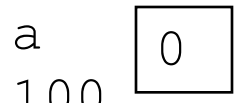
Memory Locations For Variables

A Binding Question

- Variables are bound (dynamically) to values
- Those values must be stored somewhere
- Therefore, variables must somehow be bound to memory locations
- How?

Functional Meets Imperative

- Imperative languages expose the concept of memory locations: **`a := 0`**
  - Store a zero in **`a`**'s memory location



- Functional languages hide it: **`let a = 0`**
  - Bind **`a`** to the value zero
- But both need to connect variables to values represented in memory
- So both face the same binding question

Outline

- Activation records
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records

Function Activations

- The lifetime of one execution of a function, from call to corresponding return, is called an *activation* of the function
- When each activation has its own binding of a variable to a memory locations, it is an *activation-specific* variable
- (Also called *dynamic* or *automatic*)

Activation-Specific Variables

- In most modern languages, activation-specific variables are the most common kind:

```
let rec fact n =
  if n = 0 then 1

  else n * fact (n-1)
```

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

Block Activations

- For block constructs that contain code, we can speak of an activation of the *block*
- The lifetime of one execution of the block
- A variable might be specific to an activation of a particular block within a function:

```
let rec fact n =
  if n=0 then 1
  else
    let b = fact (n-1)
    in
      n*b
```

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

Other Lifetimes For Variables

- Most imperative languages have a way to declare a variable that is bound to a single memory location for the entire runtime
- Obvious binding solution: static allocation (classically, the loader allocates these)

```
int count = 0; // global scope
int nextcount() {
  return ++count;
}
```

Scope And Lifetime Differ

- In most modern languages, variables with local *scope* have activation-specific *lifetimes*, at least by default
- However, these two aspects can be separated, as in C:

```
int nextcount() {
  static int count = 0;  // local scope
  count = count + 1;
  return count;
}
```

Other Lifetimes For Variables

- Object-oriented languages use variables whose lifetimes are associated with object lifetimes
- Some languages have variables whose values are persistent: they last across multiple executions of the program
- Will focus on activation-specific variables

Activation Records

- Language implementations usually allocate all the activation-specific variables of a function together as an *activation record*
- The activation record also contains other activation-specific data, such as
  - Return address: where to go in the program when this activation returns
  - Link to caller's activation record: more about this in a moment

Block Activation Records

- When a block is entered, space must be found for the local variables of that block
- Various possibilities:
  - Preallocate (static) in the containing function's activation record
  - Extend the function's activation record when the block is entered (and revert when exited)
  - Allocate separate block activation records
- Our illustrations will show the static option

Outline

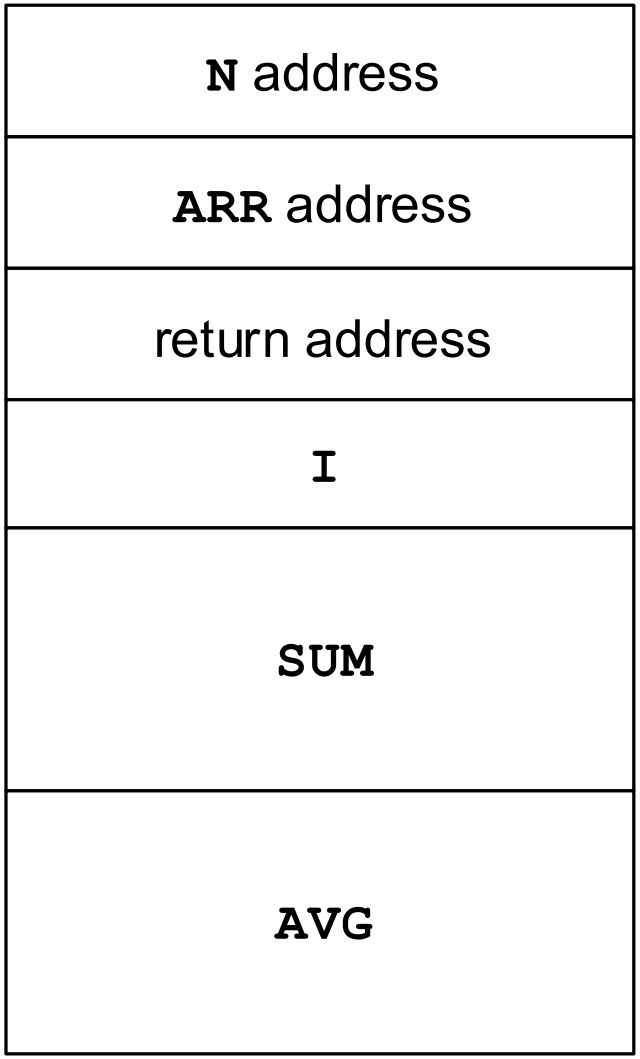
- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records

Static Allocation

- The simplest approach: allocate one activation record for every function, statically
- Older dialects of Fortran and Cobol used this system
- Simple and fast

Fortran Example

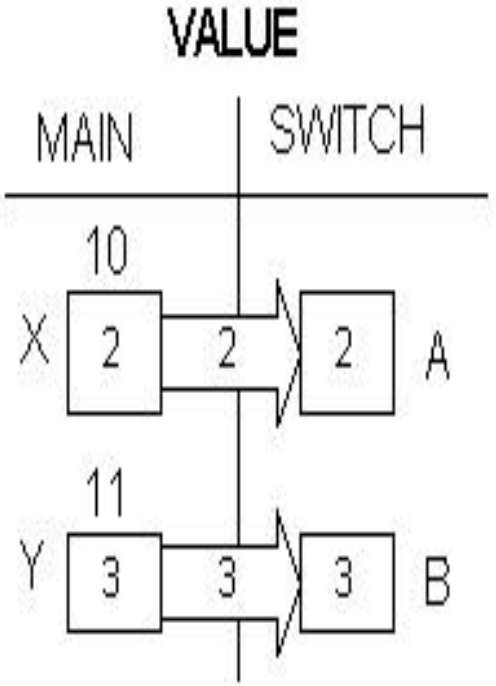
```
      FUNCTION AVG (ARR, N)
      DIMENSION ARR(N)
      SUM = 0.0
      DO 100 I = 1, N
        SUM = SUM + ARR(I)
100   CONTINUE
      AVG = SUM / FLOAT(N)
      RETURN
      END
```

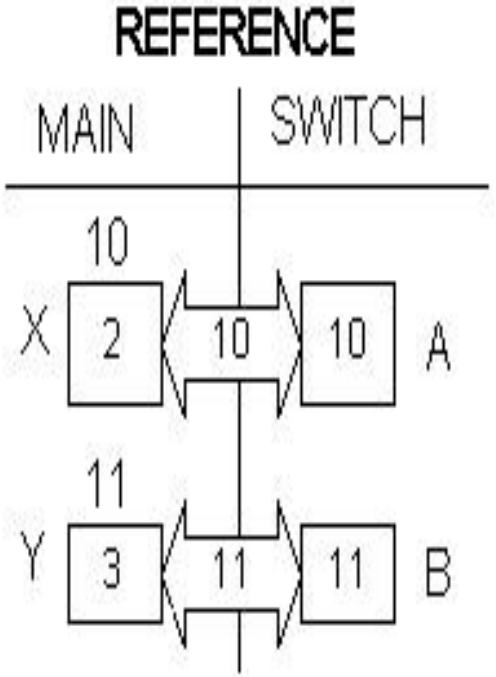


How are parameters passed?

Value and Reference Parameter Passing Review

```
x = 2;
y = 3;
switch(x, y);
     :
void switch(
     float &a,
     float &b)
{
          float t;
          t = a;
          a = b;
          b = t;
 }
```

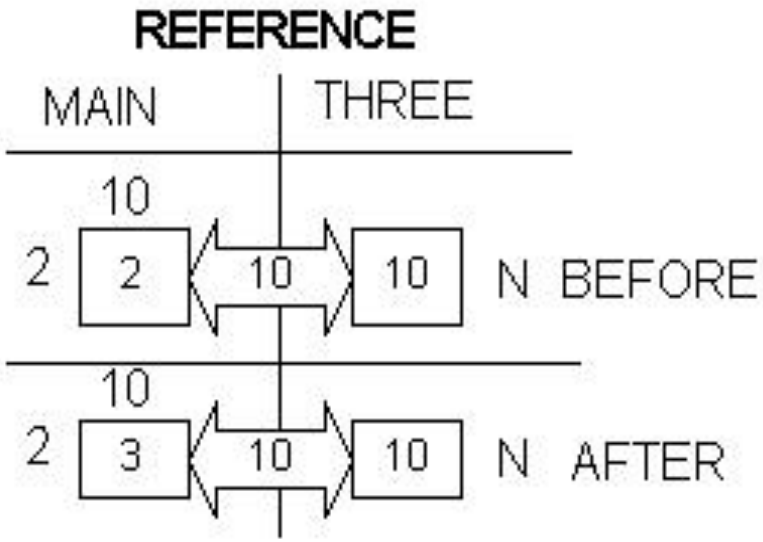




Reference passing potentially dangerous

Passing literals by reference must be prevented

```
x = 2 + 2;
three(2);
x = 2 + 2;
   :
void three(int &n) {
   n=3;
}
```



Question: How can this problem be prevented?

Exercise 1

What is the output of the following C++ program?

```
 void SUB(int &K, float &X) {
   K = 1;
   X = 20;
}

void main(void) {
    float A[2];
    int I;
    I = 0;
   A[0] = 10;
   A[1] = 11;
   SUB(I, A[I]);
   cout << A[0] << " " << A[1] << "\n";
}
```

Static Allocation

1. Simple, only one activation record per procedure.
2. Memory allocation done at load time.
3. Does not allow recursion. Why?
4. Does not allow for nested scope. Why?
5. Faster than dynamic allocation.
6. We'll look at an example patterned after FORTRAN language.

Definition Key to Activation Records

1. **M –** Memory, a linear array M[0],...,M[n].
2. **PAR –** Address of caller's parameters, PAR[1],...PAR[n].
3. **IP –** Caller's return address.
4. **TMP –** Temporary storage of Caller's registers.
5. **DL** – Dynamic Link to Caller's Activation Record.
6. **AR(S)** – The Activation Record address of function S.

Static Activation

Caller Code for S

$F(p_1,...,p_n)$ =>

```
  M[AR(S)].TMP = REGs
  M[AR(S)].IP = resume
  M[AR(F)].PAR[1]= &p1
        :
  M[AR(F)].PAR[1]= &pn
  M[AR(F)].DL=AR(S)
  goto entry(F)
resume:
  REGs=M[AR(S)].TMP
```

Callee Code for F return

```
  goto M[M[AR(F)].DL].IP
```

AR(S)	**M**emory	**Comment**
	S resume Line 4	IP
	AR(caller)	DL
	S Regs	TMP
AR(F)		
x	$\&p_1$	PAR[1]= $\&p_1$
y	$\&p_2$	PAR[2]= $\&p_2$
z	$\&p_3$	PAR[3]= $\&p_3$
		IP
	AR(S)	DL
		TMP

```
1.  void S(void) {
2.        int p1, p2, p3;
3.        F(p1, p2, p3);
4.  }
5.  void F(int &x, int &y, int &z) {
6.        z = x + y;
7.  }
```

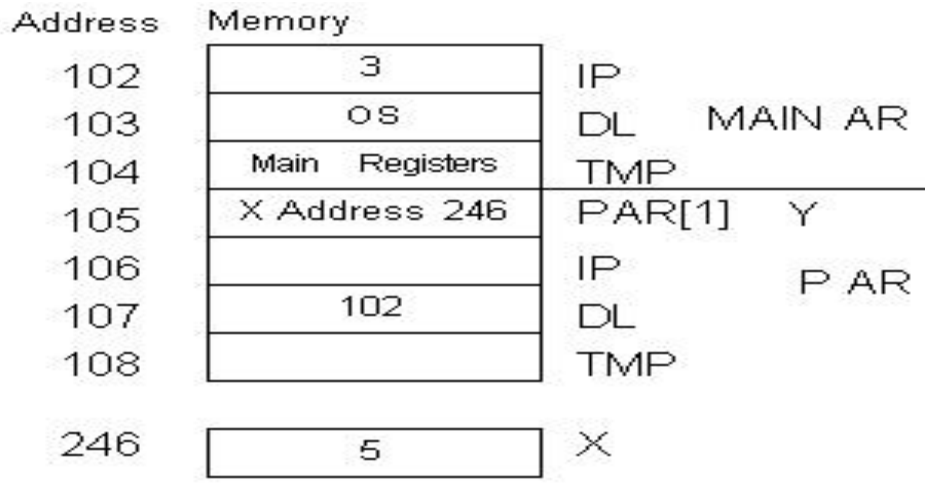
```
1.  X = 5;
    VOID MAIN {
2.  CALL P(X)
3.  PRINT, X
    }
```

a) M[AR(MAIN)].IP = 3 // Resume address
b) M[AR(P)].Par[1] = 246 // X reference
c) M[AR(P)].DL = 102 // AR(MAIN) Addr
d) M[AR(MAIN)].Tmp = Registers
e) Goto Algorithm address 4 // Entry of P
f) Registers = M[AR(MAIN)].Tmp

```
4.  VOID P(Y){
5.      Y = 12
6.      RETURN
    }
```

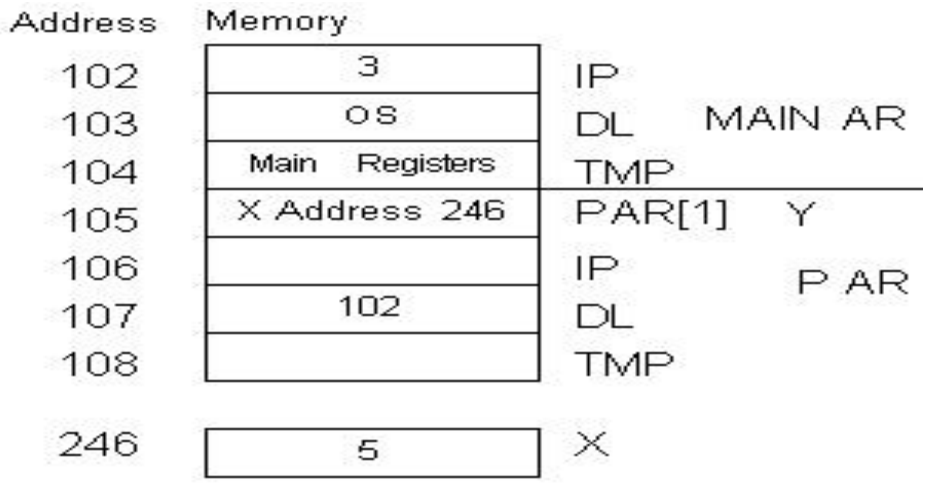
a) M[M[AR(P)].PAR[1]] = 12 // Y = 12
b) goto M[M[AR(P)].DL].IP // Return address
where M[AR(P)].DL is 102
then M[102].IP is 3, the return address

CALL and RETURN Code



Parameter Access in Subprogram

Parameters are passed by reference through the PAR[ ] parameter list.



