1. Config.

⟨ config.h 1 ⟩ ≡

```
#define TALLOC_BUILD_VERSION_MAJOR 2
#define TALLOC_BUILD_VERSION_MINOR 3
#define TALLOC_BUILD_VERSION_RELEASE 3
```

2. Header.

⟨talloc.h 2⟩ ≡

```
#ifndef _TALLOC_H_
#define _TALLOC_H_
```

See also sections 3, 4, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29, and 30.

3. Unix SMB/CIFS implementation. Samba temporary memory allocation functions

Copyright (C) Andrew Tridgell 2004-2005 Copyright (C) Stefan Metzmacher 2006

** NOTE! The following LGPL license applies to the talloc ** library. This does NOT imply that all of Samba is released ** under the LGPL

This library 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 library 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 Lesser General Public License for more details.

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

⟨talloc.h 2⟩ +≡

```
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#ifdef __cplusplus
  extern "C" {
#endif     /* for old gcc releases that don't have the feature test macro __has_attribute */
#ifndef __has_attribute
#define __has_attribute(x) 0
#endif
#ifndef _PUBLIC_
#if __has_attribute (visibility)
#define _PUBLIC___attribute__ ((visibility("default")))
#else
#define _PUBLIC_
#endif
#endif
```

4. The talloc API

talloc is a hierarchical, reference counted memory pool system with destructors. It is the core memory allocator used in Samba.

⟨talloc.h 2⟩ +≡

```
#define TALLOC_VERSION_MAJOR 2
#define TALLOC_VERSION_MINOR 3
  _PUBLIC_
      int talloc_version_major(void); _PUBLIC_
          int talloc_version_minor(void);     /* This is mostly useful only for testing */
          _PUBLIC_
              int talloc_test_get_magic(void);
```

5.

Define a talloc parent type

As talloc is a hierarchial memory allocator, every talloc chunk is a potential parent to other talloc chunks. So defining a separate type for a talloc chunk is not strictly necessary. TALLOC_CTX is defined nevertheless, as it provides an indicator for function arguments. You will frequently write code like

⟨ sample.c 5 ⟩ ≡

```
struct foo *foo_create(TALLOC_CTX * mem_ctx) { struct foo *result;
    result = talloc(mem_ctx, struct foo);
    if (result ≡ Λ) return Λ;
    ... initialize foo
        ... return result; }
```

6. In this type of allocating functions it is handy to have a general TALLOC_CTX type to indicate which parent to put allocated structures on.

⟨ talloc.h 2 ⟩ +≡

```
typedef void TALLOC_CTX;
```

7. this uses a little trick to allow `__LINE__` to be stringified

⟨ talloc.h 2 ⟩ +≡

```
#ifndef __location__
#define __TALLOC_STRING_LINE1__(s)#s
#define __TALLOC_STRING_LINE2__(s)__TALLOC_STRING_LINE1__ (s)
#define __TALLOC_STRING_LINE3____TALLOC_STRING_LINE2__ (__LINE__)
#define __location____FILE__":" __TALLOC_STRING_LINE3__
#endif
#ifndef TALLOC_DEPRECATED
#define TALLOC_DEPRECATED 0
#endif
```

8.

⟨ talloc.h 2 ⟩ +≡

```
#ifndef PRINTF_ATTRIBUTE
#if __has_attribute(format) ∨ (__GNUC__ ≥ 3)    /** Use gcc attribute to check printf fns. a1 is the
        1-based index of * the parameter containing the format, and a2 the index of the first * argument.
        Note that some gcc 2.x versions don't handle this * properly **/
#define PRINTF_ATTRIBUTE(a1, a2)__attribute__ ((format(__printf__, a1, a2)))
#else
#define PRINTF_ATTRIBUTE (a1, a2)
#endif
#endif
#ifndef _DEPRECATED_
#if __has_attribute(deprecated) ∨ (__GNUC__ ≥ 3)
#define _DEPRECATED___attribute__ ((deprecated))
#else
#define _DEPRECATED_
#endif
#endif
```

9.

