UNIT — 4

(1) Storage Organisation
- Activation Tree
- Activation Record.

(2) Allocation Strategies
- Static
- Stack
- Heap

(3) Parameter passing
- call by value
- call by reference
- Call by name

(4) Symbol Table - Related data structure.

(5) Dynamic Allocation of Storages.

(1) ~~storage organisation of run time memory.~~

for allocating storage space for the life time of a variable, there are three possibility

(1) Static Storage
(2) Dynamic Storage
(3) Global Storage.

(1) Static storage → If the life time of a variable is the life time of the program and space for its value, once allocated can not be released later. Such storage is refered to as static storage.

(2) Dynamic Storage :— If the life time of variable is a particular block, function or procedure, in which variable is

declared the storage allocated to variable may be deallocated when the execution of block, function, or procedure is over. Such storage is referred to as dynamic storage

(3) Global storage :- Storage may be allocated for values, not necessarily associated with variables, at a particular point in the execution of a program not necessary corresponding to the start of a block or the entry to a procedure. The storage is then ~~release~~ required from that point on until it is released either by language mechanism or through simply being no longer reachable from program. However, the movement of these release, this space is known at runtime Such a storage is referred to as global storage.

Source language issues :- Implementation mechanism may be different for different languages

(I) Procedure - ~~Activation Tree~~
(II) Activation Tree
(III) Control Stack
(IV) The Scope of declaration
(V) Binding of names.
(VI) life time of variables.

(1) Procedures :- A procedure definition is a declaration that associates an identifier with statements. The identifier is procedure name & statement is procedure body.

Example

```
procedure readarray;
  var i: integer
  begin
    for i = 1 to 9 do read(a[i])
  end;
```

Procedure body

Procedure name

- Procedure that returns values is called function

- Some of the identifiers appearing within a procedure definition are formal parameters (or formal arguments, dummy arguments or formals) of the procedure.

- Arguments, Actual parameters are substituted for the formal parameters when the procedure is called. There are various methods of parameter passing, depends on language implementation.

(2) Activation Tree :- During the execution of a program following assumptions are made about the flow of control among procedures.

(i) Control flows Sequentially : that is the activation of a program consist of a sequence of steps with control being at some specific point in the program at each step.

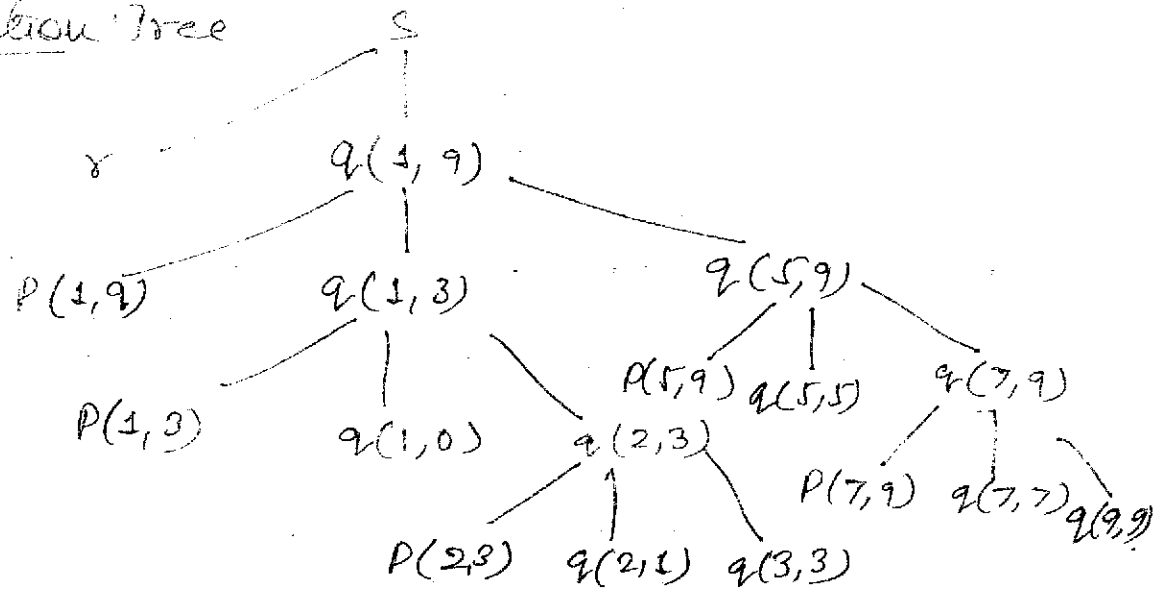
(ii) Each execution of a procedure starts at the beginning of the procedure body & eventually returns control to the point immediately following the place where the procedure was called. This means the flow of control between procedure can be depicted using Tree.

