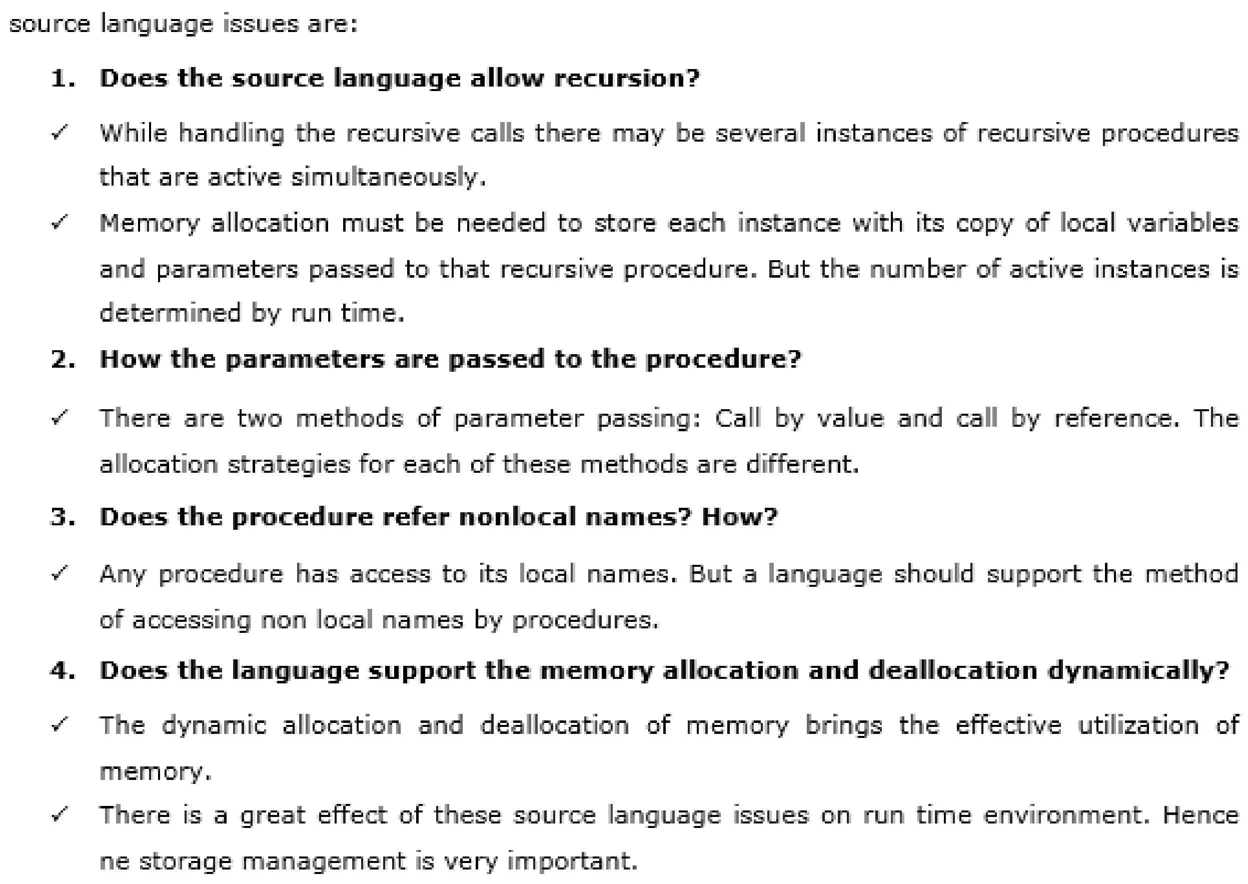
Unit 5: Run-Time Environment

Source language issues, storage organization and location strategies, parameter passing, symbol table organization and generation, dynamic storage allocation.



source language issues are:

1. **Does the source language allow recursion?**

✓ While handling the recursive calls there may be several instances of recursive procedures that are active simultaneously.

✓ Memory allocation must be needed to store each instance with its copy of local variables and parameters passed to that recursive procedure. But the number of active instances is determined by run time.

2. **How the parameters are passed to the procedure?**

✓ There are two methods of parameter passing: Call by value and call by reference. The allocation strategies for each of these methods are different.

3. **Does the procedure refer nonlocal names? How?**

✓ Any procedure has access to its local names. But a language should support the method of accessing non local names by procedures.

4. **Does the language support the memory allocation and deallocation dynamically?**

✓ The dynamic allocation and deallocation of memory brings the effective utilization of memory.

