Run Time Storage Organization

ASU Textbook Chapter 7.1–7.5, and 7.7–7.9

Tsan-sheng Hsu

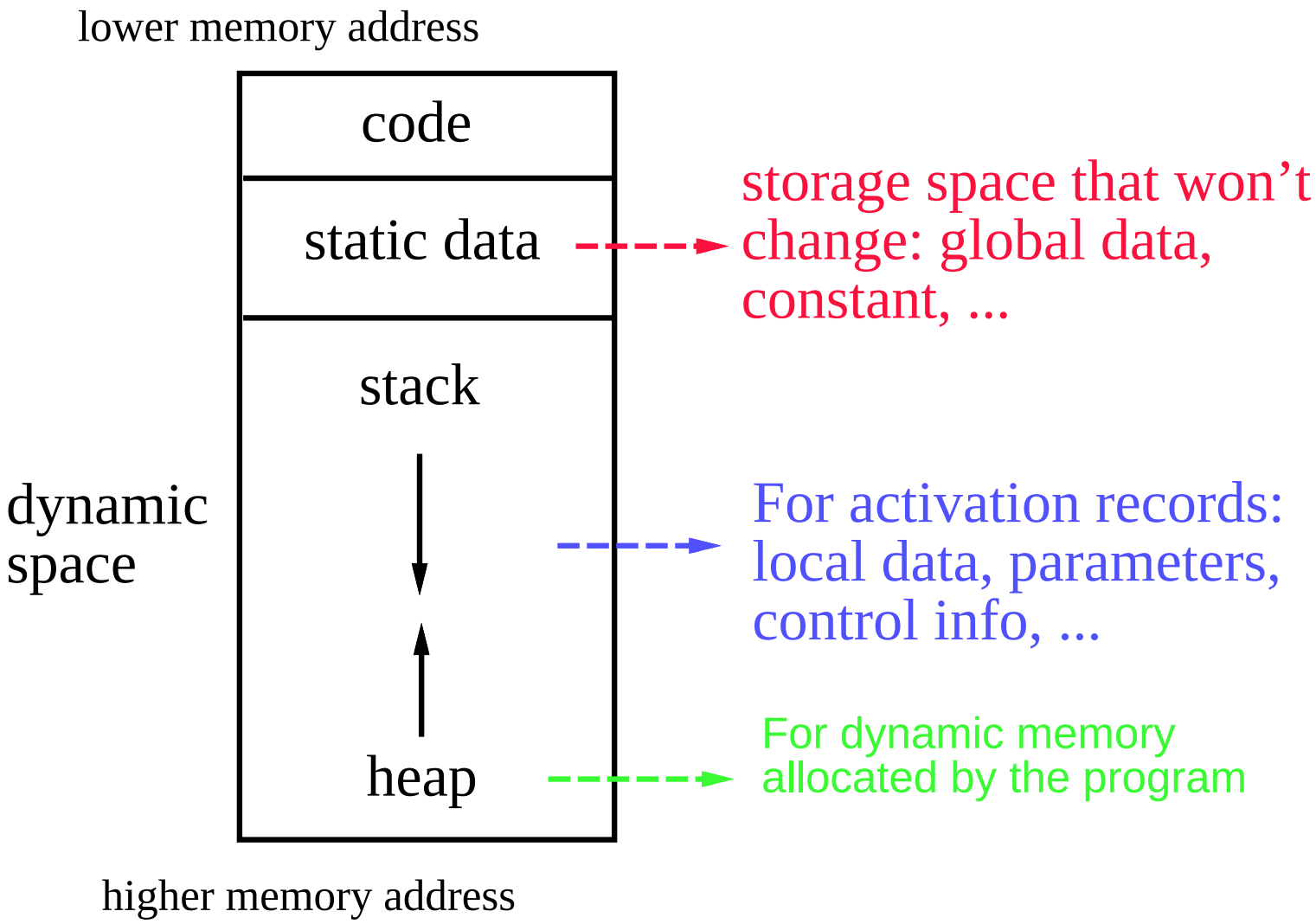
tshsu@iis.sinica.edu.tw

`http://www.iis.sinica.edu.tw/~tshsu`

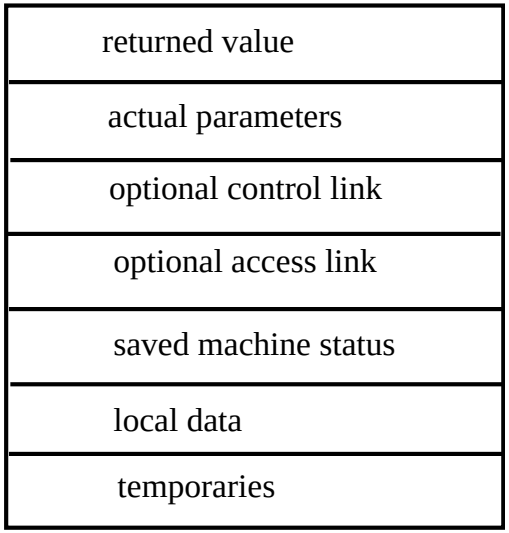
Definitions

- During the execution of a program, the same name in the source can denote different data objects in the computer.
- The allocation and deallocation of data objects is managed by the **run-time support package**.
- Terminologies:
  - name $\rightarrow$ storage space: the mapping of a name to a storage space is called **environment**.
  - storage space $\rightarrow$ value: the current value of a storage space is called its **state.**
  - The association of a name to a storage location is called a **binding.**
- Each execution of a procedure is called an **activation**.
  - If it is a recursive procedure, then several of its activations may exist at the same time.
  - **Life time:** the time between the first and last steps in a procedure.
  - A recursive procedure needs not to call itself directly.

General run time storage layout



Activation record



- **Activation record: data about an execution of a procedure.**
  - **Parameters:**
    - ***Formal parameters:*** *the declaration of parameters.*
    - ***Actual parameters:*** *the values of parameters for this activation.*
  - **Links:**
    - ***Access (or static) link:*** *a pointer to places of non-local data,*
    - ***Control (or dynamic) link:*** *a pointer to the activation record of the caller.*

Static storage allocation (1/3)

- **There are two different approaches for run time storage allocation.**
  - **Static allocation.**
  - **Dynamic allocation.**
- **Static allocation: uses no stack and heap.**
