

# Names, Scope, and Bindings

COMS W4115



Prof. Stephen A. Edwards

Fall 2003

Columbia University

Department of Computer Science

# What's In a Name?

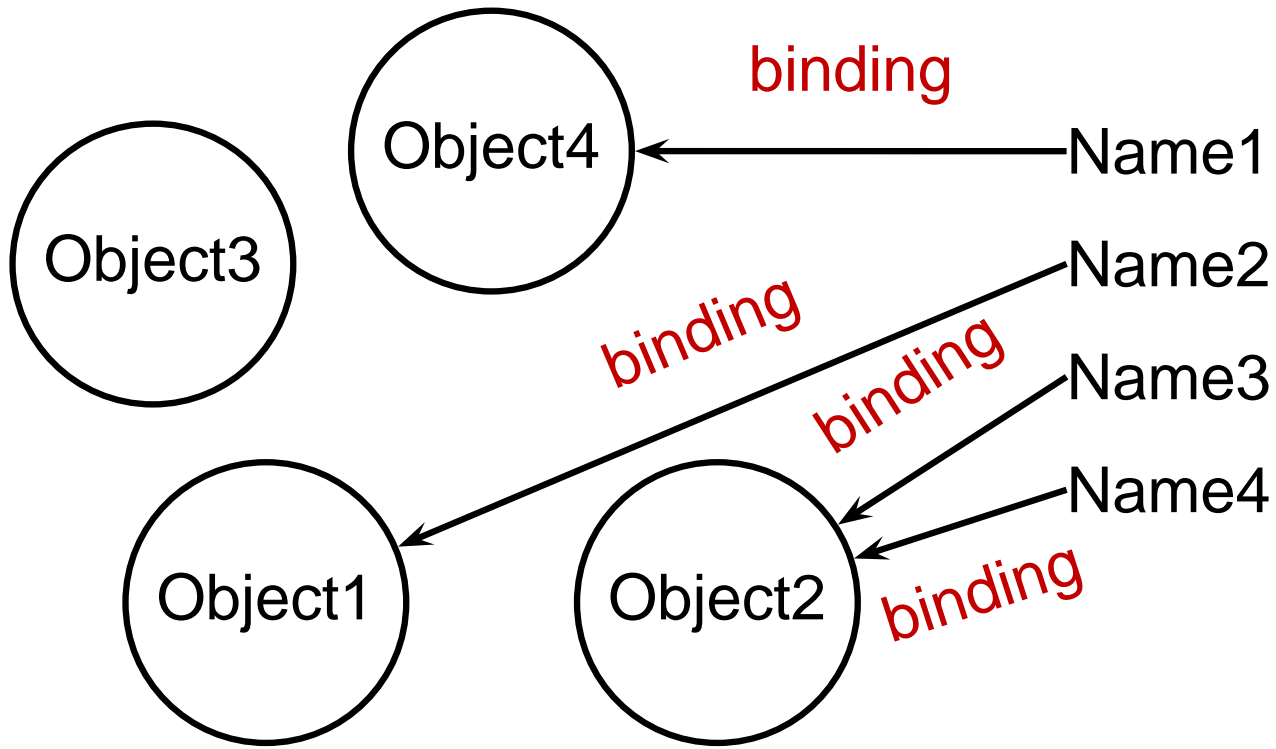
Name: way to refer to something else

variables, functions, namespaces, objects, types

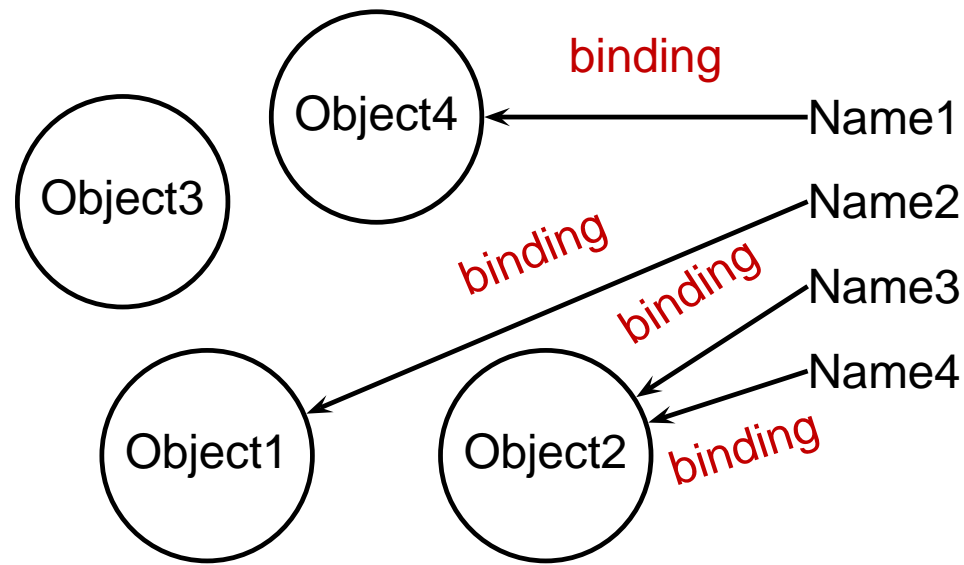
```
if ( a < 3 ) {  
    int bar = baz(a + 2);  
    int a = 10;  
}
```



