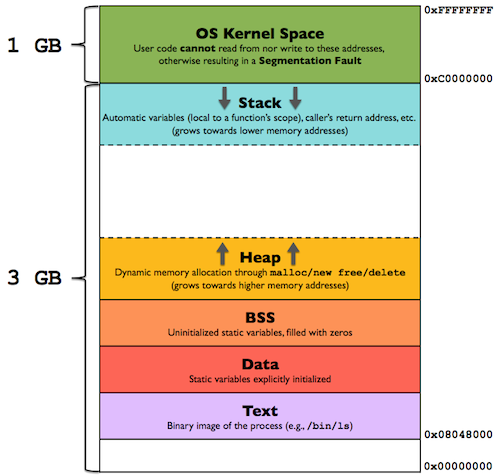
**Chapter 8 Subroutines and Control Abstraction**

8.1 Review and Recap

* Subroutine
  + a named routine that performs a well-defined operation (hopefully)
  + a principle mechanism for control abstraction, it performs an op on behalf of a *caller* , who stop and waits for the subroutine to finish
* Allocation strategies for various program components

http://www.geeksforgeeks.org/memory-layout-of-c-program/



1. Static allocation (location does not change during execution) – the following are statically allocated:

* Program Code (ex., binary image of process)
* Global (variables, constants, etc)
* static variables (not global but history sensitive)
* Explicit constants (not declared)

ex., 3.14 as in the statement

area = 3.14\*radius\*radius;

1. Stack allocation – used for subroutines

* Parameters (or argument)
* local variables
* bookkeeping information: return address, saved register value, …

1. Heap

* dynamic variables
* Why store local variables on a stack, instead of a permanent location as did in Fortran I-III?

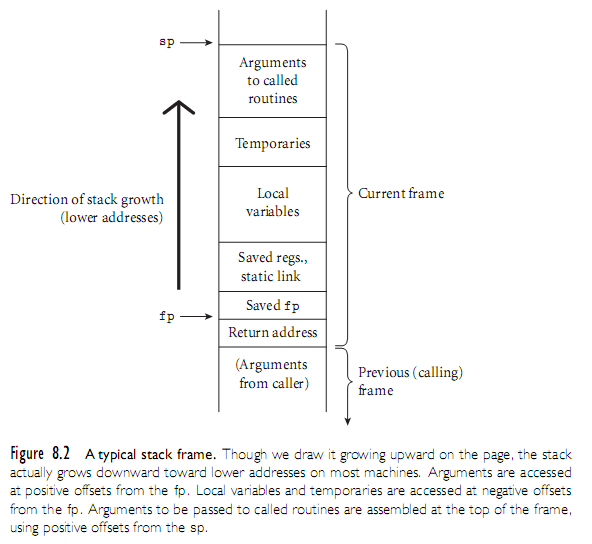
To allow recursive sub calls where there are any number of instances of a sub at any time.

* Contents of *a stack frame* (also known as A*ctivation Record*) determine the *Local Referencing Environment* (LRE) of the statements being executed

1. Bookkeeping – stuff saved for caller

* return address
* saved registers
* line number
* saved display entries
* saved static link

1. arguments and returns
2. local variables
3. temporaries



8.2 Calling Sequences

* Maintenance of stack is responsibility of calling sequence and subroutine prologue and epilog (see also Chapter 3 notes)
  + space is saved by putting as much in the prolog and epilog as possible
  + time may be saved by putting stuff in the caller instead, where more information may be known
* Saving and Restoring registers – somewhat tricky
  + The issue: different subroutines may share the same registers
  + Method 1 :
    - caller can save contents of all registers he uses
    - callee restore everything for caller
    - safe, but less inefficient
  + Method 2
    - only those registers that are used by both parties (i.e., caller and callee) need to be saved
    - ideal but difficult to know which ones fit the bill

* + Common strategy is to divide registers into *caller-saves* and *callee-saves* sets
    - caller uses the "callee-saves" registers first
    - use "caller-saves" registers if necessary
* Local variables and arguments are assigned fixed *OFFSETS* either from the stack pointer sp (less often) or frame pointer fp (preferred method) at compile time
  + sp – a stack pointer which points to the first unused location on the stack
  + fp – a frame pointer that points to a location in current frame