For **Y = 12** the compiler generates **M[M[AR(P)].PAR[1]] = 12**

AR(P) is 105
for M[AR(P)].PAR[1] then M[105].PAR[1] is 246
for M[M[AR(P).PAR[1]]] then M[246] is 5 which is altered to 12.

```
Addr
100     void main(void)
          { float x=1400;
101         float y=1600;
102         float a;
10          Ave(x, y, a);
11        }

          void Ave(
278         float &m,
279         float &n,
280         float &u)
104     { float z;
12        Sum(m, n, z);
13        u = z / 2.0;
14      }

          void Sum(
284         float &r,
285         float &s,
286         float &t)
105     { float w;
15        w = r + s;
16        t = w;
17      }
```

Ave

u ___
m ___
n ___
z ___

Sum

r ___
s ___
t ___
w ___

main

a ___
y ___
x ___

Caller Code for S

$F(p_1,...,p_n)$ =>
  M[AR(S)].TMP = REG
  M[AR(S)].IP = resume
  M[AR(F)].PAR[1]= &p_1
  :
  M[AR(F)].PAR[1]= &p_n
  M[AR(F)].DL=AR(S)
  goto entry(F)
resume:
  REG=M[AR(S)].TMP

Callee Code for F

  goto M[M[AR(F)].DL].IP

main	Memory	Comment
275	11	IP
276	OS	DL
277	Regs	TMP
Ave		
278	100	PAR[1]=&x
279	101	PAR[2]=&y
280	102	PAR[3]=&a
281		IP
282	275	DL
283	Regs	TMP
Sum		
284		PAR[1]=
285		PAR[2]=
286		PAR[3]=
287	NA	IP
288		DL
289	NA	TMP

Drawbacks to Static AR Allocation

- Each function has one activation record
- There can be only one activation alive at a time
- Modern languages (including modern dialects of Cobol and Fortran) do not obey this restriction:
  - Recursion
  - Multithreading
  - Nested scopes

Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records

Stacks Of Activation Records

- To support recursion, we need to allocate a new activation record for *each* activation
- Dynamic allocation: activation record allocated when function is called
- For many languages, like C, it can be deallocated when the function returns
- A stack of activation records: *stack frames* pushed on call, popped on return

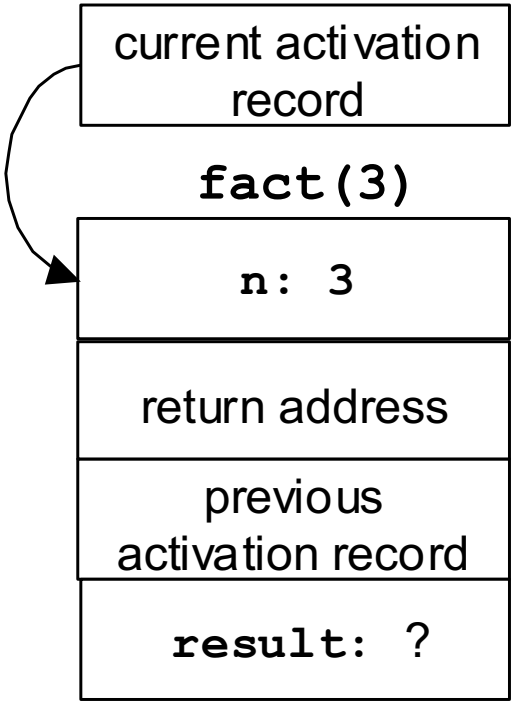
Current Activation Record

- STATIC: location of activation record was determined before runtime
- DYNAMIC: location of the *current* activation record is not known until runtime
- A function must know how to find the address of its current activation record
- Often, a special machine register (eBp on Intel) holds *current activation record* address

C Example

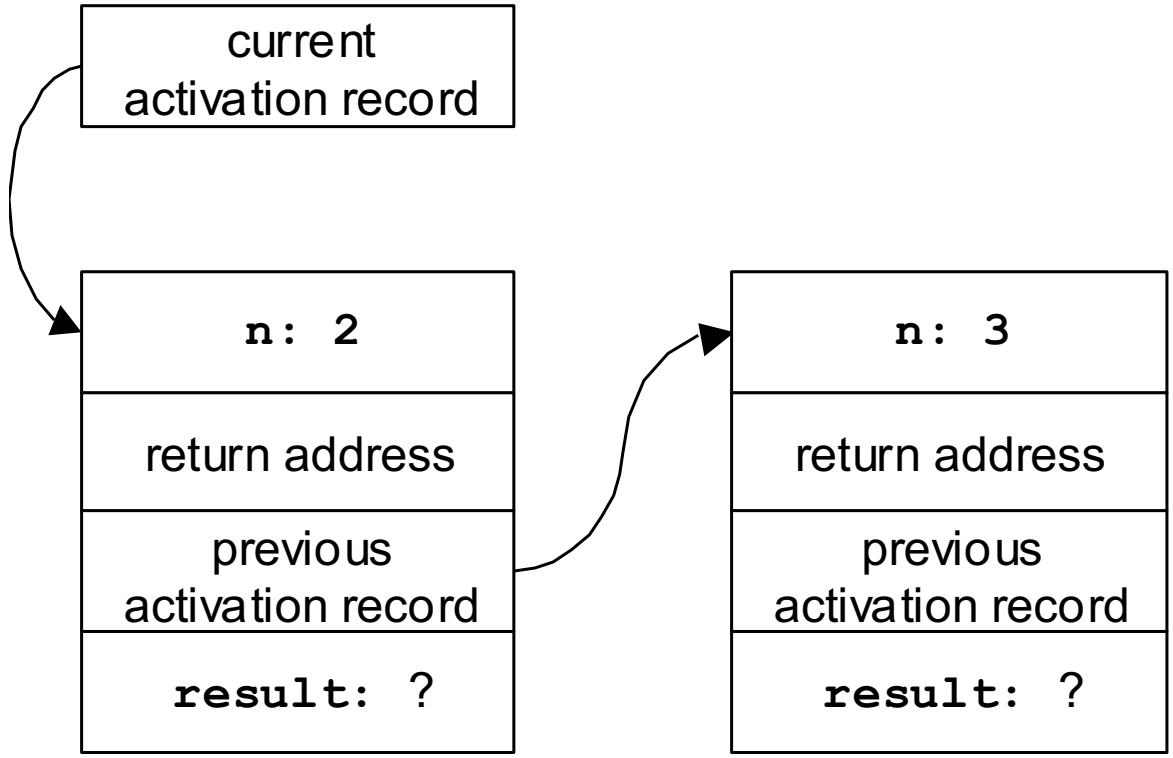
We are evaluating **`fact(3)`**. *This shows the contents of memory just before the recursive call that creates a second activation.*

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



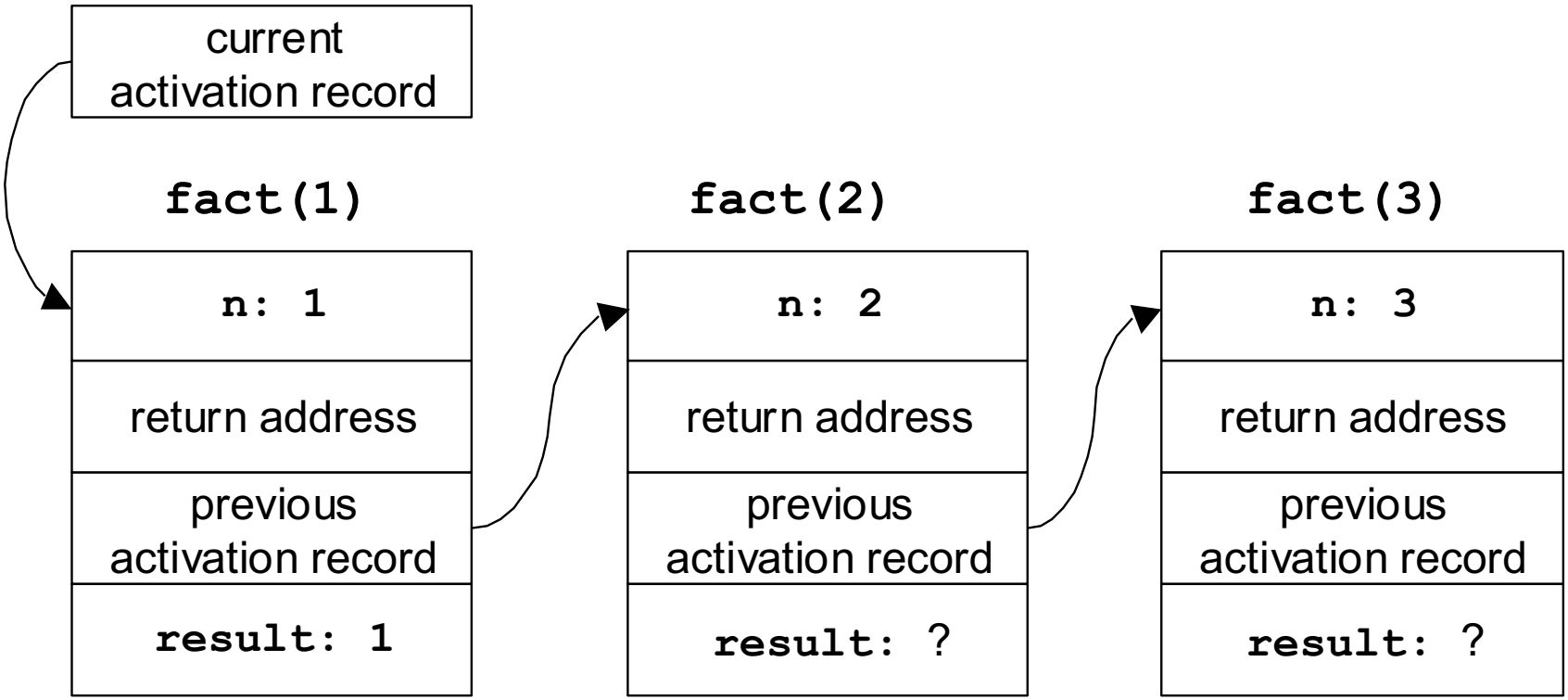
This shows the contents of memory just before the third activation.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



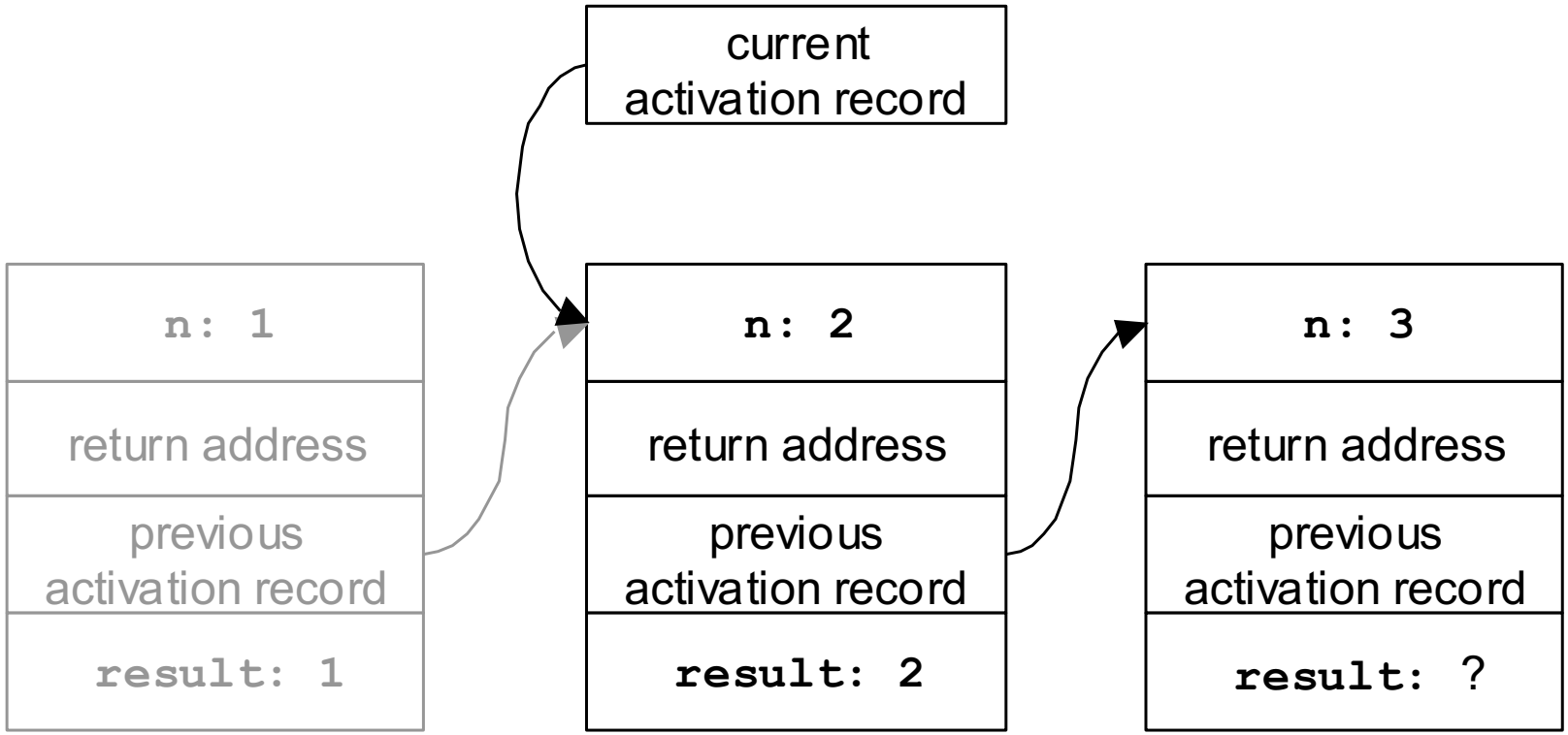
This shows the contents of memory just before the third activation returns.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



