Unit – 4
Storage Organization-Run Time Memory Management

Mrs. Ruchi Sharma

ruchi.sharma@bkbiet.ac.in

Run Time Storage

Definitions

- During the execution of a program, the same name in the source can denote different data objects in the computer.
- The allocation and deallocation of data objects is managed by the run-time support package .
- **Terminologies:**
    - **name** → storage space: the mapping of a name to a storage space is called environment .
    - **storage space** → value: the current value of a storage space is called its state.
    - **The association of a name to a storage location is called a binding.**
- Each execution of **a procedure is called an activation .**
    - If it is a recursive procedure, then several of its activations may exist at the same time.
    - **Life time:** the time between the first and last steps in a procedure.
    - **A recursive procedure needs not to call itself directly.**

Source Language Issues

Source Language Issues

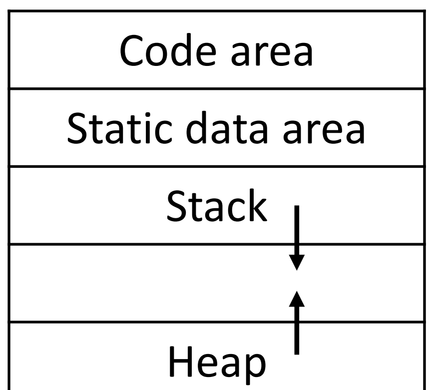
- Source language issues are:

1. Procedure call
2. Activation tree
3. Activation Records
4. Control stack
5. Scope of declaration
6. Binding of names

Storage Organization

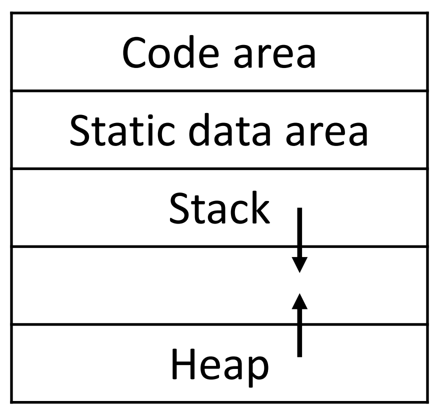
Subdivision of Runtime Memory

- The compiler demands for a block of memory to operating system.
- The compiler utilizes this block of memory executing the compiled program. This block of memory is called **run time storage.**
- The run time storage is subdivided to hold code and data such as, the generated target code and data objects.
- The size of generated code is fixed. Hence the target code occupies the determined area of the memory.



Subdivision of Runtime Memory

- The amount of memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory.
- Stack is used to manage the active procedure.
- Managing of active procedures means when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack.
- Heap area is the area of run time storage in which the other information is stored.



Activation Record

- The execution of a procedure is called its activation.
- An activation record contains all the necessary information required to call a procedure.

Activation Record

- **Temporary values**: stores the values that arise in the evaluation of an expression.
- **Local variables:** hold the data that is local to the execution of the procedure.
- **Machine status:** holds the information about status of machine just before the function call.
- **Access link (optional):** refers to non-local data held in other activation records.
- **Control link (optional):** points to activation record of caller.
- **Actual parameters:** This field holds the information about the actual parameters.
- **Return value:** used by the called procedure to return a value to calling procedure.

Temporary value
Local variables
Machine status
Access link
Control link
Actual parameters
Return value

Activation Trees

- The sequence of instructions when combined into number of procedures is known as a program and the procedure instructions are executed sequentially.
- There is a start and end delimiter and the rest is the body of the procedure.
- The body of the procedure comprises of the procedure identifier and the sequence of finite instructions.
- The process of executing a procedure is called activation. The necessary information for calling a procedure is available in an activation record.

Activation Trees

- Depending upon the source language used, the activation record contains some of the units such as:

  1. A program consist of procedures, a procedure definition is a declaration that, in its simplest form, associates an **identifier (procedure name)** with a statement **(body of the procedure**).
  2. Each execution of procedure is referred to as an activation of the procedure. Lifetime of an activation is the sequence of steps present in the execution of the procedure.
  3. **If ‘a’ and ‘b’ be two procedures then their activations will be non-overlapping (when one is called after other) or nested (nested procedures).**
  4. A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended. **An activation tree shows the way control enters and leaves activations.**

