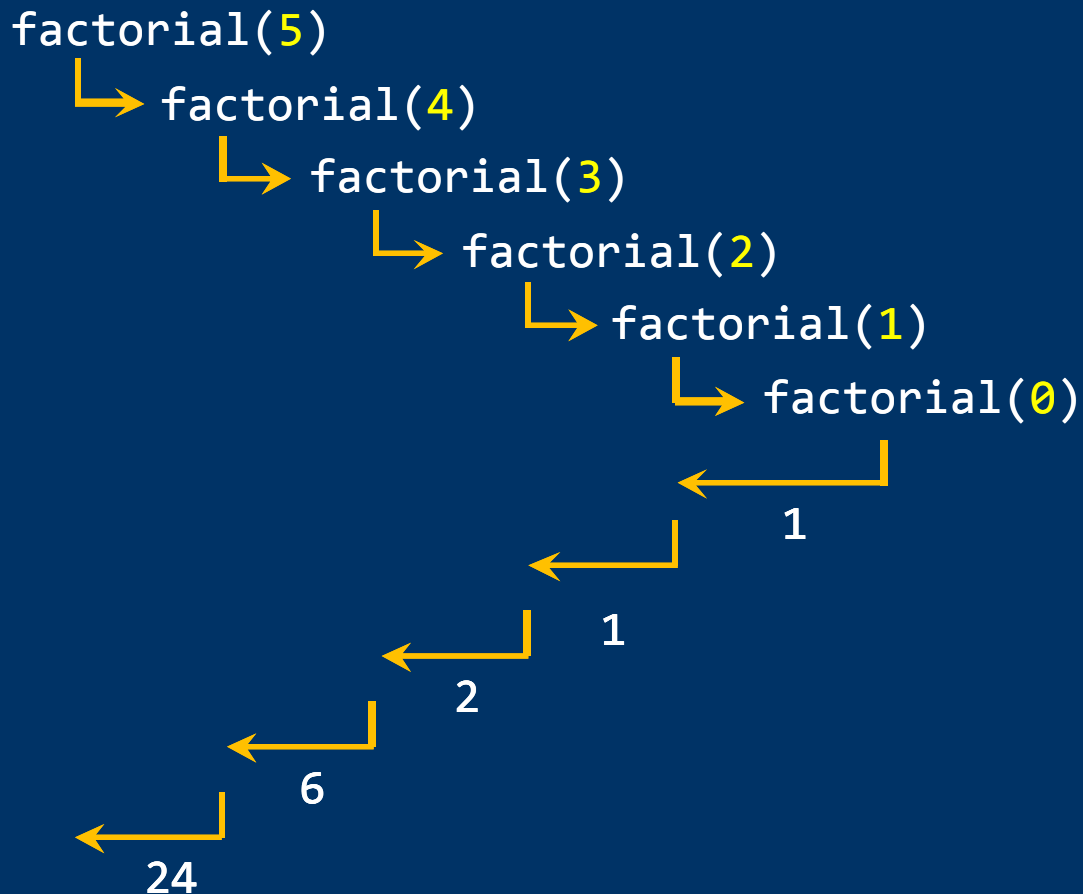


resultado = 24
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()