  - **A.R. in static data area, one per procedure.**
  - **Names bounds to locations at compiler time.**
  - **Every time a procedure is called, its names refer to the same pre-assigned location.**
  - **Disadvantages:**
    - ▷ *No recursion.*
    - ▷ *Waste lots of space when inactive.*
    - ▷ *No dynamic allocation.*
  - **Advantage:**
    - ▷ *No stack manipulation or indirect access to names, i.e., faster in accessing variables.*
    - ▷ *Values are retained from one procedure call to the next.*
      For example: static variables in C.

Static storage allocation (2/3)

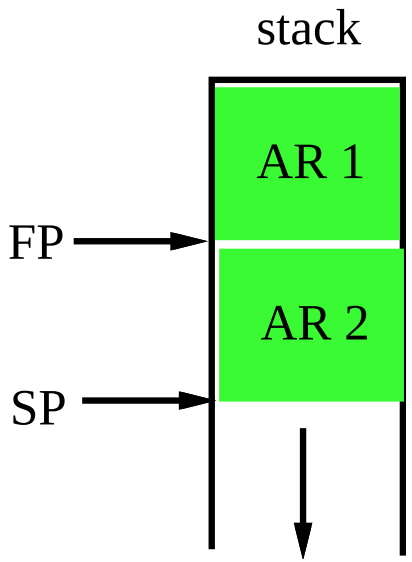
- **On procedure calls:**
  - **the calling procedure:**
    - *First evaluate arguments.*
    - *Copies arguments into parameter space in the A.R. of called procedure.*
      Convention: call that which is passed to a procedure **arguments** *from the calling side, and* **parameters** *from the called side.*
    - *May save some registers in its own A.R.*
    - *Jump and link: jump to the first instruction of called procedure and put address of next instruction (return address) into register RA (the return address register).*
  - **the called procedure:**
    - *Copies return address from RA into its A.R.'s return address field.*
    - *May save some registers.*
    - *May initialize local data.*

Static storage allocation (3/3)

- **On procedure returns,**
  - **the called procedure:**
    - ▷ *Restores values of saved registers.*
    - ▷ *Jump to address in the return address field.*
  - **the calling procedure:**
    - ▷ *May restore some registers.*
    - ▷ *If the called procedure was actually a function, put return value in an appropriate place.*

