NAME

Devel::Size – Perl extension for finding the memory usage of Perl variables

SYNOPSIS

```
use Devel::Size qw(size total_size);

my $size = size("A string");

my @foo = (1, 2, 3, 4, 5);
my $other_size = size(\@foo);

my $foo = {a => [1, 2, 3],
    b => {a => [1, 3, 4]}
       };
my $total_size = total_size($foo);
```

DESCRIPTION

This module figures out the real size of Perl variables in bytes, as accurately as possible.

Call functions with a reference to the variable you want the size of. If the variable is a plain scalar it returns the size of this scalar. If the variable is a hash or an array, use a reference when calling.

FUNCTIONS

size($ref)

The `size` function returns the amount of memory the variable returns. If the variable is a hash or an array, it only reports the amount used by the structure, *not* the contents.

total_size($ref)

The `total_size` function will traverse the variable and look at the sizes of contents. Any references contained in the variable will also be followed, so this function can be used to get the total size of a multidimensional data structure. At the moment there is no way to get the size of an array or a hash and its elements without using this function.

EXPORT

None but default, but optionally `size` and `total_size`.

UNDERSTANDING MEMORY ALLOCATION

Please note that the following discussion of memory allocation in perl is based on the perl 5.8.0 sources. While this is generally applicable to all versions of perl, some of the gory details are omitted. It also makes some presumptions on how your system memory allocator works so, while it will be generally correct, it may not exactly reflect your system. (Generally the only issue is the size of the constant values we'll talk about, not their existence)

The C library

It's important first to understand how your OS and libraries handle memory. When the perl interpreter needs some memory, it asks the C runtime library for it, using the `malloc()` call. `malloc` has one parameter, the size of the memory allocation you want, and returns a pointer to that memory. `malloc` also makes sure that the pointer it returns to you is properly aligned. When you're done with the memory you hand it back to the library with the `free()` call. `free` has one parameter, the pointer that `malloc` returned. There are a couple of interesting ramifications to this.

Because malloc has to return an aligned pointer, it will round up the memory allocation to make sure that the memory it returns is aligned right. What that alignment is depends on your CPU, OS, and compiler settings, but things are generally aligned to either a 4 or 8 byte boundary. That means that if you ask for 1 byte, `malloc` will silently round up to either 4 or 8 bytes, though it doesn't tell the program making the request, so the extra memory can't be used.

Since `free` isn't given the size of the memory chunk you're freeing, it has to track it another way. Most libraries do this by tacking on a length field just before the memory it hands to your program. (It's put before the beginning rather than after the end because it's less likely to get mangled by program bugs) This

size field is the size of your platform integer, Generally either 4 or 8 bytes.

So, if you asked for 1 byte, malloc would build something like this:

```
+------------------+
| 4 byte length    |
+------------------+ <----- the pointer malloc returns
| your 1 byte      |
+------------------+
| 3 bytes padding  |
+------------------+
```

As you can see, you asked for 1 byte but `malloc` used 8. If your integers were 8 bytes rather than 4, `malloc` would have used 16 bytes to satisfy your 1 byte request.

The C memory allocation system also keeps a list of free memory chunks, so it can recycle freed memory. For performance reasons, some C memory allocation systems put a limit to the number of free segments that are on the free list, or only search through a small number of memory chunks waiting to be recycled before just allocating more memory from the system.

The memory allocation system tries to keep as few chunks on the free list as possible. It does this by trying to notice if there are two adjacent chunks of memory on the free list and, if there are, coalescing them into a single larger chunk. This works pretty well, but there are ways to have a lot of memory on the free list yet still not have anything that can be allocated. If a program allocates one million eight-byte chunks, for example, then frees every other chunk, there will be four million bytes of memory on the free list, but none of that memory can be handed out to satisfy a request for 10 bytes. This is what's referred to as a fragmented free list, and can be one reason why your program could have a lot of free memory yet still not be able to allocate more, or have a huge process size and still have almost no memory actually allocated to the program running.

Perl

Perl's memory allocation scheme is a bit convoluted, and more complex than can really be addressed here, but there is one common spot where Perl's memory allocation is unintuitive, and that's for hash keys.

