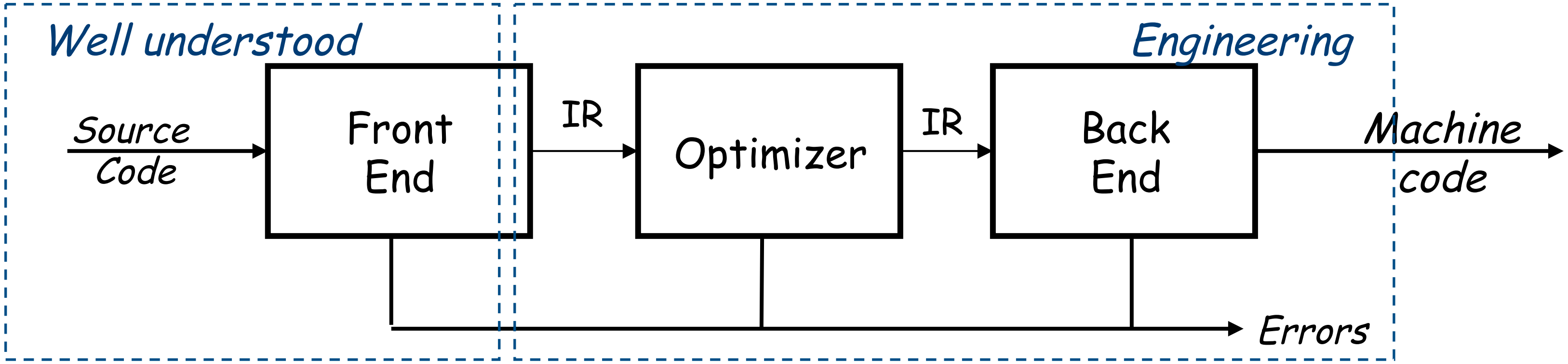
Chapters 6 and 7 of EaC explore techniques that compilers use to implement various language features.

The Procedure Abstraction

Note by Baris Aktemur:

Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

Where are we?



The latter half of a compiler contains more open problems, more challenges, and more gray areas than the front half

- This is "compilation," as opposed to "parsing" or "translation"
- Implementing promised behavior
  - Defining and preserving the meaning of the program
- Managing target machine resources
  - Registers, memory, issue slots, locality, power, …
  - These issues determine the quality of the compiled code

Conceptual Overview

The compiler must provide, for each programming language construct, an implementation (or at least a strategy).

Those constructs fall into two major categories

- Individual statements
- Procedures

We will look at procedures first, since they provide the surrounding context needed to implement statements

Object-oriented languages add some peculiar twists

- We will treat OOL features in a separate lecture or two

In EaC, Chapter 6 covers implementation of procedures & Chapter 7 covers statements.

Conceptual Overview

Procedures provide the fundamental abstractions that make programming practical & large software systems possible

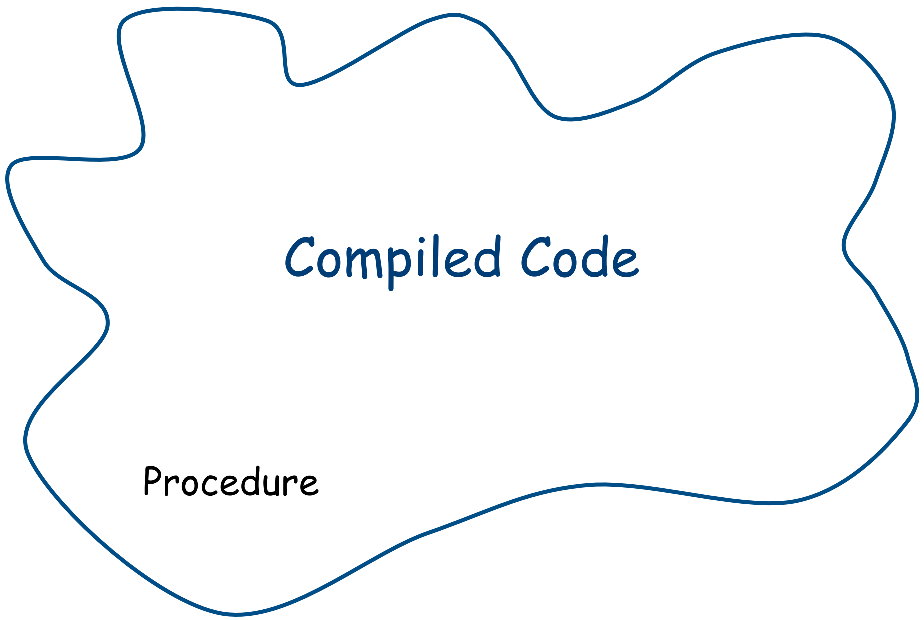
- Information hiding
- Distinct and separable name spaces
- Uniform interfaces

Hardware does little to support these abstractions

- Part of the compiler's job is to implement them
  - *Compiler makes good on lies that we tell programmers*
- Part of the compiler's job is to make it efficient
  - *Role of code optimization*

