The Procedure Abstraction
Part I: Basics

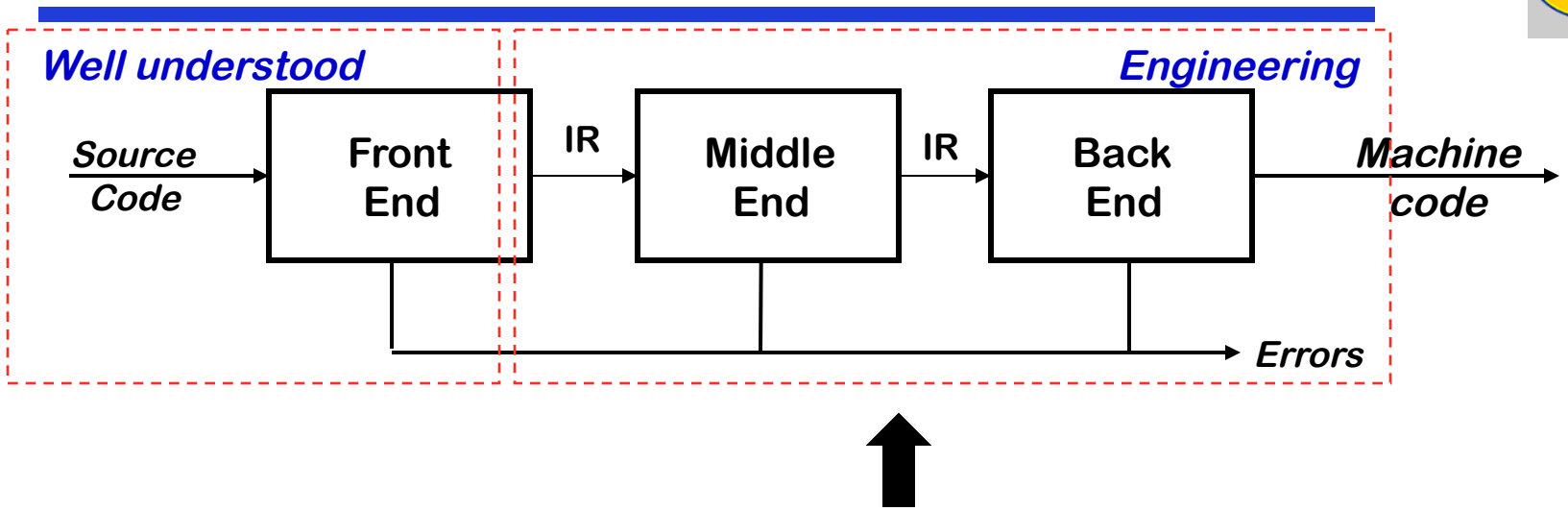
Procedure Abstraction

- Begins Chapter 6 in EAC
- The compiler must deal with interface between compile time and run time
  - → Most of the tricky issues arise in implementing "procedures"

Procedure Abstraction Issues

- Compile-time versus run-time behavior
- Finding storage for EVERYTHING and mapping names to addresses
- Generating code to compute addresses
- Interfaces with other programs, other languages, and the OS (libraries)
- Efficiency of implementation

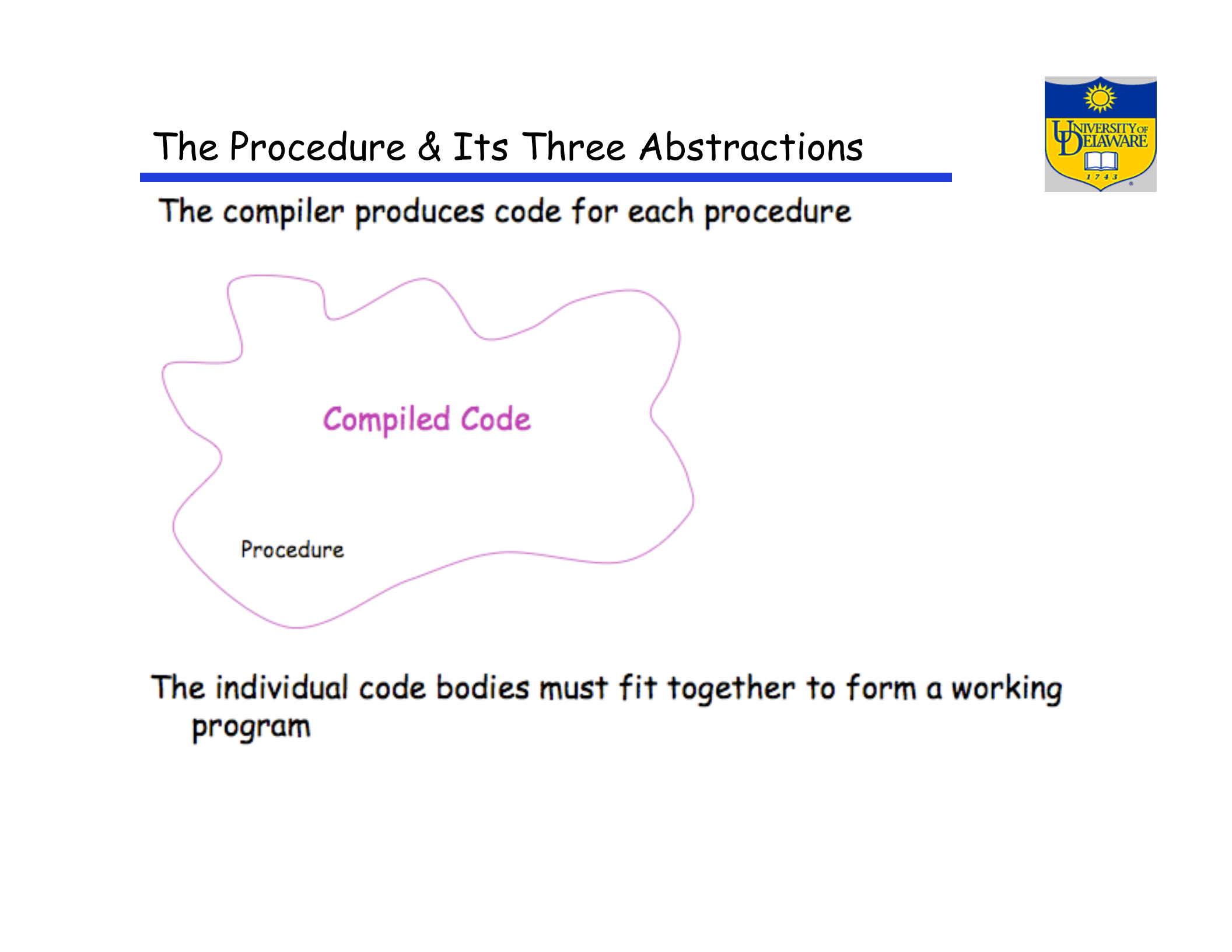
Where are we?