~~Each execution of~~ a procedure body is referred to as activation of the procedure.

We use a Tree to depict the way control enters and leaves activation. Called Activation Tree. Main activation tree.

- each node represents an activation of procedure
- the root represents the activation of the main program
- the node for a is the parent of the node b if and only if control flows from activation a to b.
- the node ~~for~~ a is to the left of the node for b if and only if the life time of a occurs before the life time of b.

Activation Tree



(3) Control Stack : The flow of control in a program corresponds to the depth first traversal of activation tree that starts at the root. Visits a node before its children and recursively visits children at each node in a left-to-right order. We use control stack to keep track of live procedure activations. The idea is to push the node ~~for~~ an activation onto the control stack as the activation begins and to pop the node when the activation ends. Then the contents of the control stack are related to paths to the root of the activation tree.

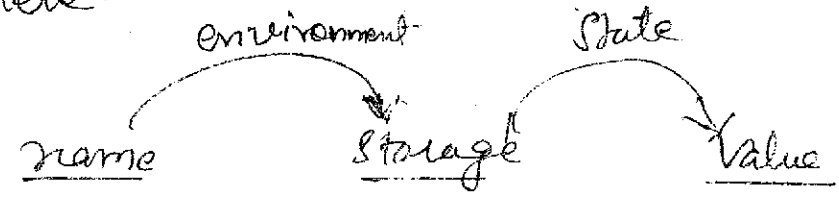
(4) Scope :- The scope of a declaration is that portion of a program where that declaration applies. Variables may be declared explicitly or implicitly in some languages. Scope rules for each language determines how to go from names to declarations.

The ~~use of~~ usage of a name in a procedure is local if it is within the scope of a declaration within that procedure; otherwise the usage is non-local. According to the scope of variable in a particular language the storage management is implemented.

(5) Binding of names :— Each use of a variable name must be associated with a declaration. This is generally done ~~in~~ a Symbol table. In most compiled languages it happens at compile time.

The informal term "data object" corresponds to a storage location that can hold values.

The term environment refers to a function that maps a name to a storage location and the term state refers to a function that maps a storage location to the value held there.



(6) Life time of variable :- The life time of variable may be entire program, particular block or function and storage may be allocated for values, so storage management will be static, dynamic storage and global storage respectively.

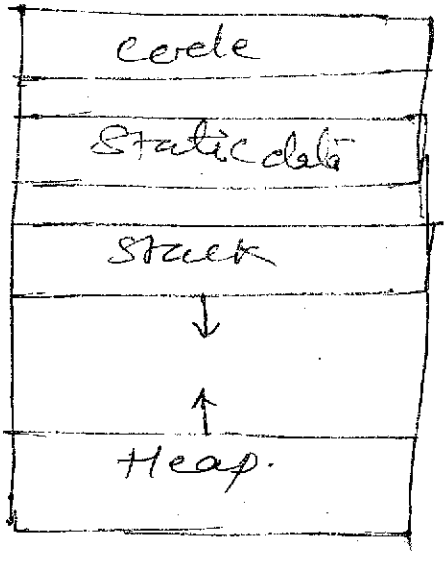
(2) Storage Organisation:

Consist of
1. Sub division of run time memory
2. Activation Records.
3. Compile time layout of local data.

(1) Sub division of run time memory :-
We assume that the compiler obtains a block of storage for the compiled program to run in. Run time memory needs to be subdivided to hold the different components of an executing program.

- Generated executable code
- Static data objects
- A Structure to keep track of procedure activation - This can generally be considered to consist of two components
  - The Stack - for objects whose life times do not exceed the life time of the activation.
  - The heap - for the objects whose life time exceeds that of the activation.

A Typical sub division might be.



