Symbol Tables

This document describes, mostly by example, what a stack of symbol tables MIGHT look like. I’ll start with a short introduction to what a symbol table is and then move to an extended example before finishing with a "generic" driver to test a symbol table application.

So a symbol table as used in a compiler is a way to store and lookup information about "names" (character strings) in a program. At this stage of the compiler project we’re not concerned about how and when a symbol (name) is inserted and searched-for in the compiler, we’re just building the symbol table. What we ARE concerned with is:

1) inserting a new symbol into the symbol table, and

2) finding the symbol table information for a symbol (name or character string)

We’re also concerned about the scope of a symbol, so our symbol table needs to consider a scope when inserting and searching for a symbol. Since scopes of program are typically handled in a last-in, first-out order (stack?) we’ll talk about a symbol tables as a stack of data structures, each capable of insertion and lookup.

Ok, that’s the short description, now let’s get to an example. There seems to be some confusion about what a "record" for a symbol might be. So, I’ll start with a few symbols as we might see them in a program.

```c
int i; // a single integer, stored in 4 bytes SOMEWHERE
double d; // a single double floating point value, in 8 bytes
int array[32]; // an array of 32 ints, requiring 128 bytes of storage
```

For this simple example we might decide that we need to store the following information about a symbol, namely its

- name,
- size,
- location

or in pseudo C

```c
struct Symbol
{
    char *name; // character string of the symbol name
    int size; // the number of bytes required
    unsigned int location; // and "address" in memory where the
}
```

And our three records might look like

```c
("i",4,1000), ("d",8,2000), and ("array",128,3000)
```

HOWEVER, note that our 3-field symbol struct will NOT be sufficient throughout the term. There will be more fields that you’ll need to add as time goes by. In fact, there is a least one field that I’m "already" missing, but that’s not the point. The point is that you should write your symbol table code so that it will be easy later on to add new fields to a "Symbol" (in whatever language you choose to use to implement your compiler) without having to make major modifications in either your symbol table OR your compiler.

Now, we want to store those three symbols in a data structure to allow fast insertion and fast lookup. You could choose any of several data structures. I’ll assume a VERY naive hashtable. (It is NOT my intention to discuss finer details of hashtable implementation here. If you’re not sure how to implement a hashtable, go look at your data structures text or do a web search.) I’ll assume what’s typically called "open address hashing" which means that I’ll
have an array of an "appropriate" size and store data in that array based upon a simple hash function. Further more, my hash function will be the (TERRIBLE) choice of corresponding an array element for each of the 26 letters of the alphabet (so I’ll have a tablesie of 26) and we’ll store data based upon the first letter of the name of the name of the symbol. So, here is a declaration (in C) for a single hashtable.

    struct Symbol *table[26];

The following page shows what a hashtable holding our three "Symbols" would look like.
<table>
<thead>
<tr>
<th>letter</th>
<th>index</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>0</td>
<td>ptr to struct for array</td>
</tr>
<tr>
<td>'b'</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>'c'</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'d'</td>
<td>3</td>
<td>ptr to struct for d</td>
</tr>
<tr>
<td>'e'</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>'f'</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>'g'</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>'h'</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>'i'</td>
<td>8</td>
<td>ptr to struct for i</td>
</tr>
<tr>
<td>'j'</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>'k'</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>'m'</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>'n'</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>'p'</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>'q'</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>'r'</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>'s'</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>'t'</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>'u'</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>'v'</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>'w'</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>'x'</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>'y'</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>'z'</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
So, that would be ONE scope’s worth of "Symbols". But you’d need to maintain a stack of such hashtables (or whatever your storage is for a scope).

Now, to (hopefully) bring this together, I’ll provide a "driver" to test "my" stack of symbol tables. It might look something like this.

--------  driver.c  -------------------------------------------
#include "myDefs.h"     // where I define a Symbol struct, and a SymbolTable
                       // where a SymbolTable is a stack of hashtables, or
                       // whatever we use to store a scope’s worth of Symbols

// Declarations of symbol table functions implemented in some other C file
Symbol *createASymbol(char *, int size, unsigned int location);
void Insert(Symbol *, char *);
Symbol *Search(char *);
void Enter_new_scope();         // must assume a global ptr to stack of scopes
void Leave_current_scope();     // must assume a global ptr to stack of scopes

int main()
{
    Symbol *sym;

    Enter_new_scope();
    sym = createASymbol("i",4,1000);
    Insert(sym,"i");
    sym = createASymbol("d",8,2000);
    Insert(sym,"d");
    sym = createASymbol("array",128,3000);
    Insert(sym,"array");

    // now let’s add a new scope onto our Symbol Table (meaning perhaps we saw // an ‘{’ in a C program.

    Enter_new_scope();

    sym = createASymbol("z",4,4000);
        // oops, I "forgot" to insert z into the symbol table.
        // That will lead to an "error" later.
    sym = createASymbol("x",4,5000);
    Insert(sym,"x");
    sym = createASymbol("y",8,6000);
    Insert(sym,"y");
    sym = createASymbol("i",8,7000);
    Insert(sym,"i");

    // And now for some searches. Of course I’d include some printf statements // for each search to determine if my (your?) symbol table routines were // working.

    sym = Search("array")       // should return a ptr to the Symbol for array, // whose "location" is 3000
                                // should return a NULL ptr, since there is no // Symbol for z that has been inserted into // the symbol table
    sym = Search("i")           // should return a NULL ptr, since there is no // Symbol for z that has been inserted into // the symbol table
                       // whose "location" is 7000, since that scope // is now at the top of the stack.

    Leave_current_scope();
sym = Search("i")  // should return a ptr to the Symbol for i,
// whose "location" is 1000, since that scope
// is now at the top of the stack.

return 0;
}