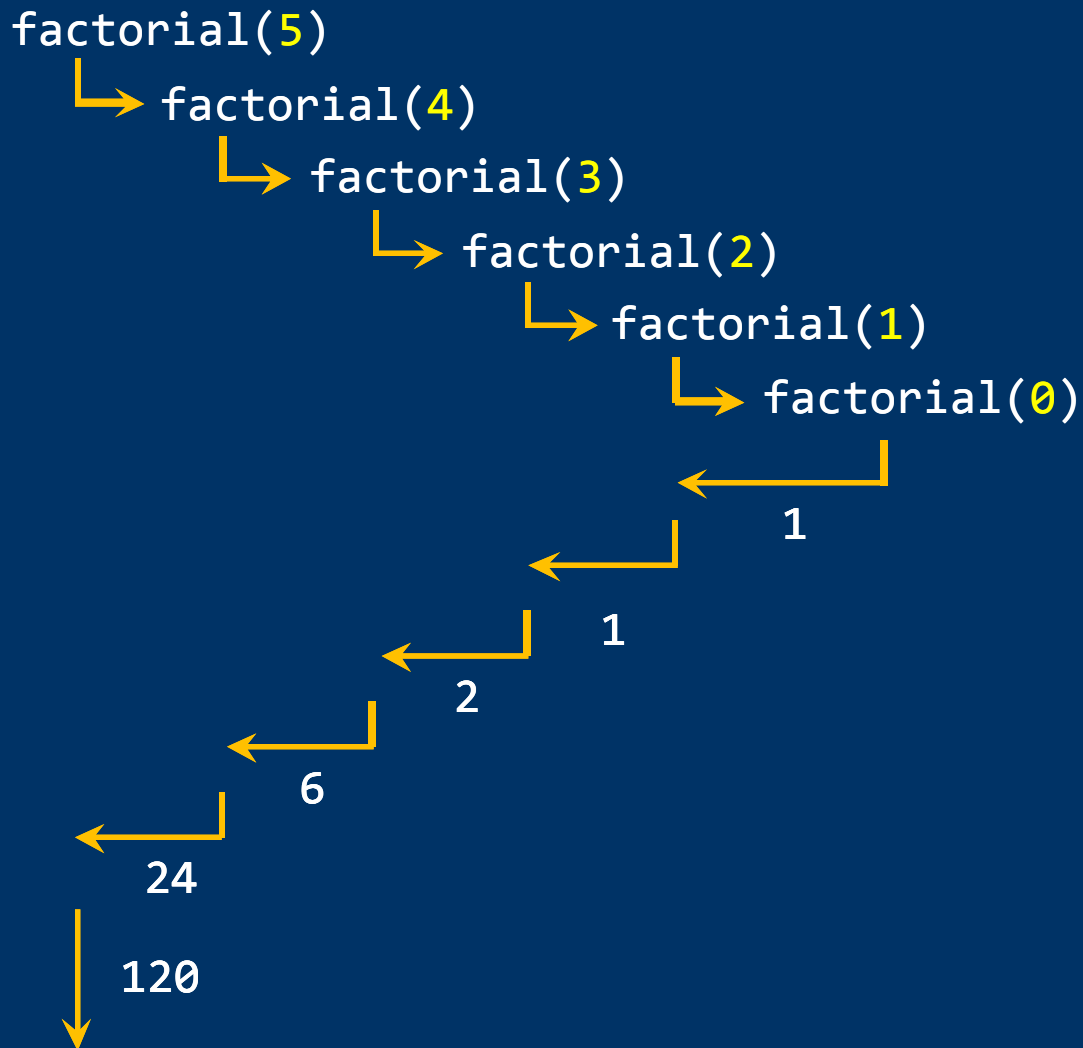


resultado = 120
n = 5

Pila



Ejecución de la función factorial()



Pila



Fundamentos de la programación

Tipos de recursión



Tipos de recursión

Recursión simple

Sólo hay una llamada recursiva

Ejemplo: cálculo del factorial de un número entero positivo

```
long long int factorial(int n) {  
    long long int resultado;  
  
    if (n == 0) // Caso base  
        resultado = 1;  
    else  
        resultado = n * factorial(n - 1);  
  
    return resultado;  
}
```

Una llamada recursiva



Tipos de recursión

Recursión múltiple

Varias llamadas recursivas

Ejemplo: cálculo de los números de *Fibonacci*

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

Dos llamadas recursivas



Los números de *Fibonacci*

```
...  
int fibonacci(int n) {  
    int resultado;  
    if (n == 0)  
        resultado = 0;  
    else if (n == 1)  
        resultado = 1;  
    else  
        resultado = fibonacci(n - 1)  
                    + fibonacci(n - 2);  
    return resultado;  
}
```

Fib(n) $\left\{ \begin{array}{ll} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{si } n > 1 \end{array} \right.$

```
int main() {  
    for (int i = 0; i < 20; i++)  
        cout << fibonacci(i) << endl;  
  
    return 0;  
}
```

```
D:\FP\Tema11>fibonacci  
0  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89  
144  
233  
377  
610  
987  
1597  
2584  
4181
```



