The Procedure Abstraction
Part I Basics

Procedure Abstraction

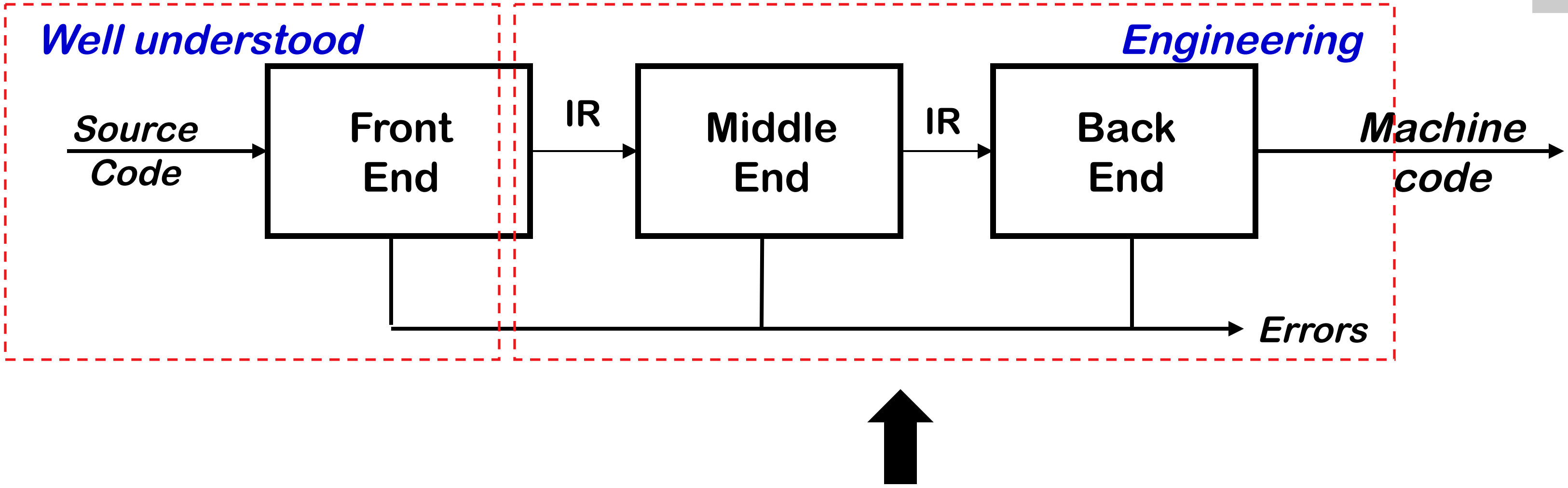
- The compiler must deal with interface between compile time and run time
  - →Most of the tricky issues arise in implementing “procedures”

Procedures are the key to building large systems

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
- Efficiency of implementation

Where are we?

