6 Run Time Environment

6.1 INTRODUCTION

- Compiler must cooperate with the operating system and other system software to support abstraction (scope, binding, data types, the flow of control construct) on the target machine.
- To do so, the compiler creates and manages a run time environment in which it assumes its target programs are being executed.
- Environment deals with a variety of issues such as layout and allocation of the storage location for the object named in the source program, the mechanism used to access variables, the linkage between procedures, the mechanism for parameter passing, and other programs.
- Run time environment act as an interface between higher–level concepts of the programming language and lower concepts supported by the target machine.

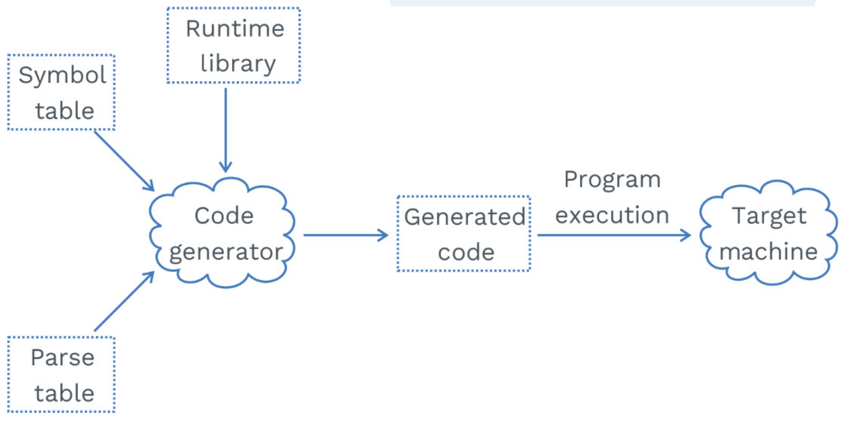


Fig. 6.1 Introduction

Procedures:

Procedures are the functions defined in the simplest form, which associates the identifiers with the statement (body of the procedure)

Activation tree:

- We represent the activation of the procedures during the running of an entire program by a tree called an activation tree.
- Each node corresponds to one activation, and the root is the activation of the main procedure that initiates the execution of the program.

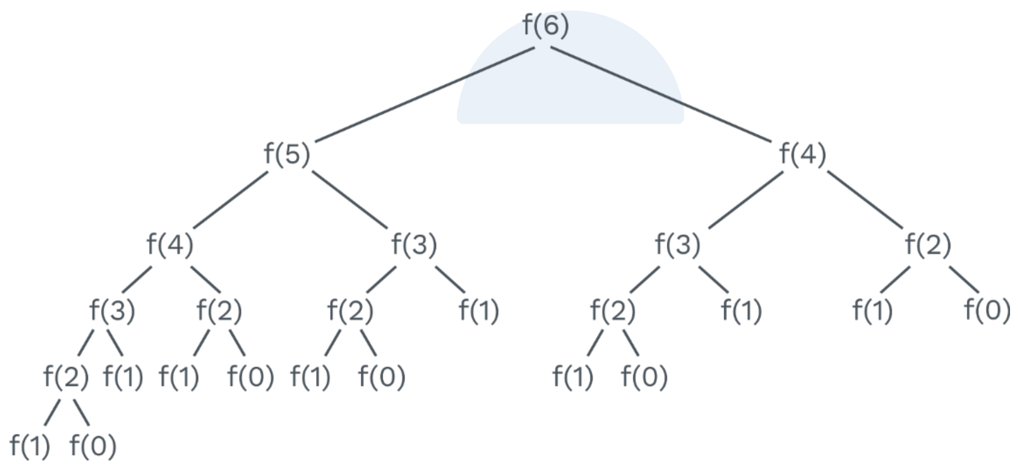
SOLVED EXAMPLES

Q1 **Given a program to print the FIBONACCI series:**

```
    f(n)                         int main ( )
{   if (n= = 0 || n= =1)         {       int n=6;
    return (n) ;                 cout << f(n);
else                             getchar( ) ;
{   a=f(n-1);                    return 0;
    b=f(n-2);                    }
    c=a+b;
    return (c);
    }
}
```

Draw an activation tree for the given program.