Tipos de recursión

Recursión anidada

En una llamada recursiva hay argumentos que son llamadas recursivas

Ejemplo: cálculo de los números de *Ackermann*

$$\text{Ack}(m, n) \begin{cases} n + 1 & \text{si } m = 0 \\ \text{Ack}(m-1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$



Argumento que es una llamada recursiva



Los números de Ackermann

...

```
int ackermann(int m, int n) {  
    int resultado;  
    if (m == 0)  
        resultado = n + 1;  
    else if (n == 0)  
        resultado = ackermann(m - 1, 1);  
    else  
        resultado = ackermann(m - 1, ackermann(m, n - 1));  
    return resultado;  
}
```

$$\text{Ack}(m, n) \begin{cases} n + 1 & m = 0 \\ \text{Ack}(m-1, 1) & m > 0, n = 0 \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & m > 0, n > 0 \end{cases}$$

```
int main() {  
    int m, n;  
    cout << "M = "; cin >> m;  
    cout << "N = "; cin >> n;  
    cout << ackermann(m, n) << endl;  
    return 0;  
}
```

Pruébalo con números muy bajos
Se generan MUCHAS llamadas recursivas



Los números de Ackermann

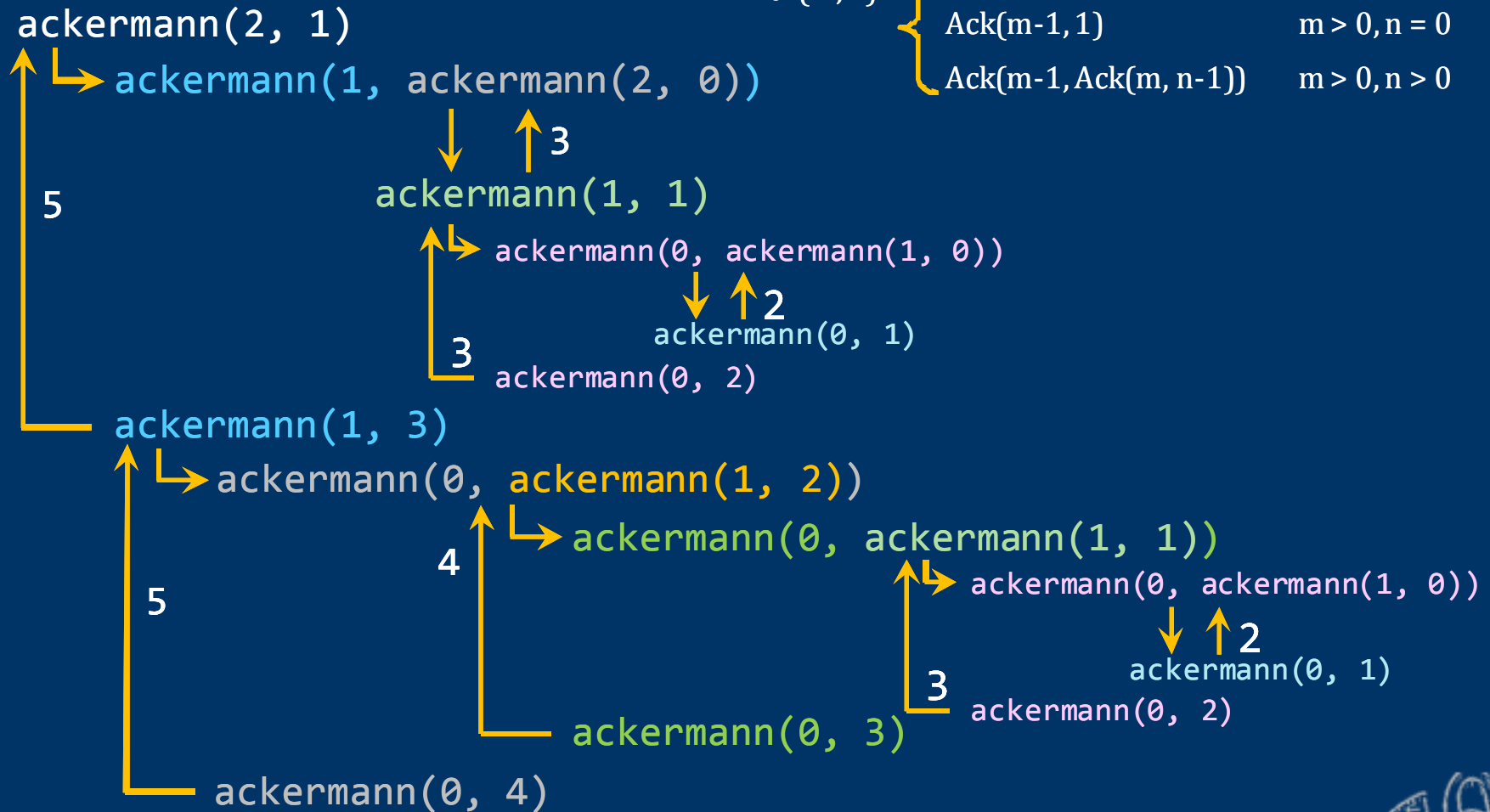
ackermann(1, 1)
└─> ackermann(0, ackermann(1, 0))
└─> 3
└─> ackermann(0, 2)
└─> ackermann(0, 1) (via 2)