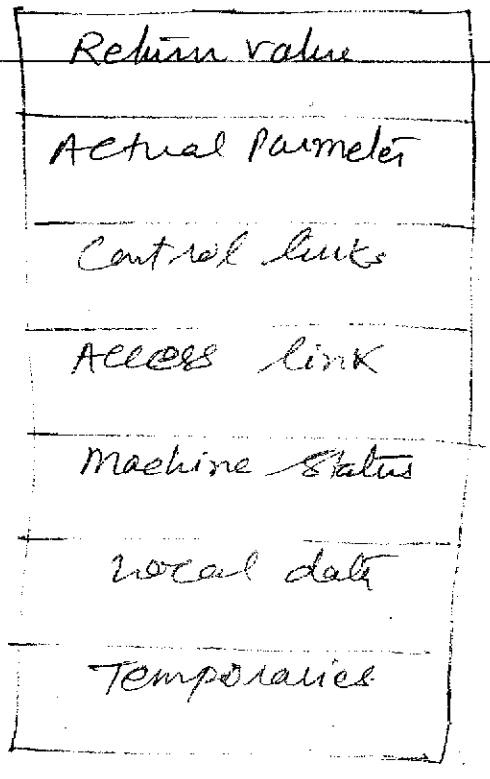
— The size of the generated target code is fixed at compile time so the compiler can place it in a statically determined area in low end of memory.

— The size of the stack and heap can change as the program executes. Data objects whose life time are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

(11) Activation Records: The information needed by a single execution or a single activation of a procedure is managed using a contiguous block of storage called an "activation record" or 'frame' consisting of the collection of fields.

The activation records contains the following information.

1. Temporary values, used during expression evaluation.
2. Local data of a procedure.
3. Saved machine status information (PC, registers, return address)
4. (Optional) access link, for access to non-local names.
5. The actual Parameters.
6. The return value, used by called procedure to return value of calling procedure.
7. (Optional) control link, points to the activation record of the caller



(11) Compile time layout of local data :- The compiler must determine where, ~~what~~ within an activation record, the memory location(s) for an object are, so that during the code generation it can refere to the correct address.

The amount of storage needed for an object is determined from its type. The field for local data in the activation record is laid out as declarations in the procedure are processed (variable length data are kept outside this field).

Storage layout must confirm to alignment requirements of the target machine.

Example elementry data type char, int or real can be stored in an integral number of bytes.

(2) Storage allocation strategies :-

(1) Static allocation :- layout storage for all data objects at compile time.

(2) Stack allocation :- manages run-time storage as a stack.

(3) Heap Allocation :- allocates and deallocate storage as needed at run time from data area known as a heap.

(1) Static Allocation :-

- Simplest form of allocation.
- Storage space once allocated was never released, so a very simple storage allocation mode that allocate storage, as required from one end of the available space towards the other, was adequate.
- early programming language such as Fortran had static storage, the amount of which was known at compile time.

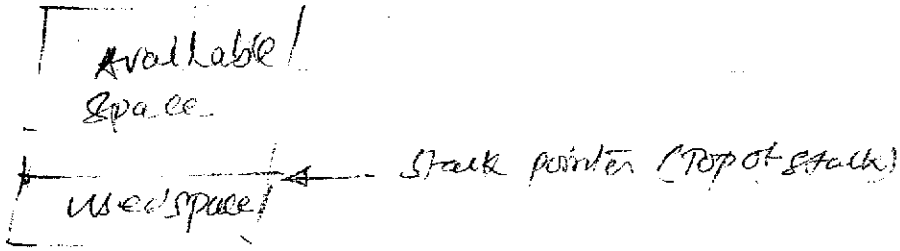
- Static storage allocation is efficient because no time or space is expended for storage mangement during execution.

Limitations :- (1) It is incompatible with recursive subprogram calls with data structure whose size is dependent on computed or input data, and with many other desirable language feature

(2) Data structure can not be created dynamically

(3) The size of the data object and constraints on its position in memory must be known at compile time.

(4) Stack allocation :-
- It is the simplest run-time storage management technique.



- Storage allocation begins at one end. Storage must be freed in reverse order of allocation so block allocated space recently must free the space first.