✓ There is a great effect of these source language issues on run time environment. Hence ne storage management is very important.

Each execution of a procedure is called as the **activation of the procedure**.

Procedure activations:

1) Execution of a procedure starts at beginning of the procedure body.
2) When a procedure is completed, it returns the controls to the point immediately after the place where that procedure is called.
3) Each execution of a procedure is called its activation.
4) A lifetime of an activation of a procedure is the sequence of the steps between the first and the last step in the execution of the procedure. (including the other procedure called by that procedure).
5) If a and b are procedure activation then their life is either non-overlapping or are nested.
6) If a procedure is recursive, a new activation can begin before on earliest activation of the same procedure has ended.

Activation Tree:

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedures identifiers and the sequence of instructions inside it is the body of the procedure.

The execution of a procedure is called its **activation**. Activation contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

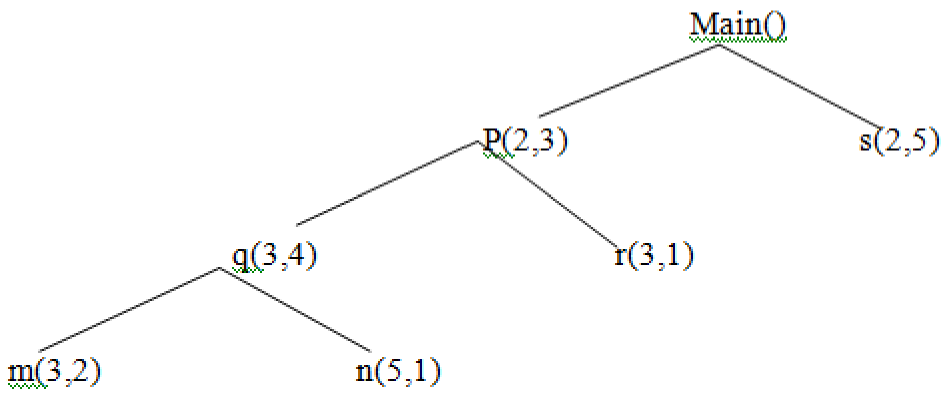
Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter, etc., before the procedure, is called.
Control Link	Stores the address of the activation record of the caller(who called the procedure) procedure.
Access Link	Stores the information of data that is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

In an activation tree:

1) Each node represents the activation of the procedure.
2) Each root represents the activation of the main program.
3) The node ‘a’ is a parent of the node ‘b’ iff the control flows from ‘a’ to ‘b’.

4) The node ‘a’ is left to the node ‘b’ iff the lifetime of ‘a’ occurs before the lifetime of ‘b’.

```
        Main()
        {
        Procedure p(2,3)
          {
          Procedure q(3,4)
           {
             Procedure m(3,2)
              {
                ……
               }
              Procedure n(5,1)
              {
               …..
               }
              }
            Procedure r(3,1)
            {
             --------
            }
         }
      Procedure s(2,5)
      {
       --------
      }
     }
```



Activation tree for procedures

Control stack:

Whenever a procedure is executed, its activation record is stored on the stack. It is also known as the control stack. When a procedure calls another procedure, the execution of the caller is hold until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows sequentially and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:

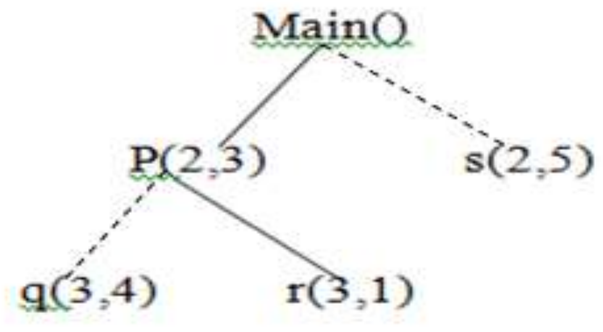
- Start at the root.
- Visit a node before its children and
- Recursively visit children at each node n left to right order.

A stack (called control stack) can be used to keep track of live procedure activations.

1) An activation record is pushed onto the control stack as the activations.

2) The activation record is popped when that activation ends.

When node 'n' is at the top of the control stack, the stack contains the nodes along the path from 'n' to the root.



When control enters the activation represented r(3,1), activation with label main P(2,3) has executed to compilation, so the figure contains dashed lines nodes. The solid lines mark the path from r(3,1) to root.