Activation Trees

Properties of activation trees are :-

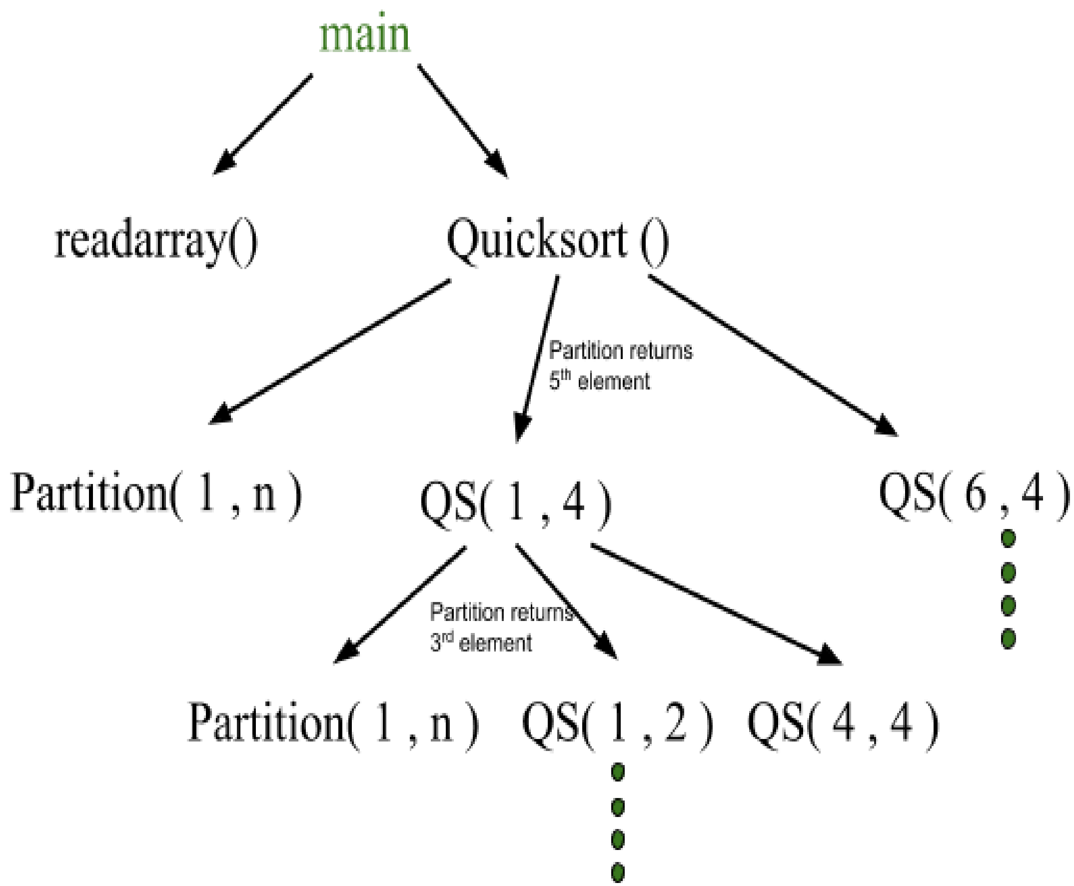
1. Each node represents an activation of a procedure.
2. The root shows the activation of the main function.
3. The node for procedure ‘x’ is the parent of node for procedure ‘y’ if and only if the control flows from procedure x to procedure y.

Activation Tree –Example:1

Program of Quicksort

```
main()
{
int n;
readarray();
 quicksort(1,n);
}
quicksort(int m, int n)
 {
int i= partition(m,n);
 quicksort(m,i-1);
 quicksort(i+1,n);
 }
```

Activation Tree of Quicksort



Activation Trees

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 activation record of the caller procedure.
Access Link	Stores the information of data which 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.

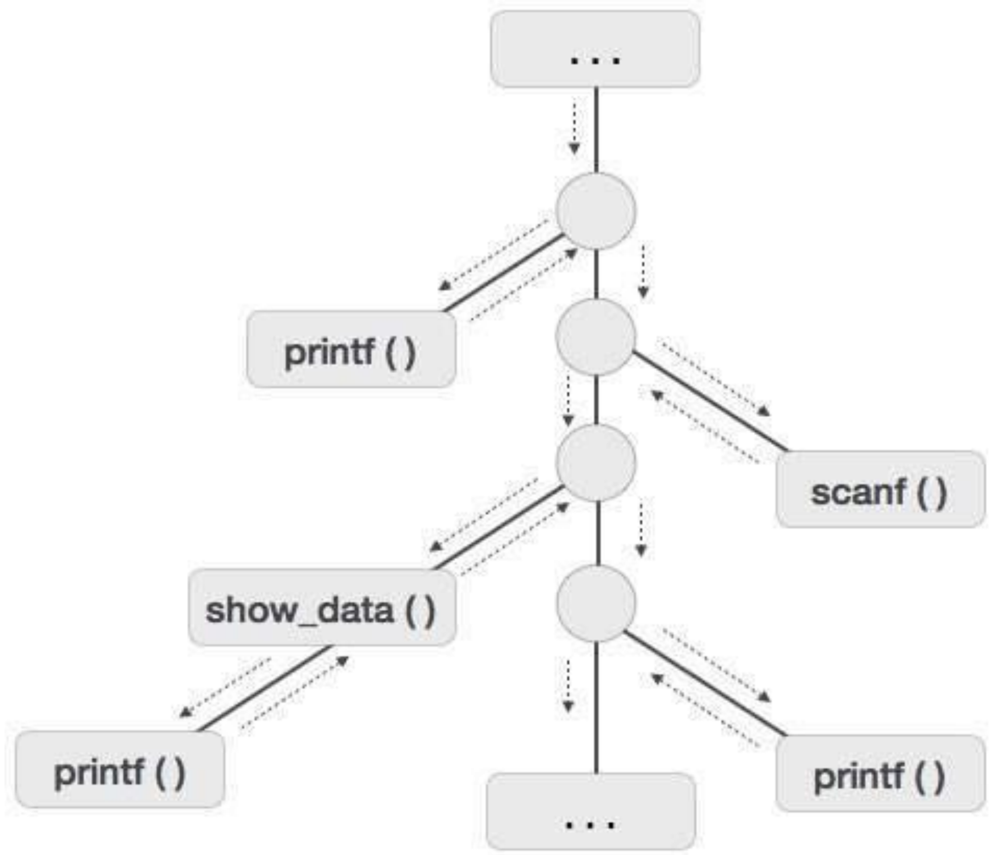
Activation Trees

- The control stack stores the activation record, when the procedure is executed. **Until execution is finished by the called procedure, the execution of the caller is suspended, when the procedure calls another procedure.** The stack stores the activation record of the called procedure.
- It is assumed that the program control flows in a sequential manner and the control of the called procedure is transferred to the called procedure. **The control is returned back to the caller on execution of the called procedure. This type of control flows represents the activation tree**.

