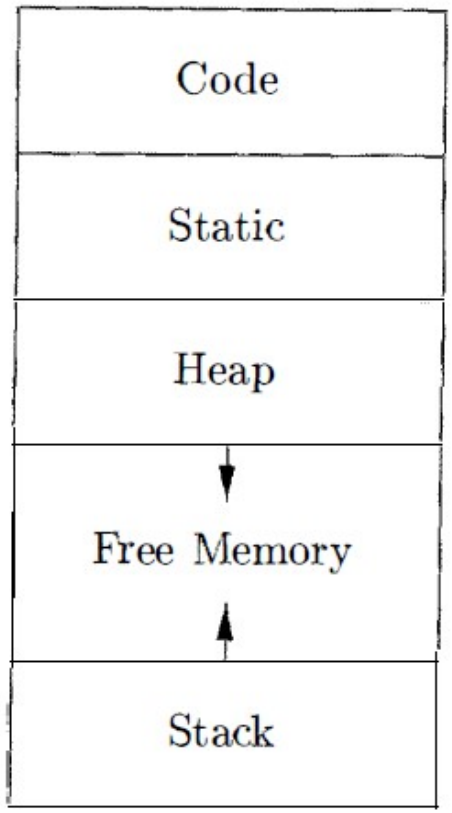
CSE504 Compiler Design
Run-Time Environments

YoungMin Kwon

Overview

- Learn the relationship between names and data objects

Storage Organization



- Typical Run-time memory
- Stack Storage
  - Variables local to a procedure are usually allocated on a stack.
- Heap Storage
  - Data that may outlive a procedure are usually allocated on a heap.

Storage Allocation Strategies

- Static allocation
  - Names are bound to storage as the program is compiled.
    - E.g. Our simple compiler.
  - Recursive procedures are restricted
  - No dynamic data structure
- Heap allocation
  - When the values of local variables must be retained.

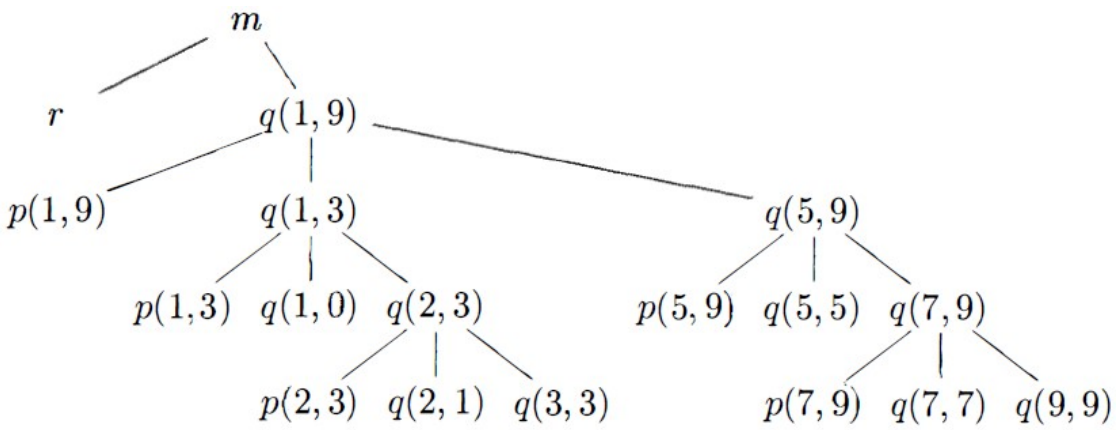
```
main()
{
    int *p;
    p = dangle();
}
```

```
int *dangle ()
{
    int i = 23;
    return &i;
}
```

Stack Allocation (Activation Trees)