Variable scope:

- The same variable name can be used in the different parts of the program.
- The scope rules of the language determine which declarations of name apply when the name appears in the program.
- An occurrence of a variable (a name) is:
    - Local: - If that occurrence is in the same procedure in which that name is declared.
    - Non-local: - It is declared outside of that procedure.

```
Procedure  P()
{
        b integer;
        Procedure     q()
        {
                a integer;
                a=10;
                b=20;
        }
}
```

Where variable name ‘a’ is local to procedure q but the variable name ‘b’ is non-local (global) to procedure q.

Storage Organization:

Runtime environment manages runtime memory requirements for the following entities:

- **Code**: It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

- **Procedures**: Their text part is static but they are called randomly. That is why; stack storage is used to manage procedure calls and activations.

- **Variables**: Variables are known at the runtime only unless they are global or local. Heap memory allocation is used for managing allocation and de-allocation of memory for variables at runtime.

Suppose that the compiler contains a block of storage from the O.S. for the complied program to run in the runtime storage may be subdivided into

1) The generated target code.

2) Data object

3) Counter parts of the control stack to keep track of procedure activation.

A separate area of run time memory called a heap holds all other information.

Activation Records:

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an **activation record or frame.**

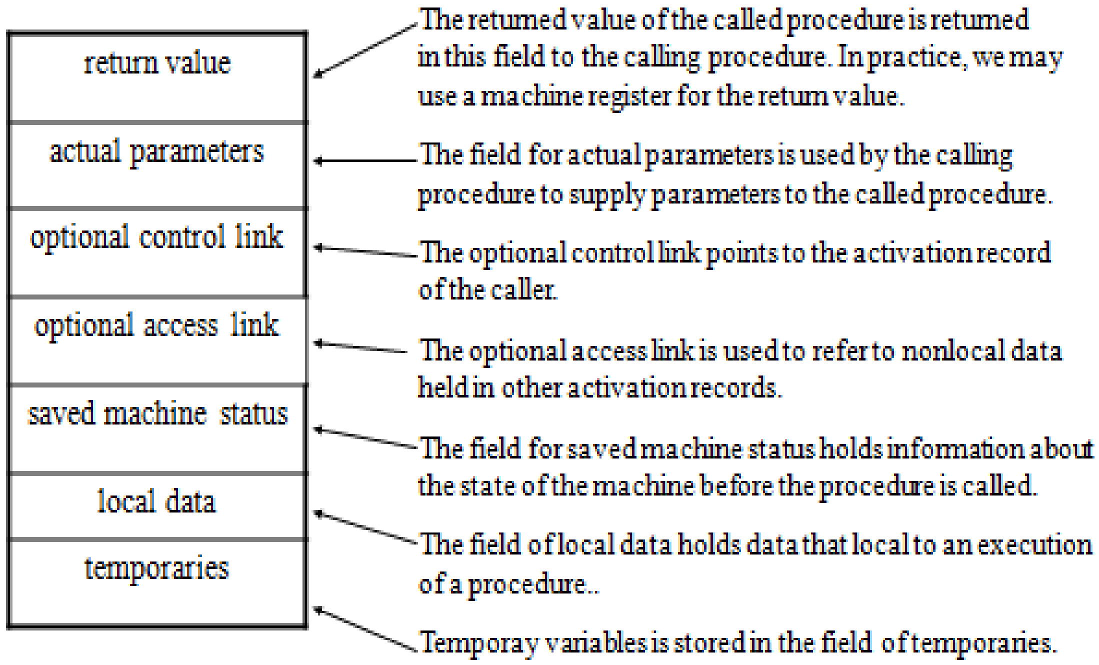

return value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries
The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.
The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.
The optional control link points to the activation record of the caller.
The optional access link is used to refer to nonlocal data held in other activation records.
The field for saved machine status holds information about the state of the machine before the procedure is called.
The field of local data holds data that local to an execution of a procedure..
Temporay variables is stored in the field of temporaries.