⟨ `talloc.h` 2 ⟩ +≡ /∗ * * @brief Create a new talloc context. * * The talloc() macro is the core of the talloc library. It takes a memory * context 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 * context 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 context then the new child disappears as well. * Alternatively you can free just the child. * * @param[in] ctx A talloc context to create a new reference on or NULL to * create a new top level context. * * @param[in] type The type of memory to allocate. * * @return A type casted talloc context or NULL on error. * * @code * **unsigned int** $*a$, $*b$; * * $a =$ *talloc*(Λ, **unsigned int**); * $b =$ *talloc*(a, **unsigned int**); * @endcode * * @see *talloc_zero* * @see *talloc_array* * @see *talloc_steal* * @see *talloc_free* ∗/

\#**define** *talloc*(*ctx*, *type*) (*type* ∗) *talloc_named_const*(*ctx*, **sizeof** (*type*), #*type*)

`_PUBLIC_`

void ∗*_talloc*(**const void** ∗*context*, **size_t** *size*);

10.

⟨ `talloc.h` 2 ⟩ +≡

/∗ * * @brief Create a new top level talloc context. * * This function creates a zero length named talloc context as a top level * context. It is equivalent to: * * @code * *talloc_named*(Λ, 0, *fmt*, . . .); * @endcode * @param[in] fmt Format string for the name. * * @param[in] ... Additional printf-style arguments. * * @return The allocated memory chunk, NULL on error. * * @see *talloc_named*( ) ∗/

`_PUBLIC_`

void ∗*talloc_init*(**const char** ∗*fmt*, . . .)`PRINTF_ATTRIBUTE`(1, 2);

11. Free a chunk of talloc memory.

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. A possible failure condition is if the pointer had a destructor attached to it and the destructor returned -1. See *talloc_set_destructor*( ) for details on destructors. Likewise, if `"ptr"` is NULL, then the function will make no modifications and return -1.

From version 2.0 and onwards, as a special case, *talloc_free*( ) is refused on pointers that have more than one parent associated, as talloc would have no way of knowing which parent should be removed. This is different from older versions in the sense that always the reference to the most recently established parent has been destroyed. Hence 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.

If `TALLOC_FREE_FILL` environment variable is set, the memory occupied by the context is filled with the value of this variable. The value should be a numeric representation of the character you want to use.

talloc_free( ) operates recursively on its children.

@param[in] ptr The chunk to be freed.

@return Returns 0 on success and -1 on error. A possible failure condition is if the pointer had a destructor attached to it and the destructor returned -1. Likewise, if ”ptr” is NULL, then the function will make no modifications and returns -1.

Example:

unsigned int $*a$, $*b$; $a =$ *talloc*(Λ, **unsigned int**); $b =$ *talloc*(a, **unsigned int**);

talloc_free(a); Frees a and b

@see *talloc_set_destructor*( ) @see *talloc_unlink*( )

⟨ `talloc.h` 2 ⟩ +≡

\#**define** *talloc_free*(*ctx*) *_talloc_free* (*ctx*, *__location__*)

`_PUBLIC_`

int *_talloc_free*(**void** $*ptr$, **const char** $*location$);

12. Free a talloc chunk's children.

The function walks along the list of all children of a talloc context and *talloc_free*( )s only the children, not the context itself.

A NULL argument is handled as no-op.

@param[in] ptr The chunk that you want to free the children of (NULL is allowed too)

⟨ `talloc.h` 2 ⟩ +≡

`_PUBLIC_`

void *talloc_free_children*(**void** $*ptr$);

13. Assign a destructor function to be called when a chunk is freed.

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 the pointer 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.

@param[in] ptr The talloc chunk to add a destructor to.

@param[in] destructor The destructor function to be called. NULL to remove it.

Example:

```
⟨example.c 13⟩ ≡
  static int destroy_fd(int *fd)
  {
    close(*fd);
    return 0;
  }
  int *open_file(const char *filename)
  {
    int *fd = talloc(Λ, int);
    *fd = open(filename, O_RDONLY);
    if (*fd < 0) {
      talloc_free(fd);
      return Λ;
    }    /* Whenever they free this, we close the file. */
    talloc_set_destructor(fd, destroy_fd);
    return fd;
  }
```

14.

@see *talloc*( ) @see *talloc_free*( )

15.

