Comparative Programming Languages

hussein suleman
uct csc304s 2003

Variables, Binding and Scope

Name Terminology

- A "name" or "identifier" is a used to identify an entity in a program.
  - Example: `someVariable, someType`
- A "keyword" has special meaning in the context of a specific language, but can be redefined.
  - Example: `INTEGER REAL` in FORTRAN declares "REAL" to be of type integer
  - Example: in English, *Buffalo buffalo buffalo*.
- A "reserved word" is like a keyword but cannot be redefined.
  - Example: `begin` in Pascal, `for` in Java

Variable

- Section of memory used to store data.
- 6 Attributes for each variable:
  - name = identifier used to refer to memory
  - address = actual physical/logical location in memory
  - type = type of data that may be stored
  - value = data that is stored
  - scope = whether/where a variable may be used
  - lifetime = whether/where a variable occupies actual storage (i.e., has an assigned address)

Aliasing

- Two or more variables may use the same address to store their values.
  - Example: variant records in Pascal, unions in C
  - Better efficiency in storing data.
  - Decrease readability as values of variables can be changed indirectly.
- Example:

```
union answer {
   int ans_integer;
   float ans_float;
};
```

Binding

- Association between language element and attribute.
- Binding times:
  - Language design time - when the language is designed, e.g., meaning of operators.
  - Compiler design time – when the compiler is designed, e.g., internal representations.
  - Compile time – when a module is compiled, e.g., type of variable.
  - Link time – when the modules are linked together, e.g., address of called subroutine.
  - Runtime – when the program is being run, e.g., value of variable.

Binding times

- Example (in Java):
  - `int x = x + 1;`

Binding	Time
Type of x	Compile time
Value of x	Runtime
Meaning of "+"	Language design time
Amount of storage required for x	Compiler design time
Meaning of "1"	Language design time
Storage for x	Runtime

Declarations and Definitions

- Some languages (e.g., C++) support the notion of splitting declarations from definitions.
- Declarations specify attributes such as type.
- Definitions specify attributes and bind storage.
- Example:
  - `forward` declaration in Pascal allows you to declare a variable/function before definition.

Static and Dynamic Types

- Static types cannot be changed.
  - Example (explicit declaration):
    - `int a;`
  - Example (implicit declaration in Perl):
    - `$newvariable = 12;`
- Dynamic types are bound at assignment.
  - Example (in APL):
    - `ABC ← 1  2  3`
    - `ABC ← 42`
- Static type can be checked at compile-time – dynamic types must be checked at runtime!

Strong typing

- Strong typing implies type errors are always detected – opposite of weak typing.
- Example:
  - C/C++ is not strongly typed.
  - Java is strongly typed but allows explicit casts.
- Coercion is when the type of an element is automatically converted when needed.
  - Example (in Java):
    - `int a =2; double x = 1.0 / a;`
  - Coerced values lead to less reliable error detection!

Type compatibility

- Types must match when parameters are passed, assignments are made, etc.
- Name compatibility: type names must match.
  - Example (C++):

```
typedef struct { int a } typea;
typedef struct { int b } typeb;
// typea is not compatible with typeb !
```

- Structure compatibility: structure of types must match.
  - Example (Pascal):

```
type typea = real; typeb=real;
{ typea is compatible with typeb }
```

- Most languages use mixtures of name and structure compatibility.

Lifetime

- A variable/parameter has lifetime when storage is allocated for it.
- Static variables have lifetime for the whole execution of the program.
- Dynamic variables have lifetime when they are elaborated (allocated/bound).
  - Examples:
    - Local variables stored on stack.
      - e.g., recursive function parameters
    - Memory blocks explicitly allocated on heap.
      - e.g., C's malloc memory allocation
    - Memory blocks implictiy allocated on heap.
      - e.g., ALGOL's flex arrays

Scope

- An identifier has scope when it is visible and can be referenced.
- An out-of-scope identifier cannot be referenced.
- Identifiers in open scopes may override older/outer scopes temporarily.
- 2 Types of scope:
  - Static scope is when visibility is due to the lexical nesting of subprograms/blocks.
  - Dynamic scope is when visibility is due to the call sequence of subprograms.

Changing Scope

- Identifiers come into scope at the beginning of a subprogram/block and go out of scope at the end.

