Bindings:

In a program objects; variables, of types Integers, Arrays, Pointers, Locations, Procedures and their Parameters, Modules and Classes etc . are all named. Each object receives a name, as well as delineation of the structure , type ex., an array of integers.. Before the program is prepared for running on a machine these are simply Names of objects. That is the state of the objects before the program Interpreted or Compiled into a machine language. In the process of Compilation for running on a machine numbers will replace many of those names, i.e., the names **will be bound t**o numbers. Some will be bound to code for specifying numbers (location of pointed to arrays, or procedure arguments) at run time. When the program is loaded into MM some of these relative numbers (locations) will be bound to fixed locations. Then during run time the code for specifying locations will be executed and that will bind the objects originally named to locations. An object with a given original name may be bound to different locations at different times during the run of the program. Also there may be times when there is no binding for the object or the object itself may be be destroyed at different times during execution. So one speaks of binding times, as static and dynamic .

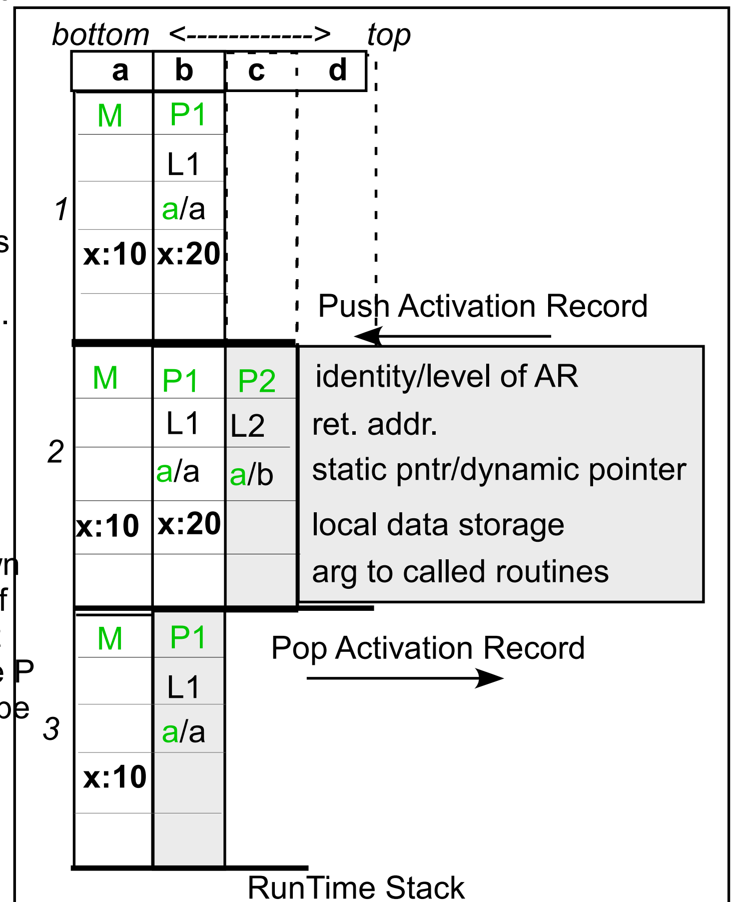
Static Allocation

Bindings require the allocation of memory. For some objects this is done once, usually at compile time it static because the binding is done at compile time. This is the case for Global variables for declared constants, numerical or string valued, #define N 40, printf("yeah, yeah\n"). It is also done for instructions which may contain constants. There is code for monitoring stacks and heaps which themselves are dynamic memory allocation areas, however that code is static during run time

Stack Based Dynamic Allocation.

When a programming language allows the use of procedures, and further there is an expectation of long sequence of calls of procedures calling each other, as may be inevitable in allowing recursive calls one cannot know exactly how much storage to set aside for the cascading number of sets of parameters, local data, and return addresses (per procedure) that may be necessary. (In Fortran initially recursive calls were not allowed and the storage necessary for each procedure, its local data and parameters could be static.)

When there are chains of calls of procedures one calling another there must be storage for each because upon return from an active procedure control goes back to the calling procedure which must have access to its local data etc. An **Activation Record, AR**, with this and some additional information is kept for each procedure in a chain of calls in a Run TIme Stack of ARs. The AR for the currently active procedure is on the top of the stack, the AR of its caller just below it, the AR of the caller of the caller jut below tha t etc. A picture of such a stack is shown on the right, in different states at 1 and 2 and 3 (the stack top is represented on the right. A call from the active procedure P1 (at 1,b) of P2 results in the AR of P2 being put on the top of the stack as shown at 2 c. When P2 completes it will be popped off the top of the stack, exposing P1 as the active procedure again. It should be clear that while running the AR for a procedure P will be at the top of the stack and also copies of P may be down on the stack. further



© M C Paull 2008

Though it impossible to know before running the program the exact amount of memory needed for the procedure related memory an upper bound is usually assumed. The stack like most statically assigned memory (global, code space) is generally alocatted at compile or load time. It is part of the protected area assigned to a Process. This memory is generally accessed and protected via page tables in a paging environment.

The Heap

There is need also for other dynamic memory changes at runtime For example, link list cells may be added and removed at run time. The Heap is a separate memory area managed by the Operating System where this growth and shrinkage is accommodated. In some systems (Lisp, and Scheme) this is organized as a l**arge link list in tree form**. In imperative languages, ex.Modula, C, however the new chunks of memory are added at the next available address as a continuous units.

In either case, such memory acquistions must be freed for reuse when no longer needed **Freeing is sometimes managed implicitly,** independent of the program when the system detects that a portion of the memory is no longer needed. It can also be done **explicitly by the programmer who has available appropriate "freeing" commands**.

It is usually done **implicitly for link lists in a tree form type of Heap**. This requires periodic **Garbage Collection, i.e.** sweeping through the tree to detect portions of memory no longer reachable from the tree root and therefore reussable. **For the chunk form of memory explicit commands are usually the main means for determining which Heap portion are freed.** Nevertheless **Garbage Collection** is still sometimes neccessary. The eventual result of such additions and subtractions is that the Heap eventually becomes a sequence of small holes and chunks. It is then hard to find enough space to accommodate the next chunk request. The memory management problem is then, keeping track of where and holes are. The data structures that are used for this purpose are

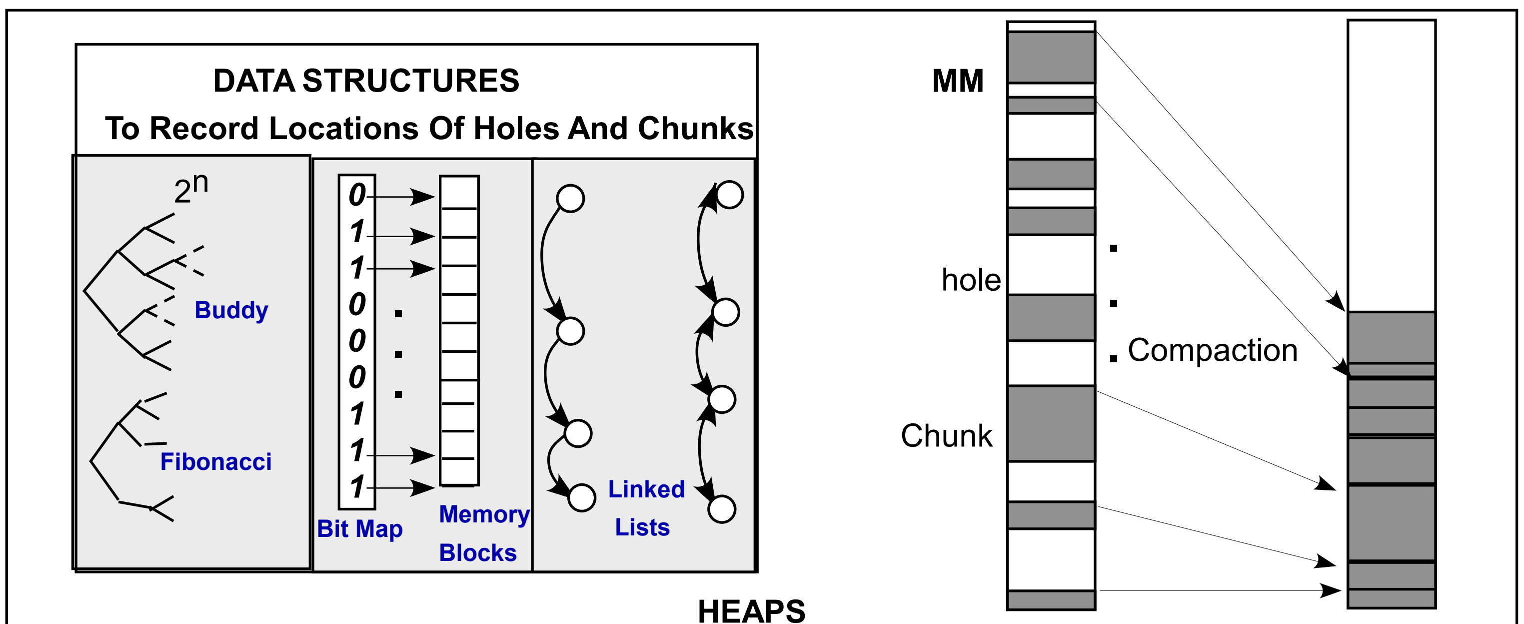
1 link lists connecting the fronts of holes and chunks,

2 bit maps, with 1 bit per block, the memory being segmented in a series of fixed size blocks,
The bit is 1 if the corresponding bit is occupied by a chunk, 0 otherwise.