Activation Trees –Example :2

1. . . .
2. printf("Enter Your Name: ");
3. scanf("%s", username);
4. show_data(username);
5. printf("Press any key to continue…");
6. . . .
7. int show_data(char *user)
8. {
9. printf("Your name is %s", username);
10. return 0;
11. }
12. . . .

The activation tree for the above code is depicted below:



Compile-Time Layout of Local Data

- The amount of storage needed for a name is determined from its type.
- The field of local data is laid out as the declarations in a procedure are examined at compile time.
- Variable length data has been kept outside this field.
- We keep a count of the memory locations that have been allocated for previous declarations.
- From the count we determine a relative address of the storage for a local with respect to some position such as the beginning of the activation record.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

Data structures used in compilers

Data structures used in compilers

- Two types of data structures used in a compiler are:
  1. Stack
  2. Heap

Stack

- The stack is a linear data structure.
- It follows LIFO (Last In First Out) rule.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, space is popped off the stack.

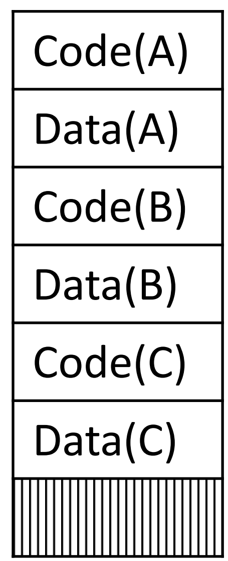
Memory allocation

Memory allocation

- Memory allocation involves three important tasks:
  1. Determining the amount of memory required for storing the value of data item.
  2. Using an appropriate memory allocation model.
  3. Developing appropriate memory mappings for accessing values stored in a data structure.
- Types of memory allocations are:
  1. Static memory allocation
  2. Dynamic memory allocation

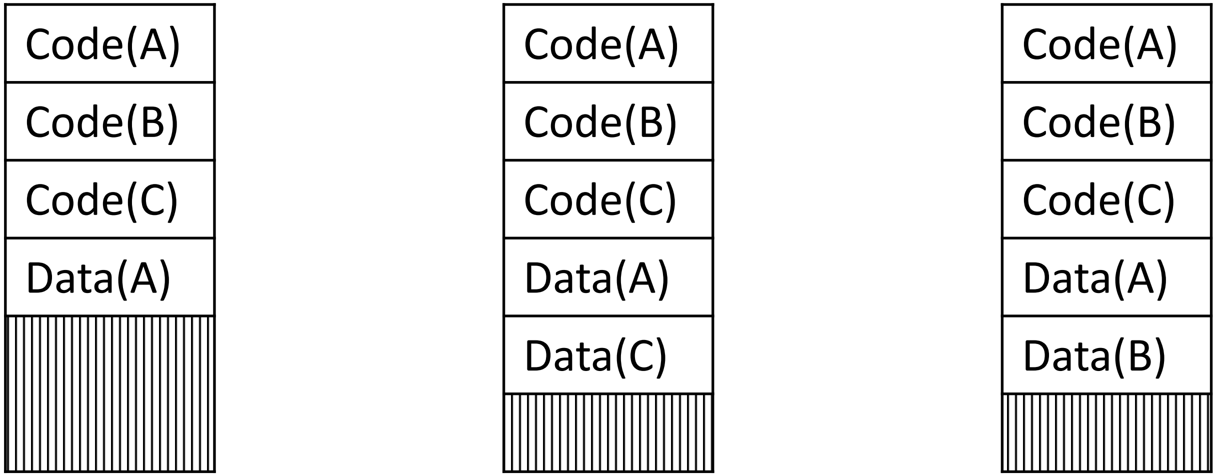
Static memory allocation

- In static memory allocation, memory is allocated to a variable before the execution of a program begins.
- No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated.



Dynamic memory allocation

- In dynamic memory allocation, memory bindings are established and destroyed during the execution of a program.