Contains more open problems and more challenges

- This is "compilation," as opposed to "parsing" or "translation"
- Implementing promised behavior
  - → What defines the meaning of the program
- Managing target machine resources
  - → Registers, memory, issue slots, locality, power, …
  - → These issues determine the quality of the compiler

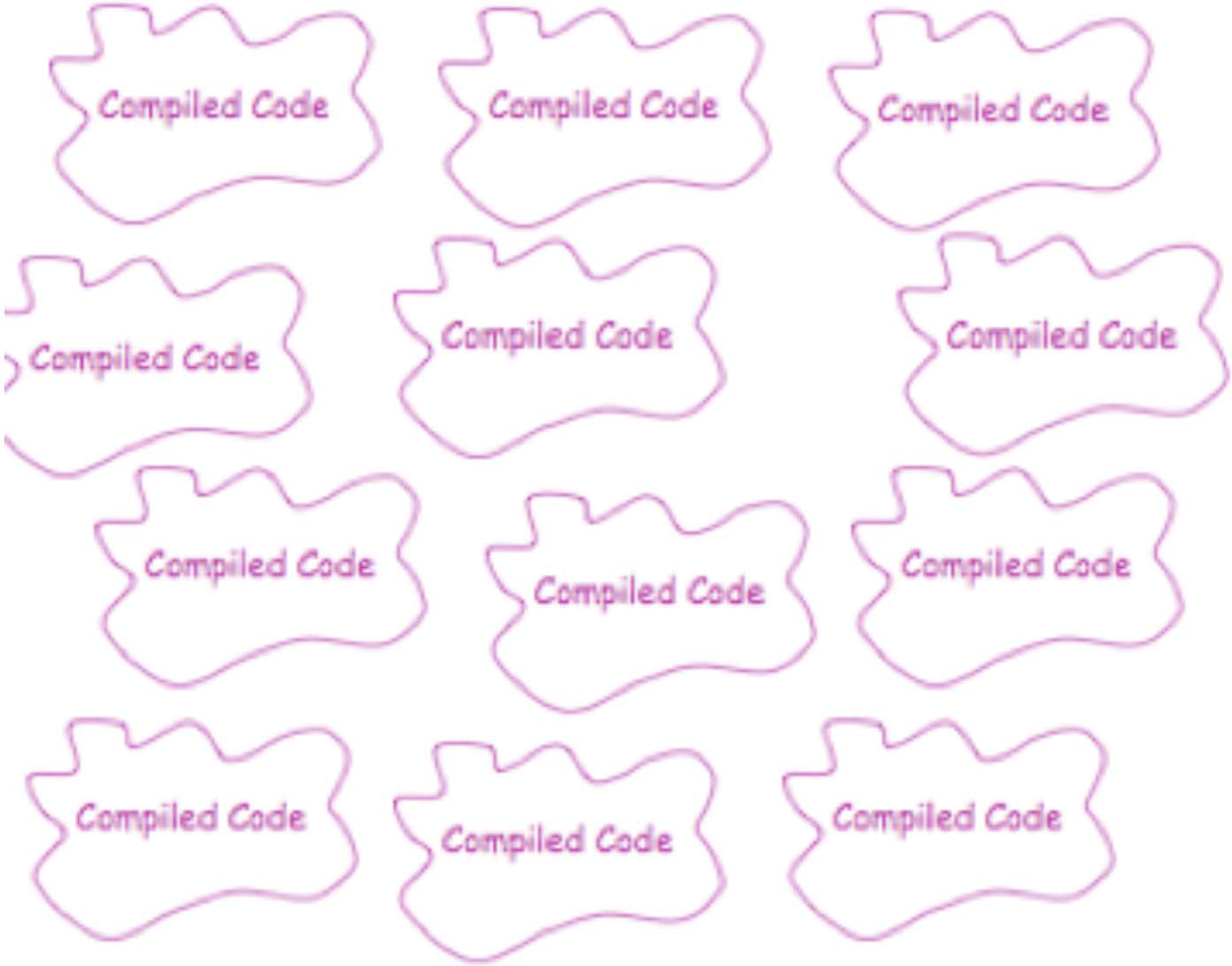
The Procedure & Its Three Abstractions



The compiler produces code for each procedure

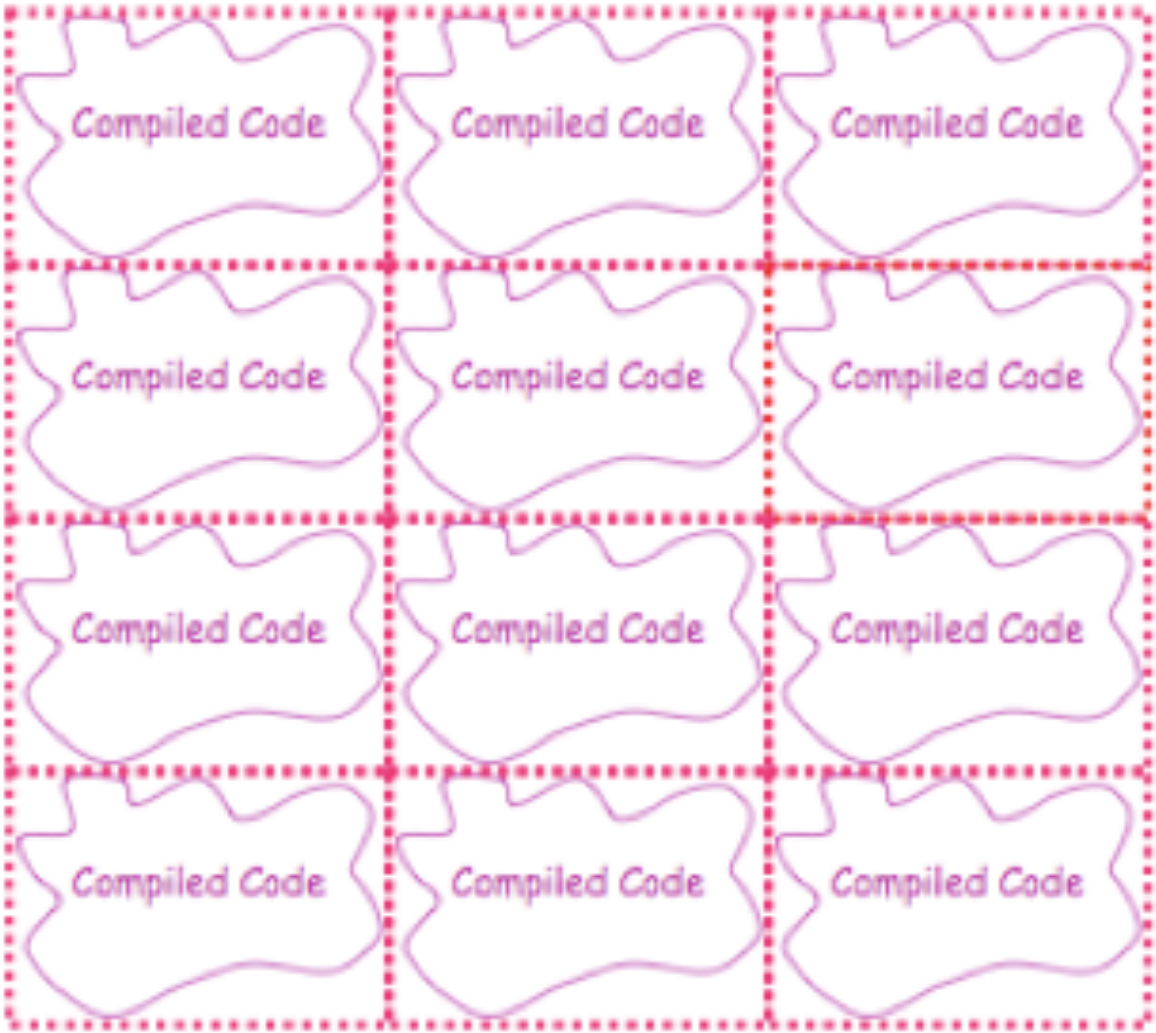
The individual code bodies must fit together to form a working program