# Names, Objects, and Bindings



# Names, Objects, and Bindings



When are objects created and destroyed?

When are names created and destroyed?

When are bindings created and destroyed?

# Object Lifetimes

When are objects created and destroyed?



# Object Lifetimes

The objects considered here are regions in memory.

Three principal storage allocation mechanisms:

1. Static

Objects created when program is compiled, persists throughout run

2. Stack

Objects created/destroyed in last-in, first-out order. Usually associated with function calls.

3. Heap

Objects created/deleted in any order, possibly with automatic garbage collection.

# Static Objects

```
class Example {  
    public static final int a = 3;  
  
    public void hello() {  
        System.out.println("Hello");  
    }  
}
```

Static class variable

Code for hello method

String constant "hello"

Information about Example class.

# Static Objects

## Advantages:

- Zero-cost memory management

- Often faster access (address a constant)

- No out-of-memory danger

## Disadvantages:

- Size and number must be known beforehand

- Wasteful if sharing is possible



# Stack-Allocated Objects



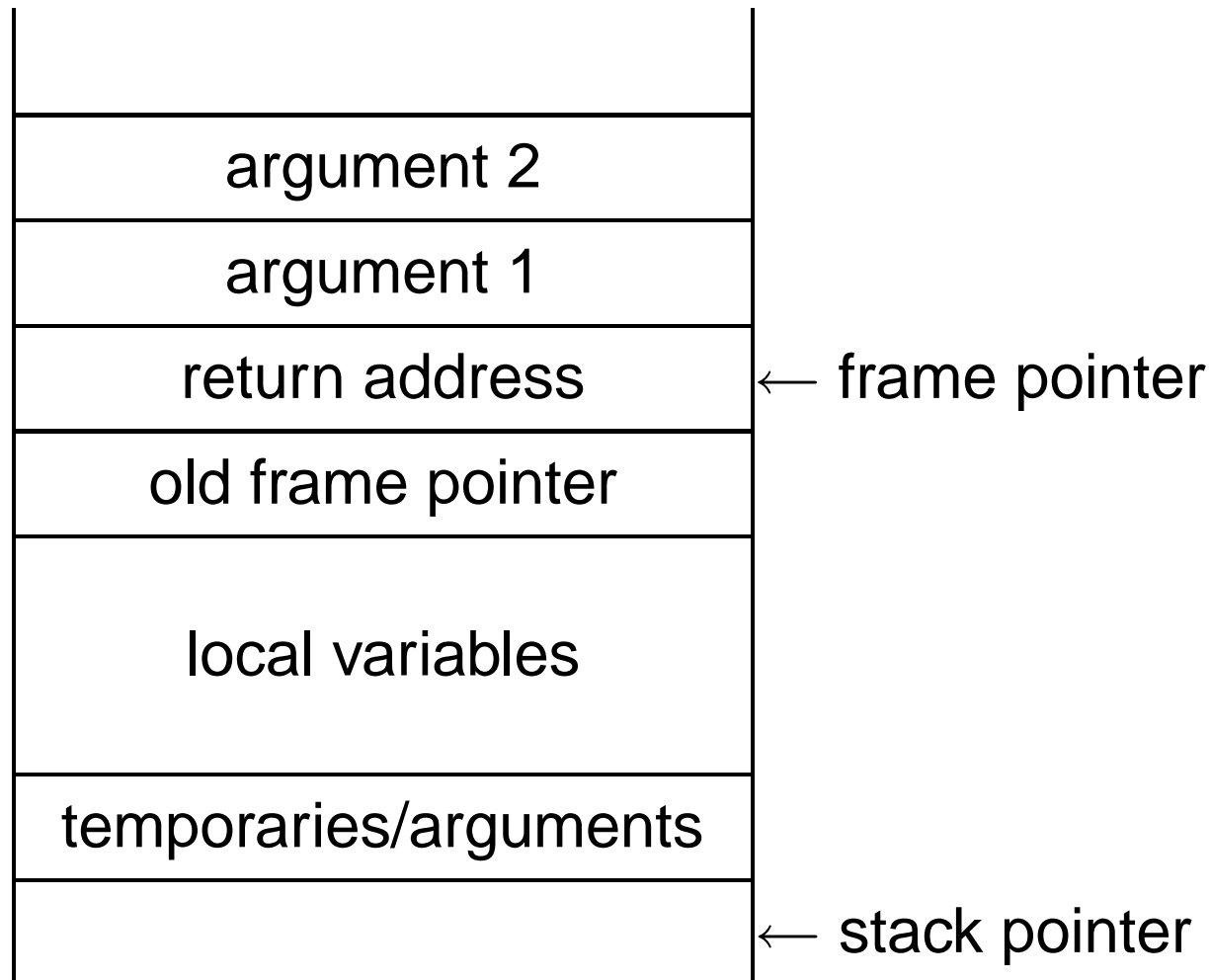
Natural for supporting recursion.