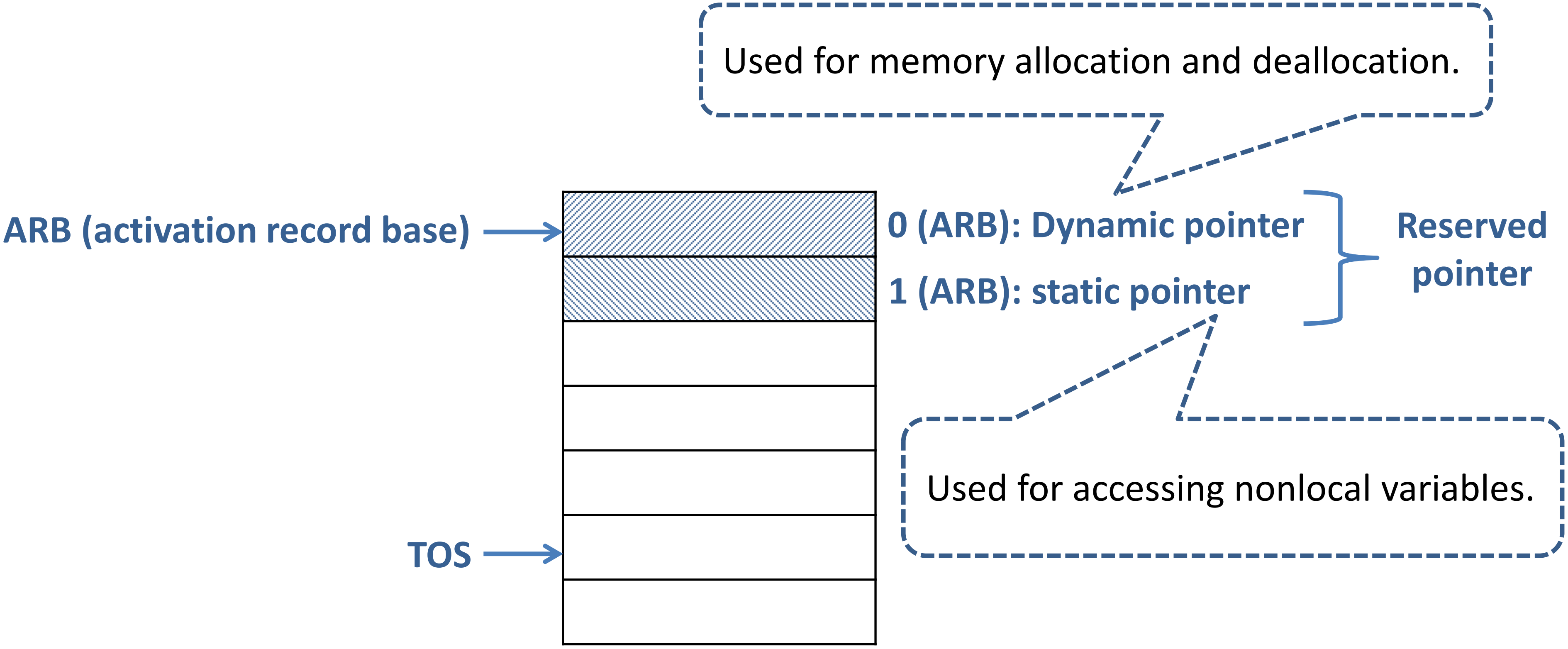


Dynamic memory allocation

- Types of dynamic memory allocation are:
  1. Automatic allocation: memory is allocated to the variables declared in a procedure when the procedure is entered during execution and is deallocated when the procedure is exited.
  2. Program controlled allocation: a program can allocate or deallocate memory at any time during its execution.

Dynamic memory allocation and access

- Each record in a stack is used to accommodate variables of one activation of a block, is called activation record.



Stack record format

Memory allocation in block structured language

- The block is a sequence of statements containing the local data and declarations which are enclosed within the delimiters.

```
A
{
        Statements
        .....
}
```

- A block structured language uses dynamic memory allocation.
- Finding the scope of the variable means checking the visibility within the block .
- Following are the rules used to determine the scope of the variable:
    1. Variable X is accessed within the block B1 if it can be accessed by any statement situated in block B1.
    2. Variable X is accessed by any statement in block B2 if block B2 is situated inside block B1.

Storage Allocation Strategies

Storage allocation strategies

The different storage allocation strategies are;

- Static allocation: lays out storage for all data objects at compile time.
- Stack allocation: manages the run-time storage as a stack.
- Heap allocation: allocates and de-allocates storage as needed at run time from a data area known as heap.

Static allocation

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, every time a procedure is activated, its names are bounded to the same storage location.
- Storage is allocated at compile time
- Static storage has fixed allocation that does not change during program execution
- As bindings do not change at runtime, no runtime support is required
- At compile time, compiler can fill the address at which the target code can find the data it operates on
- FORTRAN uses the static allocation strategy

Limitations

- Size of data objects should be known at compile time
- Recursion is not supported
- Data structures cannot be created at runtime

