

## Subprograms

---

Subprograms are a construct for giving a name to a piece of coding; the piece is referred to as the subprogram *body*.

- Subprogram *call* is the explicit request that the subprogram body be executed.
- Each execution of the body is called *activation* of the body.
- Functions and Procedures.
- Subprogram *header*: First line of the definition.
  - define start using special word.
  - provides a name for the subprogram.
  - may specify a list of parameters.
  - can itself be a subprogram definition.
- Can access data two ways: direct access to nonlocal variables; parameter passing.

### Parameters : Formal and Actual

- Positional parameters : binding of actual parameters to formal parameters is done by simple position.
- Keyword parameters : name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.

## The procedure abstraction

---

Separate compilation:

- allows us to build large programs
- keeps compile times reasonable
- requires independent procedures

The linkage convention:

- a social contract
- machine dependent
- division of responsibility

The linkage convention ensures that procedures inherit a valid run-time environment *and* that they restore one for their parents.

Linkages execute at *run time*

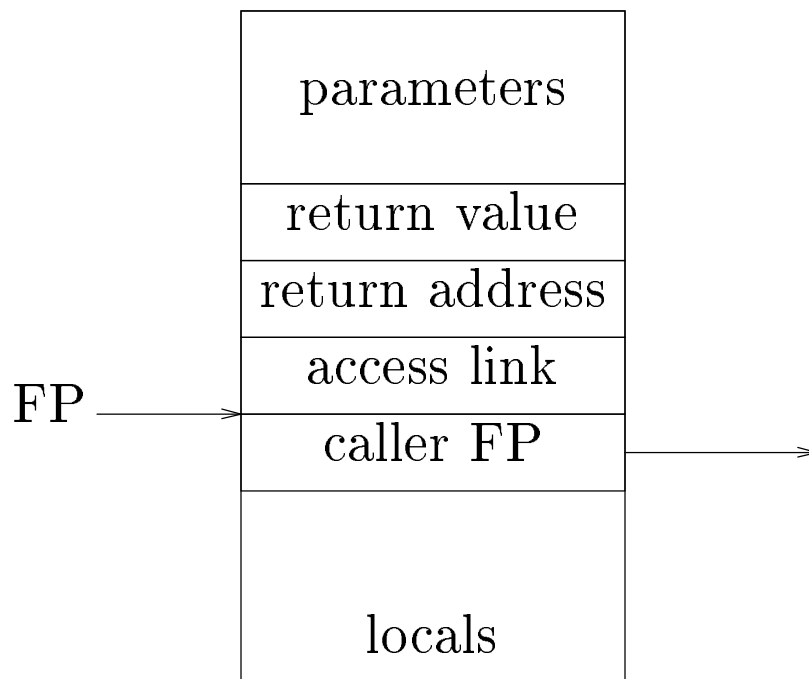
Code to make the linkage is generated at *compile time*



## Procedure linkages

---

Assume that each procedure activation has an associated *activation record* or *frame* (at run time)



Assumptions:

- call by reference parameter passing
- RISC architecture
- can always expand an allocated block
- locals stored in frame

## Procedure linkages

---

The linkage divides responsibility between *caller* and *callee*

	Caller	Callee
Call	<i>call sequence</i>	<i>prolog code</i>
	allocate basic frame evaluate & store params. store return address store FP set FP for child jump to child	save registers, state extend basic frame (for local data) find static data area initialize locals fall through to code
Return	<i>return sequence</i>	<i>epilog code</i>
	copy return value deallocate basic frame restore params. (?)	store return value restore state unextend basic frame restore parent's FP jump to return address

*At compile time, we generate the code to do this*

*At run time, that code manipulates the frame & data areas*

## Run-time storage organization

To maintain the illusion of procedures, the compiler must adopt some conventions to govern memory use.

