Variables in Memory

A Binding Question

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

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:

Example

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

Example

```
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:

Example

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

Example

```
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)

Example

```
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:

Example

```
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 soon

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

Static Allocation

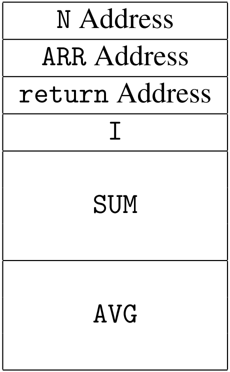
Static Allocation

- 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

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
```



Value and Reference Parameter Passing

Example

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

	x after	y after
pass by reference	3	2
pass by value	2	3

Reference passing danger

- Passing literals by reference must be prevented
- Question: How can this problem be prevented?

Example

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

Exercise

Example (C++ Result?)

```
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

- Simple, only one activation record per procedure.
- Memory allocation done at load time.
- Does not allow recursion. Why?
- Does not allow for nested scope. Why?
- Faster than dynamic allocation.

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

Stacks of ARs

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

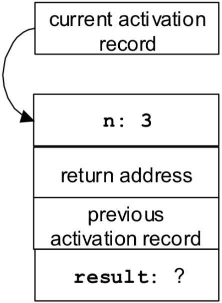
- Static: location of activation record was determined by compile time
- Dynamic: location of the *current* activation record is not known until runtime
- A function must know how to find the address of its current activation record
- Often, a special machine register (ebp on Intel) holds *current* activation 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.

Example

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



current activation record

n: 3

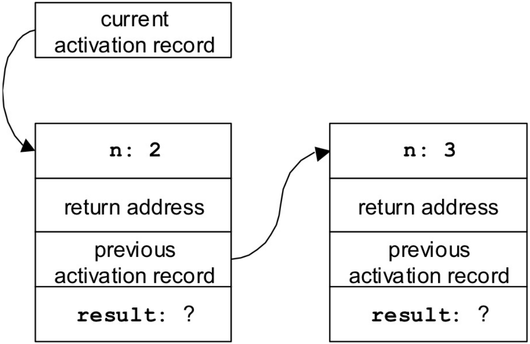
return address

previous activation record

result: ?

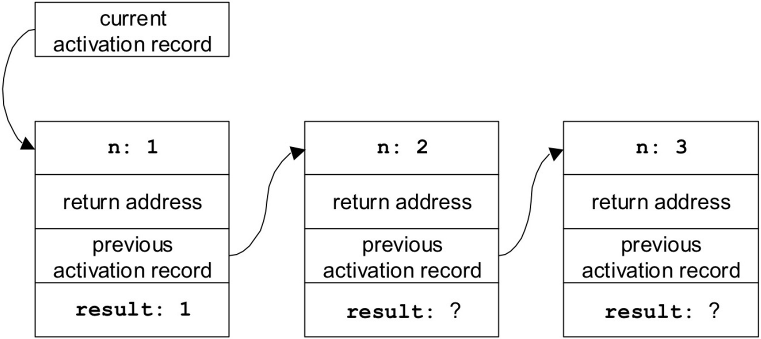
Second call

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



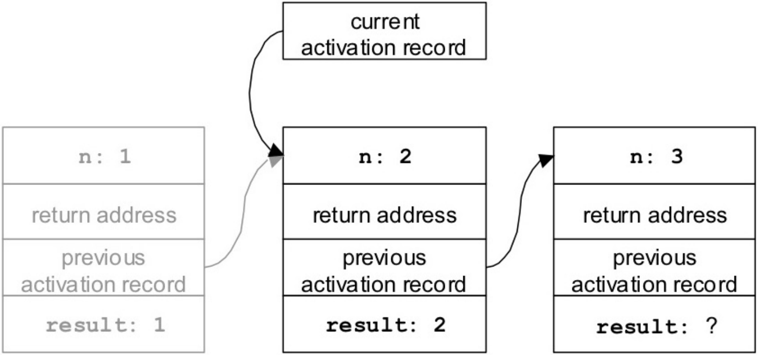
Third call

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



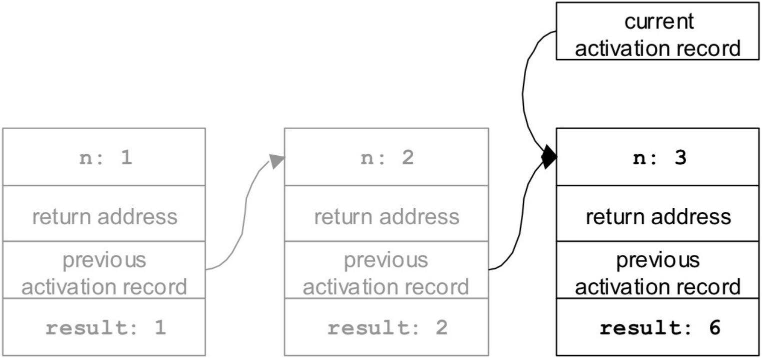
Returning from second call

The second activation is about to return.



