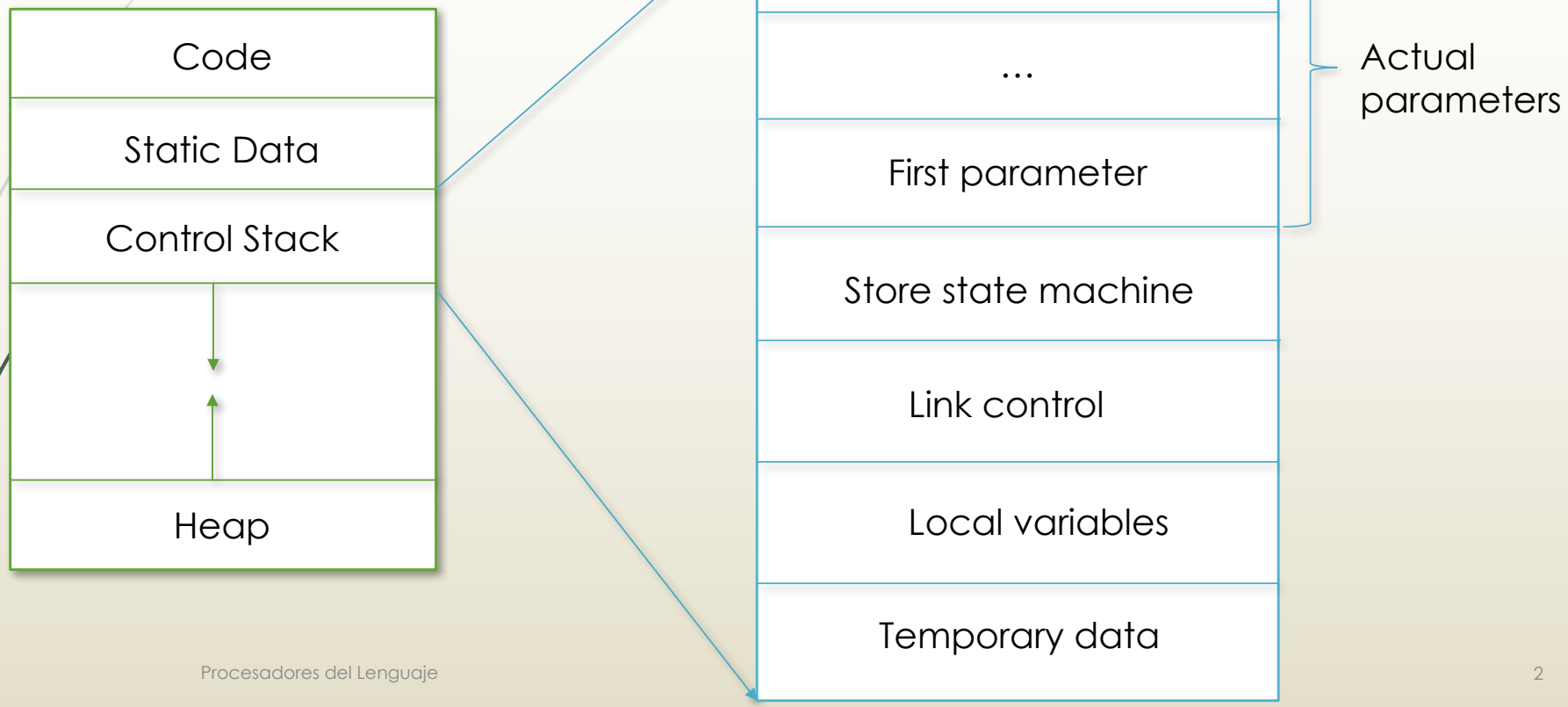
Item 2: Runtime memory

Mari Paz Guerrero Lebrero

Grado en Ingeniería Informática

Curso 2018/2019

Previous concepts



Calling sequence

- When a function is activated, you will follow a strict protocol consisting of:
  - Pass parameters
  - Save the processor status
  - Establish link
- The first half of this code is responsibility of the **caller** and the other half is responsibility of the **process called**.

Restoration sequence

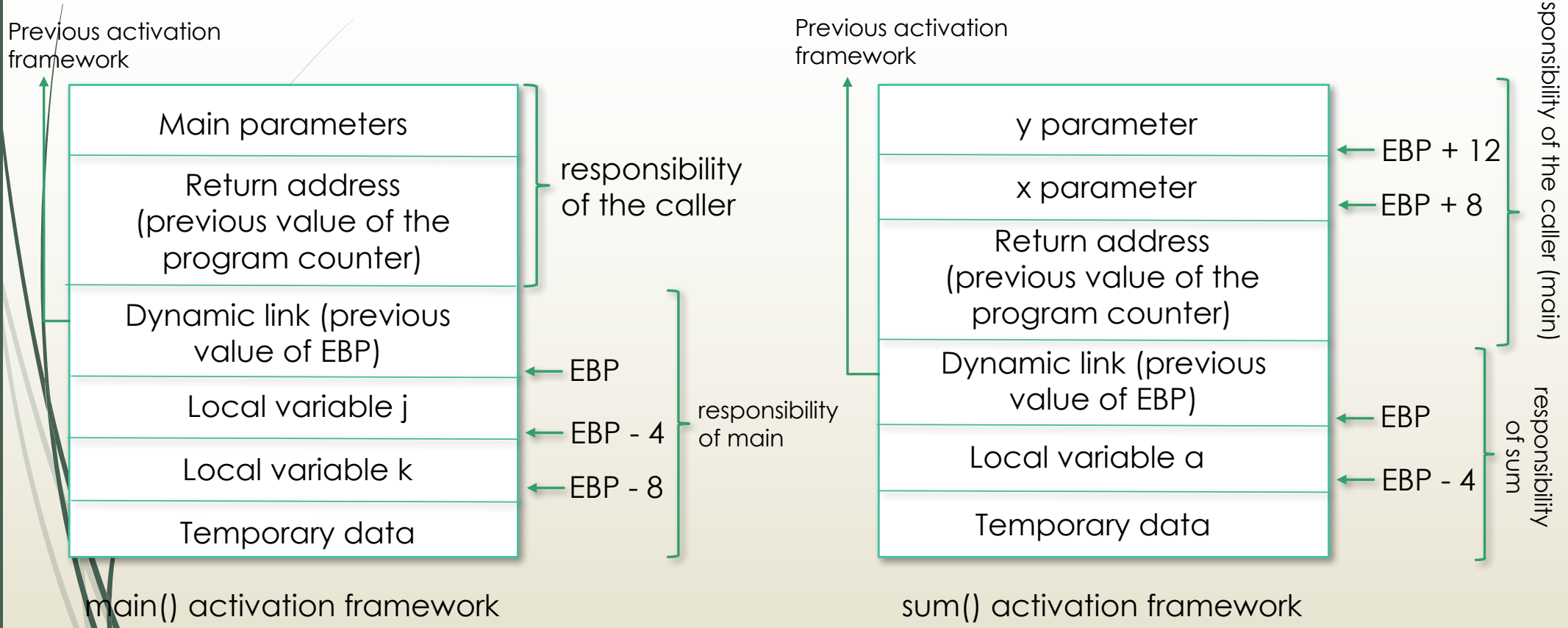
- When the activation is finished, you will follow other protocol consisting of:
  - Return a value
  - Establish links
  - Remove parameters
- The first half of this code is responsibility of the **process called** and the other half is responsibility of the **caller**.