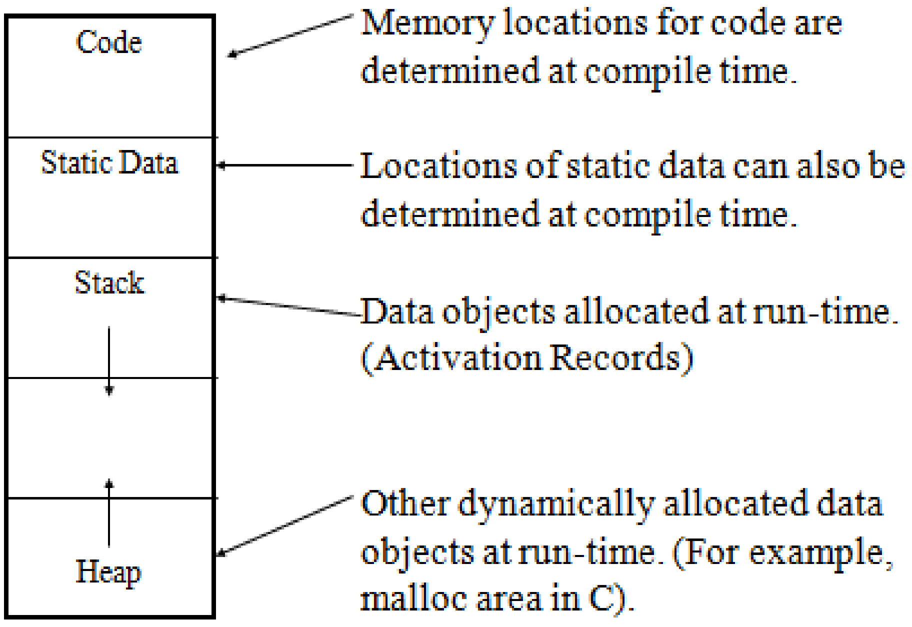
Fig. Activation Records

The purpose of the field of activation records is as follows:

1) Temporary values such as those arising in the evaluation of an expression are stored in temporaries.
2) The local data holds data that is local to an execution of a procedure.
3) The saved machine status hoods information about the state of the machine before the procedure called.
4) The optional access link is used to refer to the non-local data held in other activation records.
5) The optional control link points for the activation records of the column.
6) The field actual parameter is used by the calling procedure to supply the parameter to called procedure.
7) The field return value is used by the called procedure to return value to a calling procedure.

Storage Allocation Strategies:

The storage allocation strategy is used in each of the three data areas namely static data area, stack, and heap.


Code
Static Data
Stack
Heap
Memory locations for code are determined at compile time.
Locations of static data can also be determined at compile time.
Data objects allocated at run-time. (Activation Records)
Other dynamically allocated data objects at run-time. (For example, malloc area in C).

Typical sub-division of code and data

1) Static Allocation:

In this allocation scheme, the compilation data is bound to a **fixed location** in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, a runtime support package for memory allocation and de-allocation is not required. The compiler determines the amount of space to store variables.

Limitation:

1) The size of data objects and constraints on its position in memory must be known at a compile-time.
2) Recursive procedure is restricted because all activation of procedure uses the same bindings for local names.
3) Data structures cannot create dynamically.

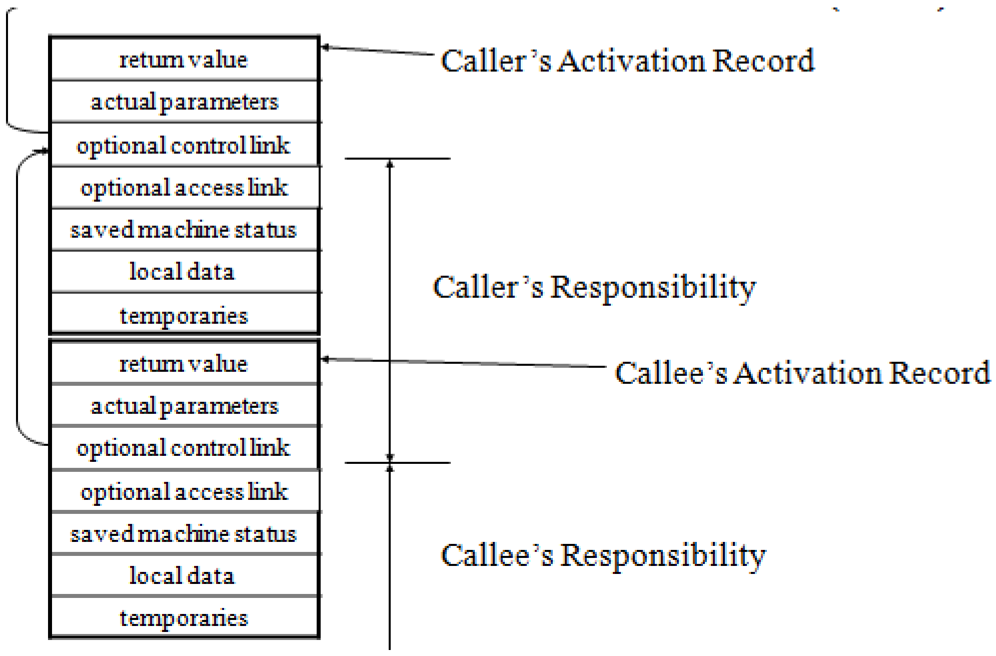
2) Stack Allocation:

It is based on the idea of a control stack. The locals are bounded to new storage in each activation.

