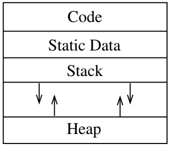
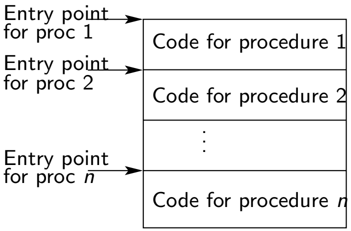
Runtime Environment

- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:
    1. Generated code for various procedures and programs.
       forms text or code segment of your program: size known at compile time.
    2. Data objects:
        - Global variables/constants: size known at compile time
        - Variables declared within procedures/blocks: size known
        - Variables created dynamically: size unknown.
    3. Stack to keep track of procedure activations.
- Subdivide memory conceptually into code and data areas:
    a) Code: Program instructions
    b) Stack: Manage activation of procedures at runtime.
    c) Heap: holds variables created dynamically



Code Segment and Global Data

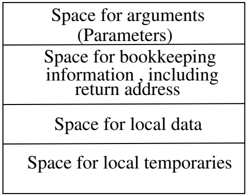
- Code segment: stores code for various procedures and main routine.
- Size of code segment fixed prior to execution $\implies$ all code addresses computable at compile time.



- You will be generating assembly routines, which then gets assembled and loaded into memory in the above format.
  Entry points for procedures etc. are all symbolic (and relocatable.)
- Global and static data of programs also have fixed size. Space for such usually allocated separately from other.
- Usually, constants such as strings are also allocated in the global space: "Hello world".

Activation Record

- An important part of memory organization.
- Associated with *each* activation of a procedure is storage for the variables declared in the procedure. Storage called **Activation Record** or **frame**.
- A typical activation record contains: i) Parameters, ii) Local variables, and iii) space for managing activation records, and iv) temporaries.


Space for arguments (Parameters)
Space for bookkeeping information , including return address
Space for local data
Space for local temporaries

- Sizes of different parts:
  - space for bookkeeping: fixed and same for all procedures.
  - space for arguments and local data: same for a procedure but vary from one procedure to another. Calculated by compiler during code generation phase
  - Space for temporaries calculated by compiler.
- Depending on language, an AR can be in a static area (FORTRAN), stack area (C, PASCAL, Java–), or heap area (LISP).

Runtime Organization -cont'd.

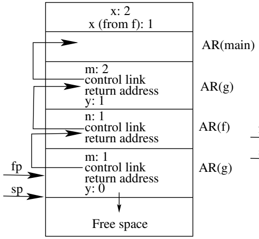
- Processor registers also part of runtime environment.
- Registers may store temporaries, local variables, and global variables. On RISC processors, entire static area or AR can be stored in registers.
- Special purpose registers: keep track of execution of machines.
  - Program counter (pc)
  - Stack pointer (sp)
- Special registers to keep track of procedure activations:
  - frame pointer (fp): points to current activation record
  - argument pointer(ap): points to area of activation record reserved for arguments.
- Important aspect of design of runtime environment: i) what to do when a procedure is called (calling sequence) and ii) what to do when procedure returns (return sequence)
- Calling sequence:
  - allocate space for AR
  - compute and store arguments
  - store and set registers etc.

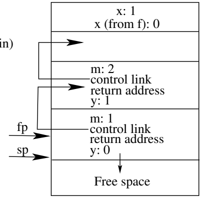
  How to divide calling sequence operations between caller and callee?
- Return sequence:
  - place returned value
  - adjust registers
  - fix activation record stack

Stack-based runtime environments without local procedures

- In a language where all procedures are global (such as C), a stack based environment requires two things:
  - Maintenance of a pointer to current AR (usually called **fp**) to allow access to local variables, and
  - a record of position or size of the immediately preceding AR (caller's AR) to allow that AR to be recovered when current call ends.
    Kept in current AR as a pointer to previous AR. (Known as **Control Link** or **Dynamic Link**).
- Example:

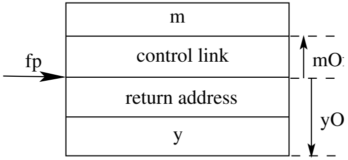
```
int x = 2;
void f (int n) {              void g(int m) {
  static int x = 1;             int y = m-1;
  g(n);                         if (y > 0) {
  x--;                            f(y);
}                                 x--;
                                }}
main() { g(x); return(0); }
```





How to access variables?