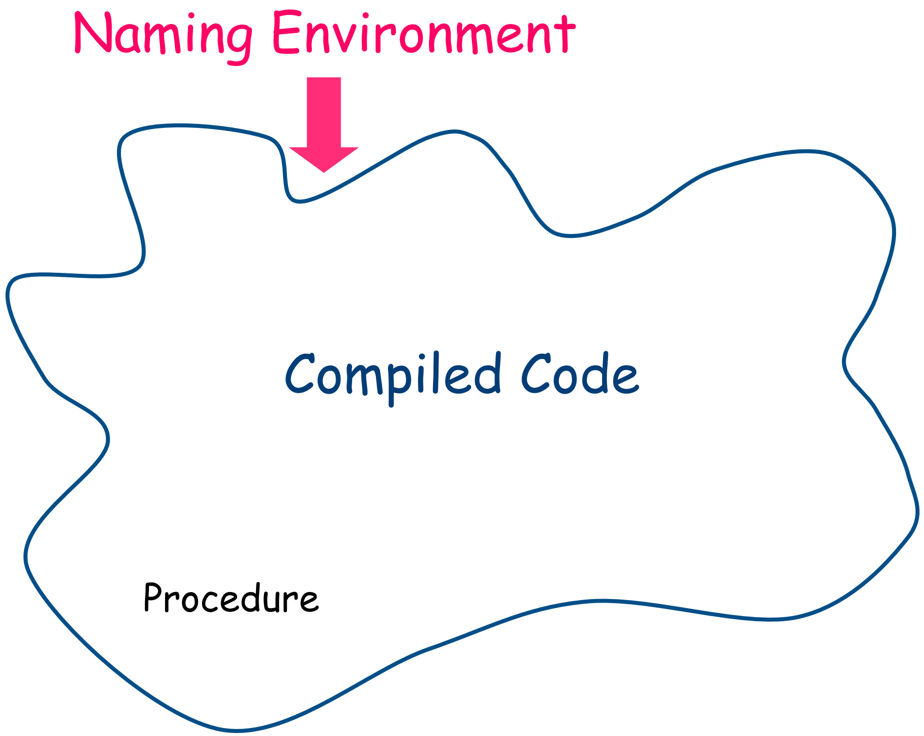
The Procedure & Its Three Abstractions

The compiler produces code for each procedure



The individual code bodies must fit together to form a working program

The Procedure & Its Three Abstractions



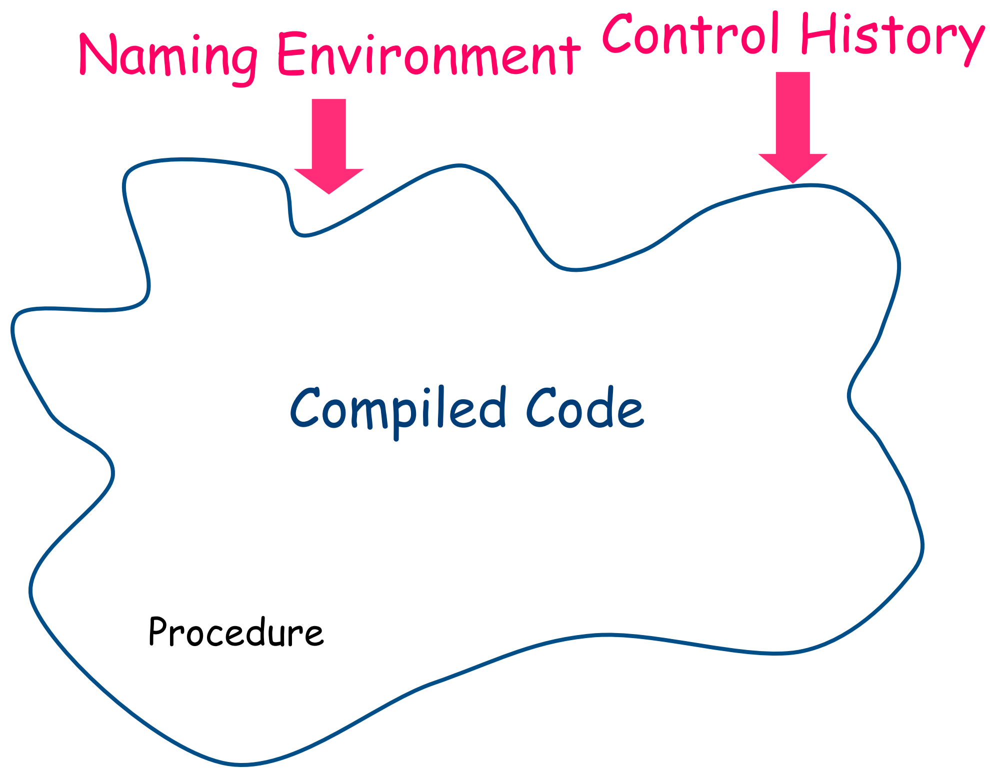
"Naming" includes the ability to find and access objects in memory

