Imperative Programming — Procedure Activations

Steven Zeil

Oct. 4, 2001

Contents

Imperative Languages

1. Statements
2. Data Structures
3. Procedure Activations

---

3 Procedure Activations

1. Basics
2. Parameter Passing
3. Scopes
4. Activation Records
5. Case study: C
6. Case study: Pascal

3.1 Basics

1. Procedures
2. Bindings

3.1.1 Procedures (Subroutines)

Two kinds of procedures:

- **Function procedures** return a value.
- **Proper procedures** do not.

A procedure has 4 parts:

1. **name** of procedure
2. **formal parameters**
3. **body**, consisting of
   - local declarations
   - statement (list)
4. an optional **return type**

```
int fact (int n) {return n*fact(n-1);}
```

```
FUNCTION Fact (n: integer): integer;
BEGIN
  Fact := n*Fact(n-1)
END
```

3.1.2 Bindings

A **binding** is an association of a value with some name/entity.

In any program, “names” must somehow be bound to their

- types
- locations (l-values)
- value (r-values)

Some of these bindings are static, others dynamic.

Some Common Bindings

name $\xrightarrow{\text{scope}}$ declaration $\xrightarrow{\text{activation}}$ l-value $\xrightarrow{\text{state}}$ r-value

---

3.2 Parameter Passing

A procedure call supplies **actual parameters** to be passed to the called routine.

Within the procedure body, these are referenced via the *formal parameter* names.

```
int fact (int n) {return n*fact(n-1);}
   ⋮
int i = fact(23);
```

`23` is the actual parameter.
`n` is the corresponding formal parameter

The process of matching formal parameters to actuals is called **parameter passing**.

There are many techniques for parameter passing:

- Call-by-Value
- Call-by-Reference
- Call-by-Value-Result
- Call-by-Name

Alternatively, we can classify parameter passing by the programmer's intent.

3.2.1 Call-by-Value

In **call-by-value**, the r-value of the actual parameter is copied into a local l-value within the called routine.

For a procedure `P` with formal parameter `x`, a call

```
P(E);
```

is equivalent to

```
x := E;
body of P;
if P is a function, return
  a result;
```

Call-by-value is the primary passing mechanism in Pascal, C, and C++.

Limitations:

- Cannot be used to send values back to the caller

```
procedure BadSwap (x, y: T);
var temp: T;
begin
  temp := x; x := y; y := temp;
end;
```

- Passing large objects (e.g., arrays) is time-consuming

3.2.2 Call-by-Reference

In call-by-reference, the formal parameter name is bound to the l-value of the actual (it becomes a synonym for the actual).

- Pascal `var` parameters are call-by-reference.
- FORTRAN uses call-by-reference.

- C programmers fake it by passing pointers (but the pointers are actually passed by value).
- C++ has reference types, which emulate call-by-reference.

Call-by-reference has constant overhead, can be used for output:

```
procedure Swap (var x: T; var y: T);
var temp: T;
begin
  temp := x; x := y; y := temp;
end;
```

C-simulation of call-by-reference:

```
void swap (T* x, T* y)
{
  T temp;
  temp = *x; *x = *y; *y := temp;
}
```

C++'s call-by-reference:

```
void swap (T& x, T& y)
{
  T temp;
  temp = x; x = y; y := temp;
}
```

Limitations of call-by-reference:

- Can only be applied to actual parameters that *have* l-values.
  - OK for `Swap(a,b)` where `a` and `b` are variables
  - OK for `Swap(A[i], Rec.field)`
  - Not allowed: `Swap(a, 2+3)`
  - Not allowed: `Swap(1,2)`
- Minimal protection against inadvertent changes

Aliasing

Aliasing refers to the ability to manipulate the same r-value through different names/expressions.

Call-by-reference can result in unexpected behavior due to aliasing.