- Note: parameters and local variables no longer accessed by fixed addresses..
  Instead find them by offset from current frame pointer.
  Offset can be determined statically by compiler.
- Example: For procedure g, each AR of g has same form, and parameters and local variables are located exactly at the same relative location:



  mOffset = size of control link = + 4 bytes
  yOffset = -(size of y + size of return address) = - 6
  Hence m and y can respectively be accessed by 4(fp) and -6(fp).
- Local arrays and structures can be similarly accessed.

How is activation record constructed?

- Calling sequence (at the site of call):
  1. Evaluate actual parameters and place their values on the activation record for the callee.
  2. Store (push) the fp as the control link in new AR
  3. Change fp so that it points to beginning of new AR.
  4. Store return address in new AR (if necessary)
  5. Perform a jump to code of procedure to be called.
- Inside calling procedure
  1. Allocate space for local variables.
  2. Allocate temporary space for partial results.
- Return sequence: :
  1. Copy fp to sp.
  2. Load control link into fp.
  3. A return value is placed where the caller can find it, and the machine registers are restored.
  4. Pop the stack
  5. Send the control to the caller.
- How are procedures with variable number of arguments (such as printf) handled?

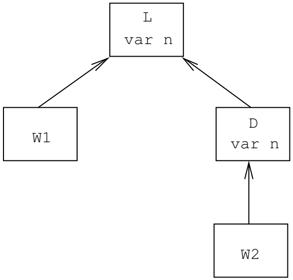
How to deal with local temporaries and nested declarations?

- Local temporaries are partial results of computations that must be saved across procedure calls:
  ```
  x[i] = (i+j)*(i/k+ f(j))
  ```
  - Store temporary values in registers
  - Compiler creates explicit variables for temps, allocates space for them on stack..
    Compiler will need to determine number of temps required and add them to symbol table.
- Nested declarations:
  ```
  void p (int x, double y) {
     char a; int i;
        ...
     A: {
     double x; int j;
           ...
     }
     B: {
     char *a; int k;
           ...
     }
  ```
  Many solutions:
  - Treat them like temporary variables: allocate variables on stack as block is entered, and deallocate as you exit.
    You can reuse the space on stack
  - Allocate space for all nested blocks once.
    Need to be careful about names and where they will appear on stack, for instance name x. Your compiler must arrange to make sure that the symbol tables store right offset.

Stack-based environments with local procedures

- Previous runtime support insufficient because no support for non-local and non-global variables.
- Example:

```
program L;
var n: char;
   procedure W
   begin
     writeln(n)
   end;
   procedure D
   var n: char;
   begin
     n := 'D';
     W
   end;
begin
   n := 'L';
   W;
   D
end.
```

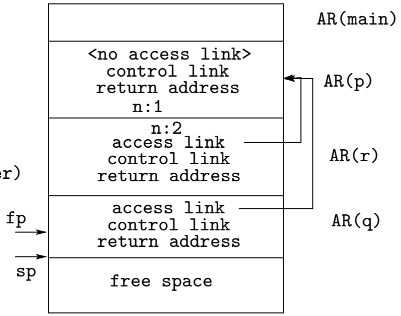


Access of `n` in first invocation of `W` (W1)
Access of `n` in second invocation of `W` (W2)

Stack-based environments with local procedures

- Solution: add an extra piece of bookkeeping information, called **access link**.
  Access link: points to AR that represents *defining environment* for a procedure.
- Example:

```
program nonleaf;

procedure p;
var n:  integer;

   procedure q;
   begin
      (* refer n *)
   end; (*q *)

   procedure r(n:integer)
   begin
      q;
   end (*r*)
begin (*p *)
 n := 1;
 r(2);
end;

begin (* main *)
 p;
end.
```

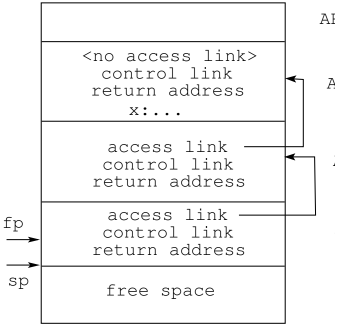


- Note that there is an access link: points to the most recent activation of the lexically enclosing block. Specifically, chains of access links are easier to follow if the access link in an activation record points to the access link field in the record for the lexically enclosing block.