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

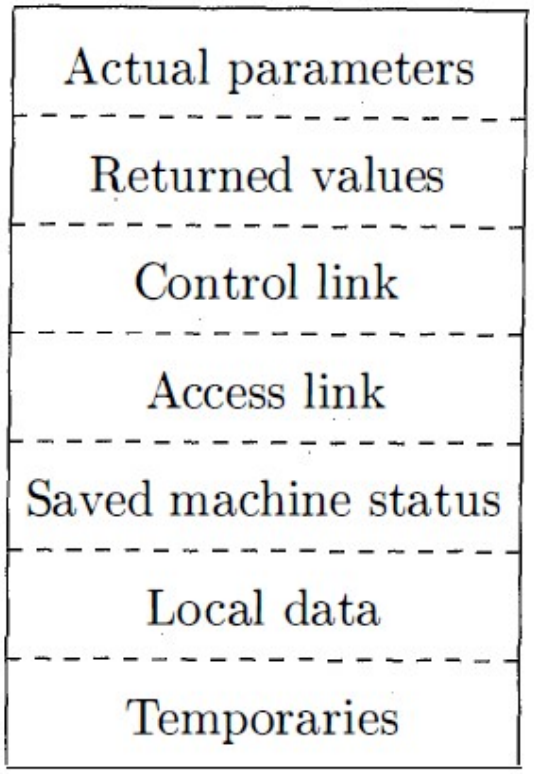
A possible execution of a quicksort



An activation tree for the execution

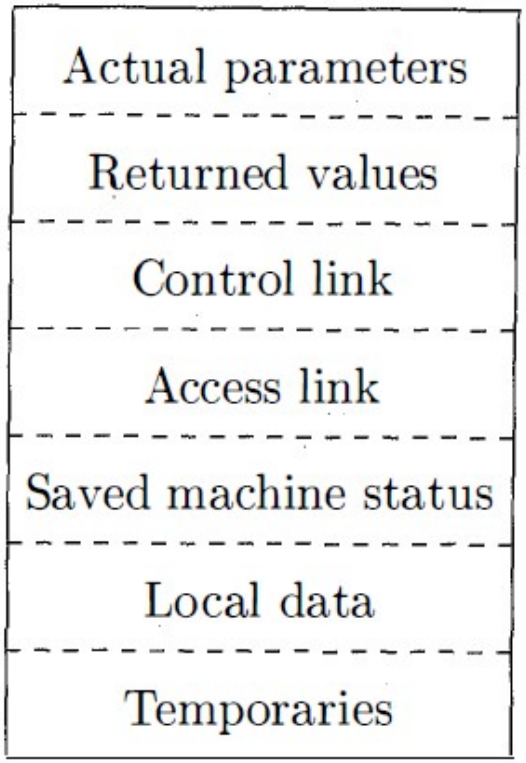
- Activation: execution of a procedure

Stack Allocation (Activation Records)



- Control stack keeps track of live procedure activations.
- Temporaries: temporary results of expressions
- Local data: local data belonging to the procedure
- Saved machine status: return address, registers used in the procedure

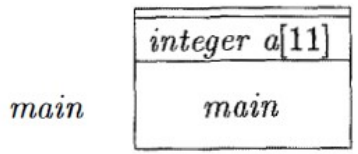
Stack Allocation (Activation Records)



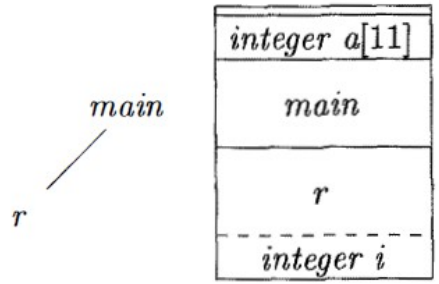
- Access link: nonlocal data held in other activation records (nested procedure)
- Control link: activation record of the caller
- Return value: space for the return value (registers are often used instead for the efficiency).
- Actual parameters: space for the actual parameters

Stack Allocation (Activation Records)

