SYMBOL TABLE:

A symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.

A symbol table is a major data structure used in a compiler which associates attributes with identifiers in a program. A symbol table information is used by the analysis and synthesis phases to verify that used identifiers have been defined (declared), to verify that expressions and assignments are semantically correct (type checking), to generate intermediate or target code.

The basic operations defined on a symbol table include:

- *allocate* – to allocate a new empty symbol table
- *free* – to remove all entries and free the storage of a symbol table
- *insert* – to insert a name in a symbol table and return a pointer to its entry
- *lookup* – to search for a name and return a pointer to its entry
- *set_attribute* – to associate an attribute with a given entry
- *get_attribute* – to get an attribute associated with a given entry

Other operations can be added depending on requirement. For example, a *delete* operation removes a name previously inserted.

Possible Implementations Techniques:

- Unordered list: for a very small set of variables.
- Ordered linear list: insertion is expensive, but implementation is relatively easy
- Binary search tree: O(log n) time per operation for n variables.
- Hash table: most commonly used, and very efficient provided the memory space is adequately larger than the number of variables.

A common implementation technique is to use a hash table. A compiler may use one large symbol table for all symbols or use separated hierarchical symbol tables for different scopes.

There are also trees, linear lists and self-organizing lists which can be used to implement symbol table. It also simplifies the classification of literals in tabular format. The symbol table is accessed by most phases of a compiler, beginning with the lexical analysis to optimization.

Implementation of Symbol Tables:

There are many ways to implement symbol tables, but the most important distinction between these is how scopes are handled. This may be done using a persistent (or functional) data structure, or it may be done using an imperative (or destructively-updated) data structure.

A persistent data structure has the property that no operation on the structure will destroy it. Conceptually, a new modified copy is made of the data structure whenever an operation updates it, hence preserving the old structure unchanged.

In the imperative approach, only one copy of the symbol table exists, so explicit actions are required to store the information needed to restore the symbol table to a previous state. This can be done by using an auxiliary stack. When an update is made, the old binding of a name that is overwritten is recorded (pushed) on the auxiliary stack.

When a new scope is entered, a marker is pushed on the auxiliary stack. When the scope is exited, the bindings on the auxiliary stack (down to the marker) are used to reestablish the old symbol table. The bindings and the marker are popped off the auxiliary stack in the process, returning the auxiliary stack to the state it was in before the scope was entered.

Uses:

An object file will contain a symbol table of the identifiers it contains that are externally visible. During the linking of different object files, a linker will use these symbol tables to resolve any unresolved references.

A symbol table may only exist during the translation process, or it may be embedded in the output of that process for later exploitation, for example, during an interactive debugging session, or as a resource for formatting a diagnostic report during or after execution of a program.

While reverse engineering an executable, many tools refer to the symbol table to check what addresses have been assigned to global variables and known functions. If the symbol table has been stripped or cleaned out before being converted into an executable, tools will find it harder to determine addresses or understand anything about the program.

At that time of accessing variables and allocating memory dynamically, a compiler should perform many works and as such the extended stack model requires the symbol table.

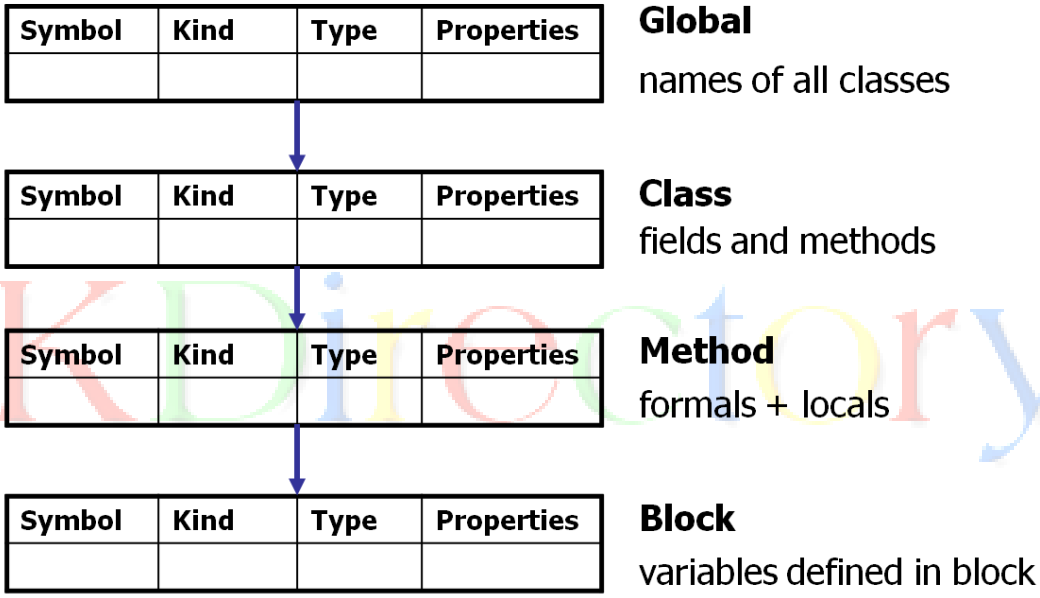
SYMBOL TABLE FORMAT:

Symbol table is an environment that stores information about identifiers, a data structure that captures scope information. Each entry in symbol table contains: Name, Kind, Type and additional properties.

Symbol table contains the following format:

Symbol	Kind	Type	Properties

Scope nesting mirrored hierarchy of symbol tables is given as below:

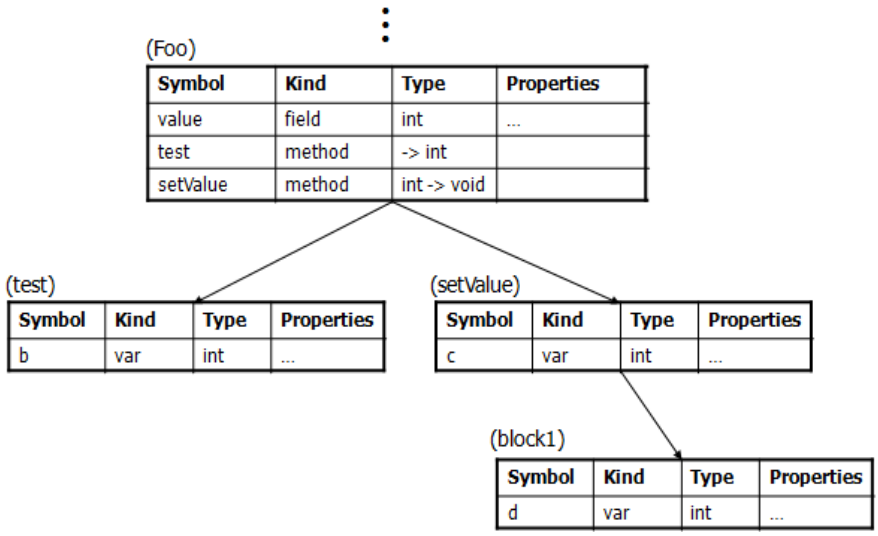


Symbol table example:

Consider the following program:

```
class Foo {
  int value;
  int test() {
    int b = 3;                          } scope of b
    return value + b;
  }
  void setValue(int c) {                                  } scope of value
    value = c;                                  } scope of c
    { int d = c;
block1{   c = c + d;          } scope of d
      value = c;
    }
  }
}

class Bar {
  int value;
  void setValue(int c) {                  } scope of c
       value = c;                                 } scope of value
  }
}
```



TREE STRUCTURE REPRESENTATION OF SCOPE INFORMATION:

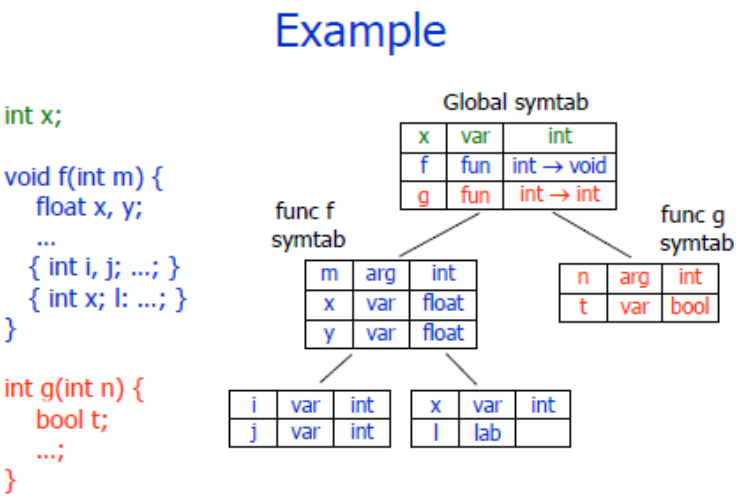
Scope information characterizes the declaration of identifiers and the portions of the program where use of each identifier is allowed, Example identifiers: variables, functions, objects, labels.