Idea: some objects persist from when a procedure is called to when it returns.

Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

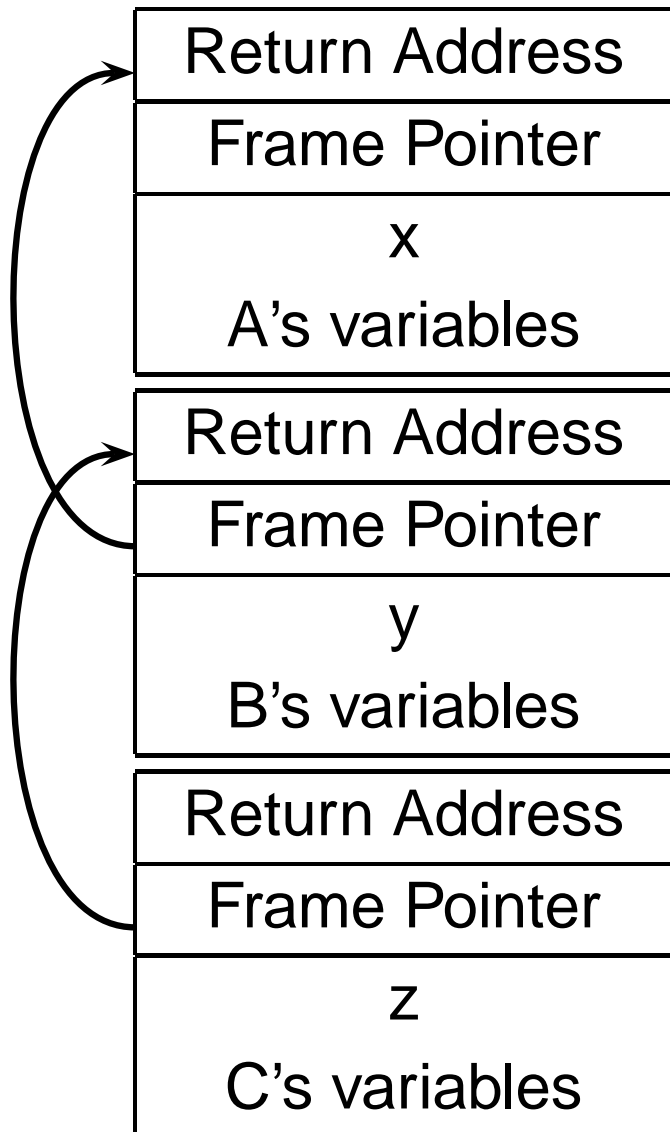
Each invocation of a procedure gets its own *frame* (*activation record*) where it stores its own local variables and bookkeeping information.

# Activation Records



↓ growth of stack

# Activation Records



```
int A() {  
    int x;  
    B();  
}
```

```
int B() {  
    int y;  
    C();  
}
```

```
int C() {  
    int z;  
}
```

# Stack-Based Languages

The FORTH language is stack-based. Very easy to implement cheaply on small processors.

The PostScript language is also stack-based.

Programs are written in Reverse Polish Notation:

```
2 3 * 4 5 * + . ( . is print top-of-stack )
```

**26 OK**

# FORTH

```
: CHANGE      0      ;
: QUARTERS 25 * + ;
: DIMES     10 * + ;
: NICKELS   5 * + ;
: PENNIES           + ;
: INTO 25 /MOD CR . ." QUARTERS "
          10 /MOD CR . ." DIMES "
           5 /MOD CR . ." NICKELS "
                   CR . ." PENNIES " ;
CHANGE 3 QUARTERS 6 DIMES 10 NICKELS
112 PENNIES INTO
11 QUARTERS
2 DIMES
0 NICKELS
2 PENNIES
```

# FORTH

Definitions are stored on a stack. FORGET discards the given definition and all that came after.

```
: FOO ." Stephen" ;
```

```
: BAR ." Nina" ;
```

```
: FOO ." Edwards" ;
```

```
FOO Edwards
```

```
BAR Nina
```

```
FORGET FOO ( Forgets most-recent FOO)
```

```
FOO Stephen
```

```
BAR Nina
```

```
FORGET FOO ( Forgets FOO and BAR)
```

```
FOO FOO ?
```

```
BAR BAR ?
```

# Heap-Allocated Storage

Static works when you know everything beforehand and always need it.