- Example (in C++):

```
void testfunc ()
{
   int a; // a enters scope;
   for ( int b=1; b<10; b++ ) // b in scope for for
   {
      int c; // c enters scope
      …
   } // b,c leave scope
   …
} // a leaves scope
```

Static Scope

- Consider the Pascal program (which uses static scoping):

```
program test;
var a : integer;

   procedure proc1;
   var b : integer;
   begin
   end;

   procedure proc2;
   var a, c : integer;
   begin
      proc1;
   end;

begin
   proc2;
end.
```

in scope: b (from proc1), a (from test)

in scope: a, c (from proc2)

in scope: a (from test)

Dynamic Scope

- Consider the Pascal-like code (assume dynamic scoping):

```
program test;
var a : integer;

   procedure proc1;
   var b : integer;
   begin
   end;

   procedure proc2;
   var a, c : integer;
   begin
      proc1;
   end;

begin
   proc2;
end.
```

in scope: b (from proc1) a, c (from proc2)

in scope: a, c (from proc2)

in scope: a (from test)

Static vs. Dynamic Scope

- Dynamic scope makes it easier to access variables with lifetime, but it is difficult to understand the semantics of code outside the context of execution.
- Static scope is more restrictive – therefore easier to read – but may force the use of more subprogram parameters or global identifiers to enable visibility when required.

Lifetime revisited

- Consider the Pascal program (which uses static scoping):

```
program test;
var a : integer;

   procedure proc1;
   var b : integer;
   begin
   end;

   procedure proc2;
   var a, c : integer;
   begin
      proc1;
   end;

begin
   proc2;
end.
```

lifetime: b (from proc1), a, c (from proc2), a (from test)

lifetime: a, c (from proc2), a (from test)

lifetime: a (from test)

Lifetime vs. Scope

- Lifetime is influenced by call sequence.
- Scope is influenced by lexical structure (static) or call sequence (dynamic).
- Identifiers can have
  - lifetime and no scope
  - lifetime and scope
  - no lifetime and no scope
- Identifiers cannot have scope without lifetime!

Types and Pointers

Common Data Types

- Integers, Floating point numbers, Characters, Booleans
- Strings
- Arrays
- Enumerations and Subranges
- Hashes
- Lists
- Records and Unions
- Pointers

Strings

- Null-terminated array of characters in C

'T'	'u'	'e'	's'	'd'	'a'	'y'	0

- Length-prefixed array of characters in Pascal

7	'T'	'u'	'e'	's'	'd'	'a'	'y'

- Object in Java
- Can have fixed or dynamic maximum length
  - Notorious buffer overflow remote exploit!
- String operations: substring, length, concat, etc.

Arrays

- Static arrays have fixed size, global scope and lifetime.
- Fixed stack-dynamic arrays are allocated on the stack, e.g., local arrays in subprograms.
- Stack-dynamic arrays have bounds that are not known until use, e.g., passing an array as a parameter.
- Heap-dynamic arrays have flexible bounds.

Array Subscripts

- Assume the 1-dimensional array:
  - `int list[1..10];`
  - address(list[x]) = address (list[1]) + (x-1)* sizeof (int);
- Assume the 3-dimensional array:
  - `someType box[f0..f][g0..g][h0..h];`
  - address(box[i][j][k]) = address(box[f0][g0][h0]) + (((i-f0)*(g-g0+1)*(h-h0+1))+((j-g0)*(h-h0+1))+(k-h0))*n
    - where n = size of someType
- Row-major order = rows stored first in memory.
- Column-major order = columns stored first.

Array Manipulations

- FORTRAN 90 supports obtaining slices of arrays as arrays of lesser dimensionality.
  - Example:
    - `CUBE(1:3, 2)` of `CUBE(1:3, 1:3)` results in a one-dimensional array with 3 elements.
- APL supports vector/matrix operations such as transpose, invert and inner product.

