

Where are we now

Front end

- scanner (lexical analysis)
- parser (syntactic analysis)
- type checker (semantic analysis)

Back end

- intermediate representation
- code generation
- code optimization

Run-time environment

- procedure abstraction
- run-time storage management

We need to understand how a program executes at run time before we can generate code for it

The procedure abstraction

Procedures provide many benefits

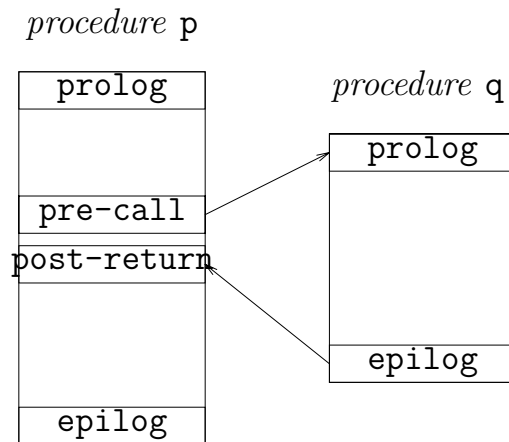
1. control abstraction
 - well defined entry, exits
 - mechanism to pass parameters, return values
2. name space
 - new name space within procedure
 - local names are protected from outside
3. external interface
 - accessed by procedure name, parameters
 - protection for both caller and callee
 - enables software libraries, systems
4. separate compilation
 - compile procedures independently
 - keeps compile times reasonable
 - allows us to build large programs

Procedure linkages

The *linkage convention* is the interface used for performing procedure calls

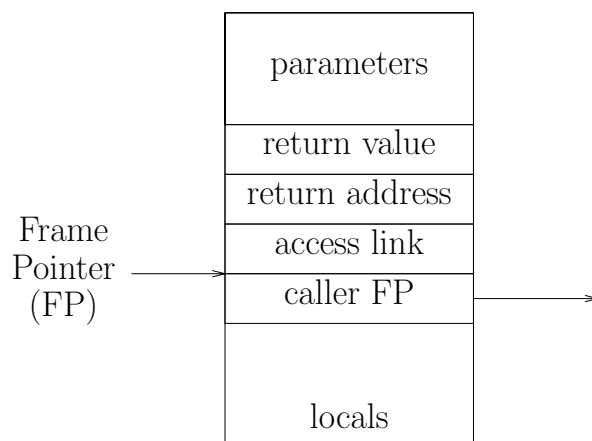
- *on entry*, establish **p**'s environment
- *at a call*, preserve **p**'s environment
- *on exit*, tear down **p**'s environment
- *in between*, handle addressability and lifetimes

Ensures each procedure inherits valid run-time environment and also restores one for its caller



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

Name spaces

Scoping

- scope – a name space which maps a set of names to a set of variables
- lexical scope – scope defined by location in program text
- nested lexical scope – in a scope, each name refers to its lexically closest declaration

During compilation

- *current scope* - the innermost scope for the current component
- *open scope* - all scopes surrounding the current scope
- *closed scope* - all other scopes

Only variables declared in current or open scopes are visible

Scopes allow compiler to map name to memory locations

Nested lexical scoping

Example

```
Procedure A
  H1, I, J: integer
  begin
    Procedure B
      X, Y: real
      begin
        ...
      end
    Procedure C
      H2, L, M: integer
      begin
        ...
      end
  end
```

What names are visible in Procedure B?

What names are visible in Procedure C?

Name spaces of languages

LISP

- dynamic scoping
- most recently invoked procedure

Algol, Pascal, Modula

- nested lexical scoping
- procedures nested within procedures

Fortran 77

- global scope – procedures, common blocks
- local scope – variables, parameters in procedures

C

- global scope – procedures, external variables
- file scope – variables declared in file
- procedure scope – local variables
- nested block scope – within { }

Java

- global scope – public classes
- package scope – fields, methods within package
- procedure scope – local variables
- nested block scope – within { }

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

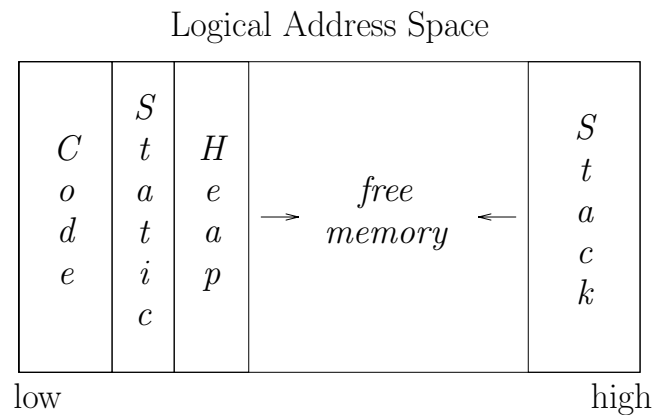
- fixed size data may be statically allocated
- variable size data must be dynamically allocated
- some data is dynamically allocated in code

Control stack

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

Run-time storage organization

Typical memory layout



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 (e.g., pass by reference)
- dynamic scoping
- lexical scoping

Upward exposure:

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

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*)

Global variables (*static*)

- addresses assigned (relocated) at link-time
- limited to fixed size objects
- naming scheme ensures universal access, handles duplicates

Local variables (*stack*)

- addresses compiled into code
- *if* sizes are fixed
- *if* lifetimes are limited

Dynamically allocated variables (*heap*)

- addresses assigned at run time
- call-by-reference, pointers, lead to non-local lifetimes
- explicit allocation, explicit/implicit deallocation

Access to non-local data

How does the code find data at *run time*?

Global variables

- visible *everywhere*
- linker chooses address
- initialization requires cooperation

Local variables

- stored on stack in frame
- need to determine offset in frame

Lexical scoping

- may access variables declared in enclosing scopes
- must generate code to calculate address (*at run-time*)
- may use *access links* or *displays*

Heap management

Functionality:

- **alloc**(k) — locates a block of *at least* k bytes in the free space pool, removes it from the pool, and returns its address
- **free**(p) — places the block pointed to by p back in the pool of free space for allocation

If unused storage is reclaimed, **free** is not needed.

Potential problems:

- wasted space — if **alloc** returns blocks that are larger than requested, the excess space is wasted
- fragmentation — after a series of **alloc** and **free** commands, the free space pool becomes fragmented, preventing allocation of large blocks
- speed — **alloc** and **free** should be inexpensive

A heap management scheme must balance these issues.

Knuth, Volume 1, §2.5

Garbage collection

Approach

- automatically free unused memory
- no pointer to a block → done with it
- deallocate when we run out of space
- standard in high-level languages

Advantages

- simplifies programmer's life
- avoids dangling references to “free” memory
- avoids memory leaks

Disadvantages

- may be costly to perform
- may require program to halt during collection
- must be conservative
- may not find all free memory
(e.g., pointer arithmetic in C)

Garbage collection algorithms

Reference counting

- concurrent with program execution
- record # references to each piece of memory
- when count reaches zero, reclaim memory
- reclaiming memory may cause more collection

Mark and scan

- halt program during collection
- start with pointers in registers, frames
- mark all accessible data, free rest

Copying collector

- either concurrent or halt program
- copies all live data to new area, free rest
- update pointers where necessary
- compacts useful memory