UCR

# Run-Time Environments

*Issues* :

- Managing the relationship between names in the source program and data objects that exist at run-time.
- Managing the allocation/deallocation of, and access to, data objects at runtime.
- Controlling and keeping track of different activations of a procedure (in case of recursion, several activations of a procedure may be "alive" at the same time).

# Flow of Control

*Assumptions* :

- Control flows sequentially: at each step during execution, control is at some specific point in the program .

  ⇒ *no program-level parallelism.*

- Each execution of a procedure starts at the beginning of the procedure body; When control eventually leaves the procedure, it returns to the point immediately following the place where the procedure was called.

  ⇒ *no coroutining, no backtracking.*

## *Characteristics of Procedure Activations*

- Our assumptions imply that given two procedure activations $a$ and $b$, their lifetimes are either disjoint or are nested.

- This implies that activations can be managed using a *control stack*:
  - Push a node for an activation when the activation begins, i.e., at the entry to the procedure.
  - Pop the node when the activation ends, i.e., at the return from the call.

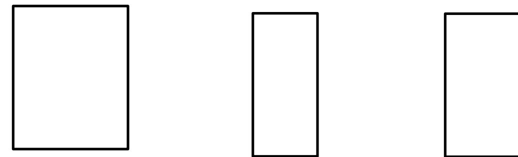
# Characteristics for Dynamically Allocated Data Objects

- The order in which objects are allocated does not determine the order in which they are deallocated.
- Therefore we allocate them from the *heap*.

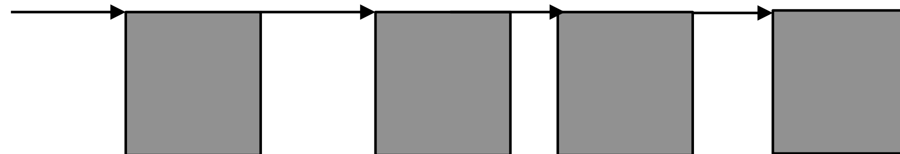
**At Start** **Free List**

**After Allocation & Deallocation Of Data Objects**

**Data Objects**



**Free List**



# Language Issues that Affect the Compiler

UCR

- Can procedures be recursive?
- What happens to the values of locals on return from a procedure?
- Can a procedure refer to non-local variables?
- How are parameters to a procedure passed?
- Can procedures be passed as parameters?
- Can procedures be returned as results?
- Can storage be allocated dynamically under program control?
- Must storage be deallocated explicitly?

UCR

## 7.1. Organization of Run-Time Memory

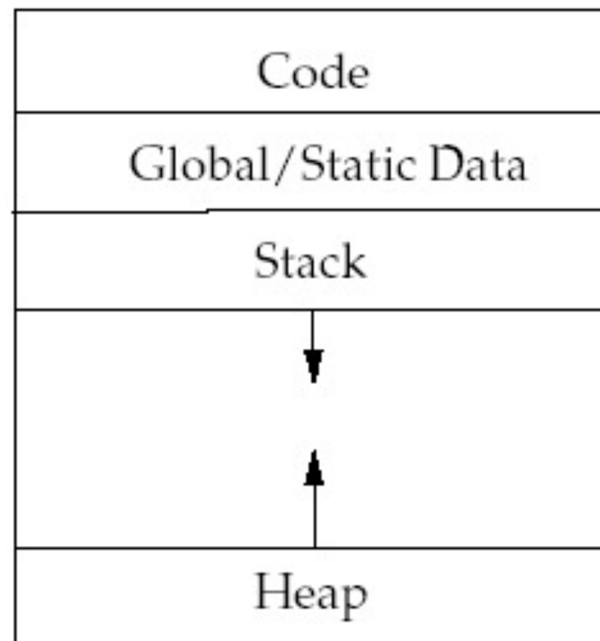
Run-time memory needs to be subdivided to hold the different components of an executing program:

- Generated executable code.
- static data objects.
- A structure to keep track of procedure activations.