Enumerations and Subranges

- Enumerations are a set of named values used to aid readability.
  - Example (Pascal):
    - `type days = (mon, tue, wed, thu, fri, sat, sun);`
- Subranges restrict the range of possible values of a scalar/ordinal type for better type checking.
  - Example (Ada):
    - `subtype WEEKEND is DAYS range sat..sun;`

Hashes and Lists

- Perl supports variable-sized associative arrays (hashes) for name/value pairs.
  - Example:

```
%days = (“mon”=>3, “tue”=>5, “wed”=>7);
$value = $days{“tues”};
```

- Most functional languages support lists of items as a fundamental data type.
  - Example (Mathematica):

```
[“mon”, “tue”, “wed”, “thu”, “fri”]
```

Records

- Collection of related heterogenous data elements.
- Stored and manipulated as an atomic “object” in some languages.
- Variant records have overlapping fields to conserve memory at the expense of reliability.
  - Example (ALGOL68):

```
union (real, int) answer;
case answer in
   (real r): someReal = r;
   (int I): someint = I;
esac
```

Sets

- Pascal has an analogue to mathematical sets, with operations to determine unions, intersections, equality and set membership.
- Example:

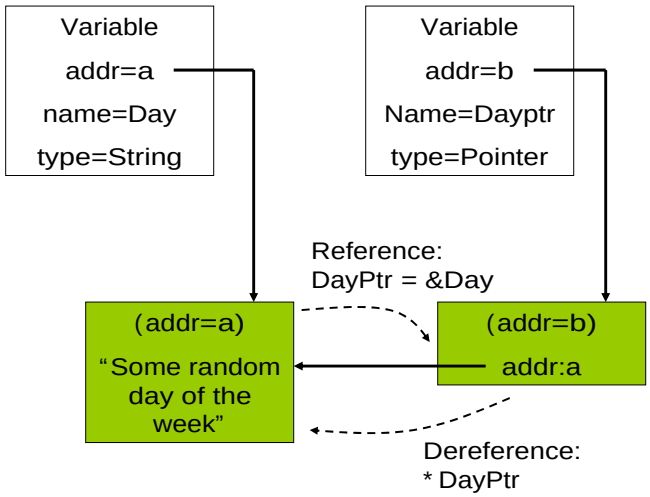
```
type dayset = set of (mon, tue, wed, thu, fri);
var day : dayset;
```

- Sets are usually implemented as fixed-length bit patterns therefore very efficient but restricted in set size.

Pointers

- Pointers hold memory addresses - effectively pointing to the contents of other variables (named or not).
- Languages that use pointers provide operators to:-
  - Get the address of a memory location.
  - Get the contents pointed to by a pointer.
  - Shift pointers.
  - Allocate memory and deallocate memory on the heap.

Referencing and Derefencing



Pointer Predicaments

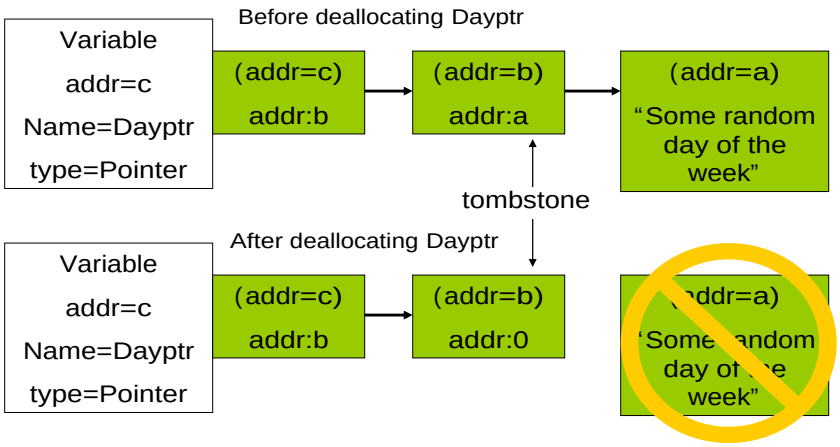
- Dangling pointers occur when a pointer points to a memory location that no longer has lifetime.
  - Example:

```
int *j = new int;
int *p = j;
delete j;
int k = *p;
```

- Memory leaks occur when explicitly allocated memory is not deallocated after use.