3 A tree, with each node corresponding to a location in the memory at which a chunk may be stored.
In the Buddy Algorithm the tree branches in 2 at every level, in the Fibonacci algorithm
the branching factor at each level k is that of the kth Fibonnaci number.

The first two require O(n) time to find a hole of the size to fit the next arriving chunk, the latter two can do this in O(logn) time but result in parts of memory becoming unuseable.

No matter which data-structure-algorithm is used however, eventually there are many small holes and a more heroic measure is necessary, to make adequate space available. That measure is **Compaction.**



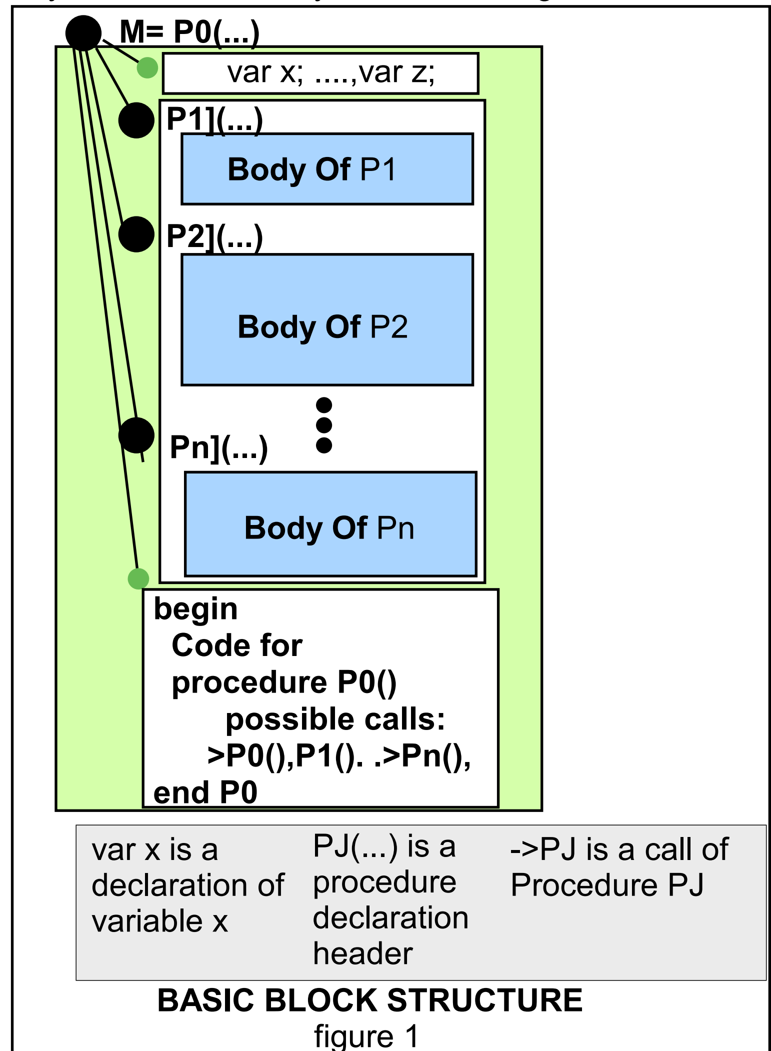
1 Scope And The Runtime Stack

Large programs usually can be partitioned into logically independent regions. Diifferent languages support different views of this **modularization**.

A functional language, such as Scheme, views a program as a function consisting of a composition of functions, each of which is itself a composition of functions etc. That view is extended somewhat in the view of a program as a set of procedures to be executed one after the other, the result of one providing the input to the next. Then each of those procedures is itself considered again as a sequence of procedures, and so on. These views are "**top-down**", both focus on the functions or procedures which compose the program. The sequence view is supported in many imperative languages based on the machine model by the availablility of block structuring. Another way in whichto modularize a program, based on data-structures and associated functions as central is supported today in many of the same languages.

1.1 **Block Structuring** ***allows the same names, of data structures, locations, and procedures, to be used in defining different*** *procedures. It also provides a* ***very structured way of passing information*** *between procedures.* With this kind of structuring different programmers can work on different procedures, and, beyond a few data structures used for communication, can name its data-structures and sub-procedures without concern for conflict. This is an **early step toward modularization-which culminates in Modules and Classes. Data** is defined and named within delineated regions. The names refer to the defined data only within those regions. The same name can then be used in another delineated block referring to a data structure within that other block. The block within which a name is applies is its **scope.**

This division of a program into logically isolated blocks or regions (blocks and regions are used interchangeably) provided by block structure for ease in program design also serves to limit the maximum amount of storage required during the run of the program. Blocks which are unlikely to be needed in storage at the same time can be identified so that memory is maintained only for relevant regions at each instances during the run of the program. This is **dynamic storage allocation** and it is generally implemented by maintainence of a **run-time stack** with storage just sufficient for data-structures at all times during the run of the program.



BASIC BLOCK STRUCTURE
figure 1

Procedure Format

The block structure we consider here is based on a fixed procedure format.

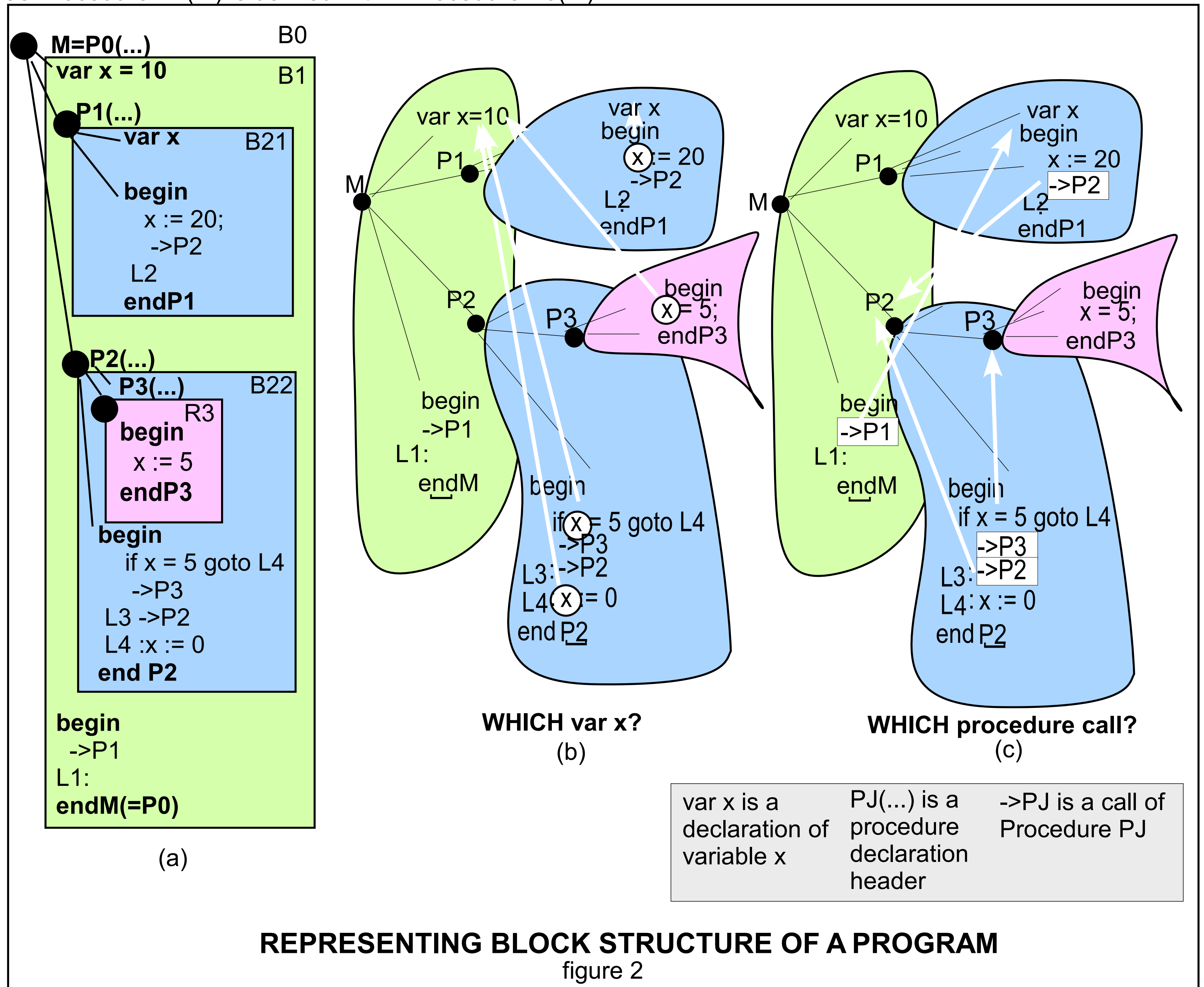
A Procedure Declaration: gives the procedure name (say P0) and its arguments followed by the Procedures Body: That body consists of three parts:

1) Declarations of Variables ,
2) Procedure Declarations: Procedure headers, and their Bodies, (which each again contain contain 3 parts
and finally
3) P0'[i]s program.
(Parts 1 and 2 are optional)

In a Block Structure there is a hierarchy of Procedure declarations. In the figure, P1 (...), P2 (...), Pn](...), all being declared and detailed within procedure M=P0(...),are considered to be the immediate children of their parent P0(...). And if M=P0(...), is at level i, then P1 (...), P2 (...), Pn](...), are all at level i+1. This is illustrated by the two levels shown in figure 1.