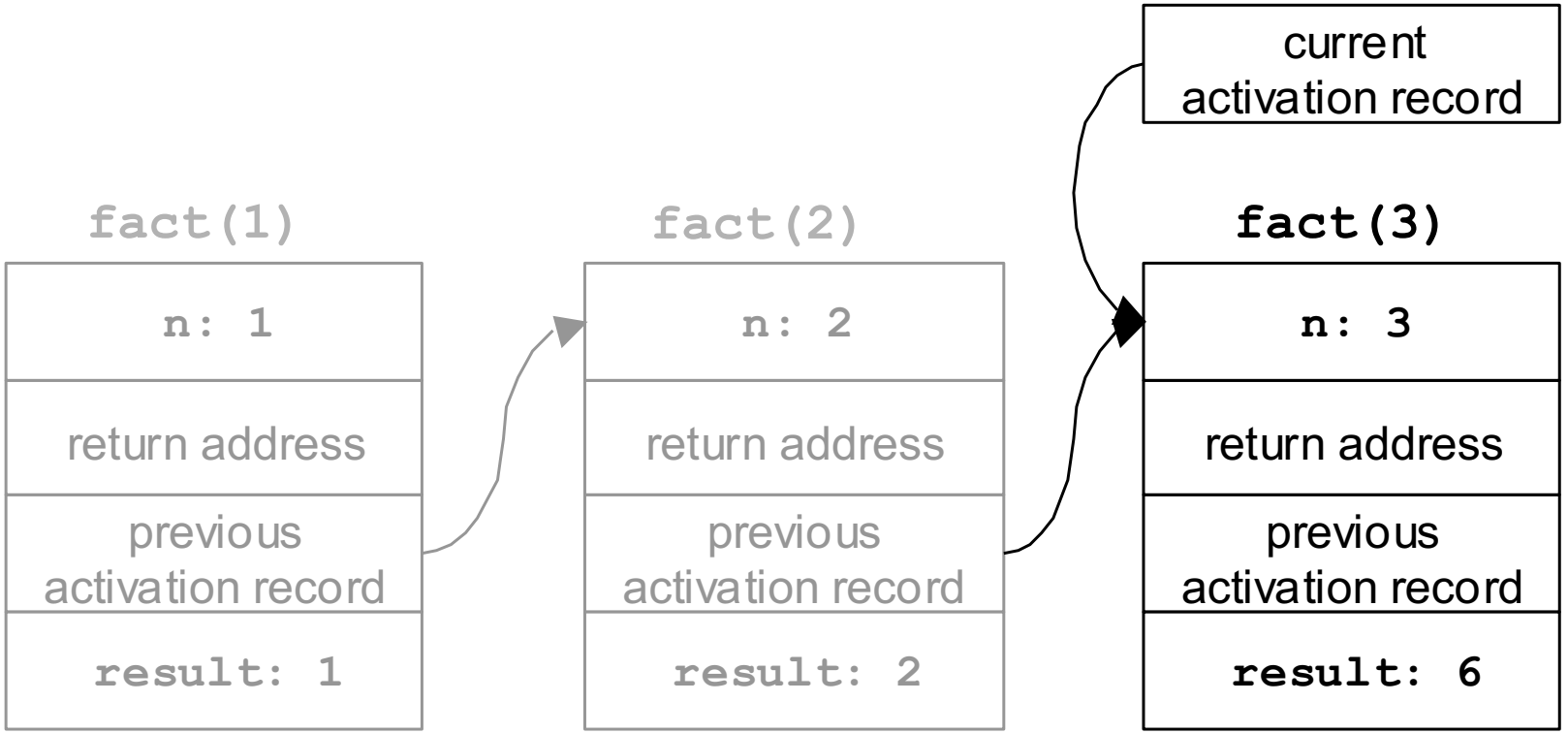
The second activation is about to return.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



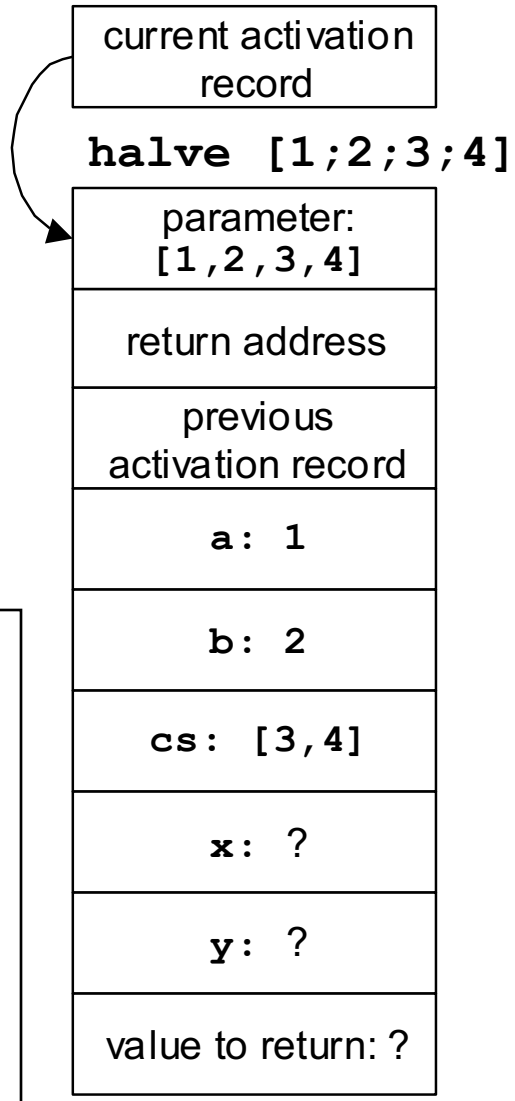
- *The first activation is about to return with the result* **`fact(3) = 6`**.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



F# Example

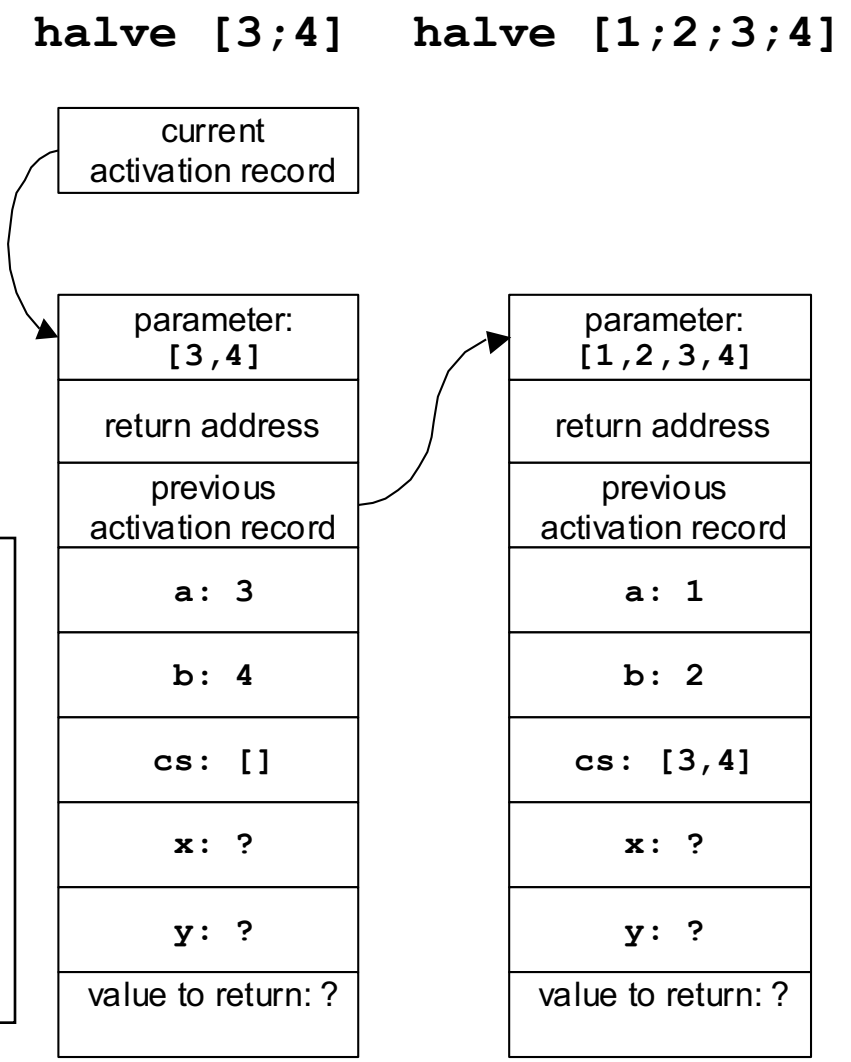
We are evaluating **`halve [1;2;3;4]`**. *This shows the contents of memory just before the recursive call that creates a second activation.*



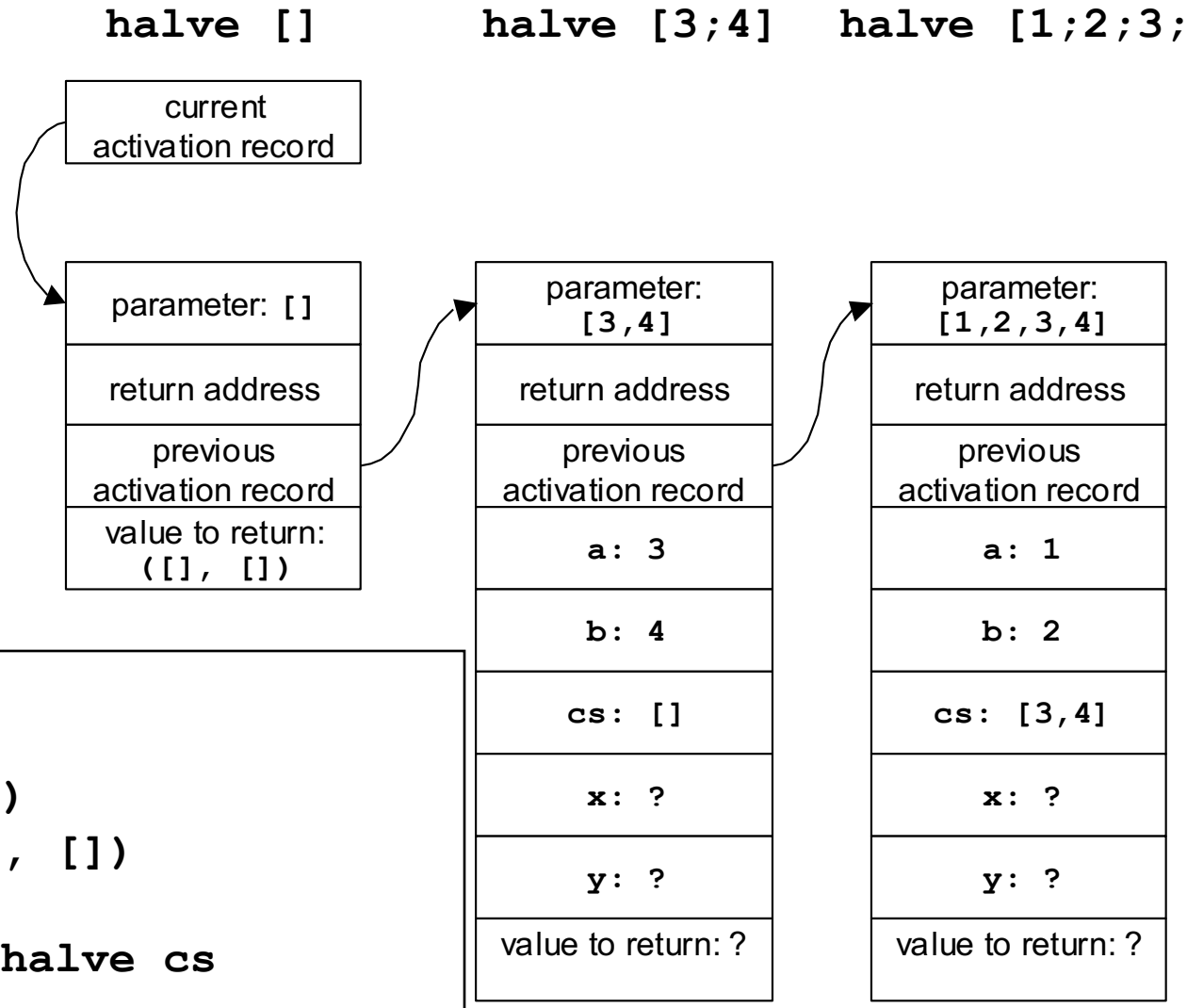

```
let rec halve L =
  match L with
    | [] -> ([], [])
    | a::[] -> ([a], [])
    | a::b::cs ->
      let (x, y) = halve cs
      in
      (a::x, b::y);;
```