Each procedure inherits a set of names

⇒ Variables, values, procedures, objects, locations, …

⇒ Clean slate for new names, "scoping" can hide other names

The Procedure & Its Three Abstractions



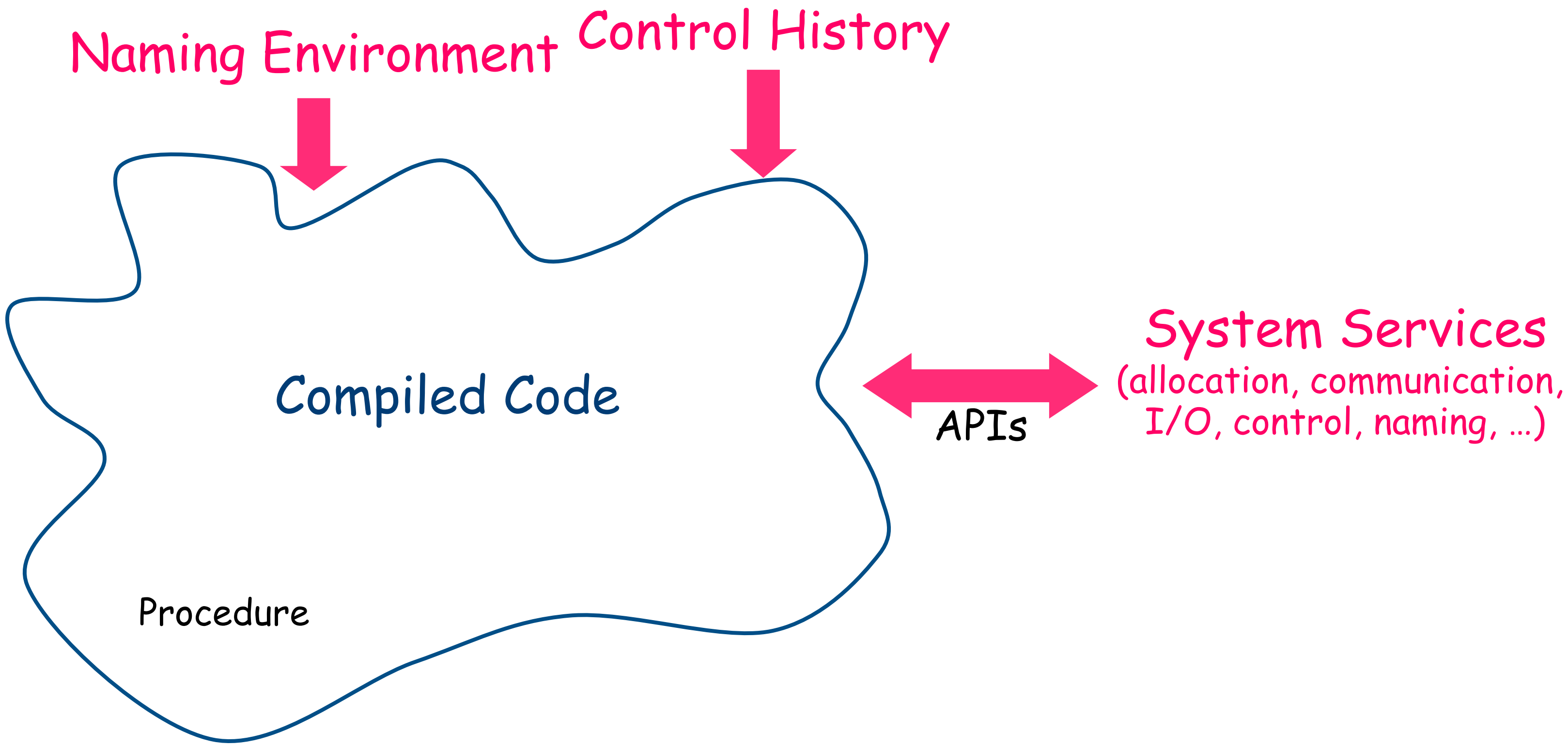
Each procedure inherits a control history

⇒ Chain of calls that led to its invocation

⇒ Mechanism to return control to caller

Some notion of parameterization (ties back to naming)

The Procedure & Its Three Abstractions



Each procedure has access to external interfaces

⇒ Access by name, with parameters *(may include dynamic link & load)*

⇒ Protection for both sides of the interface

The Procedure (More Abstract View)

A procedure is an abstract structure constructed via software

Underlying hardware directly supports little of the abstraction—it understands bits, bytes, integers, reals, & addresses, but not:

- Entries and exits
- Interfaces
- Call and return mechanisms
  - Typical machine supports the transfer of control (call and return) but not the rest of the calling sequence (*e.g.*, preserving context)
- Name space
- Nested scopes

All these are established by carefully-crafted mechanisms provided by compiler, run-time system, linker, loader, and OS;

The compiler's job is to make good on the lies told by the programming language design!