Dynamic storage allocation for STACK (1/3)

- **Stack allocation:**
  - **Each time a procedure is called, a new A.R. is pushed onto the stack.**
  - **A.R. is popped when procedure returns.**
  - **A register (SP for stack pointer) points to top of stack.**
  - **A register (FP for frame pointer) points to start of current A.R.**



Dynamic storage allocation for STACK (2/3)

- **On procedure calls,**
  - **the calling procedure:**
    - ▷ *May save some registers (in its own A.R.).*
    - ▷ *May set optional access link (push it onto stack).*
    - ▷ *Pushes parameters onto stack.*
    - ▷ *Jump and Link: jump to the first instruction of called procedure and put address of next instruction into register RA.*
  - **the called procedure:**
    - ▷ *Pushes return address in RA.*
    - ▷ *Pushes old FP (control link).*
    - ▷ *Sets new FP to old SP.*
    - ▷ *Sets new SP to be old SP + (size of parameters) + (size of RA) + (size of FP). (These sizes are computed at compile time.)*
    - ▷ *May save some registers.*
    - ▷ *Push local data (maybe push actual data if initialized or maybe just their sizes from SP)*

Dynamic storage allocation for STACK (3/3)

- **On procedure returns,**
  - **the called procedure:**
    - ***Restore values of saved registers if needed.***
    - ***Loads return address into special register RA.***
    - ***Restores SP (SP := FP).***
    - ***restore FP (FP := saved FP).***
    - ***return.***
  - **the calling procedure:**
    - ***May restore some registers.***
    - ***If it was in fact a function that was called, put return value into an appropriate place.***

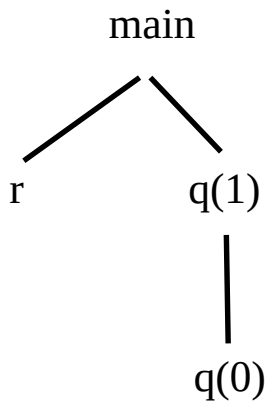
Activation tree

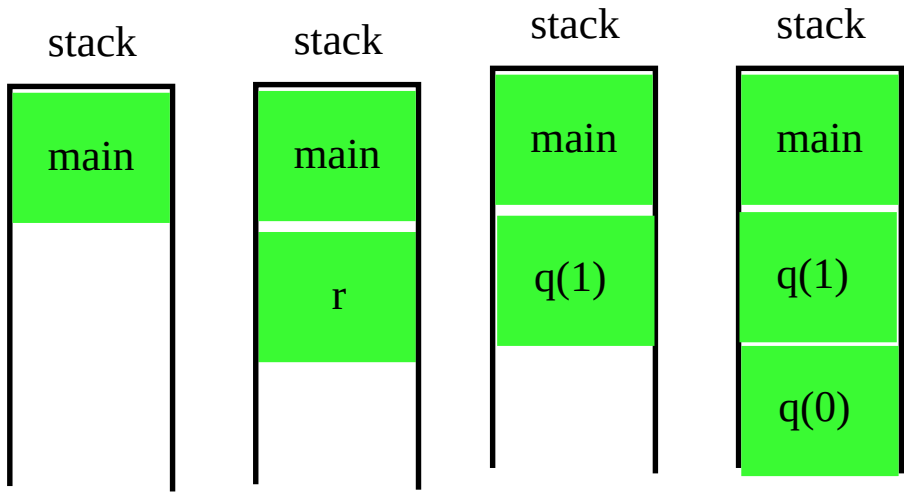
- **Use a tree structure to record the changing of the activation records.**
- **Example:**

```
main{
    r();
    q(1);
}

r{
...}

q(int i)
{
if(i>0) then q(i-1)
}
```