$Ack(m, n)$ $\begin{cases} n + 1 & m = 0 \\ Ack(m-1, 1) & m > 0, n = 0 \\ Ack(m-1, Ack(m, n-1)) & m > 0, n > 0 \end{cases}$



Los números de Ackermann

$$\text{Ack}(m, n) \begin{cases} n + 1 & m = 0 \\ \text{Ack}(m-1, 1) & m > 0, n = 0 \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & m > 0, n > 0 \end{cases}$$



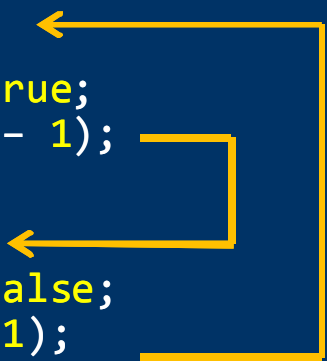
Tipos de recursión

Recursión cruzada o indirecta

La recursión cruzada o indirecta se da cuando una función llama a otra y ésta a su vez llama a la primera.

Por ejemplo, saber si un número es par o impar:

```
bool par(int n) {  
    if (n == 0) return true;  
    else return impar(n - 1);  
}  
  
bool impar(int n) {  
    if (n == 0) return false;  
    else return par(n - 1);  
}  
  
int main() {  
    int x;  
    cout << "Num: "; cin >> x;  
    if (par(x)) cout << "Es par";  
    else cout << "Es impar";  
    ...  
}
```

A diagram with two yellow arrows illustrating cross-recursion. One arrow starts from the 'else return impar(n - 1);' line in the 'par' function and points to the 'bool impar(int n)' function definition. The other arrow starts from the 'else return par(n - 1);' line in the 'impar' function and points back to the 'bool par(int n)' function definition.

Fundamentos de la programación

Código del subprograma recursivo



Código del subprograma recursivo

Código anterior y posterior a la llamada recursiva

Recursión simple:

```
{  
    Código anterior  
    Llamada recursiva  
    Código posterior  
}
```

El *código anterior* se ejecuta repetidas veces hasta el caso base.

Se ejecuta en orden directo para las distintas entradas.

El *código posterior* se ejecuta repetidas veces tras el caso base.

Se ejecuta en orden inverso para las distintas entradas.