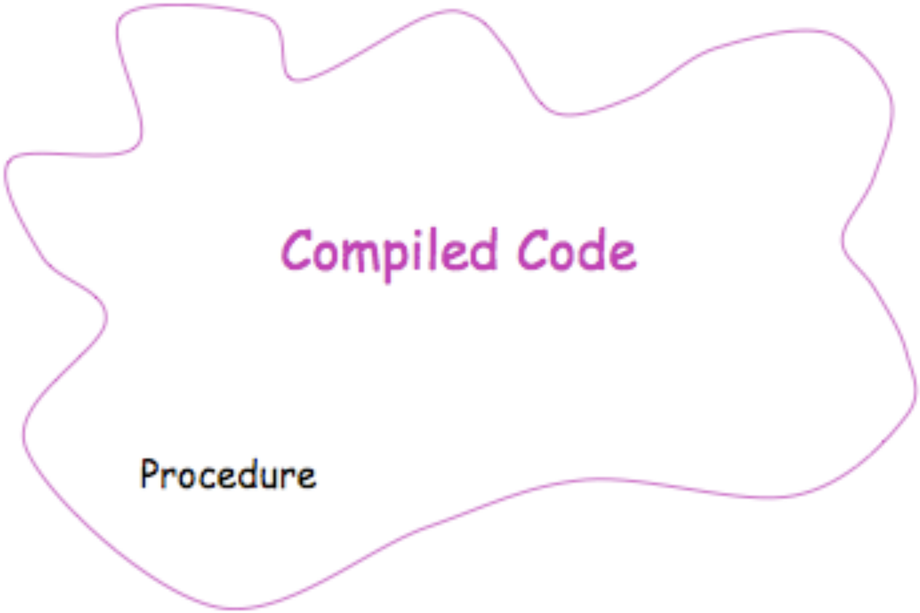


Contains more open problems and more challenges

- 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

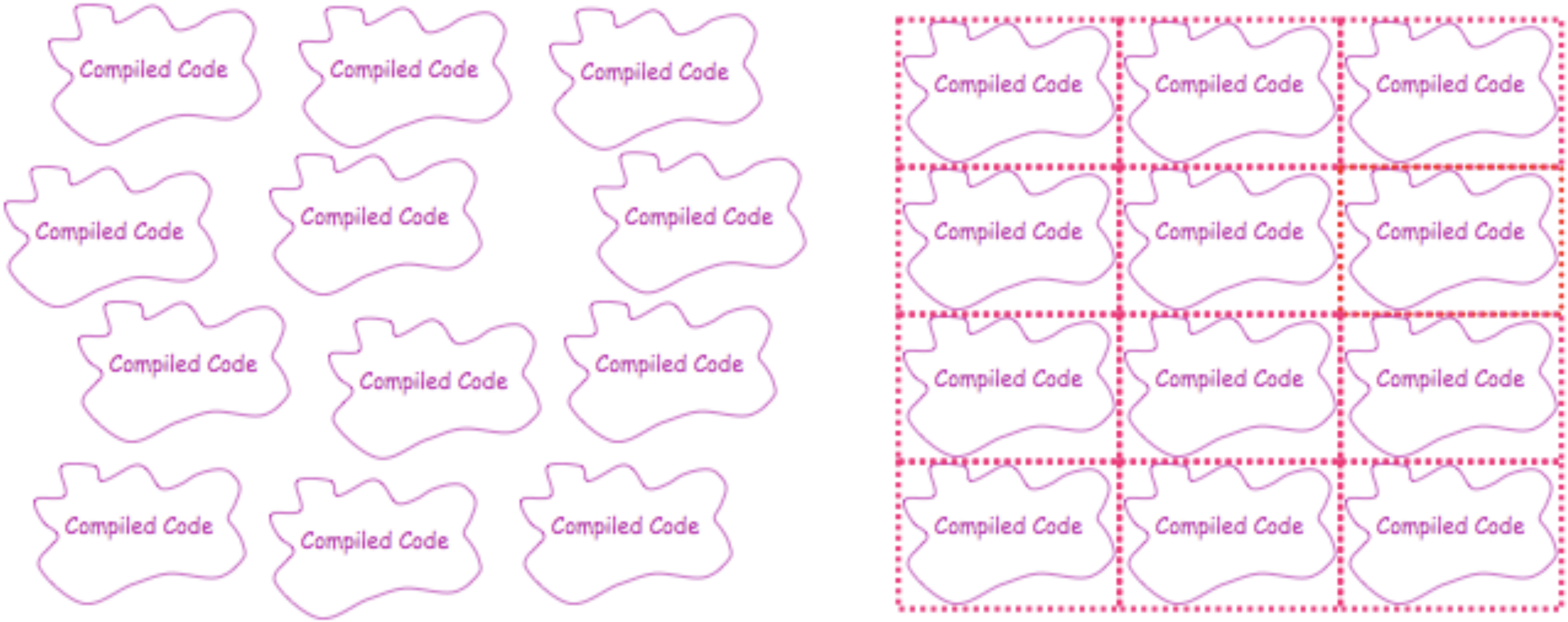
The compiler produces code for each procedure



