

**NAME**

talloc - hierarchical reference counted memory pool system with destructors

**SYNOPSIS**

```
#include <talloc.h>
```

**DESCRIPTION**

If you are used to talloc from Samba3 then please read this carefully, as talloc has changed a lot.

The new talloc is a hierarchical, reference counted memory pool system with destructors. Quite a mouthful really, but not too bad once you get used to it.

Perhaps the biggest change from Samba3 is that there is no distinction between a "talloc context" and a "talloc pointer". Any pointer returned from talloc() is itself a valid talloc context. This means you can do this:

```
struct foo *X = talloc(mem_ctx, struct foo);
X->name = talloc_strdup(X, "foo");
```

and the pointer X->name would be a "child" of the talloc context X which is itself a child of mem\_ctx. So if you do talloc\_free(mem\_ctx) then it is all destroyed, whereas if you do talloc\_free(X) then just X and X->name are destroyed, and if you do talloc\_free(X->name) then just the name element of X is destroyed.

If you think about this, then what this effectively gives you is an n-ary tree, where you can free any part of the tree with talloc\_free().

If you find this confusing, then I suggest you run the testsuite program to watch talloc in action. You may also like to add your own tests to testsuite.c to clarify how some particular situation is handled.

**TALLOC API**

The following is a complete guide to the talloc API. Read it all at least twice.

**(type \*)talloc(const void \*ctx, type);**

The talloc() macro is the core of the talloc library. It takes a memory *ctx* and a *type*, and returns a pointer to a new area of memory of the given *type*.

The returned pointer is itself a talloc context, so you can use it as the *ctx* argument to more calls to talloc() if you wish.

The returned pointer is a "child" of the supplied context. This means that if you `talloc_free()` the `ctx` then the new child disappears as well. Alternatively you can free just the child.

The `ctx` argument to `talloc()` can be `NULL`, in which case a new top level context is created.

**`void *talloc_size(const void *ctx, size_t size);`**

The function `talloc_size()` should be used when you don't have a convenient type to pass to `talloc()`. Unlike `talloc()`, it is not type safe (as it returns a `void *`), so you are on your own for type checking.

**`(typeof(ptr)) talloc_ptrtype(const void *ctx, ptr);`**

The `talloc_ptrtype()` macro should be used when you have a pointer and want to allocate memory to point at with this pointer. When compiling with `gcc >= 3` it is typesafe. Note this is a wrapper of `talloc_size()` and `talloc_get_name()` will return the current location in the source file. and not the type.

**`int talloc_free(void *ptr);`**

The `talloc_free()` function frees a piece of `talloc` memory, and all its children. You can call `talloc_free()` on any pointer returned by `talloc()`.

The return value of `talloc_free()` indicates success or failure, with 0 returned for success and -1 for failure. The only possible failure condition is if `ptr` had a destructor attached to it and the destructor returned -1. See "`talloc_set_destructor()`" for details on destructors.

If this pointer has an additional parent when `talloc_free()` is called then the memory is not actually released, but instead the most recently established parent is destroyed. See "`talloc_reference()`" for details on establishing additional parents.

For more control on which parent is removed, see "`talloc_unlink()`".

`talloc_free()` operates recursively on its children.

From the 2.0 version of `talloc`, as a special case, `talloc_free()` is refused on pointers that have more than one parent, as `talloc` would have no way of knowing which parent should be removed. To free a pointer that has more than one parent please use `talloc_unlink()`.

To help you find problems in your code caused by this behaviour, if you do try and free a pointer with more than one parent then the `talloc` logging function will be called to give output like this:

```
ERROR: talloc_free with references at some_dir/source/foo.c:123
reference at some_dir/source/other.c:325
reference at some_dir/source/third.c:121
```

Please see the documentation for `talloc_set_log_fn()` and `talloc_set_log_stderr()` for more information on `talloc` logging functions.

**void \*talloc\_reference(const void \*ctx, const void \*ptr);**

The `talloc_reference()` function makes *ctx* an additional parent of *ptr*.

The return value of `talloc_reference()` is always the original pointer *ptr*, unless `talloc` ran out of memory in creating the reference in which case it will return `NULL` (each additional reference consumes around 48 bytes of memory on intel x86 platforms).

If *ptr* is `NULL`, then the function is a no-op, and simply returns `NULL`.