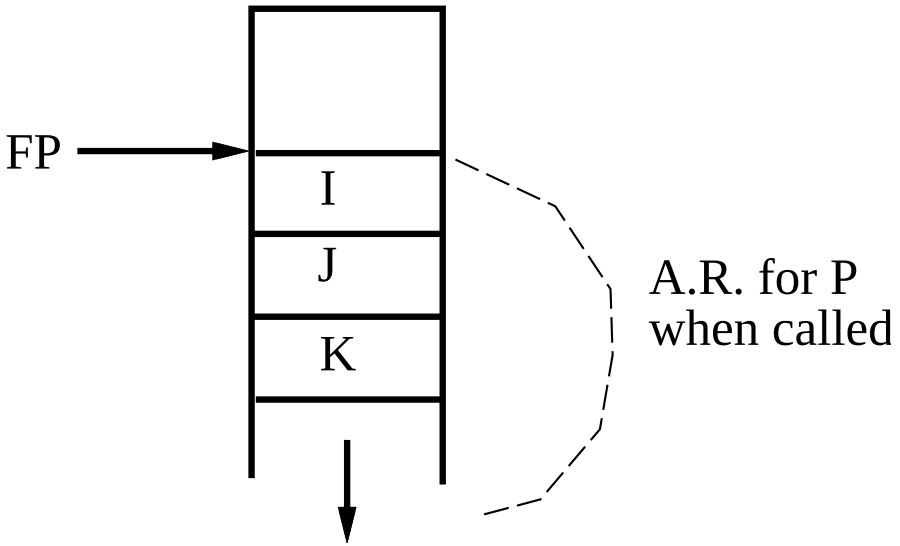
Dynamic storage allocation for HEAP

- **Storages requested from programmers during execution:**
  - **Example:**
    - ***PASCAL:*** new **and** $free$**.**
    - ***C:*** $malloc$ **and** $free$**.**
  - **Issues:**
    - ***Garbage collection.***
    - ***Segmentation.***
    - ***Dangling reference.***
- **More or less O.S. issues.**

Run time variable accesses

- **Local variables:**
  - **Stored in the activation record of declaring procedure.**
  - **Access by offset from the frame pointer (FP).**
- **Example:**

```
P()
{
int I,J,K;
...
}
```



  - **Address of J is $FP + 1 * sizeof(int)$.**
  - **Offset is $1 * sizeof(int)$.**
  - **Actual address is only known at run time.**
  - **Offset for each local variable is known at compile time.**

Accessing global and non-local variables

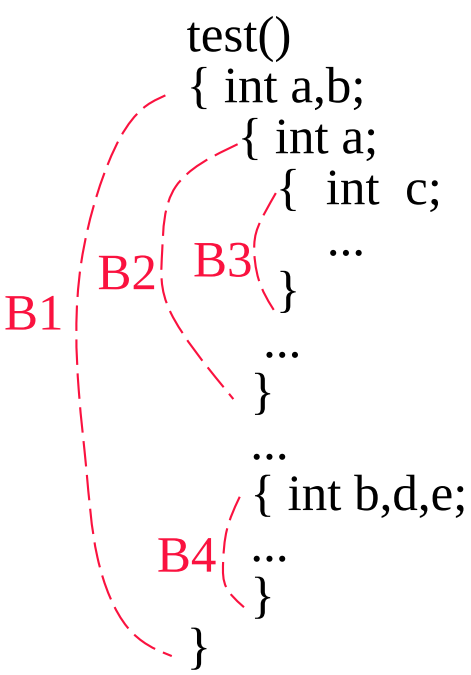
- **Global variables stored in static data area:**
  - **Access by using names.**
  - **Addresses known at compile time.**
- **Two scoping rules for accessing non-local data.**
  - **Lexical or static scoping.**
    - ▷ ***PASCAL, C and FORTRAN.***
    - ▷ ***The correct address of a non-local name can be determined at compile time by checking the syntax.***
  - **Dynamic scoping.**
    - ▷ ***LISP.***
    - ▷ ***A use of a non-local variable corresponds to the declaration in the "most recently called, still active" procedure.***
    - ▷ ***The question of which non-local variable to use cannot be determined at compile time. It can only be determined at run-time.***

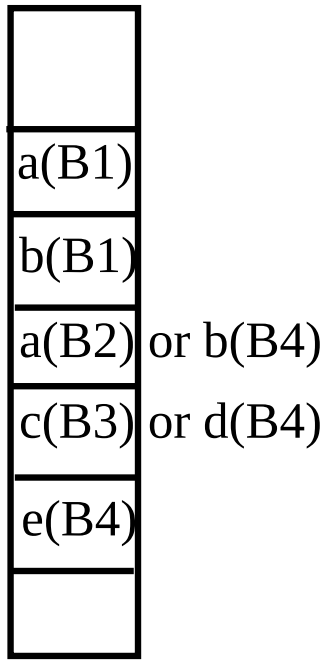
Lexical scoping with blocked structures

- **Block: a statement containing its own local data declaration.**
- **Scoping is given by the following so called most closely nested rule.**
  - **The scope of a declaration in a block B includes B itself.**
  - **If x is used in B, but not declared in B, then we refer to x in a block B', where**
    - ***B' has a declaration x, and***
    - ***B' is more closely nested around B than any other block with a declaration of x.***