This shows the contents of memory just before the third activation.

```
let rec halve L =
  match L with
    | [] -> ([], [])
    | a::[] -> ([a], [])
    | a::b::cs ->
      let (x, y) = halve cs
      in
      (a::x, b::y);;
```

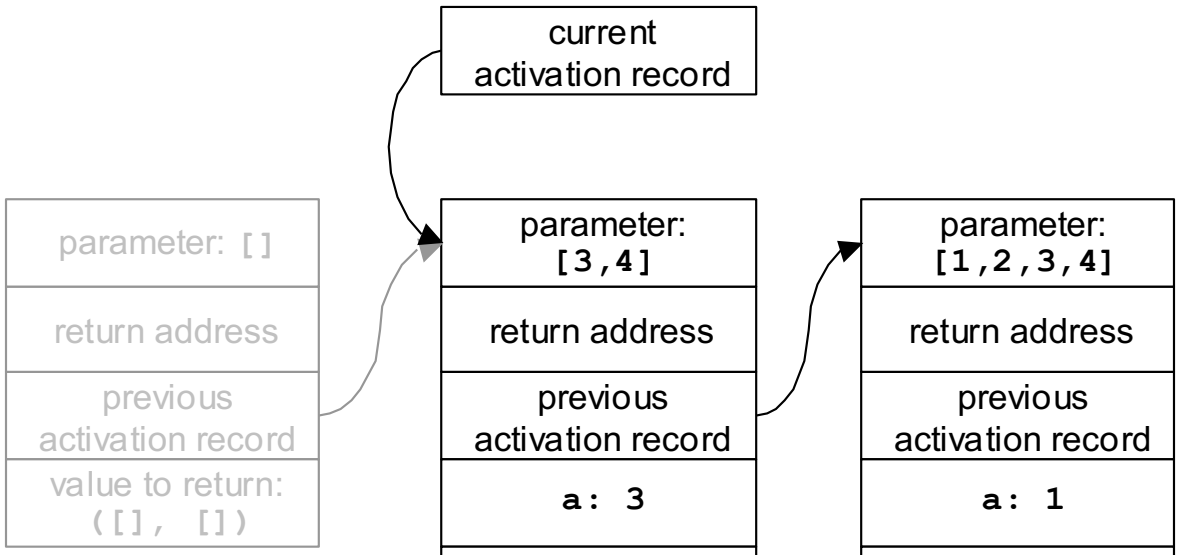


This shows the contents of memory just before the third activation returns.




```
let rec halve L =
  match L with
    | [] -> ([], [])
    | a::[] -> ([a], [])
    | a::b::cs ->
      let (x, y) = halve cs
      in
      (a::x, b::y);;
```

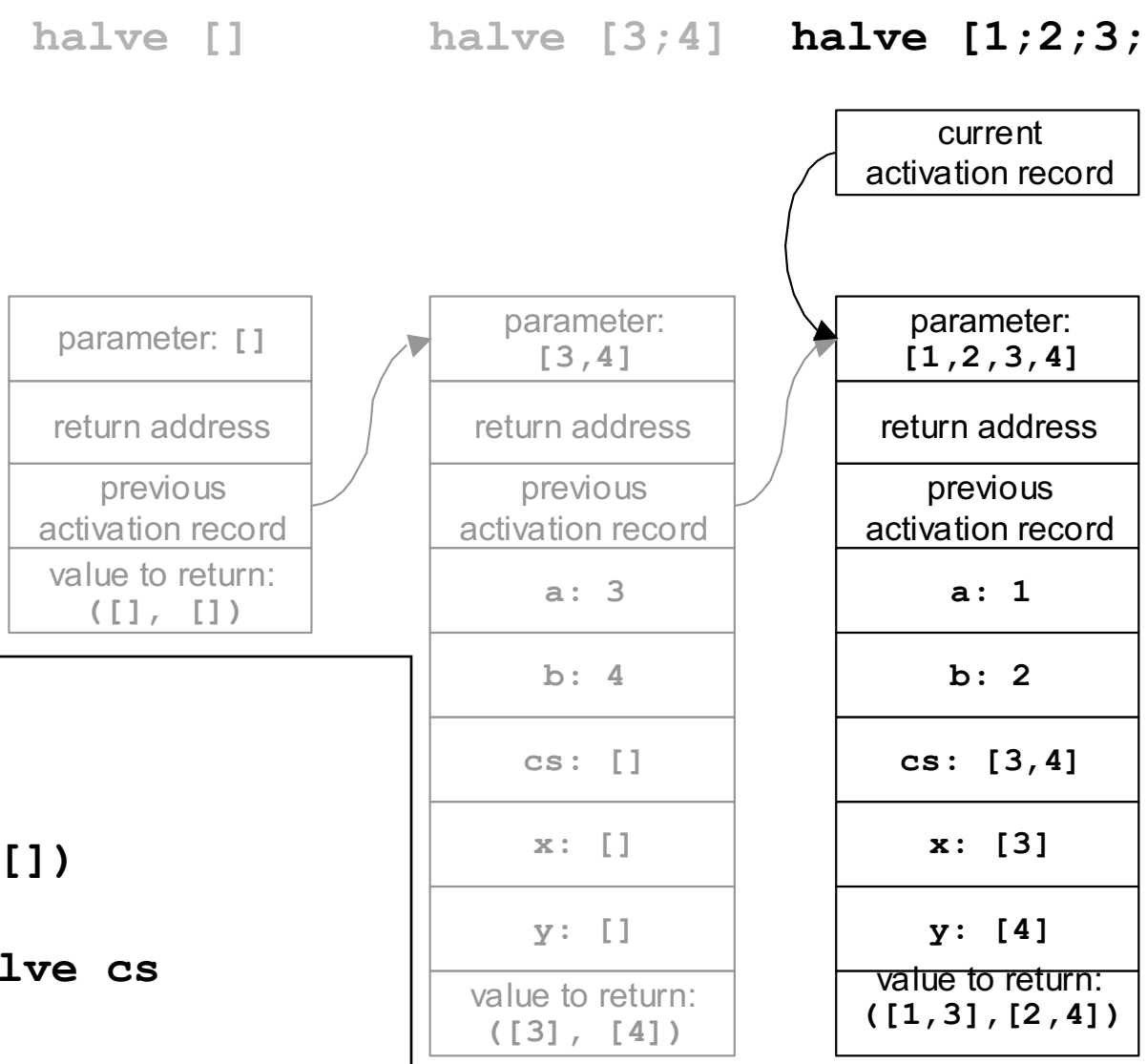
The second activation is about to return.




```
let rec halve L =
  match L with
    | [] -> ([], [])
    | a::[] -> ([a], [])
    | a::b::cs ->
      let (x, y) = halve cs
      in
      (a::x, b::y);;
```

The first activation is about to return with the result

halve [1,2,3,4] = ([1,3],[2,4])



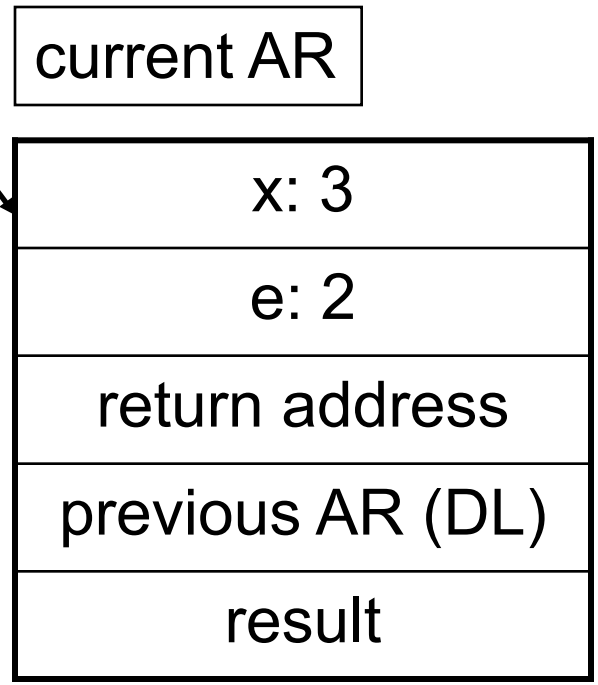

```
let rec halve L =
  match L with
    | [] -> ([], [])
    | a::[] -> ([a], [])
    | a::b::cs ->
      let (x, y) = halve cs
      in
      (a::x, b::y);;
```

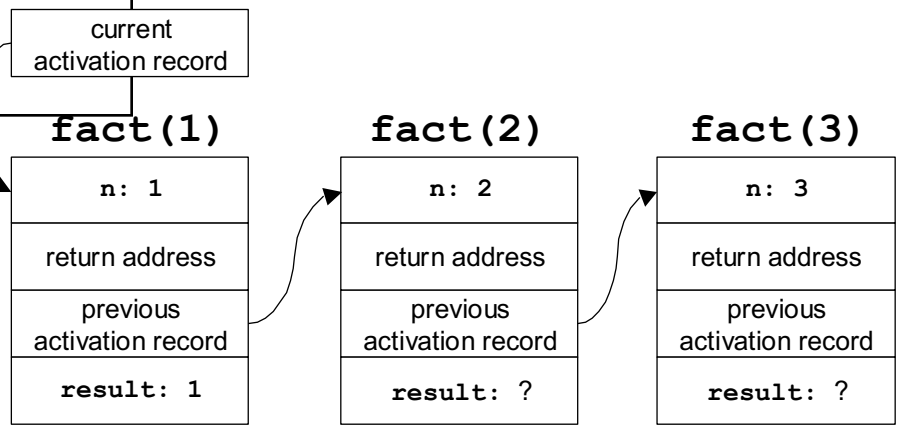
Exercise 3

Diagram stack to deepest call:

```
let rec power x e =
  match (x,e) with
    | (_, 0) -> 1
    | (x, e) -> x * (power x  (e-1))
  }

power  3 2
```

```
int power (int x, int e) {
        if (e == 0) return 1;
        else return x * power( x,  e-1);
}

power (3, 2);
```





Outline

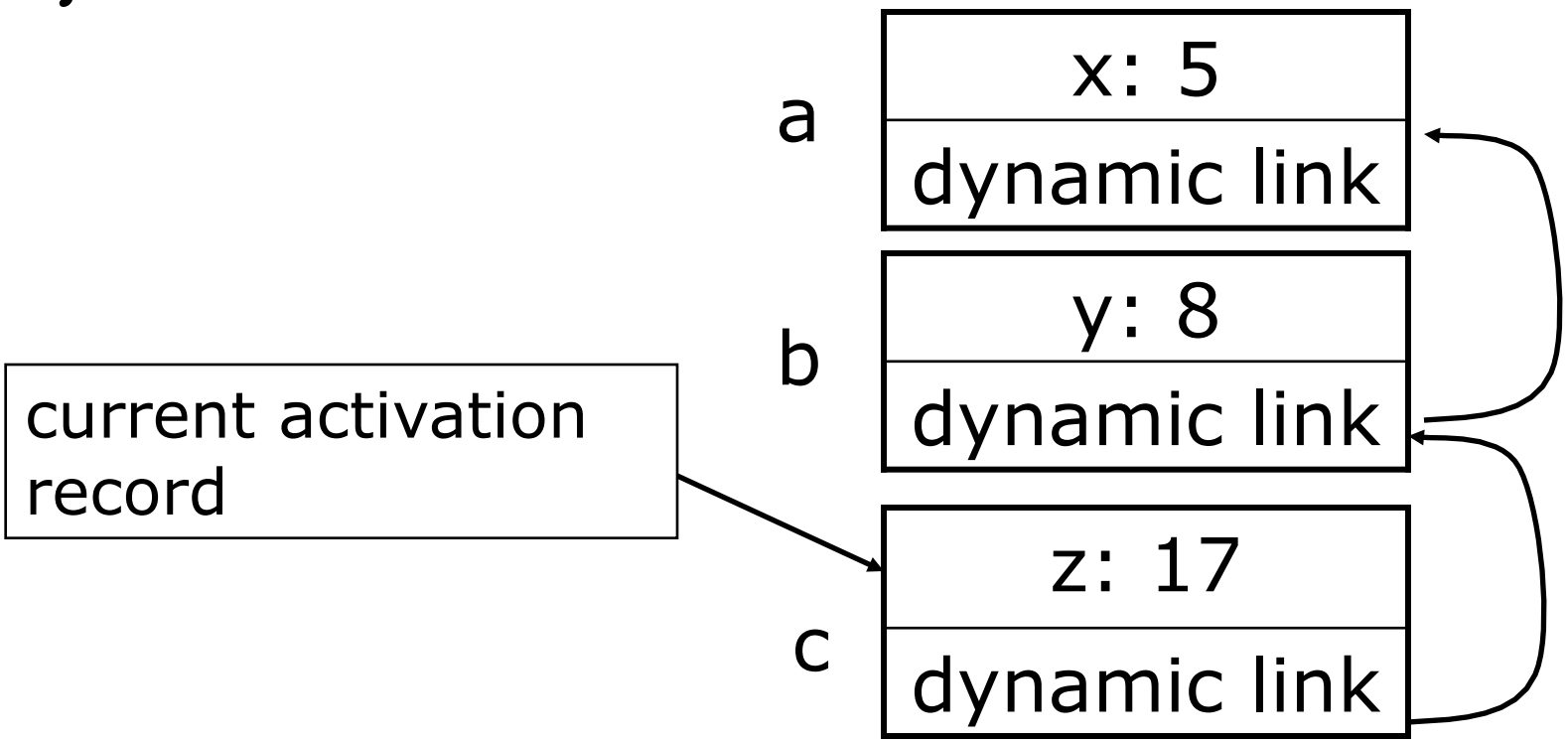
- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records

Nesting Functions

- What we just saw is adequate for many languages, including C
- But not for languages that allow:
  - Function definitions can be nested inside other function definitions
  - Inner functions can refer to local variables of the outer functions (under the usual block scoping rule)
- Like F#, Scala, JavaScript, Pascal, Java, etc.