Stack allocation

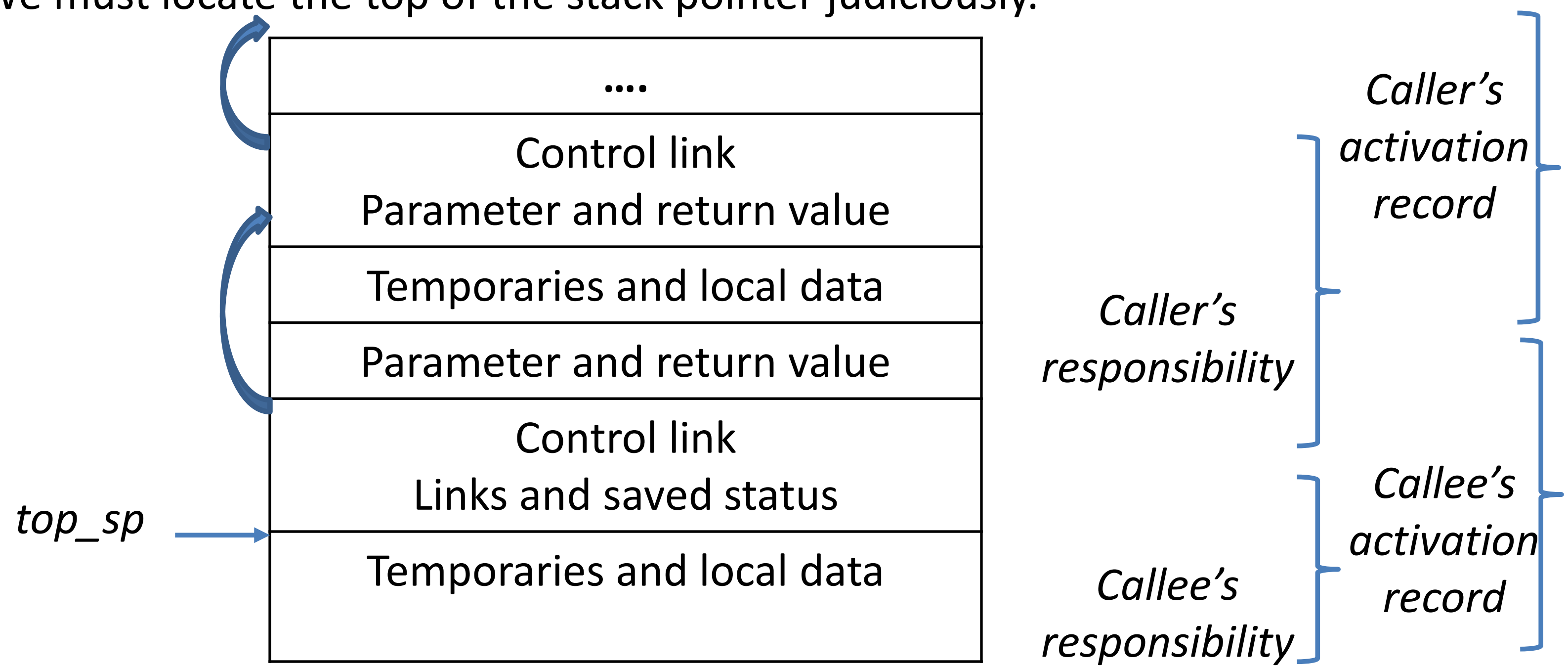
- All compilers for languages that use procedures, functions or methods as units of user define actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, the space is popped off the stack.
- Stack allocation manages the runtime storage as a stack, i.e., control stack
- Activation records are pushed and popped as activation begins and end respectively
- Locals are always bound to fresh storage in each activation, because a new activation is onto a stack when a call is made
- Values of locals are deleted as activation ends
- The data structure can be created dynamically for stack allocation

Stack allocation: Calling Sequences

- Procedures calls are implemented by generating what are known as calling sequences in the target code.
- A Return sequence restore the state of machine so the calling procedure can continue its execution.
- The code is calling sequence of often divided between the calling procedure (caller) and procedure is calls (callee).

Stack allocation: Calling Sequences

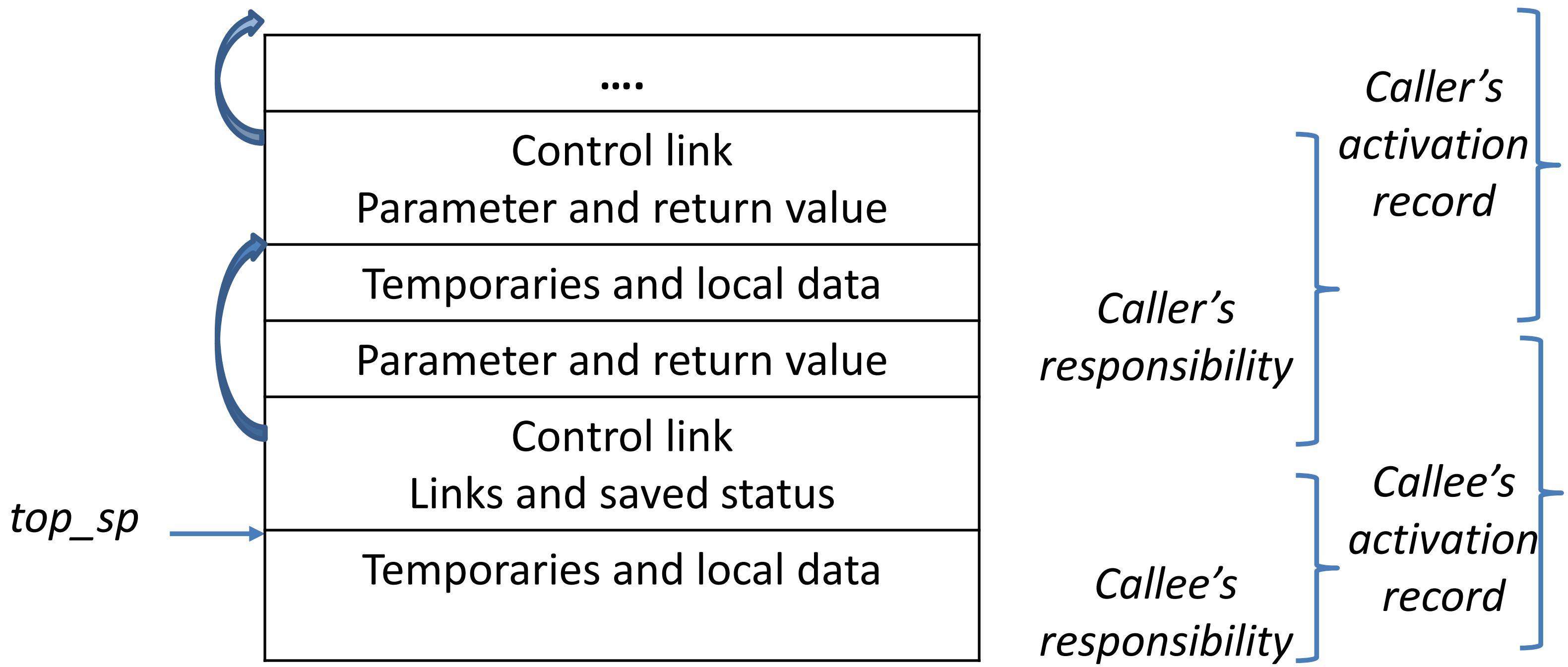
- Value communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.
- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.
- Items whose size may not be known early enough are placed at the end of the activation record. We must locate the top of the stack pointer judiciously.