After creating a reference you can free it in one of the following ways:

⊕

can `talloc_free()` any parent of the original pointer. That will reduce the number of parents of this pointer by 1, and will cause this pointer to be freed if it runs out of parents.

⊕

can `talloc_free()` the pointer itself if it has at maximum one parent. This behaviour has been changed since the release of version 2.0. Further informations in the description of "`talloc_free`".

For more control on which parent to remove, see "`talloc_unlink()`".

**int talloc\_unlink(const void \*ctx, void \*ptr);**

The `talloc_unlink()` function removes a specific parent from *ptr*. The *ctx* passed must either be a context used in `talloc_reference()` with this pointer, or must be a direct parent of *ptr*.

Note that if the parent has already been removed using `talloc_free()` then this function will fail and will return -1. Likewise, if *ptr* is `NULL`, then the function will make no modifications and return -1.

Usually you can just use `talloc_free()` instead of `talloc_unlink()`, but sometimes it is useful to have the additional control on which parent is removed.

**void talloc\_set\_destructor(const void \*ptr, int (\*destructor)(void \*));**

The function `talloc_set_destructor()` sets the *destructor* for the pointer *ptr*. A *destructor* is a function that is called when the memory used by a pointer is about to be released. The destructor receives *ptr* as an argument, and should return 0 for success and -1 for failure.

The *destructor* can do anything it wants to, including freeing other pieces of memory. A common use

for destructors is to clean up operating system resources (such as open file descriptors) contained in the structure the destructor is placed on.

You can only place one destructor on a pointer. If you need more than one destructor then you can create a zero-length child of the pointer and place an additional destructor on that.

To remove a destructor call `talloc_set_destructor()` with `NULL` for the destructor.

If your destructor attempts to `talloc_free()` the pointer that it is the destructor for then `talloc_free()` will return `-1` and the free will be ignored. This would be a pointless operation anyway, as the destructor is only called when the memory is just about to go away.

**int talloc\_increase\_ref\_count(const void \*ptr);**

The `talloc_increase_ref_count(ptr)` function is exactly equivalent to:

```
talloc_reference(NULL, ptr);
```

You can use either syntax, depending on which you think is clearer in your code.

It returns `0` on success and `-1` on failure.

**size\_t talloc\_reference\_count(const void \*ptr);**

Return the number of references to the pointer.

**void talloc\_set\_name(const void \*ptr, const char \*fmt, ...);**

Each `talloc` pointer has a "name". The name is used principally for debugging purposes, although it is also possible to set and get the name on a pointer in as a way of "marking" pointers in your code.

The main use for names on pointer is for "talloc reports". See "`talloc_report_depth_cb()`", "`talloc_report_depth_file()`", "`talloc_report()`" "`talloc_report()`" and "`talloc_report_full()`" for details. Also see "`talloc_enable_leak_report()`" and "`talloc_enable_leak_report_full()`".

The `talloc_set_name()` function allocates memory as a child of the pointer. It is logically equivalent to:

```
talloc_set_name_const(ptr, talloc_asprintf(ptr, fmt, ...));
```

Note that multiple calls to `talloc_set_name()` will allocate more memory without releasing the name. All of the memory is released when the `ptr` is freed using `talloc_free()`.

**void talloc\_set\_name\_const(const void \*ptr, const char \*name);**

The function `talloc_set_name_const()` is just like `talloc_set_name()`, but it takes a string constant, and is much faster. It is extensively used by the "auto naming" macros, such as `talloc_p()`.

This function does not allocate any memory. It just copies the supplied pointer into the internal representation of the talloc ptr. This means you must not pass a *name* pointer to memory that will disappear before *ptr* is freed with `talloc_free()`.

**void \*talloc\_named(const void \*ctx, size\_t size, const char \*fmt, ...);**

The `talloc_named()` function creates a named talloc pointer. It is equivalent to:

```
ptr = talloc_size(ctx, size);
talloc_set_name(ptr, fmt, ....);
```

**void \*talloc\_named\_const(const void \*ctx, size\_t size, const char \*name);**

This is equivalent to:

```
ptr = talloc_size(ctx, size);
talloc_set_name_const(ptr, name);
```

**const char \*talloc\_get\_name(const void \*ptr);**

This returns the current name for the given talloc pointer, *ptr*. See "talloc\_set\_name()" for details.

