# Symbol table

In computer science, a **symbol table** is a data structure used by a language translator such as a compiler or interpreter, where each identifier (or symbol) in a program's source code is associated with information relating to its declaration or appearance in the source. In other words, the entries of a symbol table store the information related to the entry's corresponding symbol.

## Contents



## Background

A symbol table may only exist in memory during the translation process, or it may be embedded in the output of the translation, such as in an ABI object file for later use. For example, it might be used during an interactive debugging session, or as a resource for formatting a diagnostic report during or after execution of a program.[1]

## Description

The minimum information contained in a symbol table used by a translator includes the symbol's name and its location or address. For a compiler targeting a platform with a concept of relocatability, it will also contain relocatability attributes (absolute, relocatable, etc.) and needed relocation information for relocatable symbols. Symbol tables for high-level programming languages may store the symbol's type: string, integer, floating-point, etc., its size, and its dimensions and its bounds. Not all of this information is included in the output file, but may be provided for use in debugging. In many cases, the symbol's cross-reference information is stored with or linked to the symbol table. Most compilers print some or all of this information in symbol table and cross-reference listings at the end of translation.

## Implementation

Numerous data structures are available for implementing tables. Trees, linear lists and self-organizing lists can all be used to implement a symbol table. The symbol table is accessed by most phases of a compiler, beginning with lexical analysis, and continuing through optimization.

A compiler may use one large symbol table for all symbols or use separated, hierarchical symbol tables for different scopes. For example, in a scoped language such as Algol or PL/I a symbol "p" can be declared separately in several procedures, perhaps with different attributes. The scope of each declaration is the section of the program in which references to "p" resolve to that declaration. Each declaration represents a unique identifier "p". The symbol table must have some means of differentiating references to the different "p"s.

A common data structure used to implement symbol tables is the hash table. The time for searching in hash tables is independent of the number of elements stored in the table, so it is efficient for a large number of elements. It also simplifies the classification of literals in tabular format.

As the lexical analyser spends a great proportion of its time looking up the symbol table, this activity has a crucial effect on the overall speed of the compiler. A symbol table must be organised in such a way that entries can be found as quickly as possible. Hash tables are usually used to organise a symbol table, where the keyword or identifier is 'hashed' to produce an array subscript. Collisions are inevitable in a hash table, and a common way of handling them is to store the synonym in the next available free space in the table.

# Applications

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 identify and resolve these symbol references. Usually all undefined external symbols will be searched for in one or more object libraries. If a module is found that defines that symbol it is linked with together with the first object file, and any undefined external identifiers are added to the list of identifiers to be looked up. This process continues until all external references have been resolved. It is an error if one or more remains unresolved at the end of the process.

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.

# Example

Consider the following program written in C:

```c
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1; i <= count; i++)
        sum += bar((double) i);
    return sum;
}
```

A C compiler that parses this code will contain at least the following symbol table entries:

| Symbol name | Type | Scope |
|---|---|---|
| `bar` | function, double | extern |
| `x` | double | function parameter |
| `foo` | function, double | global |
| `count` | int | function parameter |
| `sum` | double | block local |
| `i` | int | for-loop statement |

In addition, the symbol table will also contain entries generated by the compiler for intermediate expression values (e.g., the expression that casts the `i` loop variable into a `double`, and the return value of the call to function `bar()`), statement labels, and so forth.

## Example: SysV ABI

An example of a symbol table can be found in the SysV Application Binary Interface (ABI) specification, which mandates how symbols are to be laid out in a binary file, so that different compilers, linkers and loaders can all consistently find and work with the symbols in a compiled object.

The SysV ABI is implemented in the GNU binutils' nm utility. This format uses a sorted memory address field, a "symbol type" field, and a symbol identifier (called "Name").[2]

The symbol types in the SysV ABI (and nm's output) indicate the nature of each entry in the symbol table. Each symbol type is represented by a single character. For example, symbol table entries representing initialized data are denoted by the character "d" and symbol table entries for functions have the symbol type "t" (because executable code is located in the *text* section of an object file). Additionally, the capitalization of the symbol type indicates the type of linkage: lower-case letters indicate the symbol is local and upper-case indicates external (global) linkage.