**A Typical Calling Sequences**

* the Caller
  1. Saves any caller-saves registers whose values will be needed after the call
  2. Compute the values of arguments and move them into stack and/or registers
  3. Compute the static link (if the language supports nested subroutine, see page 9 of chapter 3 notes), and pass it as a hidden argument
  4. Jump to the subroutine and pass the return address (on stack or register)
* In prologue , the Callee
  1. Allocate a frame by subtracting a frame size from current sp
  2. saves callee-saves registers used anywhere inside callee
  3. assign appropriate value for fp (save old fp first)

\*\*\*\*\*\*\* Subroutine execution \*\*\*\*\*\*\*\*\*\*

* In epilog (at the termination of a sub), the Callee
  1. puts return value into registers
  2. restore fp and sp
  3. restores callee-saved registers
  4. jump back to return address
* After call, Caller
  1. moves return value from register to wherever it's needed (if appropriate)
  2. restores caller-saves registers lazily over time, as their values are needed

Note that

* All arguments is allocated space in the stack, whether passed in registers or not
* This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope
* Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  + particularly *LEAF routines* (those that don't call other routines)
    - leaving things out saves time
    - simple leaf routines may not use the stack – all registers 🡪 fast execution

8.2.1 *Displays and static chain*

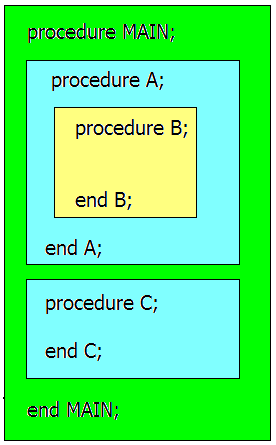
Implementation of static scope (used in many languages that supports nested subroutines such as PASCAL, ADA, or even C++ with its {}).

Finding the correct Local Referencing Environment using static scope – two techniques

1. Static Chain
2. A ***static chain***is a chain of static links that connects child frames to their ancestor frames
3. The static link in a frame for subprogram A points to the frame of A's static parent
4. The static chain from a frame connects it to all of its static ancestors
5. To find the declaration/definition for a reference to a nonlocal variable
   * you could follow the static chain until the frame that has the variable is found, searching each frame as it is found

**Static depth**

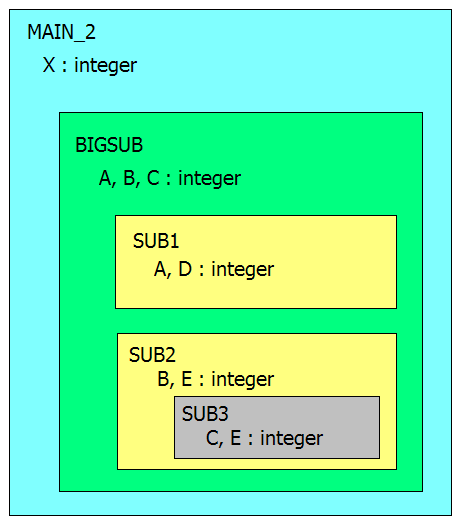
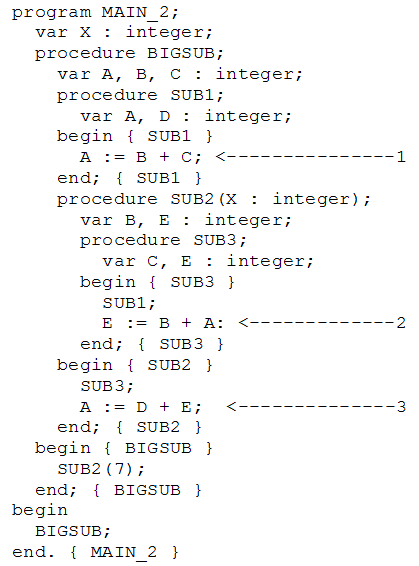
* An integer associated with a static scope
* Static depth is the depth of nesting of that scope
* In the program diagram below,
* MAIN has static depth 0
* A has static depth 1
* B has static depth 2
* C has static depth 1



**Chain offset**

* The ***chain offset*** *(*or ***nesting depth)***of a nonlocal reference is the difference between the static depth of the reference and that of the scope where it is declared
  + Chain offset = Depth(reference) – depth(declaration)
  + This number indicates how many times we have to follow the static chain to get to the declaration of the nonlocal reference
  + What does it mean if chain\_offset is 0? It’s local
* A variable reference can be represented by the pair: **(chain\_offset, local\_offset)** where local\_offset is the offset of the variable in the frame where it’s defined.