Stack enables, but also requires, recursive behavior.

A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)

# Dynamic Storage Allocation in C

```
struct point { int x, y; };
int play_with_points(int n)
{
    struct point *points;
    points = malloc(n * sizeof(struct point));
    int i;
    for ( i = 0 ; i < n ; i++ ) {
        points[i].x = random();
        points[i].y = random();
    }

    /* do something with the array */

    free(points);
}
```



# Dynamic Storage Allocation



↓ `free()`



↓ `malloc(`  `)`



# Dynamic Storage Allocation

Rules:

Each allocated block contiguous (no holes)

Blocks stay fixed once allocated

**malloc ( )**

Find an area large enough for requested block

Mark memory as allocated

**free ( )**

Mark the block as unallocated

# Simple Dynamic Storage Allocation

Maintaining information about free memory

Simplest: Linked list

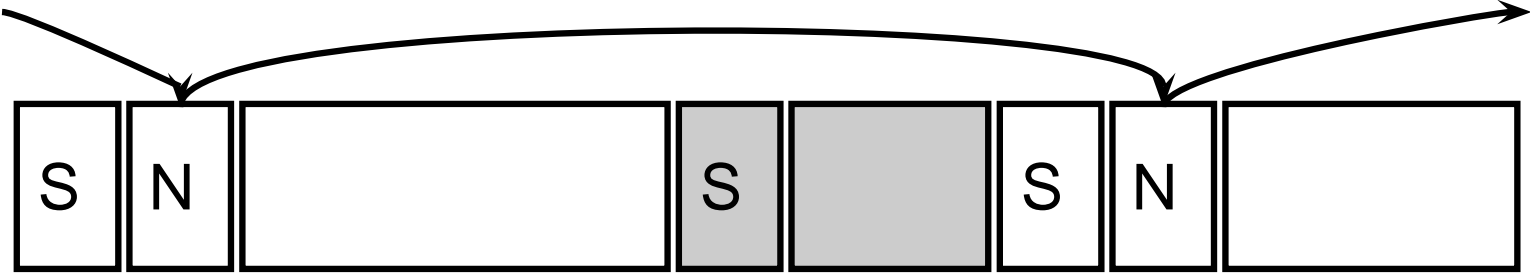
The algorithm for locating a suitable block


Simplest: First-fit

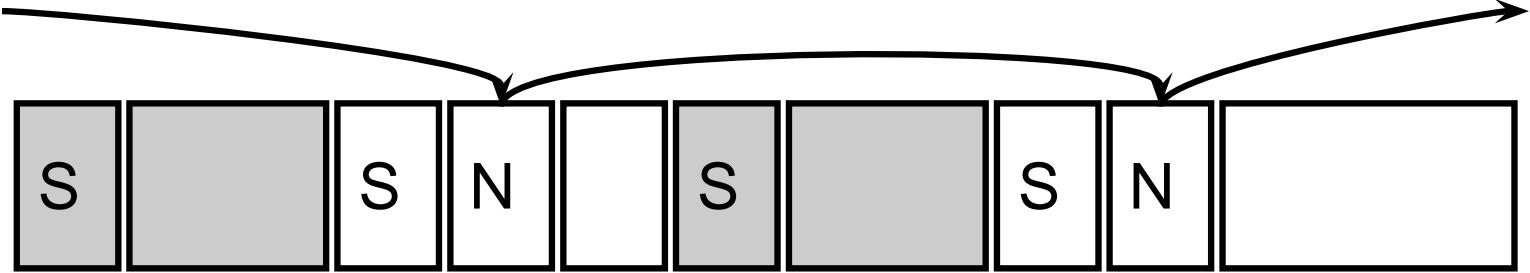
The algorithm for freeing an allocated block