When you have a hash, each entry has a structure that points to the key and the value for that entry. The value is just a pointer to the scalar in the entry, and doesn't take up any special amount of memory. The key structure holds the hash value for the key, the key length, and the key string. (The entry and key structures are separate so perl can potentially share keys across multiple hashes)

The entry structure has three pointers in it, and takes up either 12 or 24 bytes, depending on whether you're on a 32 bit or 64 bit system. Since these structures are of fixed size, perl can keep a big pool of them internally (generally called an arena) so it doesn't have to allocate memory for each one.

The key structure, though, is of variable length because the key string is of variable length, so perl has to ask the system for a memory allocation for each key. The base size of this structure is 8 or 16 bytes (once again, depending on whether you're on a 32 bit or 64 bit system) plus the string length plus two bytes.

Since this memory has to be allocated from the system there's the malloc size-field overhead (4 or 8 bytes) plus the alignment bytes (0 to 7, depending on your system and the key length) that get added on to the chunk perl requests. If the key is only 1 character, and you're on a 32 bit system, the allocation will be 16 bytes. If the key is 7 characters then the allocation is 24 bytes on a 32 bit system. If you're on a 64 bit system the numbers get even larger.

DANGERS

Since version 0.72, Devel::Size uses a new pointer tracking mechanism that consumes far less memory than was previously the case. It does this by using a bit vector where 1 bit represents each 4− or 8−byte aligned pointer (32− or 64−bit platform dependent) that could exist. Further, it segments that bit vector and only allocates each chunk when an address is seen within that chunk. Since version 0.73, chunks are allocated in blocks of 2**16 bits (ie 8K), accessed via a 256−way tree. The tree is 2 levels deep on a 32 bit system, 6 levels deep on a 64 bit system. This avoids having make any assumptions about address layout on 64 bit systems or trade offs about sizes to allocate. It assumes that the addresses of allocated pointers are

reasonably contiguous, so that relevant parts of the tree stay in the CPU cache.

Besides saving a lot of memory, this change means that Devel::Size runs significantly faster than previous versions.

Messages: texts originating from this module.

Errors

"Devel::Size: Unknown variable type"

The thing (or something contained within it) that you gave to **total_size()** was unrecognisable as a Perl entity.

warnings

These messages warn you that for some types, the sizes calculated may not include everything that could be associated with those types. The differences are usually insignificant for most uses of this module.

These may be disabled by setting

```
$Devel::Size::warn = 0
```

"Devel::Size: Calculated sizes for CVs are incomplete"
"Devel::Size: Calculated sizes for FMs are incomplete"
"Devel::Size: Calculated sizes for compiled regexes are incompatible, and probably always will be"

New warnings since 0.72

Devel::Size has always been vulnerable to trapping when traversing Perl's internal data structures, if it encounters uninitialised (dangling) pointers.

MSVC provides exception handling able to deal with this possibility, and when built with MSVC Devel::Size will now attempt to ignore (or log) them and continue. These messages are mainly of interest to Devel::Size and core developers, and so are disabled by default.

They may be enabled by setting

```
$Devel::Size::dangle = 0
```

"Devel::Size: Can't determine class of operator OPx_XXXX, assuming BASEOP\n"
"Devel::Size: Encountered bad magic at: 0xXXXXXXXX"
"Devel::Size: Encountered dangling pointer in opcode at: 0xXXXXXXXX"
"Devel::Size: Encountered invalid pointer: 0xXXXXXXXX"

BUGS

Doesn't currently walk all the bits for code refs, formats, and IO. Those throw a warning, but a minimum size for them is returned.

Devel::Size only counts the memory that perl actually allocates. It doesn't count 'dark' memory — memory that is lost due to fragmented free lists, allocation alignments, or C library overhead.

AUTHOR

Dan Sugalski dan@sidhe.org

Small portion taken from the B module as shipped with perl 5.6.2.

Previously maintained by Tels <http://bloodgate.com>

New pointer tracking & exception handling for 0.72 by BrowserUK

Currently maintained by Nicholas Clark

COPYRIGHT

SEE ALSO

perl (1), Devel::Size::Report.