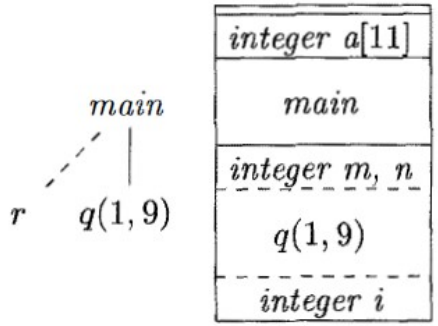
```
int a[11];
void readArray() {
    int i;
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```



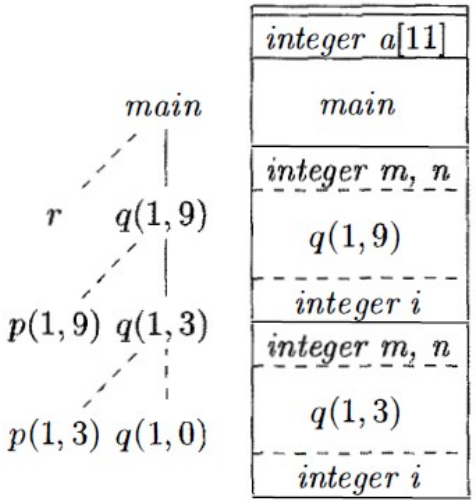
(a) Frame for *main*



(b) r is activated

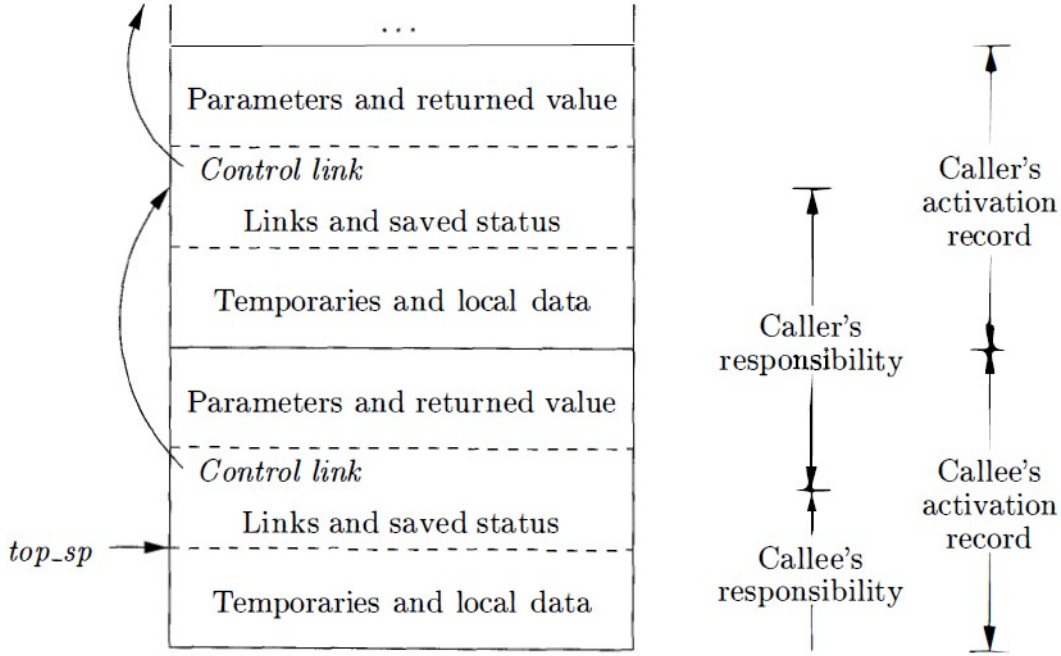


(c) r has been popped and $q(1,9)$ pushed



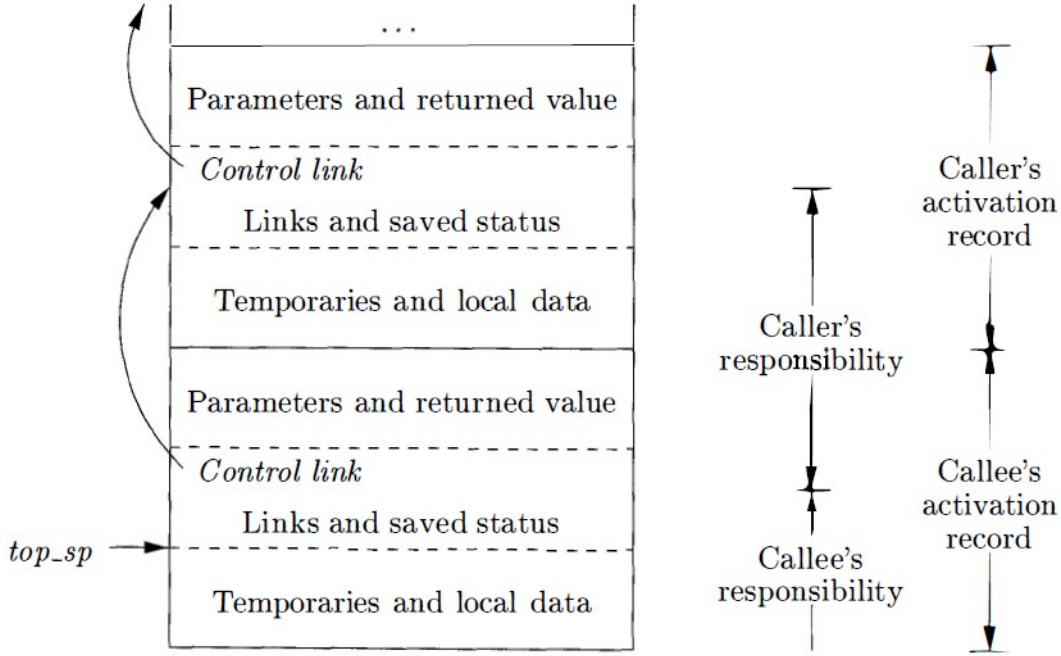