**void \*talloc\_init(const char \*fmt, ...);**

This function creates a zero length named talloc context as a top level context. It is equivalent to:

```
talloc_named(NULL, 0, fmt, ...);
```

**void \*talloc\_new(void \*ctx);**

This is a utility macro that creates a new memory context hanging off an existing context, automatically naming it "talloc\_new: \_\_location\_\_" where `__location__` is the source line it is called from. It is particularly useful for creating a new temporary working context.

**(type \*)talloc\_realloc(const void \*ctx, void \*ptr, type, count);**

The `talloc_realloc()` macro changes the size of a talloc pointer. It has the following equivalences:

```
talloc_realloc(ctx, NULL, type, 1) ==> talloc(ctx, type);
talloc_realloc(ctx, ptr, type, 0) ==> talloc_free(ptr);
```

The *ctx* argument is only used if *ptr* is not NULL, otherwise it is ignored.

`talloc_realloc()` returns the new pointer, or `NULL` on failure. The call will fail either due to a lack of memory, or because the pointer has more than one parent (see `"talloc_reference()"`).

**void \*talloc\_realloc\_size(const void \*ctx, void \*ptr, size\_t size);**

the `talloc_realloc_size()` function is useful when the type is not known so the type-safe `talloc_realloc()` cannot be used.

**TYPE \*talloc\_steal(const void \*new\_ctx, const TYPE \*ptr);**

The `talloc_steal()` function changes the parent context of a `talloc` pointer. It is typically used when the context that the pointer is currently a child of is going to be freed and you wish to keep the memory for a longer time.

The `talloc_steal()` function returns the pointer that you pass it. It does not have any failure modes.

It is possible to produce loops in the parent/child relationship if you are not careful with `talloc_steal()`. No guarantees are provided as to your sanity or the safety of your data if you do this.

Note that if you try and call `talloc_steal()` on a pointer that has more than one parent then the result is ambiguous. `Talloc` will choose to remove the parent that is currently indicated by `talloc_parent()` and replace it with the chosen parent. You will also get a message like this via the `talloc` logging functions:

```
WARNING: talloc_steal with references at some_dir/source/foo.c:123
reference at some_dir/source/other.c:325
reference at some_dir/source/third.c:121
```

To unambiguously change the parent of a pointer please see the function `"talloc_reparent()"`. See the `talloc_set_log_fn()` documentation for more information on `talloc` logging.

**TYPE \*talloc\_reparent(const void \*old\_parent, const void \*new\_parent, const TYPE \*ptr);**

The `talloc_reparent()` function changes the parent context of a `talloc` pointer. It is typically used when the context that the pointer is currently a child of is going to be freed and you wish to keep the memory for a longer time.

The `talloc_reparent()` function returns the pointer that you pass it. It does not have any failure modes.

The difference between `talloc_reparent()` and `talloc_steal()` is that `talloc_reparent()` can specify which parent you wish to change. This is useful when a pointer has multiple parents via references.

**TYPE \*talloc\_move(const void \*new\_ctx, TYPE \*\*ptr);**

The `talloc_move()` function is a wrapper around `talloc_steal()` which zeros the source pointer after the move. This avoids a potential source of bugs where a programmer leaves a pointer in two structures, and uses the pointer from the old structure after it has been moved to a new one.

**size\_t talloc\_total\_size(const void \*ptr);**

The `talloc_total_size()` function returns the total size in bytes used by this pointer and all child pointers. Mostly useful for debugging.

Passing NULL is allowed, but it will only give a meaningful result if `talloc_enable_leak_report()` or `talloc_enable_leak_report_full()` has been called.

**size\_t talloc\_total\_blocks(const void \*ptr);**

The `talloc_total_blocks()` function returns the total memory block count used by this pointer and all child pointers. Mostly useful for debugging.

Passing NULL is allowed, but it will only give a meaningful result if `talloc_enable_leak_report()` or `talloc_enable_leak_report_full()` has been called.

**void talloc\_report(const void \*ptr, FILE \*f);**

The `talloc_report()` function prints a summary report of all memory used by *ptr*. One line of report is printed for each immediate child of *ptr*, showing the total memory and number of blocks used by that child.

You can pass NULL for the pointer, in which case a report is printed for the top level memory context, but only if `talloc_enable_leak_report()` or `talloc_enable_leak_report_full()` has been called.

**void talloc\_report\_full(const void \*ptr, FILE \*f);**

This provides a more detailed report than `talloc_report()`. It will recursively print the entire tree of memory referenced by the pointer. References in the tree are shown by giving the name of the pointer that is referenced.