Recursión por delante: si no hay código anterior

Recursión por detrás (final): si no hay código posterior



Código anterior y posterior a la llamada

```
void func(int n) {  
    if (n > 0) {  
        cout << "Entrando (" << n << ")" << endl; // Código anterior  
        func(n - 1);                               // Llamada recursiva  
        cout << "Saliendo (" << n << ")" << endl; // Código posterior  
    }  
}
```

func(5);

El código anterior a la llamada
se ejecuta para los sucesivos valores de n

El código posterior a la llamada, al revés

```
D:\FP\Tema11>prueba  
Entrando (5)  
Entrando (4)  
Entrando (3)  
Entrando (2)  
Entrando (1)  
Saliendo (1)  
Saliendo (2)  
Saliendo (3)  
Saliendo (4)  
Saliendo (5)
```



Recursión por delante

Recorrido de una lista (directo)

Procesamos el elemento antes de la llamada recursiva:

```
...  
void mostrar(const tLista &lista, int pos);  
  
int main() {  
    tLista lista;  
    lista.cont = 0;  
    // Carga del array...  
    mostrar(lista, 0);  
  
    return 0;  
}  
  
void mostrar(const tLista & lista, int pos) {  
    if (pos < lista.cont) {  
        cout << lista.elementos[pos] << endl;  
        mostrar(lista, pos + 1);  
    }  
}
```

```
D:\FP\Tema11>directo  
1  
3  
8  
13  
17  
22  
23  
39  
52  
55
```



Recursión por detrás

Recorrido de una lista (inverso)

Procesamos el elemento después de la llamada recursiva:

...

```
void mostrar(const tLista & lista, int pos);
```

```
int main() {  
    tLista lista;  
    lista.cont = 0;  
    // Carga del array...  
    mostrar(lista, 0);  
  
    return 0;  
}
```

```
void mostrar(const tLista & lista, int pos) {  
    if (pos < lista.cont) {  
        mostrar(lista, pos + 1);  
        cout << lista.elementos[pos] << endl;  
    }  
}
```

```
D:\FP\Tema11>inverso  
55  
52  
39  
23  
22  
17  
13  
8  
3  
1
```



Fundamentos de la programación

Parámetros y recursión



Parámetros y recursión

Parámetro acumulador por valor (recursión final)

Para construir el resultado, a lo largo de las distintas llamadas, se utilizan parámetros (denominados acumuladores) donde se va realizando la operación posterior a la llamada recursiva:

```
int factorial(int n, int fact) { // devuelve fact*n!  
    if (n == 0) return fact;  
    else return factorial(n - 1, n * fac)  
}
```

La llamada inicial debe ser `factorial(m, 1)`, `fact` toma el valor 1, y se va multiplicando por el `n` de la llamada.

`factorial(2,1) -> factorial(1,2) -> factorial(0,2) -> 2`



Parámetros y recursión

Parámetro acumulador por referencia (final)

Para construir el resultado, a lo largo de las distintas llamadas, también se pueden usar parámetros por referencia (misma variable en las llamadas):

```
void factorial(int n, int &fact) { // calcula fact*n!  
    if (n > 0) { // parámetro de entrada/salida  
        fact = n * fact;  
        factorial(n - 1, fact);  
    }  
}
```

La llamada inicial debe ser `r = 1; factorial(m, r)`, `r` tiene el valor 1, y se va multiplicando por el `n` de la llamada.

`r = 1;`

<code>factorial(2,r)</code>	<code>-> factorial(1,r)</code>	<code>-> factorial(0,r)</code>
<code>r = 2 * r</code>	<code>r = 1 * r</code>	



Parámetros y recursión

Parámetro acumulador por referencia (no final)

Para construir el resultado, a lo largo de las distintas llamadas, también se pueden usar parámetros por referencia (misma variable en las llamadas):

```
void factorial(int n, int &fact){ // parámetro de salida para n!
    if (n == 0) fact = 1;
    else {
        factorial(n - 1, fact);
        fact = n * fact;
    }
}
```

La llamada inicial debe ser `factorial(m, r)`, en el caso base, `r` toma el valor 1, y a la vuelta, se le multiplica por el `n` de la llamada anterior.

`factorial(2,r) -> factorial(1,r) -> factorial(0,r)`
`r = r * 2 <- r = r * 1 <- r = 1`



Fundamentos de la programación

Ejemplos de algoritmos recursivos



Búsqueda binaria recursiva

Partiendo el problema en subproblemas más pequeños:
El segmento [ini .. fin] de búsqueda se reduce a la mitad...