The Procedure as a Name Space

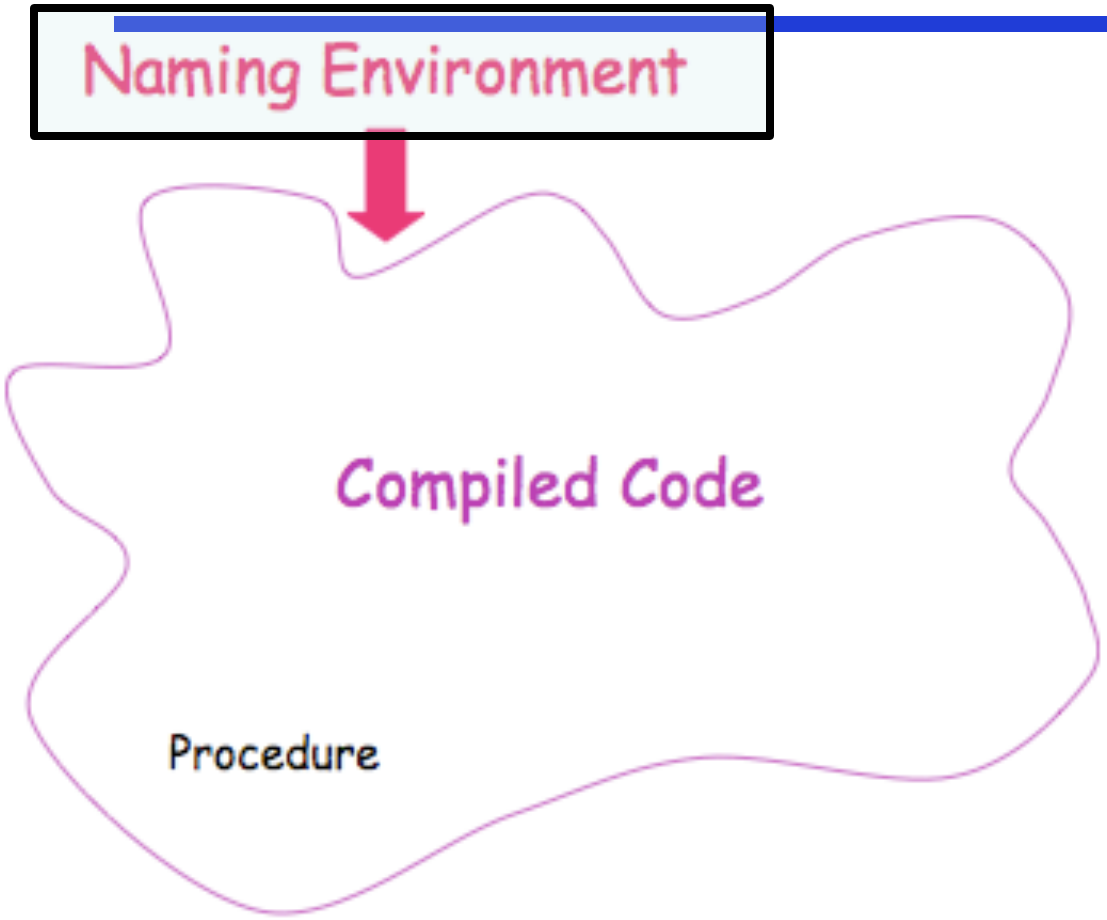


The Procedure as a Name Space

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



The Three Procedure Abstractions: Naming



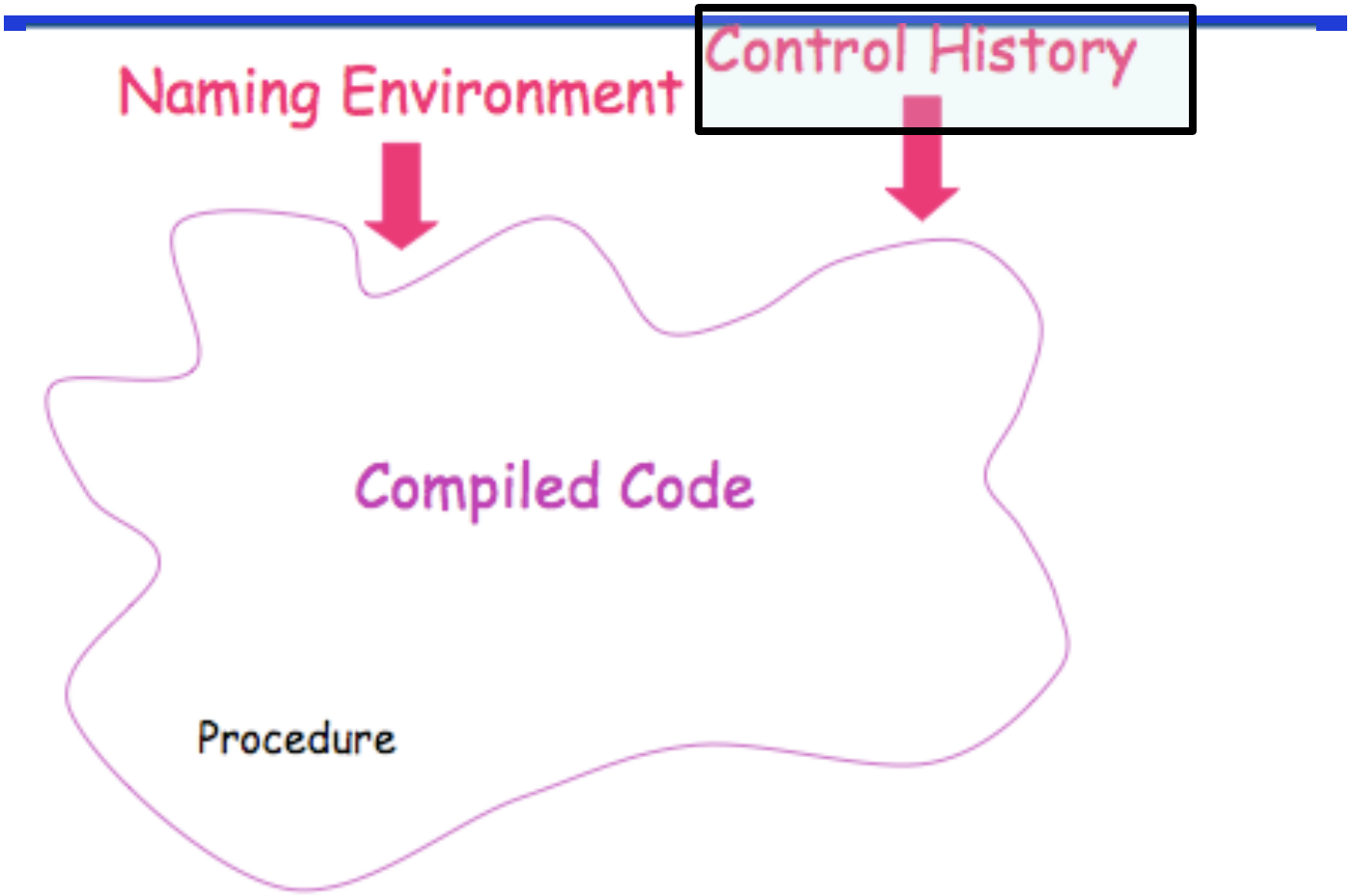
"Naming" includes the ability to find and access the object in memory