- Only one stack pointer is all that is needed to control storage management.
- The ~~storage~~ stack pointer always points to the top of the stack, the next available word of free storage in the stack block.
- Compaction occurs automatically as part of freeing storage. Freeing a block of storage automatically recovers the freed storage & makes it available for reuse.
- We know that most of the time programs and data elements requiring storage are tied to sub program activations.

when a Sub Prgram is called, per call a new activation records is created on the top of the Stack, and termination causes its deletion from the stack.

Handling Procedures Calls & Returns:

Stack allocation is used for data in recursive procedures that may have multiple bindings. Each procedure call creates a new activation record on the control stack in which the actual parameters, the local variables, the temporaries & the returned value of that activation can be ~~used~~ stored. The activation record remains on the stack until the procedure activation returns to its caller. Stack-allocated data items are referenced using displacements from a register, top, pointing to the top of the control stack.

A Call sequence is a sequence of m/c instructions that is executed every time a procedure is called. It works as follows.

- Allocates an activation record for the called procedure.
- Loads actual parameters.
- Saves machine state (return address etc)
- Transfer control to callee.

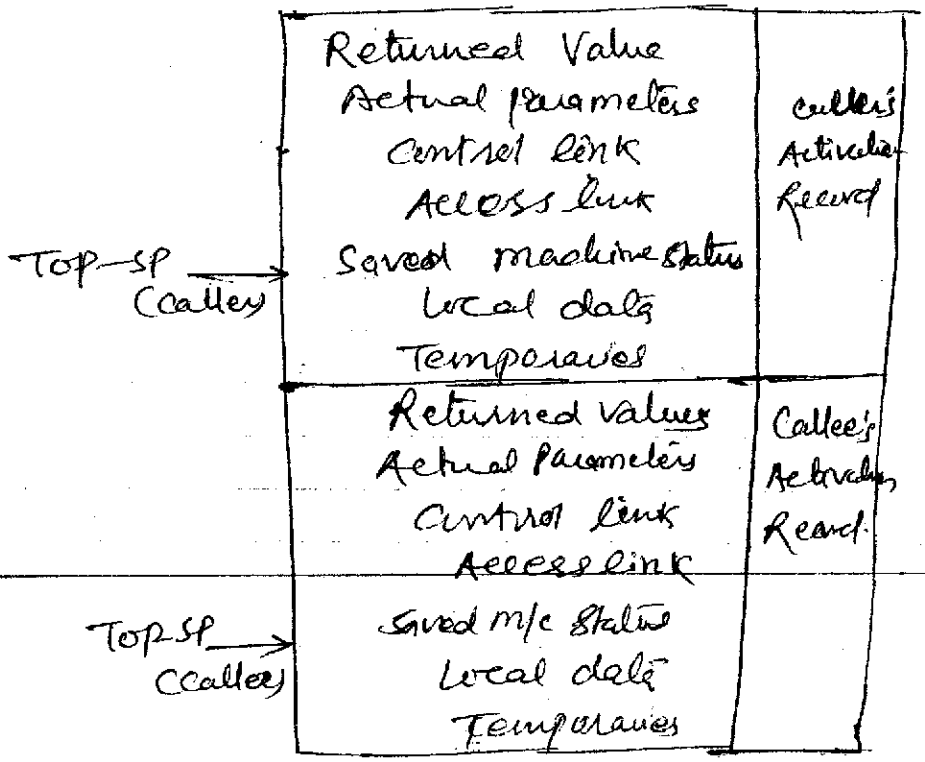
A return sequence is a sequence of machine instruction that is executed every time a procedure returns control to its caller.

It does following.

- Deallocates activation records of the called procedure.

- Set up returns value (if any)
- Restores machine state (stack pointer, etc)

Activation records and calling sequences differ from machine to machine. The calling sequences are often divided up between the caller and procedure being called. (the callee)



The following Call Sequence. assumes there is a register (top_sp) holding a pointer to the start of local data in the current activation record.

(1) The caller evaluates the actual parameters and insert them into the appropriate place of the the callee's activation record.

(2) The caller ~~returns~~ stores a return address and the value of its top-sp register into saved machine status of the callee's record.

(3) The top sp register is set to point to the start of the callee's local data and control is sent to the callees code.
(4) The callee saves other register values and status information.
(5) The callee initialise its local data and starts execution.