Sol:



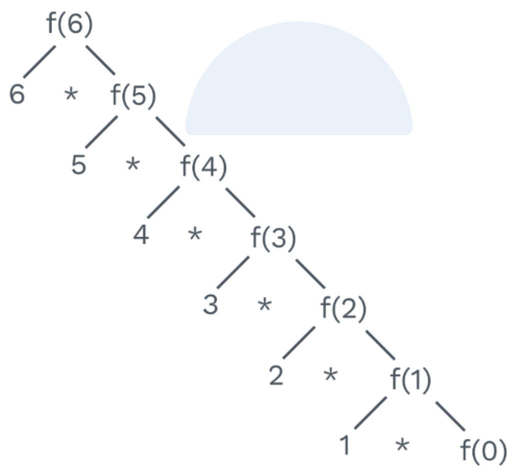
This is the activation tree in which f(n) represents the calling function during the execution of the program.

Q2 **Given a program for finding factorial:**

```
long fact(int n)
{   if (n–1)
    return n*fact(n-1);
    else
    return 1;
}
int main ( )
{   int n=6;
    printf (“Enter a positive integer:”);
    scanf (“%d”, &n);
    printf (“factorial of %d=%d”, n, fact(n));
    return 0;
}
```

Draw an activation tree for the function calls of the above code.

Sol:



Where f(n) equivalent to fact (n).

Control stack:

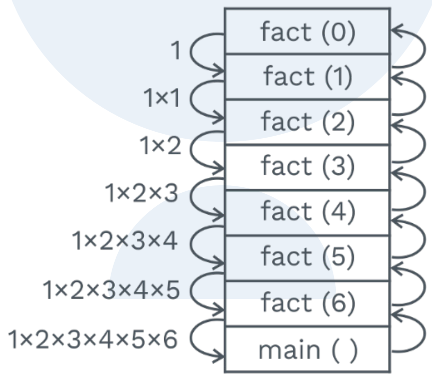
- Control stack maintains or tracks how the procedures are activated.
- When a procedure is called, it is pushed on the control stack.
- When a procedure completes its execution, it is popped out of the control stack.

SOLVED EXAMPLES

Q1 **During the execution of the given code what will be the size of stack required.**

```
int fact(int n)                  int main ( ) {
{   if n ≥1                      int n=6;
    return n*fact(n-1);          int factorial=fact(n);
    else                         Printf(" factorial of 6 = %d", factorial);
    return 1;                    return 0;
}                                }
```

Sol:



The size of the control stack that is required during the execution of the above program is 8 blocks of the stack.

Scope of declaration:

- Scope of declaration determines the declaration of an identifier when it appears in the text of the program.
- Portion of the program to which the declaration applies is called the scope of that declaration.
- Scope of declaration is of two types
  i) Local scope: Scope which is limited to a function or small segment of the program.
  ii) Global scope: The scope of a variable or identifier which is present in the entire program is known as a global scope.

At compilation time, with the help of the symbol table, we can determine whether the scope of the variable is local or global.

SOLVED EXAMPLES

Q1 **Given a code below:**

```
#include<stdio.h>
    int a ;
    void function1( )
    {         int b ;
              printf ('b');
    }
int main ( )
{   int c ;
    c=a+b;
    printf ("%d",c);
    return 0;
}
```

From the above code, identify variables that have local and global scope separately.

Sol: Variable 'a' has a global scope; its scope is limited to be an entire program Variables b and c have a local scope because the scope of variable b is limited to function 1 ( ) and scope of variable c is limited to function main ( ). Both b and c can not be used outside the main function.

Binding of names:

- Binding of name is a process in which any name present in the code is bound to the storage location.

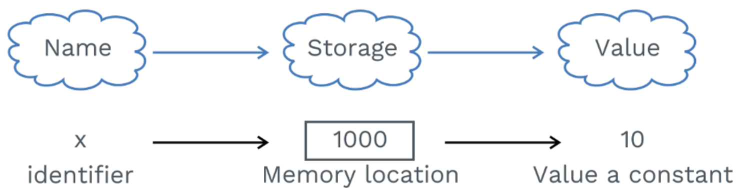


Fig. 6.2 Binding of Names

- A name 'x' an identifier is bound with the memory location referred by the address [1000]. This is known as the binding of names.

6.2 STORAGE ORGANISATION

- According to the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organisation of this logical address space are shared between the compiler, operating system, and target machine.
- Operating system maps the logical address into a physical address which is usually spread throughout the memory.