Reference & Aliasing

```
void addVectors(const Vector& a,
                const Vector& b,
                Vector& c)
{ // Vector addition: c = a + b
  assert (a.size() == b.size());

  // make c empty
  c.erase(c.begin(), c.end());

  for (int i = 0; i < a.size(); ++i)
    // append a[i]+b[i] to c
    c.push_back (a[i] + b[i]);
}
```

What happens if the application code says:

```
// x = x + y;
addVectors (x, y, x);
```

3.2.3 Call-by-Value-Result

Also called **copy-in, copy-out**.
For a procedure `P` with formal parameter `x`,

```
P(E);
```

is equivalent to

```
x := E;
body of P;
E := x;
if P is a function, return
  a result;
```

So the parameter value is copied twice, once for input and once for output.

Properties of Value-Result

- Somewhat more predictable behavior in the presence of aliasing.
- Can be used for output
- High overhead (2 copies) for large objects

Value-Result & Aliasing

```
procedure addVectors (
    a, b, c: in out Vector) is
begin -- Vector add: c := a + b
  c.clear(); -- make c empty
  for i in 1..a'length loop
    c.push_back (at(a,i) + at(b,i));
  end loop;
end addVectors;
```

If application does

```
addVectors (x, y, x);
```

the function will run properly, but final value of `x` depends on which parameter gets copied last.

3.2.4 Call-by-Name

Call-by-name is equivalent to passing the actual text of the actual parameter to the procedure, substituting it for each occurrence of the formal parameter name:

```
#define min(x,y) (x<y) ? x : y
```

expands in the following calls as:

- `min(a,b)` becomes `(a<b) ? a : b`
- `min(a,0)` becomes `(a<0) ? a : 0`
- `min(a,4*a*c - b)` becomes `(a<4*a*c - b) ? a : 4*a*c - b`
- `min(a,b++)` becomes `(a<b++) ? a : b++`

Problems occur if an actual parameter

- is a time-consuming expression
- has side-effects

Call-by-name is seen mostly in macro expansion (e.g., the C/C++ `#define`).

- was used in Algol 60
  - now generally regarded as a bad decision
- but many specialized, interpreted languages still exist that work by macro expansion (e.g., TeX, TCL)

Implementing Call-by-Name

For compiled languages, call-by-name is not easy to do.
In Algol 60, a call `foo(a+f(b))` is translated by

- compiling the actual parameter expression `a+f(b)` into a small chunk of "stand-alone" object code, called a "**thunk**".
- passing the address of the thunk to the `foo` routine.

So, in the body of `foo`,

```
procedure foo (x: integer);
```

every reference to the formal `x` is translated as a subroutine call to the address `x`.

3.2.5 Intent

Classify a programmer's expectations when choosing formal parameters for a new procedure:

Direction:

- An **in** parameter supplies input to the procedure
- An **out** parameter receives output from the procedure
- An **in out** parameter supplies an input value that can be modified, with the modified value forming an output of the procedure.

Size: The actual parameters may range from *small* (1–2 words at most) to *large* (thousands of bytes).

Preferred Passing Modes: Time

Dir.		Val.	Ref.	Val/Res.
in:	small	√	—	√
	large	**x**	√	**x**
out:	small		—	√
	large		√	**x**
in out:	small		—	√
	large		√	**x**

√: nearly optimal, **x**: poor, —: acceptable

Preferred Passing Modes: Safety

Dir.	Val.	Ref.	Val/Res.
in	√	**x**	**x**
out		√	√
int out		√	√

Languages can aid in safety by forbidding modification of "in" parameters.

Parameter Passing in Pascal

- Default is call-by-value
- `VAR` parameters are passed by reference

Dilemma: how to pass array in

```
PROCEDURE FIND
    (A: ARRAY[1..1000] OF INTEGER;
     N: INTEGER;
     VAR POSITION: INTEGER);
```

- Pass by value is slow
- `VAR` is unsafe and may give callers false impression about in/out intention

---

Preferred Passing Modes in Pascal

Dir.	small	large
in	(value)	VAR
out	VAR	VAR
in out	VAR	VAR

Parameter Passing in C++

- Call-by-value
- C++has reference types (&) that hold a "reference" to an object
  - Passing a reference by value is functionally and lexically equivalent to call-by-reference

---

Parameter Passing in C++(cont.)