You can pass NULL for the pointer, in which case a report is printed for the top level memory context, but only if `talloc_enable_leak_report()` or `talloc_enable_leak_report_full()` has been called.

**void talloc\_report\_depth\_cb(const void \*ptr, int depth, int max\_depth,  
void (\*callback)(const void  
\*ptr, int depth, int max\_depth, int is\_ref, void \*priv),  
void \*priv);**

This provides a more flexible reports than `talloc_report()`. It will recursively call the callback for the entire tree of memory referenced by the pointer. References in the tree are passed with `is_ref = 1` and the pointer that is referenced.

You can pass `NULL` for the pointer, in which case a report is printed for the top level memory context, but only if `talloc_enable_leak_report()` or `talloc_enable_leak_report_full()` has been called.

The recursion is stopped when `depth >= max_depth`. `max_depth = -1` means only stop at leaf nodes.

**`void talloc_report_depth_file(const void *ptr, int depth, int max_depth, FILE *f);`**

This provides a more flexible reports than `talloc_report()`. It will let you specify the depth and `max_depth`.

**`void talloc_enable_leak_report(void);`**

This enables calling of `talloc_report(NULL, stderr)` when the program exits. In Samba4 this is enabled by using the `--leak-report` command line option.

For it to be useful, this function must be called before any other `talloc` function as it establishes a "null context" that acts as the top of the tree. If you don't call this function first then passing `NULL` to `talloc_report()` or `talloc_report_full()` won't give you the full tree printout.

Here is a typical `talloc` report:

```
talloc report on 'null_context' (total 267 bytes in 15 blocks)
libcli/auth/spnego_parse.c:55 contains 31 bytes in 2 blocks
libcli/auth/spnego_parse.c:55 contains 31 bytes in 2 blocks
iconv(UTF8,CP850)          contains 42 bytes in 2 blocks
libcli/auth/spnego_parse.c:55 contains 31 bytes in 2 blocks
iconv(CP850,UTF8)         contains 42 bytes in 2 blocks
iconv(UTF8,UTF-16LE)      contains 45 bytes in 2 blocks
iconv(UTF-16LE,UTF8)      contains 45 bytes in 2 blocks
```

**`void talloc_enable_leak_report_full(void);`**

This enables calling of `talloc_report_full(NULL, stderr)` when the program exits. In Samba4 this is enabled by using the `--leak-report-full` command line option.

For it to be useful, this function must be called before any other `talloc` function as it establishes a "null



context" that acts as the top of the tree. If you don't call this function first then passing NULL to `talloc_report()` or `talloc_report_full()` won't give you the full tree printout.

Here is a typical full report:

```
full talloc report on 'root' (total 18 bytes in 8 blocks)
p1      contains 18 bytes in 7 blocks (ref 0)
  r1    contains 13 bytes in 2 blocks (ref 0)
    reference to: p2
  p2    contains 1 bytes in 1 blocks (ref 1)
  x3    contains 1 bytes in 1 blocks (ref 0)
  x2    contains 1 bytes in 1 blocks (ref 0)
  x1    contains 1 bytes in 1 blocks (ref 0)
```

```
(type *)talloc_zero(const void *ctx, type);
```

The `talloc_zero()` macro is equivalent to:

```
ptr = talloc(ctx, type);
if (ptr) memset(ptr, 0, sizeof(type));
```

```
void *talloc_zero_size(const void *ctx, size_t size)
```

The `talloc_zero_size()` function is useful when you don't have a known type.

```
void *talloc_memdup(const void *ctx, const void *p, size_t size);
```

The `talloc_memdup()` function is equivalent to:

```
ptr = talloc_size(ctx, size);
if (ptr) memcpy(ptr, p, size);
```

```
char *talloc_strdup(const void *ctx, const char *p);
```

The `talloc_strdup()` function is equivalent to:

```
ptr = talloc_size(ctx, strlen(p)+1);
if (ptr) memcpy(ptr, p, strlen(p)+1);
```

This function sets the name of the new pointer to the passed string. This is equivalent to:

```
talloc_set_name_const(ptr, ptr)
```

**char \*talloc\_strndup(const void \*t, const char \*p, size\_t n);**

The `talloc_strndup()` function is the `talloc` equivalent of the C library function `strndup(3)`.