Stack allocation: Calling Sequences

- The calling sequence and its division between caller and callee are as follows:

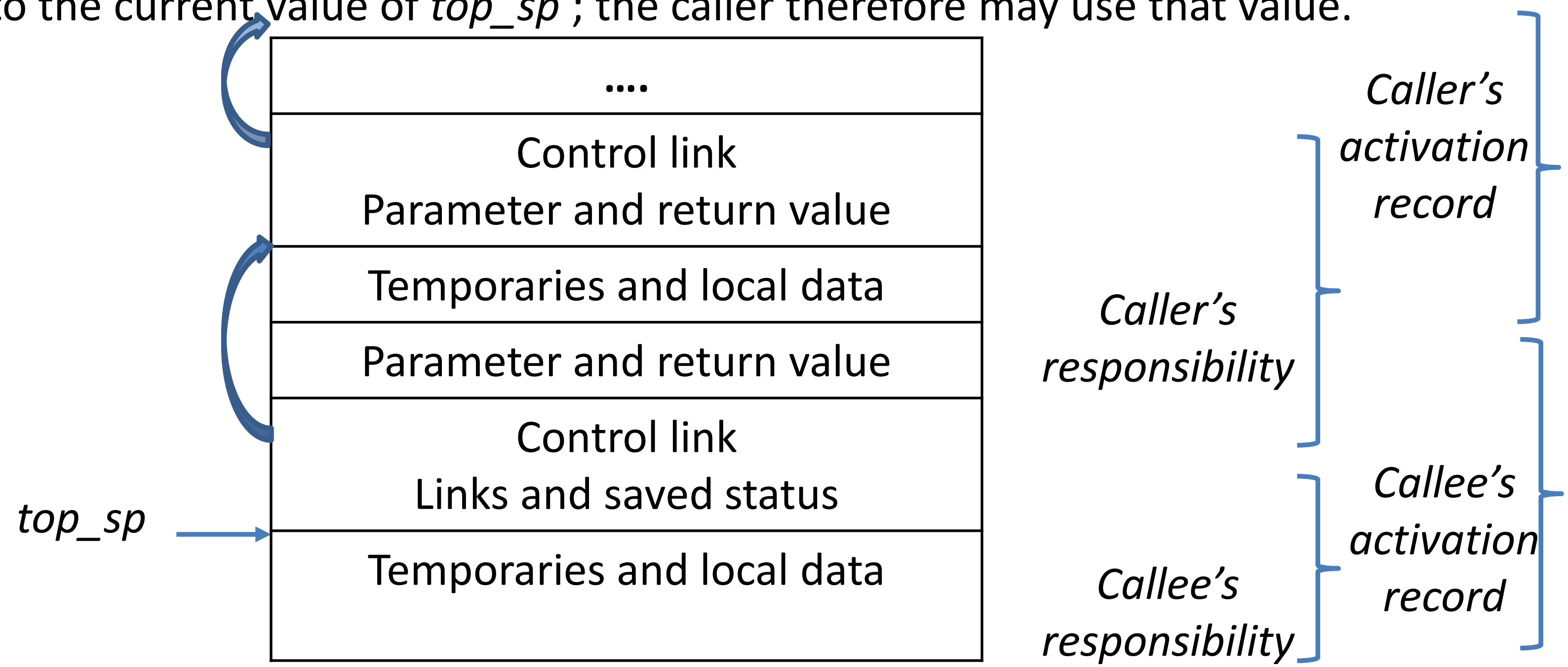
1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
3. The callee saves the register values and other status information.
4. The callee initializes its local data and begins execution.



