Type Checking and Run Time Environments

1. Type Checking:

Type systems, Specification of a simple type checker, Equivalence of type expressions, Type conversions

2. Run Time Environments:

Parameter passing by value, reference, and name; activation records, stack and static allocation of activation records; translating a function call, allocating offsets to variables, generating code for function prologue, function epilogue, call sequence, and return sequence.

RUN-TIME ENVIRONMENTS - SOURCE LANGUAGE ISSUES

Procedures:

A procedure definition is a declaration that associates an identifier with a statement.

The identifier is the procedure name, and the statement is the procedure body.

For example, the following is the definition of procedure named ***readarray*** :

```
procedure readarray;
        var i : integer;

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

When a procedure name appears within an executable statement, the procedure is said to be called at that point.

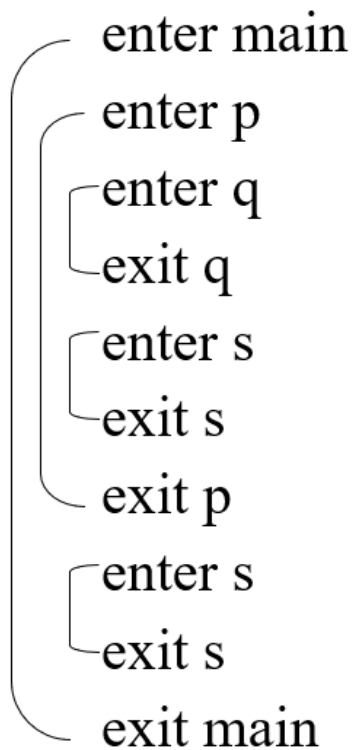
Activation trees

An activation tree is used to depict the way control enters and leaves activations.

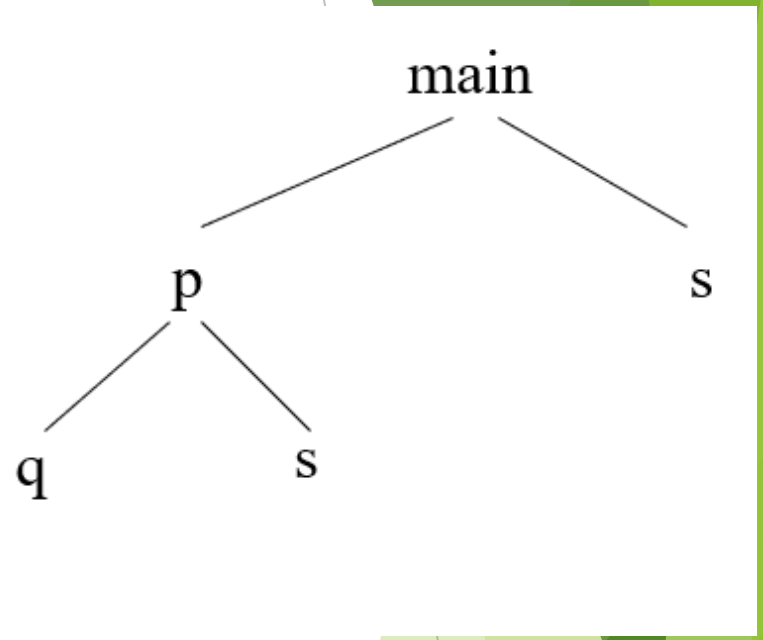
In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for "***a***" is the parent of the node for ***"b"*** if and only if control flows from activation "***a***" to ***"b"***.
4. The node for "***a***" is to the left of the node for "***b***" if and only if the lifetime of "***a***" occurs before the lifetime of "***b***".