⟨ talloc.h 2 ⟩ +≡ /** * @brief Change a talloc chunk's parent. * * 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. * * To make the changed hierarchy less error-prone, you might consider to use * *talloc_move*( ). * * 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. * * @param[in] *new_ctx* The new parent context. * * @param[in] ptr The talloc chunk to move. * * @return Returns the pointer that you pass it. It does not have * any failure modes. * * @note 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. */

```
  /* try to make talloc_set_destructor( ) and talloc_steal( ) type safe, if we have a recent gcc */
#if (__GNUC__ ≥ 3)
#define _TALLOC_TYPEOF(ptr)__typeof__ (ptr)
#define talloc_set_destructor(ptr, function) do
  {
    int(*_talloc_destructor_fn)(_TALLOC_TYPEOF(ptr)) = (function);
    _talloc_set_destructor((ptr), (int(*)(void *))_talloc_destructor_fn);
  }
  while (0)
    /* this extremely strange macro is to avoid some braindamaged warning stupidity in gcc 4.1.x */
#define talloc_steal(ctx, ptr) (
  {
    _TALLOC_TYPEOF(ptr)__talloc_steal_ret = (_TALLOC_TYPEOF(ptr))_talloc_steal_loc((ctx), (ptr),
        __location__);
    __talloc_steal_ret;
  }
  )
#else /* __GNUC__ ≥ 3 */
#define talloc_set_destructor(ptr, function)_talloc_set_destructor ((ptr), (int(*)(void *))(function))
#define _TALLOC_TYPEOF(ptr) void *
#define talloc_steal(ctx, ptr)(_TALLOC_TYPEOF(ptr))_talloc_steal_loc ((ctx), (ptr), __location__)
#endif /* __GNUC__ ≥ 3 */
      _PUBLIC_
          void _talloc_set_destructor(const void *ptr, int(*_destructor)(void *)); _PUBLIC_
              void *_talloc_steal_loc(const void *new_ctx, const void *ptr, const char *location);
```

16.

⟨talloc.h 2⟩ +≡ /* * * @brief Assign a name to a talloc chunk. * * 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*( ) 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*, ...)); * * @param[in] ptr The talloc chunk to assign a name to. * * @param[in] fmt Format string for the name. * * @param[in] ... Add printf-style additional arguments. * * @return The assigned name, NULL on error. * * @note 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*( ). */

```
_PUBLIC_
    const char *talloc_set_name(const void *ptr, const char *fmt, ...)PRINTF_ATTRIBUTE(2, 3);
#ifdef DOXYGEN
```

/* * * @brief Change a talloc chunk's parent. * * This function has the same effect as *talloc_steal*( ), and additionally sets * the source pointer to NULL. You would use it like this: * **struct** *foo* *X = *talloc*(*tmp_ctx*, **struct** *foo*); **struct** *foo* *Y; Y = *talloc_move*(*new_ctx*, &X); * * @param[in] *new_ctx* The new parent context. * * @param[in] pptr Pointer to a pointer to the talloc chunk to move. * * @return The pointer to the talloc chunk that moved. * It does not have any failure modes. * */

```
    _PUBLIC_
        void *talloc_move(const void *new_ctx, void **pptr);
#else
#define talloc_move(ctx, pptr)(_TALLOC_TYPEOF(*(pptr)))_talloc_move ((ctx), (void *)(pptr))
        _PUBLIC_
            void *_talloc_move(const void *new_ctx, const void *pptr);
#endif
```

/* * * @brief Assign a name to a talloc chunk. * * The function 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 the ptr * is freed with *talloc_free*( ). * * @param[in] ptr The talloc chunk to assign a name to. * * @param[in] name Format string for the name. */

```
_PUBLIC_
    void talloc_set_name_const(const void *ptr, const char *name);
```

/* * * @brief Create a named talloc chunk. * * The *talloc_named*( ) function creates a named talloc pointer. It is * equivalent to: * * *ptr* = *talloc_size*(*context*, *size*); * *talloc_set_name* (*ptr*, *fmt*, ...) ; * * @param[in] context The talloc context to hang the result off. * * @param[in] size Number of char's that you want to allocate. * * @param[in] fmt Format string for the name. * * @param[in] ... Additional printf-style arguments. * * @return The allocated memory chunk, NULL on error. * * @see *talloc_set_name*( ) */

```
_PUBLIC_
    void *talloc_named(const void *context, size_t size, const char
        *fmt, ...)PRINTF_ATTRIBUTE(3, 4);
```