Si no queda lista (caso base)... terminar (lista vacía: no encontrado)

En caso contrario...

Comparar el elemento en la mitad con el buscado

Si el buscado es menor que el elemento mitad...

Quedarse con la primera mitad de la lista y repetir el proceso

Si el elemento mitad es menor que el buscado...

Quedarse con la segunda mitad de la lista y repetir el proceso

En otro caso (encontrado)... terminar (caso base)

Empezaremos con la lista completa en la que hay que buscar



Búsqueda binaria recursiva

Añadir a los parámetros de entrada:

- ✓ Índice del inicio del segmento de búsqueda (*ini*)
- ✓ Índice del final del segmento de búsqueda (*fin*)

```
bool buscar(const tLista &lista, int buscado,  
            int ini, int fin, int & pos)  
// Devuelve la posición y si se ha encontrado
```

¿Cuáles son los casos base?

- ✓ Que ya no quede segmento ($ini > fin$) → No encontrado
- ✓ Que se encuentre el elemento



Búsqueda binaria recursiva

```
bool buscar(const tLista &lista, int buscado,
            int ini, int fin, int & pos) {
    bool enc;
    if (ini <= fin) {
        int mitad = (ini + fin) / 2;
        if (buscado < lista.elementos[mitad])
            enc = buscar(lista, buscado, ini, mitad - 1, pos);
        else if (lista.elementos[mitad] < buscado)
            enc = buscar(lista, buscado, mitad + 1, fin, pos);
        else { pos = mitad; enc = true; } // encontrado
    }
    else { pos = ini; enc = false; } // (ini > fin) -> no encontrado
    return enc;
}
```

Llamada inicial:

```
bool enc = buscar(lista, buscado, 0, lista.cont - 1, pos);
```



Búsqueda binaria recursiva

```
bool buscar(const tLista &lista, int buscado,
            int ini, int fin, int & pos) {
    if (ini <= fin) {
        pos = (ini + fin) / 2;
        if (buscado < lista.elementos[pos])
            return buscar(lista, buscado, ini, pos - 1, pos);
        else if (lista.elementos[pos] < buscado)
            return buscar(lista, buscado, pos + 1, fin, pos);
        else return true; // encontrado
    }
    else {pos = ini; return false;} //(ini > fin) -> no encontrado
}
```

Llamada inicial:

```
if (buscar(lista, buscado, 0, lista.cont - 1, pos)) ...
```



Las torres de Hanoi

Cuenta una leyenda que en un templo de Hanoi se dispusieron tres pilares de diamante y en uno de ellos 64 discos de oro, de distintos radios y colocados por orden de tamaño con el mayor debajo.



Torre de ocho discos (wikipedia.org)

Cada monje, en su turno, mueve un único disco de un pilar a otro, para, con el tiempo, conseguir llevar la torre entera a uno de los otros dos pilares. Sólo hay una regla: no poner un disco sobre otro de menor tamaño. Cuando se haya conseguido, se acaba el mundo.

[Se requieren al menos $2^{64}-1$ movimientos; si se hiciera uno por segundo, se concluiría en más de 500 mil millones de años]

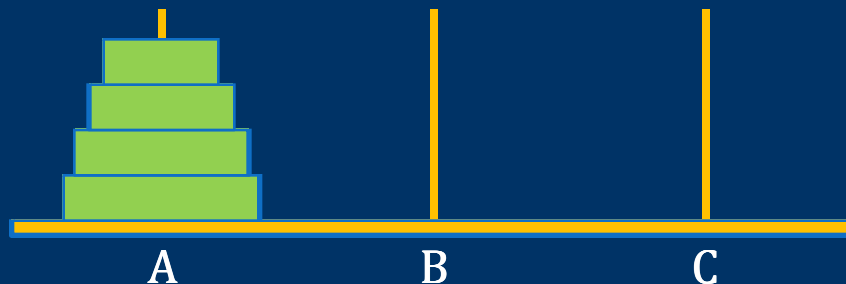


Las torres de Hanoi

Menor número de movimientos posibles

¿Qué disco hay que mover en cada paso y a dónde?

