Chapter 5
SYMBOL TABLE DESIGN AND RUNTIME STORAGE MANAGEMENT

……

Topics:

Symbol Table Design

Function of symbol Table

Symbol Table Entries

Items stored in Symbol table:

Information used by compiler from Symbol Table

Storing Names in Symbol Table

Usage of Symbol Table Information

Operations of symbol table

Data Structures of Symbol table

Run-time Storage Management

……

5.1 Symbol Table Design

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc. It is built in lexical and syntax analysis phases. The information is collected by the analysis phases (front end) of compiler and is used by synthesis phases (back end) of compiler to generate code. It is used by compiler to achieve compile time efficiency.

It is used by various phases of compiler as follows:

- **Lexical Analysis:** Creates new table entries in the table, example like entries about token.
- **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
- **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type checking) and update it accordingly.
- **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
- **Code Optimization:** Uses information present in symbol table for machine dependent optimization.
- **Target Code generation:** Generates code by using address information of identifier present in the table.

5.1.1 Functions of a Symbol Table

A symbol table may have the following functions depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared or not.
- To determine the scope of a name (scope resolution).
- To access information associated with a given name.
- To add new information with a given name.
- To delete a name or group of names from the table.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.

5.1.2 Symbol Table Entries

Each entry in symbol table is associated with attributes that support compiler in different phases.

Items Stored in Symbol Table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings

- Compiler generated temporaries
- Labels in source languages

Information Used by Compiler from Symbol Table:

- **Name**
  - Name of the identifier
  - May be stored directly or as a pointer to another character string in an associated string table names can be arbitrarily long.
- **Type**
  - Type of the identifier: variable, label, procedure name etc.
  - For variable, its type: basic types, derived types etc.
- **Location**
  - Offset within the program where the identifier is defined
- **Scope**
  - Region of the program where the current definition is valid
- **Other attributes:** array limits, fields of records, parameters, return values etc.

Storing Names in Symbol Table:

There are two types of name representation:

1. **Fixed Length Name:**

   A fixed space for each name is allocated in symbol table. In this types of storage if name is too small then there is wastage of space.

   Example:

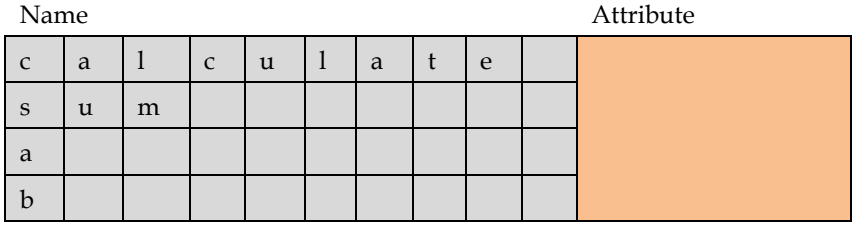

Figure 5.1: Fixed length name

   The name can be referred by pointer to symbol table entry.

2. **Variable Length Name:**

   The amount of space required by string is used to store the names. The name can be stored with the name of starting index and length of each name.

   Example:

Name	**Attribute**

Starting index	length	
0	10	
10	4	
14	2	
16	2	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	l	c	u	l	a	t	e	$	s	u	m	$	a	$	b	$

Figure 5.2: Variable length name

5.1.3 Operations of Symbol Table

The basic operations defined on a symbol table include:

1. **Insert:** This operation is more frequently used by 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 in hand.

   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 about the symbol. The **insert** function takes the symbol and its attributes as arguments and stores the information in the symbol table.

   Example:

   int x;

   should be processed by the compiler as:

   insert(x, int);

2. **Lookup:** This operation is used to search name in the symbol table to determine:
   - The existence of symbol in the table.
   - The declaration of the symbol before it is used.
   - Check whether the name is used in the scope.
   - Initialization of the symbol.
   - Checking whether the name is declared multiple times.

   The format of **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. **Modify:** When name is defined, all information may not be available, may be updated later. This operation is used to modify the information of the names.

4. **Delete:** This is operation is not used frequently but needed sometimes, such as when a procedure body ends.
5. **Allocate:** This operation is used to allocate a new empty symbol table.
6. **Free:** This operation is used to remove all entries and free storage of symbol table
7. **Set_attribute:** This operation is used to associate an attribute with a given entry
8. **Get_attribute:** This operation is use to get an attribute associated with a given entry

5.1.4 Data Structures for Symbol Table

Following are commonly used data structure for implementing symbol table:

1. **List Data Structure:**

   In this method, an array is used to store names and associated information. A pointer **"available"** is maintained at end of all stored records and new names are added in the order as they arrive.