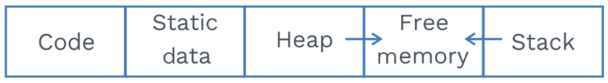


Fig. 6.3 Storage Organisation

Sub-division of runtime memory into code and data areas.

6.3 ACTIVATION RECORD

- Procedure calls and returns are managed by the control stack.
- When the procedure is called, its activation record is pushed into the stack with the root of the activation tree at the bottom of the stack.
- Contents of the activation record are:

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Table 6.1 Activation Record

A general activation record

1) Temporaries:
Temporary values such as those arising from the evaluation of expressions, in the case where those temporaries can not be held in a register.

2) Local data:
Local data is the one that is limited to its procedure only.

3) Save machine status:
A save machine status information about the status of the machine just before the call of the procedures. The information typically includes the return address (value of the program counter to which the called procedure must return) and the content of the register that was used by the calling procedure.

4) Access link:
The non-local data of other activation records can be referred by access link if required.
It mainly helps to access the data that are not local to the activation record.

Code memory:
During compilation time, generated target code size is fixed thus; target code can be placed at the area of the code, which is statically determined. This static area of the code is usually present at the lower end.

- **Static data:**

The size of some program data objects, such as global constants, and the data generated by the compiler, such as information to support garbage collection, may be known at compile-time, and these data can be placed in another statically determined area known as static data.

- **Stack and heap:**
- To maximise the utilisation of space at the run time, two areas stack and heap at opposite ends of the address space.
- These two areas grow towards each other as needed. In usual practice, the stack grows towards lower addresses, and heap grow towards higher addresses.
- Stack is used to store data structures called activation records that get generated during the procedural calls.
- Many programming languages allocate and deallocate data under program control, for eg: C has a function malloc and free. The heap is used to manage this kind of data.

Rack Your Brain

What is the reason for statically allocating as many data objects as possible?

5) Control link:
The control link points to the activation record of the caller.

6) Returned values:
Returned values are the values that are returned by the functions.
Not all procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7) Actual parameter:
The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in the register, when possible, for greater efficiency.

SOLVED EXAMPLES

Q1

```
main ( )
{  ----
   ----
   P ( )
   {
      Q ( );
   }

Q ( )
{  ----
   ----
   R ( )
   {
      S ( );
   }
   ----
   T ( );
}

S ( )
{  ----
   ----
   T ( );
   U ( );
   {
      V ( );
   }
}

T ( )
{
   P ( );
}
V ( )
{
   S ( );
   T ( );
}
```

For the above program find the access link and control link for each function.

Sol: Access link of the function is the one in which the function is defined.

Function	**Access link**
main ( )	NULL
P ( )	main ( )
Q ( )	main( )
R ( )	Q ( )
T ( )	main ( )
S ( )	main ( )

U ( )	S ( )
V ( )	main ( )

The control link of a procedure is the procedure which is called inside it.

Function	**Control link**
main ( )	NULL
P ( )	Q ( )
Q ( )	T ( )
R ( )	S ( )
S ( )	T ( )
T ( )	P ( )
U ( )	V ( )
V ( )	S ( ), T ( )

6.4 STORAGE ALLOCATION STRATEGIES

There are three storage allocation techniques:

1) Static storage allocation

2) Stack storage allocation

3) Heap storage allocation

Static storage allocation:

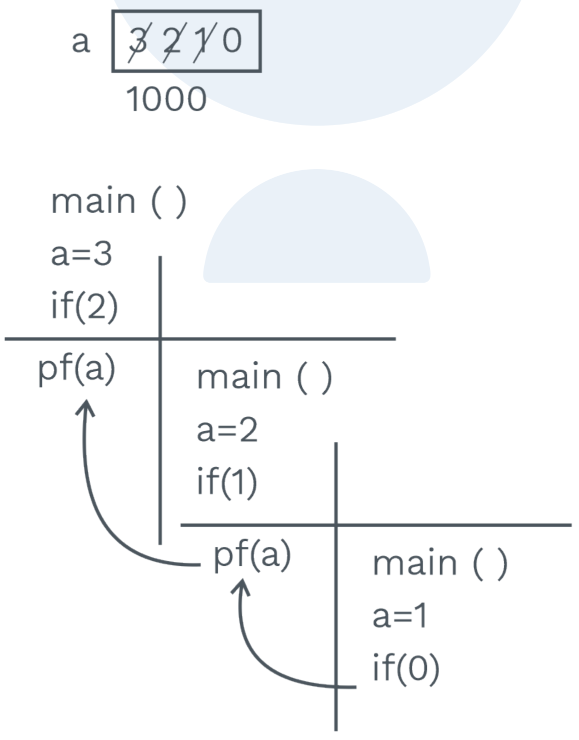
- If we do static allocation of any variable, then we put the keyword static before the data type of the variable.
- For static variables, memory is allocated in the static area, and it will be allocated only once.
- For static variable, memory will be allocated at compilation time only.
- initialisation of static variables can be done only once.
- Recursion does not support static storage allocation.
- Dynamic data structures are also not supported in static storage allocation.
- For static variables, its binding is done at compile-time and it can not be changed at runtime.
- Binding is mapping the variable to its storage location, if it is done at runtime, then it is known as runtime binding.

SOLVED EXAMPLES

Q1 **What will be the output for the given code:**

```
void main ( )
{   static int a=3;
    if (- -a)
    {        main ( );
             printf (a);
    }
}
```

Sol: Memory allocation in static area thus can be initialised only once.



$\boxed{\text{Output = aa}}$

NOTE

If in the above program, variable 'a' is not static, then the given function will never terminate because whenever main ( ) will be called again a new memory space will be allocated to variable 'a', and it will again be initialised to '3' thus if condition will never fail.

Stack storage allocation:

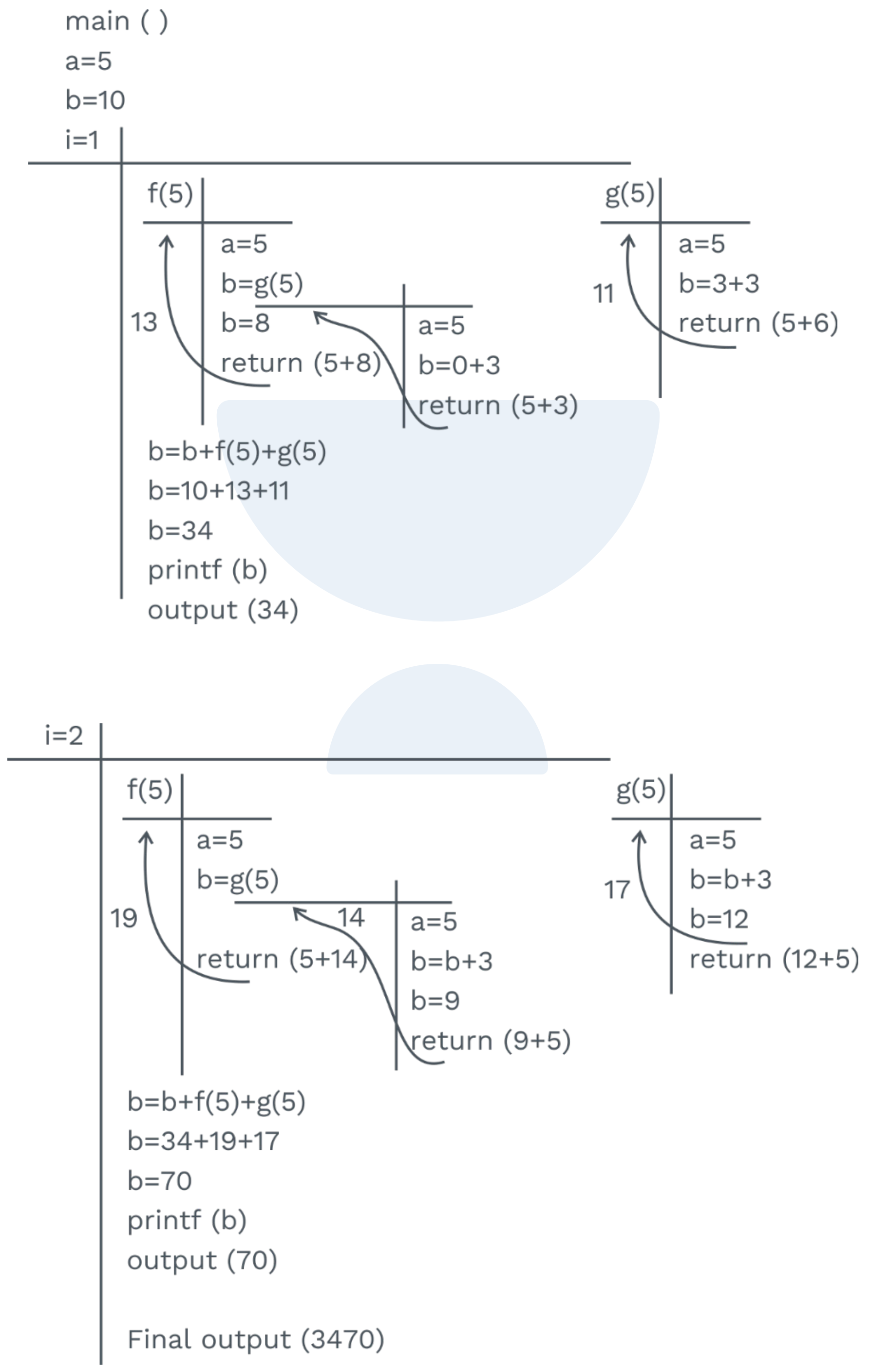
- Whenever the procedure is called, its activation record is pushed into the stack.
- Whenever the procedure finished its execution, its activation record is popped out from the stack.
- One of the most important use of stack storage allocation is that it supports recursion.
- Locals are contained in the activation record, so they have bound a new storage space each time a function is called.
- Drawback of stack storage allocation is that dynamic data structures are not supported.
- One major drawback with stack storage allocation is that once the procedure completed its execution, it popped out of the stack, and we can not get it back if it is required afterwards.