  This can generally be considered to consist of two components:

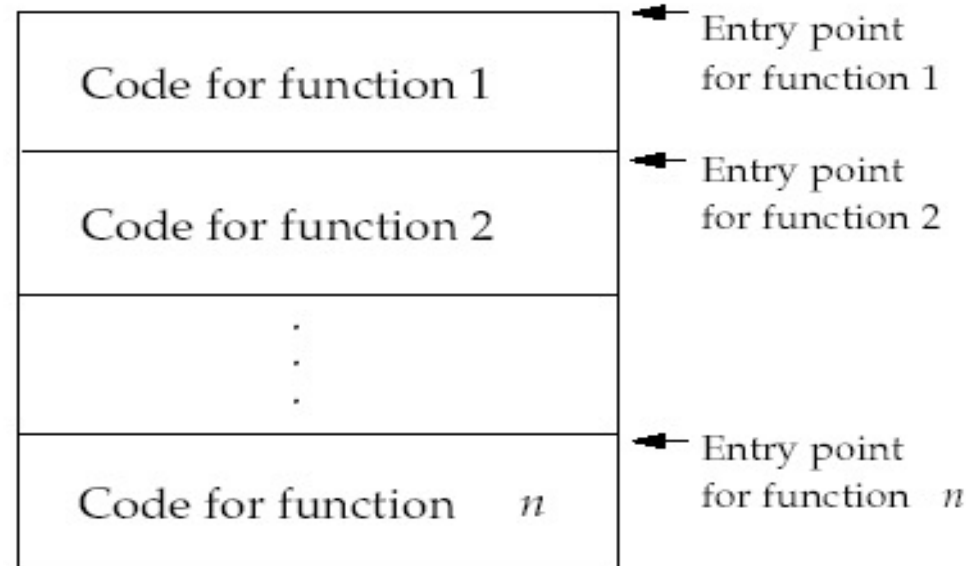
  - *the stack* : for objects whose lifetimes do not exceed the lifetime of the activation; and
  - *the heap* : for objects whose lifetimes exceed that of the activation.

- A typical subdivision might be:



# Organization of Code Area

- Usually, code is generated a function at a time. Thus, the code area layout is of the form:




- Within a function, the compiler has freedom to organize the code in any way.

  Careful layout of code within a function can reduce pipeline "bubbles" and improve i-cache utilization, and thus better performance.

- Careful attention to the order in which the functions are processed can improve i-cache utilization.

## Storage Allocation Strategies for Activation Records

1. *Static allocation* (Fortran 77) :

   - Storage for all data objects laid out at compile time.
   - Can be used only if size of data objects and constraints on its position in memory can be resolved at compile time. No dynamic data structures.
   - Recursive procedures are restricted, since all activations of a procedure must share the same locations for local names.

2. Stack Allocation (Pascal, C) :

- Storage organized as a stack.
- Activation record pushed when an activation begins, and popped when it ends.
- Cannot be used if the values of local names must be retained when an activation ends, or if a called invocation outlives the caller.

3. Heap Allocation (Lisp, Scheme) :

- Activation records may be allocated and deallocated in any order.
- Some form of garbage collection or compaction necessary to reclaim free space.

# Activation Records

An *activation record*, or *stack frame*, manages the information needed by a single activation of a procedure.

**Fields of an Activation Record** (language and compiler dependent):

- temporary values, used during expression evaluation;
- local data;
- saved machine status information (PC, registers, return address);
- (optional) access link, for access to non-local names;
- (optional) control link, points to the activation record of the caller.
- the actual parameters;
- the returned value.

UCR

# Compile-time Layout of Local Data

The compiler must determine where, within an activation record, the memory location(s) for an object are, so that during code generation it can refer to the correct address (use displacement off the stack pointer).

- The amount of storage needed for an object is determined from its type.
- The field for local data in the activation record is laid out as declarations in the procedure are processed. (Variable-length data are kept outside this field.)
- Storage layout must conform to *alignment requirements* of the target machine, e.g., many RISC machines require longwords to be longword-aligned. This may require *padding*.

# **Activation Records** : *Example 2: Intel x86*

## Handling Procedure Calls and Returns

- Procedure calls are handled using *calling sequences* in the code generated.

  Returns from procedures are handled by *return sequences*.