C++ Scope Nested

```
a:   int x = 5;
  {  b:   int y = x+3;
     {  c:   int z = x + y + 4;
     }
  }
```



Example – Nested Functions

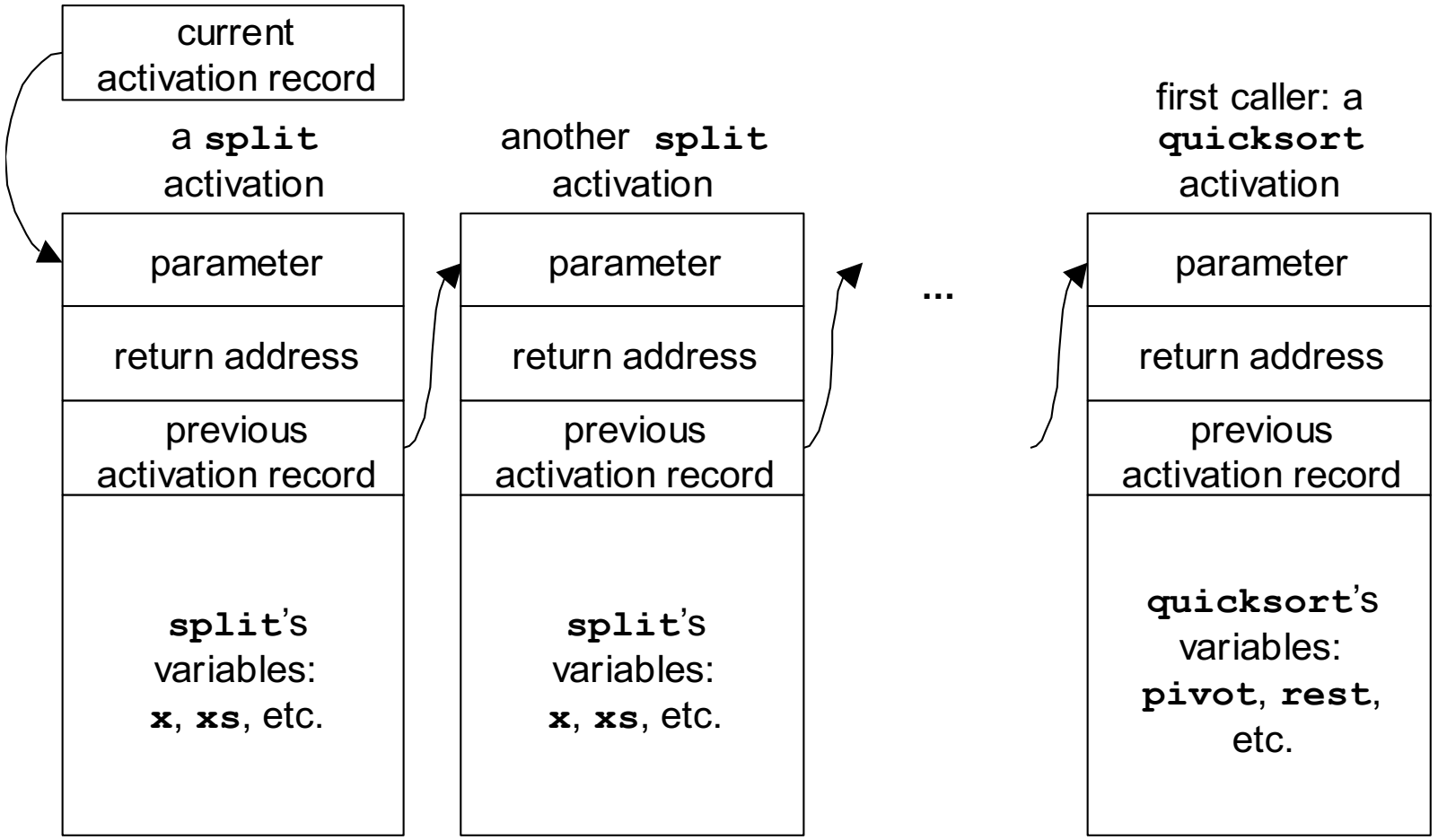
Other variables accessible through nested scope?

```
let rec quicksort L1 =   match L1 with
    | [] -> []
    | pivot::rest ->
      let rec split L2 =
        match L2 with
          | [] -> ([], [])
          | x::xs ->
            let (below, above) = split xs
            in
            if x<pivot then (x::below, above)
            else (below, x::above)
      in
      let (below, above) = split rest
      in
      quicksort below@[pivot]@(quicksort above)
```

The Problem

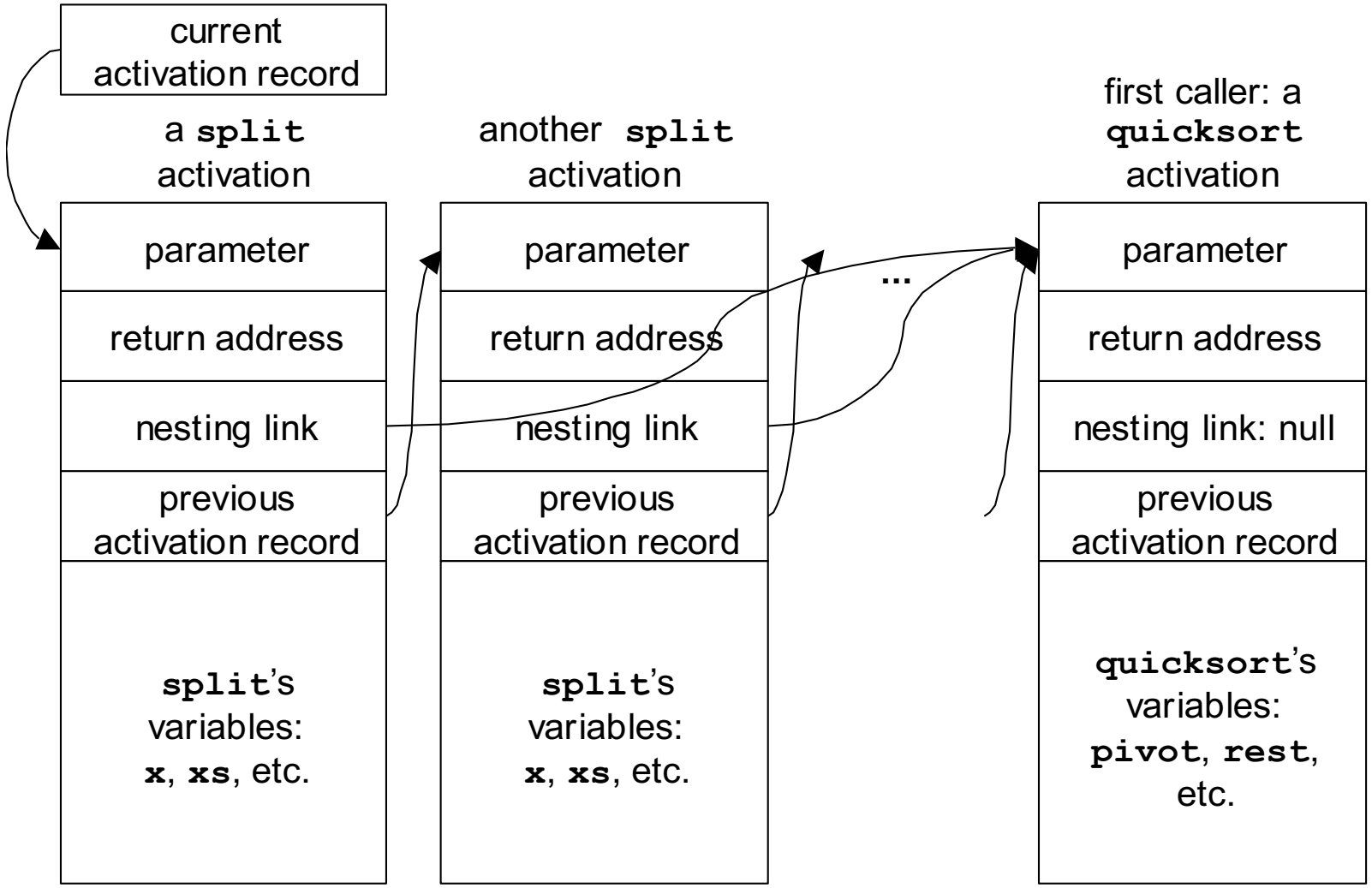
- How can an activation of the inner function (**`split`**) find the activation record of the outer function (**`quicksort`**)?
- It isn't necessarily the previous activation record, since the caller of the inner function may be another inner function
- Or it may call itself recursively, as **`split`** does…

`Dynamic link` points to **`caller's`** activation record



Nesting Link

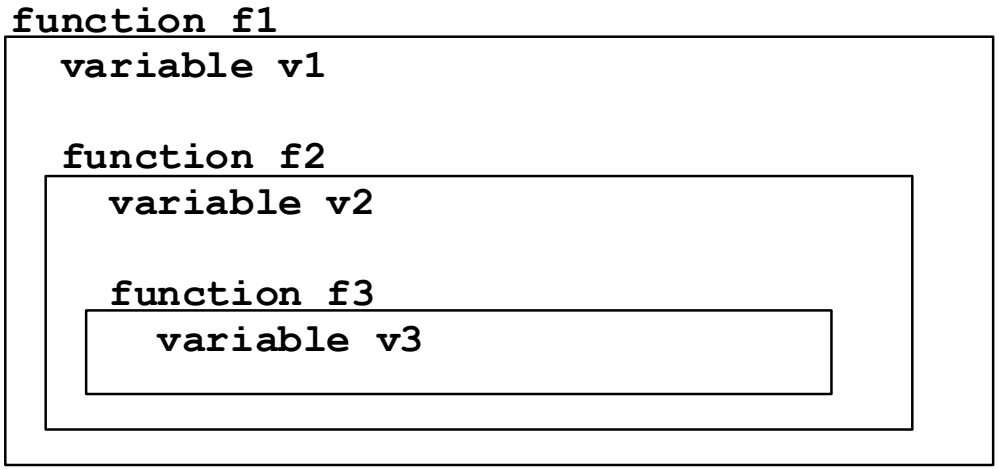
- An inner function needs to be able to find the address of the most recent activation for the outer function
- We can keep this *nesting link* in the activation record…


current
activation record
a split
activation
another split
activation
first caller: a
quicksort
activation
parameter
return address
nesting link
previous
activation record
split's
variables:
x, xs, etc.
parameter
return address
nesting link
previous
activation record
split's
variables:
x, xs, etc.
...
parameter
return address
nesting link: null
previous
activation record
quicksort's
variables:
pivot, rest,
etc.

Setting The Nesting Link

- Easy if there is only one level of nesting:
  - Calling outer function: set to null
  - Calling from outer to inner: set nesting link same as caller's activation record
  - Calling from inner to inner: set nesting link same as caller's nesting link
- More complicated if there are multiple levels of nesting…

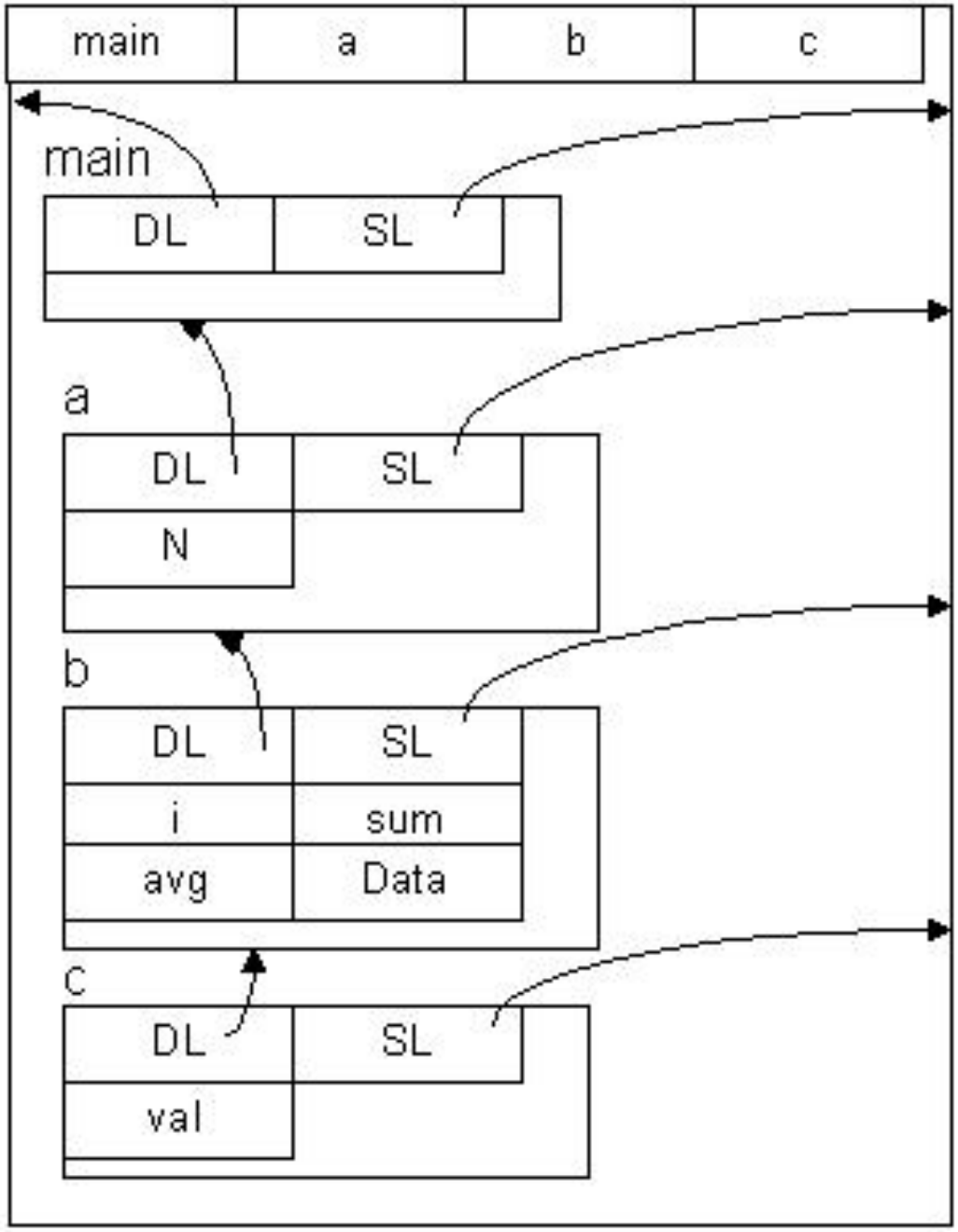
Multiple Levels Of Nesting



- References at the same level (**`f1`** to **`v1`**, **`f2`** to **`v2`**, **`f3`** to **`v3`**) use current activation record
- References *n* nesting levels away chain back through *n* nesting links
- **Static Link -** Points to activation record of *enclosing block.*
- **Dynamic Link -** Points to activation record of *caller*.