Stack allocation: Calling Sequences

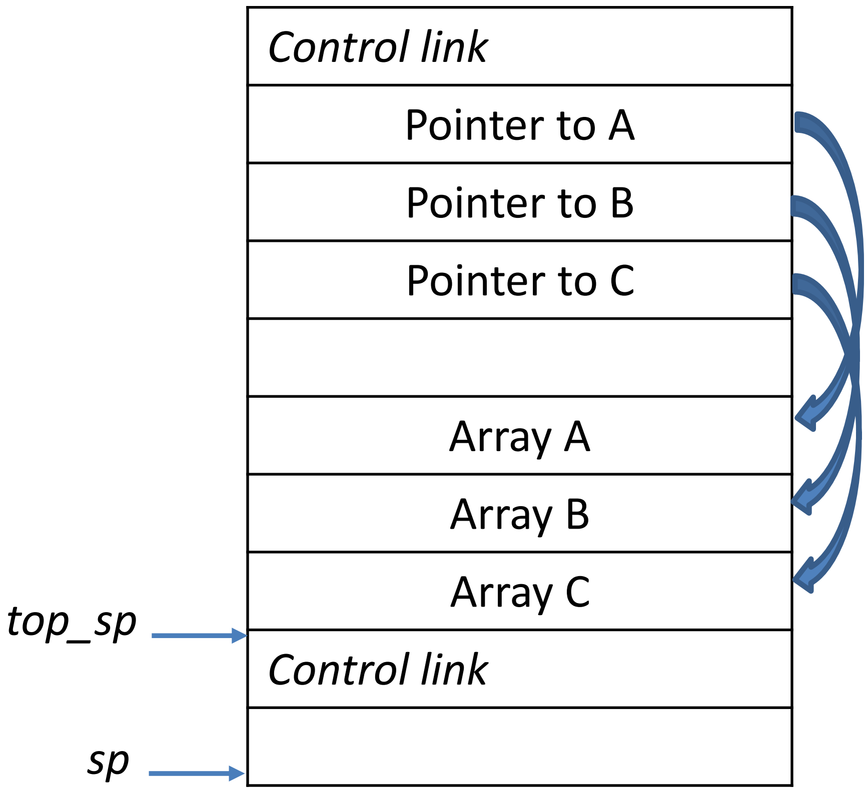
- A suitable, corresponding return sequence is:

1. The callee places the return value next to the parameters.
2. Using the information in the machine status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
3. Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp* ; the caller therefore may use that value.



Stack allocation: Variable length data on stack

- The run time memory management system must deal frequently with the allocation of objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The same scheme works for objects of any type if they are local to the procedure called have a size that depends on the parameter of the call.



Stack allocation: Dangling Reference

- Whenever storage can be allocated, the problem of dangling reference arises. **The dangling reference occurs when there is a reference of storage that has been allocated.**
- It is a logical error to use dangling reference, since, the value of de-allocated storage is undefined according to the semantics of most languages.

Limitations

- Values of locals cannot be retained once activation ends
- The memory addressing can be done using pointers and indexed registers
- This type of allocation is slower than static allocation

Heap Allocation

- 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 statically allocated memory area, 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.
- Storage can be allocated and deallocated in any order
- If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation
- It is used for retaining of local variables
- The heap allocation allocates the continuous block of memory when required for storage of activation records. This allocated memory can be deallocated when activation ends
- Free space can be further reused by heap manager
- It supports for recursion and data structures can be created at runtime

Access to Non local names

- A procedure may sometimes refer to variables which are not local to it.
- For the non local names the rules can be defined as: static and dynamic

Static scope rule

- The static scope rule is also called as lexical scope.
- Scope is determined by examining the program text.
- PASCAL, C and ADA are the languages use the static scope rule.
- These languages are also called as block structured language.

Dynamic scope rule

- For non block structured languages this dynamic scope allocation rules are used.
- The dynamic scope rule determines the scope of declaration of the names at run time by considering the current activation.
- LISP and SNOBOL are the languages which use the dynamic scope rule.

Scope rules

Scope rules

- A block in a program is a function, a procedure, or simply a unit of code that may contain data declaration.
- Entities declared in a block must have unique names.
- These entities can be accessed only within the block.
- Thus, places where an entity can be accessed or visible is referred to the **scope of that entity**.
- There are two types of variable situated in the block structured language:
  1. Local variable
  2. Non local variable

Local and nonlocal variables

Consider following program:

```
Procedure A
{
        int x,y,z
        Procedure B
        {
            int a,b
        }
}
```

Procedure	Local variables	Nonlocal variables
A		
B		

Limitations

- Heap manages overhead.

Scope rules

Scope rules

- A block in a program is a function, a procedure, or simply a unit of code that may contain data declaration.
- Entities declared in a block must have unique names.
- These entities can be accessed only within the block.
- Thus, places where an entity can be accessed or visible is referred to the **scope of that entity**.
- There are two types of variable situated in the block structured language:
  1. Local variable
  2. Non local variable

Local and nonlocal variables

Consider following program:

```
Procedure A
{
        int x,y,z
        Procedure B
        {
            int a,b
        }
}
```

Procedure	Local variables	Nonlocal variables
A		
B		

Parameter Passing Methods

Parameter passing methods

- There are two types of parameters, Formal parameters & Actual parameters.
- And based on these parameters there are various parameter passing methods, the common methods are:
  1. Call by value
  2. Call by reference
  3. Copy restore
  4. Call by name

Call by Value

- This is the simplest method of parameter passing.
- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
- The operations on formal parameters do not change the values of a parameter.

Call by Reference

- This method is also called as call by address or call by location.
- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter.
- Inside the function, the address is used to access the actual argument used in the call.
- It means the changes made to the parameter affect the passed argument.