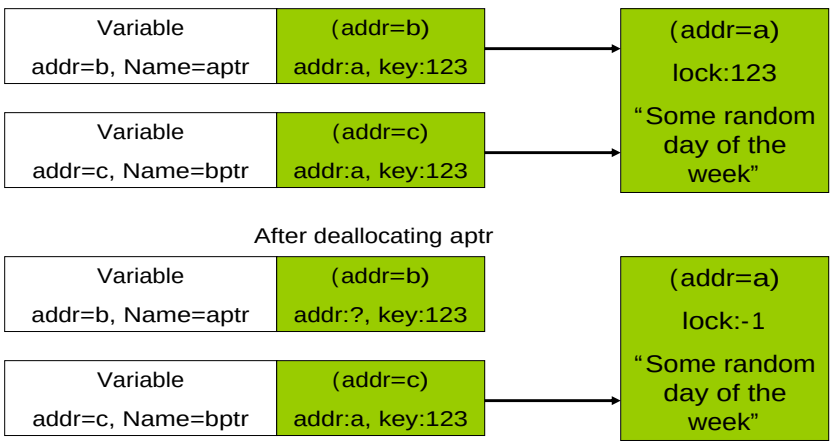
Tombstones

- Pointers point to an intermediary memory block, which is never deallocated.



Locks and Keys

- Each block of memory has a lock with corresponding keys for active pointers.



Reference Counting

- Without explicit deallocation of memory, reference counts can be attached to each memory chunk to count the number of pointers pointing to the memory.
  - When the reference count reaches zero, the memory can be disposed of and reused.
- How do we reference-count circular linked lists?

Mark-and-sweep Garbage Collection

- Traverse every pointer and mark the memory it points to as being used - then dispose of all allocated memory which isnt marked.
  - All pointers must be followed, even those within allocated blocks of memory.
- How efficient is this?

Assignments and Expressions

Precedence and Associativity

- Precendence refers to the relative order in which operators are evaluated within a larger expression.
  - E.g., * usually has precedence over +
- Associativity refers to the order in which operators of the same type are evaluated.
  - E.g., Assuming left associativity, `1-2-3=-4`
  - E.g., Assuming right associativity, `1-2-3=2`
- Parentheses can force a different order.
  - E.g., `(1-(2-3))` is always 2

Ternary operators

- Ternary operators provide a result based on 3 parameters.
- Popular example is ?:
  - Example: `value = (xy == 5) ? 1 : 2;`
  - Equivalent to:
    - `if (xy == 5) { value = 1; }`
    - `else { value = 2; }`
- In functional languages, every expression is a function with flexible numbers of parameters.
  - Example (Mathematica):
    - `value = If[xy==5,1,2]`

Side-effects

- An expression has a side-effect when the act of evaluating the expression has a persistent effect on other parts of the program.
  - E.g., a global variable is incremented by:
    - `(x + 1) + y++`
- Side-effects decrease the readability/reliability.
- Most functional languages completely disallow side-effects.

Overloading

- C++ allows classes to redefine the semantics of built-in operators when applied to instance variables.
- Can be useful when applied to obvious scenarios such as the definition of Vector and Matrix classes.
  - Otherwise detrimental to readability.

Short-circuit Evaluation

- Short circuit evaluation is when all parts of a boolean expression are not evaluated because such evaluations are not necessary to determine the result.
  - E.g., "`(true) and (false) and (xyz)`" will never evaluate xyz since the expression is already false.
- Some languages provide both operators for options, while others provide one or leave it as a compiler-level option.

Chaining Assignments

- In Java, assignment is regarded as an expression whose value is the same as the LHS.
  - Example:
    - `a = b = c = d = 10;`
- Assignment is right-associative.
- This is another type of side-effect!

Control Structures

Branching Selection

- "if" statements exist in most languages to select among alternative control flow paths.
  - Example (Pascal):

```
if (num > 1)
   then
      val := 7;
   else
      val := 5;
```

- In general, a boolean expression is used to determine which branch to take.

Dangling Else

- A dangling else is when the compiler cannot determine which if to match an else to when if statements are nested.
  - Example (in Java):

```
if (a == b)
   if (b == c)
      d=e;
   else
      f=g;
```

Which "if" does this "else" refer to?

