



COMP 181

Lecture 11

The Procedure Abstraction

October 17, 2005

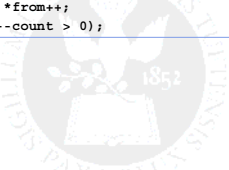




Prelude

- Copying an array

```
do {
    *to++ = *from++;
} while (--count > 0);
```

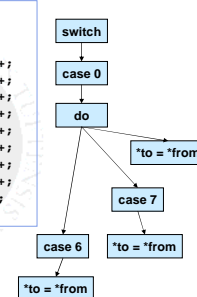
/* count > 0 assumed */


Tufts University Computer Science

Duff's device

```
int n = (count + 7) / 8;
switch (count % 8)
{
  case 0: do { *to++ = *from++;
  case 7: *to++ = *from++;
  case 6: *to++ = *from++;
  case 5: *to++ = *from++;
  case 4: *to++ = *from++;
  case 3: *to++ = *from++;
  case 2: *to++ = *from++;
  case 1: *to++ = *from++;
} while (--n > 0);
```




- Why is this faster?



Tufts University Computer Science

Schedule

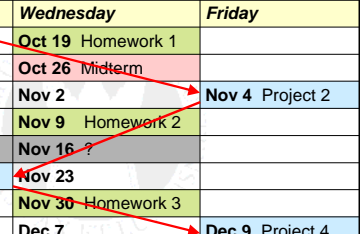

Monday	Wednesday	Friday
Oct 17 -- Today --	Oct 19 Homework 1	
Oct 24	Oct 26 Midterm	
Oct 31	Nov 2	Nov 4 Project 2
Nov 7	Nov 9 Homework 2	
Nov 14 ?	Nov 16 ?	
Nov 21 Project 3	Nov 23	
Nov 28	Nov 30 Homework 3	
Dec 5	Dec 7	Dec 9 Project 4



Tufts University Computer Science

Schedule

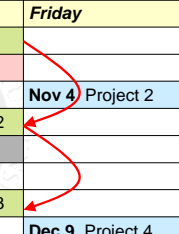

Monday	Wednesday	Friday
Oct 17 -- Today --	Oct 19 Homework 1	
Oct 24	Oct 26 Midterm	
Oct 31	Nov 2	Nov 4 Project 2
Nov 7	Nov 9 Homework 2	
Nov 14 ?	Nov 16 ?	
Nov 21 Project 3	Nov 23	
Nov 28	Nov 30 Homework 3	
Dec 5	Dec 7	Dec 9 Project 4

Tufts University Computer Science

Schedule

Monday	Wednesday	Friday
Oct 17 -- Today --	Oct 19 Homework 1	
Oct 24	Oct 26 Midterm	
Oct 31	Nov 2	Nov 4 Project 2
Nov 7	Nov 9 Homework 2	
Nov 14 ?	Nov 16 ?	
Nov 21 Project 3	Nov 23	
Nov 28	Nov 30 Homework 3	
Dec 5	Dec 7	Dec 9 Project 4

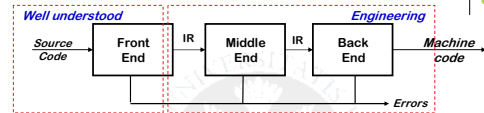
Tufts University Computer Science

Procedure Abstraction

- The compiler must deal with interface between **compile time** and **run time** (static versus dynamic)
 - Most of the tricky issues arise in implementing "procedures"
- Issues
 - Finding storage, and mapping names to addresses
 - Generating code to compute addresses that the compiler cannot know !
 - Interfaces with other programs, other languages, and the OS
 - Efficiency of implementation



Where are we?



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

- 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: Three Abstractions

- Control** Abstraction
 - Well defined entries & exits
 - Mechanism to return control to caller
 - Some notion of parameterization (usually)
- Clean **Name Space**
 - Clean slate for writing locally visible names
 - Local names may obscure identical, non-local names
 - Local names cannot be seen outside
- External **Interface**
 - 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)

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 architecture (ISA), OS, & compiler
- Provides shared **access to system-wide facilities**
 - Storage management, flow of control, interrupts
 - Interface to input/output devices, protection facilities, timers, synchronization flags, counters, ...
- Establishes a **private context**
 - Create private storage for each procedure invocation
 - Encapsulate information about control flow & data abstractions



The Procedure (Realist's View)

Procedures allow us to use **separate compilation**

- Keeps compile times reasonable
- Lets multiple programmers collaborate
- Requires independent procedures
- Without it, we would not build large systems

The procedure **linkage convention**

- Ensures that each procedure:
 - Inherits a valid run-time environment
 - Callers environment is restored on return
 - The compiler generates code to ensure this happens according to conventions



The Procedure (More Abstract View)

- A procedure is an **abstract structure** constructed via software
- Underlying hardware does explicitly not support:
 - Entries and exits
 - Interfaces and parameter passing
 - Call and return mechanism (may be special instructions)
 - Name spaces and nested scopes
- Abstraction created by cooperation between:
 - Compiler
 - Run-time system
 - Linkage editor and loader
 - Operating system

ABI
Application Binary Interface



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
- More confusion:
 - Dynamic linking and shared libraries
 - Just-in-time compilers



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



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;
}
    
```

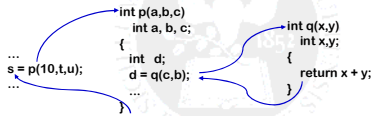


The Procedure as a Control Abstraction

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



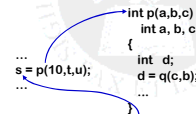
- Most languages allow recursion



The Procedure as a Control Abstraction

Implementing procedures with this behavior

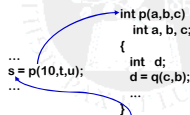
- Requires code to *save* and *restore* a "return address"
- Must map *actual parameters* to *formal parameters* ($c \rightarrow x, b \rightarrow y$)
- Must create storage for *local variables* (&, maybe, parameters)
 - p needs space for d (and, maybe, $a, b,$ & c)
 - Where does this space go in recursive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



The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Must preserve p 's *state* while q executes
 - *Recursion causes the real problem here*
- **Strategy:** Create unique location for each procedure call
 - Each block holds local storage, return addresses
 - Organize these blocks into a stack



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



The Procedure as a Name Space

Each procedure creates its own name space

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

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
 - *Procedure scope (typically) contains formal parameters*



The Procedure as a Name Space

- Why introduce lexical scoping?
 - Flexibility for programmer
 - Simplifies rules for naming & resolves conflicts
 - 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 in memory?
- The Answer
 - *Lexically scoped symbol tables*



Examples

- In C++ and Java

```

{
  for (int i=0; i < 100; i++) {
    ...
  }

  for (Iterator i=list.iterator(); i.hasNext(); ) {
    ...
  }
}
    
```

- This is actually useful!



Dynamic vs static

- Static scoping
 - Most compiled languages – C, C++, Java, Fortran
 - Scopes only exist at compile-time
 - We'll see the corresponding run-time structures that are used to establish addressability later.
- Dynamic scoping
 - Interpreted languages – Perl, Common Lisp

```
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```



Lexically-scoped Symbol Tables

- The problem
 - Compiler needs a distinct record for each declaration
 - Nested lexical scopes admit duplicate declarations
- The interface
 - `enter()` – enter a new scope level
 - `insert(name)` – creates entry for `name` in current scope
 - `lookup(name)` – lookup a name, return an entry
 - `exit()` – leave scope, remove all names declared there



Example

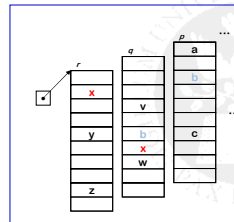
```
procedure p {
  int a, b, c
  procedure q {
    int v, b, x, w
    procedure r {
      int x, y, z
      ...
    }
    procedure s {
      int x, a, v
      ...
    }
    ... r ... s
  }
  ... q ...
}
```

```
L0: { int a, b, c
L1: { int v, b, x, w
L2a: { int x, y, z
      ...
L2b: { int x, a, v
      ...
}
```



Chained implementation

- Create a new table for each scope, chain them together for lookup



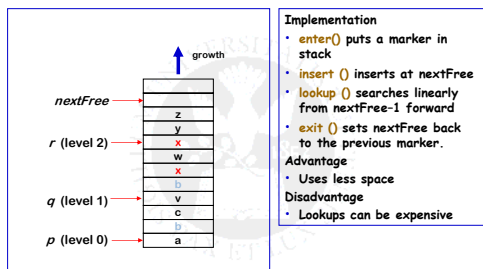
"Sheaf of tables" implementation

- `enter()` creates a new table
- `insert()` adds at current level
- `lookup()` walks chain of tables & returns first occurrence of name
- `exit()` throws away table for level `p`, if it is top table in the chain

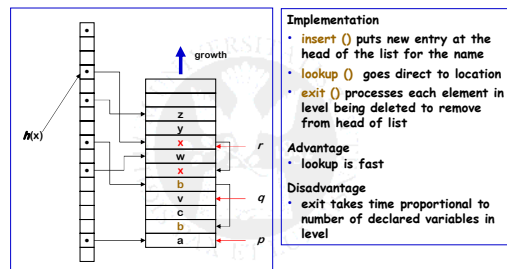
If the compiler must preserve the table (for, say, the debugger), this idea is actually practical. Individual tables can be hash tables.



Stack implementation



Threaded stack implementation



Symbol tables in C

- Identifiers
 - Mapping from names to declarations
 - Fully nested – each '{' opens new scope
- Labels
 - Mapping from names to labels (for goto)
 - Flat table – one set of labels for each procedure
- Tags
 - Mapping from names to struct definitions
 - Fully nested
- Externals
 - Record of extern declarations
 - Flat table – redundant extern declarations must be identical

In general, rules can be very subtle



Examples

- Example of typedef use:

```
typedef int T;
struct S { T T; }; /* redefinition of T as member name */
```

- Example of proper declaration binding:

```
int; /* syntax error: vacuous declaration */
struct S; /* no error: tag is defined or elaborated */
```

- Example of declaration name spaces

- Declare "a" in the name space before parsing initializer

```
int a = sizeof(a);
```

- Declare "b" with a type before parsing "c"

```
int b, c[sizeof(b)];
```



The Procedure as an External Interface

OS needs a way to start the program's execution

- Programmer needs a way to indicate where it begins
The "main" procedure in most languages

UNIX/Linux specific discussion

- When user invokes "grep" at a command line
 - OS finds the executable
 - OS creates a process and arranges for it to run "grep"
 - "grep" is code from compiler, linked with run-time system
 - Starts the run-time environment & calls "main"
 - After main, it shuts down run-time environment & returns
- When "grep" needs system services
 - It makes a system call, such as fopen()



Where Do All These Variables Go?

- Automatic & Local

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

- Static

- Procedure scope ⇒ storage area tied to procedure name
- File scope ⇒ storage area tied to file name
- Lifetime is entire execution

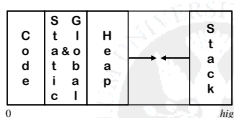
- Global

- 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

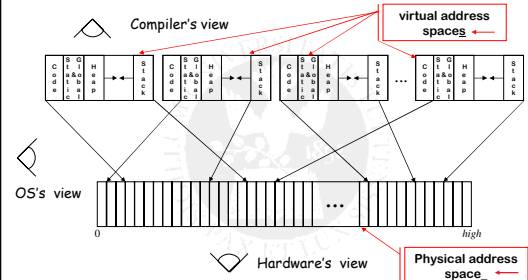


Single Logical Address Space

- Better utilization if stack & heap grow toward each other
- Code, static, & global data have known size
- Very old result (Knuth)
- Code & data separate or interleaved
- Heap & stack both grow & shrink over time
- Use symbolic labels in the code
- Uses address space, not allocated memory
- This is a virtual address space



How Does This Really Work?



Where Do Local Variables Live?

- A Simplistic model
 - Allocate a data area for each distinct scope
 - One data area per "sheaf" 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 proc's scopes share a single AR (*may share space*)
 - Static relationship between scopes in single procedure



Translating Local Names

How does the compiler represent a specific instance of x ?

- Name is translated into a **static coordinate**
 - $\langle \text{level}, \text{offset} \rangle$ 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
 - "level" is a function of the table in which x is found
 - Stored in the entry for each x
- "offset" must be assigned and stored in the symbol table
 - Known at compile time
 - Used to generate code that executes at run-time



Storage for Blocks within a Single Procedure

```

L0:
{
  int a, b, c
L1:
{
  int v, b, x, w
L2a:
{
  int x, y, z
L2b:
{
  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 to a
B1:
{
  int v(a), b, x
B2:
{
  int x, y(b)
  ...
}
}
    
```

- 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 block in which it is allocated



Includes variable length data for all blocks in the procedure ...

Variable-length data

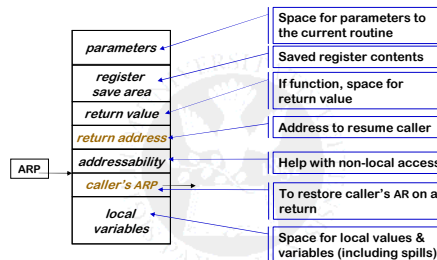


Variable stack data in C

- C does not support variable array declarations (like Fortran)
- Run-time call to allocate on stack
 - `void *alloca(size_t size);`
 - The `alloca` function allocates `size` bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.



Activation Record Basics



One AR for each invocation of a procedure



Activation Record Details

How does the compiler find the variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a "loadAI" operation
 - **Level** specifies an ARP, **offset** is the constant

Variable-length data

- If AR can be extended, put it below local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

- Must generate explicit code to store the values
- Among the procedure's first actions



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
 - If a procedure can outlive its caller, **OR**
 - If it can return an object that can reference its execution state
 - ARs **must** be kept in the heap
 - If a procedure makes no calls
 - AR can be allocated statically

Efficiency prefers static, stack, then heap



Communicating Between Procedures

Most languages provide a parameter passing mechanism
⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- **Call-by-reference** passes a pointer to actual parameter
 - Requires slot in the AR (for **address** of parameter)
 - Multiple names with the same address? `call fee(x,x,x):`
- **Call-by-value** passes a copy of its value at time of call
 - Requires slot in the AR
 - Each name gets a unique location *(may have same value)*
 - Arrays are mostly passed by reference, not value
- Can always use global variables ...



Next time...

- No class on Wednesday
 - I'm at OOPSLA 2005
 - Object-oriented Programming, Systems, Language and Applications
- Next Monday
 - Intermediate representations