Example:



When we reference C in sub1, where is C defined?

Static scope: the C declared in bigsub 🡨 can be determined by looking at the program structure

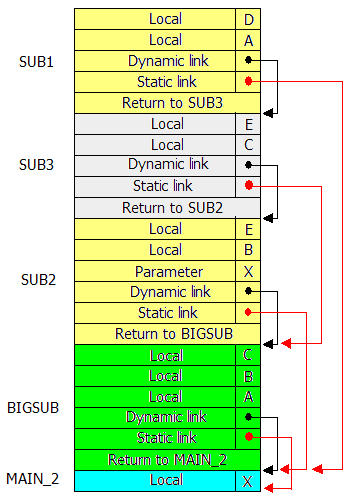
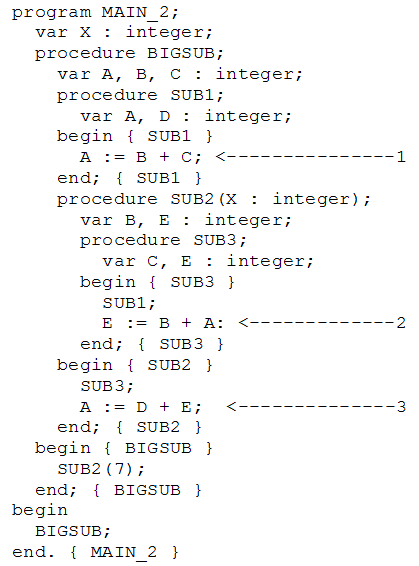
Dynamic scope: depend on the calling sequence:

* + 1. Main\_2 🡪 bigsub 🡪 sub2 🡪 sub3 🡪 sub1

Use the C declaration from sub3

* + 1. Main\_2 🡪 sub3 🡪 bigsub 🡪 sub2 🡪 sub1

Use the C declaration from bigsub



|  |  |
| --- | --- |
| **Static scope**  **(chain\_offset, local\_offset) for each variable** | **Dynamic scope**  **(chain\_offset, local\_offset) for each variable** |
| **At position 1 in** SUB**1:**   * A **- (0, 3)**   + B **- (1, 4)**   + C **- (1, 5)** | **At position 1 in** SUB**1:**   * A **- (0, 3)**   + B **- (2, 1)**   + C **- (1, 1)** |
| **At position 2 in** SUB**3:**   * + E **– (0,4)**   + B **– (1, 4)**   + A **– (2, 3)** | **At position 2 in** SUB**3:**   * + E **– (0, 4)**   + B **– (1, 1)**   + A **– (2, 3)** |
| **At position 3 in** SUB**2:**   * + A **– (1, 3)**   + D **– Error**   + E **– (0, 5)** | **At position 3 in** SUB**2:**   * + A **– (1,3)**   + D **– error**   + E **– (0,1)** |

Potential problems from using static chains:

* + A nonlocal reference is slow if the number of scopes between the reference and the declaration of the referenced variable is large
  + Time-critical code is difficult, because the costs of nonlocal references are not equal, and can change with code upgrades

How do we reduce the time spent searching through the static chain?

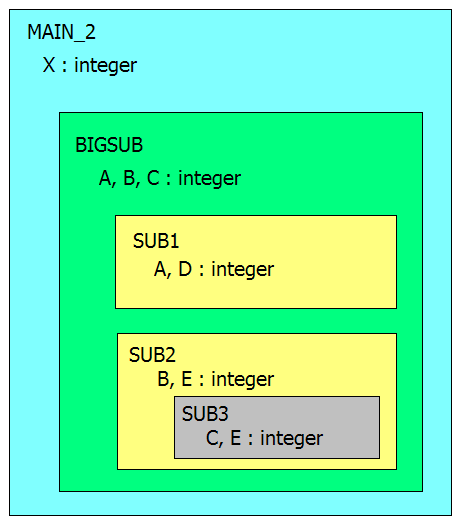
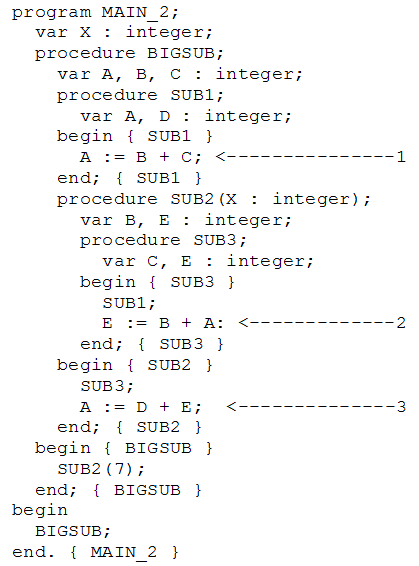
**Displays**