Each procedure inherits a set of names

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

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

The Three Procedure Abstractions: Control

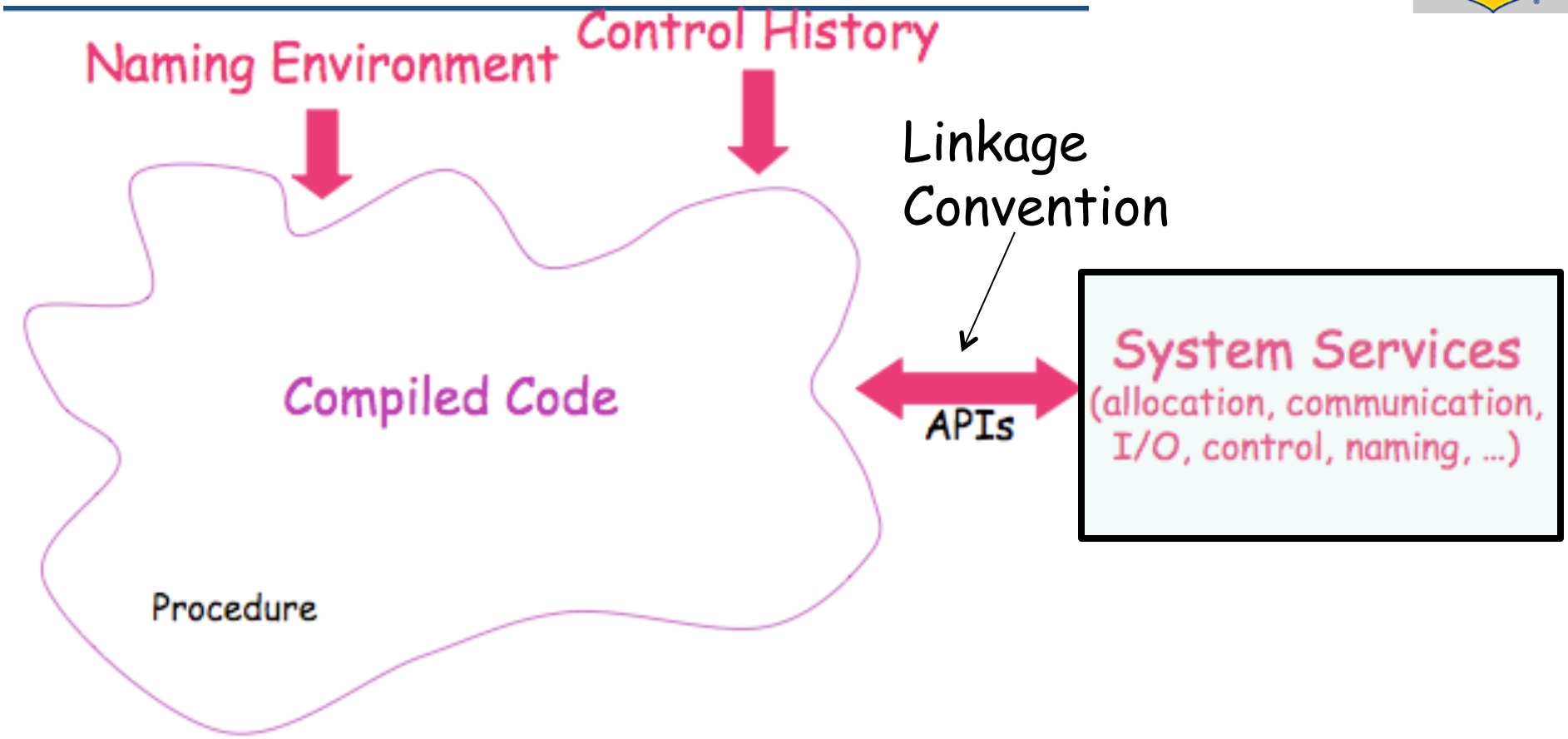


Each procedure inherits a control history

⇒ Chain of calls that led to its invocation

⇒ Mechanism to return control to caller

The Three Procedure Abstractions: External



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 Three Procedure Abstractions Summary