Identifiquemos los elementos (torre de cuatro discos):



Cada pilar se identifica con una letra

Mover del pilar X al pilar Y :

Coger el disco superior de X y ponerlo encima de los de Y



Las torres de Hanoi

Resolver el problema en base a problemas más pequeños...

Mover N discos del pilar A al pilar C:

Mover N-1 discos del pilar A al pilar B

Mover el disco del pilar A al pilar C

Mover N-1 discos del pilar B al pilar C

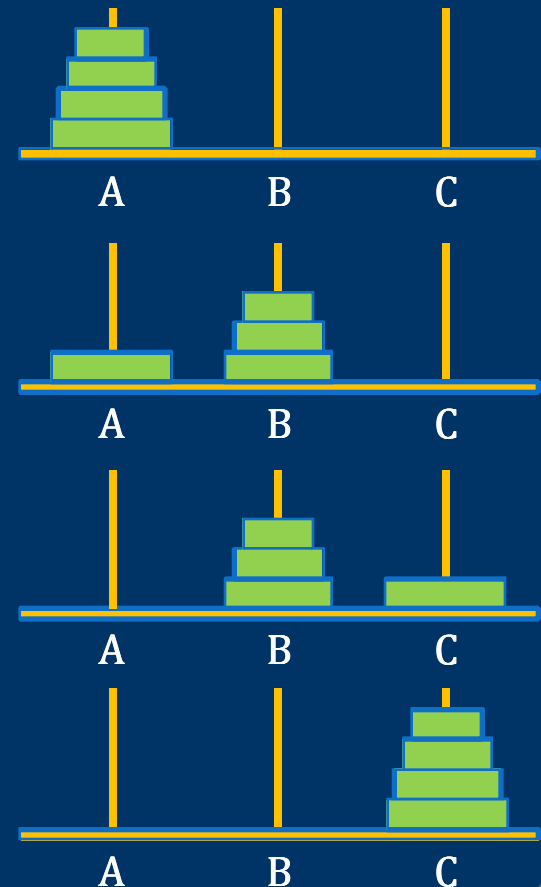
El tercer pilar se usa como *auxiliar*:

Mover N-1 discos del *origen* al *auxiliar*

Mover el disco del *origen* al *destino*

Mover N-1 discos del *auxiliar* al *destino*

Mover 4 discos de A a C



Las torres de Hanoi

Mover N-1 discos: igual, pero con otros pilares como origen y destino

Mover N-1 discos del pilar A al pilar B:

Mover N-2 discos del pilar A al pilar C

Mover el disco del pilar A al pilar B

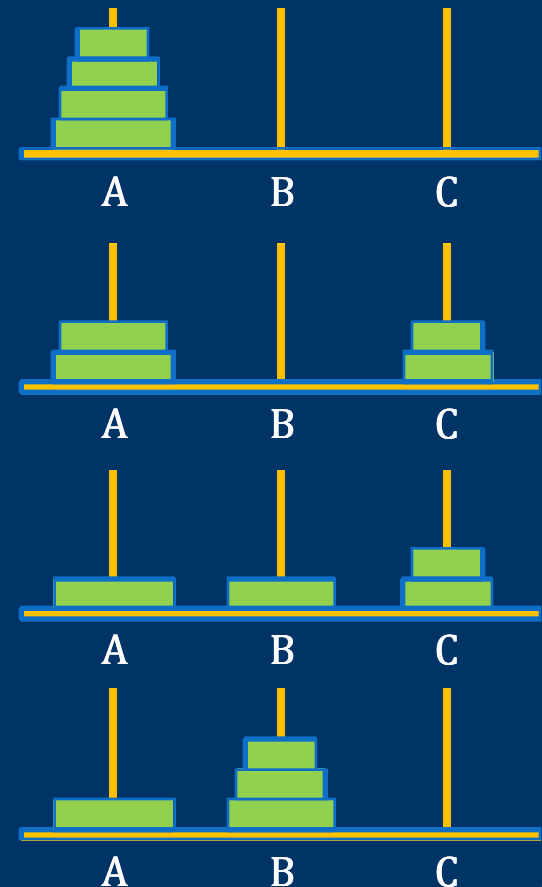
Mover N-2 discos del pilar C al pilar B

Sencilla implementación recursiva



Simulación para 4 discos (wikipedia.org)