Run Time versus Compile Time

These concepts are often confusing to the newcomer

- Linkages (*and code for procedure body*) execute at run time
- Code for the linkage is emitted at compile time
- The linkage is designed long before either of these

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int  d;
  d = q(c,b);
  …
}
```

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

- Most languages allow recursion

The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Requires code to save and restore a "return address"
- Must map actual parameters to formal parameters ($c \rightarrow x$, $b \rightarrow y$)
- Must create storage for local variables (&, maybe, parameters)
  - *p* needs space for *d* (&, maybe, *a*, *b*, & *c*)
  - where does this space go in recursive invocations?

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

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

Compiler emits code that causes all this to happen at run time

The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Must preserve *p*'s state while *q* executes
- *Strategy:* Create unique location for each procedure activation
  - In simple situations, can use a "stack" of memory blocks to hold local storage and return addresses

```
...
s = p(10,t,u);
...
```

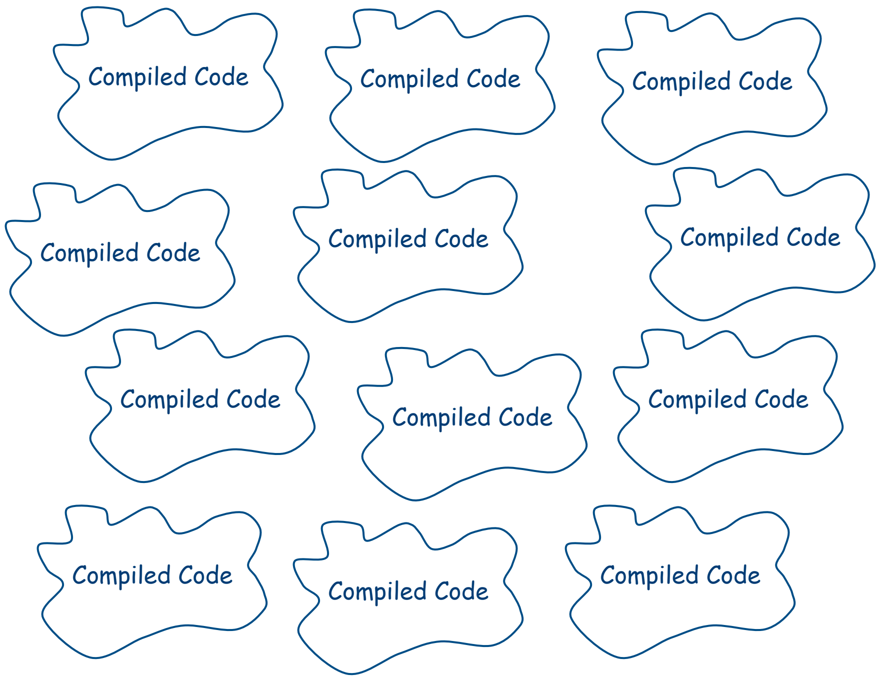
```
int p(a,b,c)
   int a, b, c;
{
   int   d;
   d = q(c,b);
   ...
}
```

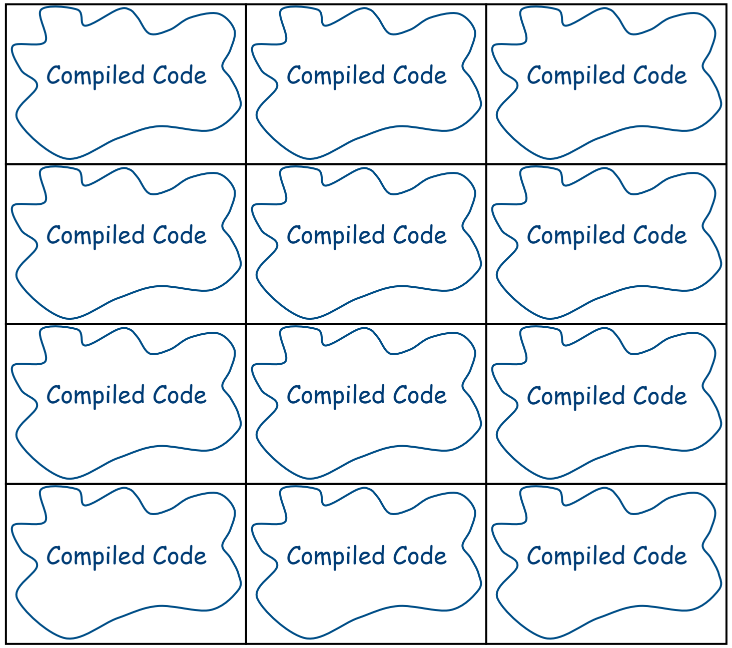
```
int q(x,y)
   int x,y;
{
   if ( ... )
      x = q(x-1,y);
   return x + y;
}
```

Compiler emits code that causes all this to happen at run time

The Procedure as a Control Abstraction

In essence, the procedure linkage wraps around the unique code of each procedure to give it a uniform interface