- Clean Name Space
  - → Clean slate for writing locally visible names
  - → Local names may obscure identical, non-local names
  - → Local names cannot be seen outside
- Control Abstraction
  - → Well defined entries & exits
  - → Mechanism to return control to caller
- External Interface
  - → Access is by procedure name & parameters
  - → Clear protection for both caller & callee
  - → Invoked procedure can ignore calling context

The Procedure (Realist's View)

Procedures are the key to building large systems

- Requires system-wide contract
  - → Conventions on memory layout, protection, resource allocation, calling sequences, & error handling
  - → Must involve contract between
    - ♦ architecture (**ISA**), **OS**, & compiler

The Procedure (Realist's View)

Procedures allow us to use separate compilation

- Separate compilation allows us to build non-trivial programs
- Keeps compile times reasonable
- Lets multiple programmers collaborate
- Requires independent procedures

Without separate compilation, we *would not* build large systems

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, and addresses, but not:

- Entries and exits
- Interfaces
- Call and return mechanisms
  - → may be a special instruction to save context at point of call
- Name space
- Nested scopes

Run Time versus Compile Time

These concepts are often confusing to the newcomer

- Linkages execute at run time
- Code for the linkage is emitted at compile time
- The linkage is designed long before either of these

Compile time vs run time can be confusing to compiler students.

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The procedure call

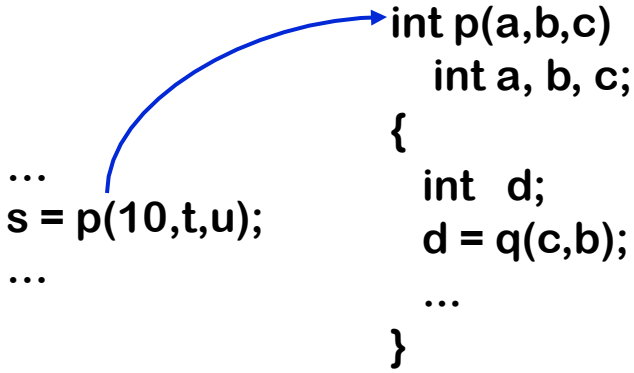
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The procedure call