- Reference types can be modified with `const`
  - attempts to modify referenced object are illegal

Preferred Passing Modes in C++

Dir.	small	large
in	(value)	`const &`
out	&	&
in out	&	&

Parameter Passing in Ada

- Programmer labels formal parameters as `in`, `out`, or `in out`.
  - modification of `in` parameters is forbidden
- Compiler must pass integers and floating point numbers by value for `in`, "copy" for `out`, and value-result for `in out`.
- For other types, compiler may choose between above techniques and call-by-reference, whichever is faster.

Preferred Passing Modes in Ada

Dir.	small	large
in	in	in
out	out	out
in out	in out	in out

3.3 Scopes

$$\textbf{name} \xrightarrow{\text{scope}} \textbf{declaration} \xrightarrow{\text{activation}} \textbf{l-value} \xrightarrow{\text{state}} \textbf{r-value}$$

Scope rules explain how a use of a *name* is mapped back to its declaration.

3.3.1 Scope

The **scope** of a declaration is the range of source code within which that declaration is effective.

Scope rules can be *static* (lexical) or *dynamic*.

Static vs. Dynamic Scope

```
int n = 0;
void increment() { ++n; }

void printn() {  cout << n;  }

int main()
{
  int n = 0;
  increment();
  printn();
}
```

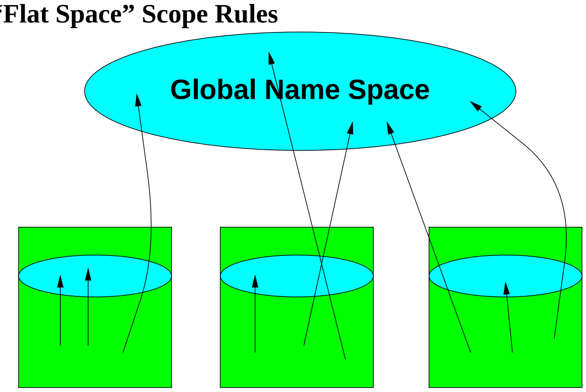
Output under static scope? Dynamic?

- Static scoping is most common.
- Dynamic is seen mainly in specialized, interpreted languages and macros.
- LISP featured dynamic scoping, but this is largely being replaced by static.

Static Scope

- Static scope rules largely work via "containment":
  The scope of a declaration extends to the end of the procedure/block/whatever that contains it.
- Syntactic structure provides barriers that divide a program's "namespace" into separate regions.
- A surprising amount of the history of HLL's is tied up in the evolution of scope rules.

"Flat Space" Scope Rules



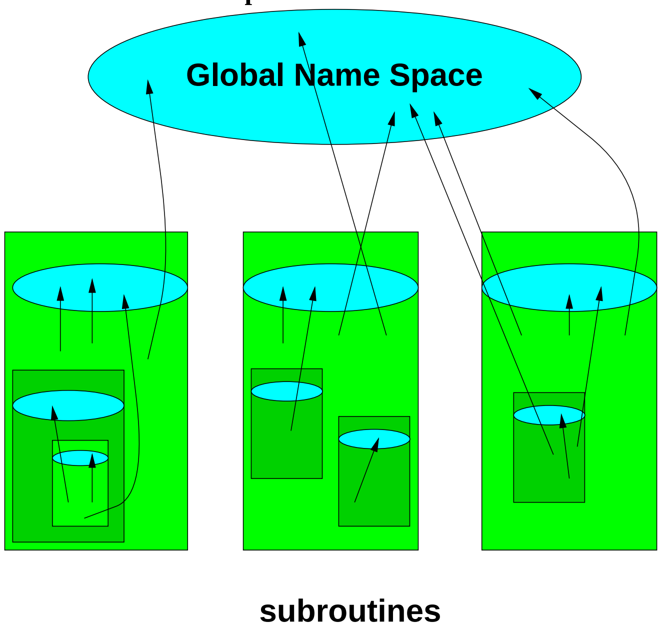
subroutines

Names are either global or local to a procedure: FORTRAN, COBOL, BASIC, ALGOL 60

FORTRAN scope example

```
SUBROUTINE IEXIT (ISTATE)
COMMON B(10), C
IF (ISTATE .NE. 2) GO TO 10
ITAIL = 3
```

“Block Structured” Scope Rules



In **block structured** HLLs, procedures can **nest** within each other. Each procedure has its own local names, and can access names of containers as well.