Simplest: Coalesce adjacent free blocks

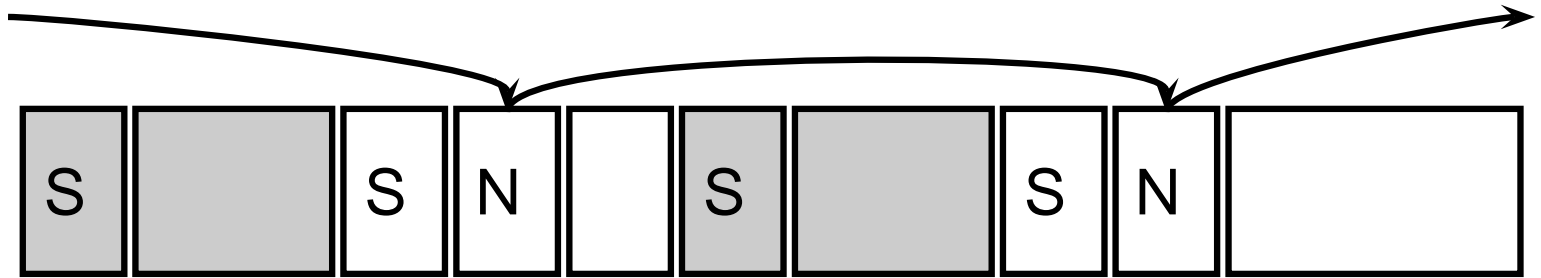
# Dynamic Storage Allocation



↓ malloc(  )



# Simple Dynamic Storage Allocation



↓ **free()**



# Dynamic Storage Allocation

Many, many other approaches.

Other “fit” algorithms

Segregation of objects by size

More clever data structures

# Heap Variants

Memory pools: Differently-managed heap areas

Stack-based pool: only free whole pool at once


- Nice for build-once data structures

Single-size-object pool:

- Fit, allocation, etc. much faster

- Good for object-oriented programs

# Fragmentation

`malloc(  )` seven times give



`free( )` four times gives



`malloc(  )` ?

Need more memory; can't use fragmented memory.



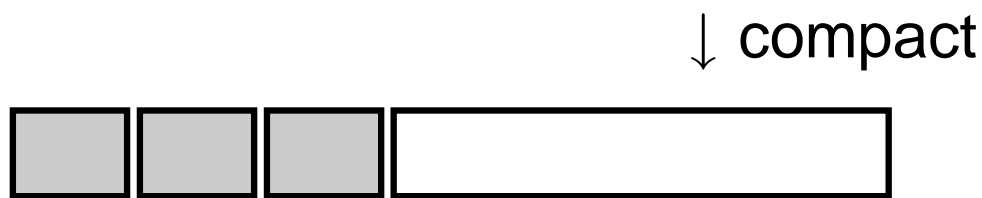
# Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”



ha hb hc  
↑ ↑ ↑  
\*a \*b \*c



ha hb hc  
↑ ↑ ↑  
\*a \*b \*c

The original  
Macintosh did  
this to save  
memory.

# Automatic Garbage Collection

Remove the need for explicit deallocation.

System periodically identifies reachable memory and frees unreachable memory.

Reference counting one approach.

Mark-and-sweep another: cures fragmentation.

Used in Java, functional languages, etc.



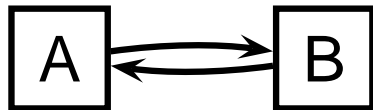
# Automatic Garbage Collection

Challenges:

How do you identify all reachable memory?

(Start from program variables, walk all data structures.)

Circular structures defy reference counting:



Neither is reachable, yet both have non-zero reference counts.

Garbage collectors often conservative: don't try to collect everything, just that which is definitely garbage.

# Scope

When are names created, visible, and destroyed?



# Scope

The scope of a name is the textual region in the program in which the binding is active.

Static scoping: active names only a function of program text.

Dynamic scoping: active names a function of run-time behavior.

# Scope: Why Bother?

Scope is not necessary. Languages such as assembly have exactly one scope: the whole program.

Reason: Information hiding and modularity.

Goal of any language is to make the programmer's job simpler.

One way: keep things isolated.



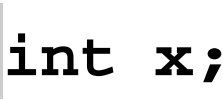



Make each thing only affect a limited area.

Make it hard to break something far away.



# Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

```
void foo()  
{  
    int x;   
  
    while ( a < 10 ) {  
        int x;   
  
          
  
  
  
    }  
}
```



# Static Scoping in Java

```
public void example() {  
    // x, y, z not visible  
  
    int x;  
    // x visible  
  
    for ( int y = 1 ; y < 10 ; y++ ) {  
        // x, y visible  
  
        int z;  
        // x, y, z visible  
    }  
  
    // x visible  
}
```

