Another motivation for macros: avoiding the cost of function calls.

Again, C++ provides an alternative:

***In-line* expansion**:

If `f` is defined with an in-line specifier:

```
inline int f(int x){return x+3;}
```

then during compilation of calls of `f`,

`f(`*e*`)` is compiled like `(`*e*`)+3`

Consequences:

- the run-time cost of a function call is eliminated
- code optimization not hindered by call/return branches

Note that `inline` is (only) a **recommendation** to the compiler.

In-line expansion can replace lots of *macro*-expansion.

`inline`'s advantages (compared to macros):

1. types are checked properly
2. names' scopes are handled statically

A needed improvement:

`inline` should apply to *individual calls* rather than to *definitions*.

(but this limitation is not hard to program around)

Lifetimes of variables

In conventional languages from Algol–60 through Pascal and C to Oberon, C++, Java, and Haskell,

- a **global** variable's lifetime
  = the duration of its computation
- each **local** variable or parameter's lifetime
  - *begins* when control *enters* its scope (e.g., when its procedure is called)
  - *ends* when control *leaves* its scope (e.g., when its procedure returns)
- space allocation for locals is LIFO ⇒ stack

  Example:

time:	0	1	2	3	4	5	6	7	8
	P	P	P	P	P	P	P	P	P
		Q	Q	Q		Q	Q	Q	
			R				S		

- between lifetimes, a local occupies no space (this was the original motivation)
- locals' values are not retained between lifetimes
- Exception: **`static`** variables (C/C++ only)
  - accessibility: local
  - lifetime: global

A consequence of the LIFO allocation model is...

Recursion support

A procedure can call *itself* (directly or indirectly).

Example:

time:	0	1	2	3	4	5	6	7	8
	P	P	P	P	P	P	P	P	P
		Q	Q	Q	Q	Q	Q	Q	
			P	P	P	P	P		
				Q	Q	Q			
					P				

Each invocation of a procedure gets its own set of local variables.

Global variables, however, are shared among all of the invocations.

Implementation of procedures in C and Pascal

Scope: variables can be declared either

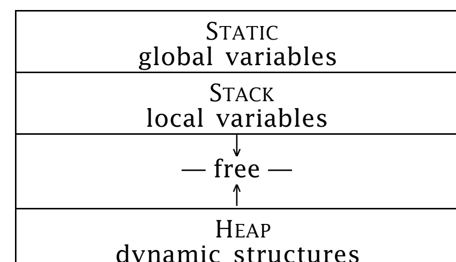
- global or
- local to ***any*** compound statement
  (including —but not limited to— function bodies).

```
if (E )
  {  int c[1000]; ...  }
else
  {  double d[300], e[200]; ...  }
```

syntax ⇒ compound statements are
either *disjoint*
or *strictly nested* (like procedure activations).

∴ storage for all variable declarations can be handled by stack allocation.

Where is "the stack"? Typical storage layout:

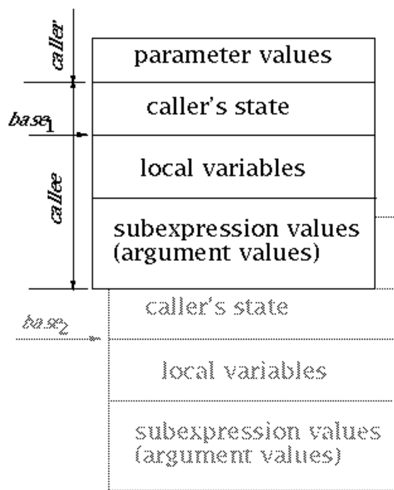


Procedure calls in Pascal/C/C++/Java/...

Typical implementation:

Each procedure activation generates an **Activation Record** (a.k.a. "frame") which is a contiguous block of storage

- taken from **freespace**
- "pushed" (by adjusting a pointer) onto the **stack**



The compiler translates...

a local-variable reference → offset (Δ address) (for each procedure, the offset for its first local variable is 0)

a parameter reference → offset<0

At procedure invocation:

1. call*er* evaluates arguments, leaving values on stack
2. call*er* saves minimal state information (including *base*, *pc*)
3. call*ee* saves other state information, requests frame allocation from storage manager (if refused, *abort*)
4. call*ee*: $base := base + \Delta\, base$ ($\Delta\, base$ is precomputed by the compiler)
5. call*ee* body executes; address of local-variable$_i$ = $base + offset_i$

At procedure return:

[1. callee copies return value over first parameter value]

2. callee restores state info saved in (3)
3. callee restores *base*, *pc*.
4. caller resumes execution

Recursion

Recursive calls:

Each invocation gets its own activation record, and ∴ its own copies of local variables.

Tail recursion: suppose

$g \to f_0 \to f_1 \to f_2$

and

f's **last** operation is another invocation of f.

Then

0. the return from f_2 is followed immediately by the return from f_1, and then by the return from f_0; f_2 might as well return directly to g;
1. the value returned by f_2 to f_1 is returned to f_0 and finally to g; since f_1 and f_0 do nothing to the value, f_2 might as well return it directly to g.

To save activation-record space:

Use the *same record* for f_0, f_1, f_2.

How? By translating **tail recursion** to **iteration**.

Translating **tail recursion** to **iteration**

Example:

Recursive

```
int f(int x, int y)
{
 if(x==0) return y;
 return f(x-1,y+1);
}

...
z := f(3,2);
```

Iterative

```
int f(int x, int y)
{
 L: if(x==0) return y;
    x:=x-1; y:=y+1;
    goto L;
}
...
z := f(3,2);
```

Either way, x and y are **bound** to new values before each execution of *body*.

Advantages of iteration over recursion:

1. saves **time** for setting up and reclaiming activation records
2. shrinks **space** requirements from *linear* (with respect to number of iterations) to *constant*

For a bigger example, see Sethi, Example 5.22.