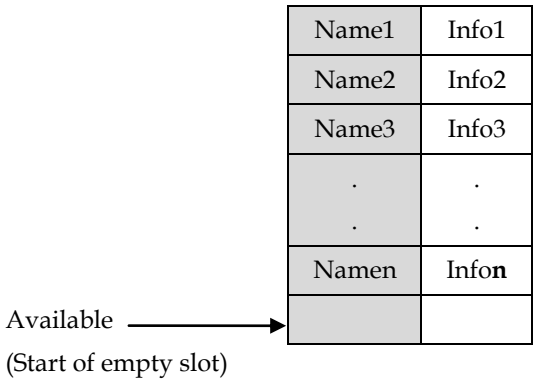

Figure 5.3: List data structure for symbol table

   To search for a new name we start from beginning of list till available pointer and if not found we get an error **"use of undeclared name"**. While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. **"Multiple defined name"**. Insertion is fast O(1), but lookup is slow for large tables – O(n) on average. **Advantage** is that it takes minimum amount of space.

2. **Self Organizing List Data Structure:**

   This implementation is using linked list. A link field is added to each record. Searching of names is done in order pointed by link of link field. A pointer "First" is maintained to point to first record of symbol table. Insertion is fast O(1), but lookup is slow for large table – O(n) on average.

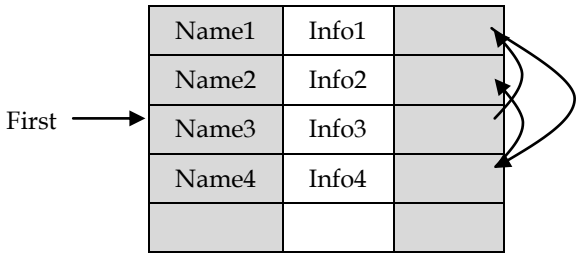

Figure 5.4: Self organizing List data structure for symbol table

The reference to these names can be Name3, Name1, Name4, Name2.

When the name is referenced or created it is moved to the front of the list. The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

Advantages:

- The list organization takes less space.
- Insertion of new symbol is easy.

3. Binary Trees:

When the organization symbol table is by means of binary tree, the node structure will be as follows:

Left child	Symbol	Information	Right child

The left child field stores the address of previous symbol and right child field stores the address of next symbol. The symbol field is used to store the name of symbol. And information field is used to give information about the symbol.

Example:

```
int m, n, p;
int compute(int a, int b, int c)
{
        t = a + b * c;
        return(t);
}
main()
{
        int k;
        k = compute(10, 20, 30);
}
```