Copy Restore

- This method is a hybrid between call by value and call by reference.
- This method is also known as copy-in-copy-out or values result.
- The calling procedure calculates the value of actual parameter and it then copied to activation record for the called procedure.
- During execution of called procedure, the actual parameters value is not affected.
- If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

Call by Name

- This is less popular method of parameter passing.
- Procedure is treated like macro.
- The procedure body is substituted for call in caller with actual parameters substituted for formals.
- The local names of called procedure and names of calling procedure are distinct.
- The actual parameters can be surrounded by parenthesis to preserve their integrity.

Symbol Table

Symbol Table

- Symbol table is a data structure used by compiler to keep track of semantics of a variable.
- Symbol table is built in lexical and syntax analysis phases.
- The items to be stored into symbol table are:
  1. Variable names
  2. Constants
  3. Procedure names
  4. Function names
  5. Literal constants and strings
  6. Compiler generated temporaries
  7. Labels in source language

Data structures for a symbol table

Data structures for a symbol table

List Data structure

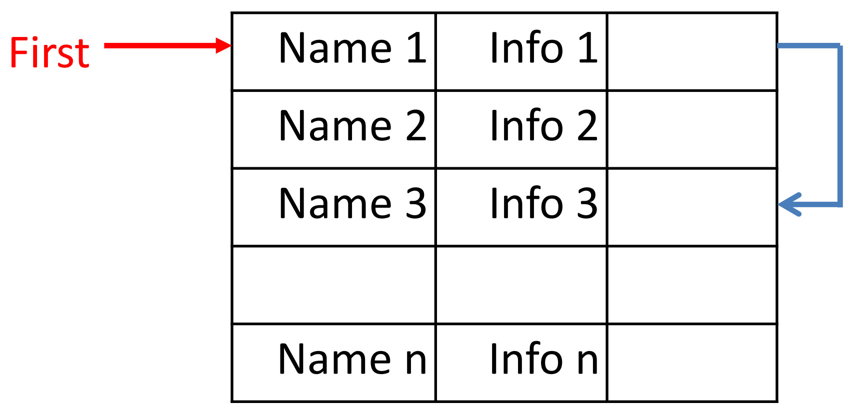
- The name can be stored with the help of starting index and length of each name.
- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in the order as they arrive.
- The list data structure using array is given below:

Name 1	Info 1
Name 2	Info 2
Name 3	Info 3
Name n	Info n

Data structures for a symbol table

Self organizing list

- This symbol table implementation is using linked list. A link field is added to each record.
- We search the records in the order pointed by the link of link field.
- The pointer “First” is maintained to point to first record of the symbol table.



Data structures for a symbol table

Binary tree

- When the organization symbol table is by means of binary tree, the node structure will as follows:
- The left child field stores the address of previous symbol.
- Right child field stores the address of next symbol.
- The symbol field is used to store the name of the symbols.
- Information field is used to give information about the symbol.

Left child	Symbols	Information	Right child

Data structures for a symbol table

Hash table

- In hashing scheme two tables are maintained-a hash table and symbol table.
- The hash table consists of k entries from 0,1 to k-1. These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such that h(name) will result any integer between 0 to k-1. We can search any name by position=h(name).
- Using this position we can obtain the exact locations of name in symbol table.
- Advantage of hashing is quick search is possible and the disadvantage is that hashing is complicated to implement.

Dynamic Storage Allocation Techniques

Dynamic Storage Allocation Techniques

- There are two techniques used in dynamic memory allocation.

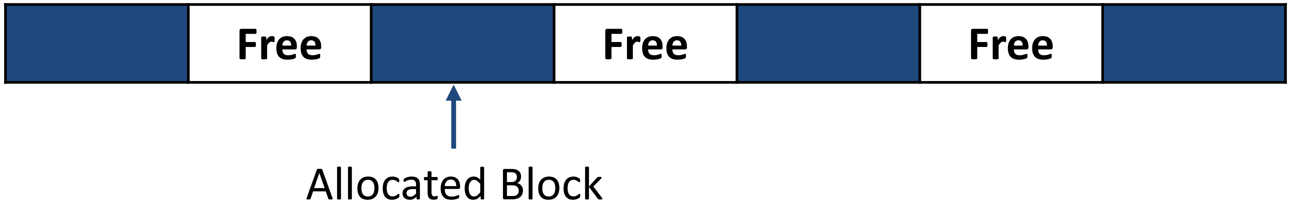
1. Explicit allocation
2. Implicit allocation

Explicit Allocation: for Fixed Size Blocks

- In explicit allocation 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.
- The blocks are linked to each other in a list structure. The memory allocation can be done by pointing previous node to the newly allocated block.
- Memory de-allocation can be done by de-referencing the previous link.
- This memory allocation and de-allocation is done using heap memory.
- The advantage of this technique is that there is no space overhead.

Explicit Allocation: for Variable Sized Blocks

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



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.

Block size
Reference Count
Mark
Pointer to Block
User Data

Block Format

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 de-allocated.
- Reference counts are best used when pointers between blocks never appear in cycle.

Implicit Allocation: 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 to the blocks that are in use.
- These pointers are then placed in the heap memory.
- Again we go through heap memory and mark those blocks which are unused.
- Using marking technique it is possible to keep track of blocks that are in use.

End of Unit-4