Mover 3 discos de A a B



Las torres de Hanoi

Caso base: no quedan discos que mover

```
...  
void hanoi(int n, char origen, char destino, char auxiliar) {  
    if (n > 0) {  
        hanoi(n - 1, origen, auxiliar, destino);  
        cout << origen << " --> " << destino << endl;  
        hanoi(n - 1, auxiliar, destino, origen);  
    }  
}  
  
int main() {  
    int n;  
    cout << "Número de discos: ";  
    cin >> n;  
    hanoi(n, 'A', 'C', 'B');  
  
    return 0;  
}
```

```
D:\FP\Tema11>hanoi  
N. torres: 4  
A --> B  
A --> C  
B --> C  
A --> B  
C --> A  
C --> B  
A --> B  
A --> C  
B --> C  
B --> A  
C --> A  
B --> C  
A --> B  
A --> C  
B --> C
```



Recursión frente a iteración



Recursión frente a iteración

¿Qué es preferible?

Siempre hay una alternativa iterativa para un algoritmo recursivo
Algoritmo recursivo: menos eficiente que su alternativa iterativa
Si resulta sencillo desarrollar una versión iterativa, será preferible
En ocasiones no resulta sencillo obtener esa versión iterativa
Preferible una versión recursiva si es mucho más simple
Desarrolla una versión iterativa para los números de Fibonacci
¿Y qué tal una para los números de Ackermann?



Fundamentos de la programación

Estructuras de datos recursivas



Estructuras de datos recursivas

*Definición recursiva de estructuras lineales o secuenciales
(flujos, listas en arrays, listas enlazadas)*

Secuencia { elemento seguido de una secuencia
secuencia vacía (ningún elemento)

Secuencia 1, 2, 3: elemento 1 seguido de la secuencia 2, 3

Secuencia 2, 3: elemento 2 seguido de la secuencia 3

Secuencia 3: elemento 3 seguido de la secuencia vacía (caso base)

Hay otras estructuras de datos con naturaleza recursiva (2º curso)



Estructuras de datos recursivas

Procesamiento de estructuras recursivas

Procesar (secuencia):

Si secuencia vacía (caso base):

En otro caso (secuencia no vacía):

Procesar el primer elemento // Código anterior

Procesar (resto(secuencia))

Procesar el primer elemento // Código posterior

resto(secuencia): secuencia sin su primer elemento



Estructuras de datos recursivas

Procesamiento de estructuras recursivas

Mostrar(secuencia):

Si secuencia No vacía

Mostrar el primer elemento // Código anterior

Mostrar (resto(secuencia))

En otro caso (secuencia vacía)

MostrarInversa(secuencia):

Si secuencia No vacía

Mostrar (resto(secuencia))

Mostrar el primer elemento // Código posterior

En otro caso (secuencia vacía)



Estructuras de datos recursivas

Procesamiento de estructuras recursivas

```
void mostrar( const tLista & list, int p) {
    if ( p < list.cont ) {
        cout << list[p]; // primer elemento //código anterior
        mostrar (list, p+1) // resto de la secuencia
    }
}
// llamada inicial: mostrar(lista, 0);
void mostrarInversa( ifstream & ent) {
    ent >> dato;
    if ( !ent.fail()) {
        mostrar (ent); // resto de la secuencia
        cout << dato; // primer elemento // código posterior
    }
}
// llamada: ent.open("..."); mostrar(ent); ent.close();
```



Referencias bibliográficas



- ✓ *C++: An Introduction to Computing* (2ª edición)
J. Adams, S. Leestma, L. Nyhoff. Prentice Hall, 1998
- ✓ *El lenguaje de programación C++* (Edición especial)
B. Stroustrup. Addison-Wesley, 2002
- ✓ *Programación en C++ para ingenieros*
F. Xhafa et al. Thomson, 2006








Licencia CC (Creative Commons)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
-  No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.
-  Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.

