

Symbol Tables

A symbol table is an abstract data type (ADT) for storing names encountered in the source program, along with the relevant attributes for those names. As with any ADT we have attributes (data) and operations.

Typical attributes consist of:

- Name
- Type
- Size
- Lifetime
- Scope
- Address
- Value

Typical operations consist of:

- *Initialize* the table to empty.
- *Store* a symbol and its attributes.
- *Find* a symbol.
- *Enter* a new scope level.
- *Leave* a scope level.
- *Remove* a symbol (on leaving a scope).

Symbol Records

How do we represent some of the attributes in our symbol table?

Name

Several alternatives are available for representing the name. Symbol names may be:

1. Fixed length, e.g. 2 bytes (the original BASIC language), 6 bytes (Fortran 66), 32 bytes (ANSI C). The symbol table record can reserve a fixed amount of memory to store the name.
2. Variable length, e.g. Pascal. A pointer to the start of the name is stored in the symbol table record. Some way to determine the length must also be provided, either by storing the length of the name or by storing an end-of-name character with the name.

The variable length alternative requires that the memory region for storing names is managed, either by:

1. using the dynamic memory management functions provided by the OS (e.g. *new* and *malloc*), or:
2. the compiler programmer.

Using dynamic memory management provided by the OS is easier to implement, but will run slower due to OS overhead. Programmer-managed memory will run faster but is harder to program; decisions about the size of the name region must be made. A hybrid approach would be to obtain memory for the name region from the OS and then manage it in the program. This would allow for the expansion of the memory region if necessary.

Type

The data structure needed to represent type information depends on the complexity of allowable types in the language.

1. Languages that allow only simple types can simply use an integer *type code*, e.g. char = 0, int = 1, float = 2, Boolean = 3.
2. Languages with compound or structured types like arrays or ranges in Pascal may need several fields, e.g. C-style arrays would require base type code and max. index fields.

3. Languages that permit user-defined types store a pointer to a type graph representing the type structure.

Lifetime

The lifetime of a variable is the period of time for which the variable has memory allocated. An integer code may be used to represent lifetime. We are familiar with *static* and *automatic* lifetimes. Variables with static lifetimes have memory allocated for the entire duration of the program, while those with automatic lifetime have memory allocated for the duration of a function invocation. The last three items in the list below describe variations on automatic lifetime have been implemented in some languages, so they are listed here.

Possible lifetimes:

1. **Static:** Memory is allocated in a fixed location and size for the duration of the program. Global variables are usually static. In some languages (e.g. Fortran 66) all variables are static.
2. **Semistatic:** Memory is allocated in a fixed size and location in the stack frame for the duration of a function invocation.
3. **Semidynamic:** During a particular function invocation, memory is allocated in a fixed size and location, but the size and location may change from invocation to invocation. Pascal conformant arrays are an example.
4. **Dynamic:** The size and location may be changed at any time. Algol-68 allows the declaration of fully dynamic arrays.

Scope

Languages that support scope rules can store the scope value as an integer value. A typical scheme may be:

- 1 "permanent" symbols such as keywords. Some languages handle keywords in this manner, particularly a language like Fortran, which does not reserve keywords.
- 0 for global symbols
- 1 for symbols defined in the main program
- >1 for nested scopes

Scoped languages should choose a symbol table data structure that supports scope rules, i.e. a variable defined in the current scope must be accessed before a variable in an enclosing scope. A stack organization provides this behavior. New scopes are pushed on the stack, and searching for a symbol begins at the top of the stack. We will look at structures that support scope rules later.

Value

Languages like Pascal and C++ allow declaration of constant variables. Initialization of variables with a value known at compile time is also allowed in most languages. The symbol table can store a binary representation of the value that can be used to generate the initialization during code generation.

An interpreted language must store values in a form that is ready to use for evaluation. Complex user-defined types make representation in a symbol table difficult; consequently allowed data types are restricted in interpreted languages.

Address

The value stored for address depends on the lifetime of the symbol.

Static variables: The address of static variables is fixed and can be computed at compile time. The address is stored as an offset from some known value. For example, SPARC variables are stored in data segments, so we begin assigning address offsets from the start of the segment (address 0).

Semistatic variables: Semistatic variables are allocated storage in a stack frame. Their size and address is fixed relative to the start of the stack frame. An offset from some known point in the stack frame (usually contained in a register called the *frame pointer*) can be computed at compile time.

Dynamic variables: Dynamic variables are allocated memory on the heap. The heap is a region of memory from which chunks can be allocated as needed at run-time. The allocated chunks may be returned and reused in any order (unlike the RTS which grows and shrinks at one end). A pointer to contain the address of the chunk is allocated, which may be in either static memory or in a stack frame.

Semidynamic variables: Semidynamic variables may be allocated either on the RTS or the heap. As for dynamic variables, allocate a pointer to access the memory.