Lexical scoping without nested procedures

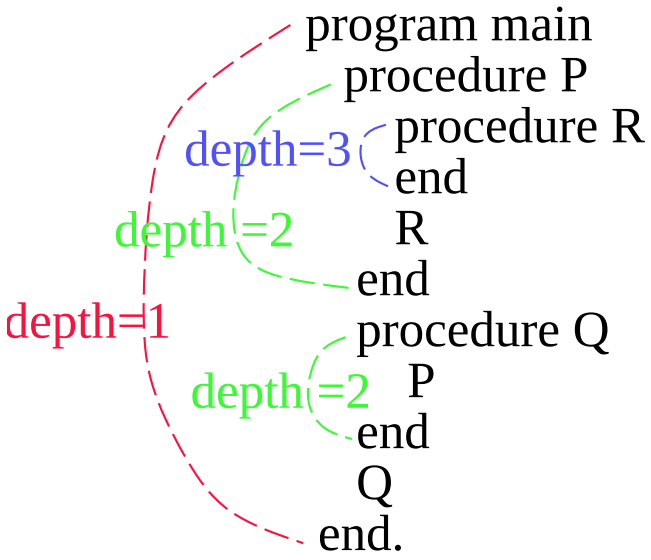
- **Nested procedure: a procedure that can be declared within another procedure.**
- **If a language does not allow nested procedures, then**
  - **A variable is either global, or is local to the procedure containing it.**
  - **At runtime, all the variables declared (including those in blocks) in a procedure are stored in its A.R., with possible overlapping.**
  - **During compiling, proper offset for each local data is calculated using information known from the block structure.**

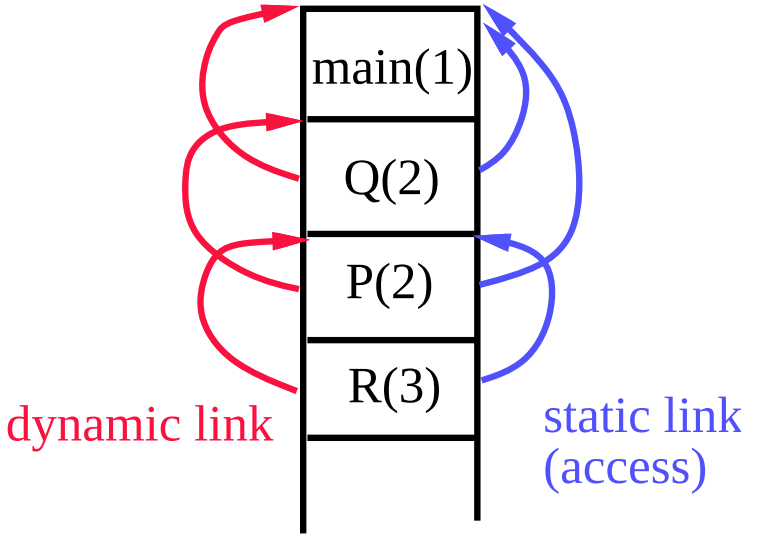




Lexical scoping with nested procedures

- **Nesting depth:**
  - **depth of main program = 1.**
  - **add 1 to depth each time entering a nested procedure.**
  - **substrate 1 from depth each time existing from a nested procedure.**
  - **Each variable is associated with a nesting depth.**
  - **Assume in a depth-h procedure, we access a variable at depth k, then**
    - $h \geq k$.
    - ***follow the access (static) link $h - k$ times, and then use the offset information to find the address.***





Algorithm for setting the links

- **The dynamic link is set to point to the A.R. of the calling procedure.**
- **How to properly set the static link at compile time.**
  - **Procedure p at depth n_p calls procedure x at depth n_x:**
  - **If $n_p < n_x$, then x is enclosed in p.**
    - ***Same with setting the dynamic link.***
  - **If $n_p \geq n_x$, then it is either a recursive call or calling a previously declared procedure.**
    - ***Observation: go up the access link once, decrease the depth by 1.***
    - ***Hence, the access link of x is the access link of p going up $n_p - n_x + 1$ times.***