The values of locals are deleted when the activation ends that are value lost because the procedure is popped.

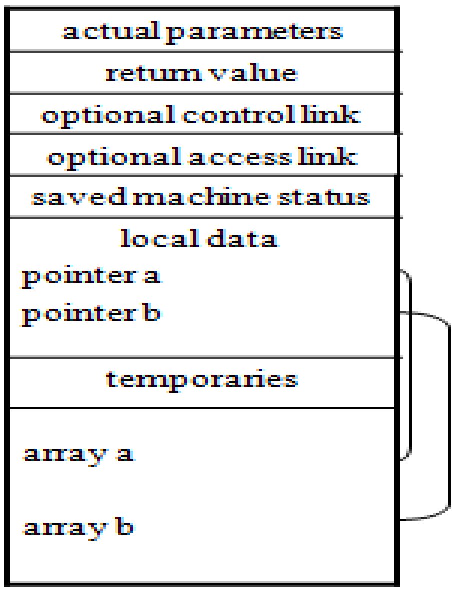
Calling Sequence:

The call sequence allocates an activation record and enters information into its fields. Each call has its own actual parameters the callers usually evaluate actual parameters and communicate them to the activation record of the called.



Variable-length data:

Variable-length data is allocated after temporizing and there is a link to form local data to that array.



Dangling Reference:

Whenever storage has been deallocated the problem of dangling references may occur. When there is a reference of storage that has been de-allocated logical uses dandling reference

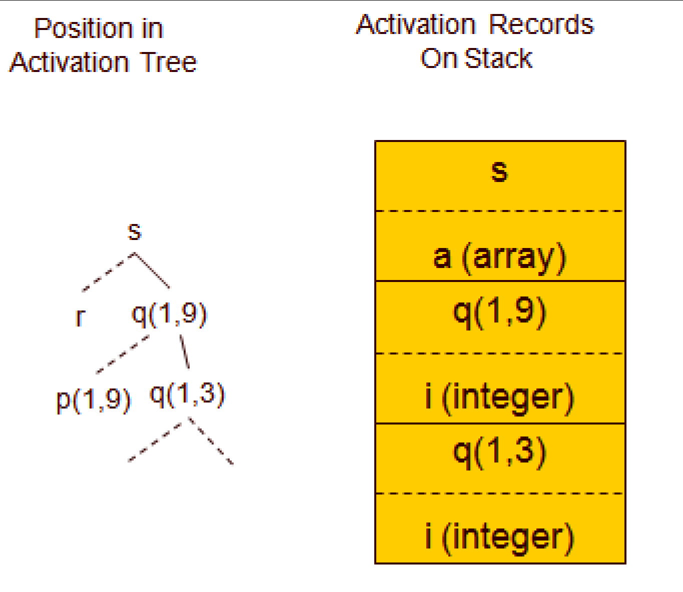
i.e. value of dislocated storage is undefined. Bugs can appear in the program with dangling references.

main ()

```
{
 int *p;
 p = dangle();
 }
int *dangle()
{
 int i=2;
 return &i;
}
```

The local variable I only exits in dangle(). When the procedure completes its execution i do not send the address. Pointer p has a dangling reference.

e.g. Example of stack allocation.

 Position in Activation Tree Activation Records On Stack s r q(1,9) p(1,9) q(1,3) s a (array) q(1,9) i (integer) q(1,3) i (integer)	Frame for S q is activated q(1,9) is popped and q(1,3) is pushed on the stack

3) Heap allocation:

The stack allocation strategy cannot be used if:

1) The value of local names must be retained when activation ends.

2) A called allocation outlives the caller.

In each case, the de-allocation of activation records need not occur in a last-in-first-out manner, so storage can be not organized as a stack. The subsystem that allocates and de-allocates space within the heap.