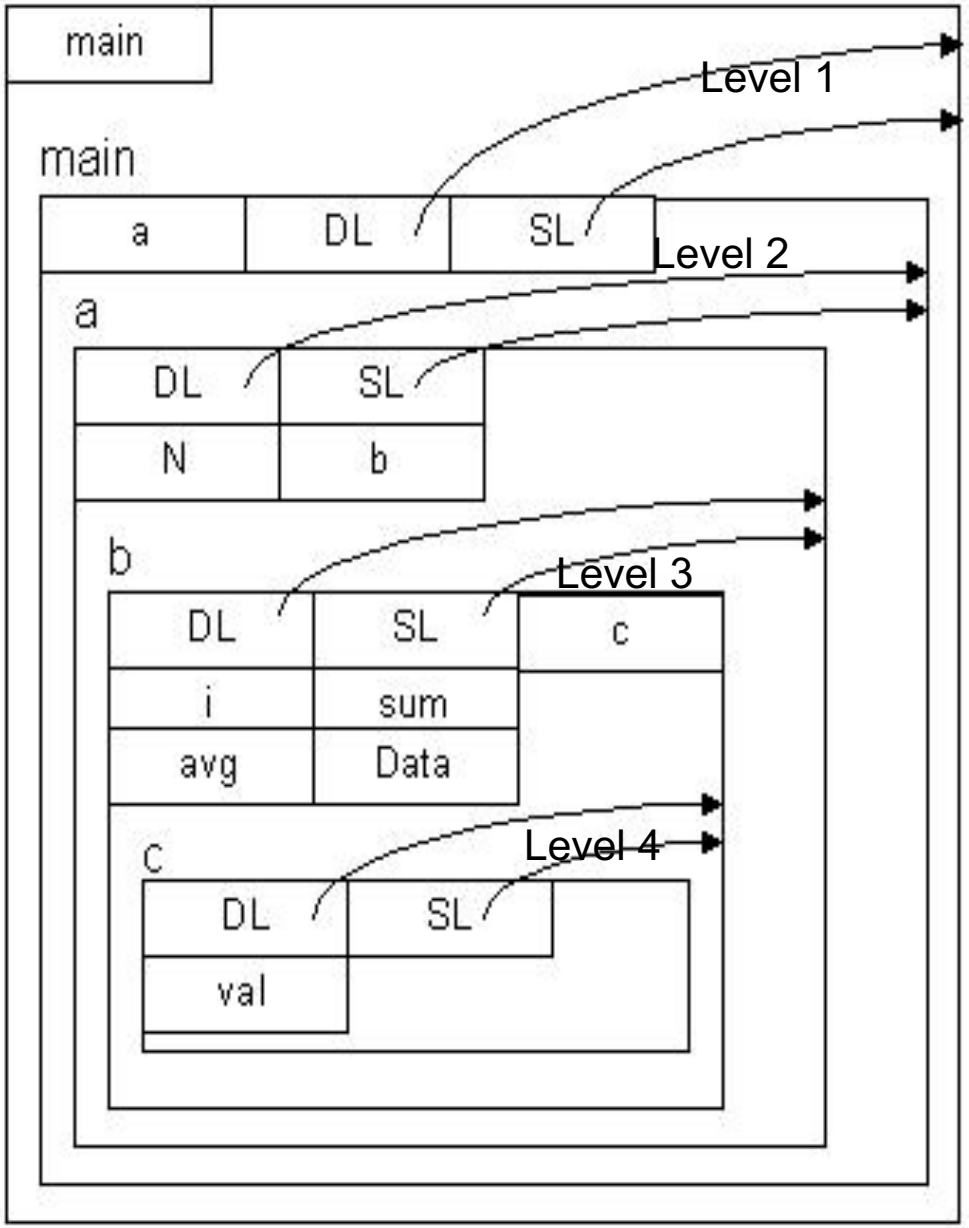
Static Nesting

```
1   Void a( )
2   {  int N;
3      N = 1;
4      b(19.3);
5   }                                  Level 1
6   Void b(
                 float sum) {
7      int i;
8      float avg;
9      float Data[2];
10      N = 2;
11     c(5.8);
12  }                                  Level 1
13  Void c (
                  float val)  {
14     cout << val;
15  }                                  Level 1
16  Void main()
17  {   a();
18  }                                  Level 1

                                       Level 0
```



Static Nesting

```
1.  void main()
2.  {  void a()
3.     {  int N;
4.        void b(
5.                 float sum)
6.        {   int i;
7.            float avg;
8.            float Data[2];
9.            void c(
10.                    float val)
11.           {  cout << sum;
12.              cout << N;
13.              a();
14.           }                    Level 4
15.           N = 2;
16.           c(5.8);
17.       }                        Level 3
18.       N = 1;
19.       b(19.3);
20.    }                           Level 2
21.    a();
22. }                              Level 1
```



Static Nesting Definitions

- **Activation record** – Contains local variables, parameters, links, etc.
- **ep** – *Environment pointer* to current activation record.
- **ip** – *Instruction pointer* to the current instruction.
- **Dynamic link** – Points to the **calling** function's activation record.
- **Static link** – Points to the **enclosing** environment's activation record. Represents the non-local data accessible to the function.
- **Static chain** – The static links from one enclosing environment to another.
- **SNL (Static Nesting Level)** – The number of enclosing environments where a symbol is defined or used.
- **SD (Static Distance) –** Difference between the SNL of *definition* and SNL of *use*, more intuitively, the number of static links in the static chain. SD to local data is 0, SD to nearest enclosing function is 1, etc.
- **Symbol table –** In statically nested environments, table recording symbol name, data type, SNL, and offset within the activation record. Used at compile time to generate code to access data bound to symbol.

Static Nesting Calculations

```
1.  void main()
2.  {  void a()
3.     {  int N;
4.        void b(
5.                 float sum)
6.        {   void c(
7.                       float val)
8.           {  cout << sum;
9.              cout << N;
10.             a();
11.          }                    Level 4
12.          N = 2;
13.          c(5.8);
14.       }                       Level 3
15.       N = 1;
16.       b(19.3);
17.    }                          Level 2
18.    a();
19. }                             Level 1
```

Level 0

SNL Definition	SNL Use
2 Line 3 N	2 Line 15 N
3 Line 5 sum	4 Line 9 N
1 Line 2 a	4 Line 8 sum
4 Line 7 val	4 Line 10 a

SD = SNL Use - SNL Definition

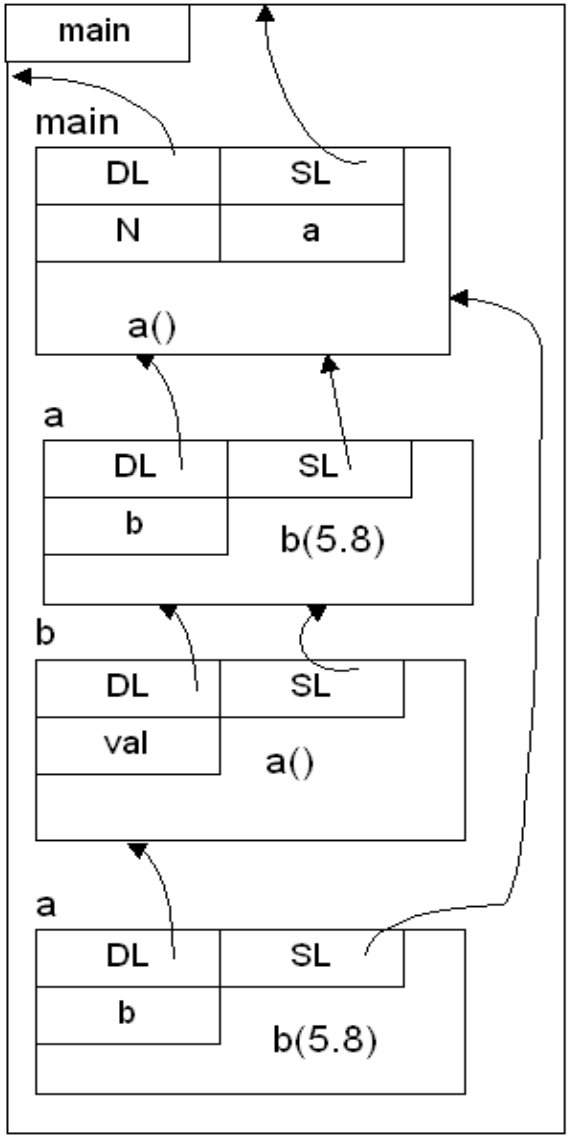
0 Line 15 SD N = 2 - 2
1 Line 8 SD sum = 4 - 3
2 Line 9 SD N = 4 - 2
3 Line 10 SD a = 4 - 1

Symbol Table

Name	Type	SNL	Offset
N	int	2	0
sum	float	3	0
val	float	4	0

```
1.  void main()
2.  {  int N;
3.     void a()
4.     {   void b(
5.            float val)
6.         {  cout << val;
7.            cout << N;
8.            a();
9.         }                    Level 3
10.        N = -9;
11.        b(5.8);
12.    }                        Level 2
13.    N = 6;
14.    a();
15. }                           Level 1
```

Level 0



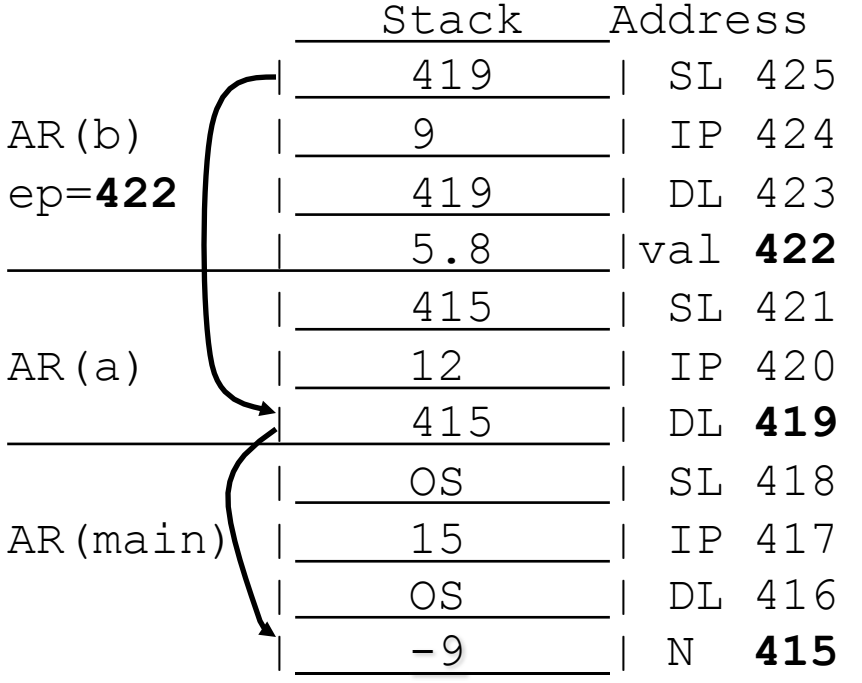
```
1.  void main()
2.  {  int N;
3.     void a()
4.     {   void b(
5.            float val)
6.         {  cout << val;
7.            cout << N;
8.            a();
9.         }                    Level 3
10.        N = -9;
11.        b(5.8);
12.    }                        Level 2
13.    N = 6;
14.    a();
15. }                           Level 1
                                Level 0
```

General access to variable:

1. ap ← ep
2. for (i=1; i<=SD(v); i++) ap←M[ap].SL
3. access M[ap+offset(v)]

Example: **cout<<N** at Line 7, SD(N)=2:

1. ap ← ep = 422
2. ap ← M[ap].SL=M[422].SL=419
   ap ← M[ap].SL=M[419].SL=415
3. cout<<M[ap+offset(N)]=M[415+0]=-9



Line 7 SD(N)=2
Line 10 SD(N)=1
Line 13 SD(N)=0

Exercise:
Compute the access of:
N=-9;
at Line 10.