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

The Procedure as a Name Space

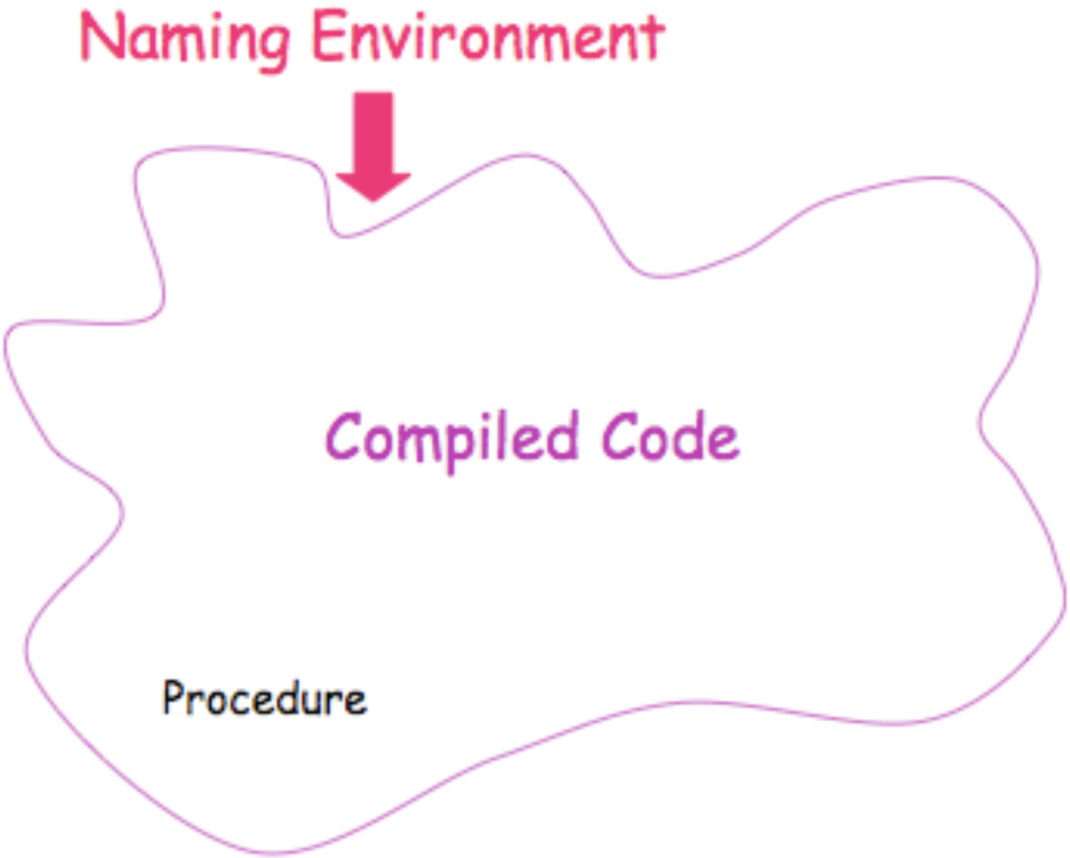
In essence, the procedure linkage wraps around the unique code of each procedure to give it a uniform interface



Similar to building a brick wall rather than a rock wall

There is a strict constraints that each procedure must adhere to!