Similar to building a brick wall rather than a rock wall

The Procedure Abstraction: Part II

The Procedure as a Name Space

Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
- We call this set of rules & conventions "lexical scoping"

The Java twist: allow fully qualified names to reach around scope rules

Examples

- C has global, static, local, and *block* scopes *(Fortran-like)*
  - Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes *(let)*
  - Procedure scope (typically) contains formal parameters

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding "free" variables
- Simplifies rules for naming & resolves conflicts
- Lets the programmer introduce "local" names with impunity

How can the compiler keep track of all those names?

The Problem

- At point *p*, which declaration of *x* is current?
- At run-time, where is *x* found?
- As parser goes in & out of scopes, how does it delete *x*?

The Answer

- The compiler must model the name space
- Lexically scoped symbol tables (see § 5.7 in EaC 1e)

§ 5.7 in EaC 1e

Lexically-scoped Symbol Tables

The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface

- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Many implementation schemes have been proposed

- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & (yes) fun
  - Good alternatives exist

Symbol tables are compile-time structures that the compiler uses to resolve references to names.
We'll see the corresponding run-time structures that are used to establish addressability later.

Example

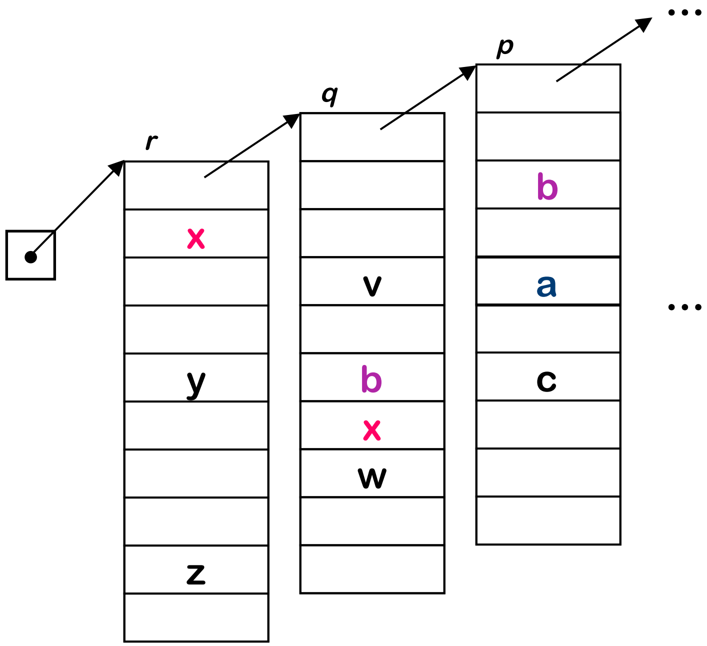
```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure  r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            ...
        }
        ... r ... s
    }
    ... q ...
}
```