© M C Paull 2008

An example of a simple set of procedures in this block-structured form is shown below in figure 2 (a). In this example the main procedure M=P0(...) has 3 parts; Variable Declarations, Procedure Declarations and P0(...)'s Code. Also the Procedures declared within P0(...) can have any or all of the same 3 parts. In this example there is a procedure (P3(...) defined within Procedure P2(...), while Procedure P2(...) as well as Procedure P1(...) is defined within Procedure P0(...).



REPRESENTING BLOCK STRUCTURE OF A PROGRAM
figure 2

In general a program is divided into Procedure specified blocks and enclosed Procedure specified sub-blocks (regions) whose relations can be modelled by a generation tree in which Procedures Blocks are associated with the nodes. If Block B at level **directly encloses** block B' Then B is the **parent** of B' at level i+1.If B **encloses** (not necessarily directly) B'' B is an **ancestor** of B''. There are examples of these definitions In figure 2(a): the Block at level 0 (specified by) **P0(...) directly encloses** blocks **P1(...)** , and Block **P2(...)** all at level 1, and **Block P0(...) encloses P3(...) at level 2.** Or using the language of a tree **P0(...)** is **the parent of P1(...)** and is the ancestor of **P1(...), P2(...),** and **P3(...).** Also **P3(...)** is a **descendant of P0(...).** An alternative representation of the same program which more clearly shows the ancestor-descendant relations is shown in (b) and (c) of the figure. These are used to display the relation between call and declaration of variables and Procedures to be considered next.

1.2 Referencing Variables and Procedures In Block Structures

The same variable or procedure name can be referenced in the code portion of different nested Procedures. There are fixed rules by which the relation between the reference to, and the declaration of a

variable or a procedure is established. In fact there are two independent sets of such rules, there are languages which use each. The most reasonable of these rule sets is are the **Static Rules.**

1.3 Reference Rules For Variables and For Procedures:

1.31 **Static Scoping**

Variables:

In general a variable V declared within any block B (instanceV-in-B) is available in B and to all descendant regions in which it is not re-declared. Therefore if the same variable is again declared in B' a descendant of B, (instance V-in-B') is available in B' and to all descendant regions in which is not re-declared.

Another way of saying the same thing is:

A variable, V, referenced in region B refers to the declaration instance of V in B or if not declared in B, in the closest ancestor (enclosing) region in which it is declared.

Procedures: (This Rule is completely analogous to that for Variables **)**

In general a procedure P declared within any block B (instance P-in-B) is available in B and to all descendant regions in which is not re-declared. Therefore if the same Procedure is again declared in R' a descendant of R ,(instance P-in-R') is available in R' and to all descendant regions in which is not re-declared.

Another way of saying the same thing is:

A Procedure, P, referenced in region R refers to the declaration instance of R or if not declared in R, in the closest ancestor (enclosing) region in which it is declared.

Purpose Of Nesting Of Procedures

One can state the general effect of the organization of program development embodied in the proposed tree structure of nested regions with **Static Scoping,**

In the Main Procedure all variables, which are to containing information to be accessed in all descendant regions associated with Procedures defined within the Main program are defined. They form a common pool of **global** variables accessable to all Procedures in descendant Regions so long as those variables are not re-declared in such a descendant. A similar statement applies to the Procedures defined in the Main program-they are accessable to all Procedures which are descendants of the Main Program.

What is true of the Main Program is analogously true of any procedure defined at any level-its name and its variables are accessable to all descendants Procedure unless redefined with the same name in any descendent.

This then extends the idea of a Main program containing global variable. The model is of a Main program outlining what is to be done and the Procedures defined therein performing the detailed operations necessary to accomplish the Main program outline-and extending the same kind of outlining functions to the Procedures which are descendants of the Main Program.

Another way to say this follows

In designing a procedure, P, one is interested in using some variables whose value is local to the procedure in that they are guaranteed not to be affected by any call within P. In addition to such variables there are those which we do wish to be altered by calls within P-these are global with respect to P. These seem to be natural distinctions amongst variables, and they imply that care be taken to insure the desired properties in a block structure environment. If variable x is to be local in P it should be declared in P, and if used in any procedure called within P and defined within P declared there also. If the variable y is global then it should not be declared in procedures called by P and defined within P which we wish to effect y.

This block-structured conceputalization of design of a program bares a close relation to the ideas of Abstract Data Structures..

Nesting and Abstract Data Structures

In an Abstract Data Structure the data structure consists of a set of variables. These can be declared within a Procedure P. This becomes an abstract data structure when, in conjunction with the data structure, the procedures which are to operate on that data structure are defined. If these Procedures P_1 ,..., P_n , are defined directly within P, then in this Block Structured world they will have access to that Data Structure. Any number of Abstract Data Structures can then be defined within P. In the Block Structured world also Procedures P_1 ,..., P_n can have Abstract Data Structures defined internal to themselves-etc.

© M C Paull 2008

This in fact is a sometimes awkward way to incorporate Abstract Data Structures in a language, Better constructs, namely Modules and Classes, can incorporated into into a language more gracefully

1.32 **Dynamic Scoping**

In some languages (Lisp) another interpretation that is made of a program consisting of nested Procedeure and variable declarations. This is the Dynamic Scoping interpretation. The structures of the Programs are the same with nested Procedure and Variable declarations, but they have an effect considerably different than with Static Scoping. Under this interpretation, when in the running Program, **variable, V, is referenced in Procedure P, V refers to its declaration in P unless it is not defined there. If it is not defined, there it refers to the declaration within P' the procedure from which P was called in this run of the Program.** If V is not defined in P' then it refers to V's definintion in P''-the caller of P' in this run of the program. Etc.

This paradign can result in a procedure P when referring to variable V **referring to a different declaration of V in different procedures on different runs.** This is a fairly incoherent way to understand the significance of the Nesting. The choice of Dynamic Scoping in a language is based on the ease in implementation of such scoping, rather than the conceptual advantage given by Static Scoping.

1**.4 Implementation Of Block Structured Language**

The implementation of a Block Structured language requires:

1. That reference to a variable or procedure by name be to its declaration in the most closely enclcosing block of the currently active procedure,.

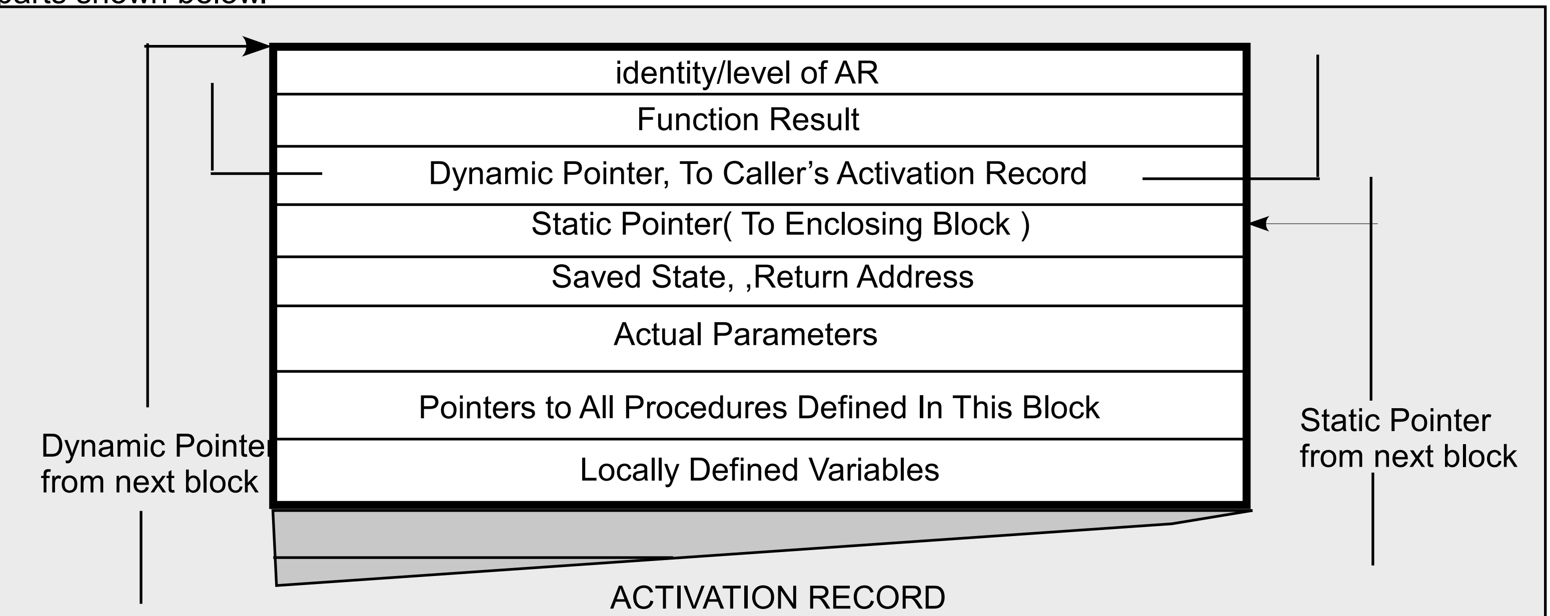
2. That while the current version of variable or procedures are correctly used, versions of the same variables **in enclosing blocks be maintained ready to be reactivated** when operation in the currently active block is completed. Parameters equal too need to be maintained in caller blocks.