SOLVED EXAMPLES

Q1 **Given a code if stack storage allocation is used than what will be the output.**

```
void main ( )                      f (int a)               g (int a)
{  int a=5, b==10, i;              {     int b;            {    static int b=0;
   for (i=1, i≤ 2;i++)                   b=g(a);                 b+=3;
   {       b+=f(a) + g(a);               return (a+b);            return (a+b);
           printf (b);             }                       }
   }
}
```

Sol:



```
main ( )
a=5
b=10
i=1
      f(5)                                         g(5)
           a=5                                          a=5
           b=g(5)                                       b=3+3
   13      b=8              a=5                    11   return (5+6)
           return (5+8)     b=0+3
                            return (5+3)
      b=b+f(5)+g(5)
      b=10+13+11
      b=34
      printf (b)
      output (34)

i=2
      f(5)                                         g(5)
           a=5                                          a=5
           b=g(5)                                       b=b+3
   19                    14 a=5                    17   b=12
           return (5+14)    b=b+3                       return (12+5)
                            b=9
                            return (9+5)
      b=b+f(5)+g(5)
      b=34+19+17
      b=70
      printf (b)
      output (70)

      Final output (3470)
```

Heap storage allocation:

- In some programs, we need to retain the local names even after the activation of the procedure ends; in such cases, we use heap memory allocation of such local names.
- Major advantage of heap memory allocation is that it supports dynamic data structures and recursion.
- Both heap and stack can be allocated and deallocated at runtime.
- Heap memory remains allocated until you explicitly free it.

6.5 SYMBOL TABLE IMPLEMENTATION

Introduction:

- One of the important data structures that the compiler uses to know information about identifiers in the source program.
- Lexical analyzer and parser populate the symbol table with information, and further code generator and optimiser use those information.
- Variable, defined constants, functions, tables, structures, etc., are stored in the symbol table.
- Even for the same language, its symbol table may vary depending upon its implantation.

Information in a symbol table:

- Identifier's name.
- Identifier's type.
- Identifier's offset.
- Scope: The specific program region where the current definition is valid.
- Other attributes : arrays, return value, records,
- parameters, etc.

Operations on the symbol table:

- **Lookup:** Lookup operation is used, whenever an identifier is seen, it is needed to check its type or create a new entry.

Previous Years' Question

Which of the following statements are correct?

1) Static allocation of all data areas by a compiler make, it impossible to implement recursion.

2) Automatic garbage collection is essential to implement recursion.

3) Dynamic allocation of activation records is essential to implement recursion.