This function sets the name of the new pointer to the passed string. This is equivalent to:

```
talloc_set_name_const(ptr, ptr)
```

**char \*talloc\_vasprintf(const void \*t, const char \*fmt, va\_list ap);**

The `talloc_vasprintf()` function is the `talloc` equivalent of the C library function `vasprintf(3)`.

This function sets the name of the new pointer to the new string. This is equivalent to:

```
talloc_set_name_const(ptr, ptr)
```

**char \*talloc\_asprintf(const void \*t, const char \*fmt, ...);**

The `talloc_asprintf()` function is the `talloc` equivalent of the C library function `asprintf(3)`.

This function sets the name of the new pointer to the passed string. This is equivalent to:

```
talloc_set_name_const(ptr, ptr)
```

**char \*talloc\_asprintf\_append(char \*s, const char \*fmt, ...);**

The `talloc_asprintf_append()` function appends the given formatted string to the given string.

This function sets the name of the new pointer to the new string. This is equivalent to:

```
talloc_set_name_const(ptr, ptr)
```

**(type \*)talloc\_array(const void \*ctx, type, unsigned int count);**

The `talloc_array()` macro is equivalent to:

```
(type *)talloc_size(ctx, sizeof(type) * count);
```

except that it provides integer overflow protection for the multiply, returning `NULL` if the multiply overflows.

**void \*talloc\_array\_size(const void \*ctx, size\_t size, unsigned int count);**

The `talloc_array_size()` function is useful when the type is not known. It operates in the same way as `talloc_array()`, but takes a size instead of a type.

**(typeof(ptr)) talloc\_array\_ptrtype(const void \*ctx, ptr, unsigned int count);**

The `talloc_array_ptrtype()` macro should be used when you have a pointer to an array and want to allocate memory of an array to point at with this pointer. When compiling with `gcc >= 3` it is typesafe. Note this is a wrapper of `talloc_array_size()` and `talloc_get_name()` will return the current location in the source file. and not the type.

**void \*talloc\_realloc\_fn(const void \*ctx, void \*ptr, size\_t size)**

This is a non-macro version of `talloc_realloc()`, which is useful as libraries sometimes want a `realloc` function pointer. A `realloc(3)` implementation encapsulates the functionality of `malloc(3)`, `free(3)` and `realloc(3)` in one call, which is why it is useful to be able to pass around a single function pointer.

**void \*talloc\_autofree\_context(void);**

This is a handy utility function that returns a `talloc` context which will be automatically freed on program exit. This can be used to reduce the noise in memory leak reports.

**void \*talloc\_check\_name(const void \*ptr, const char \*name);**

This function checks if a pointer has the specified *name*. If it does then the pointer is returned. If it doesn't then `NULL` is returned.

**(type \*)talloc\_get\_type(const void \*ptr, type);**

This macro allows you to do type checking on `talloc` pointers. It is particularly useful for `void*` private pointers. It is equivalent to this:

```
(type *)talloc_check_name(ptr, #type)
```

**talloc\_set\_type(const void \*ptr, type);**

This macro allows you to force the name of a pointer to be a particular *type*. This can be used in conjunction with `talloc_get_type()` to do type checking on `void*` pointers.

It is equivalent to this:

```
talloc_set_name_const(ptr, #type)
```

**talloc\_set\_log\_fn(void (\*log\_fn)(const char \*message));**

This function sets a logging function that `talloc` will use for warnings and errors. By default `talloc` will not print any warnings or errors.

**talloc\_set\_log\_stderr(void);**

This sets the `talloc` log function to write log messages to `stderr`

**PERFORMANCE**

All the additional features of `talloc(3)` over `malloc(3)` do come at a price. We have a simple performance test in Samba4 that measures `talloc()` versus `malloc()` performance, and it seems that `talloc()` is about 10% slower than `malloc()` on my x86 Debian Linux box. For Samba, the great reduction in code complexity that we get by using `talloc` makes this worthwhile, especially as the total overhead of `talloc/malloc` in Samba is already quite small.

**SEE ALSO**

`malloc(3)`, `strndup(3)`, `vasprintf(3)`, `asprintf(3)`, <http://talloc.samba.org/>

**AUTHOR**

The original Samba software and related utilities were created by Andrew Tridgell. Samba is now developed by the Samba Team as an Open Source project similar to the way the Linux kernel is developed.

**COPYRIGHT/LICENSE**

Copyright (C) Andrew Tridgell 2004

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, see <http://www.gnu.org/licenses/>.