Pascal scope example

Pascal has very pure nesting rules:

- procedures can nest within one another
- the “main” program is treated as an outermost procedure, within which all others are nested
- any use of a name refers to the innermost nested declaration of that name
  - cannot reference declarations that are not local to the current routine or to one of the routines it is nested within
  - additionally, each name must be declared before being used
- a declaration of a name hides any outer declarations of the same name throughout the rest of this procedure and any later procedures nested within this one

```
program Compiler (input , output);
var i: integer;
  procedure scan;
    procedure getch;
      ... getch...
    end
  begin
    ... scan...
  end;

  procedure parse;
    procedure expr;
      var value: integer;
      procedure term;
        var value: integer;
        procedure factor;
          var value: integer;
        begin
          ... factor...
        end
      begin
        ... term...
      end
    begin
      ... expr...
    end
  begin
    ... parse...
  end;
begin
  ... main...
end.
```

```
program Compiler (input, output);
var i:  integer;
  procedure scan;
    procedure getch;
      ...getch...
    end
  begin
    ...scan...
  end;
  procedure parse;
    procedure expr;
      var value:  integer;
      procedure term;
        var value:  integer;
        procedure factor;
          var value:  integer;
        begin
          ...factor...
        end
      begin
        ...term...
      end
    begin
      ...expr...
    end
  begin
    ...parse...
  end;
begin
  ...main...
end.
```

The procedures calls in `parse` will mimic the structure of the grammar:

$$
\begin{array}{rcl}
\langle exp\rangle & ::= & \langle exp\rangle \texttt{+} \langle term\rangle \\
 & | & \langle exp\rangle \texttt{-} \langle term\rangle \\
 & | & \langle term\rangle \\
\langle term\rangle & ::= & \langle term\rangle * \langle factor\rangle \\
 & | & \langle term\rangle \texttt{/} \langle factor\rangle \\
 & | & \langle factor\rangle \\
\langle factor\rangle & ::= & \texttt{id}|\texttt{number} \\
 & | & \texttt{(} \langle exp\rangle \texttt{)}
\end{array}
$$

Note that the nesting rules support this nicely.

“Nesting is for the Birds†”

Nesting is meant as a way to impose control on the namespace.

- Why is control needed?
- What happens if `factor` and `getch` need to share a symbol?
  - The shared symbol must be promoted to the innermost common container.

    In many cases, this forces symbols to be global.

†Clarke et al., *Nesting in Ada is for the Birds*

- Studies showed that, in practice, nesting in production code seldom went more that a few levels deep.

C scope example

C’s rules fall between the flat structure of early languages and the full nesting of Pascal.

- Procedures (functions) cannot nest within one another
- Statement lists {`...`} can nest within one another.
- the “main” program is just another procedure, at the same level as the others.
- All functions are declared at “file” scope - the declaration remains in effect to the end of the containing file.
- Other names may be declared at file scope, within a function, or within a statement list.
  - scope of that declaration extends to the end of the innermost containing statement list, function, or file.
  - hides any outer declarations of the same name
- any use of a name refers to the innermost nested declaration of that name
- Some names may be used before/without declaration. These are implicitly declared at file scope.

```
int i;
  char getch()
  {
    ...
  }
  void scan()
  {
    ...
  }
void expr();
void term();
  :
  :
  void factor()
  {
    int value;
    ...
      {
        int value;
        int x;
        ...
      }
      {
        int x, y;
        ...
      }
    ...
  }
  void term()
  {
    ...
  }
    :
    :
  .
  :
  void expr()
  {
    ...
  }
  void parse()
  {
    ...
  }
  int main()
  {
    ...
  }
```