- Solutions:
  - Discipline of programmers
  - Language restrictions

Dangling Else Prevention

- Ada requires “if” statements to be terminated by an “end if”.
  - Example:

```
if (a = b) then
   if (b = c) then d=e;
              else f=g;
   end if;
end if;
```

- Perl requires all if/then statements to be blocks.
  - Example:

```
if (a == b) {
   if (b = c) { d=e; } else { f=g; }
}
```

Multiway Selection

- Select among multiple control paths based on a single expression.
  - Example (Pascal):

```
case numero of
   1, 3, 5 : begin
                odd := true;
                even := false;
             end;
   2, 4, 6 : begin
                even := true;
                odd := false;
             end;
   else odd := true; even := true;
end
```

Multiway: C vs. Pascal

- C does not use independent blocks and relies on the programmer using “break” statements when necessary.
- The independence of blocks within the control structure is not guaranteed.
- C has greater flexibility but readability is decreased.

Counter controlled loops

- “for” loops are based on a variable that controls the number of iterations and provides a parameter in each iteration.
  - Example (ALGOL 60):

```
for index := 1, 4, 13, 41
             step 2 until 47,
             3 * index while index < 1000,
             34, 2, -24
    sum := sum + index
```

- In general, there are 3 steps:
  - Initialisation of variables
  - Test before (or after) each iteration
  - Update control variable for next iteration

Logical Loops

- Pretest loops test for exit before the first iteration.
  - Example (C):

```
while (<expression>) {
   ...
}
```

- Posttest loops test for exit after the statement (therefore at least one iteration):
  - Example (C):

```
do {
   ...
} while (<expression>)
```

Other Loops

- Ada allows infinite loops by not specifying a test as part of the construct. Exitting from the loop must then be explicit.
- Other languages can simulate infinite loops by using a constant test.
  - Example:

```
while (true) { ... }
```

- Iterative loops iterate over items of a list.
  - Example (Perl):

```
foreach my $node (@nodelist)
{ print $node->value; }
```

Goto

- Branches unconditionally to a different location in the program.
- Locations can be labelled by names (Pascal) or line numbers (FORTRAN).
- Branching can be restricted to a specific scope (Pascal) or can be global (BASIC).
- Goto is a controversial structure because it reduces readability - hence many modern languages do not include it.

Guarded Commands

- Dijkstra proposed guarded commands, which select statements non-deterministically from list of those with guards that evaluate to true.
  - Example:

```
if (a<b)  -> a = -1;
[] (a==b) -> a = 0;
[] (a>b)  -> a = 1;
```

- Guarded commands can be proven correct more easily than constructs such as "goto".

Subprograms

Types of Subprograms

- Procedures
  - Collection of statements that define a new “statement” for use by the programmer.
- Function
  - Collection of statements to compute a result.
- Is there really a difference?
- Rule
  - Specification of an assertion and the conditions under which it can be made.
- Template
  - Replacement text and the conditions under which replacement can be effected.

Structure of Subprograms

- Subprogram Call
  - e.g., `doCalc ();`
- Declaration: header
  - e.g., `int doCalc ();`
- Definition: header+body
  - e.g., `int doCalc () { return 1; }`
- Header vs. Body
  - Header is name of subprogram, parameters and return values.
  - Body is block of statements.

Parameters

- Formal Parameters
  - names used in subprogram definition.
- Actual Parameters
  - names/values used in subprogram invocation (call).
- Keyword Parameters
  - specified as name/value pairs.
- Positional Parameters
  - names are (usually) specified in header, corresponding values are bound from call by position.
- Example procedure call (Perl):
  - `callProg (1, 2, { start=>1, end=>2 });`
  - formal/actual? keyword/positional?

Parameter Passing 1/2

- Pass by value
  - Value is passed/copied to subprogram from caller.
- Pass by result
  - Value is passed/copied from subprogram back to caller.
  - Function return values are pass-by-result.
- Pass by value-result
  - Value is first passed/copied to subprogram from caller upon invocation, then passed/copied back to caller after invocation.

Parameter Passing 2/2

- Pass by reference
  - Variable is aliased so that both formal and actual parameter can access/change the same memory location – like using a pointer but safer!
- Pass by name (ALGOL, SIMULA)
  - Equivalent to actual parameter being "textually inserted" wherever it occurs in the subprogram.
  - Implemented using a "thunk" – parameterless subprogram that is evaluated in caller's environment each time the pass-by-name formal parameter is encountered.