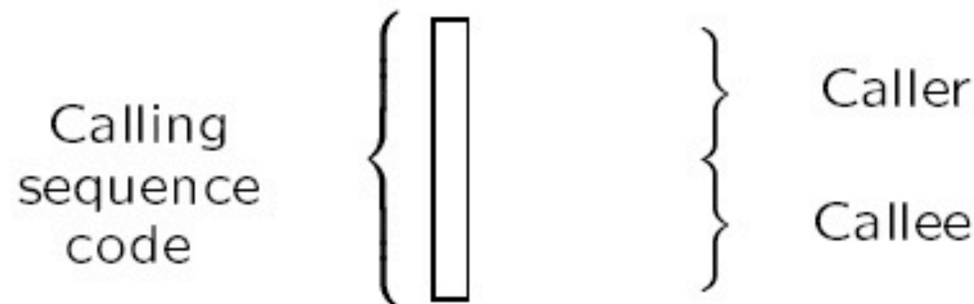
- *Calling sequence* : a code sequence that "sets up a procedure call" :
  - allocates an activation record (model-dependent);
  - loads actual parameters;
  - saves machine state (return address etc.);
  - transfers control to callee.

UCR

- *Return sequence* : a code sequence that handles the return from a procedure call:
  - deallocates activation record;
  - sets up return value (if any);
  - restores machine state (stack pointer, PC, etc.);

## Calling Sequences : Division of responsibilities

- The code in a calling sequence is often divided up between the caller and the callee:




- If there are $m$ calls to a procedure, then the instructions in caller's part of the calling sequence is repeated $m$ times, while the callee's part is repeated exactly once.

  This suggests that we should try to put as much of the calling sequence as possible into the callee.

  However, it may be possible to carry out more call-specific optimization by putting more of the code into the caller instead of the callee.

## Calling/Return Sequences : typical actions

UCR

- Typical calling sequence:
  1. caller evaluates actuals, pushes them on the stack;
  2. caller saves machine status on the stack (in the callee's AR) and updates the stack pointer.
  3. caller transfers control to the callee.
  4. callee saves registers, initializes local data, and begins execution.
- Typical return sequence:
  1. callee stores return value in the appropriate place;
  2. callee restores registers and old stack pointer;
  3. callee branches to the return address.

## 7.3.2. Access to Non-Local Names

*Basic Issue* : In a language with lexical scope and nested procedures, e.g. Pascal or Scheme, how do we know where to find (at runtime) a variable declared in an enclosing scope?

*Example* : Consider the program

```
procedure p(m:integer)
begin
     x : integer;

     function q(n:integer): integer;
     begin
          if (n > 0) then
               return 2*q(n-1);
          else
               return x+1;
     end

    print q(m+2);
end
```

The variable `x` lives in `p`'s activation record. But we don't know how deep in the stack this may be, since we don't know how many levels of recursion there will be in `q` at runtime.

<u>*Basic Idea*</u> : Generate code to pass an *access link* at each procedure call.

- Suppose a procedure $p$ is nested immediately within a procedure $q$, then the access link to a procedure $p$ is a pointer to the activation record of the most recent activation of $q$.

- The code to set up access links can be determined at compile time, using the idea of *nesting depth*.

  Intuitively:

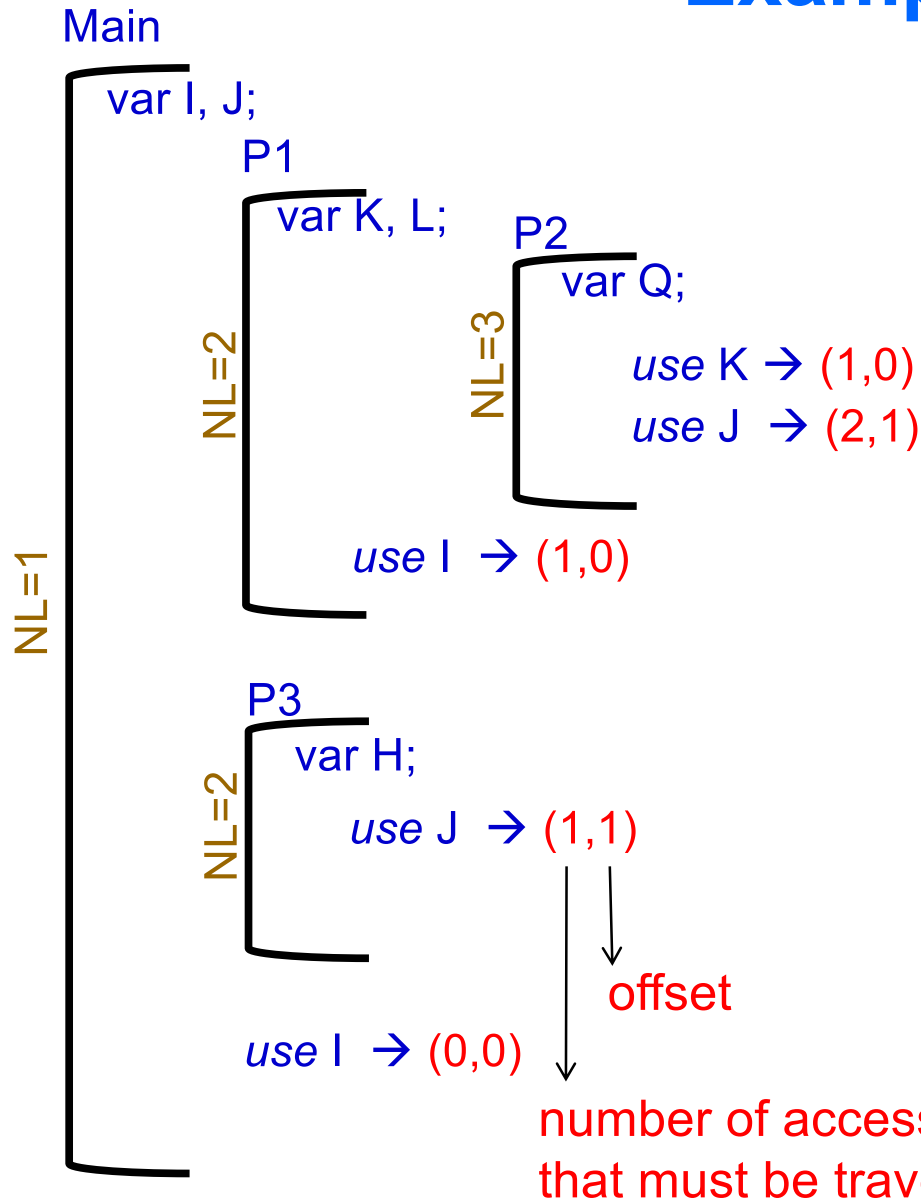
  - The outermost scope has nesting depth $= 0$.
  - The nesting depth increases by 1 each time we enter a new scope, decreases by 1 when we leave a scope.

- Suppose a procedure $p$ at nesting depth $n_p$ refers to a non-local variable a whose nesting depth is $n_a$: the storage for a is given by the pair:

  $\langle\ n_p - n_a$, offset of a in AR $\rangle$

  This pair is computed at compile time. The first number gives the no. of access links to be traversed.