3. When the current block i**s completed the storage for information local to that block need no longer be maintained**

Note that requirements must also be satisfied by parameters of called procedure. (Parameters become local variables within a procedure given value by the caling prlocedure). These requirements are extensions of the idea of a pushdown used for keeping track of currently active parameters and local variables. Forthe implementation of the block structured language then we use a Run-Time Stack

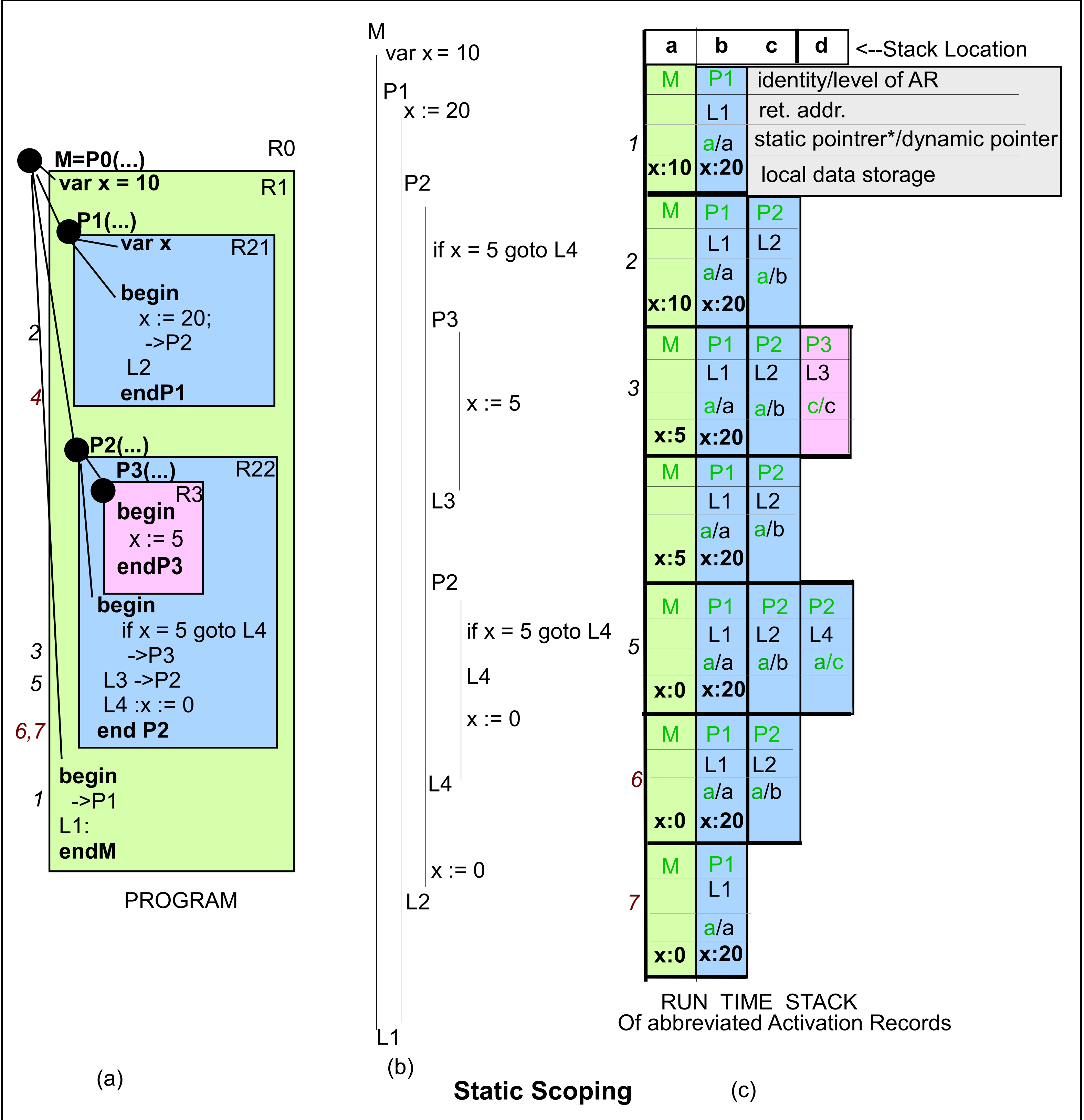
T**he Run-Time Stack**

This implementation is used for both static and dynamic scoping, in fact for any language, block structured or not, which allows recursive calls, and thus is virtually universal. For both both naming isolation and dynamic allocation of storage is to be implemented. The implementation uses a **run time stack** of **activation records each** whose sizes is determined by the number an size of the local variables, parameters and procedure pointers of the corresponding Procedure, i.e., they differ in size . This neccesitates use of a pointer to indicate significant predecessors entries in the stack. These pointers should always be found in the same location relative to the top of the stack frame. Each time a procedure is entered a new entry is pushed onto the stack, and each time a procedure is vacated the top entry is popped off. An entry or stack frame contains the parts shown below.



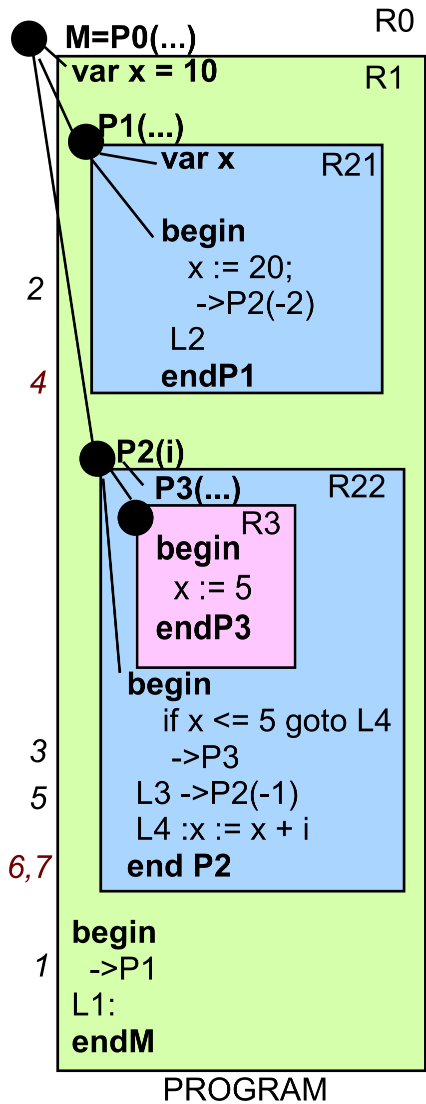
© M C Paull 2008

A detailed example of the history of the run time stack in the execution of a program with nested procedures is given in figure 4 (b). This is the same simple block structured program example as that in figure 2 . The trace follows the program as it starts in the code for M (at the bottom of the Program) and contnues through a call of P1 then the from P1 a call of P2 then in P2 a call of P3 followed by a recursive call of P2 again. The consequence of this execution sequence is traced in the run time stack shown in (a). The stack are shown horizontally with their tops to the right. The stack evolution starts at the top of the page in (a). The initial entry in the stack is made when program M starts. New versions of the stack are shown after each call and after each completion of a called Procedure. These events are numbered 1 through 7 in the two program representation and in the stack-black for a call and red for a procedure completion.



Static Scoping

© M C Paull 2008



Step	a	b	c	d
				<--Stack Location
1	M x:10	P1 L1 a/a x:20		
2	M x:10	P1 L1 a/a x:20	P2 L2 a/b i:-2	
3	M x:5	P1 L1 a/a x:20	P2 L2 a/b i:-2	P3 L3 c/c
	M x:5	P1 L1 a/a x:20	P2 L2 a/b i:-2	
5	M x:5	P1 L1 a/a x:20	P2 L2 a/b i:-2	P2 L4 a/c i:-1
6	M x:4	P1 L1 a/a x:20	P2 L2 a/b i:-2	
7	M x:2	P1 L1 a/a x:20		

Key to each activation record (top to bottom):
- identity/level of AR
- ret. addr.
- static pntr/dynamic pointer
- local data storage
- parameters

P2 doesn't look at i until x becomes 5

RUN TIME STACK
Of abbreviated activation record

(c)
TRACE With Parameter Passing

© M C Paull 2008

The Closest Enclosing Static Block In The Run Time Stack

Consider static scoping and the use of the run time stack. When a variable or Procedure is referred to in procedure P and found to be absent from the top activation record it must be in one of the enclosing regions. However this need not be the activation record of the calling procedure to which the current activation points. We need a rule for finding the activation record of the **statically** enclosing block of a given block.

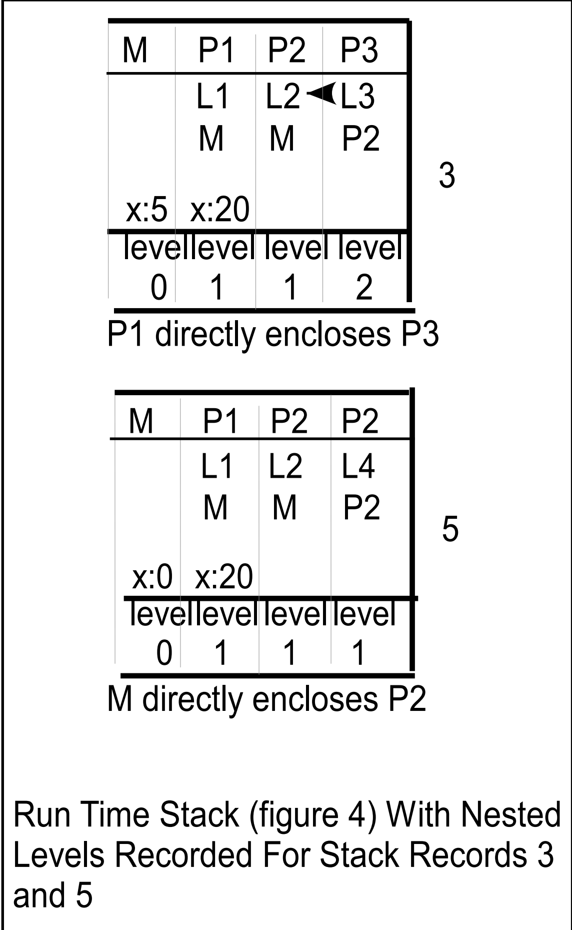
If the level of nesting of a block in the program were kept in the stack frame then:

For block B, with level *x* the closest enclosing block's entry in the stack. is the the first block of level *x-1* found, looking back into the stack,. This is not obvious since block B at level *x* is enclosed in **a** level *x-1* block,but there are, in general, other *x-1* level blocks which do not enclose B, though they may also be in the stack.