4) Both heap and stack are essential to implement recursion.

a) 1 and 2 only
b) 2 and 3 only
c) 3 and 4 only
d) 1 and 3 only

Sol: d) **[GATE: CS 2014]**

Rack Your Brain

Why does static allocation not support recursion?

- **Insert:** Insertion usually occurs in lexical or syntax analysis phases in which we add new names to the table.
- **Modify:** Sometimes, everything about the identifier is not present at the time when it is defined thus, we need to update it later.
- **Delete:** It is needed when the procedure body ends.

Various issues in symbol table design:
- **Format of entries:** From linear arrays to tree-structured tables, various formats of entries are present.
- **Access methodology:** Linear search, binary search, hashing, etc.
- **Location of storage:** Primary memory partial storage in secondary storage.
- **Scope issues:** Identifier whether it has a global scope or local scope.

Commonly used techniques for symbol table:
- Linear table
- Hash table
- Ordered list
- Trees

Linear list implementation of symbol table:
- Simple array of records with each record corresponding to an identifier in the program.
- Eg:

```
int a, b, c, ;
real z ;
------
------
------
procedure abc
------
------
------
L1 ;
------
------
------
```

Name	Type	Location
a	integer	offset of a
b	integer	offset of b
c	integer	offset of c
z	real	offset of z
abc	procedure	offset of abc
L_1	Lable	offset of L_1

Table 6.2 Linear List Implementation of Symbol Table

- Lookup, insert, modify can take O(n) where 'n' is the number of entries because it will simply apply linear search on the array.
- Insertion can be done in O(1) time when we know the pointer to the next free location.

Ordered list implementation of the symbol table:

- It is simply a variation of the linear list in which the list is organised either in ascending or descending order.
- Therefore, binary search can be applied, which takes O(log(n)) times for 'n' entries.
- Thus, lookup and modification can be done in O(logn) time, but insertion will take O(nlogn) time because we have to sort the list.

Tree implementation of the symbol table:

- Node of the tree represents each entry of the symbol table.
- Based on the string comparison of names, entries lesser than a reference node are kept in the left subtree, otherwise in the right sub-tree.
- Lookup time for height-balanced tree is O(logn).

Proper height-balanced techniques should be used.

Eg:

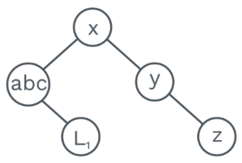


Fig. 6.3 Tree Implementation of the Symbol Table

Hash table implementation of the symbol table:

- Most of the compilers used hash tables for the implementation of the symbol table.
- mapping is done using a hash function that results in a unique location in the table organised as an array.
- Access time for a hash table is O(1)
- Improper hash function results in mapping several symbols to the same location. To overcome this, a proper collision resolution technique is used.
- To keep collisions reasonable, the hash table is chosen to be of size between n and 2n for n keys.

Rack Your Brain

Which is the best data structure for implementation of symbol table and why?

Chapter Summary

- Runtime environment provides proper services to the executing processes by including the software libraries and environment variables.
- Activation of procedure during the running of the entire program represented in the form of the tree is known as the activation tree.
- Control stack keeps track of the activation of the procedures.
- Scope of declaration is of two types:
  i) Global scope of declaration: Identifier is defined for the entire program.
  ii) Local scope of declaration: Identifier is defined over a block or a procedure.
- Binding of name is the process in which we bind the name to a particular storage location.
- Activation record exists for each procedure; when the procedure is called activation record is pushed into the control stack, and when completed its execution pops out of the stack.
- General activation record consists of:
  i) Actual parameter
  ii) Returned values
  iii) Control link
  iv) Access link
  v) Saved machine states
  vi) Local data
  vii) Temporaries

- Subdivision of runtime memory is as follows:

Code memory
Static memory
Stack memory
Heap memory

Table 6.3 Runtime Memory

- Storage allocation techniques are of three types:
  i) Static allocation
  ii) Stack allocation
  iii) Heap allocation
- Symbol table is one of the most important data structures used by the compilers to know information about the identifiers and for error correction.
- Common techniques for the implementation of symbol table:
  i) Linear list
  ii) Ordered list
  iii) Trees
  iv) Hash table
- Hash table is one of the most important used data structure that is widely used by the compiler.