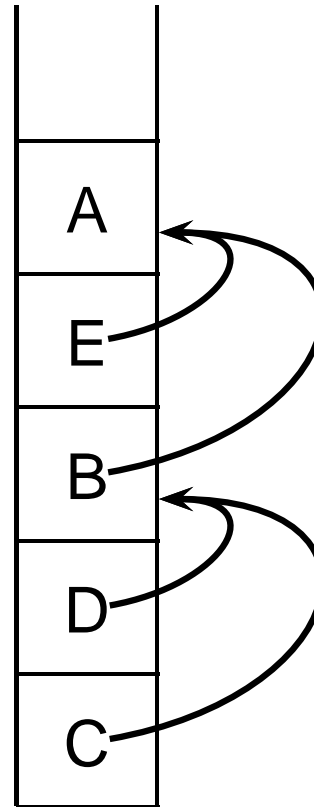
# Nested Subroutines in Pascal

```
procedure mergesort;  
var N : integer;  
  
    procedure split;  
    var I : integer;  
    begin .. end  
  
    procedure merge;  
    var J : integer;  
    begin .. end  
  
begin .. end
```



# Nested Subroutines in Pascal

```
procedure A;  
  procedure B;  
    procedure C;  
    begin .. end  
  
    procedure D;  
    begin C end  
  begin D end  
  
  procedure E;  
  begin B end  
begin E end
```



# Scope in Tiger

The `let` expression defines scopes:

```
let
```

```
  var x := 8
```

```
in
```

```
end
```

# Scope in Tiger

Scopes can nest to produce holes

```
let
```

```
  var x := 8
```

```
in
```

```
  let
```

```
    var x := 10
```

```
  in
```

```
    end
```

```
end
```

# Scope in Tiger

Mutual recursion possible because of odd scoping rules.

Scope of f1, f2, and f3:

```
let
  function f0() = (...)
  var x := 8
  function f1() = (...)
  function f2() = (...)
  function f3() = (...)
in
end
```

# Nested Functions in Tiger

Static (lexical) scope like Pascal

```
let
  var a := 3
  function f1() = ( a := a + 1 )
in
  let
    var a := 4
    function f2() = ( f1() )
  in
    f2()
  end
end
```

# Dynamic Scoping in TeX

```
% \x, \y undefined
{
  % \x, \y undefined
  \def \x 1
  % \x defined, \y undefined

  \ifnum \a < 5
    \def \y 2
  \fi

  % \x defined, \y may be undefined
}
% \x, \y undefined
```



# Static vs. Dynamic Scope

```
program example;  
var a : integer; (* Outer a *)  
  
    procedure seta;      begin a := 1 end  
  
    procedure locala;  
    var a : integer; (* Inner a *)  
    begin seta end  
  
begin  
    a := 2;  
    if (readln() = 'b') locala  
    else seta;  
    writeln(a)  
end
```

# Static vs. Dynamic Scope

Most languages now use static scoping.

Easier to understand, harder to break programs.

Advantage of dynamic scoping: ability to change environment.

A way to surreptitiously pass additional parameters.

# Application of Dynamic Scoping

```
program messages;  
var message : string;  
  
    procedure complain;  
    writeln(message);  
  
    procedure problem1;  
    var message : string;  
    message := "Out of memory"; complain  
  
    procedure problem2;  
    var message : string;  
    message := "Out of time"; complain
```

# Forward Declarations

Languages such as C, C++, and Pascal require *forward declarations* for mutually-recursive references.

```
int foo();  
int bar() { ... foo(); ... }  
int foo() { ... bar(); ... }
```

Partial side-effect of compiler implementations. Allows single-pass compilation.

# Open vs. Closed Scopes

An *open scope* begins life including the symbols in its outer scope.

Example: blocks in Java

```
{ int x;  
  for (;;) { /* x visible here */ }  
}
```

A *closed scope* begins life devoid of symbols.

Example: structures in C.

```
struct foo {  
  int x; float y;  
}
```

# Overloading

What if there is more than one object for a name?



# Overloading versus Aliases

Overloading: two objects, one name

Alias: one object, two names

In C++

```
int foo(int x) { ... }  
int foo(float x) { ... } // foo overloaded
```

```
void bar()  
{  
    int x, *y;  
    y = &x; // Two names for x: x and *y  
}
```

# Examples of Overloading

Most languages overload arithmetic operators:

```
1 + 2 // Integer operation
```

```
3.1415 + 3e-4 // Floating-point operation
```

Resolved by checking the *type* of the operands.

Context must provide enough hints to resolve the ambiguity.



# Function Name Overloading

C++ and Java allow functions/methods to be overloaded.

```
int foo();  
int foo(int a);    // OK: different # of args  
float foo();      // Error: only return type  
int foo(float a); // OK: different arg types
```

Useful when doing the same thing many different ways:

```
int add(int a, int b);  
float add(float a, float b);
```

```
void print(int a);  
void print(float a);  
void print(char *s);
```

# Function Overloading in C++

Complex rules because of *promotions*:

```
int i; long int l;  
l + i
```

Integer promoted to long integer to do addition.

```
3.14159 + 2
```

Integer is promoted to double; addition is done as double.