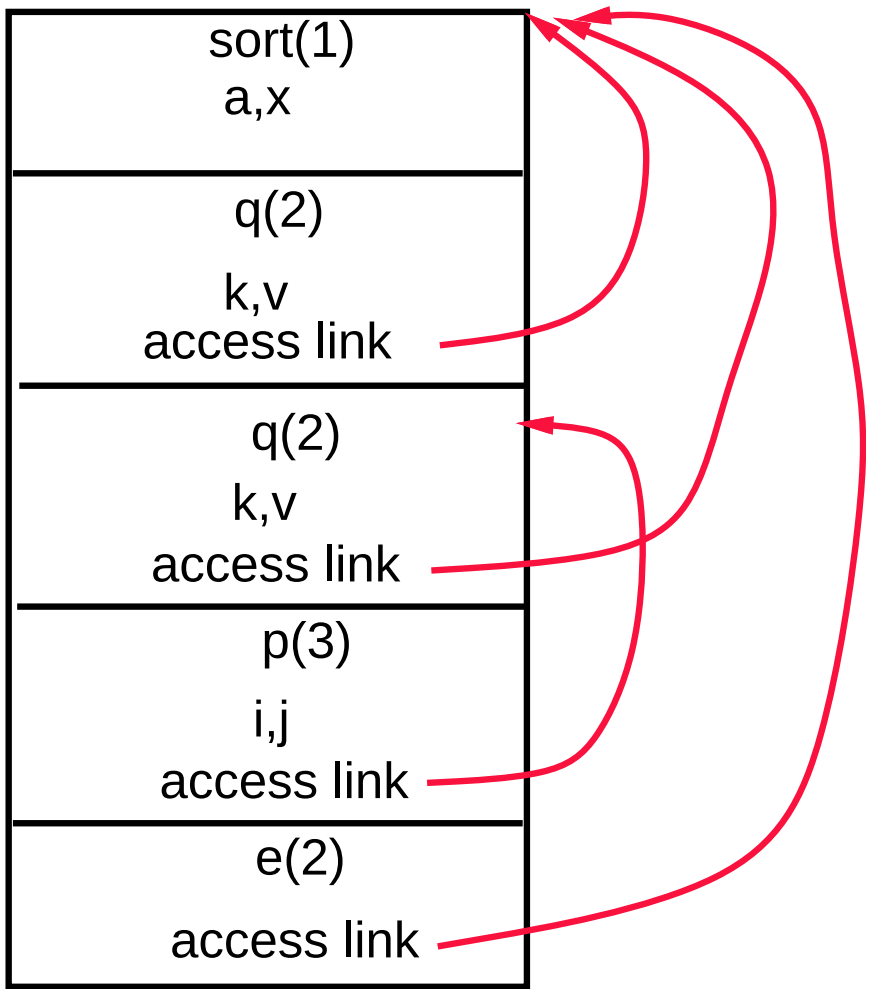
Example

```
Program sort
   var a: array[0..10] of int;
       x: int;
   procedure r
   var i: int;
   begin ...  r
   end

   procedure e(i,j)
   begin ... e
      a[i] <-> a[j]
   end

   procedure q
      var k,v: int;
      procedure p
      var i,j;
      begin ... p
        call e
      end
   begin ... q
   end

begin ... sort
   call q
end
```



Accessing non-local data using DISPLAY

- **Idea:**
  - **Maintain a global array called DISPLAY**
    - ▷ *Using registers if available.*
    - ▷ *Otherwise, in static data area.*
  - **When procedure P at nesting depth k is called,**
    - ▷ *DISPLAY[1], . . . , DISPLAY[k-1] hold pointers to the A.R.'s of the most recent activation of the $k-1$ procedures that lexically enclose P.*
    - ▷ *DISPLAY[k] holds pointer to P's A.R.*
    - ▷ *To access a variable with declaration at depth x, use DISPLAY[x] to get to the A.R. that holds x, then use the usual offset to get x itself.*
    - ▷ *Size of DISPLAY equals maximum nesting depth of procedures.*
  - **Bad for languages allow recursions.**
- **To maintain the DISPLAY**
  - **When a procedure at nesting depth k is called**
    - ▷ *Save the current value of DISPLAY[k] in the save-display field of the new A.R.*
    - ▷ *Set DISPLAY[k] to point to the new A.R. (e.g., to its save-display field).*
  - **When the procedure returns, restore DISPLAY[k] using the value saved in save-display field.**