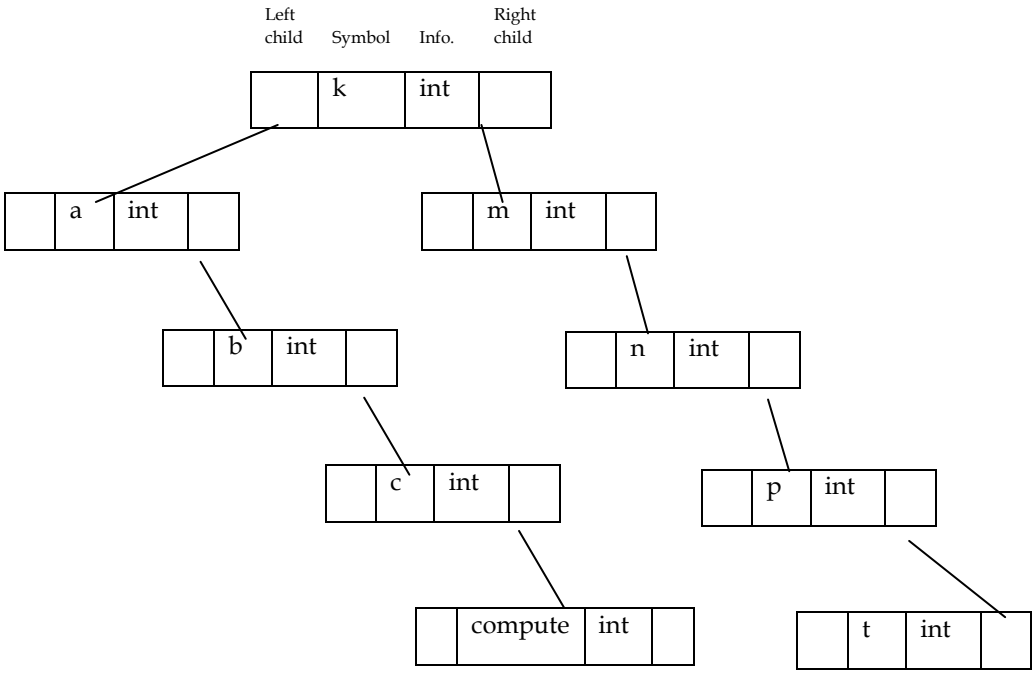

Figure 5.5: Binary tree data structure for symbol table

The binary tree structure is basically a binary search tree in which the value of left node is always less than the value of parent node. Similarly the value of right node is always more or greater than the parent node.

Advantages:

- Insertion of any symbol is efficient.
- Any symbol can be searched efficiently using binary search method.

Disadvantages:

- This structure consumes lot of space in storing left pointer, right pointer and null pointers.

4. Hash Tables:

Hashing is an important technique used to search the records of symbol table. This method is superior to list organization. In hashing scheme two tables are maintained: A Hash Table and Symbol Table. The hash table consist 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 location of name in symbol table.

Example:

```
int m, n, p;
int compute(int a, int b, int c)
{
        t = a + b * c;
        return(t);
}
main()
{
        int k;
        k = compute(10, 20, 30);
}
```

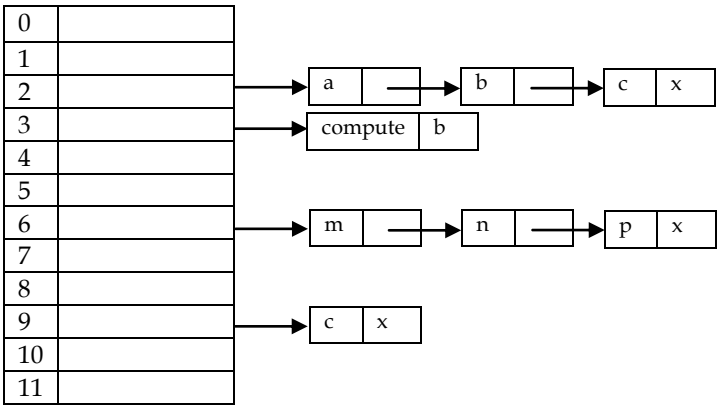

Figure 5.6: Hash table data structure for symbol table

Advantages:

- The advantage of hashing is quick search is possible.

Disadvantages:

- Hashing is complicated to implement.
- Some extra space is required.
- Obtaining scope of variables is very difficult.

5.2 Run-time Storage Management

The information which required during an execution of a procedure is kept in a block of storage called an activation record. "Activation record is a block of memory used for managing information needed by a single execution of a procedure."

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.

The following three-address statements are associated with the run-time allocation and de-allocation of activation records:

1. Call
2. Return
3. Halt
4. Action, a placeholder for other statements

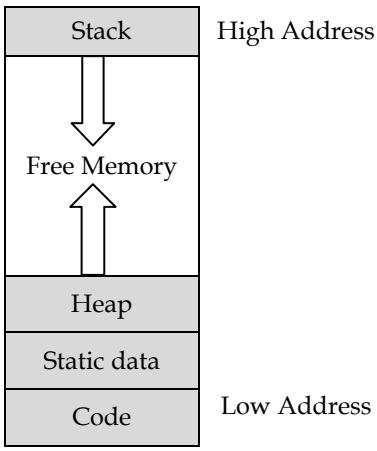

Figure 5.7: Storage allocation

Static Storage Allocation

In static allocation, if memory is created at compile time, memory will be created in the static area and only once. It don't support dynamic data structure i.e. memory is created at compile time and de-allocated after program completion.

Disadvantages

- The drawback with static storage allocation is recursion is not supported.
- Another drawback is size of data should be known at compile time.