Returning from first call

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



Exercise

Diagram the stack to deepest call:

Example

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

power (3, 2);
```

Nested functions

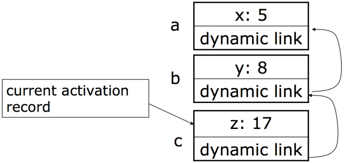
Handling Nesting Functions

- 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 that can refer to local variables of the outer functions (under the usual block scoping rule)
- Like F#, Scala, JavaScript, Pascal, Java, etc.

C++ Nested Scope

Example

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



current activation record

a
x: 5
dynamic link

b
y: 8
dynamic link

c
z: 17
dynamic link

F# Nested Scope

pivot on the last line refers to a variable outside the current scope.

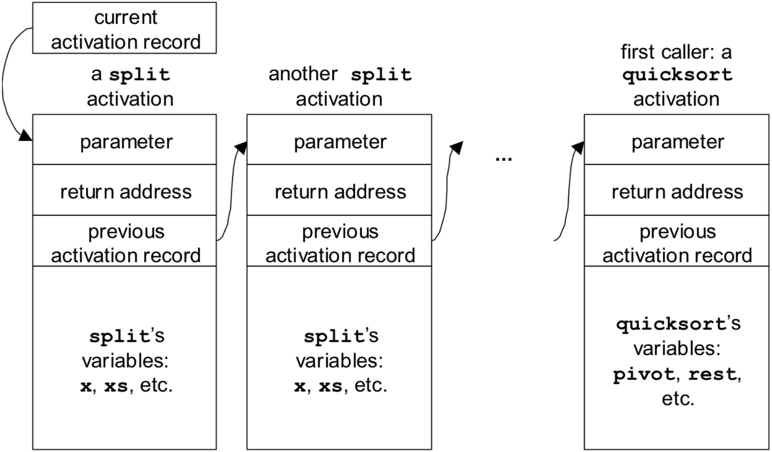
Example

```
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

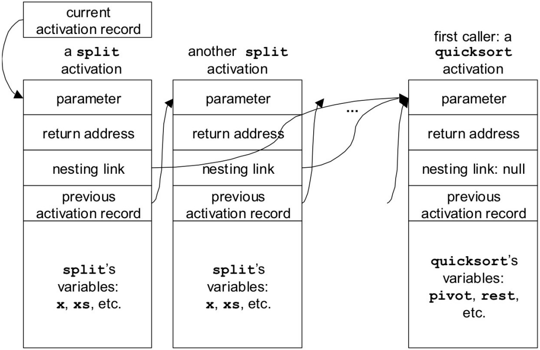
- How can an activation of the inner function (**split**) find the activation record of the outer function (**quicksort**)?
- It isn't necessarily the previous activation record, since the caller of the inner function may be another inner function
- Or it may call itself recursively, as **split** does…

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

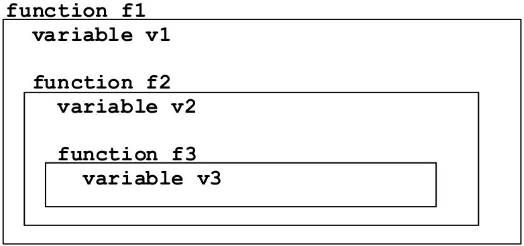


Setting The Nesting Link

- 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*.

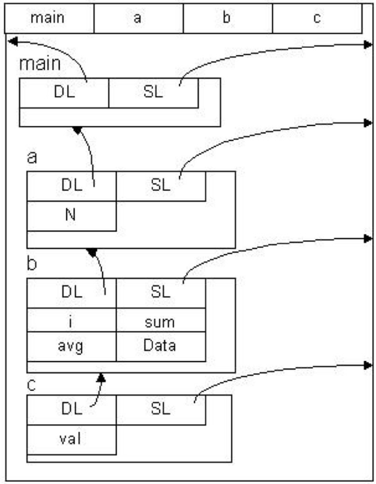


```
function f1
  variable v1

  function f2
    variable v2

    function f3
      variable v3
```

