# Symbol table

From Wikipedia, the free encyclopedia

This article **does not cite any references or sources**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(November 2012)*

In computer science, 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,



## Implementation [edit]

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 a 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.

## Uses [edit]

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**.

## Example [edit]

Consider the following program written in 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;
}
```

Wikipedia The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Deutsch
Español
فارسی
Français
Nederlands
日本語
Português
Русский
中文
Edit links

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.

As another example, the symbol table of a small program is listed below. The table itself was generated using the GNU binutils' nm utility. There is one data symbol, (noted by the "D" type), and many functions (self defined as well as from the standard library). The first column is where the symbol is located in the memory, the second is "The symbol type" and the third is the name of the symbol. By passing suitable parameters, the symbol table was made to sort on basis of address.

**Example table**

| 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 |

## See also [edit]

- Debug symbol