Symbol table data structures

We'll look at several data structures for our table, including arrays, linked lists and trees and hash tables. We'll consider the following factors when evaluating data structures:

1. ease of programming
2. speed of operations:
 - search
 - insert
 - scope deletion (Symbol deletion, which is used in our project, is not a typical operation in a compiler. It has the same efficiency as a search.)
3. size limits
4. scope support
5. popularity

Unsorted array

<p>New symbols are added to the end of the table. Searching for a symbol starts at the end and is linear. The most recent scope is always at the end of the table.</p>	<ol style="list-style-type: none"> 1. easy to program 2. operations: <ul style="list-style-type: none"> • slow search: $O(n)$ • fast insert: $O(1)$ • fast scope deletion $O(1)$ 3. fixed size 4. good scope support 5. medium popularity
--	---

Sorted array

<p>New symbols are inserted in sorted order. Binary search is used.</p> <p>No scope support; how do you handle multiple instances of symbols with the same name?</p> <p>Could improve scope support by:</p> <ul style="list-style-type: none"> • using a sorted table of linked lists, or • using a series of tables for each scope. 	<ol style="list-style-type: none"> 1. moderately easy to program 2. operations: <ul style="list-style-type: none"> • improved search: $O(\log n)$ • slow insert: $O(n)$ • slow scope deletion $O(n)$ 3. fixed size 4. poor scope support 5. medium popularity
--	---

Linear linked list

<p>New records are inserted at the front of the list.</p> <p>Search the entire list for a symbol.</p> <p>Good scope support: the most recently added scope is at the front of the list.</p>	<ol style="list-style-type: none"> 1. moderate programming difficulty 2. operations: <ul style="list-style-type: none"> • slow search: $O(n)$ • fast insert: $O(1)$ • fast scope deletion $O(1)$ 3. no size limit 4. good scope support 5. high popularity
---	--

Balanced search tree

<p>The structure of the tree represents the sorted order of the records.</p> <p>Maintaining balance is important because of the tendency of programmers to use systematic names with common prefixes/suffixes.</p> <p>No scope support; same problems as with sorted arrays. Can use a "forest" of trees, one tree for each scope</p>	<ol style="list-style-type: none"> 1. moderate to high programming difficulty for balanced trees. 2. operations: <ul style="list-style-type: none"> • moderate search: $O(\log n)$ • moderate insert: $O(\log n)$ • scope deletion: implementation dependent $O(1)$ [if using a forest] 3. no size limit 4. poor scope support 5. low popularity
---	--

Closed hash table

<p>An array of symbol table records inserted in a random order. A hash function applied to the name determines the location in the table where the symbol record should be stored. A collision resolution strategy must be employed to deal with names that hash to the same value.</p> <p>No scope support: symbols with the same name hash to the same location.</p>	<ol style="list-style-type: none"> 1. moderate programming difficulty 2. operations: <ul style="list-style-type: none"> • fast search: $O(1)$ • fast insert: $O(1)$ • slow scope deletion $O(n)$ 3. fixed size 4. poor scope support 5. moderate popularity
--	---

Open hash table

<p>An array of pointers to linear linked lists of symbol table records. All symbols that have the same name hash to the same list. Insert new records at the start of the list.</p> <p>Good scope support: symbols in the most recently entered scope are at the front of the list. Scope deletion is slow because all the lists must be checked for records in the deleted scope.</p>	<ol style="list-style-type: none"> 1. high programming difficulty 2. operations: <ul style="list-style-type: none"> • fast search: $O(1)$ • fast insert: $O(1)$ • slow scope deletion $O(n)$ 3. no size limit 4. good scope support 5. high popularity
--	--

Collision resolution strategies for closed hash tables

Collision resolution strategies must work correctly for both insertion of a new symbol and searching for a symbol that may or may not be present in the table. Deletion of a symbol may involve moving other symbols. We will consider search in this discussion only.

Searching for a symbol proceeds as follows:

```
int i = hash(symbol.getName());
if(isEmpty(table[i])
    // symbol is not in table
else if(symbol.getName().equals(table[i].getName()))
    // symbol is found
else
    // compute another value for i and try again
```

Three methods for finding the next location to check are commonly used:

1. Linear probing: try the next array location. Wrap around to the beginning of the array if you reach the end. The problem with this method is that it tends to create *islands* of symbols. A collision will cause the symbol table entry to be placed immediately after the previous one. Further collisions in this group of symbols will keep growing the island, decreasing the efficiency of the search.
2. Rehashing: after a collision, apply a different hash function to find the next location. The problem with this method is that it is hard to devise an unlimited number of good hash functions.
3. Probe step: compute a probe step size k that depends on the original hash location. To compute the next location after a collision, use $(k+i) \bmod \text{TableSize}$. Now collisions in a block of symbols should not create an island, but be part of a chain of symbols with the same step size. In order to ensure all table locations will be searched, the table size must be relatively prime to all possible values of step size. The easiest way to ensure this is to pick a prime table size.

Hash functions

Hash functions must do a good job of randomly calculating an index value given a symbol name. Complicated scrambling functions don't necessarily perform well because they may not consider underlying patterns in seemingly random names. For instance, bytes of ASCII text that can be typed from a keyboard all have a zero high-order bit (a so-called *boring bit*). A good performer is simply to treat the string of characters as a big integer and compute the index value as this integer mod the (prime) table size. The

problem with this technique is that if we allow variable length names, we don't have a built-in mod function for arbitrarily large integers. The solution is to use *compression* to compact the string into a machine-size integer. Compression involves breaking up the symbol name into small chunks, then using either addition or exclusive-OR to compress the chunks into a single value. We may also use bit shifting on each chunk before it is compressed so that boring bits don't overlap.

e.g. the hashpjw function by P. J. Weinberger

```
static final int TABLE_SIZE = 101;

public static int st_hash(String symbol){
    int h = 0;
    int testTop4;
    for(int i = 0; i < symbol.length(); i++){
        h = (h << 4) + (int)symbol.charAt(i);
        testTop4 = h & 0xf0000000;
        if(testTop4 != 0){
            h = h ^ (testTop4 >>> 24);
            h = h ^ testTop4;
        }
    } // for
    return h % TABLE_SIZE;
} // st_hash
```

Java shift and bitwise operators:

<<	left shift
>>>	logical (unsigned) right shift
&	bitwise AND
^	bitwise XOR