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

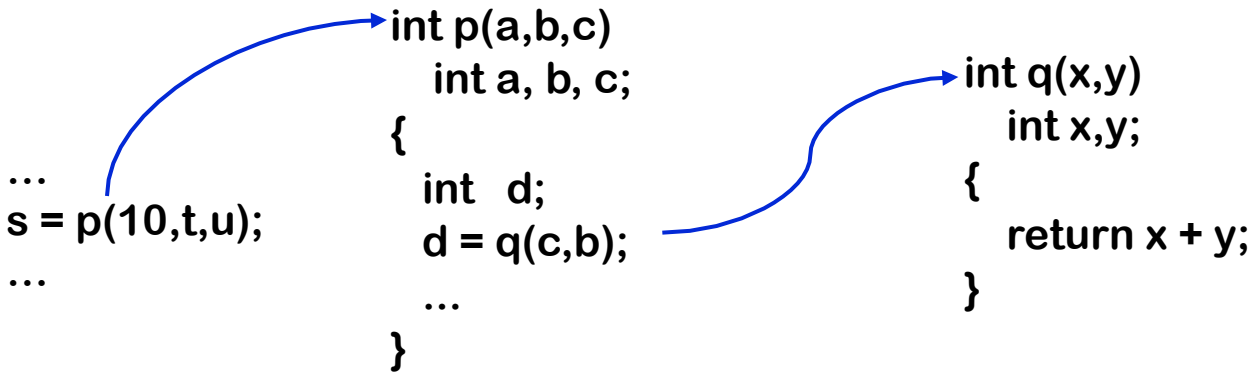


The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The procedure call

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

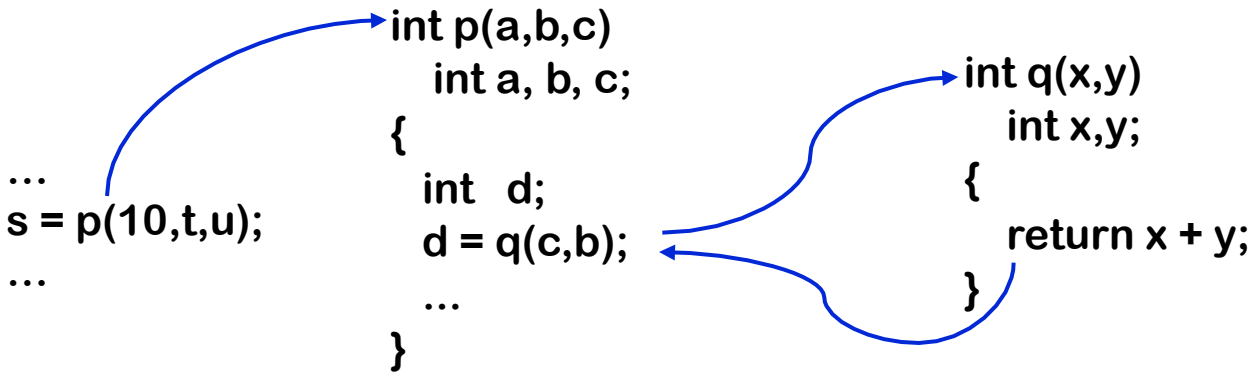


The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The procedure call

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