C's nested statement lists

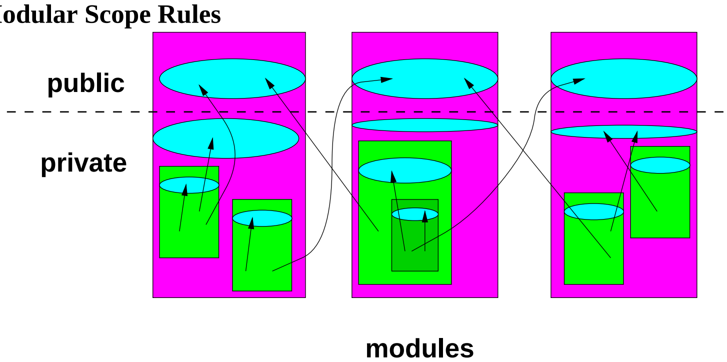
- provide some protection against unintended alteration of common variable names,
- but seem mainly oriented toward allowing sharing of memory by variables with disjoint lifetimes

---

Scopes: more to come

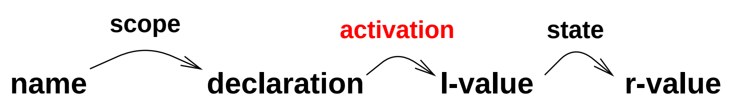
As we will see in the coming weeks, later languages refine the scope rules still further.

Modular Scope Rules



- Procedures are grouped into **modules**.
- Each module has **public** and **private** namespaces.
- Only procedures belonging to a module may access its private names.

3.4 Activation Records



A procedure is **activated** when a call to it begins.

The **activation** ends when it returns to the caller.

In early FORTRAN (no recursion), procedure calls could be implemented by associating with each procedure a hidden variable to hold the return address.

- Caller would
  - place its current PC (program counter) in that variable,
  - then jump to start of procedure code
- Called routine would
  - execute its code body variable,
  - then jump to address stored in that variable

This does not work with recursion, because the same procedure may have many simultaneous activations.

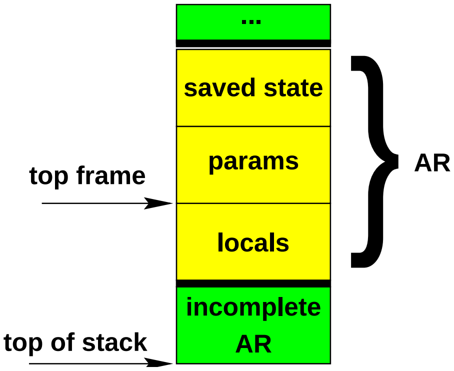
In general, call-and-return follow a LIFO discipline.

- The data required for each activation is collected into an **activation record** or **frame**.
- Activation records are collected into an **activation stack** or **run-time stack**.

3.4.1 Anatomy of an Activation

Structure of AR's is machine/compiler dependent.

A typical one is



Typical contents of saved state includes

- return address
- contents of critical registers
- access link or display level (later)

Parameters include

- actual parameter values
- space for function return value

Activating a procedure: Caller

For a call `foo(a, b+c, d);`

1. Push state information, including return address, `TF`, and `TOS`
   - text calls saved value of `TF` a **control link**.
2. Evaluate each actual expression, push result/address onto stack
3. jump to `foo`'s starting address . . .
4. Copy function return value (if any)
5. Restore state information from below `TF`

Activating a procedure: Called

For a call `foo(a, b+c, d);`

a. Set `TF` to `TOS`

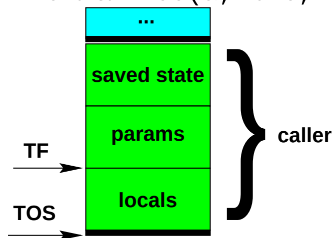
b. Push enough bytes to hold local variables

c. Execute code body

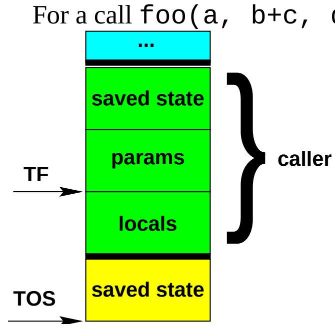
- parameters accessed as negative offsets from `TF`
- locals accessed as positive offsets from `TF`

d. Jump to return address in saved state area.