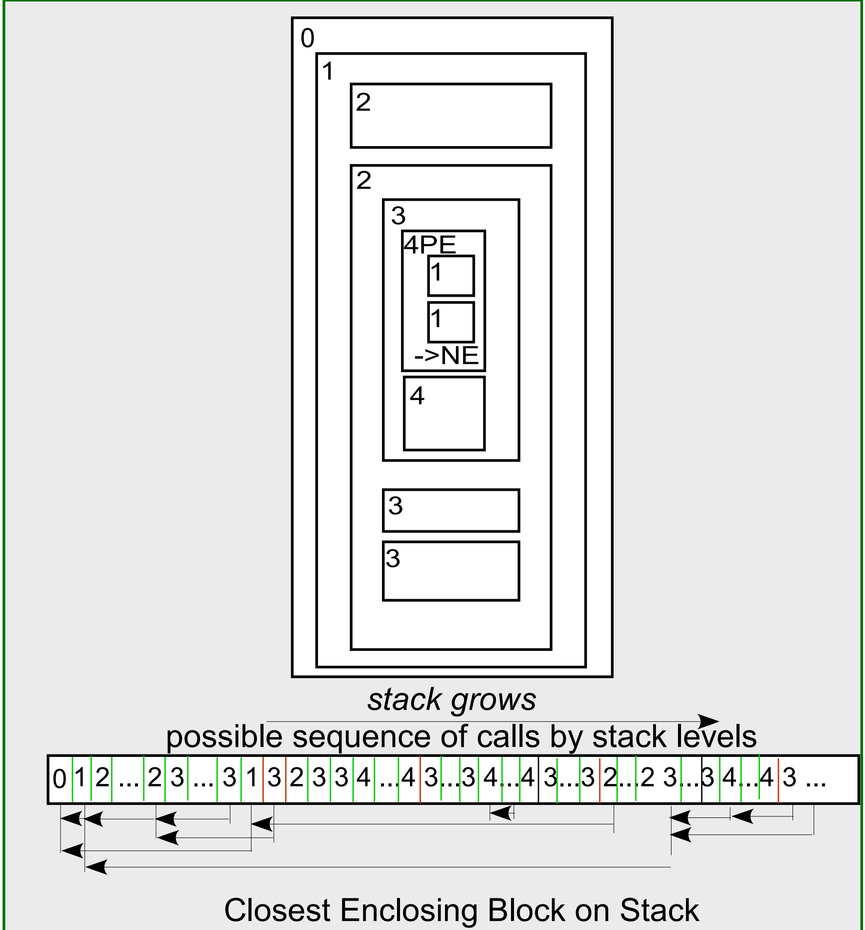
This needs to be demonstrated, which is done in the next paragraph.

Properties Of The Run-Time Stack

The object is to show that for any block enclose in M, looking back in the run time stack there will always be the relevant Activation Record for a **directly** enclosing block and furthermore that directly enclosing block can be found by following the dynamic links back from an entry, E, of level *x* to the first of level *x-1*, and that is the first enclosing block of E. So The static stack pointer can be determined from the static levels of the blocks, by **following the dynamic links back from an entry, E, of level *x* to the first of level *x-1. t*hat is the first enclosing block of E.** In figure 5 (a) we have shown the runtime stack 3 and 5 of figure 4 with the their block levels shown verifiying the claim. In figure 5 (b) a more extended example of a possible sequence of calls their corresponding run time stack entries. ond the location of the directly enclsing entries is shown.



Run Time Stack (figure 4) With Nested Levels Recorded For Stack Records 3 and 5



Closest Enclosing Block on Stack

figure 5

Proof: Let the outermost Procedure Block be numbered 0, a Procedure nested directly in Region 0 be in block 1. In general the Procedures directly nested in Block i, is numbered i+1. Then in running a block structured program, we are dealing with a sequence of numbers indicating the Block to which a call or a

© M C Paull 2008

return takes us. That **number** can:

(a) remain the same as procedures at the same level (depth) number, possibly recursively are called, or

(b) (the number) get lower at by a return.or by calling a procedure at a smaller level number (enclosing block),

(c) but it can only get higher level number by adding 1 to the previous level number. And this adding of 1 occurs when a procedure in a block calls a procedure in a directly enclosing block.

Assume after *n* entries in the procedure stack have the desired property namely that the enclosing block for any stack entry at level *i* is the first at level *i-1* looking back in the stack. It follows that

if there is a return decreasing the stack length by one activation record this property still holds or

if a new entry is made at the top and is at the same level as the current top or lower then the closest enclosing block is the same as it was for the previous block.

if a new entry, the *nth*, is made at the top and is at level *i+1* then the previous top at level *i* is its encloser.

If the new entry is at level *i-j* then we can trace back to the level *i-1* block that enclosed the former top member of the stack at level *i,* from that *i-1* level encloser, we can trace back to the *i-2* encloser, etc. till finally we trace back to the *i-j-1* encloser of the now top member of the stack.

The following algorithm implements and verifies.these conclusions.

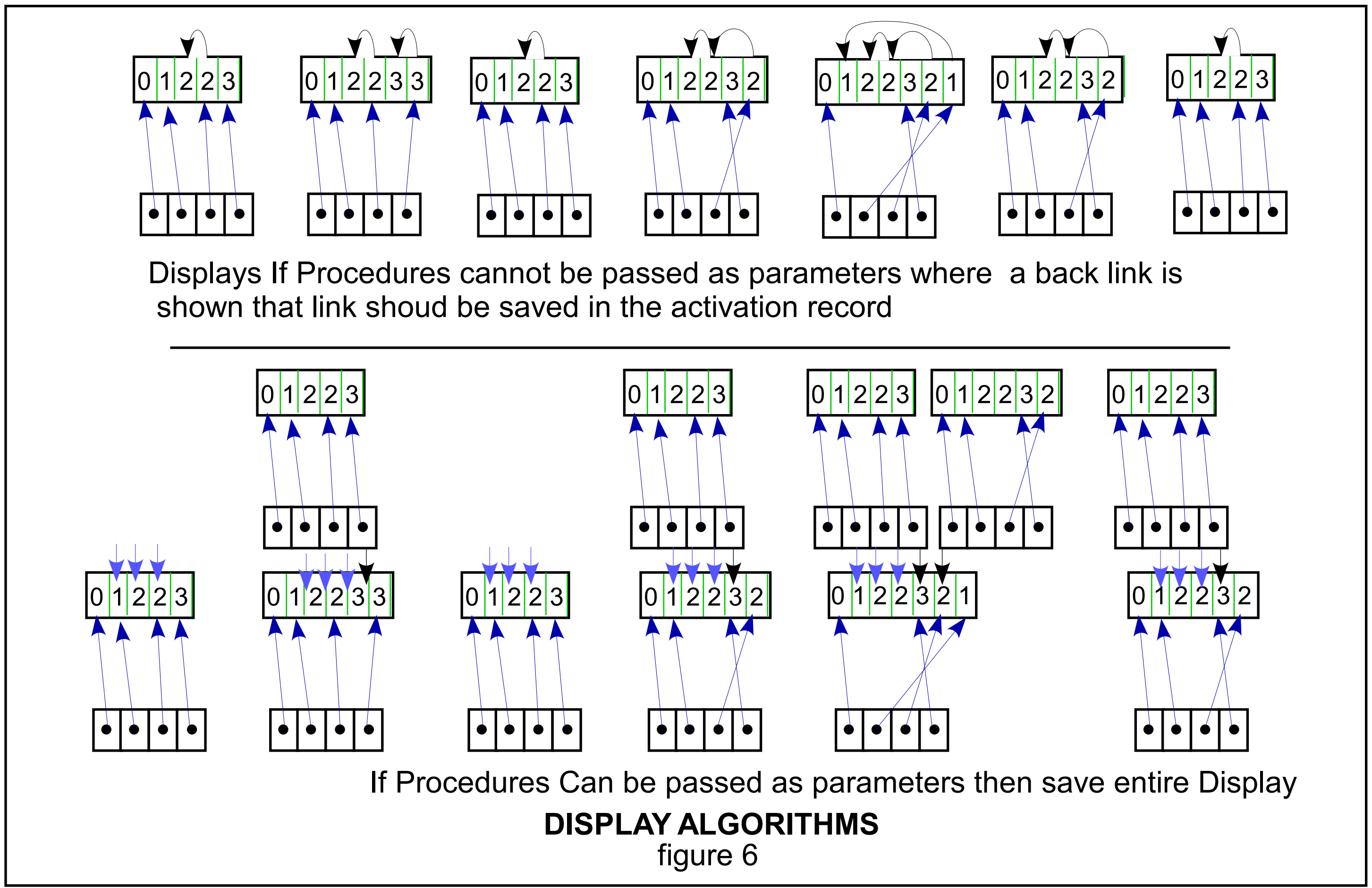
Implementation For Finding the Closest Enclosing Block.