```
program main;
  procedure s;
    begin ... end;
  procedure p;
    procedure q;
      begin ... end;
    begin q; s; end;
  begin p; s; end;
```

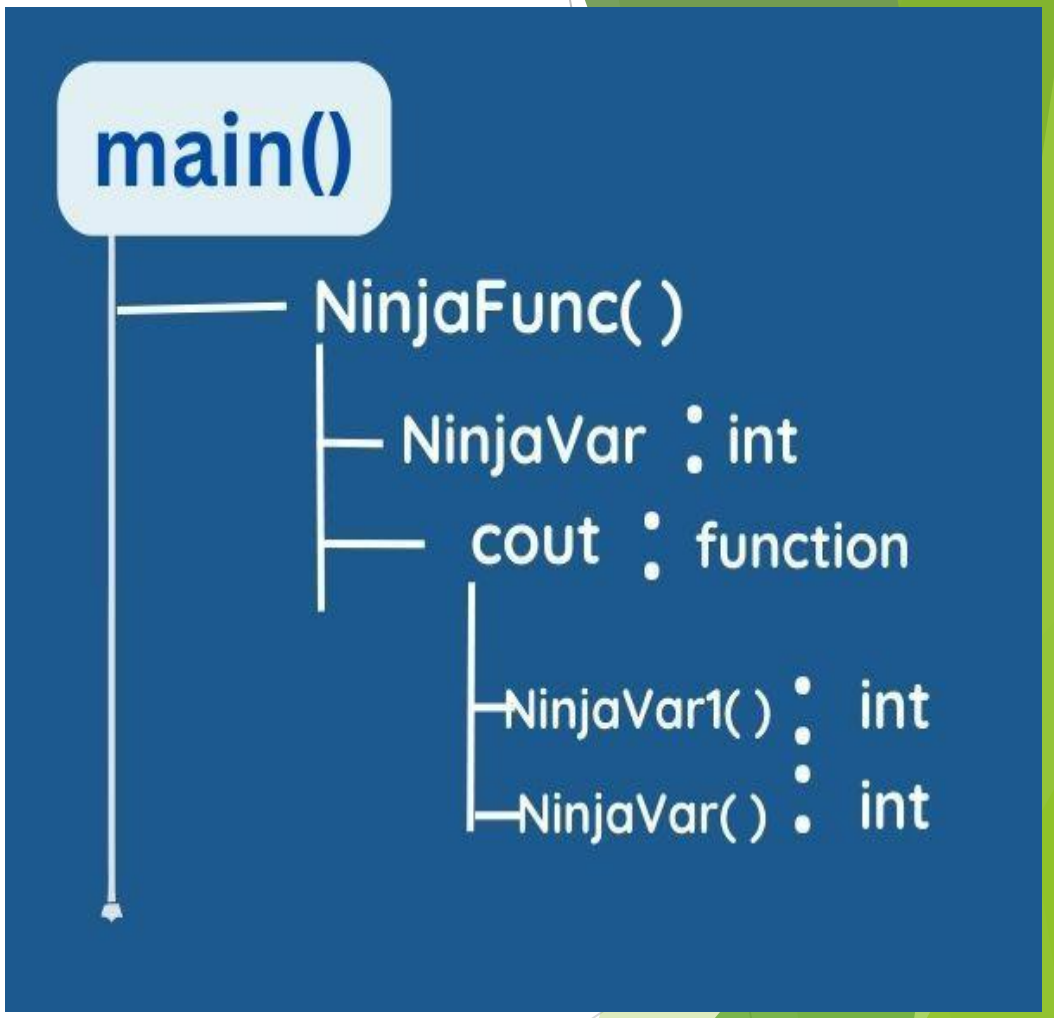


A Nested Structure

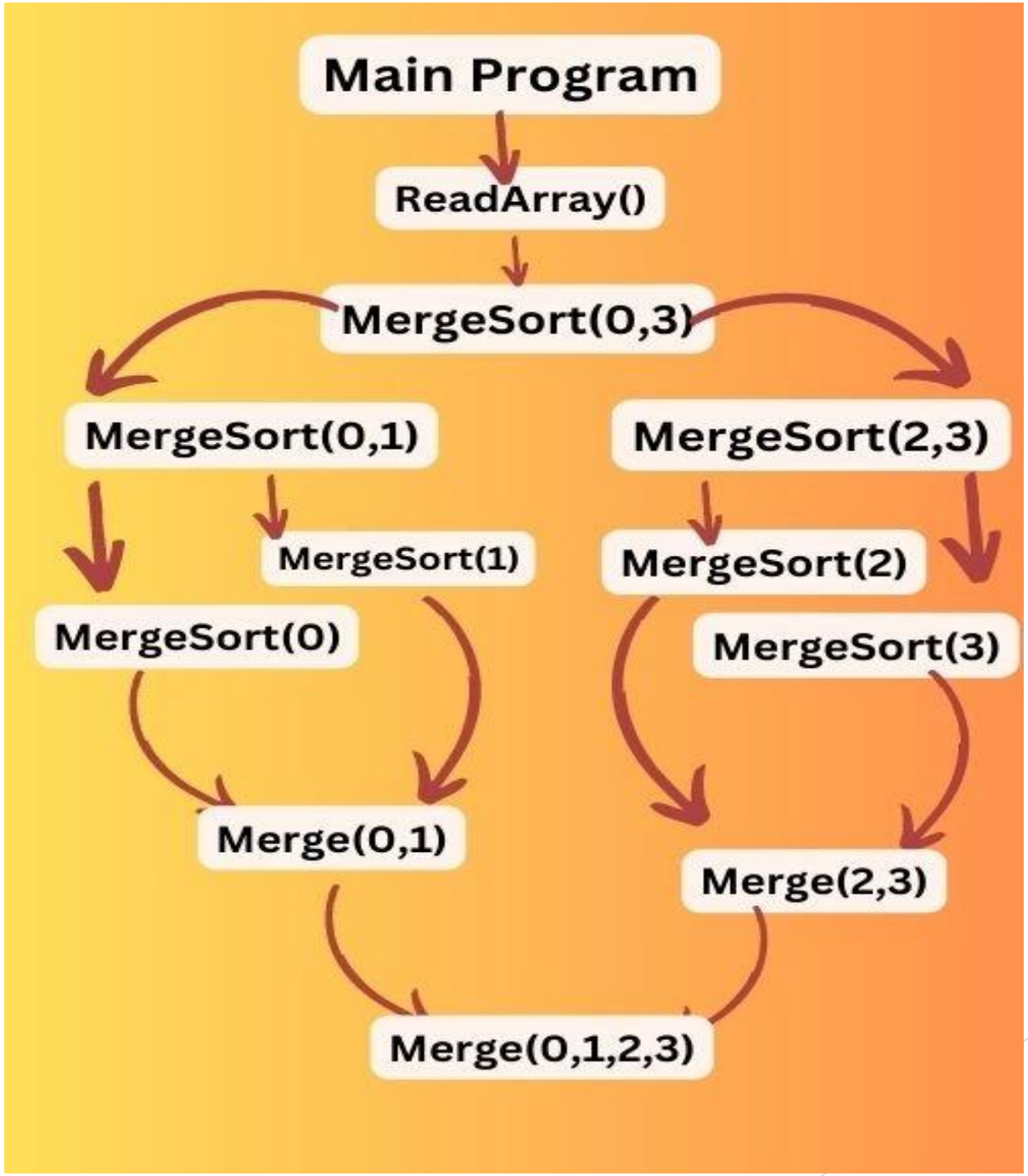


Code