* Instead of storing and using static links in the frame on the execution (or runtime) stack, the static links can be placed in a ***separate stack called a display*** – see picture on next page
* *The entries in a display are pointers to the ancestor frames that store the variables in the current referencing environment (no more chain search).*
* Using this technique, addresses of variables take the form (*display\_offset, local\_offset*) where:
  + *display\_offset* is the static depth of the variable reference. It serves as an index into the display
  + *local\_offset* is the offset of the variable within the activation record.

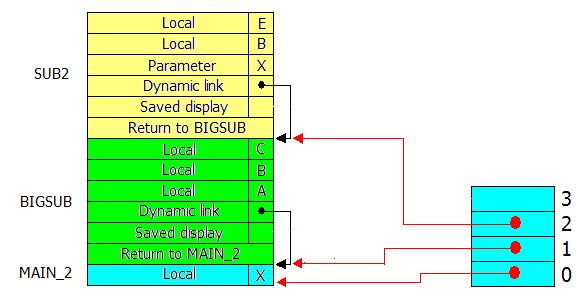
Display maintenance

* The display\_offset is the static depth of the procedure whose frame is being built.
* There are k+1 entries in the display, where k is the static depth of the currently executing unit (k = 0 is for the main program)
* For a call to procedure P with a static depth of k:
  + Save display[k] array in the new frame
  + Set display[k] = fp of frame P at depth k (and it’s ancestors)
* When P returns:
  + Restore display from saved info
* To speed access, display could be kept in registers

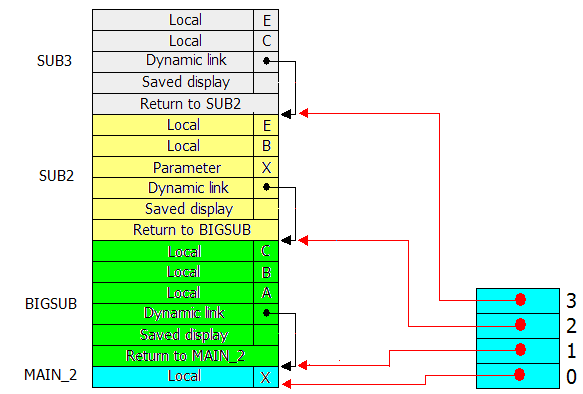
Example:



BIGSUB calls SUB2



SUB2 calls SUB3



Static chain vs. display

* Reference to locals
  + has similar performance
* References to nonlocals
  + - For static chain, the further away the more costly the access
    - All references cost the same for display
* Procedure calls
  + - For 1 or 2 levels of depth, static chain is faster
    - Otherwise, display is faster
* Procedure returns
  + - Static chain is slightly faster because it does not have to update the display
    - Overall, if there are few references to distant variables, static chain is better.

8.3 Parameter Passing

* Parameter passing mechanisms have three basic implementations
  + Value (input argument)
  + value/result (copying, input/output argument)
  + reference (aliasing)
* Many languages (e.g., Pascal) provide value and reference directly
* C functions:
  + parameters passed by value (C)
  + parameters passed by reference can be simulated with pointers (C)

**void swap1 (int \*a, int \*b) { int tmp=\*a; \*a=\*b; \*b=tmp; }**

**/\* how do we call this to swap value of variables Y and Z? \*/**

**swap(&Y, &Z).**

* or directly passed by reference (C++)

**void swap (int &a, int &b)**

**{**

**int tmp=a;**

**a=b;**

**b=tmp;**

**}**

* Ada goes for semantics: specify exactly what one can do with the parameters
  + In: callee reads only
  + Out: callee writes and can then read (formal not initialized); actual modified
  + In out: callee reads and writes; actual modified
    - Ada in/out is always implemented as
      * value/result for scalars, and either
      * value/result or reference for structured objects
* In a language with a reference model of variables (ex. lisp), pass by reference (sharing) is the obvious approach
* It's also the only option in Fortran
  + If you pass a constant, the compiler creates a temporary location to hold it
  + If you modify the temporary, who cares?