```
B0: {
        int a, b, c
B1:      {
            int v, b, x, w
B2:          {
                int x, y, z
                ....
            }
B3:          {
                int x, a, v
                ...
            }
            ... r ... s
        }
        ... q ...
}
```

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



"Chain of tables" implementation

- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away level *p* table if it is top table in the chain

If the compiler must preserve the table (*for, say, the debugger*), this idea is actually practical.

Individual tables are hash tables.

This high-level idea can be implemented as shown, or it can be implemented in more space-efficient (albeit complex) ways.

Where Do All These Variables Go?

Automatic & Local

- Keep them in the procedure activation record or in a register
- Automatic ⇒ lifetime matches procedure's lifetime

Static

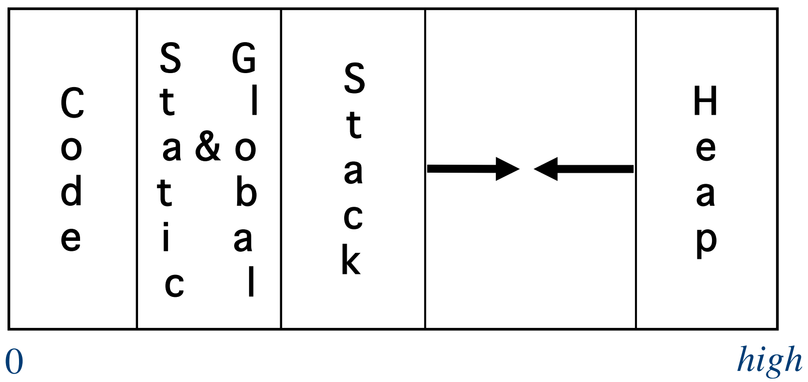
- Procedure scope ⇒ storage area affixed with procedure name
  - &_p.x
- File scope ⇒ storage area affixed with file name
- Lifetime is entire execution

Global

- One or more named global data areas
- One per variable, or per file, or per program, …
- Lifetime is entire execution

Placing Run-time Data Structures

Classic Organization



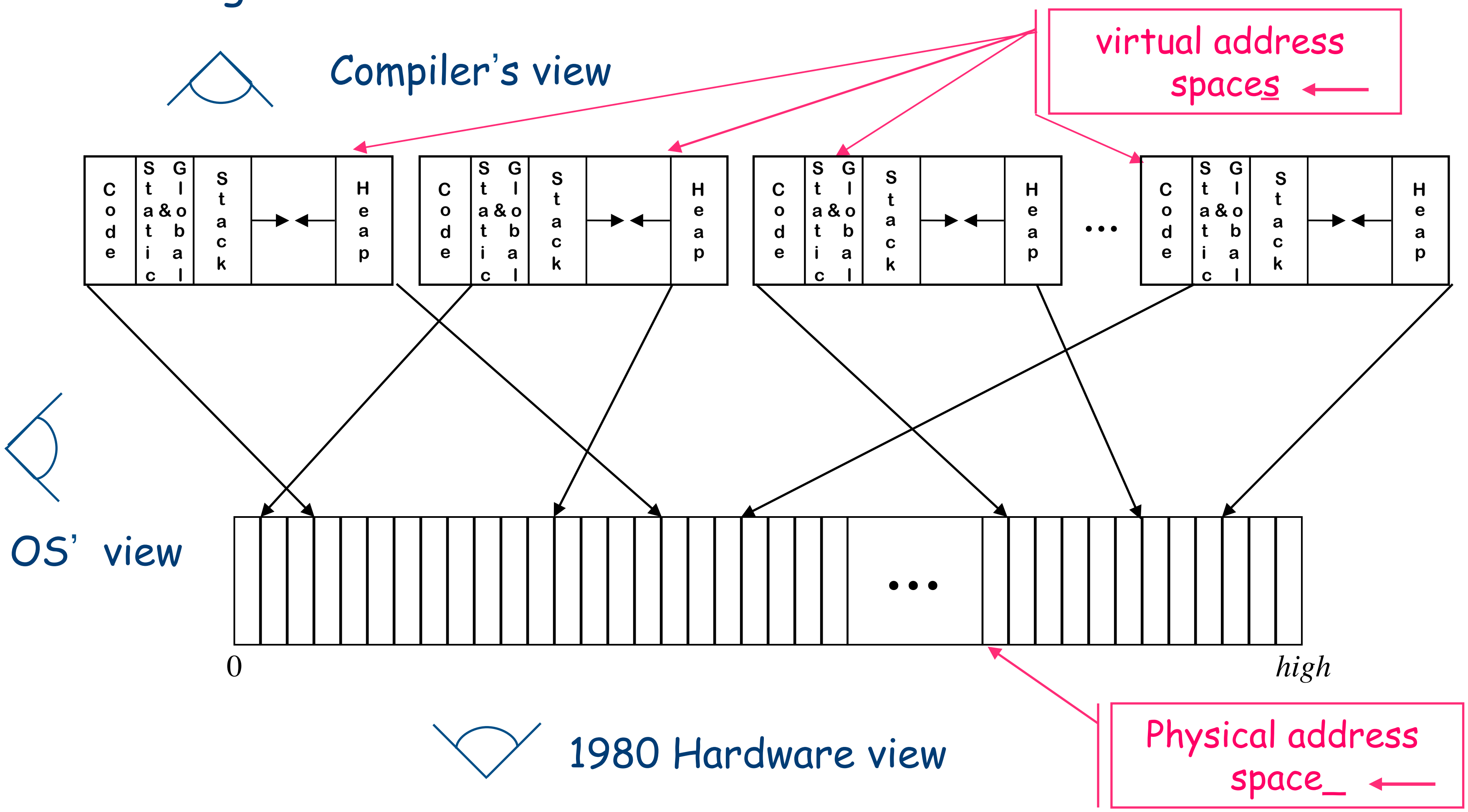
Single Logical Address Space

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

- Code, static, & global data have known size
  – Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a *virtual* address space