A possible return sequence is.
(1) The callee places a returned value at the start of its activation record.
(2) Using the saved machine status in its record, the callee ~~sto~~ restores top_sp and other registers and branches to the return address in the caller's code.
(3) The Caller resumes execution: The location of the returned value is a known displacement from top_sp.

(3) The Heap Storage Management :-

We already know that, the heap is used for the storage of the values which may require to be accessible from the time the storage is allocated until the program terminates.

The major problem arises in these type of allocation is that, how to reallocate or reuse the allocated space.

- The heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

- for efficiency reason use small activation records or records of predictable size as
  (i) ~~for each size of interest, keep a linked list of free blocks of that size~~
  (ii) If possible, fill a request of size s with a block of size s', where s' is the smallest size greater than or equal to s. When the block is eventually deallocated, it is returned to the linked list it came from.
  (iii) for the large block of storage use the heap manager.

(9)

Parameter passing :⇒

When one procedure called another, the usual method of communication between them is through non-local names and through parameters of the called procedure.

There are several methods for associating actual & formal parameter depends on language implementation.

1. Call-by-Value
2. Call-by-Reference
3. Copy-Restore
4. call-by name

* l-value - refers to the storage represented by Expression
* r-value - Value contained in the storage.

(1) Call-by-value - Usually C & Pascal use Call by Value - the r-values of the actual parameters are passed to the called procedure

(A) A formal parameter is assigned a storage location in activation record of the called procedure.

(B) The caller evaluates the actual parameters and place their r-values in the storage locations of the formal parameters.

If the called ~~param~~ procedure changes a formal parameter the change only occurs in the activation record of the called procedure the change is lost when the record is deallocated. So the actual parameter in the caller's records is not changed.

(2) Call-by-reference :- Call by reference (or call by address, or call by location) passes l-values to the called procedure.

(1) If an actual parameter is a name

or an expression having an l-value, then that l-value is passed to the called procedure.

(ii) If the actual parameter is an expression like a+b or 2 which has no l-values, then the expression is evaluated in a new location and the address of that location is passed to the called procedure.

(3) Copy-Restore:- Copy restore is a hybrid of call-by-value and call-by-reference

(i) The actual parameters are evaluated before the call & their r-values are passed to the called procedure as in call-by-value But the caller also remembers the l-values of those actuals that have l-values.

(ii) When control returns to the caller it, copies back the r-values of the formal parameters into the l-values of the actuals usually, copy-restore has the same effect as call-by-reference. It could have a different effect when the called procedure refers to one or more of the actual parameters as a non-local.

(4) call-by-name :- Call by name used by ALGOL.

(i) The procedure is treated as if it were macro. that is its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals.

(ii) Local names in the procedure are kept distinct from names in the caller.

(iii) Actual parameters are surrounded by parenthesis if necessary to preserve their integrity.

(5) Dynamic Storage Allocation

The Dynamic allocation of storage for data is done under the program control. Storage for such data is usually taken from heap. The allocated data is often retained until it is explicitly deallocated. Allocation can be

(i) Implicit
(ii) Explicit.

Implicit allocation occurs when evaluation of an expression result in storage being obtained to hold the value of the expression. Example - Lisp

Explicit allocation in pascal is performed using procedure "new" & Deallocated by calling 'dispose'.

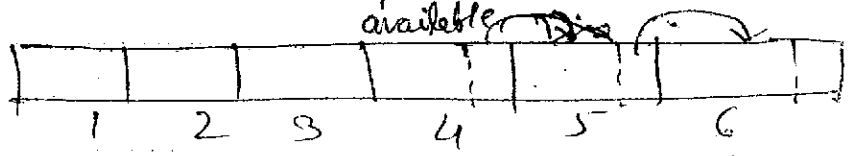
- Dynamically allocated storage can become unreachable. Storage that a program allocates but cannot refer to it is called garbage.