The Static pointer in the Activation Record points to statically enclosing block. That pointer points to the calling block if it was the directly enclosing block, otherwise one could always find the closest enclosing block as follows: Given the block level number N of the top AR on the stack search downward through the stack (using the dynamic links) to find the first AR block level number equal to N-1. (This assumes the level Number of the block represented by an AR is kept in each AR.)

A more efficient implementation for finding the closest enclosing block is the idea of the "**Display**" which maintains a pointer to the current most recent stack entry at every level of ARs on the stack.

In order to keep the display updated we need to know the new value for display pointers 1) When a new block is put on the stack. and 2) when blocks are removed from the stack.

In order to do this it is necessary to keep information in the ARs themselves as well as in the display. The algorithm is described in detail on the figure containing an example of the use of the display.



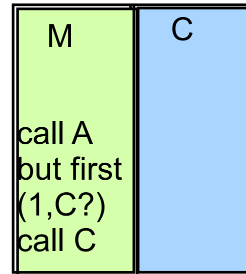
DISPLAY ALGORITHMS

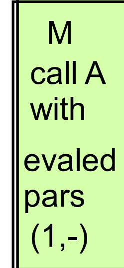
figure 6

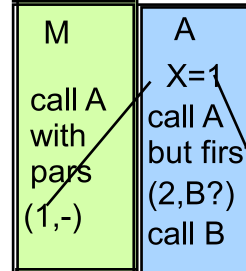
```
M: program binding_ex(in, out);
procedure A(X, integer: procedure P);

    procedure B;
    begin
       writeln( X );
    end;

  begin (*A*)
   if X > 1 then P
   else A(2,B);
  end

procedure C;
 begin
 end;

begin (*M *)
  A(1,  C);
end
```

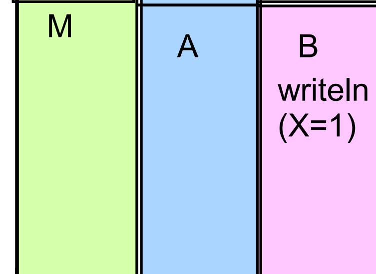
```
M: program binding_ex(in, out);
procedure A(X, integer: procedure P);

    procedure B;
    begin
       writeln( X );
    end;

 begin (*A*)
   if X > 1 then P
   else rB=B;
        A(2,rB);
 end

procedure C();
 begin
 end;

begin (*M *)
  rC=C()
  A(1,  rC);
end
```

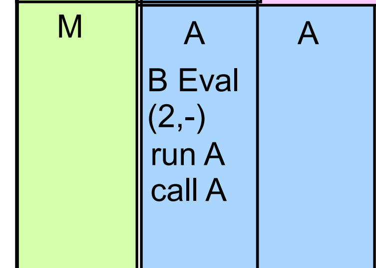
An equivalent Program For Deep Binding







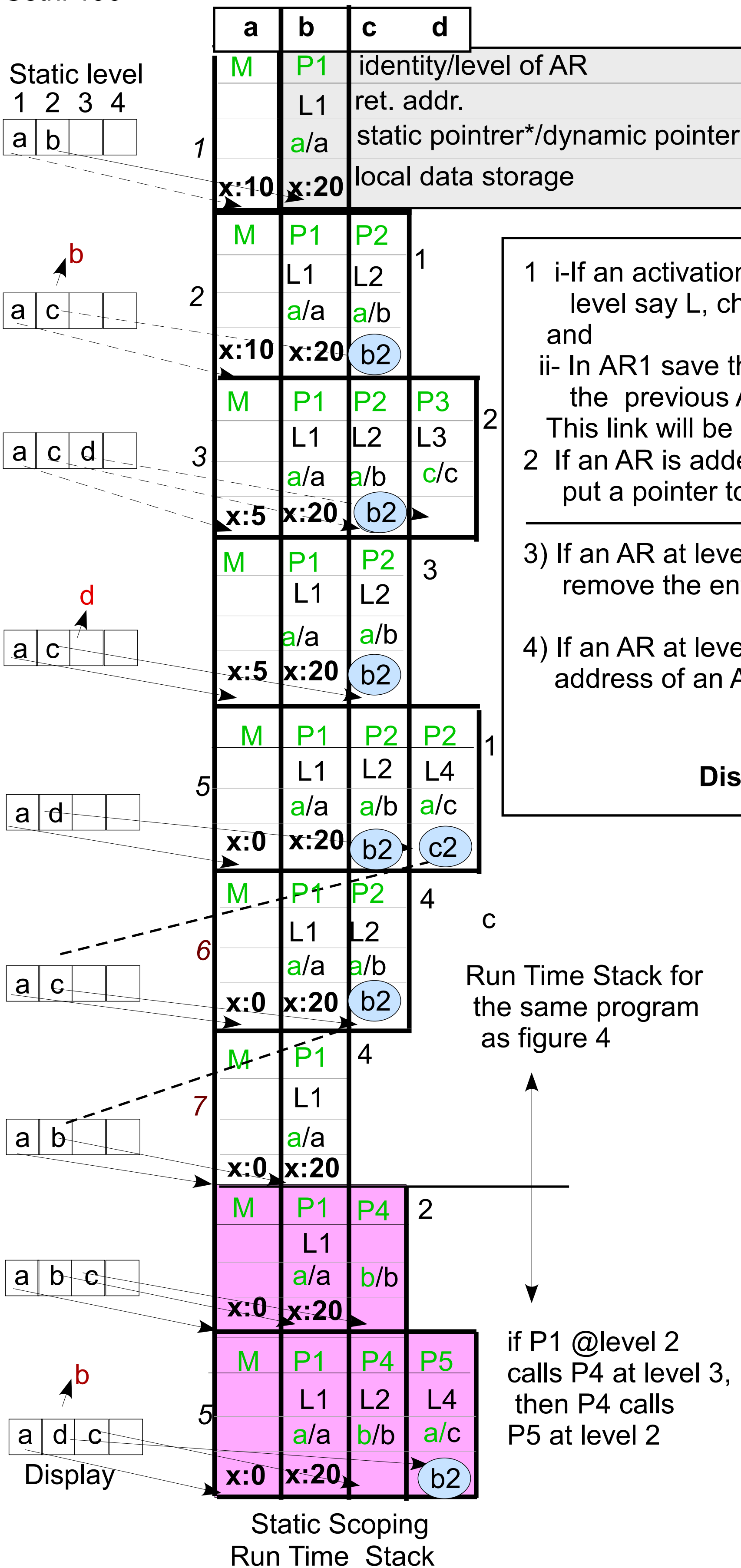




Evaluate all Parameters-which may require evaluating procedure calls before making call. with All evaluated parameters.

DEEP BINDING (WHEN PROCEDURE PARAMETERS REQUIRE PROCEDURE CALLS)

© M C Paull 2006



1 i-If an activation record (AR1) is added at the **same or lower** level say L, change pointer in Display[L] to point to AR1 and
ii- In AR1 save the pointer replaced in Display[L] as a *link* to the previous AR at level L.- (stack loc, display index ex. b2)
This link will be used when AR1 is removed from the stack.
2 If an AR is added at the **1 higher (deeper) level** say L, just put a pointer to it in display[L].

3) If an AR at level L is removed and it does not contain any *link* remove the entry at Display[L]

4) If an AR at level L is removed and it does contain a link address of an AR. Place that address at the display[L]

Display Control Algorithm

level=level number

proc	Static Level
M	1
P1	2
P2	2
P3	3
P4	3
P5	2

If B is a level x block in the Run Time Stack, then looking back in the stack - the first level x-1 block is that which encloses B in the program.

Enclosing Block Rule

Display points to the most recent activation record (Stack Frame) at all levels. In addition in the stack a pointer is kept to the previous stack frame at level i in each frame at level i starting with the second at level i.