The Procedure: Three Abstractions



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

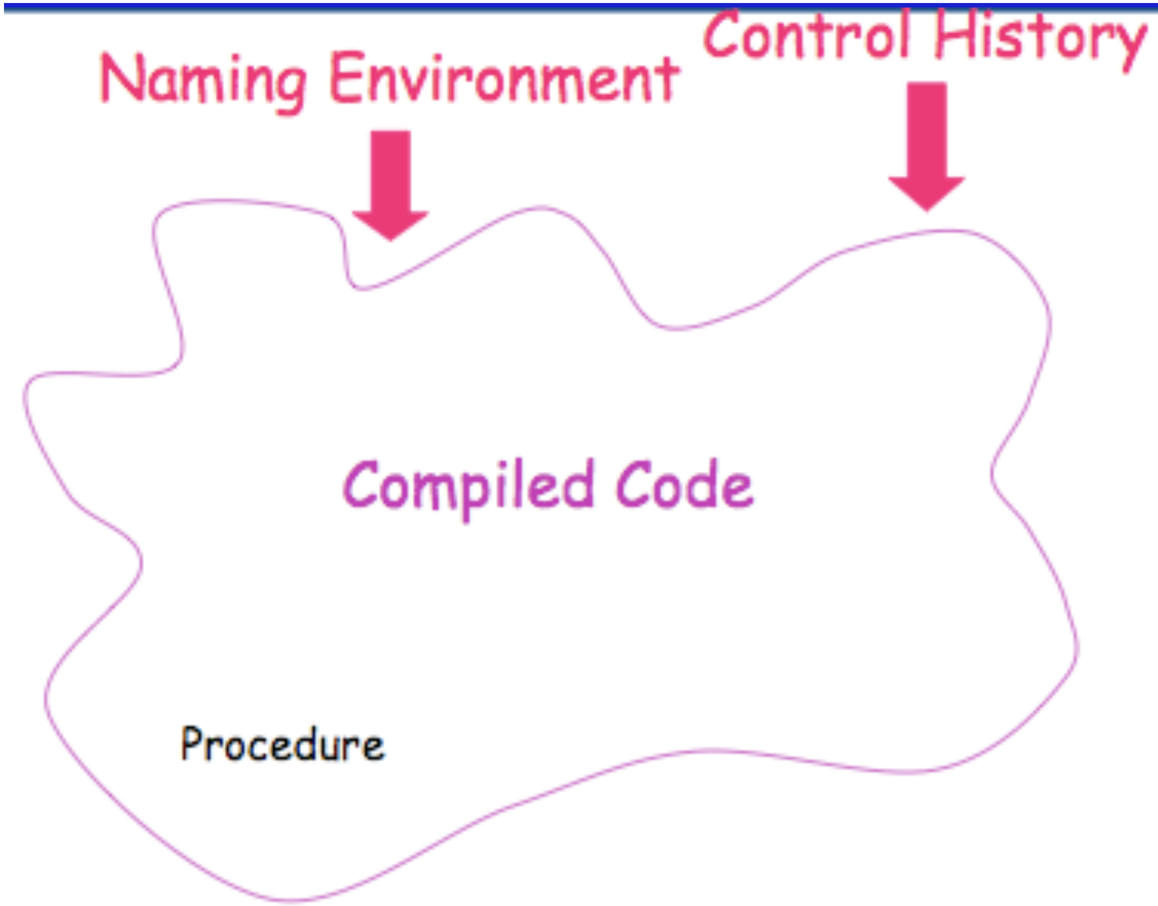
Each procedure inherits a set of names

- ⇒ Variables, values, procedures, objects, locations, …
- ⇒ Clean slate for new names, "scoping" can hide other names

The Procedure: Three Abstractions

1. Name Environment
   - →Clean slate for writing locally visible names
   - →Local names may obscure identical, non-local names
   - →Local names cannot be seen outside