For a call `foo(a, b+c, d);`

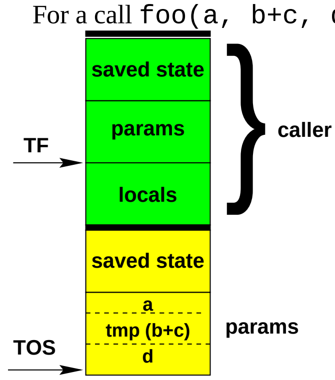


For a call `foo(a, b+c, d);`



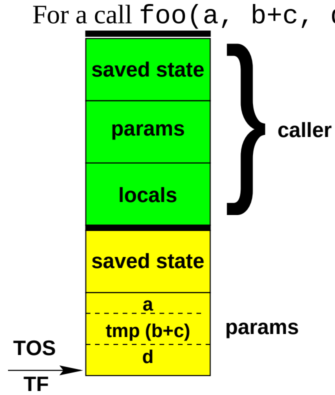
1. Push state information, including return address, `TF`, and `TOS`

For a call `foo(a, b+c, d);`



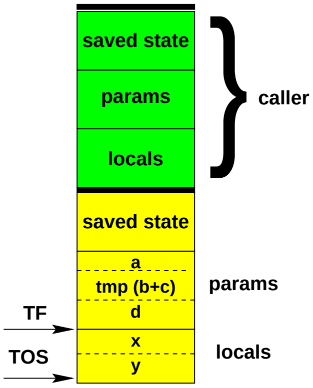
2. Evaluate each actual expression, push result/address onto stack
3. jump to `foo`'s starting address

For a call `foo(a, b+c, d);`



a. Set `TF` to `TOS`

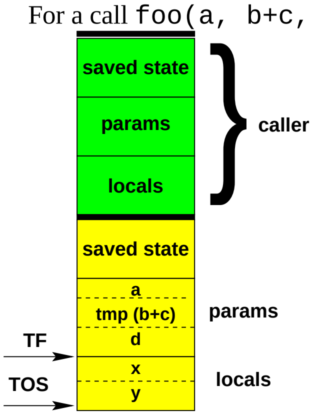
For a call `foo(a, b+c, d);`



b. Push enough bytes to hold local variables

c. Execute code body

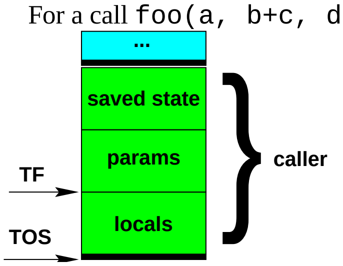
For a call `foo(a, b+c, d);`



d. Jump to return address in saved state area.

4. Copy function return value (if any)

For a call `foo(a, b+c, d);`



5. Restore state information from below `TF`

3.5 Case study: C

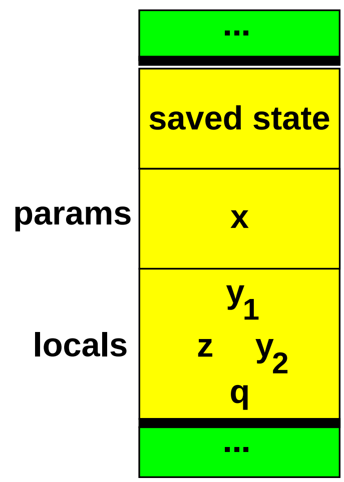
- Any statement list can have local variables.
- The scope of each local declaration ends with the enclosing { }.

