Activation Records

- Modern imperative programming languages typically have local variables.
  - Created upon entry to function.
  - Destroyed when function returns.
- Each invocation of a function has its own instantiation of local variables.
  - Recursive calls to a function require several instantiations to exist simultaneously.
  - Functions return only after all functions it calls have returned ⇒ last-in-first-out (LIFO) behavior.
  - A LIFO structure called a stack is used to hold each instantiation.
- The portion of the stack used for an invocation of a function is called the function’s stack frame or activation record.

The Stack

- Used to hold local variables.
- Large array which typically grows downwards in memory toward lower addresses, shrinks upwards.
- Push(r1):
  
  ```
  stack_pointer--; 
  M[stack_pointer] = r1;
  ```
- r1 = Pop():
  
  ```
  r1 = M[stack_pointer];
  stack_pointer++;
  ```
- Previous activation records need to be accessed, so push/pop not sufficient.
  - Treat stack as array with index of stack pointer.
  - Push and pop entire activation records.
Consider:

```plaintext
let
    function g(x:int) =
        let
            var y := 10
        in
            x + y
        end
    function h(y:int):int =
        y + g(y)
    end
in
    h(4)
end
```

---

**Example**

**Step 1: h(4) called**
Chunk of memory allocated on the stack in order to hold local variables of h. The activation record (or stack frame) of h is pushed onto the stack.

```
Stack
Frame for h
      y=4
```

**Step 2: g(4) called**
Activation record for g allocated (pushed) on stack.

```
Stack
Frame for h
      y=4
Stack
Frame for g
      x=4
      y=10
```

---

**Example**

**Step 3: g(4) returns with value 14**
Activation record for g deallocated (popped) from stack.

```
Stack
Frame for h
      y=4
```

**Step 4: h(4) returns with value 18**
Activation record for h deallocated (popped) from stack. Stack now empty.
Can have multiple stack frames for same function (different invocations) on stack at any given time due to recursion.

Consider:

```plaintext
let
  function fact(n:int):int =
    if n = 0 then 1
    else n * fact(n - 1)
in
  fact(3)
end
```

**Step 1: Record for fact(3) pushed on stack.**

<table>
<thead>
<tr>
<th>Stack Frame for fact</th>
<th>n=3</th>
</tr>
</thead>
</table>

**Step 2: Record for fact(2) pushed on stack.**

<table>
<thead>
<tr>
<th>Stack Frame for fact</th>
<th>n=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Frame for fact</td>
<td>n=2</td>
</tr>
</tbody>
</table>

**Step 3: Record for fact(1) pushed on stack.**

<table>
<thead>
<tr>
<th>Stack Frame for fact</th>
<th>n=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Frame for fact</td>
<td>n=2</td>
</tr>
<tr>
<td>Stack Frame for fact</td>
<td>n=1</td>
</tr>
</tbody>
</table>

**Step 4: Record for fact(0) pushed on stack.**

<table>
<thead>
<tr>
<th>Stack Frame for fact</th>
<th>n=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Frame for fact</td>
<td>n=2</td>
</tr>
<tr>
<td>Stack Frame for fact</td>
<td>n=1</td>
</tr>
<tr>
<td>Stack Frame for fact</td>
<td>n=0</td>
</tr>
</tbody>
</table>

**Step 5:** Record for fact(0) popped off stack, 1 returned.

**Step 6:** Record for fact(1) popped off stack, 1 returned.

**Step 7:** Record for fact(2) popped off stack, 2 returned.

**Step 8:** Record for fact(3) popped off stack, 3 returned. Stack now empty.
In some functional languages (such as ML, Scheme), local variables cannot be stored on stack.

```ml
fun f(x) =
  let
    fun g(y) = x + y
  in
    g
  end

Consider:
- val z = f(4)
- val w = z(5)
```

Assume variables are stack-allocated.

---

**Functional Languages**

**Step 1:** `f(4)` called.
Frame for `f(4)` pushed, `g` returned, frame for `f(4)` popped.

![Stack Frame for f with x=4]

Stack empty.

**Step 3:** `z(5)` called

![Stack Frame for z with y=5]

Memory location containing `x` has been deallocated!

---

**Functional Languages**

Combination of nested functions and functions returned as results (higher-order functions):

- Requires local variables to remain in existence even after enclosing function has been returned.
- Activation records must be allocated on heap, not stack.

Concentrate on languages which use stack.
Stack Frame Organizations

How is data organized in stack frame?

- Compiler can use any layout scheme that is convenient.
- Microprocessor manufactures specify “standard” layout schemes used by all compilers.
  - Sometimes referred to as Calling Conventions.
  - If all compilers use the same calling conventions, then functions compiled with
    one compiler can call functions compiled with another.
  - Essential for interaction with OS/libraries.

