There's more to the activation record than a static link, of course.

- parameters
- local variables
- dynamic link (to be discussed)
- return address (""")

These items are generally accessed as offsets from a place in the A.R. pointed to by a register called the frame pointer (also known as the base pointer). *FP*

- There is also a stack pointer register that points to the "top" of the stack, for new data that is pushed onto the stack or data that is popped off the stack.

- The FP isn't absolutely necessary, but makes the compiler's job simpler.
Example layout

Everything is accessed via an offset from the FP.

- e.g. \[ \text{ARG1 is accessed as [FP+12]} \]
- \[ \text{ARG2 [FP+16]} \]
- \[ \text{LocalVar1 [FP-4]} \]
- \[ \text{STATICLink [FP+8]} \]
What is the dynamic link (DL)?
- just a pointer to the previous AR
- that is, the value of the FP when the previous function was executing

- really just used to be able to restore the FP when a function returns.

How is the dynamic link created?
- upon entry to function:
  
  ```
  Push FP
  Assign FP ← SP
  
  - now FP points to old value of ← FP.
  ```
Procedure Call — performed by the calling procedure

Passing Parameters:

Parameters are pushed onto the stack by the calling function.

- note that the parameters are often pushed in reverse order (at least in C compilers)

- why? So arg1 will be closest to FP
- why? Permits variable number of arguments (think of printf in C).

After parameters are pushed, the static link is computed and pushed.

Finally, the function is called with a call instruction which pushes the return address on the stack and jumps to the called procedure.

- return address is just the address of the instruction following the call in the calling procedure. ← fp

At this point, the stack looks like:

```
| Args | ← fp |
| SL   |     |
| Ret Addr | ← SP |
```
Procedure call (continued)

- \underline{Called procedure performs the following steps:}
  
  - pushes FP to create the dynamic link and set FP = SP
  
  - allocates space for local variables by subtracting the appropriate number from the SP
    (e.g. if there are five local variables that are each 4 bytes, subtract 20 from the SP)

To summarize procedure call:

\textbf{Calling-fn:}

\begin{itemize}
  \item push argN
  \item push arg1
  \item push SL
  \item call called-fn
\end{itemize}

\textbf{called-fn:}

\begin{itemize}
  \item push FP
  \item set FP = SP
  \item set SP = SP - K
\end{itemize}
Procedure Return - In Called Function

Essentially, pop off local variables or temporary data, reset the FP to the old value (the dynamic link), and execute the return instruction (RET).

Then, in calling function, pop the SL off the stack and then remove the arguments
- that is, if the SL is 4 bytes and there are 4 arguments that are each 4 bytes, increment SP by 20 (= 16 + 4).

To summarize procedure return:

called_fn:

\[\text{pops locals off stack} \rightarrow \text{set } sp = \text{FP}\]
\[\text{restore old FP} \rightarrow \text{pop FP}\]
\[\Rightarrow \text{RET}\]

return instruction pops return address off stack and jumps to it.

calling_fn:

\[\text{set } sp = sp + L\]

(where L is the total space occupied by the SL and the args)
Saving registers

When a procedure call occurs, it may be necessary to save the contents of registers (typically by pushing them onto the stack)

- Some registers are saved by the calling function ("caller-saved registers")

- Some are saved by the called function ("callee-saved registers")

The convention that a compiler uses to determine which registers are saved by whom and what gets pushed on the stack for a procedure call is called the **CALLING CONVENTION**.

- The rules governing where the args, SL, local variables, etc. can be found in the activation record.
The C programming language doesn't have nested procedures - so no SL needed.

AR looks like

```
argN
arg1
RET
DL
Local Vars
```

So, the offsets are different:

- `arg1`: `[FP+8]` (these are different than before)
- `arg2`: `[FP+12]`
- `LocalVar1`: `[FP-4]`

Your compiler won't generate static links, so the offsets will be like in C.
Return Values

Where return values are put are part of the calling convention.

In the GCC compiler for Intel x86:

- 4 byte return values are put in a register (EAX)

- All larger return values (e.g. records) are put on the stack.
  1) The calling function allocates space on the stack for the return value.
  2) Then the arguments are pushed.
  3) Then a pointer to the return value space (above) is pushed. - this pointer is an "extra argument"

SEE CODE SAMPLES ON WEB PAGE