Local variables: C

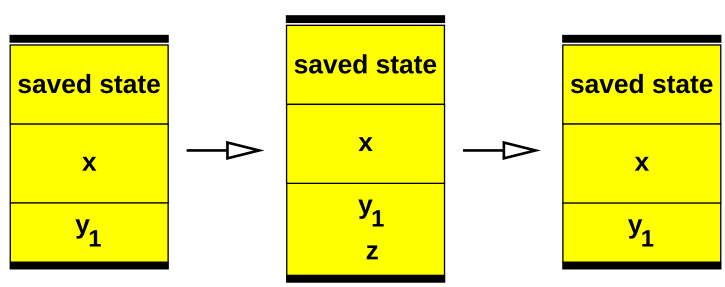
```
int foo (int x) {
  int y;
  y = 0;
  if (x < 0) {
    int z;
    z = x;
    y = foo(z);
  }
  else {
    int y, q;
    y = x + 1;
    q = x - 1;
    x = y*q;
  }
  return x+y;
}
```

- As compiler sees parameter & local declarations, assigns an *offset* to each declaration. Offset gives position of variable in AR relative to TF.
- Because `z`'s scope is disjoint from that of y_2 and `q`, it can share storage with one of these variables.

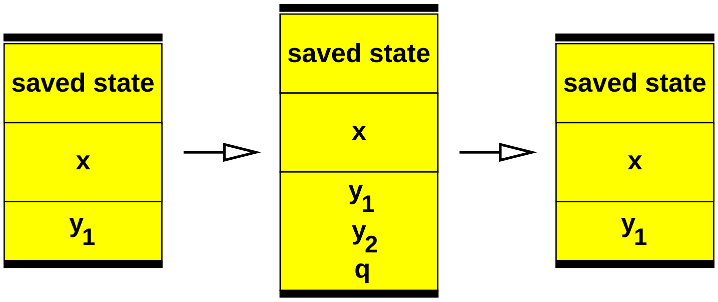
Most compilers will allocate the local storage all at once:



A few will place code in each { } to push and pop locals for each statement list:
`then`:

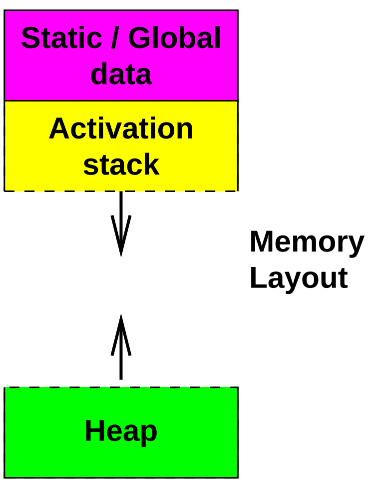


`else`:



Access to nonlocals: C
That which is not local must be global (static).

- Globals are easy to access because they reside at a fixed address.



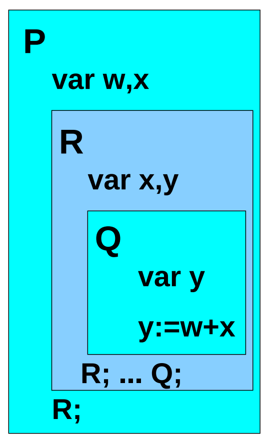
Procedure Parameters: C
C allows (pointers to) functions to be passed as parameters to other functions.

- No big deal — because C functions don't nest
- We'll see that this is more of a problem for nesting languages

3.6 Case study: Pascal

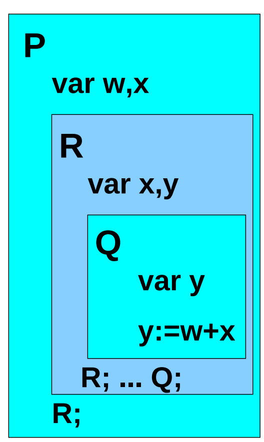
Pascal nests procedures, but not statement lists.
The combination of nesting and recursion complicates AR's.