### Code space

- fixed size
- statically allocated *(link time)*

### Data space

- fixed size data may be statically allocated
- variable size data must be dynamically allocated
- some data is dynamically allocated in code

### Control stack

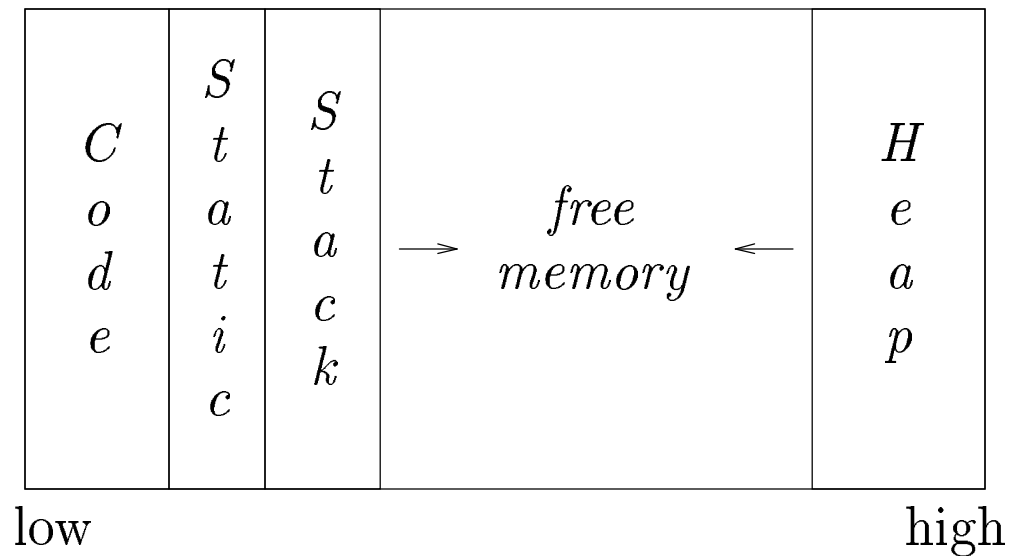
- dynamic slice of activation tree
- return addresses
- usually implemented in hardware

# Run-time storage organization

---

## Typical memory layout

### Logical Address Space



### The classical scheme

- allows both stack and heap maximal freedom
- code and static may be separate or intermingled

## Run-time storage organization

---

Where do local variables go?

When can we allocate them on a stack?

*Key issue is lifetime of local names*

Downward exposure:

- called procedures may reference my variables
- dynamic scoping
- lexical scoping

Upward exposure:

- can I return a reference to my variables?
- functions that return functions

With only *downward exposure*, the compiler can allocate the frames on the run-time stack

## Run-time storage organization

Each variable must be assigned a storage class  
(*base address*)

Static variables:

- addresses compiled into code (*relocatable*)
- (*usually*) allocated at compile-time
- limited to fixed size objects
- control access with naming scheme

Global variables:

- almost identical to static variables
- layout may be important (*exposed*)
- naming scheme ensures universal access

Linkage editor must handle duplicate definitions

## Run-time storage organization

---

Storage classes (*con't*):

Procedure local variables

*Put them on the stack* —

- *if* sizes are fixed
- *if* lifetimes are limited
- *if* values are not preserved

Dynamically allocated variables

*Must be treated differently* —

- pointers, lead to non-local lifetimes
- (*usually*) an express allocation
- express or implicit deallocation

## Access to non-local data

How does the code find non-local data at *run-time*?

### Real globals

- visible *everywhere*
- naming convention gives an address
- initialization requires cooperation

### Lexical nesting

- view variables as  $(level, offset)$  pairs  
(*compile-time*)
- chain of non-local access links
- more expensive to find (*at run-time*)

## Access to Non-local Data

---

Two important problems arise

*How do we map a name into a (level,offset) pair?*

use a block structured symbol table

- look up a name, want its most recent declaration  
the name
- declaration may be at current level or any lower  
level

*Given a (level,offset) pair, what's the address?*

Two classic approaches

⇒ access links *(static links)*

⇒ displays

## Access to Non-local Data

---

To find the value specified by  $(l, o)$

- need current procedure level,  $k$
- if  $k = l$ , is a local value
- if  $k > l$ , must find  $l$ 's activation record
- $k < l$  cannot occur

Maintaining access links: *(static links)*

- calling level  $k + 1$  procedure
  1. pass my FP as access link
  2. my backward chain will work for lower levels
- calling procedure at level  $l < k$ 
  1. find my link to level  $l - 1$  and pass it
  2. its access link will work for lower levels

## The display

---

To improve run-time access costs, use a *display*.