(d) Control returns to $q(1,3)$

Calling Sequence



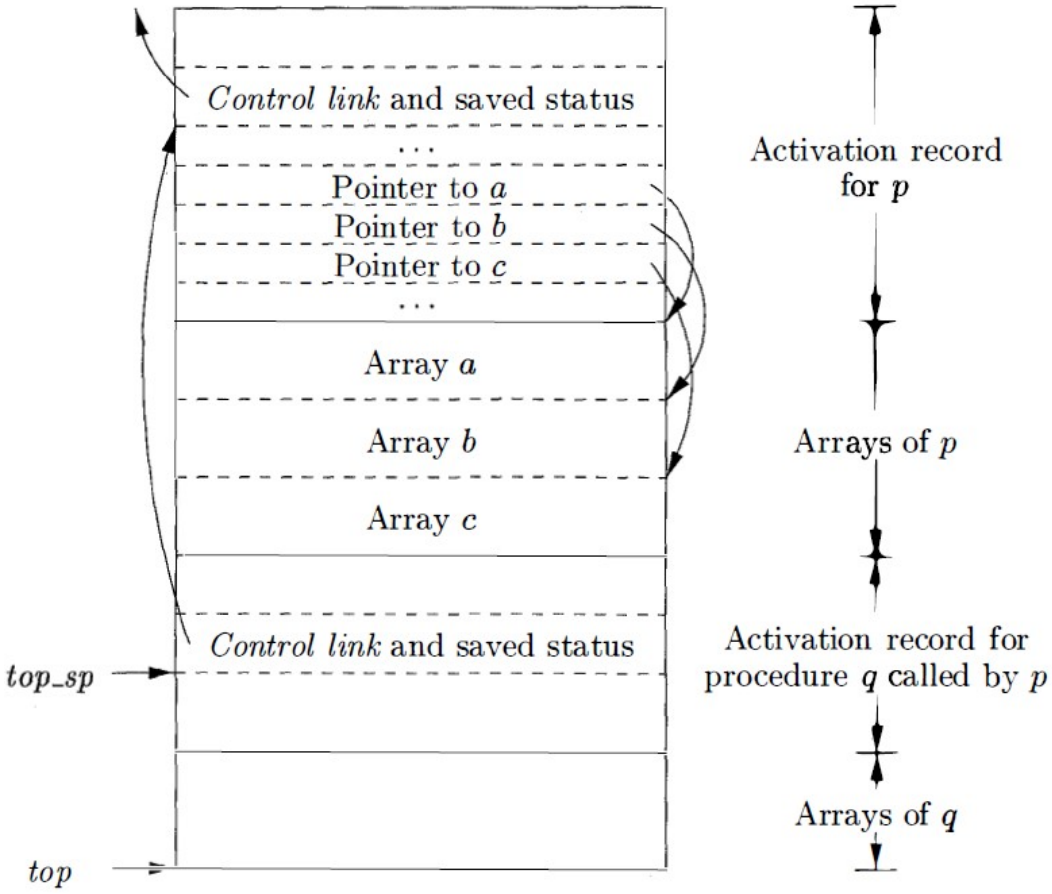
- Caller: eval actuals, allocate return address, temporaries, and local data, move top_sp
- Callee: save register values, initialize local variables

Return Sequence



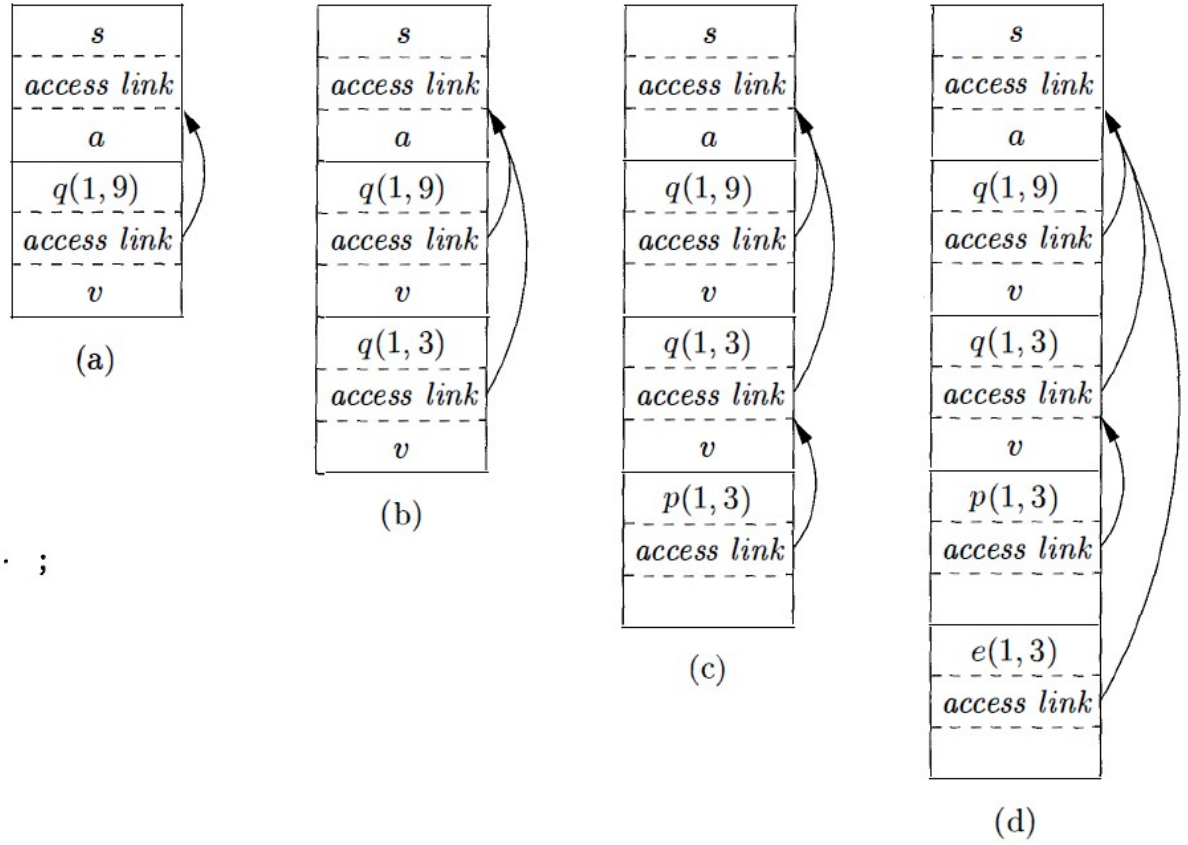
- Callee: place a return value, restore top_sp and other registers, jump back to caller's code.