8.4 Generic Subroutines and Modules

* Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects
* Generic subroutines (methods) are needed in generic modules (ex., classes), and may also be useful in their own right, ex., swap

// generic swap function

templace <class T>

void swap(T &a, T &b) {.. }

// call it

swap<int>(Y, Z);

8.5 Exception Handling

* What is it?
  + a hardware-detected run-time error or
  + unusual condition detected by software
  + Examples
    - arithmetic overflow
    - end-of-file on input
    - wrong type for input data
    - user-defined conditions, not necessarily errors
* What is an exception handler?
  + code executed when exception occurs
  + may need a different handler for each type of exception
* Why design an exception handling facilities?
  + allow user to explicitly handle errors in a uniform manner
  + allow user to handle errors without having to check these conditions
  + explicitly in the program everywhere they might occur

See eh.cpp for example in C++

8.6 Coroutines

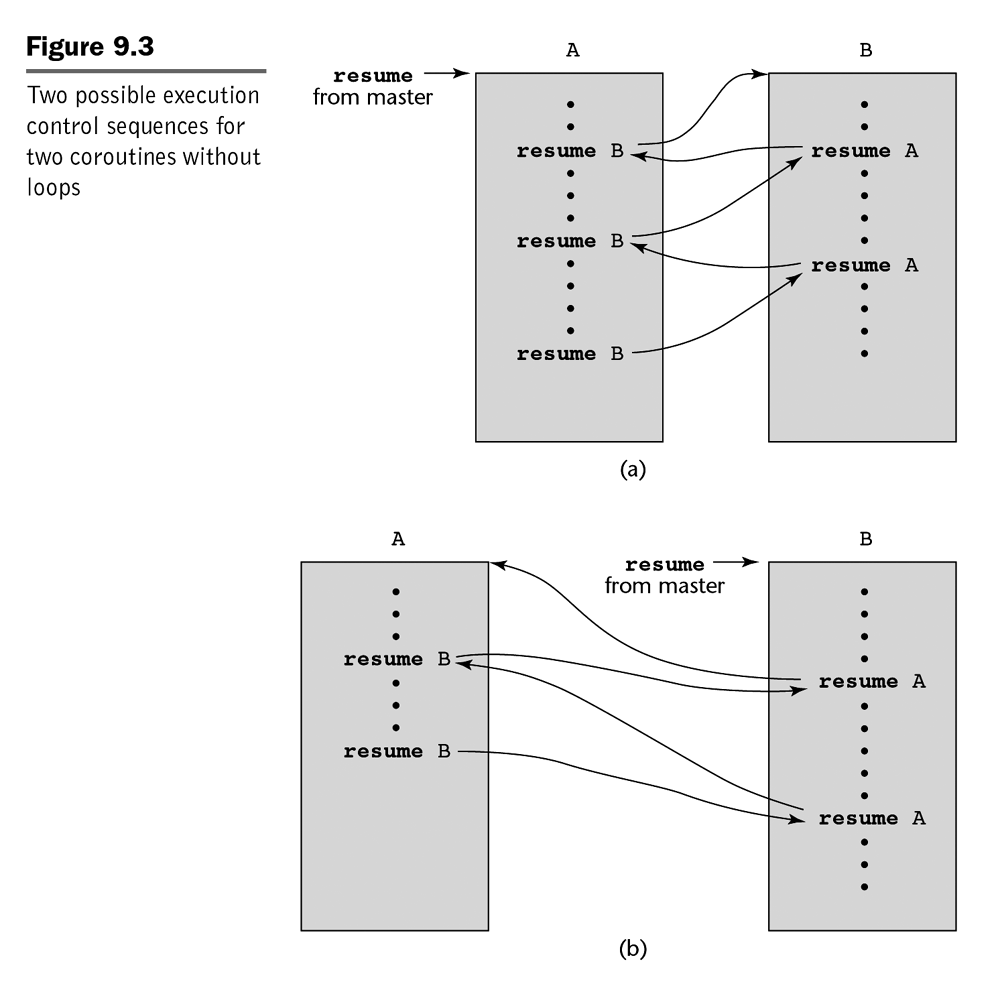
* Subprograms that call each other and run concurrently.