TRACE WITH DISPLAYS IF PROCEDURES CANNOT BE PASSED AS PARAMETERS

figure 7

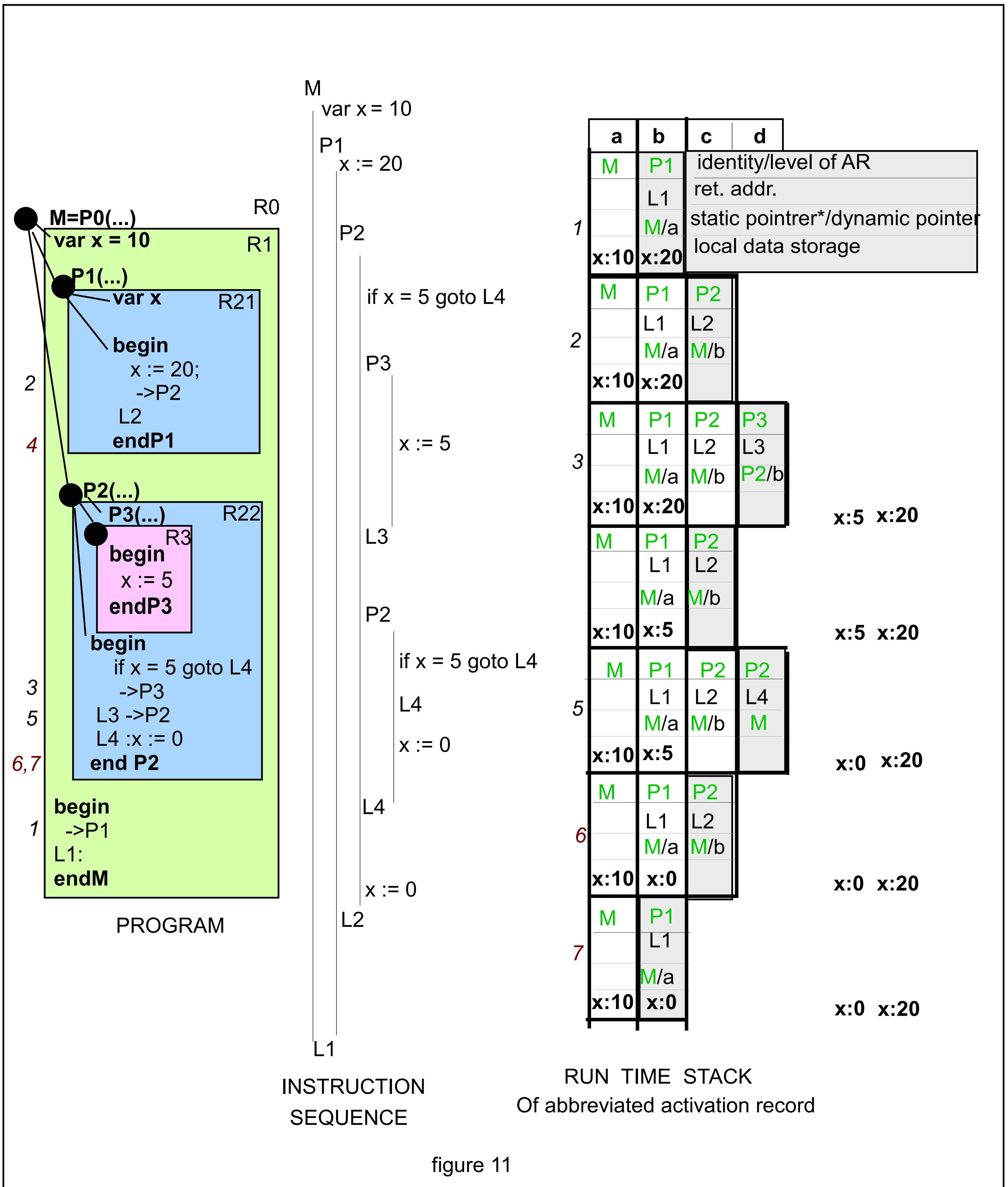
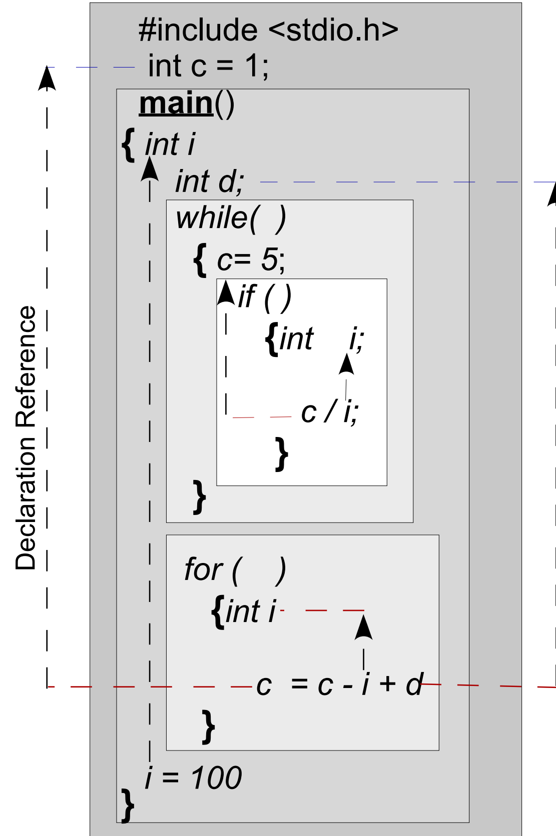


figure 11

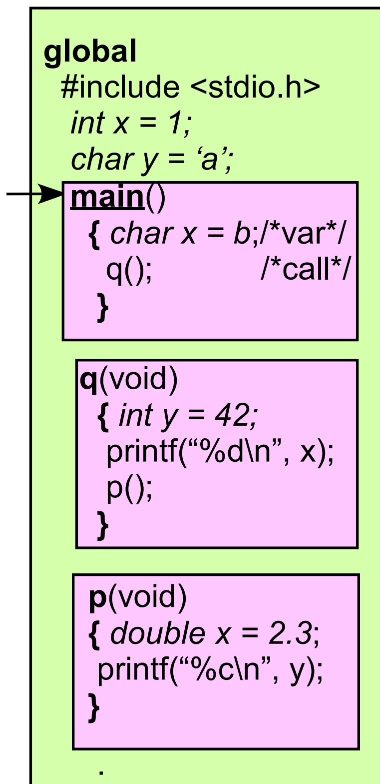
Dynamic Scoping

© M C Paull 2008

Block Structure In C:
In C the blocking is accomplished by the use of brackets, { }. inside the brackets variables can be defined they are alive until the end of the bracket when they are released. Procedure definitions are enclosed within brackets so variables defined there are local to that procedure. Procedures cannot be defined within procedures.



Block Nesting In C



block levels

0	1	1	1
glob	main		
	glob		
x:1	x:'b'		
y:'a'			
glob	main	q	
	glob	glob	
x:1	x:'b'		
y:'a'		y:42	
glob	main	q	p
	glob	glob	glob
x:1	x:'b'		x:2.3
y:'a'		y:42	

printf("%d\n", x);

printf("%c\n", y);

static	dynamic
1	92 [= 'b']
a	*[=*42]

RESULTANT RUN-TIME STACK

EXAMPLE OF SCOPING IN C

figure 9

© M C Paull 2008

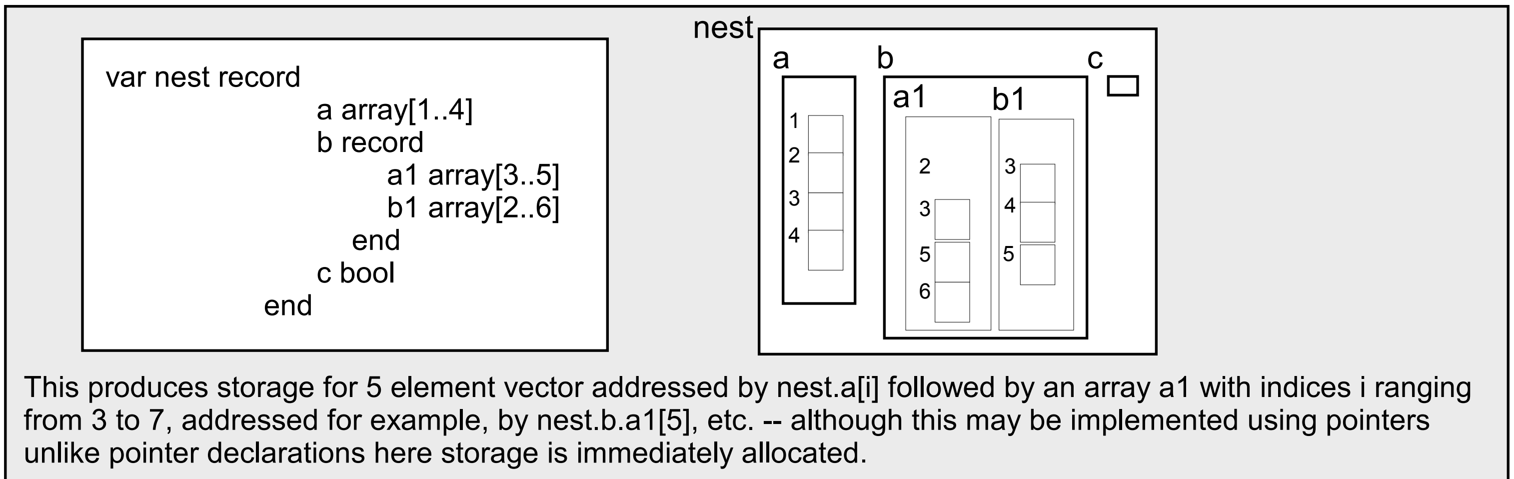
The HEAP and Pointers And Block Structure--Dangling Pointers, Cycles

In a block structured language, storage for a local variable defined within a block disappears when the execution of code in that block is completed--that is the storage devoted to that variable is automatically released for use in other blocks. Therefore, if a pointer p points to that location, then when that storage is released for other uses the pointer will still point to the same location, but there may now be something entirely different there--**p becomes a "dangling pointer"**. In C which does not have as extensive a block structure as that we have discussed here-dangling pointers are possible. For example

int *f(void) { int x =1; return &x; } returns a dangling pointer. because the lifetime of x ends after a return from f.

Modula 2 **avoids this source of dangling pointers by making a pointer to named data illegal.** This is done by guaranteeing that a pointer, which has not been explicitly disconnected from its destination, **can only point to storage which is dynamic, usually kept in the Heap**(in Modula this is created by a **new(p)** command), or to nil. This results from the following limitations imposed on the use of pointers.

1) For most part A pointer p must be declared to point to some type.
i.e, given the declaration:
var p: pointer to array [1..10] of integer
the compiler will allocated static space for the pointer p, but not to the array to which it will point. Space for the array is only allocated when the new(p) command is executed.(there is also a more general allocate command.)



This produces storage for 5 element vector addressed by nest.a[i] followed by an array a1 with indices i ranging from 3 to 7, addressed for example, by nest.b.a1[5], etc. -- although this may be implemented using pointers unlike pointer declarations here storage is immediately allocated.