Exercise 4

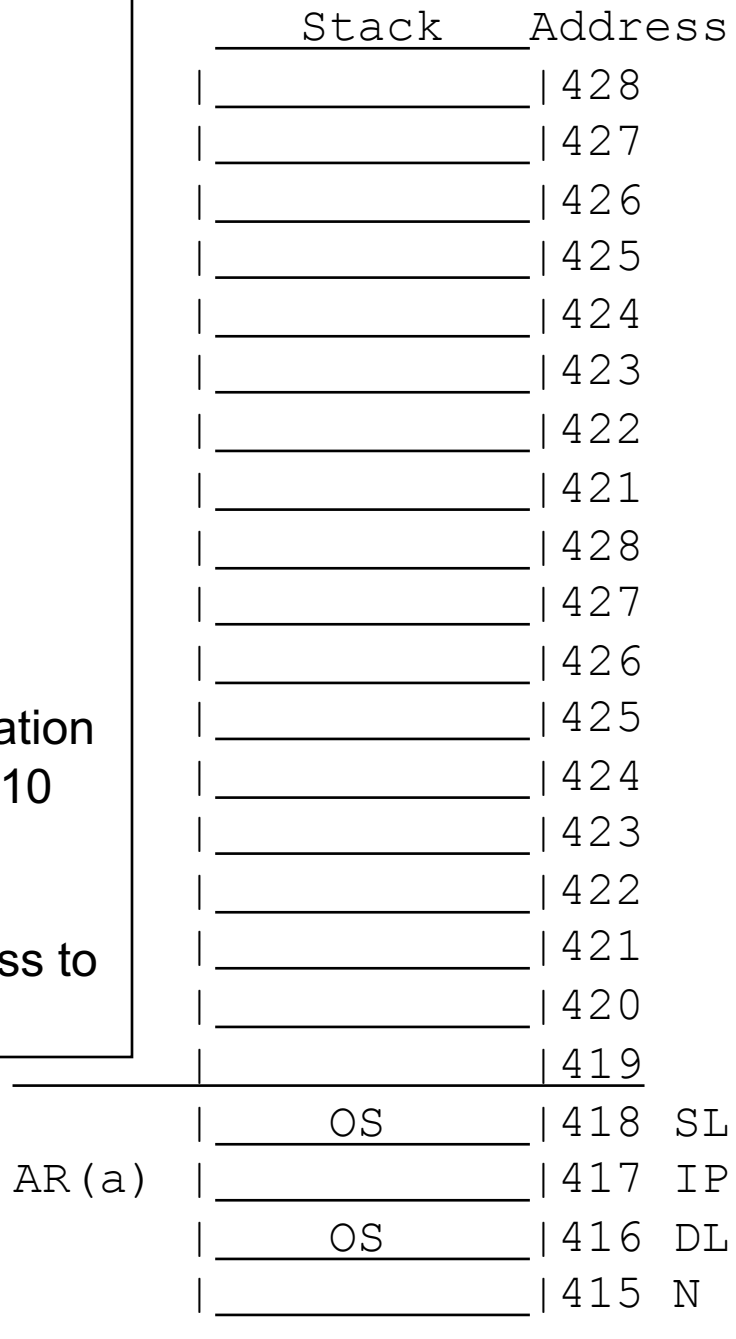
```
1.  void a()
2.  { int N;

3.     void b()
4.     {  N = -2;
5.        c(5.8);
6.      }

7.     void c(
8.           float val)
9.     {  cout << val;
10.       cout << N;
11.     }

12.    N = -1;
13.    b();
14. }
```

1. Static nesting level of definition:
   a) N __
   b) b __
   c) c __
   d) val __

2. Static distance of identifiers at lines:
   a) Line 12, N __
   b) Line 10, N __
   c) Line 9, val __

3. Fill in stack information line 1 through line 10 execution.

4. Compute the access to N at Line 10.



General access to variable:

1. ap←ep
2. for (i=1; i<=SD(v); i++) ap←M[ap].SL
3. access M[ap+offset(v)]

Other Solutions

- The problem: references from inner functions to variables in outer ones
  - Nesting links in activation records: as shown
  - Displays: nesting links not in the activation records, but collected in a single static array
  - Lambda lifting: problem references replaced by references to new, hidden parameters

Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records

Functions As Parameters

- When you pass a function as a parameter, what really gets passed?
- Code must be part of it: source code, compiled code, pointer to code, or implementation in some other form
- For some languages, something more is required…

Exercise 5 - C++ Example

```
1.  void p(int x) {
2.    cout << "p " << 2*x;
3.  }

4.  void t(int x) {
5.    cout << "t " << x*x;
6.  }

7.  void q( void fp(int), int x) {
8.    fp(x);
9.  }

10. void main(void) {
11.   q( p, -4 );
12.   q( t, -5 );
13. }
```

1. Trace lines executed.

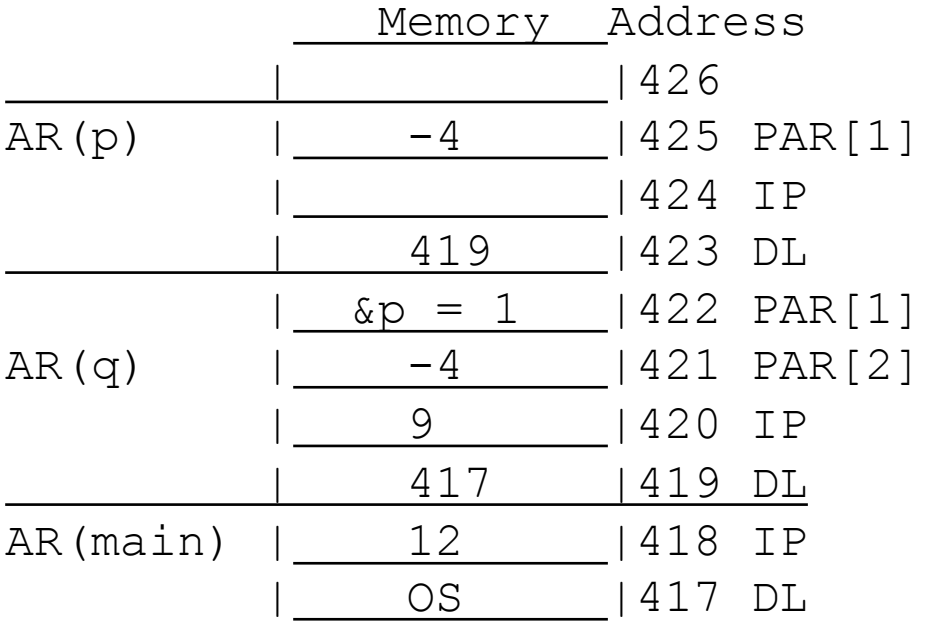
2. What is the output?

C++ Example – Function Parameters

```
1.  void p(int x) {
2.    cout << "p " << 2*x;
3.  }

4.  void t(int x) {
5.    cout << "t " << x*x;
6.  }

7.  void q( void fp(int), int x) {
8.    fp(x);
9.  }

10. void main(void) {
11.   q( p, -4 );
12.   q( t, -5 );
13.}
```

Without nested environments, only the function address is passed as a parameter.

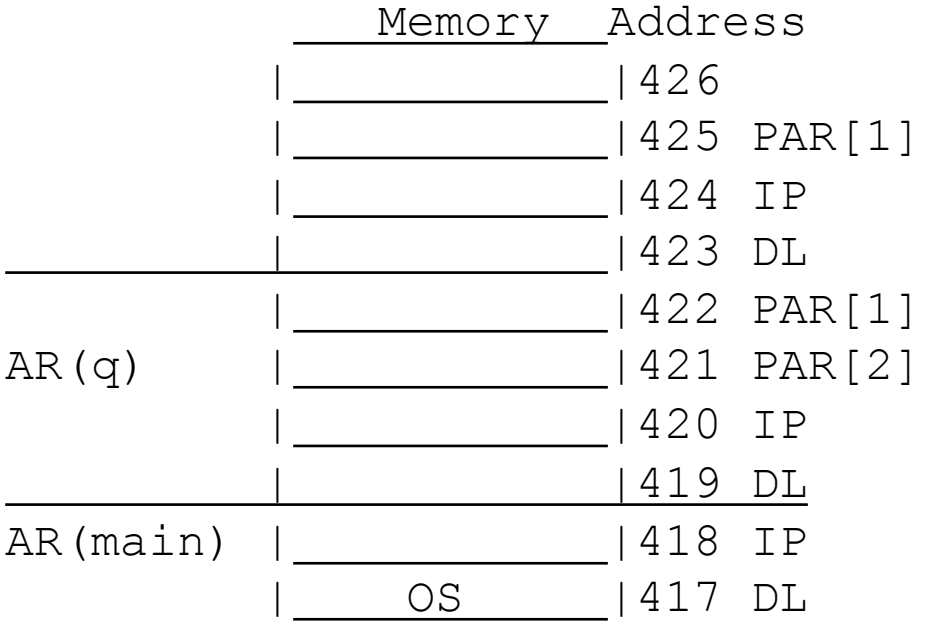
Execution of Line 11.



Exercise 6 - Function Parameters

```
1.  void p(int x) {
2.    cout << "p " << 2*x;
3.  }

4.  void t(int x) {
5.    cout << "t " << x*x;
6.  }

7.  void q( void fp(int), int x) {
8.    fp(x);
9.  }

10. void main(void) {
11.   q( p, -4 );
12.   q( t, -5 );
13.}
```

Fill in memory for execution starting at line 12.



F# Example

```
let rec map f L =
  match L with
     | [] -> []
     | h::t -> f h:(map f t)
```

```
let addXToAll x L =
  let addX y = y + x
  in
  map addX L
```

- This function adds **x** to each element of **theList**
- Notice: **addXToAll** calls **map**, **map** calls **addX**, and **addX** refers to a variable **x** in **addXToAll**'s activation record

Nesting Links Again

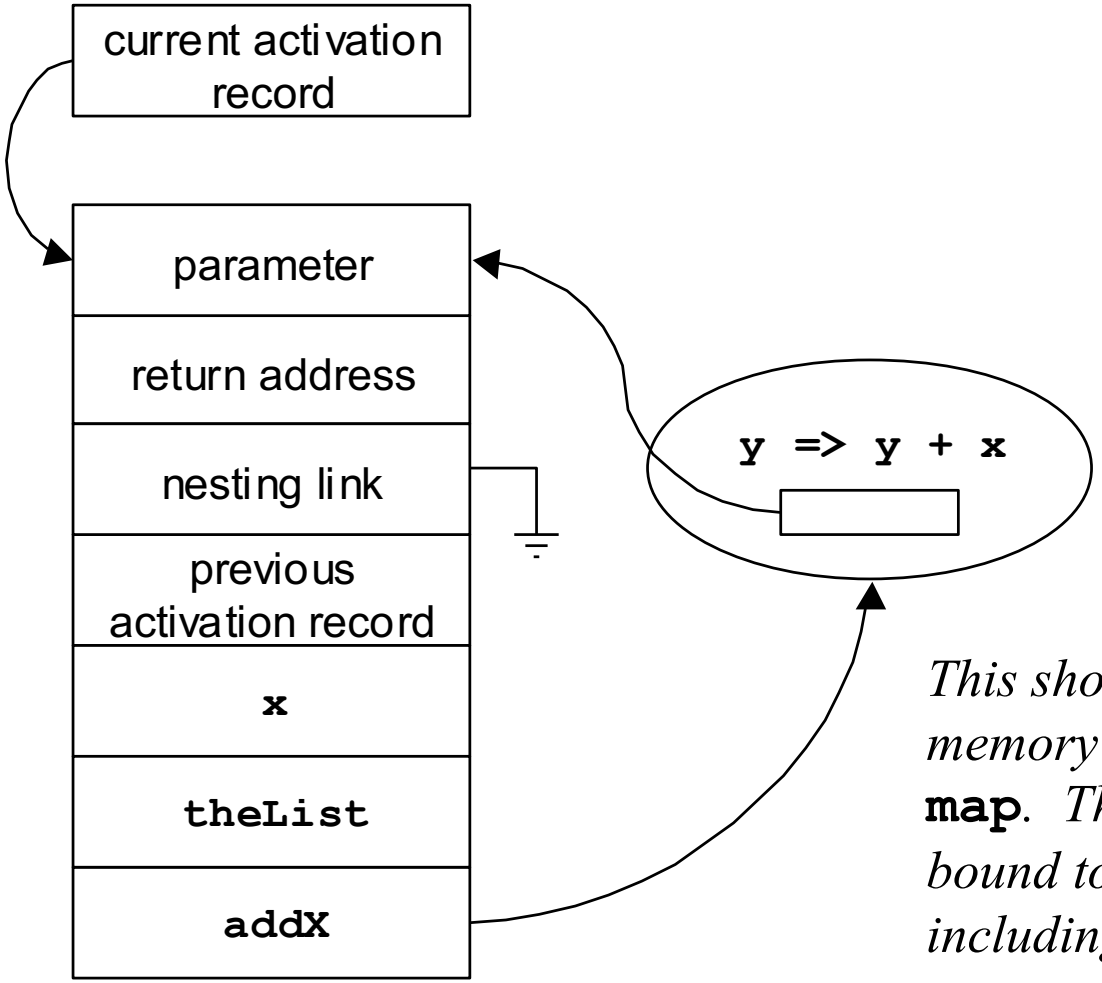
- When **`map`** calls **`addX`**, what nesting link will **`addX`** be given?
  - Not **`map`**'s activation record: **`addX`** is not nested inside **`map`**
  - Not **`map`**'s nesting link: **`map`** is not nested inside anything
- To make this work, the parameter **`addX`** passed to **`map`** must include the nesting link to use when **`addX`** is called