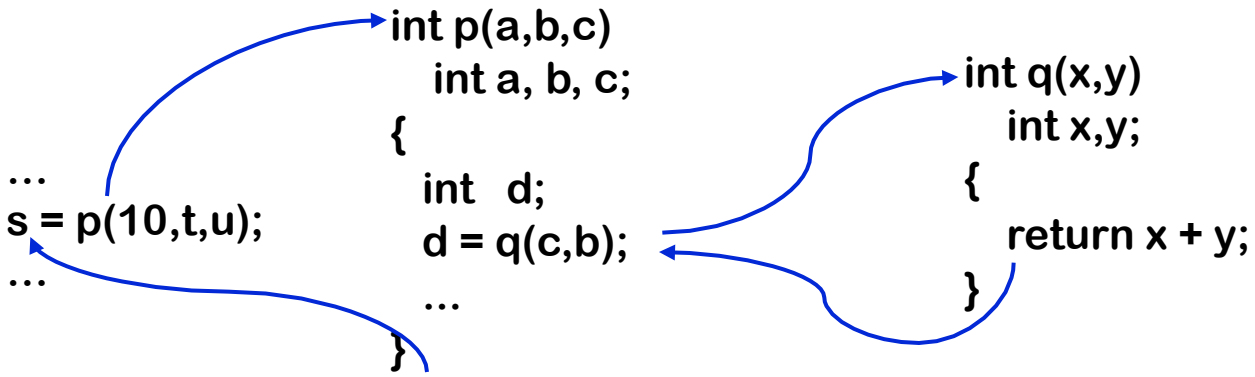


The Procedure as a Control Abstraction

Procedures have well-defined control-flow

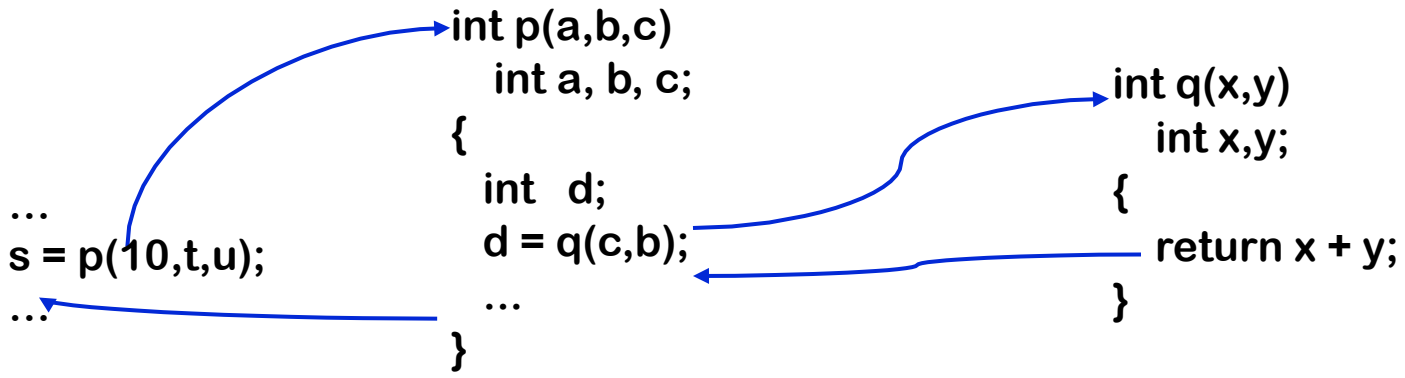
The procedure call

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



Implementing procedures with this behavior

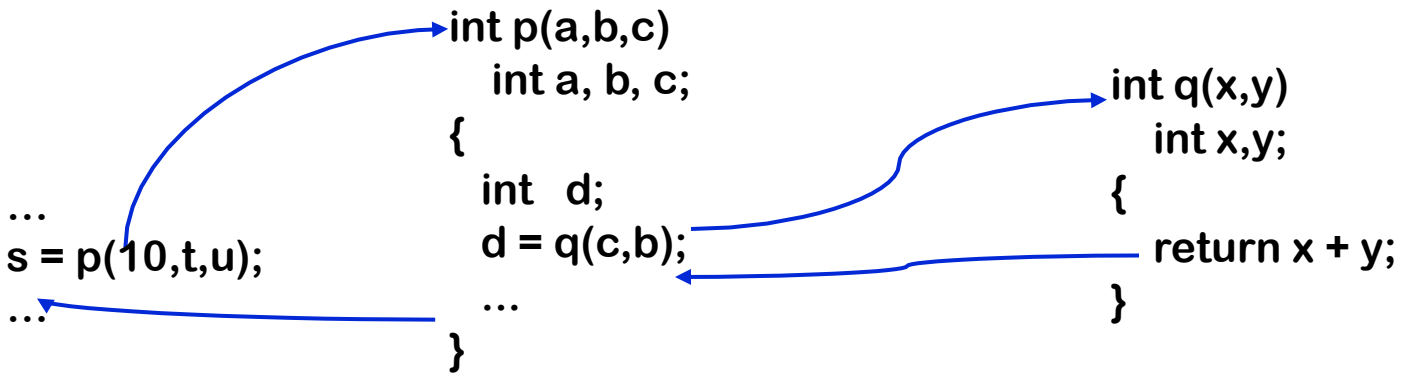
- Requires code to save and restore a "return address"