# Function Overloading in C++

1. Match trying trivial conversions

`int a[]` to `int *a`, `T` to `const T`, etc.

2. Match trying promotions

`bool` to `int`, `float` to `double`, etc.

3. Match using standard conversions

`int` to `double`, `double` to `int`

4. Match using user-defined conversions

```
operator int() const { return v; }
```

5. Match using the elipsis ...

Two matches at the same (lowest) level is ambiguous.

# Symbol Tables

How does a compiler implement scope rules?

# Symbol Tables

Basic mechanism for relating symbols to their definitions in a compiler.

Eventually need to know many things about a symbol:

- Whether it is defined in the current scope. “Undefined symbol”
- Whether its defined type matches its use.  
`1 + "hello"`
- Where its object is stored (statically allocated, on stack).

# Symbol Tables

Implemented as a collection of dictionaries in which each symbol is placed.

Two operations: insert adds a binding to a table and lookup locates the binding for a name.

Symbol tables are created and filled, but never destroyed.

# Symbol Tables

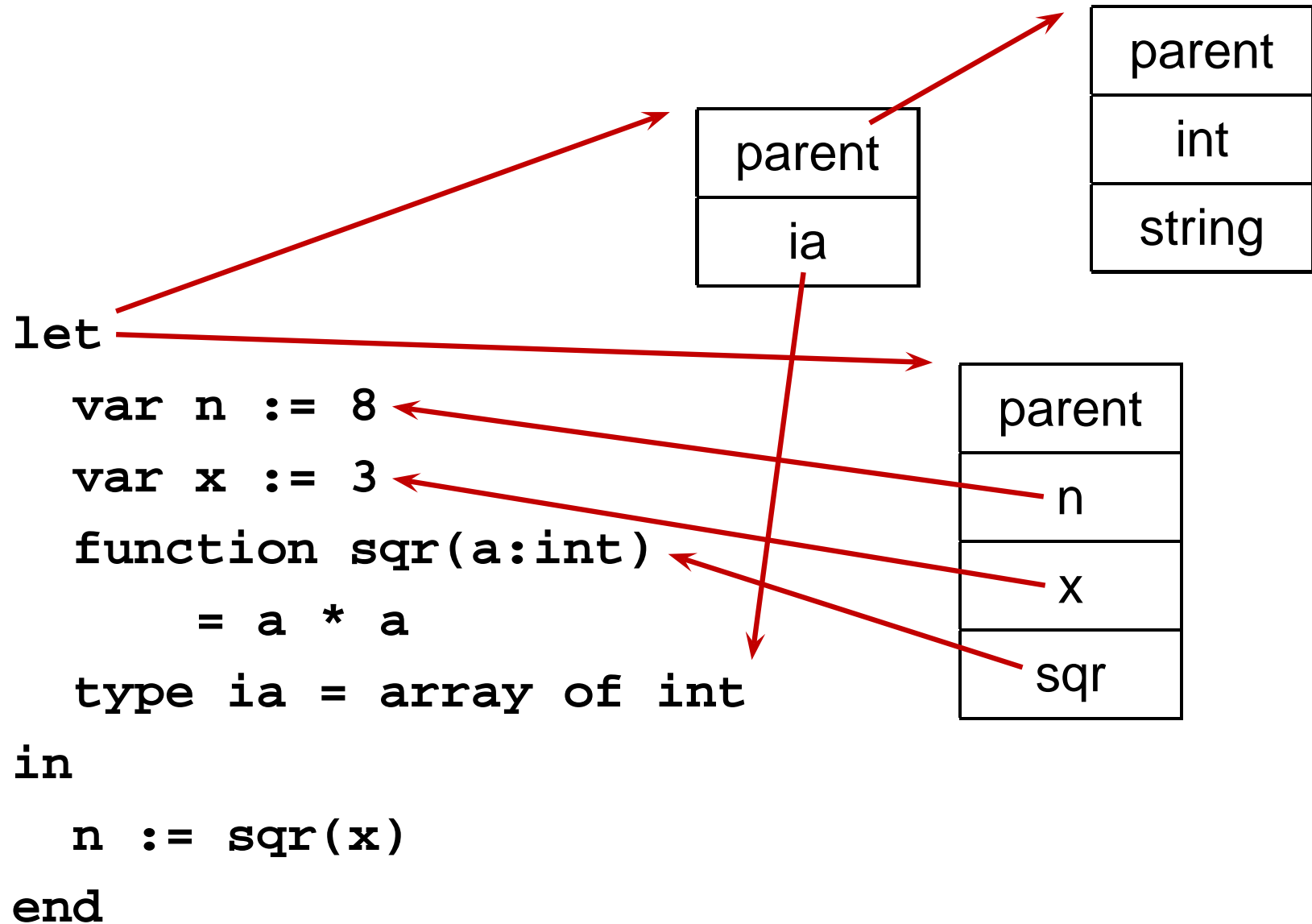
There are three namespaces in Tiger:

- functions and variables
- types
- record names

How many namespaces are there in Java?