- Caller: copy to returned value to its activation record.

Variable Length Data



- When data size is unknown at the compile time
  - E.g. Array size is passed by the parameter
- Activation record has pointers to actual arrays

Nested Procedures
(Quicksort in ML)

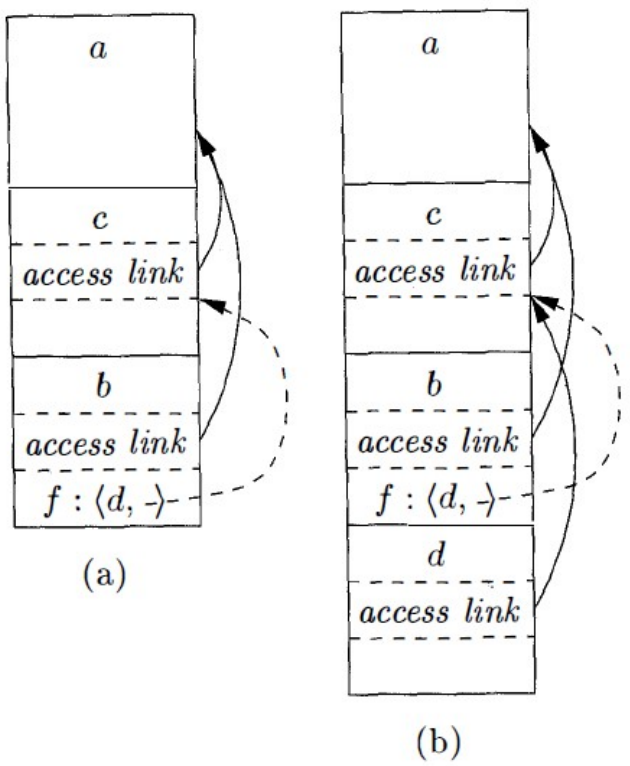



