• Chapter 7: Runtime Environment
  – Run time memory organization.

```c
char   abc[1000];
char *foo() { char buf[50], *c; buf[0] = ‘\0’; c=malloc(50);
             return( c );}
main() {char *c; c = foo();}
```

• We need to use memory to store:
  – code
  – static data (global variables)
  – dynamic data objects
    » data that are used when executing a certain procedure.
    » Dynamically allocated objects (malloc, free).
- Typical organization of run-time memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code</strong></td>
<td><strong>Static Data (global variables)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>stack (memory for procedures)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>heap (memory for dynamically allocated data)</strong></td>
</tr>
</tbody>
</table>
• Activation Records:
  • also called frames
  • Information (memory) needed by a single execution of a procedure
  • A general activation record:

<table>
<thead>
<tr>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters</td>
</tr>
<tr>
<td>optional control link</td>
</tr>
<tr>
<td>optional access link</td>
</tr>
<tr>
<td>machine status</td>
</tr>
<tr>
<td>local variables</td>
</tr>
<tr>
<td>temporaries</td>
</tr>
</tbody>
</table>
– Storage Allocation Strategies
  • static allocation lays out storage for all data objects at compile time.
    – Restrictions:
      » size of object must be known and alignment requirements must be known at compile time.
      » No recursion.
      » No dynamic data structure

• Stack allocation manages the run time storage as a stack
  – The activation record is pushed on as a function is entered.
  – The activation record is popped off as a function exits.
  – Restrictions:
    » values of locals cannot be retained when an activation ends.
    » A called activation cannot outlive a caller.
• Heap allocation -- allocates and deallocates storage as needed at runtime from a data area known as heap.
  – Most flexible: no longer requires the activation of procedures to be LIFO.
  – Most inefficient: need true dynamic memory management.

• Note: static allocation too restricted, heap allocation too inefficient. Most current compiler/language/processor uses the stack allocation scheme.
Example of stack allocation:

Program sort

var

procedure readarray;

function partition(…)

procedure quicksort(…)

Begin

Main

readarray              quicksort(1, 9)

partition(1, 9)       quicksort(1, 3)

partition(1, 3)       quicksort(1, 0)

……

readarray

quicksort

quicksort

……

Begin

end
• How would this happen (push and pop the activation record)?
  – Everything must be done by the compiler.
  – What makes this happen is known as **calling sequence** (how to implement a procedure call).
    • A calling sequence allocates an activation record and enters information into its fields (push the activation record).
  
  – On the opposite of the calling sequence is the **return sequence**.
    • Return sequence restores the state of the machine so that the calling procedure can continue execution.
A possible calling sequence:

- The caller evaluates actuals and push the actuals on the stack
- The caller saves return address (pc) the old value of sp into the stack
- The caller increments the sp
- The callee saves registers and other status information
- The callee initializes its local variables and begin execution.

A possible return sequence:

- The callee places a return value next to the activation record of the caller.
- The callee restores other registers and sp and return (jump to pc).
- The caller copies the return value to its activation record.

In today’s processors, there is usually special support for efficiently realizing calling/return sequence: executing procedures is too important!!
• Access to nonlocal variables.
  – Nonlocal variables in C (without nested procedures):
    • Still have nested scopes (blocks).
    • Solution:
      – All data declared outside procedures are static.
      – Other names must be at the activation record at the top of the stack, can be accessed from sp.
        » Treat a block as a parameter-less procedure
        » Allocates space for all blocks in a procedure.

• Example: Fig. 7.18 in page 413.
• Access to nonlocal variables.
  – Nonlocal variables in PASCAL (with nested procedures):
    • the scheme for C will break.

```pascal
(1) program sort(input, output);
(2)   var a : array [0..10] of integer;
(3)     x : integer;
(4)   procedure readarray;
(5)     var i : integer;
(6)     begin ... a ... end { readarray };
(7)   procedure exchange( i, j: integer);
(8)     begin
(9)       x := a[i]; a[i] := a[j]; a[j] := x
(10)    end { exchange } ;
(11)  procedure quicksort(m, n: integer);
(12)  var k, v : integer;
(13)  function partition(y, z: integer): integer;
(14)     var i, j : integer;
(15)     begin ... a ... 
(16)       ... v ...
(17)       ... exchange(i,j); ...
(18)     end { partition } ;
(19)   begin ... end { quicksort };
(20)   begin ... end { sort } .
```

Fig. 7.22. A Pascal program with nested procedures.
• Access to nonlocal variables.
  – Nonlocal variables in PASCAL (with nested procedures):
    • The scheme for C will break (static for all non-locals).
  • Access links
    – If p is nested immediately within q in the source text, then the access link in an activation record for p points to the access link in the record for the most recent activation of q.
    – A procedure p at nesting depth n_p accesses a nonlocal a at nesting depth n_a: (1) following n_p – n_a links and (2) using the relative offset in the activation record.
(1) program sort(input, output);
(2) var a : array [0..10] of integer;
(3)   x : integer;
(4)   procedure readarray;
(5)     var i : integer;
(6)     begin ... a ... end { readarray };
(7)   procedure exchange(i, j: integer);
(8)     begin
(9)       x := a[i]; a[i] := a[j]; a[j] := x
(10)      end { exchange };
(11)  procedure quicksort(m, n: integer);
(12)    var k, v: integer;
(13)    function partition(y, z: integer): integer;
(14)      var i, j: integer;
(15)      begin ... a ...
(16)          ... v ...
(17)          ... exchange(i, j); ...
(18)      end { partition };
(19)      begin ... end { quicksort };
(20)    begin ... end { sort }.

Fig. 7.22. A Pascal program with nested procedures.
Fig. 7.23. Access links for finding storage for nonlocals.
• **Display:**
  • An alternative to access link (a faster method to access nonlocals).
  • Using an array d of pointers to activation records, the array is called a **display**.
    – Referencing nonlocal variables always requires only two memory references.
  • Suppose control is in a procedure p at nesting depth j, then the first j-1 elements of the display point to the most recent activation of the procedures that lexically enclose procedure p, and d[j] points to the activation of p.
• Setting up the display:
  • When a new activation record for a procedure at nesting depth \( k \):
    – save the value of \( d[k] \) in the new activation record
    – set \( d[k] \) to point to the new activation record.
– Parameter passing

• The method to associate actual parameters with formal parameters.

• The parameter passing method will effect the code generated.

• Call-by-value:
  – The actual parameters are evaluated and their r-values are passed to the called procedure.
  – Implementation:
    » a formal parameter is treated like a local name, so the storage for the formals is in the activation record of the called procedure.
    » The caller evaluates the actual parameters and places their r-values in the storage for the formals.
– Call-by-reference:
  • also called call-by address or call-by-location.
  • The caller passes to the called procedure a pointer to the storage address of each actual parameter.
    – Actual parameter must have an address -- only variables make sense, an expression will not (location of the temporary that holds the result of the expression will be passed).

– Copy-restore:
  • A hybrid between call-by-value and call-by-reference.
    – The actual parameters are evaluated and its r-values are passed to the called procedure as in call-by-value.
    – When the control returns, the r-value of the formal parameters are copied back into the l-value of the actuals.