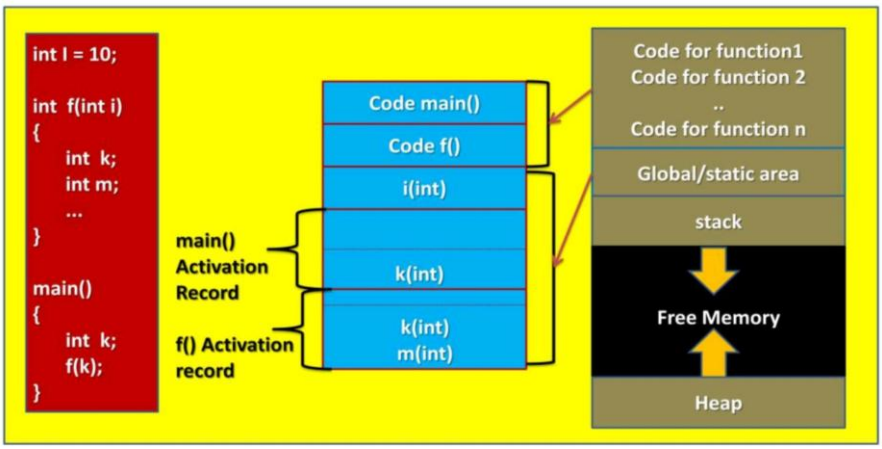

Figure 5.8: Static storage allocation

Stack Storage Allocation

Stack allocation is based on the idea of a control stack.

- A stack is a Last In First Out storage device where new storage is allocated and de-allocated at only one "end", called the top of the stack.
- Storage is organized as a stack and activation records are pushed and popped as activations begin and end, respectively.
- Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when a call is made.
- Furthermore, the values of local are detected when the activation ends. That is, the values are lost because the storage for locals disappears when the activation record is popped.
- At run time, an activation record can be allocated and de-allocated by incrementing and decrementing top of the stack respectively.

Advantages:

- It supports recursion as memory is always allocated on block entry.
- It allows creating data structure dynamically.
- It allows an array declaration like A(I,J), since actual allocation is made only at execution time. The dimension bounds need not be known at compile time.

Disadvantages:

- Memory addressing can be done using pointers and index registers. Hence, this type of allocation is slower than static allocation.

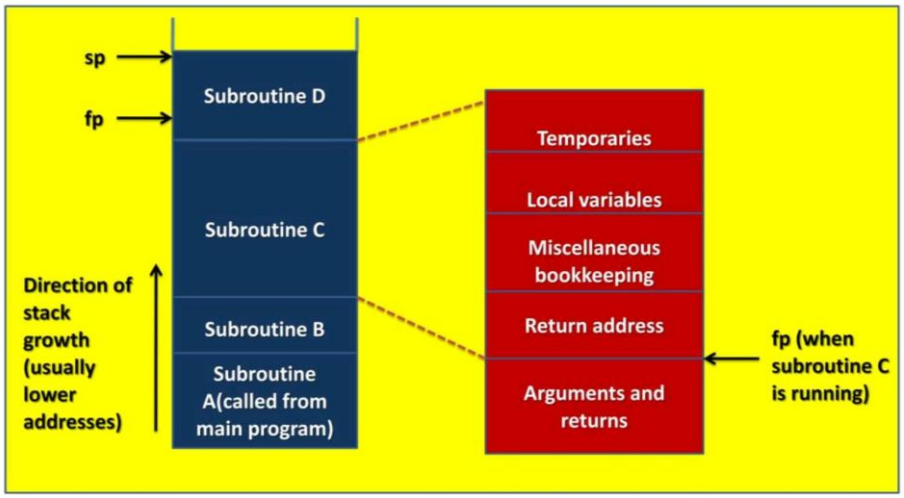

Figure 8.9: Stack storage allocation

Heap Storage Allocation

The de-allocation of activation records need not occur in a last-in first-out fashion, so storage cannot be organized as a stack. Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be de-allocated in any order. So over time the heap will consist of alternate areas that are free and in use. Heap is an alternate for stack.

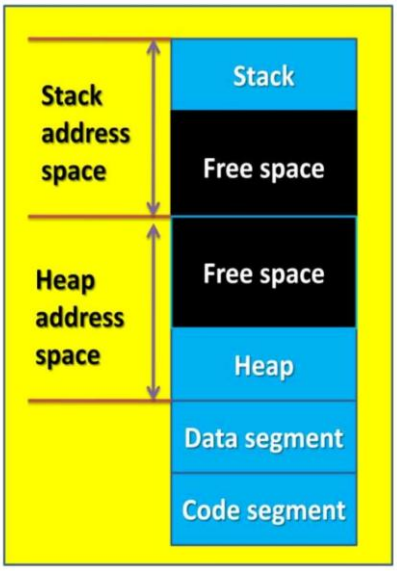

Figure 5.10: Heap storage allocation

Exercise

1. Define

❑❑❑