The following are the only operations allowed on pointers. So a pointer may be declared:

1) **new(p)** creates an instance of the type that p was declared to point to and points p to it. The space is allocated globally in the **Heap area** (like Algol 68), so that exit from a block will not destroy it.
This is the only way a pointer gets to point at any declared type.
2) **p|** is the result of dereferencing p once.
3) **p := nil** makes pointer p point nowhere.
4) **p := q** pointers can be assigned the value of other pointers to the same type
5) The predicates: **p = q and p /= q**is valid.
6) **dispose(p)** frees the area in the Heap to which p points

Again notice that
var p: pointer to array[1:10] of integer and
type a = array[1:10] of integer
type array pointer to a
both give a pointer that can only point to array[1:10] of integer. Neither creates an array for p to point to. This is a different outcome than other nested declarations:

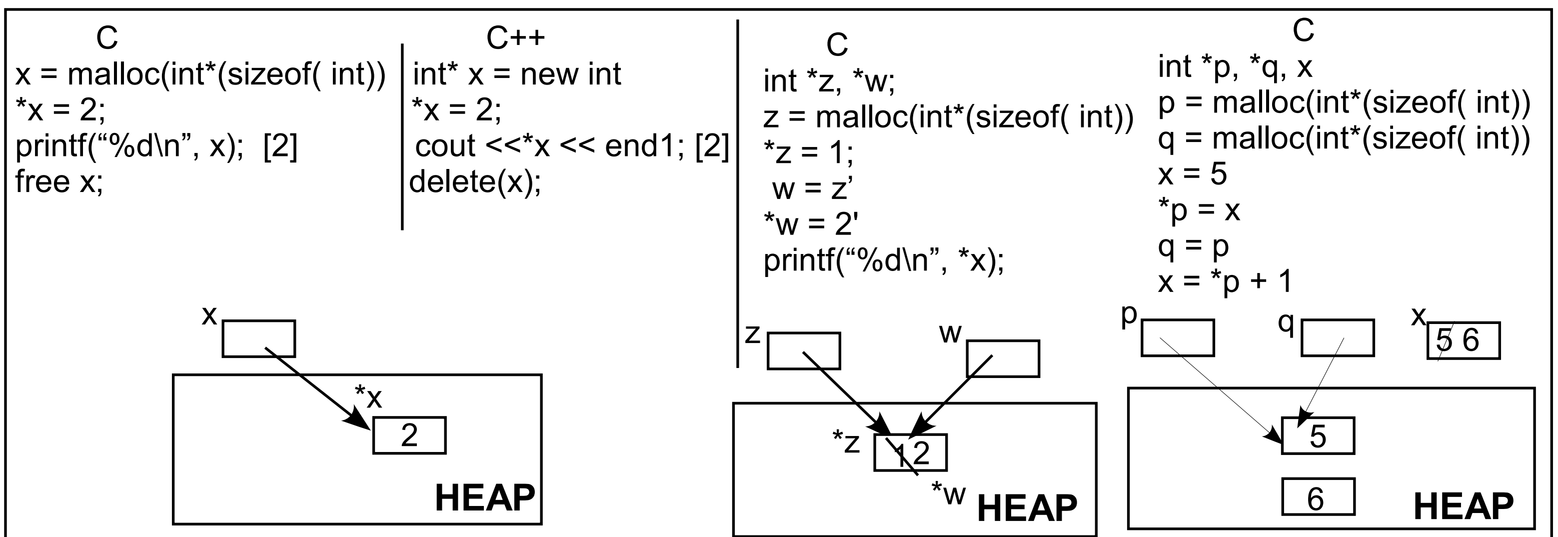
It follows from the constraints on pointer operations and the pointer declaration interpretation that, a pointer, p, either points to nil or a dynamic object (an un-named one created by new) or the command dispose(p) has been executed. This means that only explicit disposal can leave a dangling pointer.

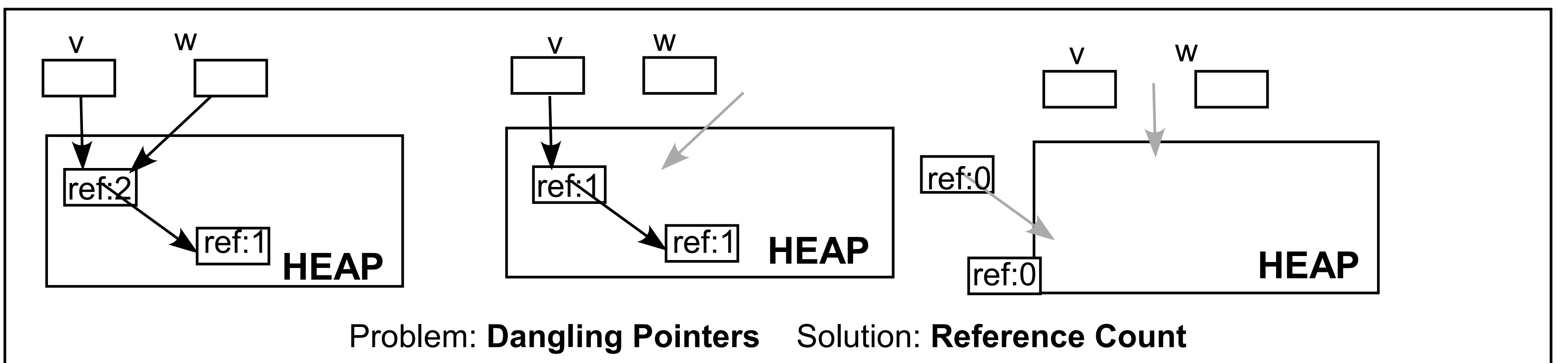
© M C Paull 2008

Note:
If p is a pointer to location x, then a function call **f(**...,p,..) moves p into body of **f** by assigning q, the corresponding formal parameter, to point to x it then may alter the contents of x so that on completion the call has been a reference call to x. So the code in the body of **f** can modify the contents of x, but except for use of return cannot produce a dangling pointer.

Garbage Collection
When dispose(p) is executed: the system can place nil in p, but what of the structure, say S,that p points to? If p is the only pointer to S, S can be returned to the heap (deallocated) and if S contains a pointer, say to S‘, and it is the only pointer to S’ the S‘ too can be deallocated, and so on. In order to determine whether the removal of a pointer to S is sufficient justification for removing S a “reference count“ is kept in S which is incremented by 1 each time a pointer to it is established, and decremented by 1 when a pointer to it is removed

. When S's reference count becomes 0, S can be removed. Even though this technique is used to return dynamic storage to available space there still can be dynamic storage unuseable and unavailable This is because when there is a loop of pointers, and p points to any of the pointer location on that loop, p‘s removal will not cause any reference count to go to 0, even if p is the only such pointer. In the latter case the removal of the pointer in p or p makes the loop inaccesible, but will not reduce the reference count of any member of the loop to 0. Because there can thus be inaccesible dynamic storage “garbage collections” are made when the supply of available dynamic storage becomes small. In garbage collection all of the declaredpointers are surveyed to mark every dynamic storage location accesible by pointers from a declared pointer. Then all unmarked dynamic storage locations (including those in isolated loops are returned to the pool of available dynamic storage (called the heap).





Problem: **Dangling Pointers** Solution: **Reference Count**

1. Explicit Allocation and Explicit De-allocation (C)
2. Explicit Allocation and Automatic De-allocation
3. Automatic Allocation and Automatic De-allocation(Scheme)

```
CONST stacksize=
TYPE element =

MODULE stack;

  IMPORT element,
            stack_size;
  EXPORT push.
             pop;
  TYPE
    stack_index = [1....stack_size]

  VAR
     s  : ARRAY stack_index OF element
     top : stack_index;

  PROCEDURE pop()
  BEGIN
    top := top -1;
    return s(top+1);
  END

  PROCEDURE push(elem :element)
  BEGIN
    s[top] := elem
    top := top +1;
  END

end STACK
```

MODULE
Inside CONTAINS data structures, procedures, variables, types
inside all mutually visible +
IMPORTS also visible
outside only EXPORTS

```
VAR x, y, : element

push(x);

y := pop
```

If declared inside a subroutine the EXPORTS have the same lifetime as local variables declared there.

There is 1 stack for this module anyone can use its push and pop within subroutine in which it was declared.

MODULE AS MANAGER

© M C Paull 2006

Class Stack()

```
{ private:
    char *elements;
    int top;                          Declare Data Structure
    int size;
  public:
    char pop();                       Declare functions
    void push(char)
        Stack(int);                   Constructor  Initialization
       ~Stack(int);                   Destructor
}
```

Private : only known within the Class
Public: Known outside class.
The defines a type and many variables can be declared as that type
It can be initialized - in this case the size of the stack.

```
char Stack::pop()                 Define Functions (could be defined
  { top = top - 1;                where declared above  without Stack::)
     return elements[top+1];
  }
 void Stack::push(char c)
  { top = top + 1;
    elements[top] = c;
  }
     Stack::Stack(int n)
  { size = n;
    elements = new char[size];
    top = 0;
  }
    Stack::~Stack
  { delete elements;}
```

Stack Procedures

```
#include <stdio.>
  main()
  { Stack s(101);                                    Name a stack instance
    s:push(‘ | ’); s:push(‘ &’); s:push(‘ @ ’);    Perform Functions on stack instance
    printf(“%c \n”, s.pop() )
    printf(“%c \n”, s.pop(), s.pop() )
 }
```

If declared inside a subroutine it has the same same lifetime as local variables declared there.

CLASS = MODULE AS TYPE

© M C Paull 2006