How many namespaces are there in your language?

# Symbol Tables in Tiger





# Implementing Symbol Tables

Many different ways:

- linked-list
- hash table
- binary tree

Hash tables are faster, but linked lists are good enough for simple compilers.

# Symbol Table Lookup

Basic operation is to find the entry for a given symbol.

In many implementation, each symbol table is a scope.

Each symbol table has a pointer to its parent scope.

Lookup: if symbol in current table, return it, otherwise look in parent.

# Static Semantic Checking

Main application of symbol tables.

A taste of things to come:

Enter each declaration into its symbol table.

Check that each symbol used is actually defined in the symbol table.

Check its type. . . (next time)

# Binding Time

When are bindings created and destroyed?

# Binding Time

When a name is connected to an object.

## Bound when

language designed

language implemented

Program written

compiled

linked

loaded

run

## Examples

`if else`

data widths

`foo bar`

static addresses, code

relative addresses

shared objects

heap-allocated objects

# Binding Time and Efficiency

Earlier binding time  $\Rightarrow$  more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```
switch (statement) {  
  case add:  
    r = a + b;  
    break;  
  case sub:                                add %o1, %o2, %o3  
    r = a - b;  
    break;  
  /* ... */  
}
```

# Binding Time and Efficiency

Dynamic method dispatch in OO languages:

```
class Box : Shape {
    public void draw() { ... }
}
class Circle : Shape {
    public void draw() { ... }
}

Shape s;
s.draw();    /* Bound at run time */
```

# Binding Time and Efficiency

Interpreters better if language has the ability to create new programs on-the-fly.

Example: Ousterhout's Tcl language.

Scripting language originally interpreted, later byte-compiled.

Everything's a string.

```
set a 1
```

```
set b 2
```

```
puts "$a + $b = [expr $a + $b]"
```



# Binding Time and Efficiency

Tcl's `eval` runs its argument as a command.

Can be used to build new control structures.

```
proc ifforall {list pred ifstmt} {  
    foreach i $list {  
        if [expr $pred] { eval $ifstmt }  
    }  
}  
ifforall {0 1 2} {$i % 2 == 0} {  
    puts "$i even"  
}  
0 even  
2 even
```

# Binding Reference Environments

What happens when you take a snapshot of a subroutine?

# References to Subroutines

In many languages, you can create a reference to a subroutine and call it later. E.g., in C,

```
int foo(int x, int y) { /* ... */ }
```

```
void bar()  
{  
    int (*f)(int, int) = foo;  
  
    (*f)(2, 3); /* invoke foo */  
}
```

Where does its environment come from?

# References to Subroutines

C is simple: no function nesting; only environment is the omnipresent global one. But what if there were?

```
typedef int (*ifunc)();
ifunc foo() {
    int a = 1;
    int bar() { return a; } /* not C */
    return bar;
}
int main() {
    ifunc f = foo(); /* returns bar */
    return (*f)(); /* call bar. a? */
}
```

# Shallow vs. Deep binding

```
typedef int (*ifunc)();  
ifunc foo() {  
    int a = 1;  
    int bar() { return a; }  
    return bar;  
}  
int main() {  
    ifunc f = foo();  
    int a = 2;  
    return (*f)();  
}
```

		<b>static</b>	<b>dynamic</b>
	<b>shallow</b>	1	2
	<b>deep</b>	1	1

# Shallow vs. Deep binding

```
void a(int i, void (*p)()) {
    void b() { printf("%d", i); }
    if (i=1) a(2,b) else (*p)();
}
```

```
void q() {}
```

```
int main() {
    a(1,q);
}
static
shallow 2
deep 1
```

main()
a(1,q) i = 1, p = q b reference
a(2,b) i = 2, p = b
b

# Shallow vs. Deep Binding

Tiger does not have function types; problem avoided.

C does not have nested subroutines; problem avoided.

Modula-2 only allows outermost procedures to be passed as parameters (like C's solution).

Pascal has lexical scoping with nested subroutines, but does not allow function pointers to be returned.

Ada 83 prohibits passing subroutines as parameters.