Access link chains

- Access chaining: May need to follow access links to access a variable:

```
program chaining;

procedure p;
var x: integer;

   procedure q;
     procedure r
     begin
         x := 2;
         ...
         if ... then p;
     end (*r*)
  begin
     r;
  end; (*q *)
begin   (*p *)
 q;
end;
begin   (* main  *)
 p;
end.
```



- To access:
  - load 4(fp) into register r
  - load 4(r) into register r
  - Now access x as -6(r)

Question: how much to follow in access link?

How to create access links and use them?

- Lexic levels (or nesting depth) of blocks and procedures:
  `Let LL(block) = block's` **lexic level**
  `LL(outermost block) = 1`
  `LL(enclosed block) = 1 + LL(enclosing block)`
- How to find access link of a procedure during a call?
  - find difference, d, in lexical level between calling and called procedure.
  - if difference is 0 or -1, point to caller.
  - else follow d access links, and copy access link of this AR.

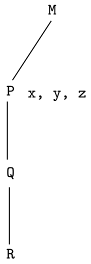
  Note that compiler will need to generate code for doing so.

  ```
  store lexical depth d in to register r;
  while (d != 0) {
     load access link of previous AR into r
     d = d - 1;
  }  copy r into access link location on AR.
  ```
- What does compiler do for finding a variable using access links?
  - Find the difference, d, between lexic level of declaration of name, and lexic level of calling procedure.
  - Generate code for following d access links to reach the right AR.
  - generate code for accessing variable through offset mechanism.

Lexical Scope: Nested Procedures and Pascal

- Example:

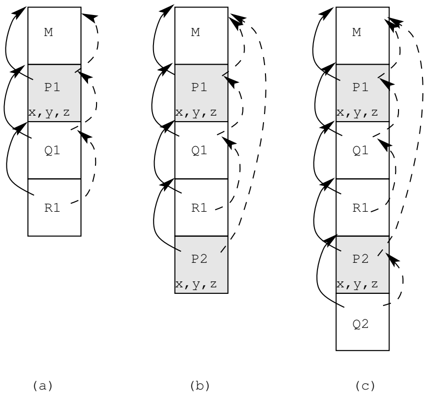
```
program M;

  procedure P;

      var x, y, z;

      procedure Q;

           procedure R;
           begin
              ...z := P; ...

           end R;

      begin

           ... y := R; ...

     end Q;

  begin

        ... x := Q; ...

  end P;
begin

  ... P; ...

end M;
```



```
LL(M) = 1
LL(P) = 2
LL(Q) = 3
LL(R) = 4
```

Activation Records

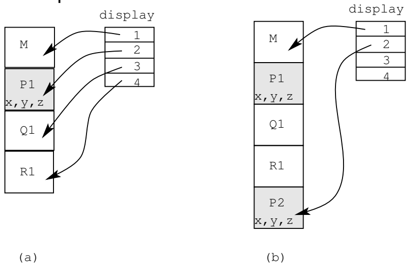
- Example:



- left side arrows: control link, right side arrows: access link
- (a): M calls P, P calls Q, Q calls R
  (b): R calls P (recursive call to P)
  (c): P calls Q

Display

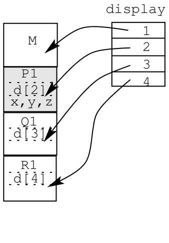
- Search by following access links can be slow. Need optimization so that activation record in which variable is located can be accessed without search.
- Display: a global array `d` of pointers to activation records, indexed by lexical nesting depth. The number of display elements known at compile time.
- Array element `d[i]` points to the most recent activation of the block at nesting depth (or lexic level) *i*.
- A nonlocal X is found in the following manner:
  1. Use one array access to find the activation record containing `X`. If the most-closely nested declaration of `X` is at nesting depth `i`, then `d[i]` points to the activation record containing the location for `X`.
  2. Use relative address within the activation record.
- Example:



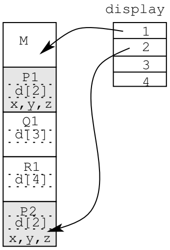
Display - cont'd.

- How to maintain display information?
  1. When a procedure is called: a procedure *p* at a nesting depth *i* is setup:
     - Save value of `d[i]` in activation record for *p*.
     - Set `d[i]` to point to new activation record.
  2. When a *p* returns:
     - Reset `d[i]` to display value stored.

  Example:



(a)



(b)

- Where can display be maintained?
  - Registers
  - In statically allocated memory (data segment)
  - Store display on control stack and create a new copy on each entry.