Not Just For Parameters

- Many languages allow functions to be passed as parameters
- Functional languages allow many more kinds of operations on function-values:
  - passed as parameters, returned from functions, constructed by expressions, etc.
- Function-values include both code to call, and nesting link to use when calling it

Example

```
let addXToAll x L =
  let addX y = y + x
  in
  map addX L
```



This shows the contents of memory just before the call to **`map`**. *The variable* **`addX`** *is bound to a function-value including code and nesting link.*

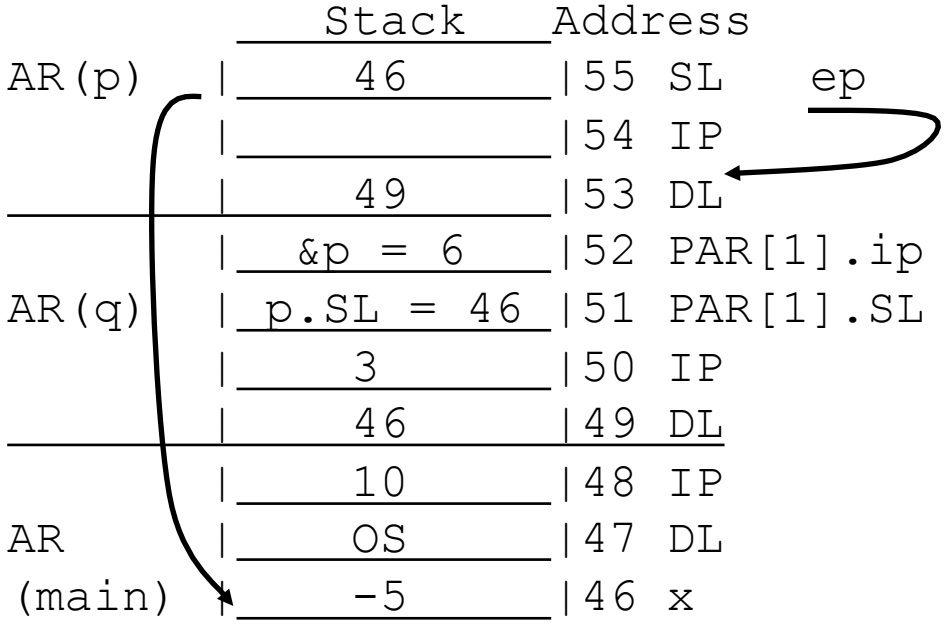
C++Style Example

```
1.  void q( void fp( ) ) {
2.    fp();
3.  }

4.  void main(void) {
5.    int x = -5;

6.    void p() {
7.      cout << x;
8.    }

9.    q( p );
10. }
```

In a nested environment, passing functions as parameters requires passing:

1. a reference to the enclosing environment, `PAR[1].SL`
2. Address of function parameter, `PAR[1].ip`

In the example, Line 7 accesses x, with SD(x) = 1.



Outline

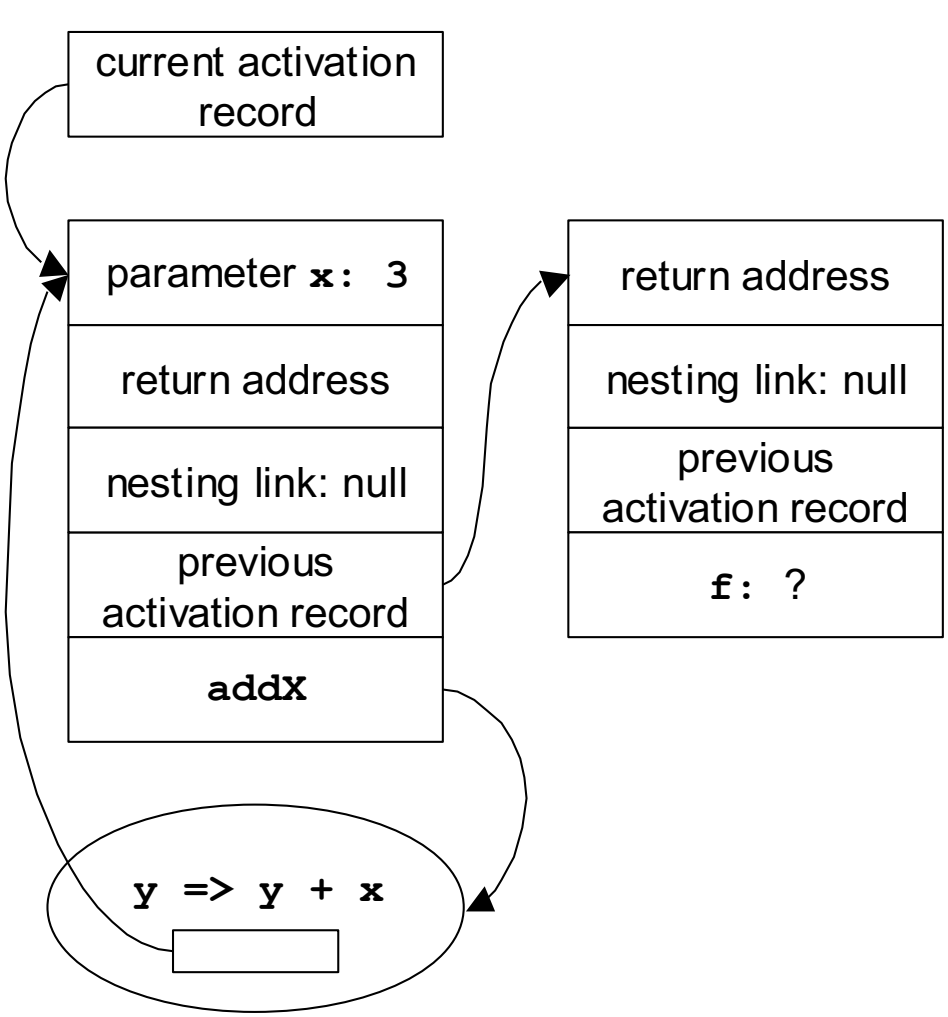
- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records

One More Complication

- What happens if a function value is used after the function that created it has returned?

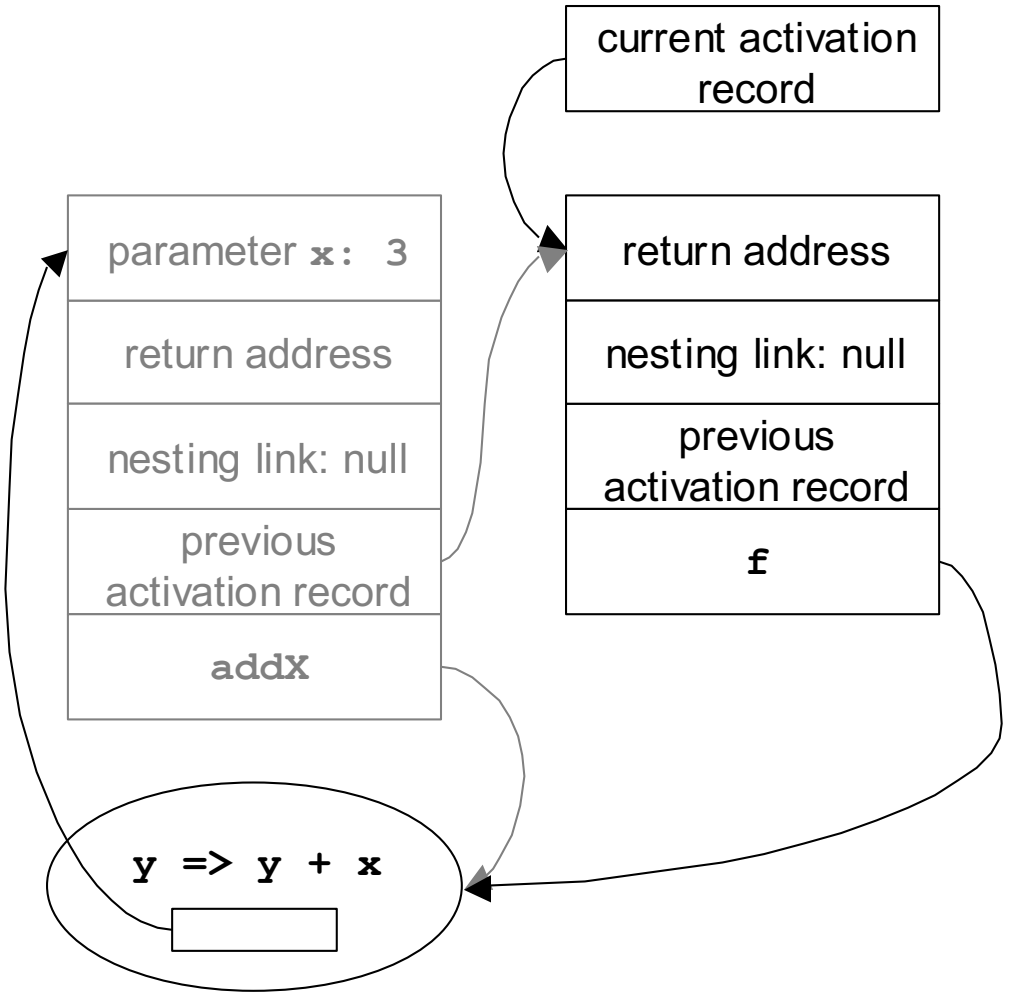
```
let funToAddX x =
  let addX y = y + x
  in
  addX;;

let test =
  let f = funToAddX 3
  in
  f 5;;
```

```
let funToAddX x =
  let addX y = y + x
  in
  addX;;

let test =
  let f = funToAddX 3
  in
  f 5;;
```



This shows the contents of memory just before **funToAddX** *returns.*

```
let funToAddX x =
  let addX y = y + x
  in
  addX;;

let test =
  let f = funToAddX 3
  in
  f 5;;
```



- **test** calls **f** which is **y => y + x**
- To access **x=3** in **test** must link to activation record for **funToAddX** that is already finished
- Fails if the language system deallocated that activation record when **funToAddX** returned

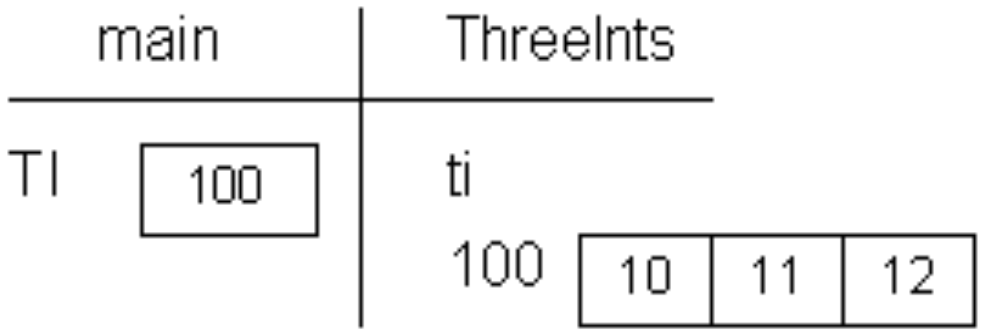
After **funToAddX** *returns,* **f** *is the bound to the new function-value.*

The Problem

- When **`test`** calls **`f`**, the function will use its nesting link to access **`x`**
- That is a link to an activation record for an activation that is finished
- This will fail if the language system deallocated that activation record when the function returned

C++ Example – Locals in AR

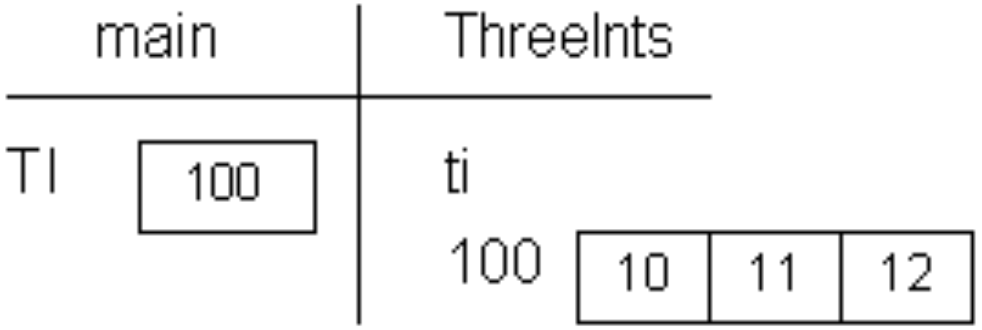
```
int * ThreeInts() {
      int ti[5] = {10,11,12};
      for (int i = 0; i<3; i++) cout << ti[i];
      return ti;
}
void main() {
      int * TI = ThreeInts();
      for (int i=0; i<3; i++) cout << TI[i];
}
```



What is the output?

Java Example

What is the output?




```
public class Example {
  public static void main(String a[]) {
     int TI[] = ThreeInts();
     for (int i=0; i<3; i++)
          System.out.print(TI[i]);
  }
  public static int[] ThreeInts() {
     int ti[] = {10,11,12};

     for (int i=0; i<3; i++)
          System.out.print(ti[i]);
     return ti;
     }
}
```

The Solution

- For F#, and other languages that have this problem, activation records cannot always be allocated and deallocated in stack order
- Even when a function returns, there may be links to its activation record that will be used; it can't be deallocated if it is reachable
- *Garbage collection*: chapter 14, coming soon!

Conclusion

- The more sophisticated the language, the harder it is to bind activation-specific variables to memory locations
  - Static allocation: works for languages that permit only one activation at a time (like early dialects of Fortran and Cobol)
  - Simple stack allocation: works for languages that do not allow nested functions (like C)

Conclusion, Continued

- Nesting links (or some such trick): required for languages that allow nested functions (like F#, Ada and Pascal); function values must include both code and nesting link
- Some languages (like F#) permit references to activation records for activations that are finished; so activation records cannot be deallocated on return