Access links v.s. DISPLAY

- **Time and space trade-off.**
  - **Access links require more time (at run time) to access non-local data. Especially when non-local data are many nesting levels away.**
  - **DISPLAY probably require more space (at run time)**
  - **Code generated using DISPLAY is simpler.**

Dynamic scoping

- **Dynamic scoping: a use of a non-local variable corresponds to the declaration in the "most recently called, still active" procedure.**
- **The question of which non-local variable to use cannot be determine at compile time.**
- **It can only be determined at run time.**
- **Must save symbol tables at run time.**
- **Two ways to implement access non-locals under dynamic scoping.**
  - **Deep access**
  - **Shallow access**

Deep access

- **Def: given a use of a non-local variable, use control links to search back in the stack for the most recent A.R. that contains space for that variable.**
  - **Note: this requires that it be possible to tell which variables are stored in each A.R.**
  - **Need to use the symbol tables at run time.**
- **Example:**

```
program main
   procedure test
   var x : int;
   begin
       x := 30
       call DeclaresX
       call UsesX
   end
   procedure DeclaresX
      var x: int;
   begin
      x := 100
      call UsesX
   end
   procedure UsesX
   begin
     write(x)
   end
begin
   call test
end
```

- **Which x is it in the procedure UsesX?**
- **If we were to use static scoping, this is not a legal statement; No enclosing scope declares x.**

Shallow access

- **Idea:**
  - Maintain a current list of variables.
  - Space is allocated (in registers or in the static data area) for every possible variable name that is in the program (i.e., one space for variable x even if there are several declarations of x in different procedures).
  - For every reference to x, the generated code refers to the same location.
- **When a procedure is called,**
  - it saves, in its own A.R., the current values of all of the variables that it declares itself (i.e., if it declares x and y, then it saves the values of x and y that are currently in the space for x and y).
  - it restores those values when it finishes.
- **Comparison of deep and shallow access:**
  - Shallow access allows fast access to non-locals, but there is overhead on procedure entry and exit proportional to the number of local variables.

Parameter passing

- **Terminology:**
  - **procedure declaration:**
    - ▷ *parameters, formal parameters, or formals.*
  - **procedure call:**
    - ▷ *arguments, actual parameters, or actuals.*
- **The value of a variable:**
  - R**-value: the current value of the variable.**
    - ▷ *right value*
    - ▷ *on the right side of assignment*
  - L**-value: the location/address of the variable.**
    - ▷ *left value*
    - ▷ *on the left side of assignment*
  - **Example:** $x := y$
- **Four different modes for parameter passing**
  - **call by value**
  - **call by reference**
  - **call by value result (copy-restore)**
  - **call by name**

Call by value

- **Usage:**
  - Used by PASCAL if you use non-var parameters.
  - Used by C++ if you use non-& parameters.
  - The only thing used in C.
- **Idea:**
  - calling procedure copies the r-values of the arguments into the called procedure's A.R.
- **Effect:**
  - Changing a formal parameter (in the called procedure) has no effect on the corresponding actual. However, if the formal is a pointer, then changing the thing pointed to does have an effect that can be seen in the calling procedure.
- **Example:**

```
void f(int *p)
{  *p = 5;
   p = NULL;
}
```

```
main()
{int *q = malloc(sizeof(int));
*q=0;
f(q);
}
```

  - In main, q will not be affected by the call of f.
  - That is, it will not be NULL after the call.
  - However, the value pointed to by q will be changed from 0 to 5.

Call by reference (1/2)

- **Usage:**
  - **Used by PASCAL for var parameters.**
  - **Used by C++ if you use & parameters.**
  - **FORTRAN.**
- **Idea:**
  - **Calling procedure copies the l-values of the arguments into the called procedure's A.R. as follows:**
    - ▷ ***If an arg has an address then that is what is passed.***
    - ▷ ***If an arg is an expression that does not have an l-value (e.g., $a + 6$), then evaluate the arg and store the value in a temporary address and pass that address.***
- **Effect:**
  - **Changing a formal parameter (in the called procedure) does affect the corresponding actual.**
  - **Side effects.**

Call by reference (2/2)