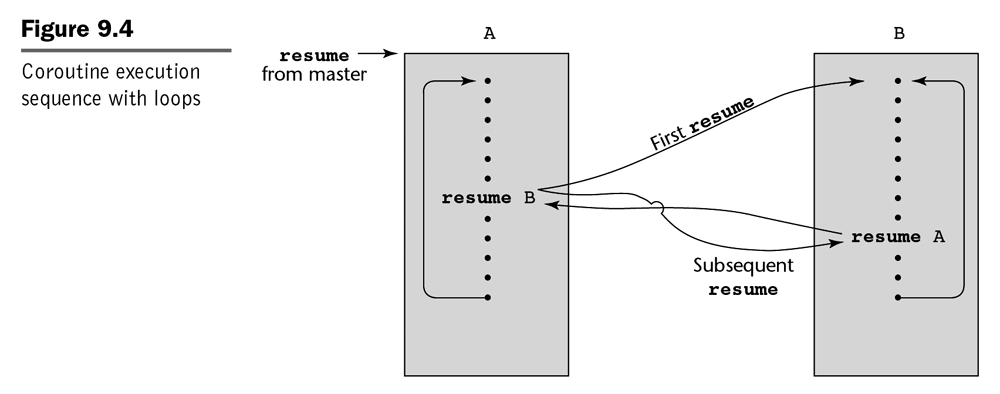
Example – see figure 9.3 & 9.4 on next page

* Their execution can halt and *resume* at previous stopping points.
* Are history-sensitive.
* Co-routines are often created by a program unit called *master unit*.
* Very useful in operating systems, ex., producer/consumer, screen saver

Subroutine vs. co-routine vs. thread

* Subroutines are LIFO, the last subroutine called is the first one done
* Coroutines terminates according to user specification or data
* Coroutines cannot progress at the same time
* Threads can progress at the same time

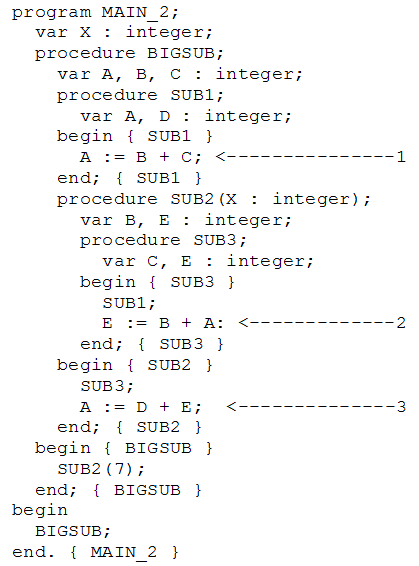


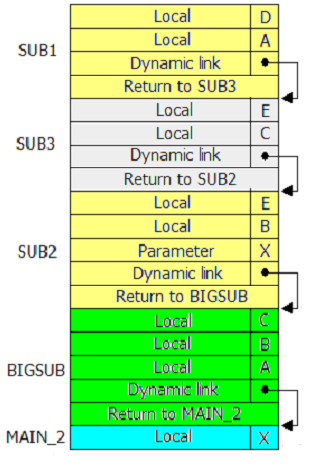


Accessing variables with dynamic scope

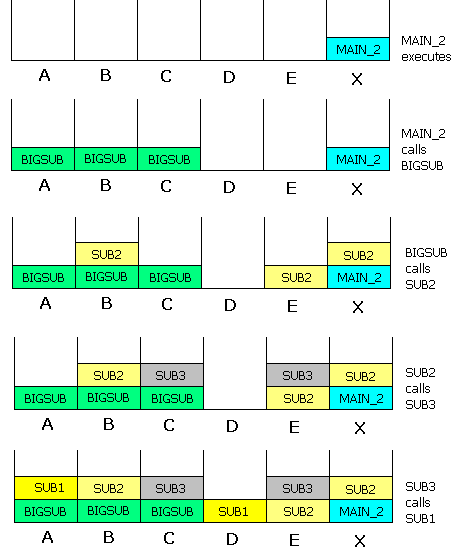
Two methods

1. **Deep access -** When you need to find a variable, you must hunt down from top of stack, following the dynamic link
   1. Use one stack (*association list*) for all active variables
   2. Use two stacks: one for LRE or one for variables.





1. **Shallow access** - keep a central table with one stack for every variable name
   1. The table layout (and the location of every slot) can be fixed at compile time as long as no new name creation are allowed at run time (most languages meet that requirement, including C++)
   2. For each variable in a called subprogram, push a new entry of the appropriate stack
   3. When subprogram exits, pop appropriate stack entries
   4. gives you slow calls but fast access



\*/