Static Nesting

Example

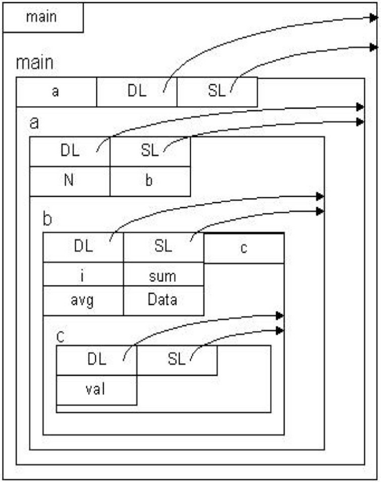
```
Void a( )
{   int N;
    N = 1;
    b(19.3);
}
Void b(float sum) {
    int i;
    float avg;
    float Data[2];
    N = 2;
    c(5.8);
}
Void c (float val)  {
    cout << val;
}
Void main() {
    a();
}
```



Static Nesting

Example

```
void main()
{  void a()
   {  int N;
      void b(float sum)
      { int i;
        float avg;
        float Data[2];
        void c(float val)
        { cout << sum;
          cout << N;
          a();
        }
        N = 2;
        c(5.8);
      }
      N = 1;
      b(19.3);
   }
   a();
}
```



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.

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

Functions as parameters

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

Trace lines executed. What is the output?

Example (C++)

```
void p(int x) {
  cout << "p " << 2*x;
}
void t(int x) {
  cout << "t " << x*x;
}
void q(void fp(int), int x) {
  fp(x);
}
void main(void) {
  q( p, -4 );
  q( t, -5 );
}
```

C++ Example

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

Table: Memory Layout

	Memory	Address
		426
AR(p)	-4	425 PAR[1]
		424 IP
	419	423 DL
	&p=1	422 PAR[1]
AR(q)	-4	421 PAR[2]
	9	420 IP
	417	419 DL
AR(main)	12	418 IP
	OS	417 DL

F# Example

Example

```
let rec map f L =
  match L with
    | [] -> []
    | h::t -> f h::(map f t)
let addXToAll x theList =
  let addX y = y + x
  in
  map addX theList
```

- 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

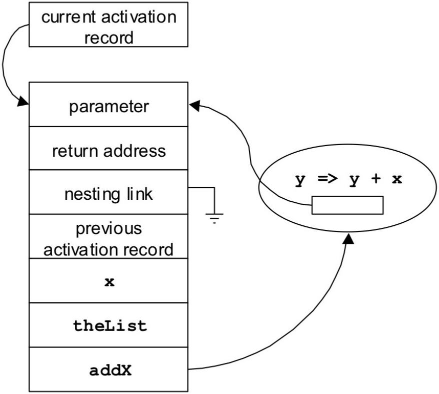
- Many languages allow functions to be passed as parameters
- Functional languages allow many more kinds of operations on function-values:
  - passed as parameters
  - returned from functions
  - constructed by expressions
  - etc.
- Function-values include both code to call, and nesting link to use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

F# Example

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.

Example

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



current activation record

parameter

return address

nesting link

previous activation record

x

theList

addX

`y => y + x`

Long-lived AR

One More Complication

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

Example

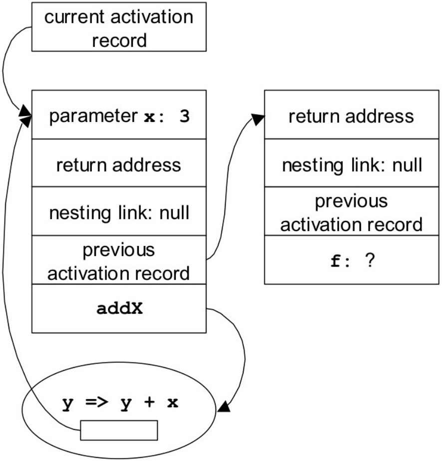
```
let funToAddX x =
  let addX y = y + x
  in
  addX;;
let test =
  let f = funToAddX 3
  in
  f 5;;
```

One More Complication

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

Example

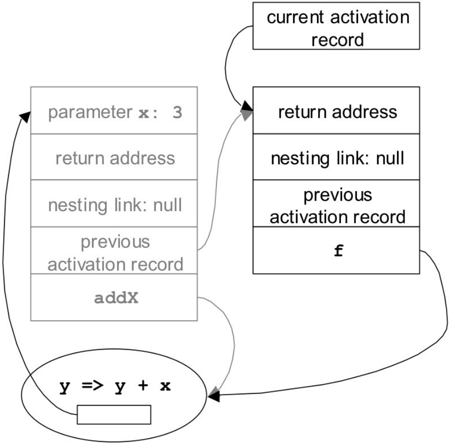
```
let funToAddX x =
  let addX y = y + x
  in
  addX;;
let test =
  let f = funToAddX 3
  in
  f 5;;
```



One More Complication

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

- **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



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

Java Example

What is the output?

Example

```
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: 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)
- 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