Compiler emits code to save and restore address.

Implementing procedures with this behavior

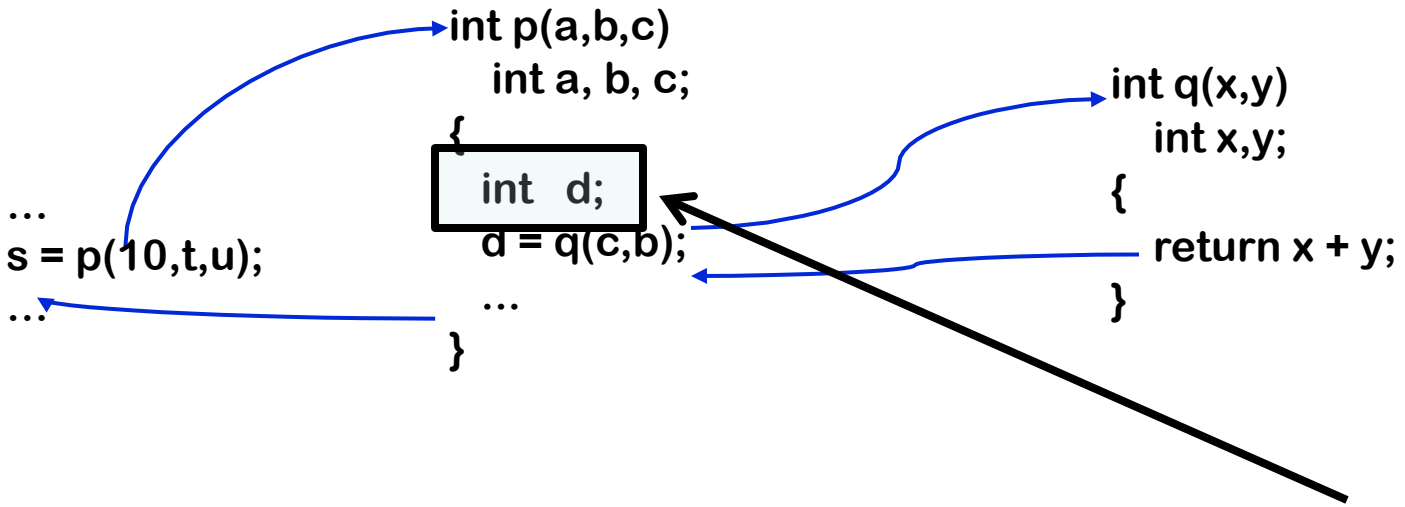
- Must map actual parameters to formal parameters
  (c→x, b→y)



Compiler emits code to copy parameters from caller to callee. Usually passed through call stack.

Implementing procedures with this behavior

- 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?

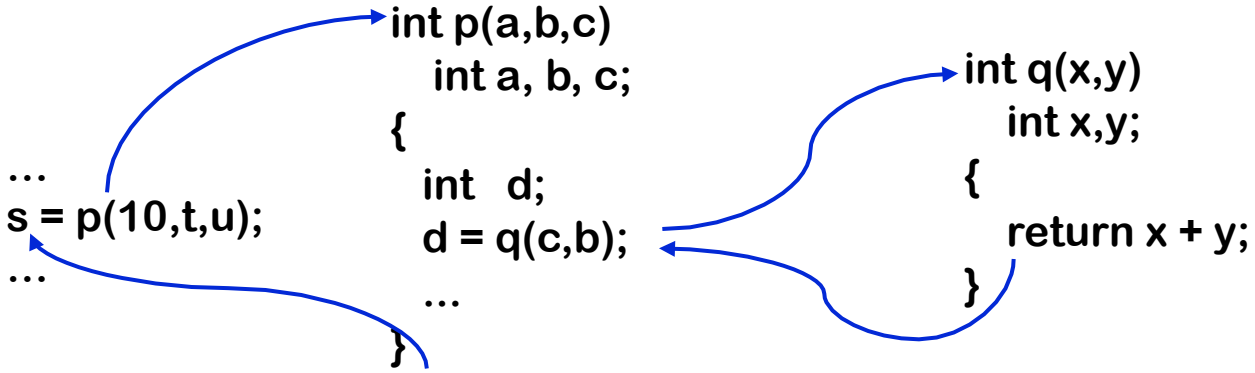


Compiler emits code to construct local area for callee procedure.

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
  - → Can use a "stack" of memory blocks to hold local storage and return addresses



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