Example

```
int n;

int sum(int x, int y)
{
        int a;
        a = x * 2 + y;
        return a;
}//end of suma()
```

```
int main()
{
        int j, k;
        printf("Give me two integers:");
        scanf("%d%d", &j, &k);
        n = sum(j, k);
        printf("The addition of %d * 2 plus %d is %d\n", j, k, n);
        return 0;
}//end of main()
```

Example


Previous activation framework
Main parameters
Return address (previous value of the program counter)
Dynamic link (previous value of EBP)
Local variable j
Local variable k
Temporary data
responsibility of the caller
EBP
EBP - 4
EBP - 8
responsibility of main
main() activation framework
Previous activation framework
y parameter
x parameter
Return address (previous value of the program counter)
Dynamic link (previous value of EBP)
Local variable a
Temporary data
EBP + 12
EBP + 8
EBP
EBP - 4
responsibility of the caller (main)
responsibility of sum
sum() activation framework

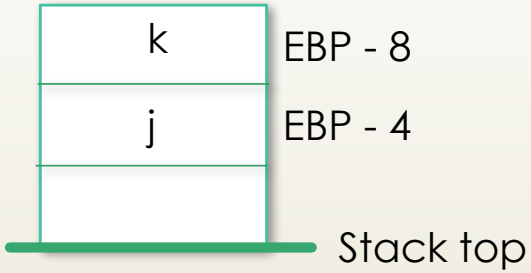
Calling sequence

main function

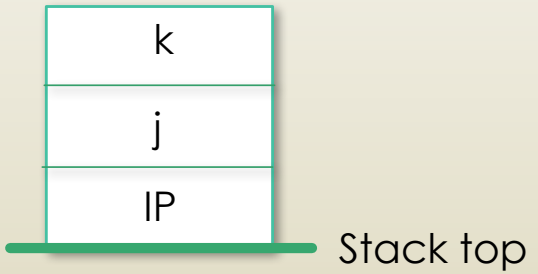
- It is responsibility of caller:
  1. Assess the current parameters and push it to the stack from right to left (first parameter is the closest to the position indicated by EBP)

```
n = sum(j, k)
```

```
pushl -8(%ebp)
pushl -4(%ebp)
```



  2. Call the function; in our example call sum()
  3. The call instruction of assembler stores the return value in the stack

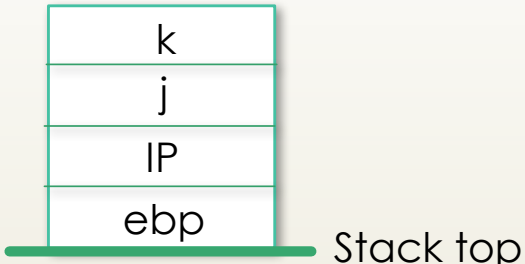


Calling sequence

sum function

- It is responsibility of called:
  1. Set the dynamic link: the value of EBP is stored in the stack and updated at the current top of the stack

```
pushl %ebp
movl %esp %ebp
```



  2. Reserve space for local variable on the stack.

  3. Execute the function body

Restoration sequence

sum function

- It is responsibility of called:
  1. Place the return value in the EAX register

     return a; → `movl -4(%ebp), %eax`

  2. Delete local and temporary variables, restoring the value of ESP before booking variables. `movl %ebp, %esp`
  3. Reset the dynamic link with the previous value stored in the stack. `popl %ebp`
  4. Return control to the program that called re-establishing the Program Counter register with the previous value that was stored on the stack. `ret`

Restoration sequence

main function

- It is responsibility of caller:
  1. Clear the parameters placed on the stack. `addl $16, %esp`
  2. Use the return value (which is in the EAX register) for calculations need to perform. `movl %eax, n`

Other example

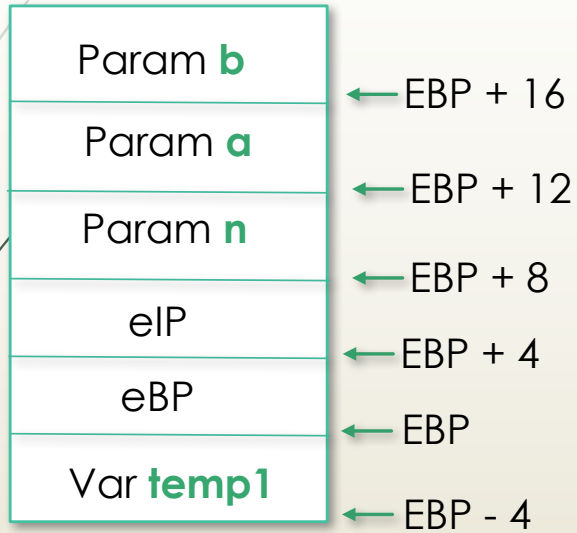
```
#include <stdio.h>
int m;
int fib_iter (int n, int a, int b) {
    int temp1;
    if (n==0)
        return a;
    else {
        temp1 = n - 1;
        return fib_iter(temp1, b, a + b);
    }
}
```

```
int fib(int n) {
    return fib_iter(n, 0, 1);
}

int main () {
    int valor;
    printf("Dame un entero positivo ");
    scanf("%d", &m);
    valor = fib(m);
    printf("El fibonacci de %d vale
            %d\n", m, valor);
    return 0;
}
```

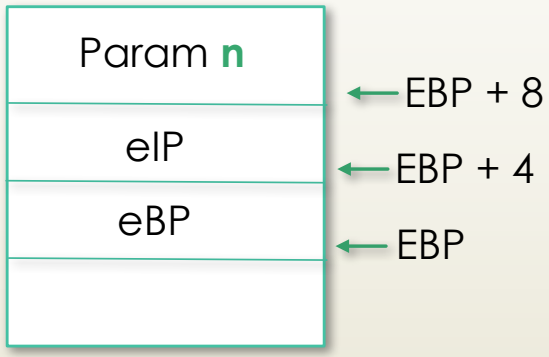
Other example

Activation framework

```
int fib_iter (int n, int a, int b) {int temp1; ...}
```




```
int fib (int n) {...}
```



Other example

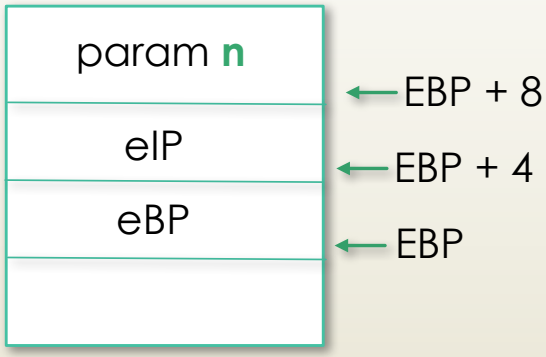
Calling sequence

- It is responsibility of caller:
  1. Evaluate and push the stack parameters from right to left

```
fib_iter(n, 0, 1);
```

```
movl $1, %eax
pushl %eax
movl $0, %eax
pushl %eax
movl 8(%ebp), %eax pushl %eax
```

```
int fib (int n) {...}
```



  2. Call to the function

```
fib_iter(n, 0, 1);
```

```
call fib_iter
```

Other example

Calling sequence

- It is responsibility of called:
  1. Set the dynamic link:

```
pushl %ebp
movl %esp %ebp
```

```
int fib_iter (int n, int a, int b) {
    int temp1;
    ...
 }
```

  2. Reserve space for local variable on the stack.

```
subl $4 %esp
```

```
int fib_iter (int n, int a, int b) {
    int temp1;
    ...
 }
```

  3. Execute the function

Other example

Restoration sequence

- It is responsibility of called:
  1. Save to EAX the return value

  2. Delete local and temporary variables, restoring the value of ESP before booking variables. `movl %ebp, %esp`
  3. Reset the dynamic link with the previous value stored in the stack. `popl %ebp`
  4. Return control to the program that called re-establishing the Program Counter register with the previous value that was stored on the stack. `ret`

Other languages: C vs. Pascal

- In C parameter passing is from right to left. Pascal contrary.Consequences:
  - In C the first parameter is closer to the dynamic link: it is always at the same depth => functions with different number of parameters
  - In Pascal functions have a fixed number of parameters known by the caller and the called
- The responsible role to remove and put parameters: caller in C and Pascal called
- Pascal can return a value on the stack, C does not know how deep it has.
- In Pascal there is "display" array of static links

Persistent dynamic variables

- There are some languages in which activation frames are created, with a new activation, but not erased when the activation function ends.
- These frameworks are created and destroyed in a stack, but they are created on the heap and deleted with the garbage.
- Variables created in these frameworks retain their value even after completion of the activation. They are persistent dynamic variables.

Observe the operation of the following Python code

```
>>> def suma(n) :
    return lambda x: x + n
>>> suma(3)
<function <lambda> at 0x83889cc>
>>> suma(3)(4)
7
>>> sumaTres = suma(3)
>>> sumaTres(4)
7
>>> sumaTres(5)
8
>>> sumaDos = suma(2)
>>> sumaDos(7)
9
>>> sumaDos(5)
7
>>> sumaTres(8)
11
```

Activation of "sum (3)" recalls the ligature n ← 3 even after completion.

The same occurs with the activation of "sum (2)" reminiscent ligation n ← 2.

The same thing in JavaScript

```
function suma(x) {
    return function(y) {
        return x+y;
    }
}

var s2 = suma(2);
var s3 = suma(3);

>>> s2(4);
6
>>> s3(7);
10
```

Activation of "sum (2)" recalls the ligature x ← 2 even after completion.

The same occurs with the activation of "sum (3)" reminiscent ligation x ← 3.

The same thing in C++ (-std=c++11)

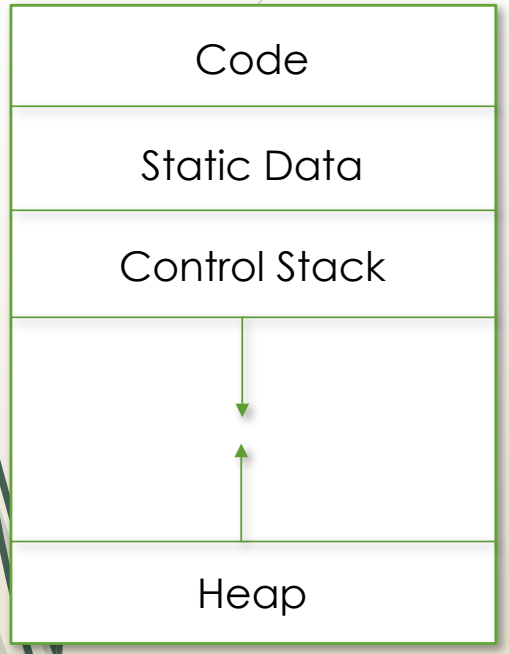
Note: C ++ can 'remember' value ligation (=) or by reference (&)

```
#include <string>
#include <iostream>
// compilar con "g++ -std=c++11 lambda.c"
int main () {
  auto f = [](int x) { return ([x] (int y) {return x+y;});} ;
  auto s4 = f(4);
  auto s3 = f(3);
  std::cout << s4(2) << ", " << s3(2) << ", " << s4(3)
            << ", " << s3(3) << std::endl;
}
```

Activation of "f (4)" recalls the ligature x ← 4 even after completion.

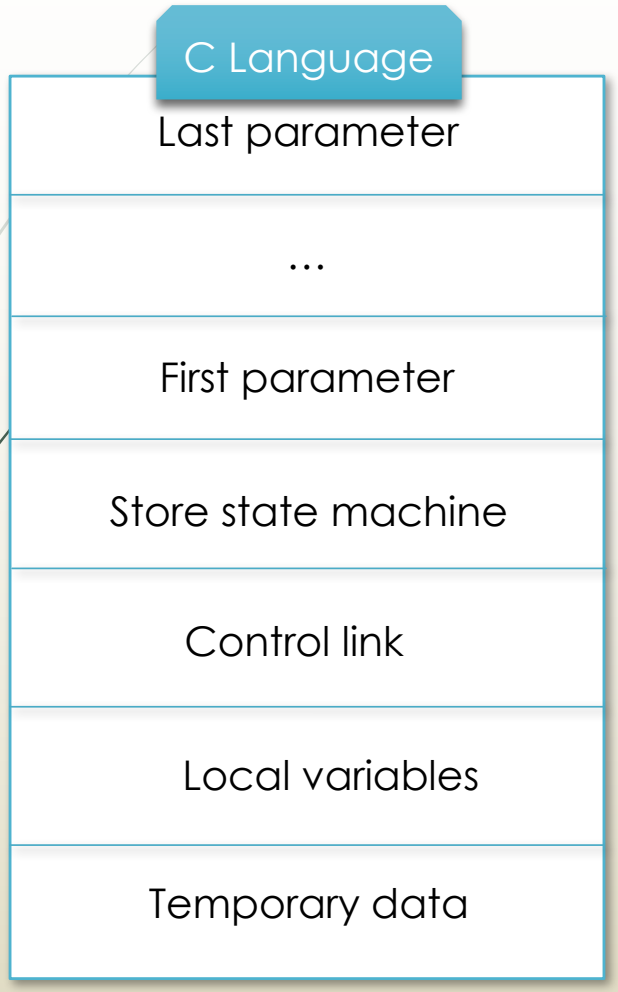
The same occurs with the activation of "f (3)" reminiscent ligation x ← 3.

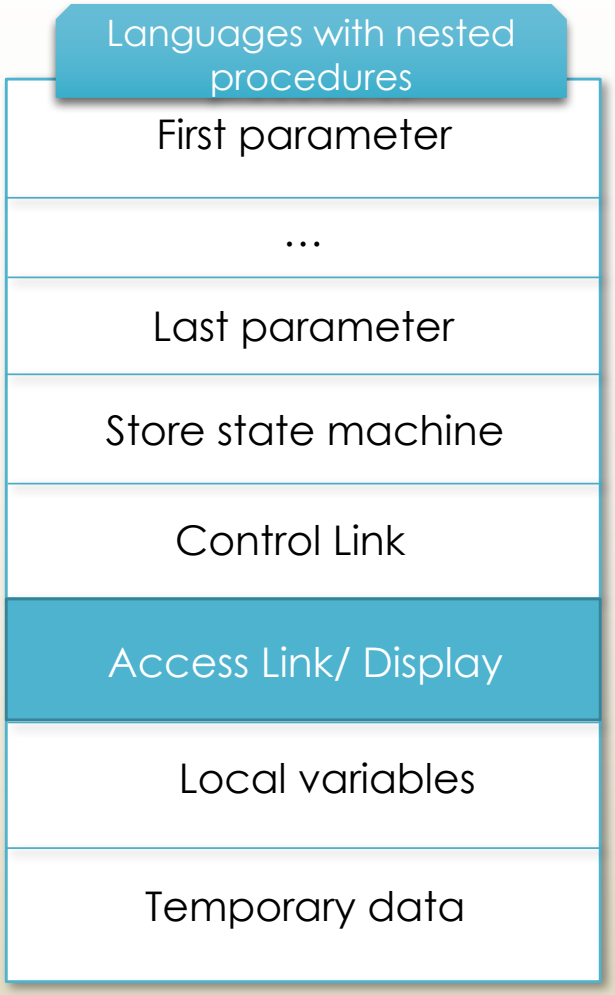
Memory organization



- When a programme is executed, a memory space is reserved by the operating system.
- **Static Data**: global variables, static local variables, constants (numbers, literal, etc.)
- **Control stack**: activation frameworks and other temporary data.
- **Heap**: dynamic data under program control (new and free)

Activation frameworks





Scope rules

- Determine the rules to follow when there is a reference to a non-local variable
- **Lexis or Static Scope**: the statement that applies to a variable name could be found by examining the program code
  - Examples: Java, C, Lisp (new versions), JavaScript, … (almost all)
- **Dynamic Scope**: the statement that applies to a variable is determined during program execution, considering ongoing activities (ancient languages).
  - Examples: Lisp (old versions), APL, SNOBOL, etc.

Scope rules

Lexis scope without nested procedures

- A procedure definition can not appear within other
- Any name non - local to a procedure, is non - local to all procedures
  - Its static address can be used by all procedures, regardless of how they are activated.
- Declared procedures can be passed freely as parameters and returned as results
  - In C language, when a function is passed a pointer to this is used.

Scope rules

Example: C Language

- A program consists of a sequence of variables and functions declarations. Nested functions are not allowed.
- A reference to a certain variable 'x' has only two possibilities:
  - 'x' is local to the procedure.
    - Access as an offset from a certain fixed position (EBP) of the activation frame of said procedure
  - 'x' is static
    - Access via absolute addressing a static memory

Scope rules

Lexical scope with nested procedures

- A procedure can be defined within a procedure.
- It is derived from Pascal languages: a reference to a variable 'x' has more likely than in the C language:
  - 'X' is local to the procedure and accessed as an offset from certain fixed position (EBP) activation frame of this procedure.
  - 'X' it is static and it is accessed by absolute address in static memory.

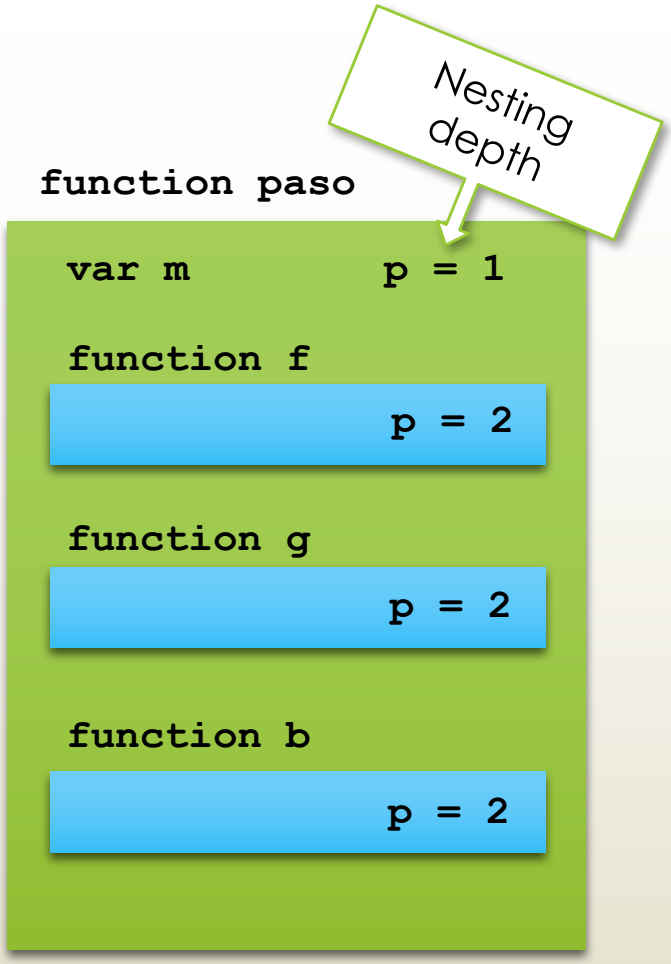
Scope rules

Lexical scope with nested procedures

- In languages derived from Pascal, the next possibility is incorporated:
  - 'x' it is not static or local to the current procedure. So:
    - 'X' it is not in the current context activation procedure.
    - 'X' is local to another procedure which includes syntactically to current.
    - 'X' it is in the last frame of activation of this procedure, which is not necessarily the one who called the present.
- Nesting depth:
  - The main program has depth 1 and we increase the depth of any procedure that is lexically nested in the previous one.

Scope rules

```
function paso() {
    var m;

    function f(n) {
        return m + n; } // end of f()

    function g(n) {
        return m * n; } // end of g()

    function b(h) {
        console.log(h(2)); } // end of b()

    m = 7;
    b(f);
    b(g);
} // end of paso()
paso();
```



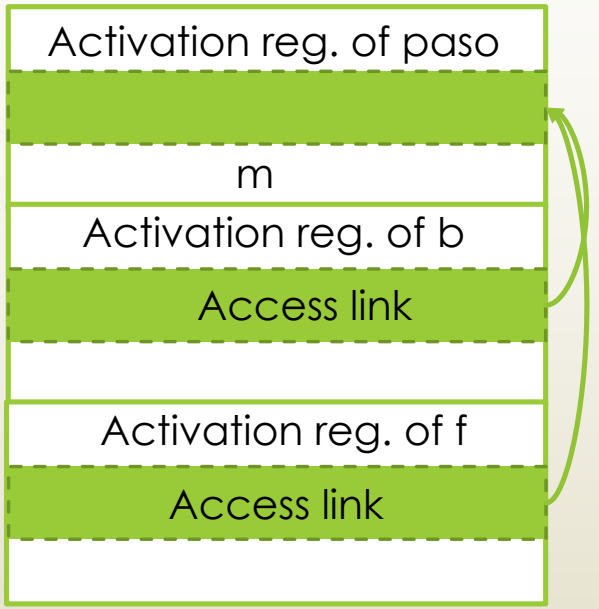
The access link

- Implementation procedures with nested lexical scope is obtained by adding a pointer, called ***Access Link*** to each activation record.
- Each access link points to another access link the latest activation of procedure that includes the current syntactically.
- If the procedure ***p*** is nested within the procedure ***c*** in the source code, then the access link in an activation record for ***p*** points to access link to the latest activation of ***c***.

The access link

```
function paso() {
    var m;

    function f(n) {
        return m + n; } // end of f()

    function g(n) {
        return m * n; } // end of g()

    function b(h) {
        console.log(h(2)); } // end of b()

    m = 7;
    b(f);
    b(g);
} // end of paso()
paso();
```

Access links to find the memory locations of non-local variables



The access link

- A variable is controlled by a pair <**p**, **d**>
  - **p** = depth: it refers to the lexical depth of the procedure where the variable definition appears.
  - **d** = displacement: it refers to the displacement of said variable within the activation frame where you have created.
- It is the task of the compiler, using the symbol table, counting and assigning go depths of nesting.

The access link

- Suppose the procedure **p**, n_p nesting depth, refers to a variable **x**(n_x, d_x). So:
  1. if $n_x = n_p$, then **x** is local to the current activation and accessed with d_x offset from the current pointer EBP.
  2. Otherwise, $n_x < n_p$. Then **x** is local to another frame activation is following $n_p - n_x$ static links from the current activation frame, and there, it is accessed to the displacement d_x.
- The direction of non-local variable **x** in the process **p** is given by the following pair calculated at compile time and stored in the symbol table:

($n_p - n_x$, displacement activation within the framework containing **x**)

Example: quicksort

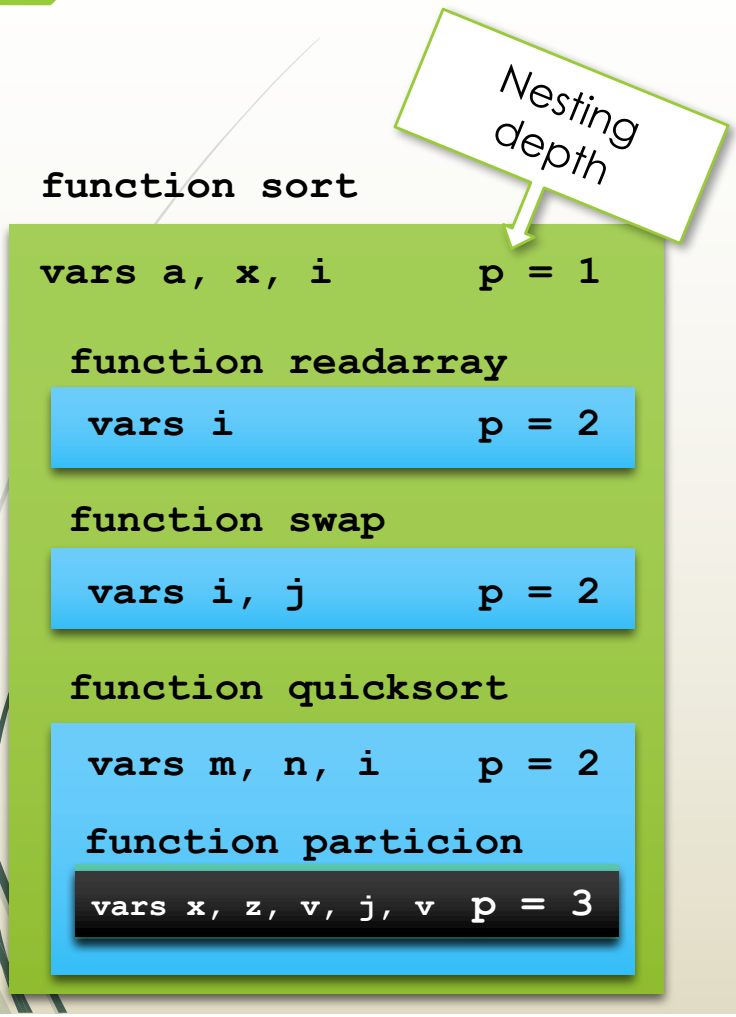
```
function sort() {
    var a = [];
    var x, i;

    function readarray() {
        var i;
        for (i = 0; i <= 10; i += 1) {
            a[i] = parseInt(prompt("Dame dato " + i + " ésimo: ", ""), 10);
        }
    } // fin de readarray()

    function swap(i, j) {
        x = a[i];
        a[i] = a[j];
        a[j] = x;
    } // fin de swap()
```

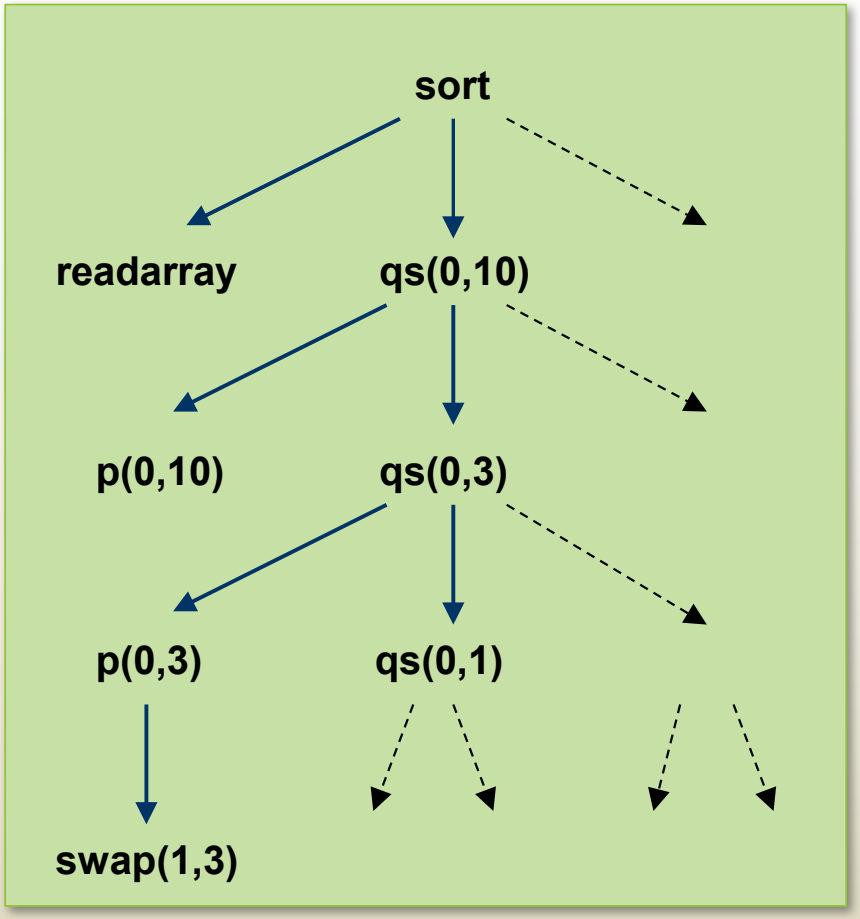
Example: quicksort

```
function quicksort(m, n) {
    var i;

    function partition(y, z) {
        var i, j, v;
        v = a[z];
        i = y - 1;
        x = 0;
        for(j = y; j < z; j++) {
            if(a[j] <= v) {
                i++;
            }
        }
        swap(i, j);
        swap(i+1, z);
        return i+1;
    } // fin de partition()
    if (m < n) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
} // fin de quicksort()
```

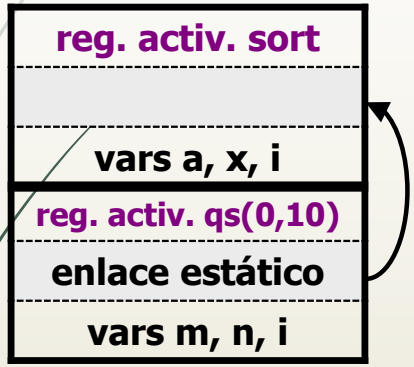
```
    readarray();
    quicksort(0, 10);
    return a;
} // fin de sort()

var m = sort();
m;
```

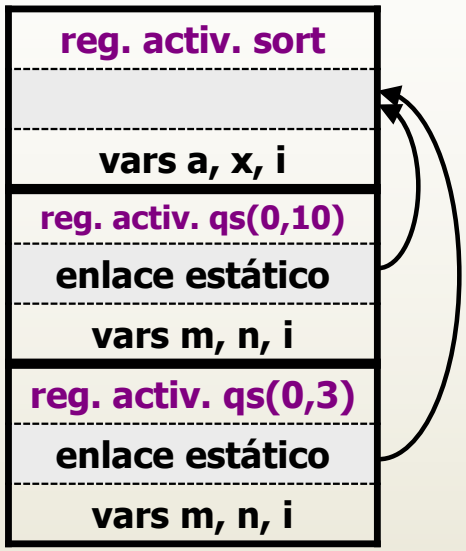
The access link

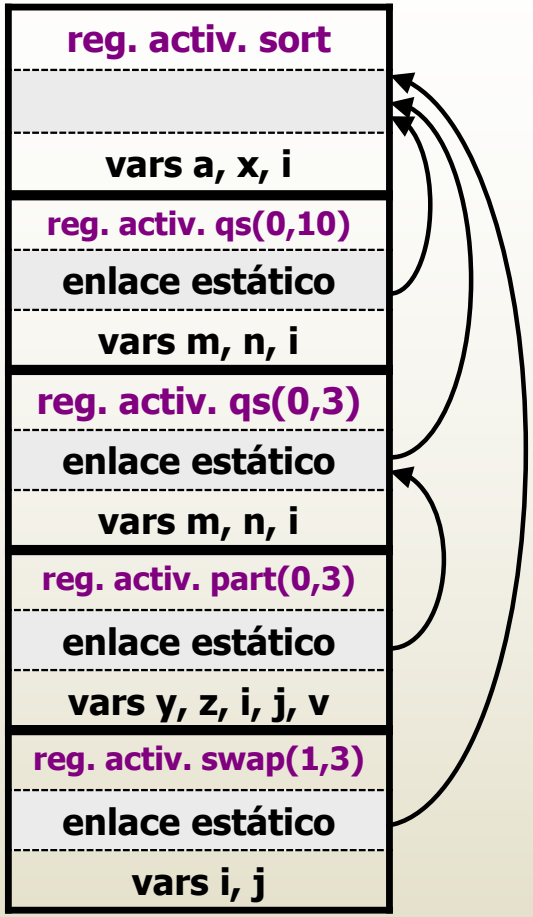




The access link







The display

- Display = array of pointers to records activation.
- It is usually stored in the activation frame of procedure, and this is generally faster than follow the links to access.
- The memory address for a non-local variable **x** to nesting depth **j** is pointed in the activation record by the element **d[j]** of display.
- A nesting depth **j**, the first **j-1** display elements point to the most recent activations of procedures that include the current procedure lexicographically.

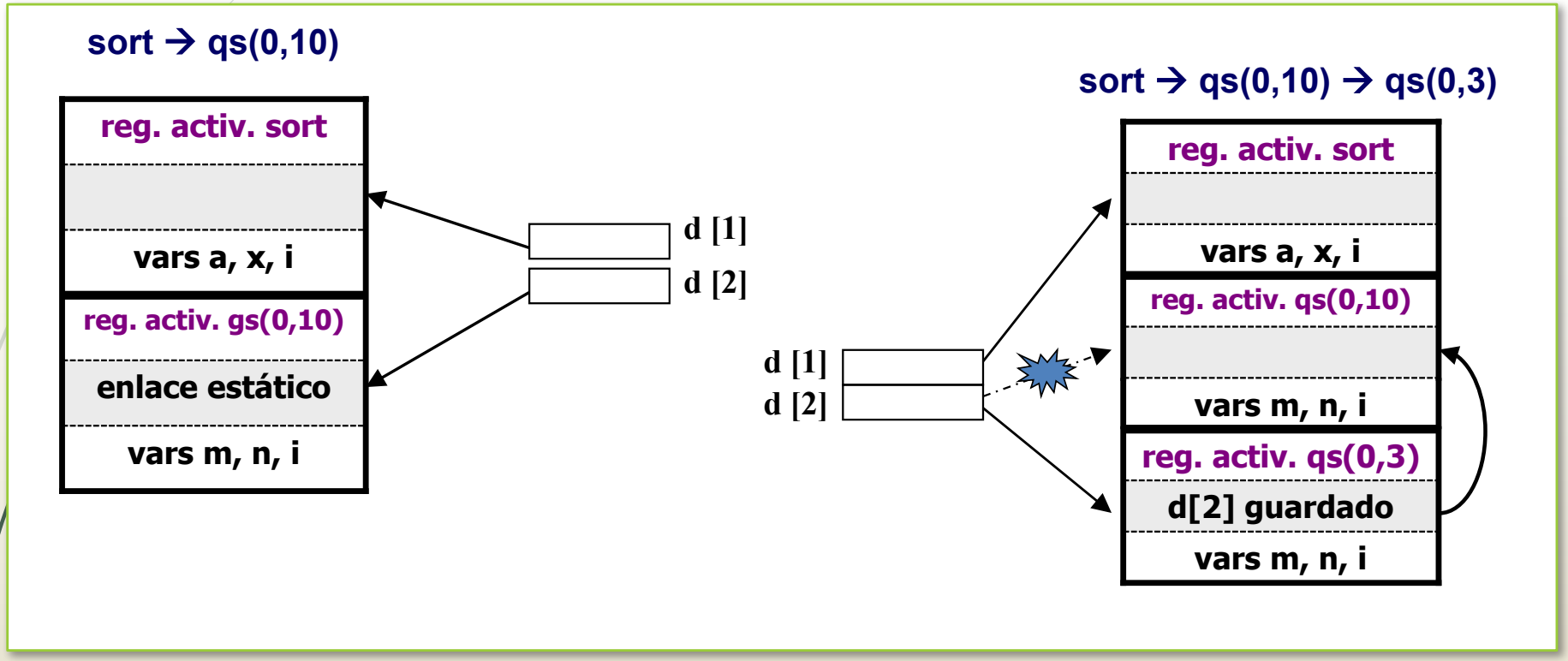
The display

- Maintenance
  - Complementing the display with access links.
  - As part of calling sequences and return, the display is updated following the chain of access links.
  - The display changes when a new activation occurs and must be reset when control returns from the new activation.

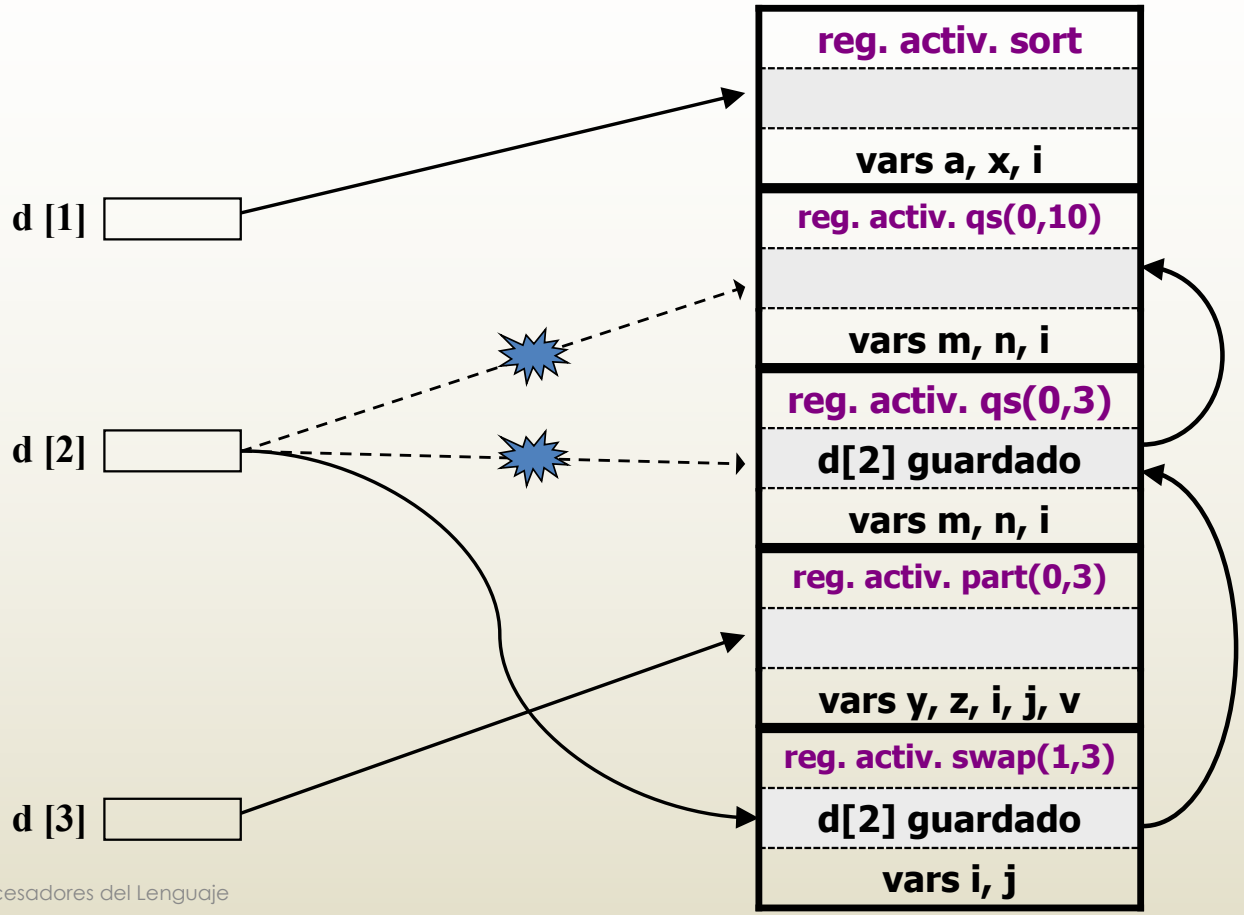
The display

- When a new activation record for a procedure nesting depth **i** is set:
  1. The value of **d[i]** is saved in the new activation record.
  2. **d[i]** is pointed to the new activation record.
- Just before the end of activation, d[i] is reset to the value stored in the activation record.

The display


sort → qs(0,10)
reg. activ. sort
vars a, x, i
reg. activ. gs(0,10)
enlace estático
vars m, n, i
d [1]
d [2]
sort → qs(0,10) → qs(0,3)
reg. activ. sort
vars a, x, i
reg. activ. qs(0,10)
vars m, n, i
reg. activ. qs(0,3)
d[2] guardado
vars m, n, i
d [1]
d [2]

The display


d [1]
d [2]
d [3]
reg. activ. sort
vars a, x, i
reg. activ. qs(0,10)
vars m, n, i
reg. activ. qs(0,3)
d[2] guardado
vars m, n, i
reg. activ. part(0,3)
vars y, z, i, j, v
reg. activ. swap(1,3)
d[2] guardado
vars i, j

Functional closure

- In case there are nested scopes and functions can be passed as parameters, then:
  - Sometimes, the compiler can not detect the nesting depth of function to be executed.
  - Or, it is accessed to local variables of activations of functions that are finished.
- The "Functional Closures", also called "Lexical Closures" are pairs formed by:
  1. A function (or reference to a function)
  2. A reference to the lexical environment in which the function was created.
     - Or a table that stores references to all the free variables of that function.
- A closure, unlike a simple pointer to a function, allows access to non-local variables, even if the function is executed outside the lexical environment in which it was created.

Functional closure

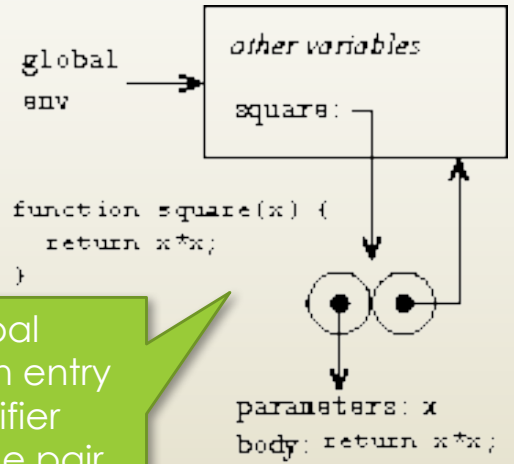
- This concept was implemented for the first time in 1975 for language Scheme (Lisp dialect).
- Functional languages allow the explicit use of closures.
- In C, and other imperative languages they are not necessary because not allows functions that return other functions in nested scopes.
- In C ++ it appears in X11 as part of the lambda expressions.
  - **auto f = [x](int y) {return x+y;} ;**

When a function is defined

- A (static) code is created and it is returned a pair formed by :
  1. A reference (or pointer) to the starting address of the code.
  2. A link to the environment where it was created and, therefore, where it should run.

```
var square = function (x) {
    return x * x;
}
```

It is assumed that the function has been defined in the global environment



In said global environment an entry for the identifier 'square' and the pair (code, environment) is created

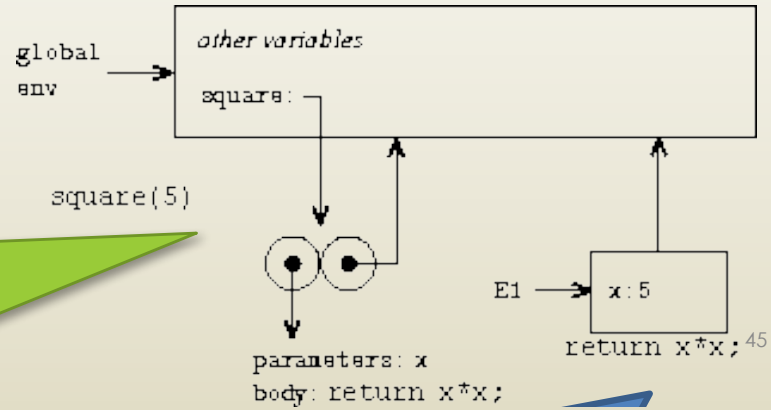
When the function is executed

- A new frame or activation record (environment) is created.
- In this framework the parameters are linked to the value of the arguments.
- The new activation frame has a static link to the environment that signals the functional closure.
- That is, the lexical environment in which the function was defined.
- In addition, a dynamic link to framework that called to a function.

square(5)

The function *square* is executed

A new environment that links the parameter x <- 5 is created. Non - local variables will be sought in the environment that created it lexically



This allows programming techniques

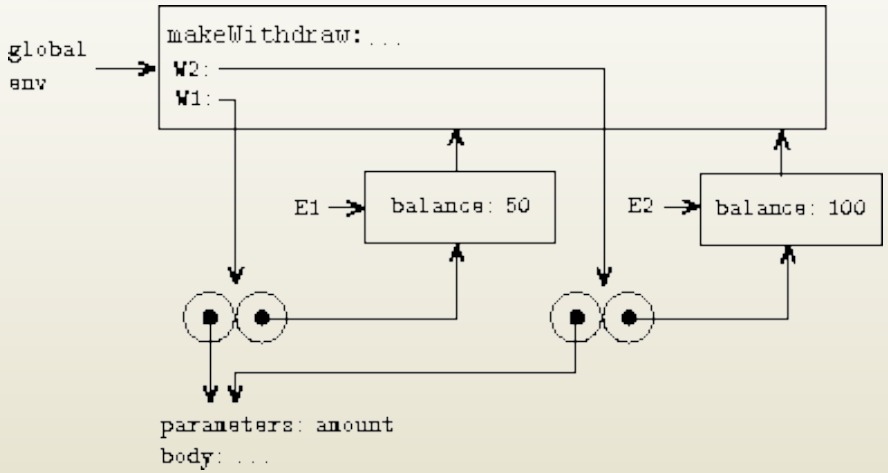
- With these techniques, a function can be internally:
  - **Local variables**:
    - Accessible only from within the function.
    - Equivalent to the properties of OOP.
  - **Local functions**:
    - They can access local variables.
    - Equivalents OOP methods.
  - So that the definition of a function equivalent to a class.
  - The activation of a function corresponds to an object of that class.
  - The activation frame contains the object properties.
  - It is a different way to implement OOP.
  - Functional Closures were implemented, for the first time, as an experiment in Scheme OOP.
  - JavaScript hides a Scheme interpreter under C syntax

Example

Definition

```
function makeWithdraw(balance) {
    return function(amount) {
        if (balance >= amount) {
            balance -= amount;
            return balance;
        }
        else
            return "Insufficient funds";
    } ;
}
```

The internal function has its own local variable 'amount' and access to 'balance' variable of lexical environment where it was created



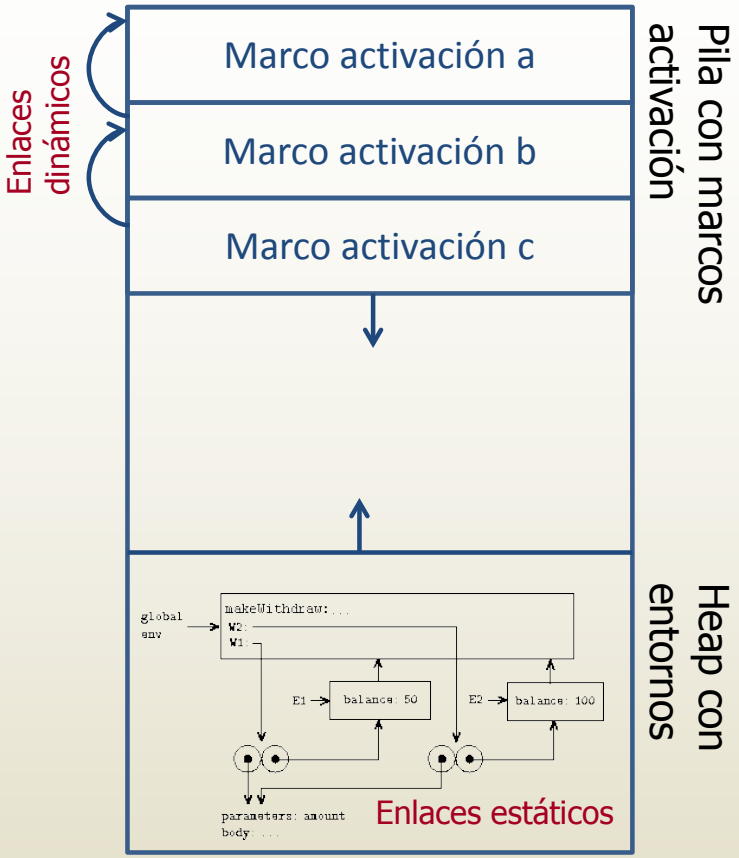
Execution

```
var W1 = makeWithdraw(100);
W1(50);
=> 50
var W2 = makeWithdraw(100);
W2(20);
=> 80
```

Two different objects are created: W1 and W2, each one with its 'balance' variable

Activation frameworks != Environment

- When you have to access local variables even after completing the activation.
- You have to separate the framework of activation.
  - It is created on the stack.
  - Containing dynamic linking.
- The environment having local variables.
  - It is created on the heap.
  - It contains the static links.



Example: decorator design pattern

```
function monitoriza(f) {
    var veces = 0;
    return function() {
        if (arguments[0] == "chivate") {
            return "veces=" + veces.toString(10); }
        else {
            veces += 1;
            return f.apply(this, arguments); }
    };
}
```

Example: decorator design pattern

```
function mediaTres(a, b, c) {
return (a+b+c)/3;
}
var makeWithdrawMon = monitoriza(makeWithdraw);
var W1 = makeWithdrawMon(100);
var W2 = makeWithdrawMon(100);
console.log("W1(50)=" + W1(50)); // W1(50)=50
console.log("W1(10)=" + W1(10)); // W1(10)=40
console.log("W2(20)=" + W2(20)); // W2(20)=80
console.log("makeWithdrawMon('chivate')=" + makeWithdrawMon('chivate'));
                                  // makeWithdrawMon('chivate')=veces=2
var m3 = monitoriza(mediaTres);
console.log("m3(1,2,3)=" + m3(1,2,3)); // m3(1,2,3)=2
console.log("m3(5,6,9)=" + m3(5,6,9)); // m3(5,6,9)=6.666666666666667
console.log("m3(4,4,5)=" + m3(4,4,5)); // m3(4,4,5)=4.333333333333333
console.log("m3('chivate')=" + m3('chivate')); // m3('chivate')=veces=3
```

Assembler code

C code

```
int a;
int p, q;

int media(int n, int m) {
    int temp;
    temp = n + m;
    temp = temp / 2;
   return temp;
} // fin de media()
```

```
int main() {
    int b, resultado;
    p = 35;
    q = 12;
    printf("p=%d, q=%d\n", p, q);
    printf("teclea dos numeros : ");
    scanf("%d%d", &a, &b);
    resultado = media(a,b);
    printf("La media de %d y %d = %d\n", a, b,
            resultado); return 0;
} // fin de main()
```

Assembler code that would generate

```
.text
.globl media
.type media, @function
media:
                          ## PROLOGO
    pushl %ebp            ## salvar enlace dinamico
    movl %esp, %ebp       ## establecer nuevo enlace dinamico
    subl $4, %esp         ## reservar espacio variables locales (int temp;)

    movl 8(%ebp), %eax    ## temp = n + m ;
    addl 12(%ebp), %eax
    movl %eax, -4(%ebp)

    movl -4(%ebp), %eax   ## temp = temp / 2 ;
    cdq                   ## extiende bit signo %eax sobre %edx
    movl $2, %ebx
    divl %ebx
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax   ## return temp;
                          ## EPILOGO
    movl %ebp, %esp       ## borra variables locales y temporales
    popl %ebp             ## restaura marco activacion anterior
    ret                   ## devuelve control
```

```
int media(int n, int m) {
    int temp;
    temp = n + m;
    temp = temp / 2;
    return temp;
} // fin de media()
```

Assembler code that would generate

```
.text
.globl main
.type main, @function
main:
                          ## PROLOGO
    pushl %ebp            ## salvar enlace dinamico
    movl %esp, %ebp       ## establecer nuevo enlace dinamico
    subl $8, %esp         ## reservar espacio variables locales (int b, resultado;)

    movl $35, p           ## p = 35
    movl $12, q           ## q = 12

    pushl q               ## printf("p=%d, q ...", p, q) ;
    pushl p
    pushl $s0
    call printf
    addl $12, %esp        ## quita parametros

    pushl $s1             ## printf("teclea ...") ;
    call printf
    addl $4, %esp         ## quita parametros
```

```
int main() {
    int b, resultado;
    p = 35;
    q = 12;
    printf("p=%d, q=%d\n", p, q);
    printf("teclea dos numeros : ");
.....
```

Assembler code that would generate

```
leal -4(%ebp), %eax
pushl %eax
pushl $a
pushl $s2
call scanf
addl $12, %esp

pushl -4(%ebp)       ## resultado = media(a,b) ;
pushl a
call media
addl $8, %esp        ## quita parametros
movl %eax, -8(%ebp)  ## guarda el resultado

pushl -8(%ebp)       ## printf("la media de ...", a, b, resultado) ;
pushl -4(%ebp)
pushl a
pushl $s3
call printf

addl $16, %esp       ## quita parametros
movl $0, %eax        ## return 0 ;
                     ## EPILOGO
movl %ebp, %esp      ## borra variables locales y temporales
popl %ebp            ## restaura marco activacion anterior
ret                  ## devuelve control
```

```
int main() {
    ..............
    scanf("%d%d", &a, &b);
    resultado = media(a,b);
    printf("La media de %d y %d = %d\n",
            a, b, resultado);
    return 0;
} // fin de main()
```

Complex expressions

- When complex expressions are evaluated, it is necessary to save intermediate values on the stack.
- We will keep as induction hypothesis, the result of an operation is in %eax
- Example: **2 * 3 + 7 / 2 – 4 * 5 * 8**

```
2 * 3 -> %eax
pushl %eax =>save intermediate value (2 * 3) on the stack
7 / 2 -> %eax
pushl %eax => save intermediate value (7 / 2) on the stack
popl %ebx  => retrieve intermediate result (7/2)
popl %eax => retrieve intermediate result(2*3)
addl %ebx, %eax
pushl %eax => save intermediate value (2 * 3) + (7 / 2) on the stack
…
```

2 * 3 + 7 / 2 – 4 * 5 * 8

```
movl $2, %eax      # 2 * 3
imull $3, %eax
pushl %eax         # save the intermediate result

movl $7, %eax      # 7/ 2
movl $2, %ebx
cdq
idivl %ebx
pushl %eax         # save the intermediate result

popl %ebx          # 7 / 2 -> %ebx
popl %eax          # 2 * 3 -> %eax
addl %ebx, %eax    # (2*3) + (7/2)
pushl %eax         # save the intermediate result

movl $4, %eax      # 4 * 5 * 8
imull $5, %eax
imull $8, %eax
pushl %eax         # save the intermediate result
popl %ebx          # 4 * 5 * 8 -> %ebx
popl %eax          # (2 * 3) + (7 / 2) -> %eax
subl %ebx, %eax    # ((2*3) + (7/2)) - (4*5*8)
```

Translation scheme

```
Sum -> Sum {pushl %eax;} + Fact {movl %eax, %ebx; popl %eax; addl %ebx, %eax;}
Sum -> Sum {pushl %eax;} - Fact {movl %eax, %ebx; popl %eax; subl %ebx, %eax;}
Sum -> Fact
Fact -> Fact {pushl %eax;} * Basico {movl %eax, %ebx; popl %eax; imull %ebx, %eax;}
Fact -> Fact {pushl %eax;} / Basico {movl %eax, %ebx; popl %eax; cdq; idivl %ebx;}
Fact -> Basico
Basico -> NUM {movl $(NUM.lexval), %eax;}
Basico -> ID {movl ID.lexval, %eax;} // static variable case
Basico -> ID {movl +tablaOffset[ID.lexval](%ebp), %eax;} // parameter case
Basico -> ID {movl -tablaOffset[ID.lexval](%ebp), %eax;} // dynamic variable case
```

It is important to wear a symbol table with the offset and if the variable is static, parameter, or local dynamics. As the offset that corresponds.

Exercises

Exercise 1: C code

```
int h2;

int pitagoras(int c1, int c2) {
    int temp, temp1, temp2;
    temp1 = c1 * c1;
    temp2 = c2 * c2;
    temp = temp1 + temp2;
   return temp;
} // fin de pitagoras()
```

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

    printf("teclea dos numeros : ");
    scanf("%d%d", &a, &b);
    h2 = pitagoras(a,b);
    printf("La hipotenusa es %d\n", h2);

    return 0;
} // fin de main()
```

Assembler code that would generate

```
.text
.globl pitagoras
.type pitagoras, @function
pitagoras:
                          ## PROLOGO
    pushl %ebp             ## salvar enlace dinamico
    movl %esp, %ebp        ## establecer nuevo enlace dinamico
    subl $12, %esp         ## reservar espacio variables locales (int temp, temp1, temp2;)

    movl 8(%ebp), %eax    ## temp1 = c1 * c1 ;
    imull 8(%ebp), %eax
    movl %eax, -8(%ebp)

    movl 12(%ebp), %eax    ## temp2 = c2 * c2 ;
    imull 12(%ebp), %eax
    movl %eax, -12(%ebp)

    movl -8(%ebp), %eax    ## temp = temp1 + temp2 ;
    addl -12(%ebp), %eax
    movl %eax, -4(%ebp)

    movl -4(%ebp), %eax   ## return temp;
                          ## EPILOGO
    movl %ebp, %esp        ## borra variables locales y temporales
    popl %ebp              ## restaura marco activación anterior
    ret                    ## devuelve control
```

```
int pitagoras(int c1, int c2) {
    int temp, temp1, temp2;
    temp1 = c1 * c1;
    temp2 = c2 * c2;
    temp = temp1 + temp2;
   return temp;
} // fin de pitagoras()
```

Assembler code that would generate

```
.text
.globl main
.type main, @function
main:
                        ## PROLOGO
    pushl %ebp            ## salvar enlace dinámico
    movl %esp, %ebp       ## establecer nuevo enlace dinámico
    subl $8, %esp         ## reservar espacio variables locales (int a, b;)

    pushl $s0
    call printf
    addl $4, %esp         ## quita parámetros

    leal -8(%ebp), %eax   ## scanf("%d%d", &a, &b);
    pushl %eax
    leal -4(%ebp), %eax
    pushl %eax
    pushl $s1
    call scanf
    addl $12, %esp        ## quita parametros
```

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

    printf("teclea dos numeros : ");
    scanf("%d%d", &a, &b);
.....
```

Assembler code that would generate

```
int main() {
    ..............
    h2 = pitagoras(a,b);
    printf("La hipotenusa es %d\n", h2);

    return 0;
} // fin de main()
```

```
pushl -8(%ebp)        ## h2 = pitagoras(a,b) ;
pushl -4(%ebp)
call pitagoras
addl $8, %esp         ## quita parámetros
movl %eax, h2         ## guarda el resultado

pushl h2              ## printf("la hipotenusa es %d\n, h2) ;
pushl $s3
call printf

addl $8, %esp         ## quita parámetros
movl $0, %eax         ## return 0 ;

                      ## EPILOGO
movl %ebp, %esp       ## borra variables locales y temporales
popl %ebp             ## restaura marco activación anterior
ret                   ## devuelve control
```