# Example




Main: NL=1

| I : 0 |
|---|
| J : 1 |
| P1 : .......... |
| P3 : .......... |

P1: NL=2

| K : 0 |
|---|
| L : 1 |
| P2 : .......... |

P2: NL=3

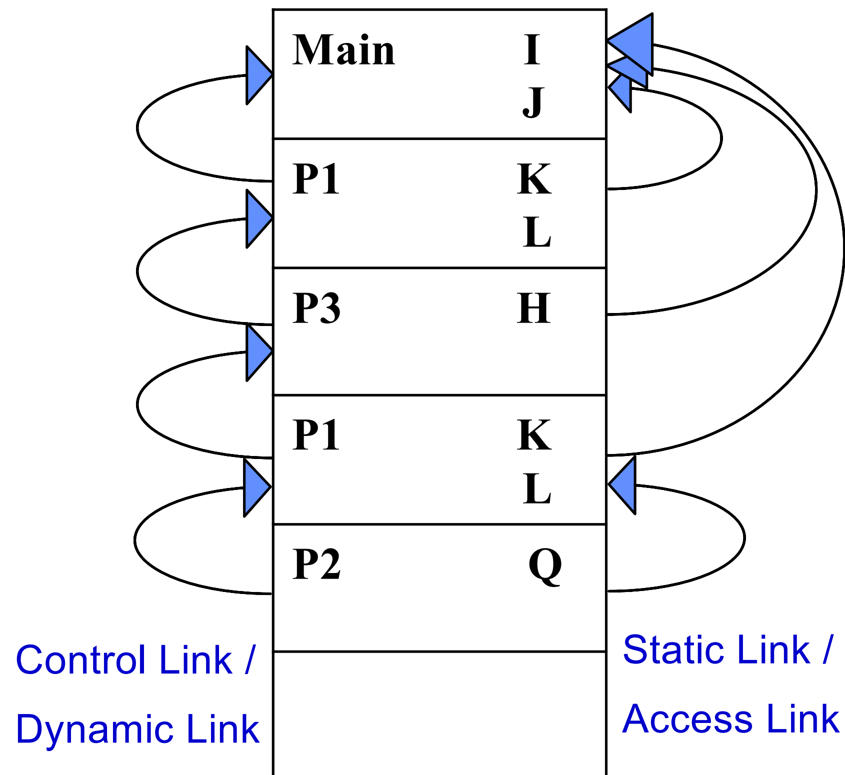
| Q : 0 |
|---|

P3: NL=2

| H : 0 |
|---|

## Example Contd.

***Call Chain:*** Main → P1 → P3 → P1 → P2




**In P2 access K → (1,0)**

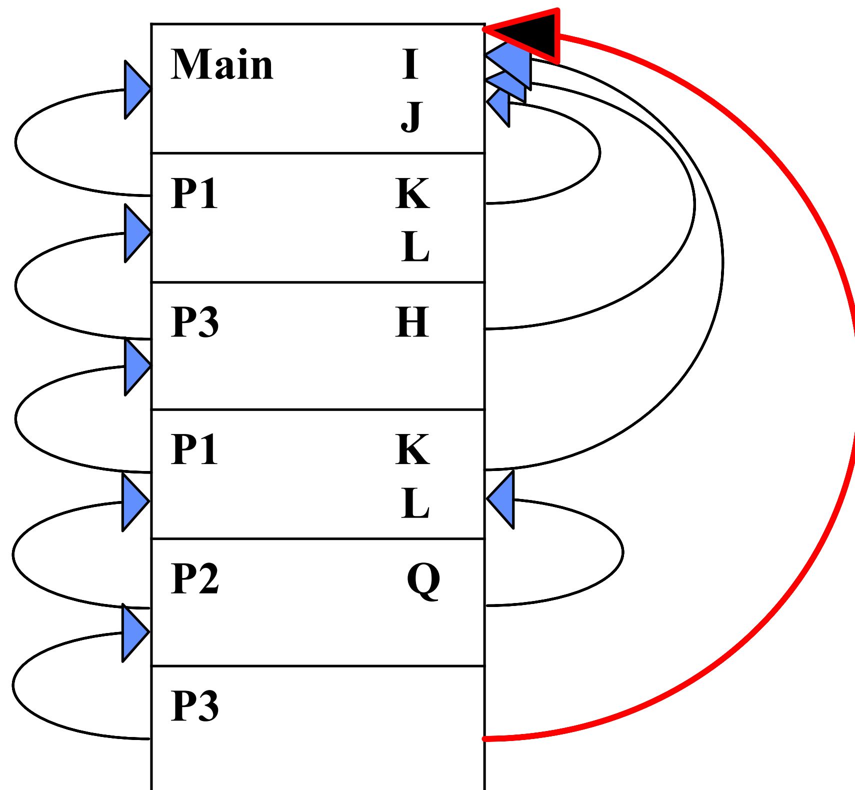
**In P2 access J → (2,1)**

**In P2 access I → (2,0)**

**………**

## Example Contd.

**Next call P3 – set up the access link**




**P2 calls P3**

**c – current nesting level**

**d – nesting level of callee's declaration**

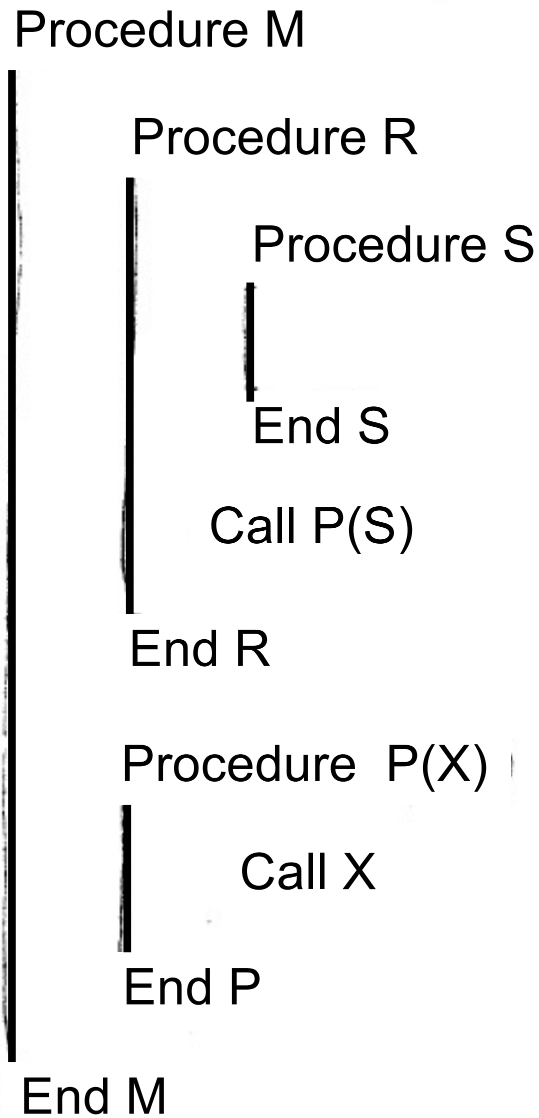