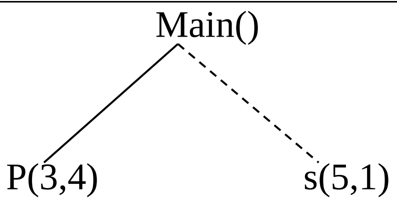
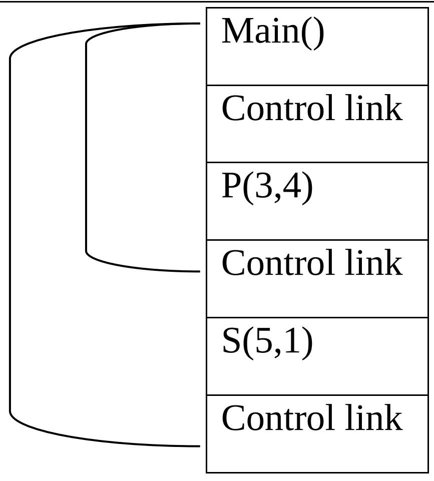
Free or delete:

Garbage collection is the process of finding space within a heap that are no longer used by the program and therefore reallocated to other data item. In java it de-allocates memory.

1) Allocation: - Program request memory = memory manager produces requested memory size, if satisfied an allocation memory available then it increases memory for virtual memory of O.S.
2) De-allocation:- The memory manager returns de-allocation space to the pool of free space. So it can reuse in another data object.

When a block is deallocated, it is returned to the linked list. For a large block of storage use heap manager. This result approach is the fast allocation and deallocation of a small amount of storage.

Procedure calls and their activations are managed by stack memory allocation. It works in the last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Position in the activation tree	**Activation records in the Heap**	**Remarks**
	 Main() Control link P(3,4) Control link S(5,1) Control link	Returned activation records

In one pass compile, information in symbol table about scope consisting a procedure body is not needed at compile time after the procedure body is processed. It may be needed at run time.

Pointer points to the recently create entry in the list.

For finding its entry, a circular linked list is used. We can use a stack to keep a track of the list containing entries to be deleted. The marker is placed in the stack when we finish the procedure the list number can be popped from the stack until the marker for the procedure is reached.

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except for statically allocated memory areas, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.

As shown in the figure typical subdivision of code and data above, the code part is allocated a fixed amount of memory. Stack and heap memory is arranged at the boundaries of the total memory allocated to the program. Both shrink and grow against each other.

Parameter Passing:

For communication with procedures, we can use parameter passing techniques. The values of the variables from a calling procedure are transferred to the called procedure. Some basic terminologies are used to pass the values in a program are:

1) r-value:

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

2) l-value:

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears on the left-hand side of an assignment operator.

In parameter transmission, commonly used terms are actual and formal parameters.

1) Formal Parameters: A formal parameter is a particular kind of local data object within a subprogram. It will be supplied when the procedure is called.
2) Actual Parameters: An actual parameter is an object that is shared with the caller program. The parameters those are used in the function definition are called actual parameters.

There are main three common methods of passing parameters as given as below:

1) Call by value (Pass by value)
2) Call by reference (pass by reference)
3) Pass by Copy- restore
4) Call by name

1. Call by value:

In this method, the actual parameter is evaluated and the r-value of the actual parameter is passed to the formal parameter.

```
#include <stdio.h>
#include <conio.h>
int fact (int );
void main()
{
int n,r;
printf("Enter the value");
scanf("%d",&n);
r=fact(n);   // n is actual parameters
printf("factorial = %d",r);
}
int fact(int a)
{
int f=1,i;
if(a==0)
{
return(1);
}
else
{
for(i=1;i<=a;i++)
{
f=f*i;
}
return(f);
}
}
```

2. Call by reference:

   In the call-by-reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location using a pointer) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

```
#include <stdio.h>
#include <conio.h>
int *fact (int* );
void main()
{
int n,*r;
printf("Enter the value");
scanf("%d",&n);
r=fact(&n);
printf("factorial =%d ",*r);
}
int *fact(int *a)
{
int *f=1,i;
if(*a==0)
{
return(1);
}
else
{
for(i=1;i<=a;i++)
{
*f=*f*(int *)i;
}
return(*f);
}
}
```

3. Pass by Copy-restore:

This parameter passing mechanism works similar to ‘pass-by-reference’ except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
   y = 10;
   copy_restore(y); //l-value of y is passed
   printf y; //prints 99
}
copy_restore(int x)
{
   x = 99; // y still has value 10 (unaffected)
   y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like a call by reference.