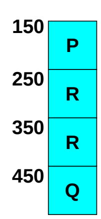
```
procedure P;
  var w, x: integer;

  procedure R;
    var x, y: integer;

    procedure Q;
      var y: integer;
    begin
      y := x + w;
    end
  begin
    ... R; ... Q; ...
  end
begin
    ... R; ...
end
```



- `w`, `x` are not local to `Q`
- neither are they global (as in C)
- located in another activation's AR

Suppose each AR takes 100 bytes.
Find `x` and `w`.





`x` is 100 bytes away.
`w` is 100∗ # of activ. of `R` bytes away

- Scope rules supply only part of the answer

  - they determine which *procedure* holds the object

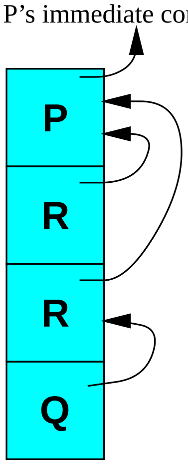
- We need the most recent activation (MRA) of that procedure

Two approaches to finding most recent activations:

- access links
- displays

3.6.1 Access Link

An **access link** in an AR for procedure P is a pointer to the MRA of P's immediate container.



Accessing Data: Access Links
For `Q` To find a non-local variable at offset x in `P`:

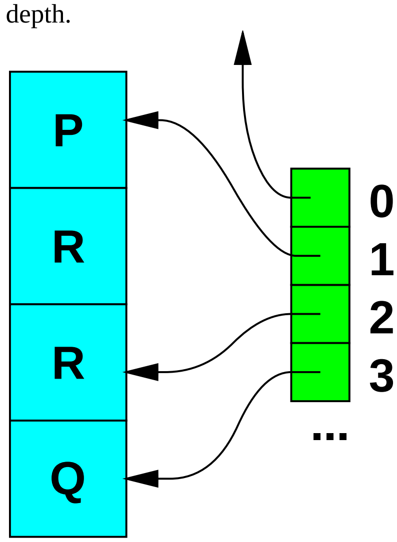
1. Let $\Delta = \text{depth}(Q) - \text{depth}(P)$
2. Follow Δ access links back to get the MRA of `P`
3. Add x to the MRA address

Constructing Access Links
Adding access links to AR's requires slight modification to calling sequence.
When saving state info on stack,

- If caller and called are at same nesting depth, copy caller's access link into the new AR.
- If called routine is deeper than caller, it must be immediately nested within the caller. New access link must point to caller's AR.
- If caller is deeper than called routine (recursion), then follow Δ access links to MRA of called routine. Copy its access link into new AR.

3.6.2 Displays

A display is a global array of pointers to MRA's, indexed by nesting depth.



Accessing Data: Displays
For `Q` To find a non-local variable at offset x in `P`:

1. Let $d = \text{depth}(P)$
2. Add x to `display[`d`]`

Constructing Displays
When saving state info on stack,

- Let $d =$ depth of called routine
- Save `display[`d`]` in new AR
- After pushing all parameters, put `TOS` in `display[`d`]`.

When returning from a routine at depth d, restore `display[`d`]` from the saved state.

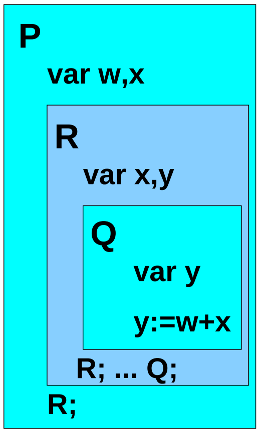
Unlike access links, displays allow constant-time access to data.

- But since most programmers don't nest deeply, it's not a significant difference.

Procedure Parameters: Pascal

Like C, Pascal allows procedures to be passed as parameters to other procedures.

- What to do with non-locals then?



Suppose `P` called `Q` directly.
There is no activation of `R`. What to do with `Q`'s reference to `x`?

- not enough access links to follow
- display contains garbage at `R`'s depth

Later languages would address this by

Ada: forbidding procedure parameters — other constructs were invented to achieve similar results

Modula 2: procedure parameters are allowed, but the actual procedure must not be nested within another procedure.

C nests statement lists, not functions.
Pascal nests functions, not statement lists.
Ada nests both, and also has variable-size data types.