The Procedure: Three Abstractions



Each procedure inherits a control history

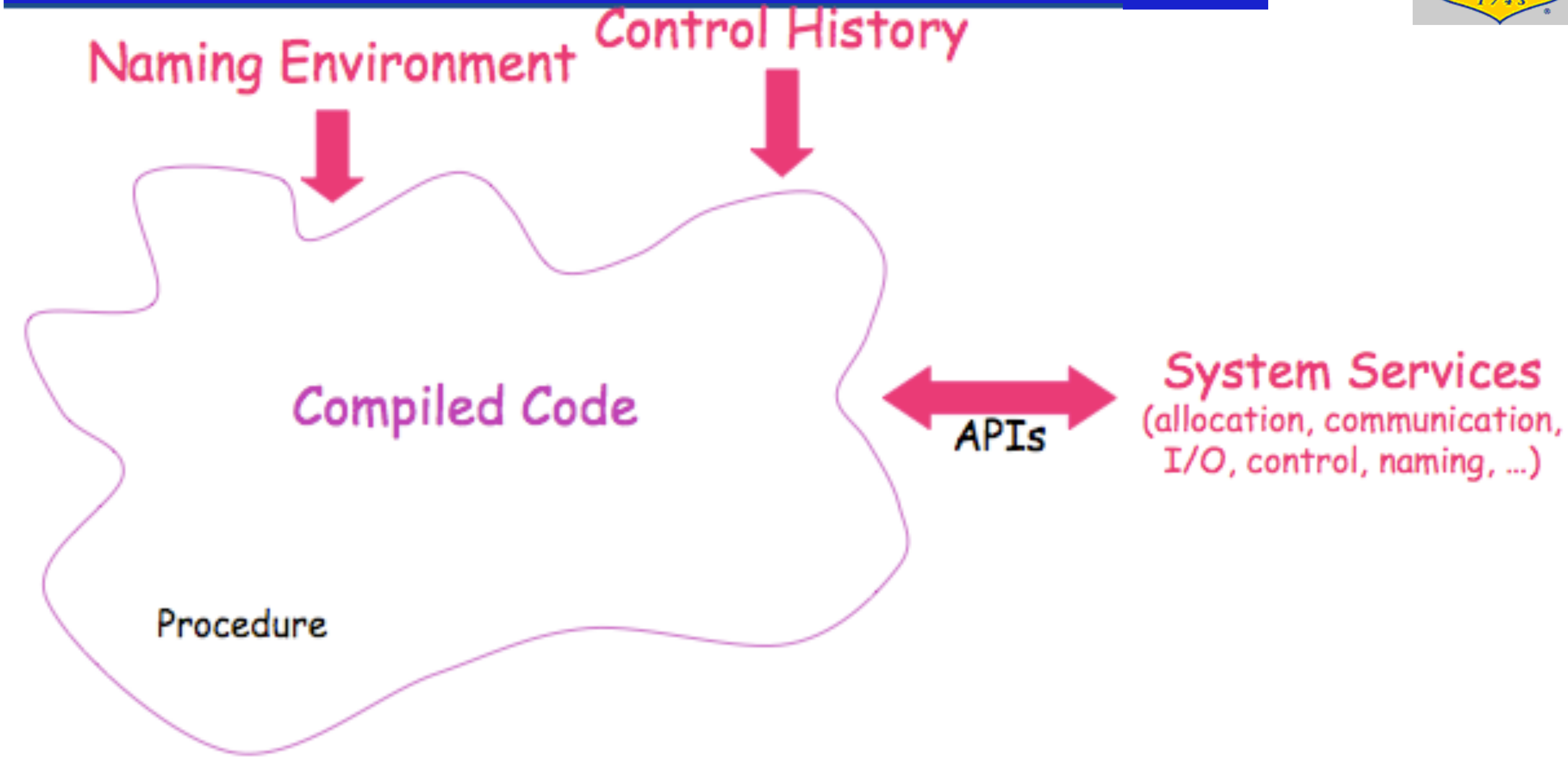
⇒ Chain of calls that led to its invocation

⇒ Mechanism to return control to caller

The Procedure: Three Abstractions

2. Control History
   - → Well defined entries & exits
   - → Mechanism to return control to caller

The Procedure: 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: Three Abstractions

3. System Services
   - → Access is by procedure name & parameters
   - → Clear protection for both caller & callee
   - → Invoked procedure can ignore calling context

Procedures permit a critical separation of concerns

The Procedure (Realist's View)

- Establishes a private context
  - → Create private storage for each procedure invocation
  - → Encapsulate information about control flow & data abstractions

The Procedure (Realist's View)

- Provides shared access to system-wide facilities
  - → Storage management, flow of control, interrupts
  - → Interface to input/output devices, protection facilities, timers, synchronization flags, counters, …

The Procedure (Realist's View)

- Requires system-wide contract
  - →Conventions on memory layout, protection, resource allocation calling sequences, & error handling
  - →Must involve 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 (Realist's View)

The procedure linkage convention

- Agreement between compiler and OS on actions taken when a procedure/function is called.
- Ensures each procedure inherits valid run-time environment and that the caller's environment is restored on return
  - →Compiler generates code to ensure this happens according to agreement established by the system

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
- Name space
- Nested scopes

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

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 versus run time can be confusing to students. We will emphasize the distinction between them.

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 (Algol-Like Languages = ALLs) procedure call

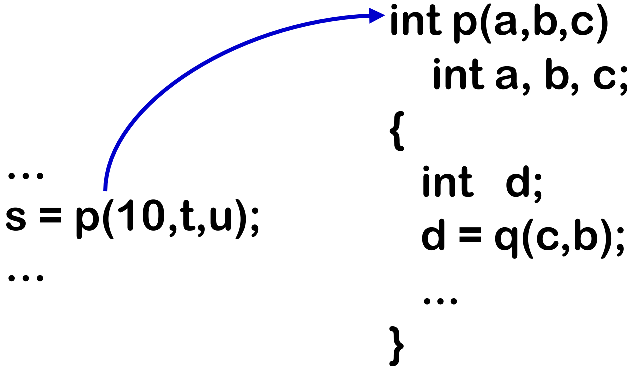
- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The 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);
...
```

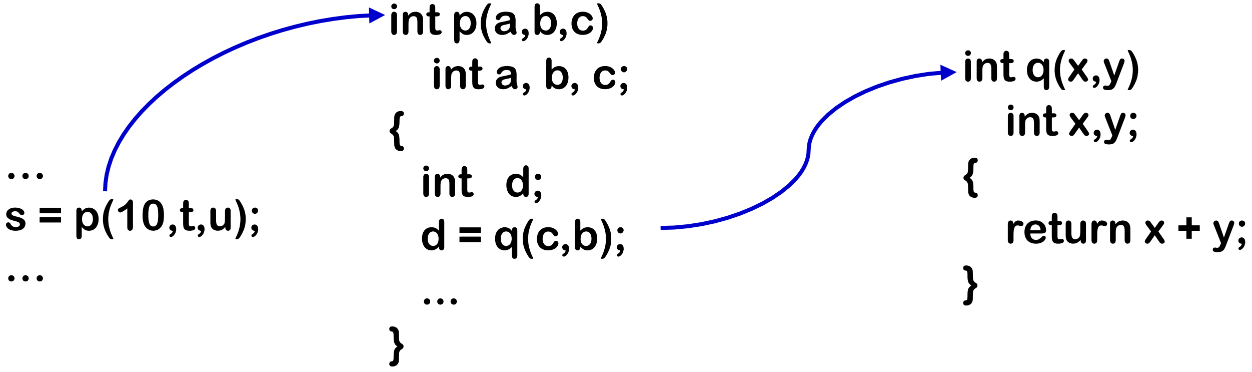
```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  ...
}
```

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);
  ...
}
```

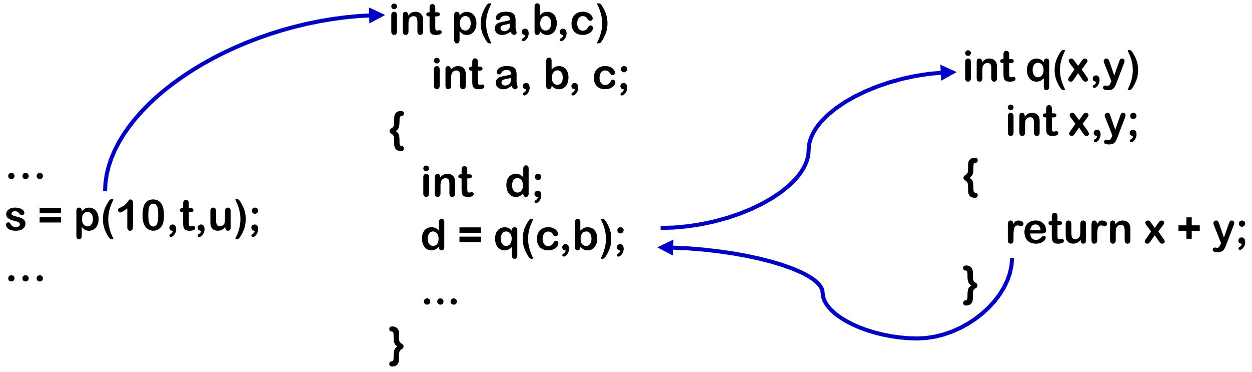
```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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);
…
```

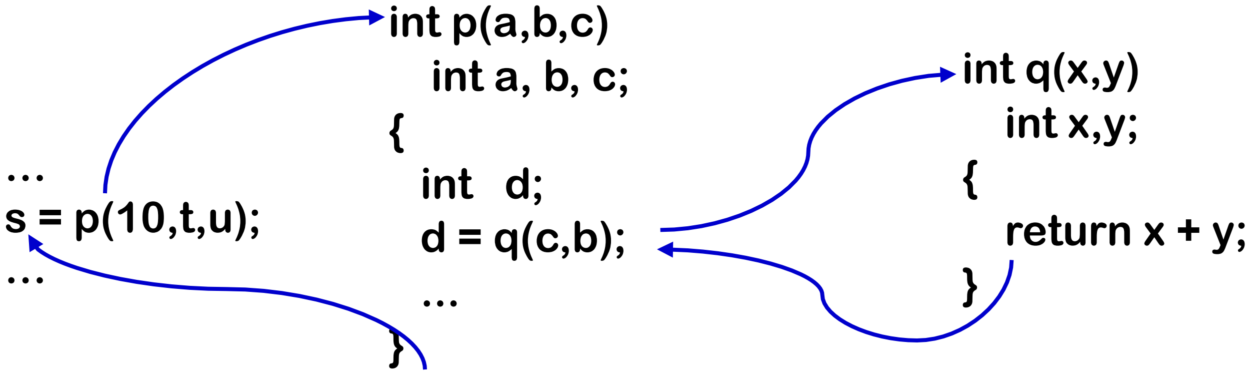
```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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);
...
```

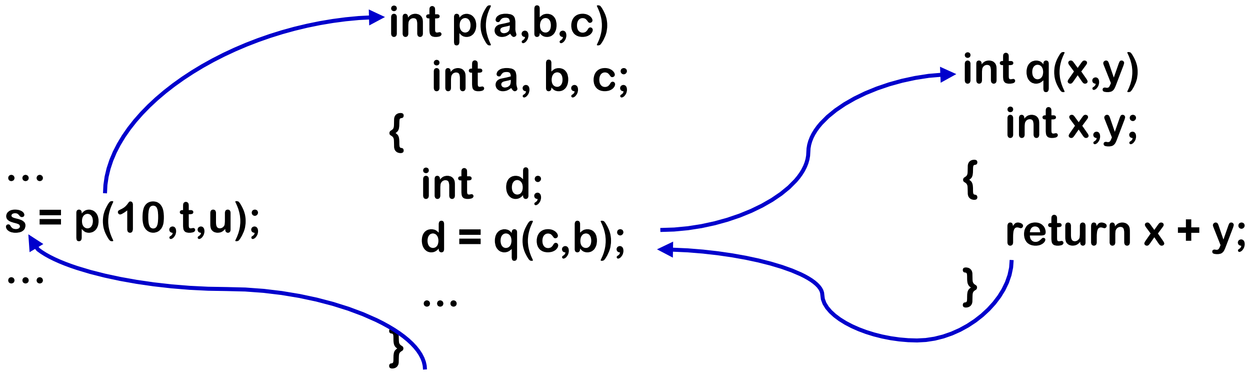
```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  ...
}
```

```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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);
  ...
}
```

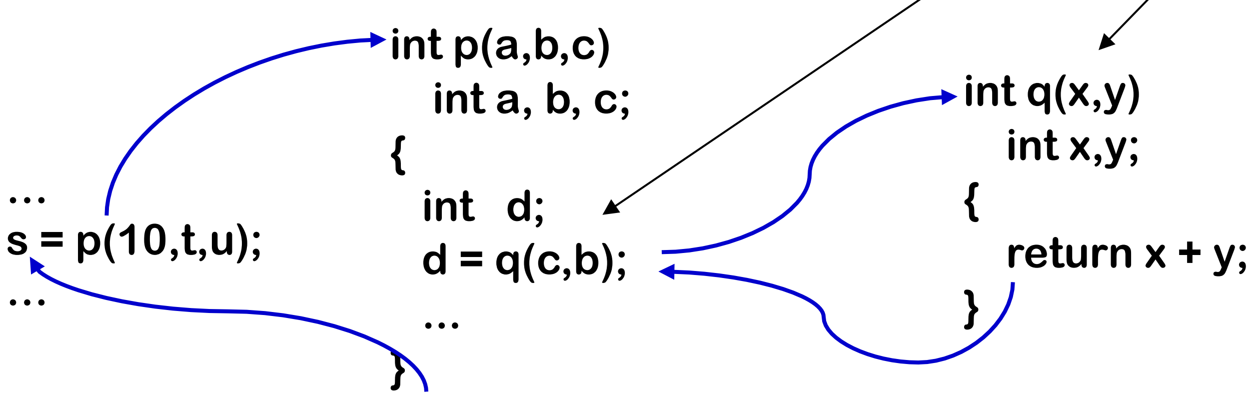
```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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*→*x*, *b*→*y*)
- Must create storage for local variables (&, maybe, parameters)
  - → *p* needs space for *d*, *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);
  ...
}
```

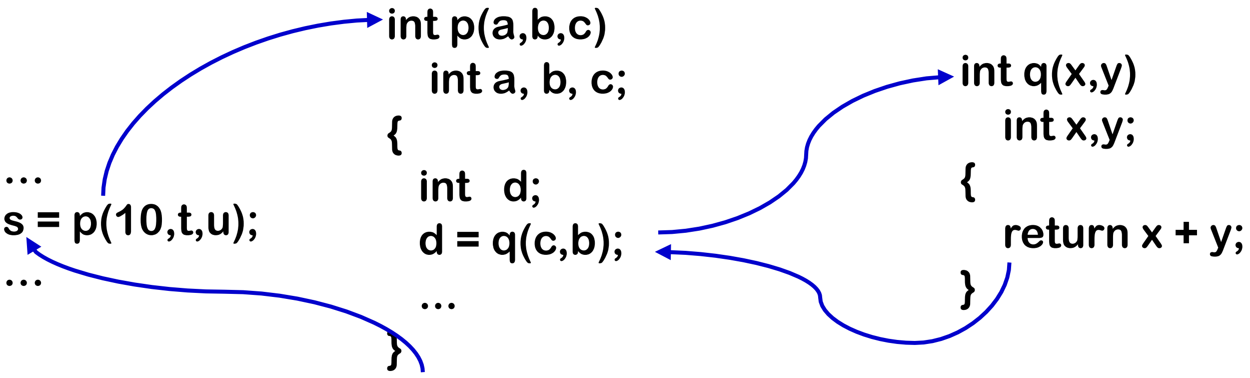
```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

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
  - → 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;
{
   return x + y;
}
```

Compiler <u>emits</u> code that causes all this to happen at run time