Semantic Rules for Scopes:

Main rules regarding scopes are; 1) Use an identifier only if defined in enclosing scope; 2) Do not declare identifiers of the same kind with identical names more than once in the same lexical scope.

When the scope information is presented in hierarchical manner then it forms a tree structure representation. To overcome the drawback of the sequential representation being slow in searching the desired identifier, this representation is been designed.

The representation of scope information in the symbol table is given in the example below:



The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes).

BLOCK STRUCTURES AND NON BLOCK STRUCTURE STORAGE ALLOCATION:

Block structured symbol tables are designed to support the scoping rules of block structured languages.

In general the Storage allocation can be done for two types of data variables:

1) Local Data

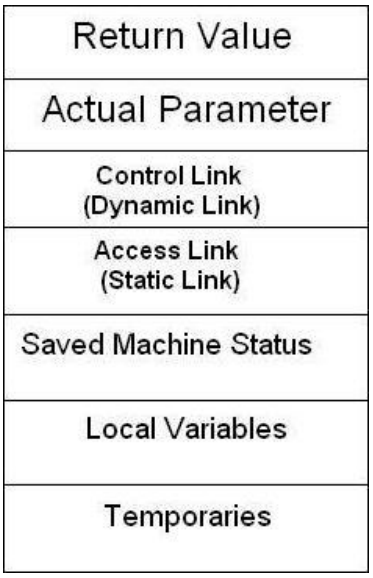
2) Non-Local Data

The local data can be handled using activation record where as non local data can be handled using scope information.

The Block structured storage allocation can be done using static scope or lexical scope. The Non-Block structured storage allocation can be done using dynamic scope.

Activation Record:

Every execution of a procedure is called as Activation. It is a block of memory used for managing information needed by single execution of a procedure. FORTRAN uses static data area to store the activation record. In PASCAL and C, activation record is situated in stack area. Contents of activation record are as shown in figure given as follows:



Actual Parameter: -

It points to activation record of calling procedure.

Saved Machine Status: -

It holds information regarding status of machine before the procedure is called.

Access Link: -

It refers to non local data in other activation record.

Procedure calls and returns are usually managed by a runtime stack called control stack. Each live activation has an activation record (some times called frame) on the control stack, with the root of the activation tree at the bottom.

Static Scope:-

one of the basic reason for scoping is to keep variables in different parts of the program distinct from one another. With static scoping, a variable always refers to its top-level environment.

A language uses static or lexical scope if it is possible to find the scope of a declaration by looking only at the program. A global identifier refers to the identifiers with the name i.e. declared in the closest enclosing scope of the program. It maintains static relationship between the blocks in the program text.

Dynamic scope:-

A global identifier refers to the identifier associated with the most recent activation record. It uses the actual sequence of calls that are executed in the dynamic execution of the program.

Both static and dynamic scope are identical when local variables are concerned. E.g.

```
int x = 1;
function g(z) = x + z;
function f(y) =
{
int x = y + 1;
return g(y*x)
};
f(3);
```

After call to g, static scope x = 1 and dynamic scope x = 4

Static Storage Allocation:-

In static allocation, compiler makes the decision regarding storage allocation by looking only at the program text. Compiler allocates space for all variables (local and global) of all procedures at compile time.

As it does not use stack/heap allocation, there will be no overheads. Variable access is fast since addresses are known at compile time. No recursion.

Dynamic Storage Allocation:

In dynamic storage allocation, decisions, Compiler allocates space only for global variables at compile time. Space for variables of procedures will be allocated at runtime. It uses stack / heap allocation.

E.g. C, C++, Java, FORTRAN 8/9

Variable access is slow (compared to static allocation) since addresses are accessed through the stack / heap pointer. Recursion can be implemented.