Typical Stack Frame

Stack Frame Example

Suppose \( f(a_1, a_2) \) calls \( g(b_1, b_2, b_3) \)

**Step 1:**

Previous Frame

Frame Pointer (FP): \( a_2 \)  
Frame for \( f \)

Stack Pointer (SP): \( \)  
Garbage

**Step 2:**

Previous Frame

Frame Pointer (FP): \( a_2 \)  
Frame for \( f \)

Stack Pointer (SP): \( b_1 \)  
Garbage
Suppose f(a1, a2) calls g(b1, b2, b3).

**Step 3:**

Dynamic link (AKA Control link) points to the activation record of the caller.
- Optional if size of caller activation record is known at compile time.
- Used to restore stack pointer during return sequence.

---

**Stack Frame Example**

Suppose f(a1, a2) calls g(b1, b2, b3), and returns.

**Step 4**

**Step 5**

---

**Parameter Passing**

\[ f(a_1, a_2, \ldots, a_n) \]

- Registers are faster than memory.
- Compiler should keep values in register whenever possible.
- Modern calling convention: rather than placing a1, a2, ..., an on stack frame, put a1, ..., a_k (k = 4) in registers \( f_{1,2}, f_{2,3}, f_{3,4} \) and \( a_{k+1}, a_{k+2}, a_{k+3}, \ldots, a_n \).
- If \( f_{1,2}, f_{2,3}, f_{3,4} \) are needed for other purposes, callee function must save incoming argument(s) in stack frame.
- C language allows programmer to take address of formal parameter and guarantees that formals are located at consecutive memory addresses.
  - If address argument has address taken, then it must be written into stack frame.
  - Saving it in "saved registers" area of stack won’t make it consecutive with memory resident arguments.
  - Space must be allocated even if parameters are passed through register.
If register argument has address taken, \textit{callee} writes register into corresponding space.

\begin{tabular}{l|l}
Frame Pointer (FP) & \\
\hline
a(0) & \\
a(k-1) & space for a(k) \\
\hline
Stack Pointer (SP) & \\
\hline
space for a(2) & space for a(1) \\
Garbage & \\
\hline
\end{tabular}

### Registers

	extbf{Registers hold:}

- Some Parameters
- Return Value
- Local Variables
- Intermediate results of expressions (temporaries)

	extbf{Stack Frame holds:}

- Variables passed by reference or have their address taken (&)
- Variables that are accessed by procedures nested within current one.
- Variables that are too large to fit into register file.
- Array variables (address arithmetic needed to access array elements).
- Variables whose registers are needed for a specific purpose (parameter passing)
- \textit{Spilled} registers. Too many local variables to fit into register file, so some must be stored in stack frame.

### Registers

Compilers typically place variables on stack until it can determine whether or not it can be promoted to a register (e.g. no references).

The assignment of variables to registers is done by the \textit{Register Allocator}. 
Registers

Register state for a function must be saved before a callee function can use them.

Calling convention describes two types of registers.
- **Callersave** register are the responsibility of the calling function.
  - Caller-save register values are saved to the stack by the calling function if they will be used after the call.
  - The callee function can use caller-save registers without saving their original values.
- **Callee-save** registers are the responsibility of the called function.
  - Callee-save register values must be saved to the stack by called function before they can be used.
  - The caller (calling function) can assume that these registers will contain the same value before and after the call.

Placement of values into callee-save vs. caller-save registers is determined by the register allocator.

Return Address and Return Value

A called function must be able to return to calling function when finished.
- Return address is address of instruction following the function call.
- Return address can be placed on stack or in a register.
- The *call* instruction in modern machines places the return address in a designated register.
- This return address is written to stack by callee function in non-leaf functions.

Return value is placed in designated register by callee function.

Frame Resident Variables

- A variable escapes if:
  - it is passed by reference,
  - its address is taken, or
  - it is accessed from a nested function.
- Variables cannot be assigned a location at declaration time.
  - Escape conditions not known.
  - Assign provisional locations, decide later if variables can be promoted to registers.
- *escape* set to true by default.
In languages that allow nested functions (e.g., Tiger), functions must access outer function’s stack frame.

```plaintext
let
  function f():int = let
    var a=5
  in
    function g(y:int):int = let
      var b=10
    in
      if y > 10 then g(y) <- b, a of outer fn
      else y + a + h(y)
    end
  in
    g(10)
end
in f() end
```

**Solution:**

Whenever `f` is called, it is passed pointer to most recent activation record of `g` that immediately encloses `f` in program text ⇒ Static Link (aka Access Link).

![Diagram](image.png)
- Need a chain of indirect memory references for each variable access.
- Number of indirect references = difference in nesting depth between variable declaration function and use function.