- table of access links for lower levels
- lookup is index from known offset
- takes slight amount of time at call
- a single display or one per frame
- for level  $k$  procedure, need  $k - 1$  slots

Access with the display

*assume a value described by  $(l, o)$*

- find slot as  $DP + 4 \times l$
- add offset to pointer from slot

“setting up the base frame” now includes display manipulation.



## Display management

---

Single global display:

*simple method*

*Key insight* – overallocate the display by 1 slot

*on entry to a procedure at level  $l$*

save the level  $l$  display value

push  $\text{FP}$  into level  $l$  display slot

*on return*

restore the level  $l$  display value

*Quick, simple, and foolproof!*

## Display management

---

Individual frame-based displays:

*call from level  $k$  to level  $l$*

if  $l \leq k$

copy  $l - 1$  display entries into child's frame

if  $l > k$       ( $l = k + 1$ )

copy  $k - 1$  entries into child's frame

copy own FP into  $k^{th}$  slot in child's frame

*no work required on return*

$\Rightarrow$  display is deallocated with frame

*Display accessed by offset from FP.*

$\Rightarrow$  one less register required

## Display versus access links

---

How to make the trade-off?

*The cost differences are somewhat subtle*

- frequency of non-local access
- average lexical nesting depth
- ratio of calls to non-local access

(Sort of) Conventional wisdom

*tight on registers*                     $\Rightarrow$  use access links

*lots of registers*                     $\Rightarrow$  use global display

*shallow average nesting*  $\Rightarrow$  frame-based display

*Your mileage will vary*

*Making the decision requires understanding reality*

## Parameter passing

---

### *Call-by-value (in)*

- store values, not addresses
- never restore on return
- arrays, structures, strings are a problem

### *Call-by-result (out)*

- do not store values or addresses
- always restore on return
- arrays, structures, strings are a problem

### *Call-by-value-result (in-out)*

- store values, not addresses
- always restore on return
- arrays, structures, strings are a problem

### *Call-by-name*

- build and pass *thunk*
- access to parameter invokes *thunk*
- all parameters are same size in frame!

### *Call-by-reference (in-out)*

- pass the access path (addresses)
- passing is efficient, access is slow
- efficient for arrays, structures, strings

## Parameter passing

---

What about variable length argument lists?

1. if *caller* knows that *callee* expects a variable number
  - (a) *caller* can pass number as 0<sup>th</sup> parameter
  - (b) *callee* can find the number directly
2. if *caller* doesn't know anything about it
  - (a) *callee* must be able to determine number
  - (b) first parameter must be closest to FP

Consider `printf` :

- number of parameters determined by the format string
- it assumes the numbers match

## Parameter passing

---

### Parameter passing in Major Languages

- Fortran – Reference
- Algol 60 – name; value as option
- Algol 68, C – value; send ptr. to do reference
- Pascal, Modula-2 – value; get ref. by adding **var**
- Ada – in, out, in-out

### Type-checking of Parameters :

- Most languages do it
- Old C did not; ANSI-C does but there are ways around it.
- C++ has type checking but can work around it.

### Passing functions :

- Hard for the compiler to check consistency
- Algol 68 or Pascal consistency checking made easier by complete description of function
- C and C++ functions cannot be passed but pointers to functions can be
- Ada does not allow function passing but the generic facility can be used to get the functionality
- Shallow binding and deep binding

## More Procedures

---

Overloaded subprogram has the same name as another subprogram in the same referencing environment.

- distinction done in terms of parameters or return types.
- C++ and Ada allows this.
- Ada PUT function takes string, integer and floating-point.
- Can be user defined functions.

Generic or polymorphic subprogram is one that takes parameters of different types on different activations.

- Ada provides parametric polymorphism through the use of **generic units**.
- Different version of the subprograms constructed by the compiler.
- C++ provides this functionality through the use of templated functions.