Parameter Example 1/2

```
int b = 0;
subprogram FunkyFunction ( int a )
{
   b = a + 1;
   a = b + 1;
}
FunkyFunction (b);
```

- Pass-by-value: b=1
- Pass-by-result: Error - use before assignment
- Pass-by-value-result: b=2
- Pass-by-reference: b=2
- Pass-by-name: b=2

Parameter Example 2/2

```
int a = 0;
int b = 1;
subprogram NameProc ( int c )
{
   b = 4;
   a = c + 1;
}
NameProc (a+b);
```

- Pass-by-value: a=2, b=4
- Pass-by-result: Error – not lvalue
- Pass-by-value-result: Error – not lvalue
- Pass-by-reference: Error – not lvalue
- Pass-by-name: a=5, b=4

Subprogram Parameters

- Pass a subprogram as a parameter to another.
  - e.g., a string sorting routine needs to know how to compare strings and this may differ across data types and applications.
- Example:

```
procedure sort3 ( a, b, c : string;
  function compare ( a, b : string ) : int )
{
   if compare (a,b) swap (a,b);
   if compare (b,c) swap (b,c);
   if compare (a,b) swap (a,b);
}
```

- Which referencing environment to use ?

Subprogram Side-Effects

- When a subprogram has a persistent effect or an effect on the non-local environment.
  - Examples:
    - static variables in C++
    - assignment to a global variable within a procedure
- Pure functional languages have no assignment, therefore cannot have side-effects!

Generic Subprograms

- Abstract data used in subprogram results in abstract subprogram that must be instantiated with actual data type before use.
  - For example, C++ has templates, which are automatically instantiated upon use.
- How do statically-bound templates compare to polymorphism with dynamic binding?

Subprogram Invocation Mechanics

- Save status of caller.
- Process parameters.
- Save return address.
- Jump to called subprogram.
- Process value-result/result parameters and function return value(s).
- Restore status of caller.
- Jump back to caller's saved position.

Activation Records

- An activation record is the layout of data needed to support a call to a subprogram.
- For languages that do not allow recursion, each subprogram has a single fixed activation record instance stored in memory (and no links).

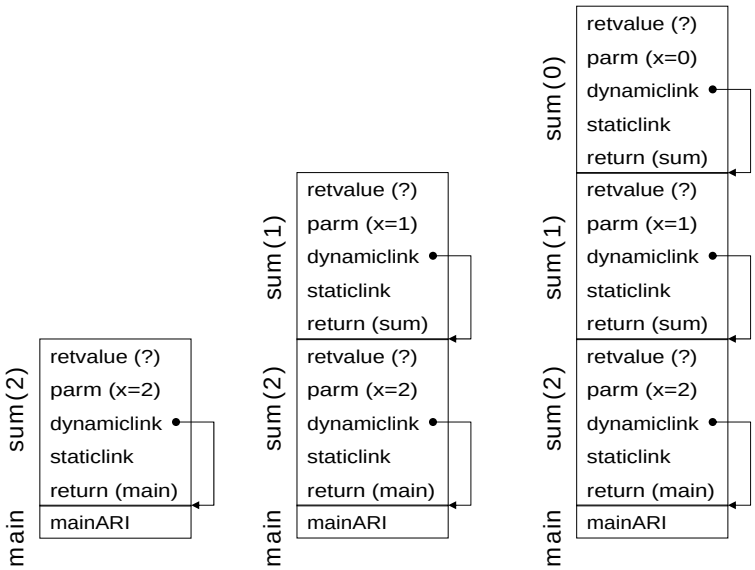
Function return value
Local variables
Parameters
Dynamic link
Static link
Return address

Stack-based Recursion 1/2

- When recursion is implemented using a stack, activation records are pushed onto the stack at invocation and popped upon return.
- Example:

```
int sum ( int x )
{
   if (x==0) return 0;
   else return (x + sum (x-1));
}
void main ()
{ sum (2); }
```