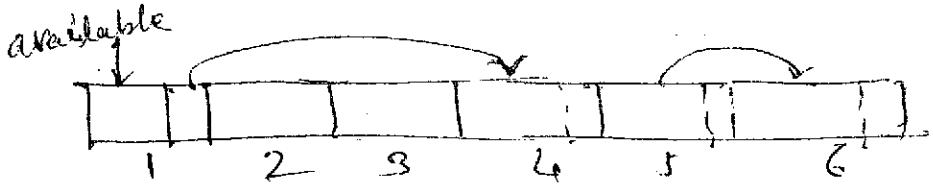
- A dangling reference occurs with explicit deallocation when storage that has been deallocated is referred to.

Techniques to implement Dynamic allocation of storage

(i) Explicit allocation of fixed sized blocks
(ii) Explicit allocation of variable sized blocks.
(iii) Implicit Deallocation — Reference counts
Marking techniques.

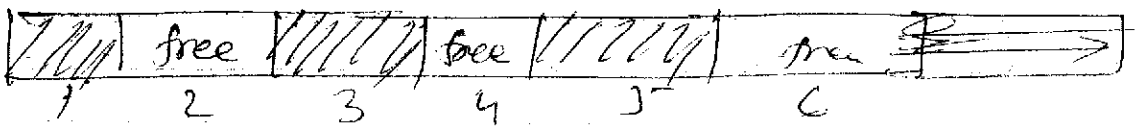
① Explicite Allocation of fixed Sized blocks
- The Simples form of dynamic allocation. used a block of fixied size.
- By linking the blocks in a list we can perform allocation & deallocation.
- managed by pointer 'available'
- Compiler routines manages blocks. Hence there is no space overheads.





---

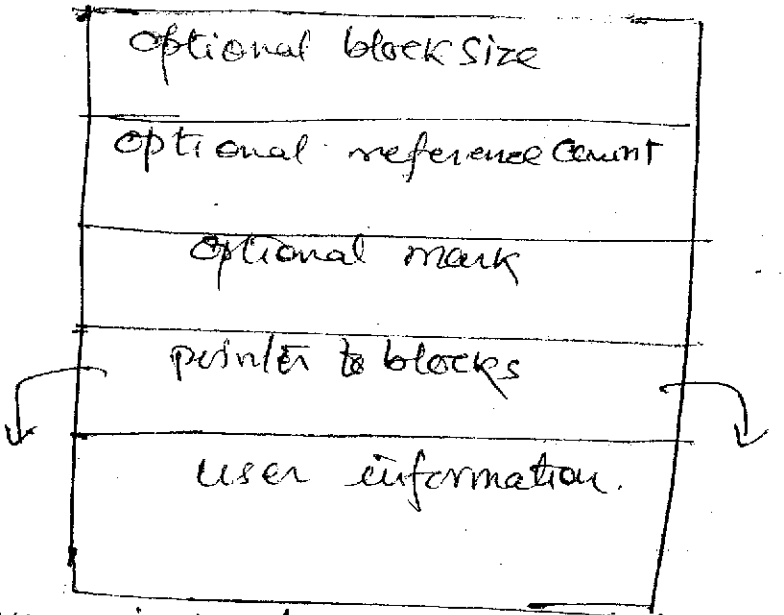
② Explicit allocation of variable-sized blocks.
- Use fragmentation to allocate & deallocate the blocks.
- heap may consist of alternate blocks that are free & in use.



allocate five blocks & deallocates (free) 2nd & 4th block.

- first fit method may be used to allocate Variable size block.
- During deallocation we see if it is next to a free block & the combined with a free block to create larger free block.

(11) Implicit deallocation : requires co-operation between the user program and the run time package . The operation is implemented by fixing the format of storage blocks



Problem with implicite representation.

(1) ~~for~~ Recognising block boundaries - If the size of block is fixed, then position information can be used.

(11) Recognising if the block is in use - can be managed by pointer or after following a sequence of pointers. So Compiler needs to know the position in the storage of all pointers.

pointers are kept in a fixed position in block.

for Deallocation - Two approaches are used.

(1) Reference count

(11) Marking techniques.

(i) Reference Count :- We keep track of the number of blocks that point directly to the present block. If this count ever drops to 0 then the block can be deallocated because it cannot be referred to.

(ii) Marking Techniques :- An alternative approach is to suspend temporarily execution of the user program & use a frozen pointers to determine which blocks are in use. This approach requires all the pointers into the heap to be known.

A Dhole