```
fun sort(inputFile, outputFile) =
    let
        val a = array(11,0);
        fun readArray(inputFile) = ··· ;
              ··· a ··· ;
        fun exchange(i,j) =
              ··· a ··· ;
        fun quicksort(m,n) =
            let
                val v = ··· ;
                fun partition(y,z) =
                     ··· a ··· v ··· exchange ···
            in
                ··· a ··· v ··· partition ··· quicksort
            end
    in
        ··· a ··· readArray ··· quicksort ···
    end;
```

Access Links

Procedure Parameters

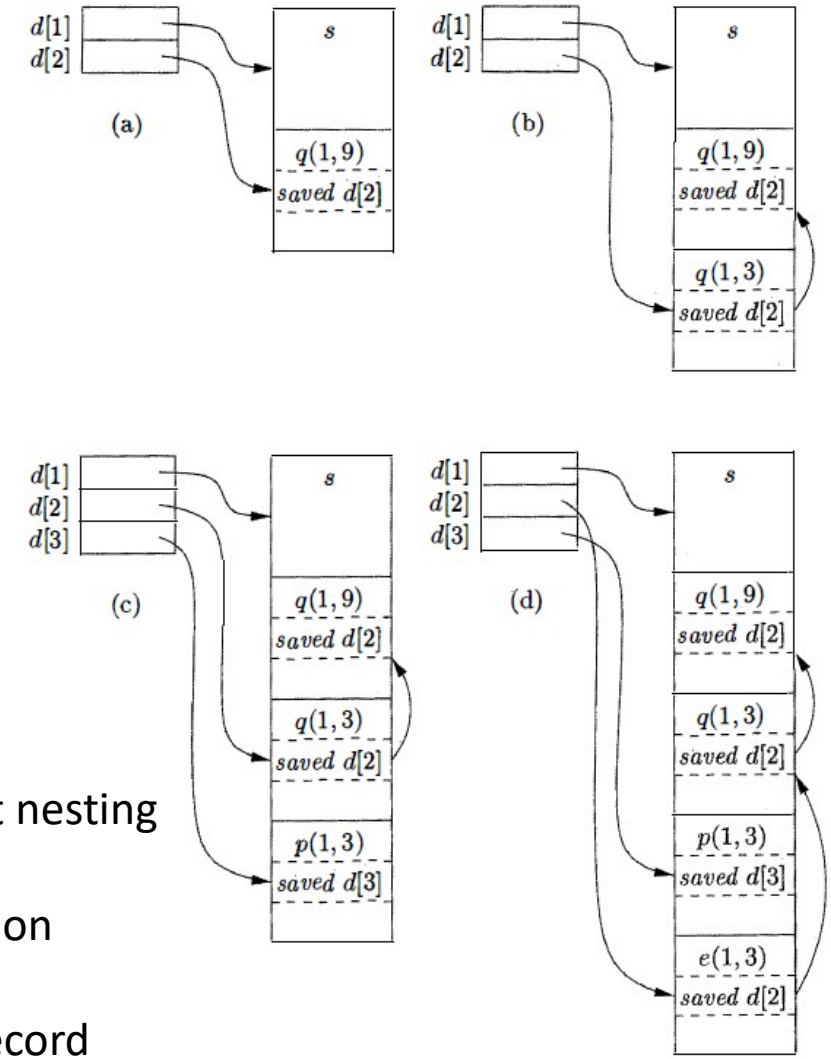
```
procedure a();
    var y: integer;
    procedure b(procedure f(x:integer));
        var y:integer;
    begin
        y := 20;
        y := f(30);
    end
    procedure c();
        var y : integer
        procedure d(x:integer);
        begin
            d := x + y;
        end
    begin
        y := 10;
        b(d);
    end
begin
    c();
end
```



- Caller needs to pass the access link along with the procedure parameter

Displays

```
procedure s();
    procedure q(x,y:integer);
        procedure p(x,y:integer);
        begin
            e(1,3);
        end
    begin
        ...
        q(1,3);
        ...
        p(1,3);
    end
    procedure e(x,y:integer);
    begin
        ...
    end
begin
    q(1,9);
end
```



When a new activation record for a procedure at nesting depth i is set up

1. Save the value of d[i] in the new activation record
2. Set d[i] to point to the new activation record

When the activation ends, d[i] is reset to the saved value

Parameter Passing

- Call-by-value
  - Formal parameters are treated like a local variable
  - Caller evaluates the actual parameters and places their r-values in the formal parameters.
- Call-by Reference
  - If an actual parameter is a name or an expression having an l-value, the l-value is passed
  - If an actual parameter does not have l-value (like 1+2), then the parameter is evaluated in a new location and the address of the location is passed.

Parameter Passing

- Copy-Restore
  - During the calling sequence, the r-values of actual parameters are passed like call-by-value.
  - During the return sequence, for the actual parameters with l-values, the updated values are copied.

```
program copyout(intput, output);
    var a: integer;
    procedure unsafe(var x: integer);
        begin
            x := 2;
            a := 0;
        end
    begin
        a := 1;
        unsafe(a);
    end
```

Parameter Passing

- Call-by-Name
  - Procedure is treated as if it were a macro
  - Local variables of called procedure are systematically renamed into a distinct new name.
  - Actual parameters are surrounded by parenthesis if necessary.

```
#define swap(a,b)\
    t = a; a = b; b = t;

swap(i, a[i])
    t = i; i = a[i]; a[i] = t;
```