## Example: the Python symbol table

The Python programming language includes extensive support for creating and manipulating symbol tables.[3] Properties that can be queried include whether a given symbol is a free variable or a bound variable, whether it is block scope or global scope, whether it is imported, and what namespace it belongs to.

## Example: Dynamic symbol tables

Some programming languages allow the symbol table to be manipulated at run-time, so that symbols can be added at any time. Racket is an example of such a language[4].

Both the LISP and the Scheme programming languages allow arbitrary, generic properties to be associated with each symbol.[5]

The Prolog programming language is essentially a symbol-table manipulation language; symbols are called *atoms*, and the relationships between symbols can be reasoned over. Similarly, OpenCog provides a dynamic symbol table, called the *atomspace*, which is used for knowledge representation.

## See also

- Debug symbol

# References


1. Nguyen, Binh (2004). *Linux Dictionary* (https://books.google.com/books?id=vdZWBQAAQBAJ). p. 1482. Retrieved Apr 14, 2018.
2. "nm" (http://sourceware.org/binutils/docs-2.17/binutils/nm.html#nm). *sourceware.org*. Retrieved May 30, 2020.
3. symtable — Python documentation (https://docs.python.org/3/library/symtable.html)
4. Symbols - Racket Documentation (https://docs.racket-lang.org/reference/symbols.html)
5. Symbols - Guile Documentation (https://www.gnu.org/software/guile/manual/html_node/Symbols.html)


Example table: SysV ABI

| Address | Type | Name |
|---|---|---|
| 00000020 | a | T_BIT |
| 00000040 | a | F_BIT |
| 00000080 | a | I_BIT |
| 20000004 | t | irqvec |
| 20000008 | t | fiqvec |
| 2000000c | t | InitReset |
| 20000018 | T | _main |
| 20000024 | t | End |
| 20000030 | T | AT91F_US3_CfgPIO_useB |
| 2000005c | t | AT91F_PIO_CfgPeriph |
| 200000b0 | T | main |
| 20000120 | T | AT91F_DBGU_Printk |
| 20000190 | t | AT91F_US_TxReady |
| 200001c0 | t | AT91F_US_PutChar |
| 200001f8 | T | AT91F_SpuriousHandler |
| 20000214 | T | AT91F_DataAbort |
| 20000230 | T | AT91F_FetchAbort |
| 2000024c | T | AT91F_Undef |
| 20000268 | T | AT91F_UndefHandler |
| 20000284 | T | AT91F_LowLevelInit |
| 200002e0 | t | AT91F_DBGU_CfgPIO |
| 2000030c | t | AT91F_PIO_CfgPeriph |
| 20000360 | t | AT91F_US_Configure |
| 200003dc | t | AT91F_US_SetBaudrate |
| 2000041c | t | AT91F_US_Baudrate |
| 200004ec | t | AT91F_US_SetTimeguard |
| 2000051c | t | AT91F_PDC_Open |
| 2000059c | t | AT91F_PDC_DisableRx |
| 200005c8 | t | AT91F_PDC_DisableTx |
| 200005f4 | t | AT91F_PDC_SetNextTx |
| 20000638 | t | AT91F_PDC_SetNextRx |
| 2000067c | t | AT91F_PDC_SetTx |
| 200006c0 | t | AT91F_PDC_SetRx |
| 20000704 | t | AT91F_PDC_EnableRx |
| 20000730 | t | AT91F_PDC_EnableTx |

| 2000075c | t | AT91F_US_EnableTx |
|---|---|---|
| 20000788 | T | __aeabi_uidiv |
| 20000788 | T | __udivsi3 |
| 20000884 | T | __aeabi_uidivmod |
| 2000089c | T | __aeabi_idiv0 |
| 2000089c | T | __aeabi_ldiv0 |
| 2000089c | T | __div0 |
| 200009a0 | D | _data |
| 200009a0 | A | _etext |
| 200009a0 | D | holaamigosh |
| 200009a4 | A | __bss_end__ |
| 200009a4 | A | __bss_start |
| 200009a4 | A | __bss_start__ |
| 200009a4 | A | _edata |
| 200009a4 | A | _end |