- **Example:**

```
FORTAN quirk (using C++ syntax)
void mistake(int & x)
{x = x+1;}
main()
{mistake(1);
cout<<1;
}
```

  - **In C++, you would get a warning from the compiler because x is a reference parameter that is modified, and the corresponding actual parameter is a literal.**
  - **The output of the program is 1.**
  - **However, in FORTRAN, you would get no warning, and the output may be 2. This happens when FORTRAN compiler stores 1 as a constant at some address and uses that address for all the literal "1" in the program.**
  - **In particular, that address is passed when "mistake" is called, and is also used to fetch the value to be written by "count". Since "mistake" increments its parameter, that address hold the value 2 when cout is executed.**

Call by value-result

- **Usage: FORTRAN IV and ADA.**
- **Idea:**
  - **Value, not address, is passed into called procedure's A.R. when called procedure ends the final value is copied back into the argument's address.**
  - **Equivalent to call by reference except when there is aliasing.**
    - *"Equivalent" in the sense the program produces the same results, NOT the same code will be generated.*
    - *Aliasing : two expressions that have the same l-value are called aliases. That is, they access the same location from different places.*
    - *Aliasing happens*
      through pointer manipulation;
      call-by-reference with an arg that can also accesses by the called procedure directly, e.g., global var;
      call-by-reference with the same expression as an argument twice; e.g., $test(x, y, x)$

Call by name (1/2)

- **Usage: Algol.**
- **Idea: (not the way it is actually implemented.)**
  - **Procedure body is substituted for the call in the calling procedure.**
  - **Each occurrence of a parameter in the called procedure is replaced with the corresponding argument, i.e., the TEXT of the parameter, not its value.**
  - **Similar to macro substitution.**

Call by name (2/2)

- **Example:**

```
void init(int x, int y)
{ for(int k = 0; k <10; k++)
    {  x++; y = 0;}
}
```

```
main()
{ int j;
  int A[10];
  j = -1;
  init(j,A[j]);
}
```

- **Conceptual result of substitution:**

```
main()
{ int j;
  int A[10];
  j = -1;
  for(int k = 0; k<10; k++)
  { j++;       /* actual j for formal x */
    A[j] = 0; /* actual A[j] for formal y */
  }
}
```

- **Call-by-name is not really implemented like macro expansion. Recursion would be impossible, for example using this approach.**

How to implement call-by-name?

- Instead of passing values or addresses as arguments, a function (or the address of a function) is passed for each argument.
- These functions are called **thunks.**, i.e., a small piece of code.
- Each thunk knows how to determine the address of the corresponding argument.
  - Thunk for j: find address of j.
  - Thunk for $A[j]$: evaluate j and index into the array A; find the address of the appropriate cell.
- Each time a parameter is used, the thunk is called, then the address return by the thunk is used.
  - $y = 0$: use return value of thunk of y as l-value.
  - $x = x + 1$: use return value of thunk x both as l-value and to get r-value.
  - For the example above, call-by-reference would execute $A[1] = 0$ ten times, while call by name initializes the whole array.
- Note: call-by-name is generally considered a bad idea, because it is hard to know what a function is doing – it may require looking at all calls to figure this out.

Advantage of call-by-value

- **Consider not passing pointers.**
- **No aliasing.**
- **Arguments unchanged by procedure call.**
- **Easier for static optimization analysis for both programmers and the complier.**
- **Example:**

```
x = 0;
Y(x);   /* call-by-value */
z = x+1; /* can be replaced by z=1 for optimization */
```

- **Compared with call-by-reference, code in called function is faster because no need for redirecting pointers.**

Advantage of call-by-reference

- **Efficiency in passing large objects.**
- **Only need to copy addresses.**

Advantage of call-by-value-result

- **More efficient than call-by-value for small objects.**
- **If there is no aliasing, can implement call-by-value-result using call-by-reference for large objects.**

Advantage of call-by-name

- **More efficient when passing parameters that are never used.**
- **Example:**

```
P(Ackerman(5),0,3)
/* Ackerman's function takes enormous time to compute */

function P(int a, int b, int c)
{ if(odd(c)){
     return(a)
  }else{ return(b)  }
}
```

- **Note: if the condition is false, then using call-by-name, it is never necessary to evaluate the first actual at all.**
- **This saves lots of time because evaluating a takes a long time.**