4. Call by name:-

   Languages like Algol provide a new kind of parameter passing mechanism that works like a preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. The actual parameter is substituted for each occurrence of the formal parameter in the called subprogram (like micro).

```
#define MAX(x,y)  (((x)>(y))? ((x) :( y)))
#define MIN(x,y)  (((x)<(y))? ((x) :( y)))
void main()
{
 int m,n;
printf("Enter any two values=");
scanf("%d%d",&m,&n);
printf("Maximum number=%d",MAX(m,n));
printf("Minimum number=%d",MIN(m,n));
}
```

Symbol Table:

The symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. The symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.

- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table that can be either linear or a hash table. It maintains an entry for each name in the following format:

```
<symbol name,  type,  attribute>
```

For example, if a symbol table has to store information about the following variable declaration:

```
static int interest;
```

then it should store the entry such as:

```
<interest, int, static>
```

The attribute clause contains the entries related to the name.

Implementation:

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations:

A symbol table, either linear or hash, should provide the following operations.

1) insert():

This operation is more frequently used by the analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type of the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

e.g.: int a;

should be processed by the compiler as:

```
insert(a, int);
```

2) lookup():

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol is declared multiple times.

The format of the lookup() function varies according to the programming language. The basic format should match the following:

```
lookup(symbol)
```

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

3) delete(): - Remove the most recently created entry deleted entries must be preserved they are just removed from the active symbol.

```
delete(symbol)
```

Scope Management:

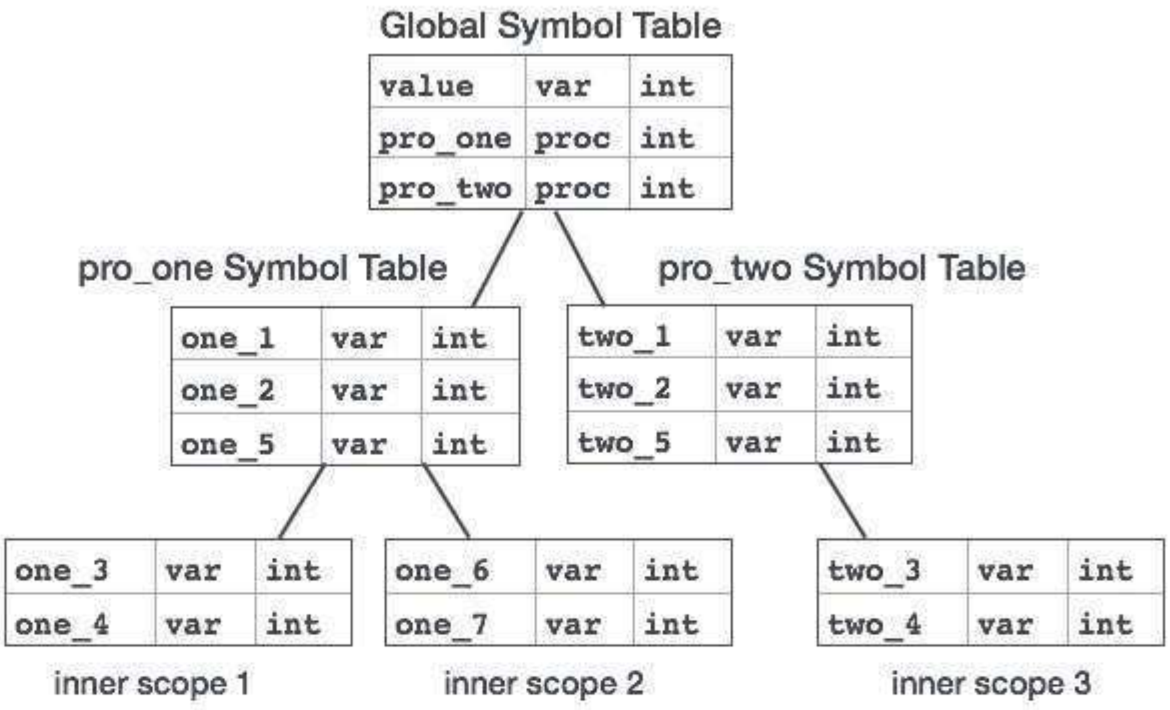
A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in a hierarchical structure as shown in the example below:

```
. . .
int value=10;

void pro_one()
 {
 int one_1;
 int one_2;

   {          \
   int one_3;     |_  inner scope 1
   int one_4;     |
   }          /

 int one_5;

   {          \
   int one_6;     |_  inner scope 2
   int one_7;     |
   }          /
 }

void pro_two()
 {
 int two_1;
```

```
  int two_2;


    {            \
    int two_3;     |_  inner scope 3
    int two_4;     |
    }            /


  int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first, a symbol will be searched in the current scope, i.e. current symbol table.

- if a name is found, then the search is completed, else it will be searched in the parent symbol table until the name is found.

Dynamic Storage Allocation Techniques

There are two techniques used in dynamic memory allocation and those are -

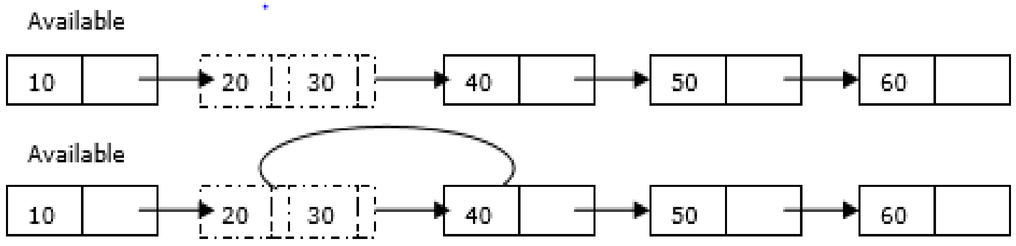
- ✓ Explicit allocation
- ✓ Implicit allocation

1. Explicit Allocation

- ✓ The explicit allocation can be done for fixed size and variable sized blocks.

  Explicit Allocation for Fixed Size Blocks
- ✓ This is the simplest technique of explicit allocation in which the size of the block for which memory is allocated is fixed.
- ✓ In this technique a free list is used. Free list is a set of free blocks. This observed when we want to allocate memory. If some memory is de-allocated then the free list gets appended.
- ✓ The blocks are linked to each other in a list structure. The memory allocation be done by pointing previous node to the newly allocated block. Memory deallocation can be done by de-referencing the previous link.
- ✓ The pointer which points to first block of memory is called Available.

✓ This memory allocation and deallocation is done using heap memory.

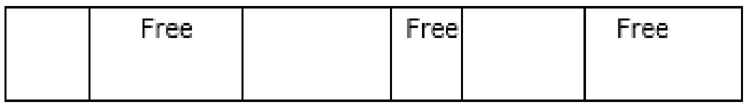


✓ The explicit allocation consists of taking a block off the list and deallocation consist of putting the block back on the list.

✓ The advantage of this technique is that there is no space overhead.

<u>Explicit Allocation of Variable Sized Blocks</u>

✓ Due to frequent memory allocation and deallocation the heap memory becomes fragmented. That means heap may consist of some blocks that are free and some that are allocated.



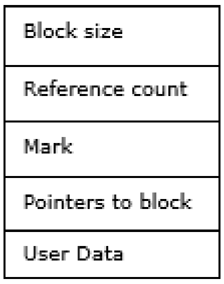
✓ In Fig. a fragmented heap memory is shown. Suppose a list of 7 blocks gets allocated and second, fourth and sixth block is deallocated then fragmentation occurs. Thus we get variable sized blocks that are available free.

✓ For allocating variable sized blocks some strategies such as first fit, worst fit and best fit are used.

✓ Sometimes all the free blocks are collected together to form a large free block. This ultimately avoids the problem of fragmentation.

2. Implicit Allocation

- ✓ The implicit allocation is performed using user program and runtime packages.
- ✓ The run time package is required to know when the **storage block** is not in use
- ✓ The format of **storage block** is as shown in following Fig



- ✓ There are two approaches used for implicit allocation.

Reference count:

- ✓ Reference count is a special counter used during implicit memory allocation. If any block is referred by some another block then its reference count incremented by one. That also means if the reference count of particular block drops down to 0 then, that means that block is not referenced one and hence it can be deallocated. Reference counts are best

used when pointers between blocks never appear in cycle.

Marking techniques:

- ✓ This is an alternative approach to determine whether the block is in use or not. In this method, the user program is suspended temporarily and **frozen pointers** are used to mark the blocks that are in use. Sometime bitmaps are used. These pointers are then placed in the heap memory. Again we go through hear memory and mark those blocks which are unused.
- ✓ There is one more technique called **compaction** in which all the used blocks are moved at the one end of heap memory, so that all the free blocks are available in one large free block.

Questions:

1) What are the issues of source language?
2) What is the activation tree? Explain with an example.
3) Explain the control stack with an example.
4) What is storage organization? Explain storage organization with an example.
5) Why do dangling references come? Explain with an example.
6) Explain the parameter passing technique with its type.
7) Explain the symbol table in detail.
8) Explain operations on the symbol table.
9) Explain the dynamic storage allocation technique with its type?