```
#include <iostream>
using  namespace std;

int NinjaVar1=5;
void NinjaFunc(){
   int NinjaVar=20;
   cout<<"NinjaFunc() called in main():"<<NinjaVar1+NinjaVar;
}

int main() {

   NinjaFunc();

   return 0;
}
```



```
enter main(){
    enter  ReadArray()
    leave  ReadArray()
    enter  MergeSort(arr,0,3){

        enter  MergeSort(0,1){

            enter  MergeSort(0,0)
            leave  MergeSort(0,0)

            enter  MergeSort(1,1)
            leave  MergeSort(1,1)

            enter  Merge(0,1)
            leave  Merge(0,1)
        }

        leave  MergeSort(0,1)

        enter  MergeSort(2,3){
            enter  MergeSort(2,2)
            leave  MergeSort(2,2)

            enter  MergeSort(3,3)
            leave  MergeSort(3,3)

            enter  Merge(2,3)
            leave  Merge(2,3)
        }

        leave  MergeSort(2,3)

        enter  Merge(0,1,2,3)
        leave  Merge(0,1,2,3)
    }
    leave  MergeSort(0,3)
}
leave main()
```


Main Program
ReadArray()
MergeSort(0,3)
MergeSort(0,1)
MergeSort(2,3)
MergeSort(1)
MergeSort(2)
MergeSort(0)
MergeSort(3)
Merge(0,1)
Merge(2,3)
Merge(0,1,2,3)

Control Stack

- The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  - starts at the root,
  - visits a node before its children, and
  - recursively visits children at each node an a left-to-right order.
- A stack (called **control stack**) can be used to keep track of live procedure activations.
  - An activation record is pushed onto the control stack as the activation starts.
  - That 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.

Variable Scopes

- The same variable name can be used in the different parts of the program.
- The scope rules of the language determine which declaration of a name applies 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**: Otherwise (ie. it is declared outside of that procedure)