How Does This Really Work?

The Big Picture

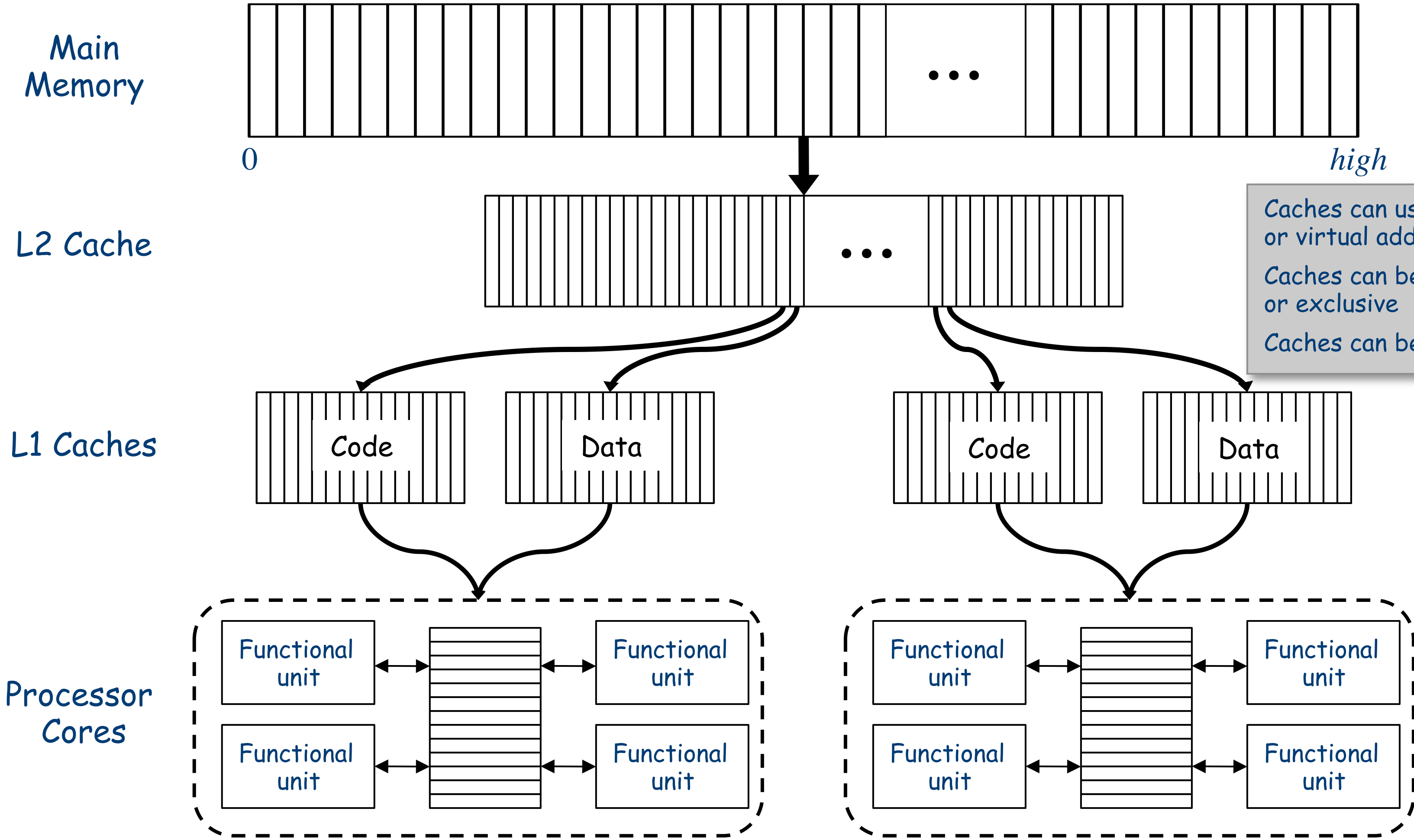


On many modern processors, L1 cache uses physical addresses
Higher level caches typically use virtual addresses

How Does This Really Work?

Most systems now include L3 caches. L4 is on its way.

Of course, the "Hardware view" is no longer that simple



Caches can use physical or virtual addresses

Caches can be inclusive or exclusive

Caches can be shared

Cache structure matters for performance, not correctness

Where Do Local Variables Live?

A Simplistic model
- Allocate a data area for each distinct scope
- One data area per "level" in scoped table

What about recursion?
- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there!

More complex scheme
- One activation record (AR) per procedure instance
- All the procedure's scopes share a single AR *(may share space)*
- Static relationship between scopes in single procedure

Used this way, "static" means knowable at *compile time* (and, therefore, fixed).

Translating Local Names

How does the compiler represent a specific instance of *x* ?

- Name is translated into a *static coordinate*
  - ‹ *level,offset* › pair
  - "*level*" is lexical nesting level of the procedure
  - "*offset*" is *unique* within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "*offset*" must be assigned and stored in the symbol table
  - Assigned at *compile time*
  - Known at *compile time*
  - Used to generate code that *executes* at *run-time*

Storage for Blocks within a Single Procedure