**Traverse (c – d) links from P2 to find where P3's access link should point to**

**(c – d) = 3 - 1 = 2**

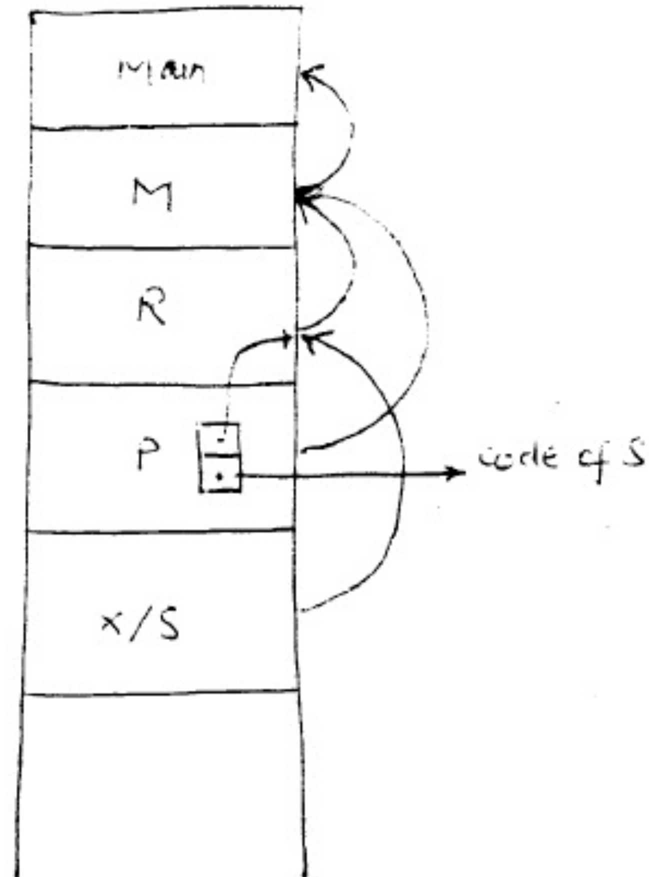
**P2 → P1 → Main**

***Access link of P3 points to Main***

# Procedure Parameters



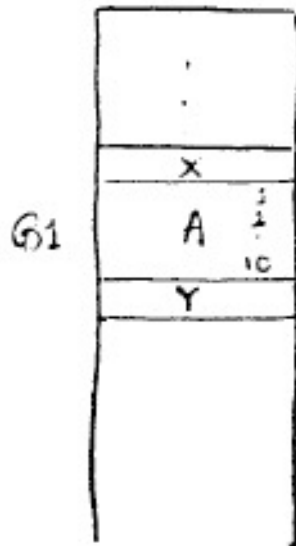

**Main → M → R → P(S) → X/S**



# Dynamic Arrays – Maintaining Constant Offsets

UCR

```
B1: begin
        Var  X;
        Array  [1 .. 10]   A;
        Var Y;
    end
```




Var X;

Array [$l_1$ .. $u_1$] A;

Var Y;