Recursion Activation Records



Non-local References

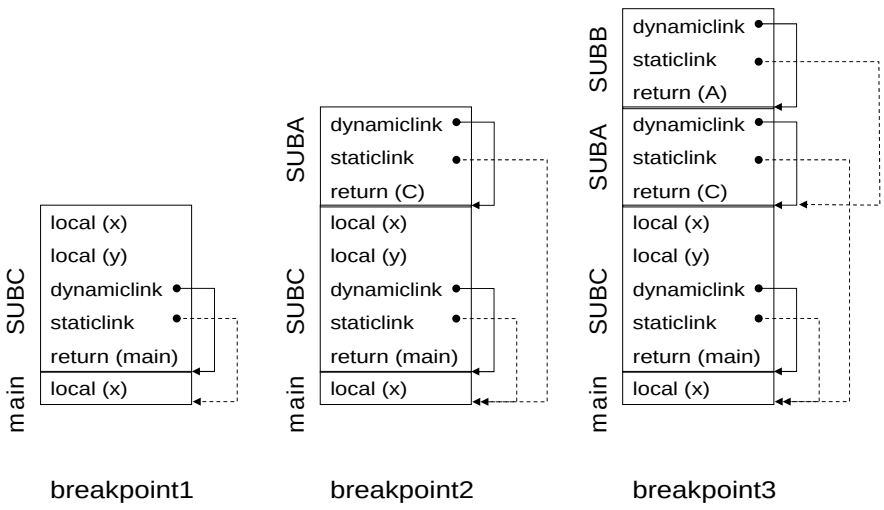
- To access non-local names in statically-scoped languages, a program must keep track of the current referencing environment.
- Static chains
  - Link a subprogram's activation record to its static parent.
- Displays
  - Keep a list of active activation records.

Non-local Reference Example

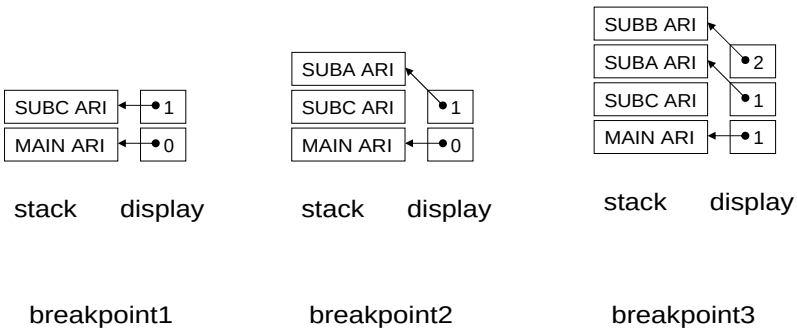
- Example:

```
main {
   int x;
   sub SUBA {
      sub SUBB {
         x = 1;          <-- breakpoint3
      }
      SUBB;              <-- breakpoint2
   }
   sub SUBC {
      int x;
      int y;
      SUBA;              <-- breakpoint1
   }
   SUBC;                 <-- breakpoint0
}
```

Static Chains



Displays



Static Chains vs. Displays

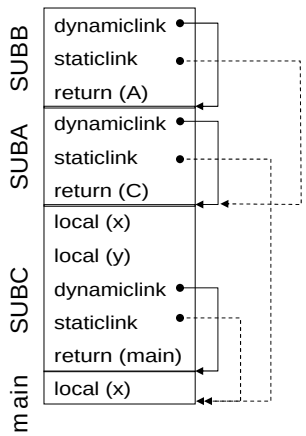
- Static chains require more indirect addressing – displays require a fixed amount of work.
- Displays require pointer maintenance on return – static chains do not.
- Displays require "backing up" of display pointer – static chains require static links in each activation record.

Dynamic Scoping

- Dynamically scoped languages can be implemented using:
- Deep Access
  - Follow the dynamic chains to find most recent non-local name definition.
- Shallow Access
  - Maintain a separate stack for each name.

Deep Access

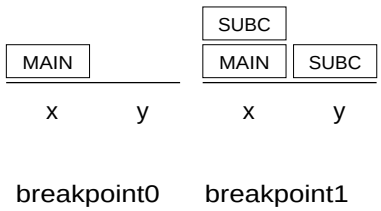
- At breakpoint3, by following dynamic links from SUBB, the closest definition of x is in SUBC.
- (Remember that for static scoping, by following static links, the closest definition is in main.)



breakpoint3

Shallow Access

- Constant-time access to all non-local names.
- Requires more maintenance in terms of pushing and popping the individual stacks.



breakpoint0 breakpoint1