```
B0: {
        int a, b, c
B1:     {
            int v, b, x, w
B2:         {
                int x, y, z
                ...
            }
B3:         {
                int x, a, v
                ...
            }
            ...
        }
        ...
    }
```

Fixed length data can always be at a constant offset from the beginning of a procedure

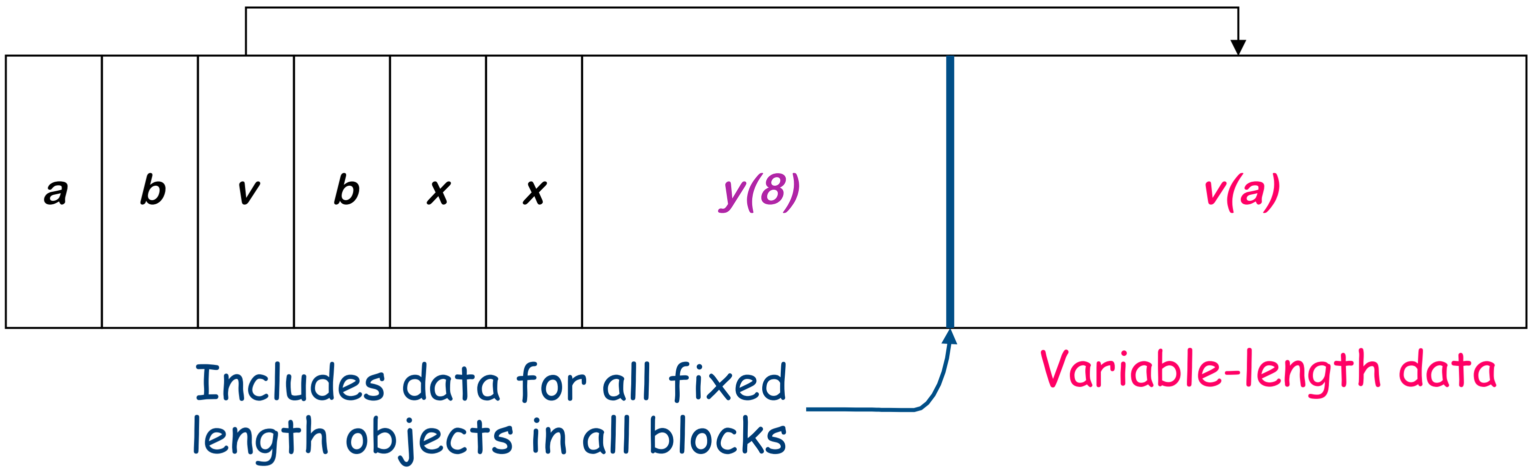
- In our example, the *a* declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area
- The *x* declared at level 1 will always be the sixth data item, stored at byte 20 in the fixed data area
- The *x* declared at level 2 will always be the eighth data item, stored at byte 28 in the fixed data area
- But what about the a declared in the second block at level 2?

Variable-length Data

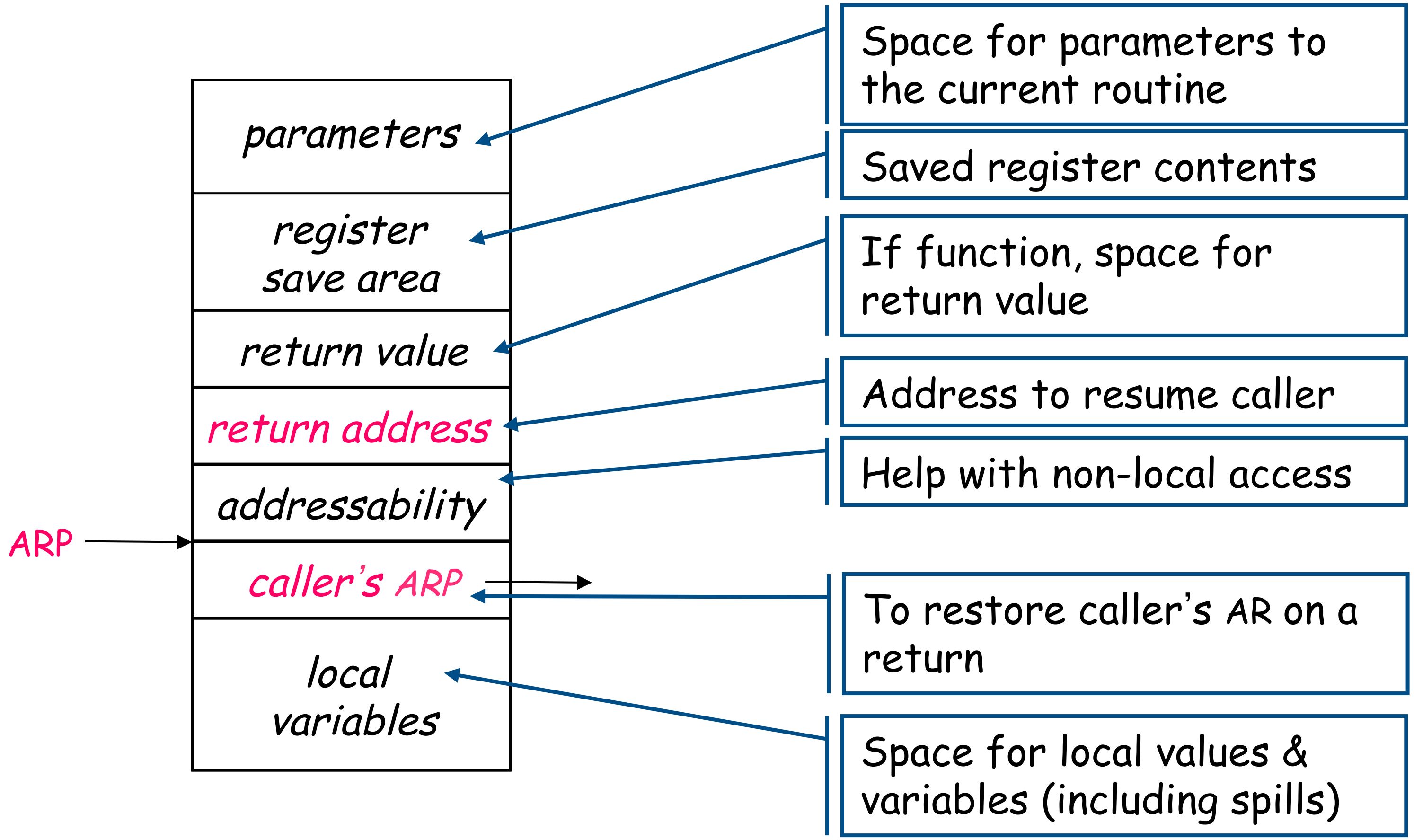
```
B0: {   int a, b
        ...
        assign value to a
        ...
B1:     {  int v(a), b, x
           ...
B2:        {  int x, y(8)
              ...
           }
        }
     }
```

Arrays

- → If size is fixed at compile time, store in fixed-length data area
- → If size is variable, store descriptor in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for the block in which it is allocated (including all contained blocks)



Activation Record Basics



One AR for each invocation of a procedure

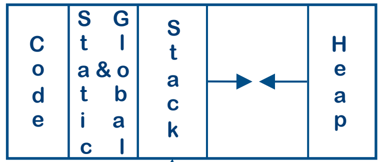
ARP ≈ Activation Record Pointer

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒ Keep ARs on a stack



Yes! That stack

- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs <u>must</u> be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap