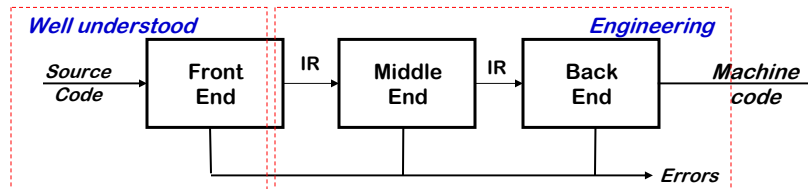


Procedure Abstraction

- The compiler must deal with interface between **compile time** and **run time** (static versus dynamic)
 - > Most of the tricky arise in implementing “procedures”
- Issues
 - > Compile-time versus run-time behavior
 - > Finding storage for EVERYTHING, 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

- 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: 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 compact**
 - > 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**

- 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 **linkage convention**

- Ensures that each procedure inherits a valid run-time environment and that the callers environment is restored on return
 - > The compiler must generate code to ensure this happens according to conventions established by the system

The Procedure

(Alternate 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**

All these are established by a carefully-crafted system of mechanisms provided by compiler, run-time system, linkage editor 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

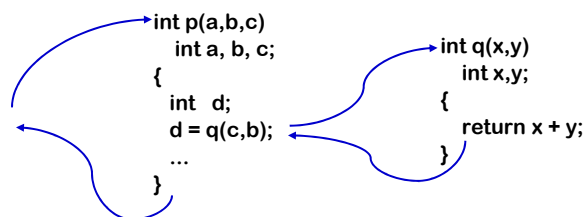
This issue (compile time versus run time) confuses students more than *any other* issue

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 (&, maybe, $a, b,$ & c)
 - > where does this space go in recursive invocations?
- Must preserve p 's **state** while q executes
 - > recursion causes the real problem here
- **Strategy.** Create unique location for each procedure **activation**
 - > Can use a “stack” of memory blocks to hold local storage and return addresses

Compiler must emit 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 obscure identical non-local names
- Local names cannot be seen outside the procedure
 - > Nested procedures are “inside” by definition
- 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 “locals”
 - > 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

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

- Lexically scoped symbol tables

Do People Use This Stuff ?

C macro from the MSCP compiler

```
#define fix_inequality(oper, new_opcode)      \  
if (value0 < value1)                        \  
{                                           \  
    Unsigned_Int temp = value0;             \  
    value0 = value1;                        \  
    value1 = temp;                          \  
    opcode_name = new_opcode;               \  
    temp = oper->arguments[0];              \  
    oper->arguments[0] = oper->arguments[1]; \  
    oper->arguments[1] = temp;              \  
    oper->opcode = new_opcode;               \  
}
```

Declares a new name

Do People Use This Stuff ?

C code from the MSCP implementation

```
static Void phi_node_printer(Block *block)
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
    {
        if (phi_node->old_name < register_count)
        {
            Unsigned_Int i;

            fprintf(stderr, "Phi node for r%d: [",
                phi_node->old_name);
            for (i = 0; i < block->pred_count; i++)
                fprintf(stderr, " r%d",
                    phi_node->parms[i]);
            fprintf(stderr, " ] => r%d\n",
                phi_node->new_name);
        }
        else
        {
            Unsigned_Int2 *arg_ptr;

            fprintf(stderr, "Phi node for %s: [",
                Expr_Get_String(Tag_Unmap(
                    phi_node->old_name)));
            Phi_Node_ForAllParms(arg_ptr, phi_node)
                fprintf(stderr, " %d", *arg_ptr);
            fprintf(stderr, " ] => %d\n",
                phi_node->new_name);
        }
    }
}
```

More local declarations!

CMSC430 Spring 2007

13

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, & fun

CMSC430 Spring 2007

14

Example

```

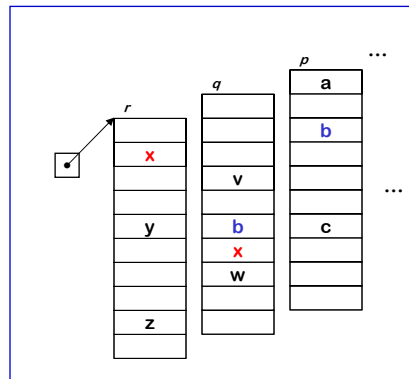
procedure p {
  int a, b, c
  procedure q {
    int v, b, x, w
    procedure r {
      int x, y, z
      ...
    }
    procedure s {
      int x, a, v
      ...
    }
    ... r ... s
  }
  ... q ...
}

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

Lexically-scoped Symbol Tables

High-level idea

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



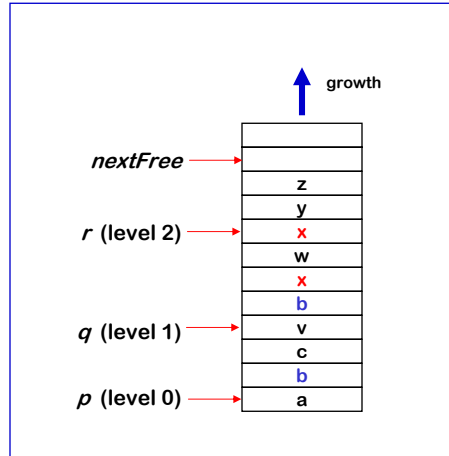
“Sheaf of tables” implementation

- *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 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.

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- **insert ()** creates new level pointer if needed and inserts at nextFree
- **lookup ()** searches linearly from nextFree-1 forward
- **delete ()** sets nextFree to the equal the start location of the level deleted.

Disadvantage

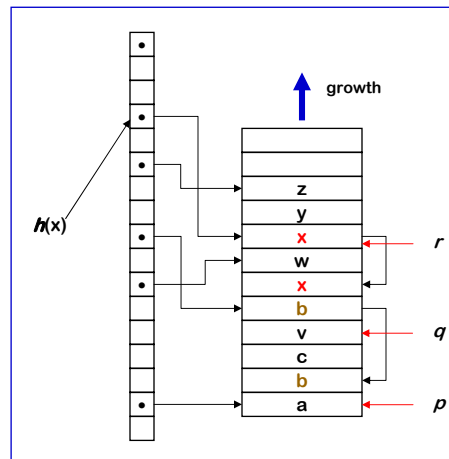
- lookups can be long

CMSC430 Spring 2007

17

Implementing Lexically Scoped Symbol Tables

Threaded stack organization



Implementation

- **insert ()** puts new entry at the head of the list for the name
- **lookup ()** goes direct to location
- **delete ()** processes each element in level being deleted to remove from head of list

Advantage

- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

CMSC430 Spring 2007

18

Where Do All These Variables Go?

Automatic & Local

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

Static

- Procedure scope \Rightarrow storage area affixed with procedure name
 - > `&_p.x`
- File scope \Rightarrow storage area affixed with file name
- Lifetime is entire execution

Global

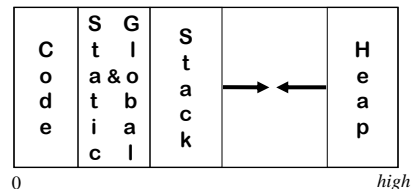
- One or more named global data areas
- One per variable, or per file, or per program, ...
- Lifetime is entire execution

CMSC430 Spring 2007

19

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

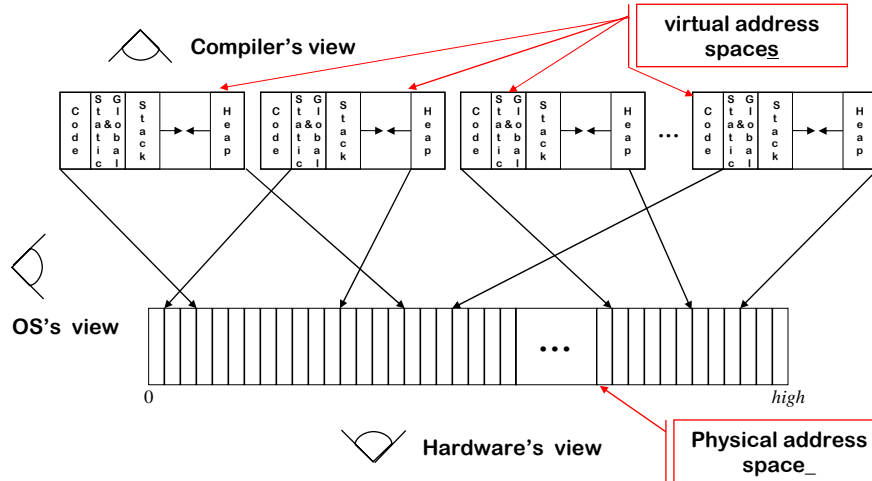
Figure 7.11 in EaC

CMSC430 Spring 2007

20

How Does This Really Work?

The Big Picture



CMSC430 Spring 2007

21

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 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).

CMSC430 Spring 2007

22

Translating Local Names

How does the compiler represent a specific instance of x ?

- Name is translated into a *static distance coordinate (s.d.c.)*
 - > $\langle \text{level}, \text{offset} \rangle$ pair
 - > “*level*” is lexical nesting level of the procedure
 - > “*offset*” is *unique* within that scope
- Subsequent code will use static distance 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
 - > Assigned at compile time
 - > Known at compile time
 - > Used to generate code that executes at run-time

CMSC430 Spring 2007

23

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 10 in the fixed data area
 - > The x declared at **level 2** will always be the eighth data item, stored at byte 14 in the fixed data area
 - > But what about the a declared in the second block at **level 2**?

CMSC430 Spring 2007

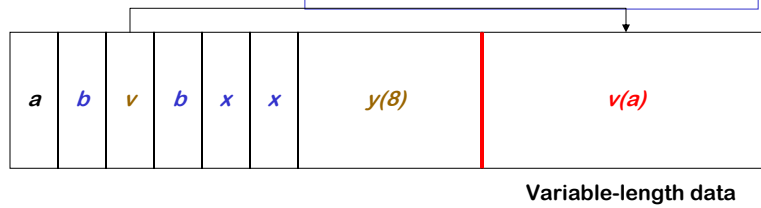
24

Variable-length Data

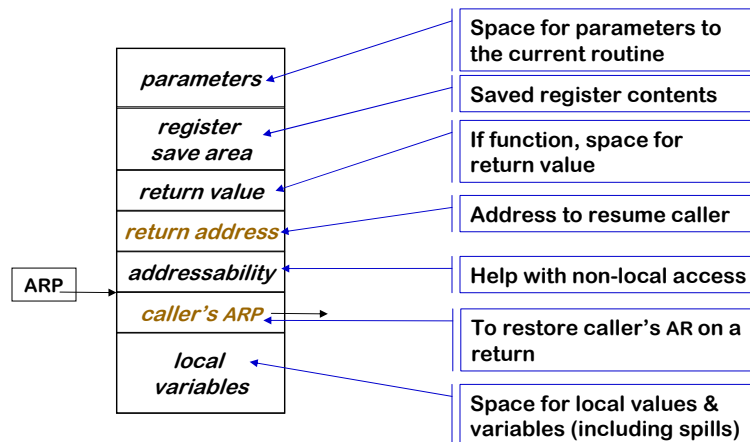
```

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 block in which it is allocated (includes fixed length areas for all contained blocks)



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 (*“offset” in its s.d.c.*)
- It can generate “loadAI” operations (*“level” must → AR*)

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

CMSC430 Spring 2007

27

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

CMSC430 Spring 2007

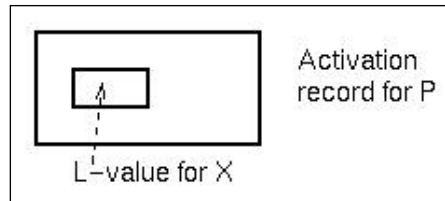
28

Subprogram control

- Remember that data storage for subprograms is in an activation record.

var X: integer;

- > X is of type integer.
- > L-value of X is some specific offset in an activation record.

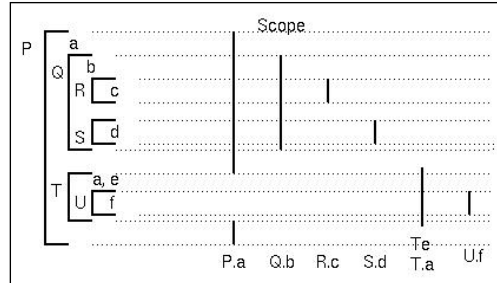


- Goal is to look at locating activation record for P.
- Given an expression: $X = Y + Z$
 - Locate activation record containing Y.
 - Get L-value of Y from fixed location in activation record.
 - Repeat process for Z and then X.

Scope rules

- Scope rules:** The scope of a variable are the set of statements where the variable may be accessed (i.e., named) in a program.
- Static scope:** Scope is dependent on the syntax of the program.
- Dynamic scope:** Scope is determined by the execution of the program.
- Static nested scope:** A variable is accessible in the procedure it is declared in, and all procedures internal to that procedure, except a new declaration of that variable name in an internal procedure will eliminate the new variable's scope from the scope of the outer variable.
- A variable declared in a procedure is **local** in that procedure; otherwise it is **global**.

Review of scope rules

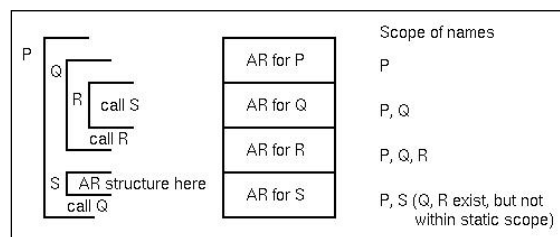


- Q and T are declarations of procedures within P, so scope of names Q and T is same as scope of declaration a.
- R and S are declarations of procedures in Q.
- U is a declaration of a procedure in T.
- **Storage managed by adding activation records, when procedure invoked, on a stack.**

CMSC430 Spring 2007

31

Activation record stack

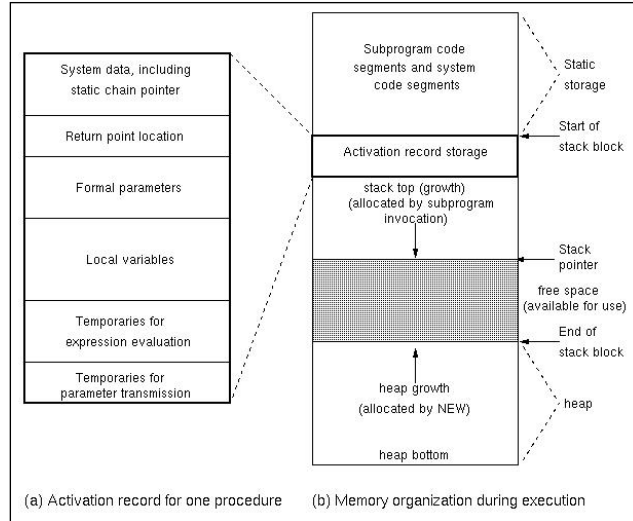


- **Problem is:** How to manage this execution stack?
- Two pointers perform this function:
 1. **Dynamic link pointer** points to activation record that called (invoked) the new activation record. It is used for returning from the procedure to the calling procedure.
 2. **Static link pointer** points to the activation record that is global to the current activation record (i.e., points to the activation record of the procedure containing the declaration of this procedure).

CMSC430 Spring 2007

32

Activation record structure

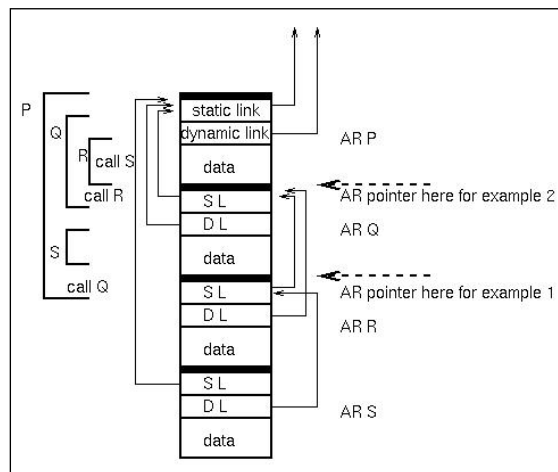


CMSC430 Spring 2007

33

Example of activation record stack

- For following examples, assume following static program and stack:



CMSC430 Spring 2007

34

Activation record example 1

Example 1. In R: $C := B+A$; \Rightarrow C local, A and B global

- For each variable, get pointer to proper activation record.
- Assume AR is current activation record pointer (R).
 1. B is **one** level back:
 - \rightarrow Follow AR.SL to get AR containing B.
 - \rightarrow Get R-value of B from fixed offset L-value
 2. A is **two** levels back:
 - \rightarrow Follow (AR.SL).SL to get activation record containing A.
 - \rightarrow Add R-value of A from fixed offset L-value
 3. C is **local**. AR points to correct activation record.
 - \rightarrow Store sum of B+A into L-value of C

Activation record example 2

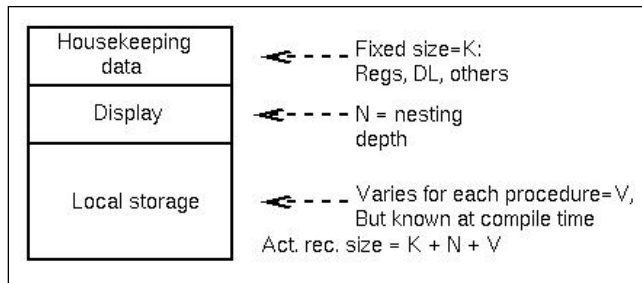
Example 2. Execution in procedure Q: $A := B$

\Rightarrow B is local, A global

- Assume AR is current activation record pointer (Q)
 1. B is **now** local. AR points to activation record
 - \rightarrow Get R-value from local activation record
 2. A is now **one** level back
- AR.SL is activation record of P
 - \rightarrow Store R-value of B into L-value of A
- **Compiler knows static structure, so it can generate the number of static link chains it has to access in order to access the correct activation record containing the data object. This is a compile time, not a runtime calculation.**

Use of displays

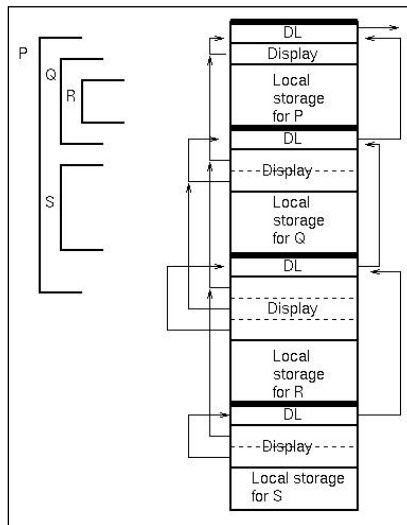
- Problem with static links:
- Many links to follow if deeply nested. (But how common is that?)
- Use of **displays** reduces access always to 2 instructions:
 1. Access Display[I] = Act. Rec. pointer at level I
 2. Get L-value of variable (fixed offset in act. rec.)
- Activation record structure:



CMSC430 Spring 2007

37

Display for previous example



CMSC430 Spring 2007

38

Accessing variables

- **Data access is always a 2-step process:**
 1. [If non-local] Load Reg1, Display[I]
 2. [For all] Load Reg2, L-value offset (Reg1)
- **Size of activation record for a procedure:**
 - > Housekeeping storage (Fixed size = K):
 - Dynamic Link
 - Return address (in source program)
 - End of activation record
 - Registers
 - > Display
 - > Local data storage

Creating activation records

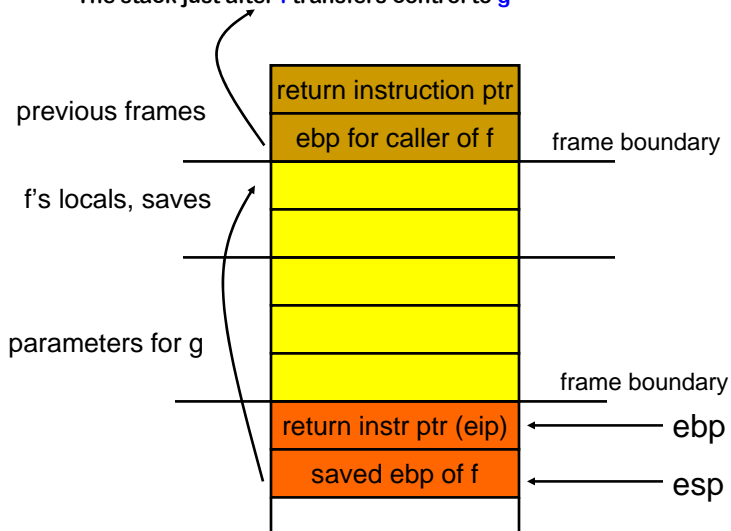
- **To create new activation record (e.g., P calls Q):**
 - > Assume have pointer to P's activation record.
 1. Get EndStack pointer from P's activation record.
 - This is start of Q's activation record.
 2. Add K+NewDisplaySize+LocalStorageSize.
 3. Save as new EndStack in Q's activation record.
 4. Set DL in Q to point to old activation record (P).
 5. Fill in housekeeping in Q's activation record.
 6. Set up new display in Q's activation record.
- **To create a new display to call from level I to level J:**
 - > Copy Display[1..J-1] from calling activation record.
 - > Add new activation record pointer as Display[J].
- **To return:**
 - > Use dynamic link in current activation record to get old activation record.
 - > Use return address in housekeeping area to jump to.

Machine Model (x86)

- The CPU has a fixed number of *registers*
 - > Think of these as memory that's really fast to access
 - > For a 32-bit machine, each can hold a 32-bit word
- Important x86 registers
 - > **eax** generic register for computing values
 - > **esp** pointer to the top of the stack
 - > **ebp** pointer to start of current stack frame
 - > **eip** the program counter (points to next instruction in text segment to execute)

The x86 Stack Frame/Activation Record

- The stack just after **f** transfers control to **g**



x86 Calling Convention

- To call a function
 - > Push parameters for function onto stack
 - > Invoke **CALL** instruction to
 - Push current value of **eip** onto stack
 - I.e., save the program counter
 - Start executing code for called function
 - > Callee pushes **ebp** onto stack to save it
- When a function returns
 - > Put return value in **eax**
 - > Invoke **LEAVE** to pop stack frame
 - Set **esp** to **ebp**
 - Restore **ebp** that was saved on stack and pop it off the stack
 - > Invoke **RET** instruction to load return address into **eip**
 - I.e., start executing code where we left off at call

CMSC430 Spring 2007

43

Example

```
int f(int a, int b) {
    return a + b;
}

int main(void) {
    int x;

    x = f(3, 4);
}
```

gcc -S a.c

```
f:
    pushl   %ebp
    movl   %esp, %ebp
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    leave
    ret

main:
    ...
    subl   $8, %esp
    pushl   $4
    pushl   $3
    call   f
1:  addl   $16, %esp
    movl   %eax, -4(%ebp)
    leave
    ret
```

CMSC430 Spring 2007

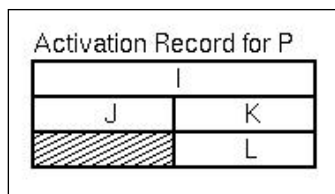
44

Blocks in C

- Blocks in C are handled as union record types:

```
P()
{ int I;
  {int J;
    ...}
  { int K; int L;
    ...}
  ... }
```

- J and K cannot exist at the same time, so use the same space for them.



CMSC430 Spring 2007

45

Optimizations in activation record design

1. Instead of displays, use M registers only:

E.g., M1, M2, M3, M4 only.

- **What if a procedure is nested 5 deep?**
 1. Illegal program?
 2. Spill to displays only in this case?

2. Note that C was designed for efficient execution:

- > No nested procedures
- > No need for displays or static links
- > But needs dynamic links.

CMSC430 Spring 2007

46

C++ storage access

- General properties
 1. Procedure allocations follow general rules of C
 - > Allocate activation records on a stack
 - > No need for static links, but need dynamic links
 2. Class variables are allocated like structs in C.
 - > [Structs in C++ can have function members. Functions in structs are by default public, while in a class are by default private.]
 3. Static functions are not represented in runtime activation records
 4. Objects requiring dynamic binding (i.e., virtual functions) are present in runtime activation records

Sample C++ allocation example

```
class A {
public: int x;
    virtual void p() {...};
    void q() {...p()...};
protected: int y;
private: int Z;
...}
class B:A {
public: int w;
    virtual void p() {...};
    void r() {...p()...};
private: int v;
    virtual void s() {...};
...}
```

Sample C++ allocation example -- Symbol table for example

<pre> class A { public: int x; virtual void p() {...}; void q() {...p()...}; protected: int y; private: int Z; ...} class B:A { public: int w; virtual void p() {...}; void r() {...p()...}; private: int v; virtual void s() {...}; ...} </pre>	<table border="0"> <thead> <tr> <th>NmType</th> <th>Access</th> <th>Mode</th> <th>Location</th> </tr> </thead> <tbody> <tr><td>X</td><td>var</td><td>public</td><td>static</td><td>A(1)</td></tr> <tr><td>P</td><td>func</td><td>public</td><td>dynamic</td><td>A(2)</td></tr> <tr><td>Q</td><td>func</td><td>public</td><td>static</td><td>addr of proc</td></tr> <tr><td>Y</td><td>var</td><td>protected</td><td>static</td><td>A(3)</td></tr> <tr><td>Z</td><td>var</td><td>private</td><td>static</td><td>A(4)</td></tr> <tr><td>W</td><td>var</td><td>public</td><td>static</td><td>B(5)</td></tr> <tr><td>P</td><td>func</td><td>public</td><td>dynamic</td><td>B(2)</td></tr> <tr><td>R</td><td>func</td><td>public</td><td>static</td><td>Addr of proc</td></tr> <tr><td>V</td><td>var</td><td>private</td><td>static</td><td>B(6)</td></tr> <tr><td>S</td><td>func</td><td>private</td><td>dynamic</td><td>B(7)</td></tr> </tbody> </table>	NmType	Access	Mode	Location	X	var	public	static	A(1)	P	func	public	dynamic	A(2)	Q	func	public	static	addr of proc	Y	var	protected	static	A(3)	Z	var	private	static	A(4)	W	var	public	static	B(5)	P	func	public	dynamic	B(2)	R	func	public	static	Addr of proc	V	var	private	static	B(6)	S	func	private	dynamic	B(7)
NmType	Access	Mode	Location																																																				
X	var	public	static	A(1)																																																			
P	func	public	dynamic	A(2)																																																			
Q	func	public	static	addr of proc																																																			
Y	var	protected	static	A(3)																																																			
Z	var	private	static	A(4)																																																			
W	var	public	static	B(5)																																																			
P	func	public	dynamic	B(2)																																																			
R	func	public	static	Addr of proc																																																			
V	var	private	static	B(6)																																																			
S	func	private	dynamic	B(7)																																																			

Class A

X: R-value
P: Addr of P in A
Y: R-value
Z: R-value

Class B

X: R-value
P: Addr of P in B
Y: R-value
Z: R-value
W: R-value
V: R-value
S: Addr of S in B

Parameter passing

- **Parameter:** A variable in a procedure that represents some other data from the procedure that invoked the given procedure.
- **Parameter transmission:** How that information is passed to the procedure.
 - > The parameter is also called the **formal argument**.
 - > The data from the invoking procedure is called the **actual argument** or sometimes just the **argument**.
- **Usual syntax:**
 - > Actual arguments: call P(A, B+2, 27+3)
 - > Parameters: Procedure P(X, Y, Z)
 - > What is connection between the parameters and the arguments?
 - Call by name
 - Call by reference
 - Call by value
 - Call by result (or value-result)

Language dependent

- Difference languages have different mechanisms:
 - > ALGOL - name, value
 - > Pascal - value, reference
 - > C - value (BUT pointers give us reference)
- Constant tension between desire for efficiency and semantic correctness in defining parameter transmission.

Call by name

- Substitute argument for parameter at each occurrence of parameter:
 - Invocation:** $P(A, B+2, 27+3)$
 - Definition:** procedure $P(X,Y,Z) \{ \text{int } I; I=7; X = I + (7/Y)^*Z; \}$
 - Meaning:** $P(X,Y,Z) \{ \text{int } I; I=7; A=I+(7/(B+2))^*(27+3); \}$
- This is a true macro expansion. Simple semantics, **BUT:**
 1. **Implementation.** How to do it?
 2. **Aliases.** What if statement of P were: $I = A$?
 3. **Expressions versus statements:** If we had $D=P(1,2,3)$ and a *return(42)* in P, what does semantics mean?
 4. **Error conditions:** $P(A+B, B+2, 27+3)$

Implementation of call by name

- A **thunk** is the code which computes the L-value and R-value of an argument.
 - > For each argument, pass code address that computes both L-values and R-values of arguments.
- **P(A, B+2, 27+3) generates:**
 - jump to subroutine P
 - address of thunk to return L-value(A)
 - address of thunk to return R-value(A)
 - address of thunk to return L-value(B+2)
 - address of thunk to return R-value(B+2)
 - address of thunk to return L-value(27+3)
 - address of thunk to return R-value(27+3)
 - > To assign to X, call thunk 1, To access X, call thunk 2
 - > To assign to Y, call thunk 3, To access Y, call thunk 4
 - > To assign to Z, call thunk 5, To access Z, call thunk 6
- **Issue:** Assignment to (B+2): How?
- Call by name is conceptually convenient, but inefficient.

CMSC430 Spring 2007

53

Examples of Call by Name

1. P(x) {x = x + x;}
Seems simple enough ...
Y = 2; P(Y); write(Y) ⇒ means Y = Y+Y
write(Y) ⇒ prints 4
2. int A[10];
for(l=0; l<10; l++) {A[l]=l;};
l=1; P(A[l]) ⇒ A[1] = A[1] + A[1] ⇒ A[1] set to 2
3. **But:** F(){l = l + 1; return l;}
What is: P(A[F()])?
P(A[F()]) ⇒ A[F()] = A[F()] + A[F()] ⇒ A[l++] = A[l++] + A[l++]
 ⇒ A[2] = A[3] + A[4]
4. Write a program to exchange values of X and Y: (swap(X,Y))
 - > Usual way: swap(x,y) {t=x; x=y; y=t;}
 - > Cannot do it with call by name. Cannot handle both of following: swap(l, A[l]) swap(A[l],l)
 - > One of these must fail.

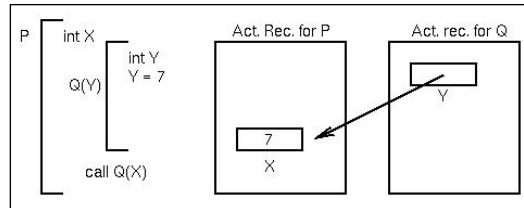
CMSC430 Spring 2007

54

Call by reference

- Pass the L-value of the argument for the parameter.
- **Invocation:** P(A, B+2, 27+3)

- **Implementation:**
Temp1 = B+2
Temp2 = 27+3
jump to subroutine P
L-value of A
L-value of Temp1
L-value of Temp2

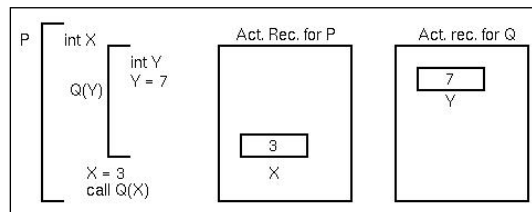


- This is the most common parameter transmission mechanism. In the procedure activation record, parameter X is a local variable whose R-value is the L-value of the argument.

Call by value

- Pass the R-value of the argument for the parameter.
- **Invocation:** P(A, B+2, 27+3)

- **Implementation:**
Temp1 = B+2
Temp2 = 27+3
jump to subroutine P
R-value of A
R-value of Temp1
R-value of Temp2



- In procedure activation record, parameter X is a local variable whose R-value is the R-value of the argument.

Call by reference in C

- C only has call by value,
- BUT pointer variables allow for simulating call by reference:
 - $P(i, j) \Rightarrow$ passes i and j by value.
 - $P(\&i, \&j) \Rightarrow$ passes L-values of i and j .
 - $P(*x, *y) \{ *x = *y + 1; \} \Rightarrow$ arguments are addresses (pointers)
- **Call by result** (or value-result): Call by value, AND pass back the final value to argument upon return.
 - > Parameter is a local value in procedure.
 - > Similar to call by reference, except for aliasing.

In-out semantics

- Parameters in Ada are based upon use (semantics), not implementation:
 - > in - argument value will be used in procedure.
 - > out - parameter value will be used in calling program.
 - > in out - both uses of arguments and parameters

```
P(X in integer;  
  Y out integer;  
  Z in out integer);  
begin ... end;
```
- In Ada 83, language definition allowed some latitude in implementation \Rightarrow as long as implementation consistent, ok.
 - > **But** this meant that the same program could give different answers from different standards conforming compilers
 - > In Ada 95, more restricted: in integer is value, out integer is value-result, composite (e.g., arrays) is reference.

Example of parameter passing

```

Main
{A = 2; B = 5; C = 8; D = 9;
 P(A, B, C, D); write(A, B, C, D);
P(U, V, W, X)
{V = U+A;
 W = A+B;
 A = A+1;
 X = A+2;
 write(U, V, W, X)}

```

Fill in table assume parameters are of the given type:

	A	B	C	D	U	V	W	X	print P	print main
Call by name										
Call by reference										
Call by value										
Call by result										

When do call by name and call by reference differ?

When L-value can change between parameter references. E.g., P(I, A[I])

Establishing Addressability

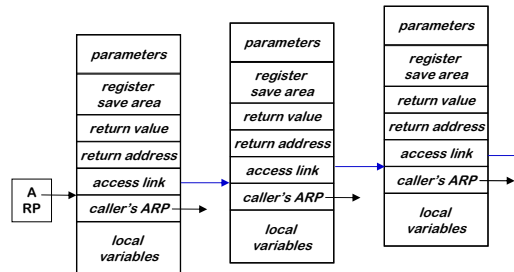
Must create base addresses

- Global & static variables
 - > Construct a label by mangling names (*i.e.*, `&_fee`)
 - Local variables
 - > Convert to static data coordinate and use `ARP + offset`
 - Local variables of other procedures
 - > Convert to s.d.c.
 - > Find appropriate ARP
 - > Use that `ARP + offset`
- }
 - Must find the right AR
 - Need links to nameable ARs

Establishing Addressability

Using access links

- Each AR has a pointer to AR of lexical ancestor
- Lexical ancestor need not be the caller



Some setup cost on each call

- Reference to $\langle p, 16 \rangle$ runs up access link chain to p
- Cost of access is proportional to lexical distance

CMSC430 Spring 2007

61

Establishing Addressability

Using access links

SDC	Generated Code
$\langle 2, 8 \rangle$	loadAI $r_0, 8 \Rightarrow r_2$
$\langle 1, 12 \rangle$	loadAI $r_0, -4 \Rightarrow r_1$ loadAI $r_1, 12 \Rightarrow r_2$
$\langle 0, 16 \rangle$	loadAI $r_0, -4 \Rightarrow r_1$ loadAI $r_1, -4 \Rightarrow r_1$ loadAI $r_1, 16 \Rightarrow r_2$

Assume

- Current lexical level is 2
- Access link is at ARP - 4

Maintaining access link

- Calling level $k+1$
→ Use current ARP as link
- Calling level $j < k$
→ Find ARP for $j-1$
→ Use that ARP as link

Access & maintenance cost varies with level

All accesses are relative to ARP

(r_0)

CMSC430 Spring 2007

62

Establishing Addressability

Using a display

- Global array of pointer to nameable ARs
- Needed ARP is an array access away

Some setup cost on each call

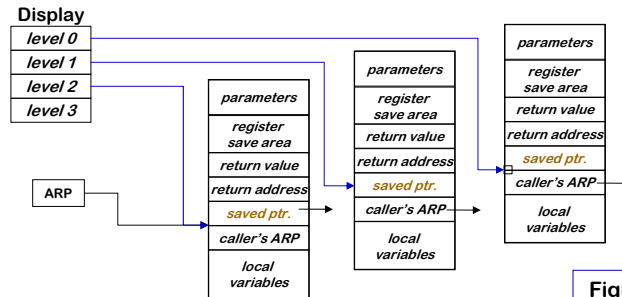


Figure 7.8 in EaC

- Reference to $\langle p, 16 \rangle$ looks up p 's ARP in display & adds 16
- Cost of access is constant ($ARP + \text{offset}$)

CMSC430 Spring 2007

63

Establishing Addressability

Using a display

SDC	Generated Code
$\langle 2, 8 \rangle$	loadAI $r_0, 8 \Rightarrow r_2$
$\langle 1, 12 \rangle$	loadI $_disp \Rightarrow r_1$ loadAI $r_1, 4 \Rightarrow r_1$ loadAI $r_1, 12 \Rightarrow r_2$
$\langle 0, 16 \rangle$	loadI $_disp \Rightarrow r_1$ loadAI $r_1, 16 \Rightarrow r_2$

Desired AR is at $_disp + 4 \times \text{level}$

Assume

- Current lexical level is 2
- Display is at label $_disp$

Maintaining access link

- On entry to level j
 - Save level j entry into AR (Saved Ptr field)
 - Store ARP in level j slot
- On exit from level j
 - Restore level j entry

Access & maintenance costs are fixed

Address of display may consume a register

CMSC430 Spring 2007

64

Establishing Addressability

Access links versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
 - > Overhead only incurred on references & calls
 - > If ARs outlive the procedure, access links still work
- Display costs are fixed for all references
 - > References & calls must load display address
 - > Typically, this requires a register *(rematerialization)*

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

For either scheme to work, the compiler must insert code into each procedure call & return

Procedure Linkages

How do procedure calls actually work?

- At compile time, callee may not be available for inspection
 - > Different calls may be in different compilation units
 - > Compiler may not know system code from user code
 - > All calls must use the same protocol

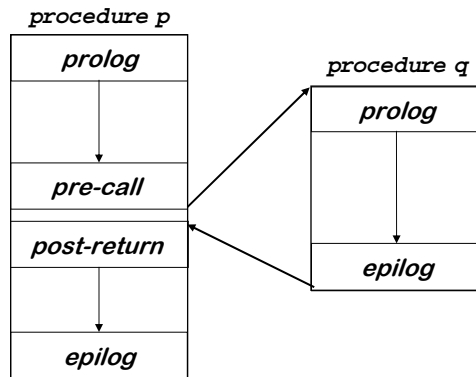
Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement *(for interoperability)*

Procedure Linkages

Standard procedure linkage



Procedure has

- standard **prolog**
- standard **epilog**

Each call involves a

- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the call site \Rightarrow depend on the number & type of the actual parameters

Procedure Linkages

Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

The details

- Allocate space for the callee's AR (except space for local variables)
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP into callee's AR
- If access links are used
 - > Find appropriate lexical ancestor & copy into callee's AR
- Save any caller-save registers
 - > Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
 - > Also copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code

- Finish setting up the callee's environment
- Preserve parts of the caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- If display is being used
 - > Save display entry for current lexical level
 - > Store current ARP into display for current lexical level
- Allocate space for local data
 - > Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee
- Handle any local variable initializations

With heap allocated AR,
may need to use a
separate heap object for
local variables

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)

The Details

- Store return value? No, this happens on the return statement
- Restore callee-save registers
- Free space for local data, if necessary (**on the heap**)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

What about Object-Oriented Languages?

What is an OOL?

- A language that supports "object-oriented programming"

How does an OOL differ from an ALL? (ALGOL-Like Language)

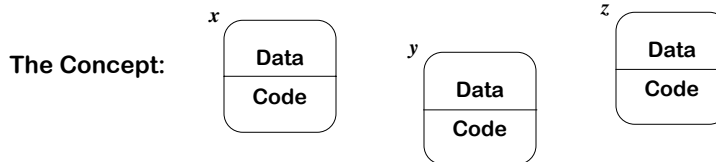
- Data-centric name scopes for values & functions
- Dynamic resolution of names to their implementations

How do we compile OOLs ?

- Need to define what we mean by an OOL
- Term is almost meaningless today — Smalltalk to C++ to Java
- We will focus on Java and C++

Object-Oriented Languages

An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.



Elaborating the concepts:

- Each object needs local storage for its attributes
 - > Attributes are static (lifetime of object); access is through methods
- Some methods are public, others are private
 - > More complex linkage convention; locating them is complex
- Object's internal state leads to complex behavior

CMSC430 Spring 2007

73

OOLs & the Procedure Abstraction

What is the shape of an OOL's name space?

- Local storage in objects (beyond attributes)
- Some storage associated with methods
 - > Local values inside a method
 - > Static values with lifetimes beyond methods
- Methods shared among multiple objects

Classes

- Objects with the same state are grouped into a *class*
 - > Same attributes, instance variables, & methods
 - > Class variables are static, shared among all objects of same class
- Allows programmer to write it down once
- Allows code reuse at both source & implementation level

CMSC430 Spring 2007

74

Implementing Object-Oriented Languages

So, what can an executing method see?

- The object's own attributes & private variables
 - > Smalltalk terminology: *instance variables*
- The attributes & private variables of the class where it is defined
 - > Smalltalk terminology: *class variables*
- Any object defined in the global name space (or scope)
 - > Objects may contain other objects (!?!)

An executing method might reference any of these

A final twist:

- Most OOLs support a hierarchical notion of inheritance
- Some OOLs support multiple inheritance
 - > More than one path through the inheritance hierarchy

Java Name Space

What is the shape of Java's name space?

Suppose I am compiling a particular method M for an object O within a class C—I can see:

- Local variables within M (block scoped)
- All instance variables for O and class variables of the class C
- All public and protected variables of any superclass of C
- Any classes defined in the same package or in a package imported
 - > public class variables and instance variables of those classes
 - > package class and instance variables in the same package
- Class declarations can be nested!
 - > These member declarations hide outer class declarations of the same name
 - > Accessibility: public, private, protected, package

Java Symbol Tables I

To compile code in method **M** for an object **O** within a class **C**, the compiler needs:

- Stack symbol table for block and class nesting
 - > Just like ALL — inner declarations hide outer declarations
- Symbol tables for all global classes (package scope)
 - > Entries for all members with visibility
 - > Need to construct symbol table for imported packages
- Chain of symbol tables for inheritance
 - > Need mechanism to find the class and instance variables of all superclasses

Java Symbol Tables I

When compiling access code for a variable use in method **M** for an object **O** within a class **C**, the compiler needs:

- For an unqualified use (i.e., **x**):
 - > Search the stack oriented symbol table for the current method
 - > Search the chain of symbol tables for the class hierarchy
 - > Search global symbol table (current package and imported)
 - Look for class (or interface)
 - > In each case check visibility attribute of **x**
- For a qualified use (i.e.: **Q.x**):
 - > Search stack-structured global symbol table for class nesting for **Q**
 - > Check visibility attribute of **x**

Implementing Object-Oriented Languages

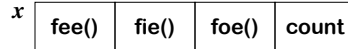
To implement OOLs, the compiler must address two critical issues

- Object representation
- Mapping a **method invocation** name to a **method implementation**

These both are intimately related to the OOL's name space

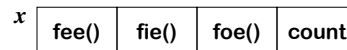
Object Representation

- Static, private storage for attributes & instance variables
 - > Heap allocate object records or "instances"
- Need consistent, fast access
 - > Known, constant offsets
- Provision for initialization in NEW



OOL Storage Layout (Java)

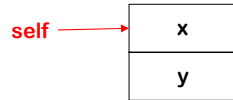
- Class variables
 - > Static class storage accessible by global name
 - Accessible via linkage symbol &_C
 - Nested classes are handled like blocks in ALLs
 - Method code can be at fixed offset from start of class area
- Object Representation
 - > Object storage heap allocated
 - Fields at fixed offsets from start of object storage
 - > Methods
 - Method code storage associated with class
 - Methods accessed by offsets from code vector
 - method references in line
 - Method local storage in object (no calls) or on stack



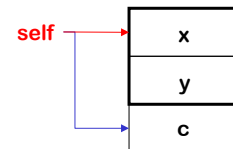
Dealing with Single Inheritance

- Use **prefixing** of storage

```
Class Point {  
  int x, y;  
}
```



```
Class ColorPoint extends Point {  
  Color c;  
}
```



Does casting work properly?

Implementing Object-Oriented Languages

Mapping message names to methods

- Static mapping, known at compile-time (Java, C++)
 - > Fixed offsets & indirect calls
- Dynamic mapping, unknown until run-time
 - > Look up name in class' table of methods

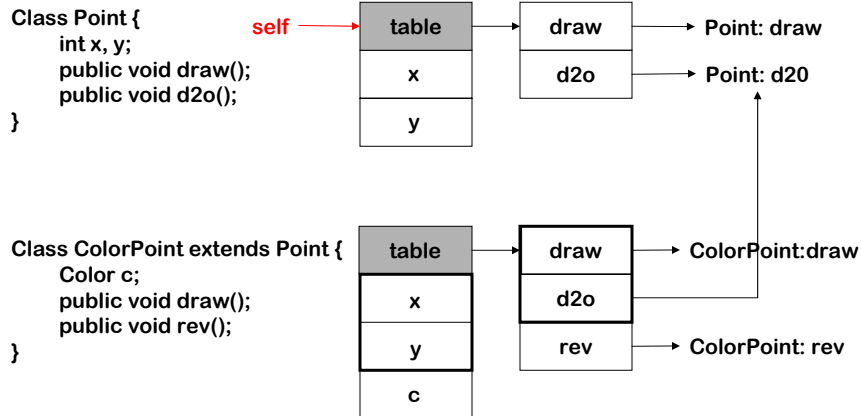
Want uniform placement of standard services *(NEW, PRINT, ...)*

This is really a data-structures problem

- Build a table of function pointers
- Use a standard invocation sequence

Single Inheritance and Dynamic Dispatch

- Use **prefixing** of tables



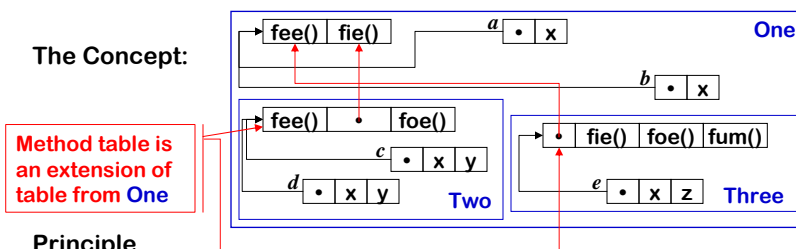
CMSC430 Spring 2007

83

The Inheritance Hierarchy

To simplify object creation, we allow a class to inherit methods from an ancestor, or *superclass*. The descendant class is called the *subclass* of its ancestor.

The Concept:



Principle

- B subclass $A \Rightarrow d \in B$ can be used wherever $e \in A$ is expected,
 - B may override a method definition from A

Subclass provides all the interfaces of superclass,

CMSC430 Spring 2007

84

The Inheritance Hierarchy

Two distinct philosophies

Static class structure

- Can map name to code at compile time
- Leads to 1-level jump vector
- Copy superclass methods
- Fixed offsets & indirect calls
- Less flexible & expressive

Dynamic class structure

- Cannot map name to code at compile time
- Multiple jump vector (1/class)
- Must search for method
- Run-time lookups caching
- Much more expensive to run

Impact on name space

- Method can see instance variables of self, class, & superclasses
- Many different levels where a value can reside

In essence, OOL differs from ALL in the shape of its name space
AND in the mechanism used to bind names to implementations

Multiple Inheritance

The idea

- Allow more flexible sharing of methods & attributes
- Relax the inclusion requirement
 - If B is a subclass of A, it need not implement all of A's methods
- Need a linguistic mechanism for specifying partial inheritance

Problems when C inherits from both A & B

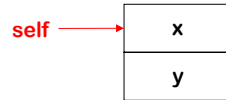
- C's method table can extend A or B, but not both
 - > Layout of an object record for C becomes tricky
- Other classes, say D, can inherit from C & B
 - > Adjustments to offsets become complex
- Both A & B might provide fum() — which is seen in C ?
 - > C++ produces a "syntax error" when fum() is used

Need a better way
to say "inherit"

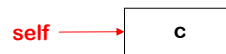
Multiple Inheritance Example

- Use **prefixing** of storage

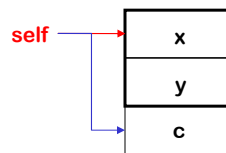
```
Class Point {
  int x, y;
}
```



```
Class ColoredThing {
  Color c;
}
```



```
Class ColorPoint extends
  Point, ColoredThing {
}
Does casting work properly?
```



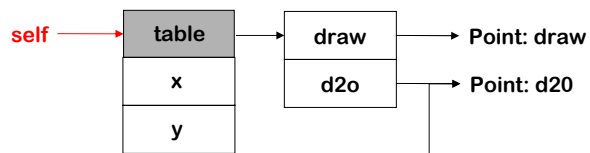
CMSC430 Spring 2007

87

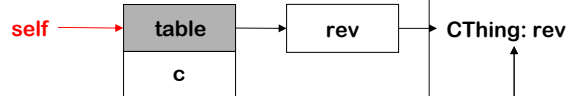
Multiple Inheritance Example

- Use **prefixing** of storage

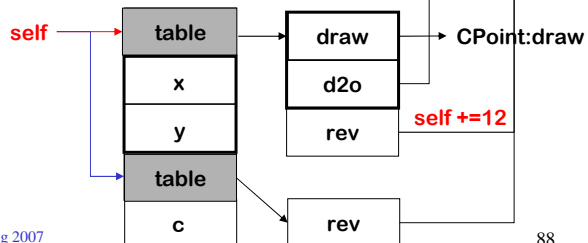
```
Class Point {
  int x, y;
  void draw();
  void d2o();
}
```



```
Class CThing {
  Color c;
  void rev();
}
```



```
Class Cpoint extends
  Point, CThing {
  void draw()
}
```



CMSC430 Spring 2007

88

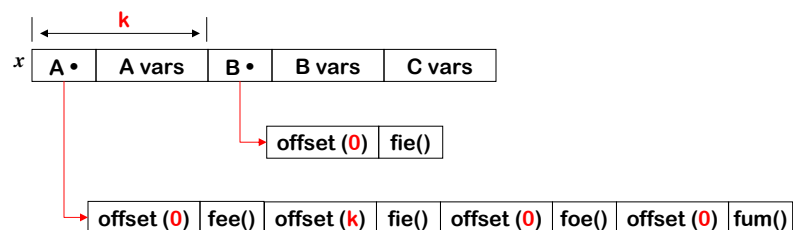
Casting with Multiple Inheritance

- Usage as Point:
 - > No extra action (prefixing does everything)
- Usage as CThing:
 - > Increment **self** by 12
- Usage as CPoint:
 - > Lay out data for CThing at **self + 16**
 - > When calling **rev**
 - Call in table points to a trampoline function that adds 12 to **self**, then calls **rev**
 - Ensures that **rev**, which assumes that **self** points to a CThing data area, gets the right data

Multiple Inheritance with Offsets (Example)

Assume that C inherits `fee()` from A, `fie()` from B, & defines both `foe()` & `fum()`.

Object record for an instance of C



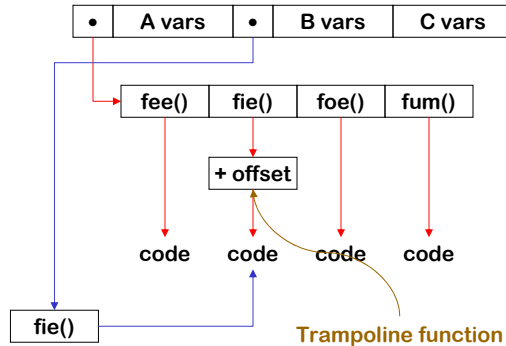
To make this work, calls must add offset to self

Works, but adds overhead to each method invocation

Multiple Inheritance (Example)

Assume that C inherits fee() from A, fie() from B, & defines both foe() and fum()

Object record for an instance of C



This implementation

- Uses **trampoline functions**
- Optimizes well with inlining
- Adds overhead where needed (Zero offsets go away)
- Folds inheritance into data structure, rather than linkage

Assumes static class structure
For dynamic, why not rebuild on a change in structure?

CMSC430 Spring 2007

91

Implementing Object-Oriented Languages

So, what can an executing method see? (*Reprise*)

- The object's own attributes & private variables
- The attributes & private variables of the classes that define it
 - > *May be many such classes, in many combinations*
 - > *Class variables are visible to methods that inherit the class*
- Object defined in the global name space (or scope)
 - > Objects may contain other objects (!?!)
- Objects that contain their definition
 - > *A class as an instance variable of another class, ...*

An executing method might reference any of these

Making this work requires compile-time elaboration for static case and run-time elaboration for dynamic case

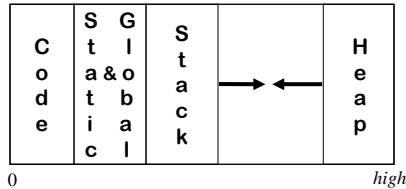
Making it run quickly takes care, planning, and trickery ... (later)

CMSC430 Spring 2007

92

Memory Layout

Placing run time data structures



- Stack & heap share free space
- Fixed-size areas together
- For compiler, this is the entire picture

Alignment & padding

- Both languages & machines have alignment restrictions
- Place values with identical restrictions next to each other
- Assign offsets from most restrictive to least
- Insert padding to match restrictions *(if needed)*

CMSC430 Spring 2007

93

Impact of Memory Model on Code Shape

Memory-to-memory model

- Compiler works within constraints of register set
- At each statement, compiler ensures demand $< k$
- Each variable has a location in memory
- Register allocation becomes an optimization

Register-to-register model

- Compiler works with an unlimited set of virtual registers
- Compiler largely ignores relationship between demand & k
- Only allocate memory for spills, parameters, & ambiguous values
 - > Complex data structures (arrays) will be assigned to memory
- Register allocation is needed for correctness *(names)*

CMSC430 Spring 2007

94

Cache Performance & Relative Offsets

Principle:

- Two items referenced together
 - > Would like them in same cache block, *or*
 - > Would like them to map into different cache sets
- This notion is sometimes called “cache packing”
- Manageable problem for two variables
- Complexity rises immensely with more variables

Cache organization

- Virtual address tags \Rightarrow cache behavior based on compiler’s model (cache blocks associate by virtual address)
- Physical tags \Rightarrow no relationship if distance \geq page size

Heap management

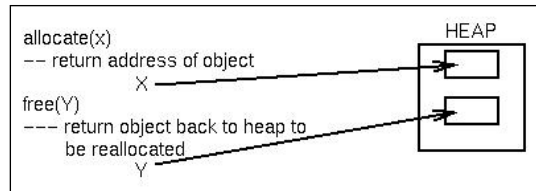
Allocate() & Free()

- Implementing these requires attention to detail
- Watch allocation & free cost, as well as fragmentation
- Many classic algorithms (first fit, first fit with rover, best fit)

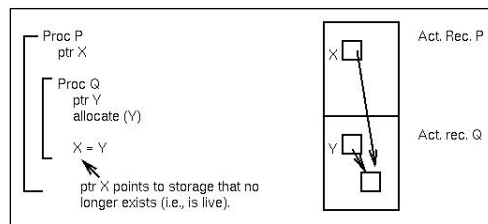
Implicit deallocation

- Humans are bad at writing calls to free()
- Major source of run-time problems
- Solution is to automate deallocation
 - > Reference counting + automatic free()
 - > Garbage collection

Heap storage



- Dynamic allocation and stacks are generally incompatible.

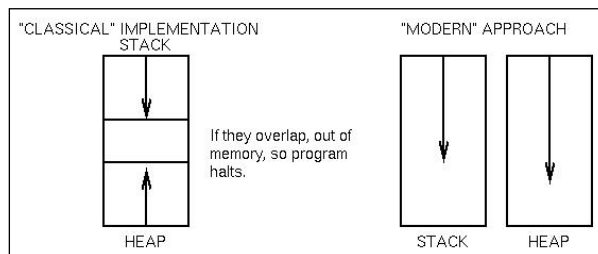


CMSC430 Spring 2007

97

Stack and heap location

- Pointer X points to stack storage in procedure Q's activation record that no longer is live (exists) when procedure Q terminates. Such a reference is called a **dangling reference**.
- Dynamic storage is usually a separate structure in C, Ada, Pascal
...



CMSC430 Spring 2007

98

Heap storage problems

- Major storage for ML, LISP and Prolog is a heap.
Get storage - `allocate(x)`
free storage - `free(x)`

- Problems:**

- > **Dangling reference:**

```
allocate(x); y = x;  
free(x);
```

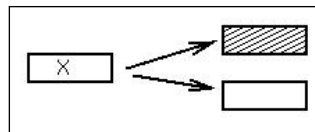
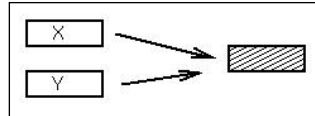
⇒ y still points to allocated

- > **Inaccessible storage:**

```
allocate(x);  
allocate(x);
```

⇒ first allocation to x now lost.

- > **Memory fragmentation** ... (next slide)

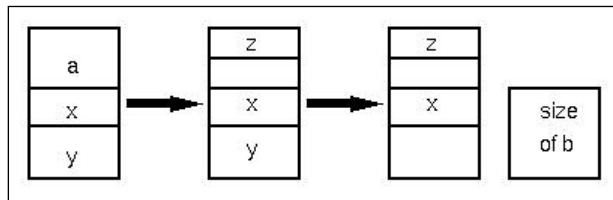


Memory fragmentation

- Memory fragmentation:**

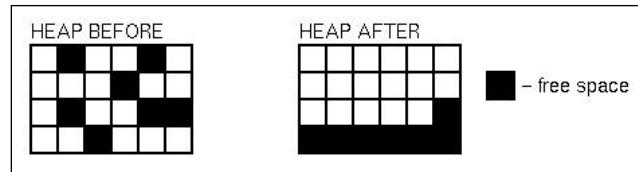
```
allocate(a);  
allocate(x);  
allocate(y);  
free(a);  
allocate(z);  
free(y);  
allocate(b);
```

⇒ No contiguous space for b



Garbage collection goal

- Process to reclaim memory. (Solve **Fragmentation problem**.)



- **Algorithm:** You can do garbage collection if you know where every pointer is in a program. If you move the allocated storage, simply change the pointer to it.
 - > This is true in LISP, ML, Java, Prolog
 - > Not true in C, C++, Pascal, Ada

Garbage collection: Mark-sweep algorithm

- First assume fixed size blocks of size k . (Later blocks of size $n*k$.)
 - > This is the LISP case.
- Two simple algorithms follow. (This is only an introduction to this topic. Many other algorithms exist)

Algorithm 1: Fixed size blocks;

- Two pass mark-sweep algorithm:
 1. Keep a linked list (free list) of objects available to be allocated.
 2. Allocate objects on demand.
 3. Ignore free commands.
 4. When out of space, perform garbage collection:
 - Pass 1. Mark every object still allocated.
 - Pass 2. Every unmarked objects added to free list of available blocks.

Heap storage errors

- **Error conditions:**
 - > Dangling reference - Cannot occur. Each valid reference will be marked during sweep pass 1.
 - > Inaccessible storage - Will be reclaimed during sweep pass 2.
 - > Fragmentation - Does not occur; fixed size objects.

CMSC430 Spring 2007

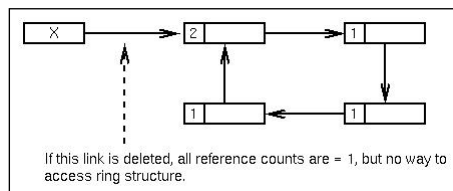
103

Garbage collection: reference counts

Algorithm 2:

1. Associate a reference counter field, initially 0, with every allocated object.
2. Each time a pointer is set to an object, up the reference counter by 1
3. Each time a pointer no longer points to an object, decrease the reference counter by 1.
4. If reference counter ever is to 0, then free object.

- **Error conditions:**
 - > Dangling reference - Cannot occur. Reference count is 0 only when nothing points to it.
 - > Fragmentation - Cannot occur. All objects are fixed size.
 - > Inaccessible storage - Can still occur, but not easily.

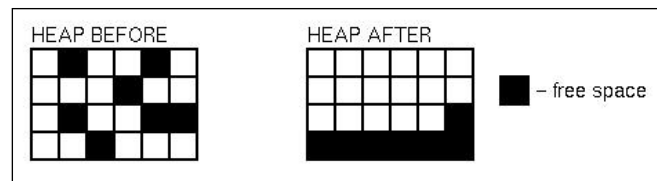


CMSC430 Spring 2007

104

Memory compaction: Variable-size elements

- With variable sized blocks, the previous two algorithms will not work.
- In the left heap below, a block of size 3 cannot be allocated, even though 7 blocks are free.
- If the allocated blocks are moved to the start of the heap (compaction) and the free blocks collected, then a block of size up to 7 can be allocated.



CMSC430 Spring 2007

105

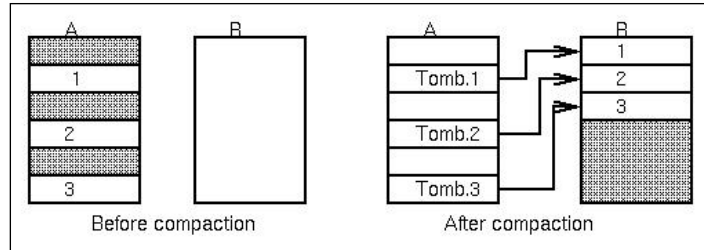
Compaction algorithm

- For variable sized blocks, in pass 2 of the mark-sweep algorithm:
 - ⇒ Move blocks to top of storage
 - ⇒ Need to reset pointers that point to the old storage location to now point to the new storage.
- **How to do this?**
 - > Use **tombstones** and two storage areas, an A area and a B area:
 1. Fill area A.
 2. When A fills, do mark-sweep algorithm.
 3. Move all allocated storage to start of B area. Leave marker in A area so other pointers pointing to this object can be modified.
 4. After everything moved, area A can be discarded. Allocate in B and later compact from B back to A.

CMSC430 Spring 2007

106

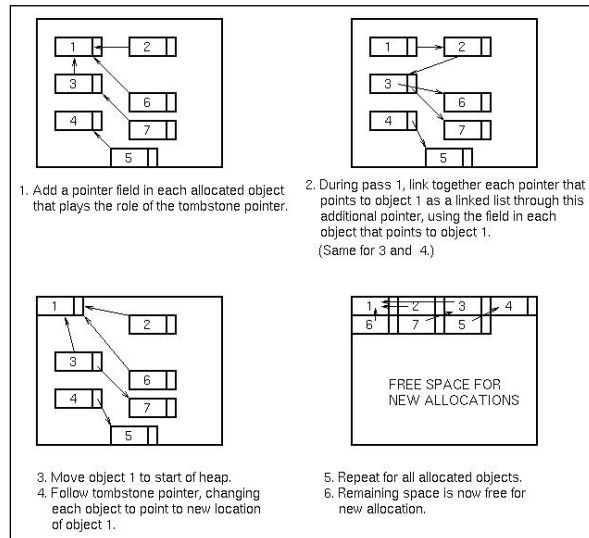
Compaction algorithm (continued)



CMSC430 Spring 2007

107

In place compaction



CMSC430 Spring 2007

108