```
procedure p;
  var b:real;
  procedure p;
    var a: integer;
    begin a := 1;  b := 2; end;
  begin ... end;
```

a is local

b is non-local

Binding of names

The term environment refers to a function that maps a name to a storage location.

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

When an environment associates storage location s with a name x, we say that x is bound to s.

This association is referred to as a binding of x.

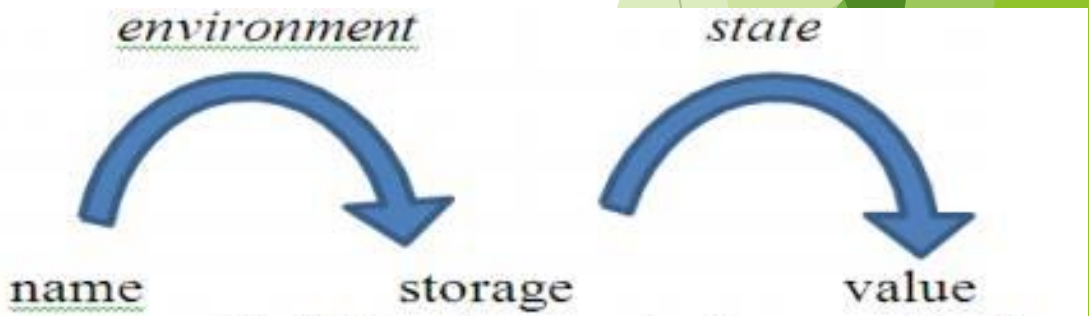
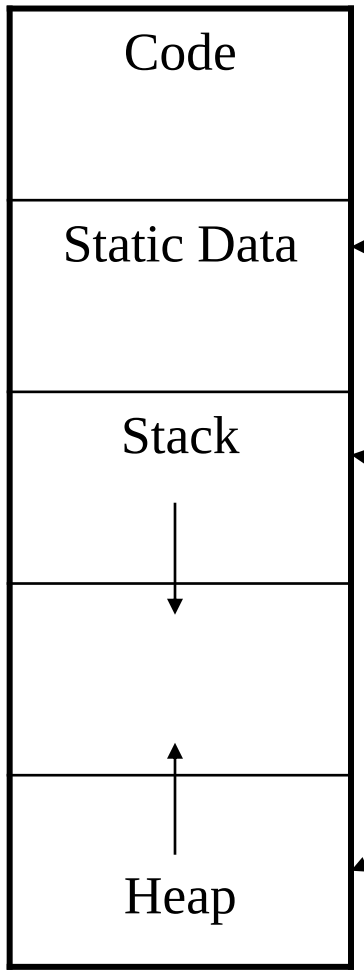


Fig. 2.8 Two-stage mapping from names to values

Run-Time Storage Organization



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).

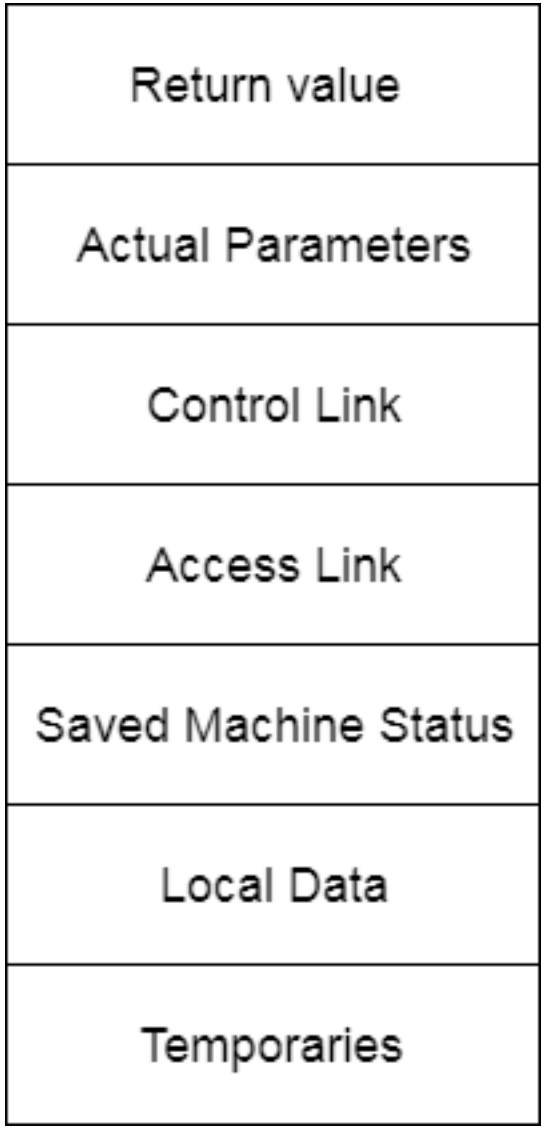
- Static Area is fixed.
- Stack & Heap can grow but in opposite direction of each other.

Run-time storage can be subdivide to hold the different components of an executing program:

- Generated executable code
- Static data objects
- Dynamic data-object that is heap
- Automatic data objects that is stack

Activation Records

1. Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
2. When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
3. Activation record is used to manage the information needed by a single execution of a procedure.
4. An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function



Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

Control Link: It points to activation record of the caller.

Access Link: It is used to refer to non-local data held in other activation records.

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data that is local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression

Storage Allocation

The information which required during an execution of a procedure is kept in a block of storage called an activation record. The activation record includes storage for names local to the procedure.

We can describe address in the target code using the following ways:

Static storage allocation

In static allocation, the position of an activation record is fixed in memory at compile time.

Stack storage allocation

In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.

Heap storage allocation

Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.

Static storage allocation

1. In static allocation, names are bound to storage locations.
2. If memory is created at compile time then the memory will be created in static area and only once.
3. Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
4. The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
5. Another drawback is restriction of the recursion procedure.

Stack Storage Allocation

1. In static storage allocation, storage is organized as a stack.
2. An activation record is pushed into the stack when activation begins and it is popped when the activation end.
3. Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
4. It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

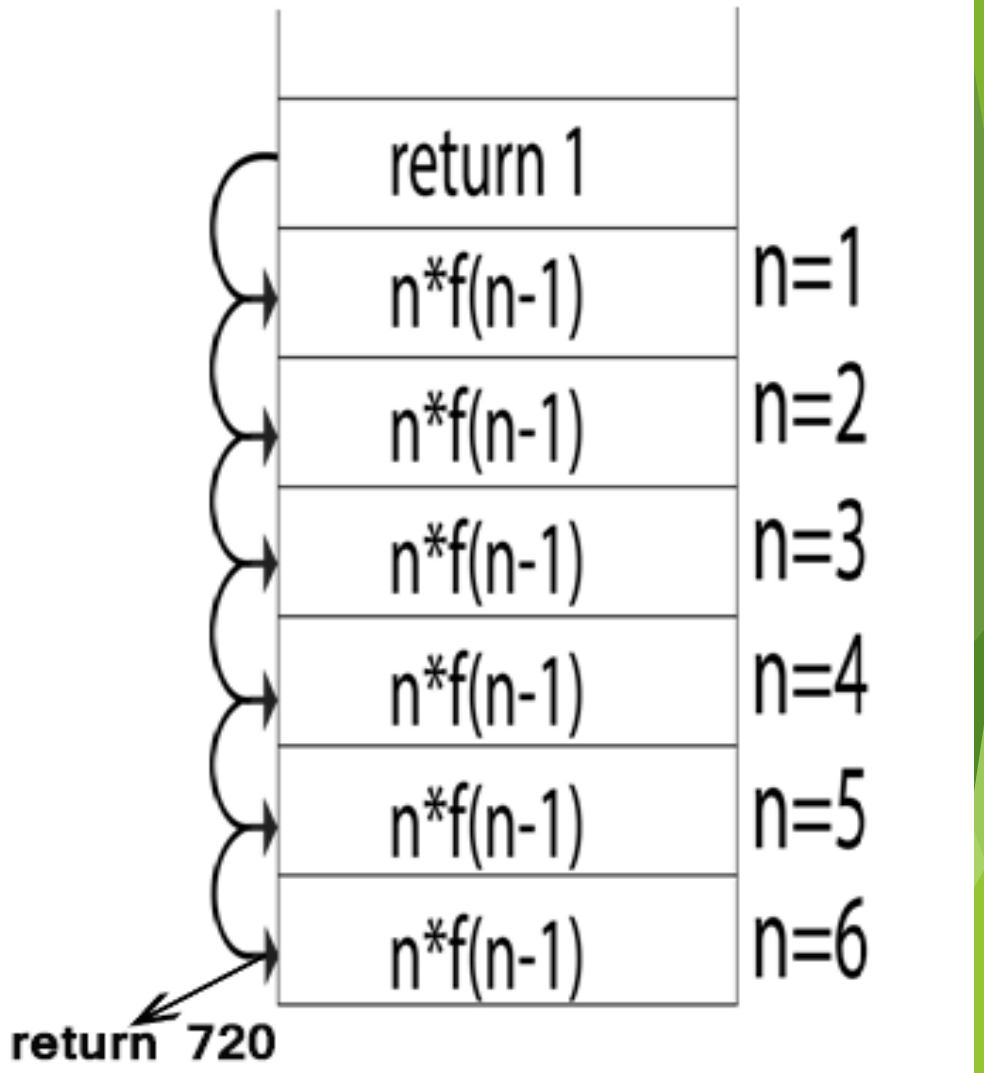
Heap Storage Allocation

1. Heap allocation is the most flexible allocation scheme.
2. Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
3. Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
4. Heap storage allocation supports the recursion process.

Example:

```
fact (int n)
{
   if (n<=1)
      return 1;
   else
      return (n * fact(n-1));
}
fact (6)
```

The stack allocation is as follows:



RUN-TIME STORAGE MANAGEMENT

For the run-time allocation and deallocation of activation records the following three-address statements are associated:

- ✓ Call
- ✓ Return
- ✓ Halt
- ✓ Action, a placeholder for other statements

We assume that the run-time memory is divided into areas for:

- ❑ Code
- ❑ Static data
- ❑ Stack

Static allocation:

1. Implementation of call statement:
The following code is needed to implement static allocation:

MOV *#here + 20*, *callee.static_area* /*it saves return address*/</p>
GOTO *callee.code_area* /* It transfers control to the target code for the called procedure*/

Where,

callee.static_area shows the address of the activation record.

callee.code_area shows the address of the first instruction for called procedure.

#here + 20 literal are used to return address of the instruction following GOTO.

2. Implementation of return statement:
The following code is needed to implement return from procedure callee:

GOTO **callee.static_area*

- It is used to transfer the control to the address that is saved at the beginning of the activation record.

3. Implementation of action statement:
The ACTION instruction is used to implement action statement.

4. Implementation of halt statement:
The HALT statement is the final instruction that is used to return the control to the operating system.

Stack allocation:

Using the relative address, static allocation can become stack allocation for storage in activation records.
In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.

1. Initialization of stack:

```
MOV #stackstart , SP    /*initializes stack*/
HALT                    /*terminate execution*/
```

2. Implementation of Call statement:

ADD *#caller.recordsize, SP* /* increment stack pointer */

MOV *#here + 16, *SP* /*Save return address */

GOTO *callee.code_area*

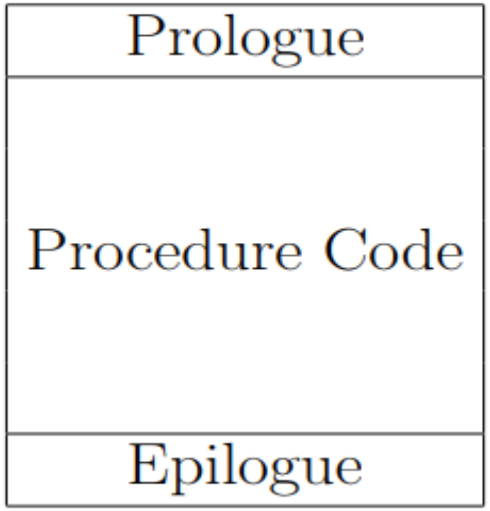
Where, *caller.recordsize* is the size of the activation record

#here + 16 is the address of the instruction following the GOTO

3. Implementation of Return statement:

GOTO **0 (SP)* /*return to the caller */

SUB *#caller.recordsize, SP* /*decrement SP and restore to previous value */



Prologue: (or *preamble*) Save registers and return address; transfer parameters.

Epilogue: (or *postamble*) Restore registers; transfer returned value; return.

A `return` statement in a procedure is compiled to:

1. Load the returned value into a register.
2. `goto` the Epilogue.