/* * * @brief Basic routine to allocate a chunk of memory. * * This is equivalent to: * * *ptr* = *talloc_size*(*context*, *size*); * *talloc_set_name_const*(*ptr*, *name*); * * @param[in] context The parent context. * * @param[in] size The number of char's that we want to allocate. * * @param[in] name The name the talloc block has. * * @return The allocated memory chunk, NULL on error. */

```
_PUBLIC_
```

void ∗*talloc_named_const*(**const void** ∗*context*, **size_t** *size*, **const char** ∗*name*);

#ifdef DOXYGEN /∗* * @brief Untyped allocation. * * The function 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. * * Best to use *talloc*( ) or *talloc_array*( ) instead. * * @param[in] ctx The talloc context to hang the result off. * * @param[in] size Number of char's that you want to allocate. * * @return The allocated memory chunk, NULL on error. * * Example: * @code ***void** ∗*mem* = *talloc_size*(Λ, 100); * @endcode ∗/

PUBLIC

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

#else

#define *talloc_size*(*ctx*, *size*)*talloc_named_const* (*ctx*, *size*, *__location__*)

#endif

#ifdef DOXYGEN /∗* * @brief Allocate into a typed pointer. * * 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. * * @param[in] ctx The talloc context to hang the result off. * * @param[in] type The pointer you want to assign the result to. * * @return The properly casted allocated memory chunk, NULL on * error. * * Example: * @code ***unsigned int** ∗*a* = *talloc_ptrtype*(Λ, *a*); * @endcode ∗/

PUBLIC

void ∗*talloc_ptrtype*(**const void** ∗*ctx*, #*type*);

#else

#define *talloc_ptrtype*(*ctx*, *ptr*)(_TALLOC_TYPEOF(*ptr*))*talloc_size* (*ctx*, **sizeof** (∗(*ptr*)))

#endif

#ifdef DOXYGEN /∗* * @brief Allocate a new 0-sized talloc chunk. * * 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. * * @param[in] ctx The talloc parent context. * * @return A new talloc chunk, NULL on error. ∗/

PUBLIC

void ∗*talloc_new*(**const void** ∗*ctx*);

#else

#define *talloc_new*(*ctx*)*talloc_named_const* (*ctx*, 0, "talloc_new:␣" *__location__*)

#endif

#ifdef DOXYGEN

/∗* * @brief Allocate a 0-initizialized structure. * * The macro is equivalent to: * * @code * ptr = talloc(ctx, type); * if (ptr) memset(ptr, 0, sizeof(type)); * @endcode * * @param[in] ctx The talloc context to hang the result off. * * @param[in] type The type that we want to allocate. * * @return Pointer to a piece of memory, properly cast to 'type *', * NULL on error. * * Example: * @code ***unsigned int** ∗*a*, ∗*b*; **a* = *talloc_zero*(Λ, **unsigned int**); **b* = *talloc_zero*(*a*, **unsigned int**); * @endcode * * @see

talloc( ) * @see *talloc_zero_size*( ) * @see *talloc_zero_array*( ) */
PUBLIC
void **talloc_zero*(**const void** **ctx*, #*type*);
/** * @brief Allocate untyped, 0-initialized memory. * * @param[in] ctx The talloc context to hang the result off. * * @param[in] size Number of char's that you want to allocate. * * @return The allocated memory chunk. */
PUBLIC
void **talloc_zero_size*(**const void** **ctx*, **size_t** *size*);
#else
#define *talloc_zero*(*ctx*, *type*) (*type* *) *_talloc_zero*(*ctx*, **sizeof** (*type*), #*type*)
#define *talloc_zero_size*(*ctx*, *size*) *_talloc_zero* (*ctx*, *size*, *__location__*)
PUBLIC
void **_talloc_zero*(**const void** **ctx*, **size_t** *size*, **const char** **name*);
#endif /** * @brief Return the name of a talloc chunk. * * @param[in] ptr The talloc chunk. * * @return The current name for the given talloc pointer. * * @see *talloc_set_name*( ) */
PUBLIC
const char **talloc_get_name*(**const void** **ptr*); /** * @brief Verify that a talloc chunk carries a specified name. * * This function checks if a pointer has the specified name. If it does * then the pointer is returned. * * @param[in] ptr The talloc chunk to check. * * @param[in] name The name to check against. * * @return The pointer if the name matches, NULL if it doesn't. */
PUBLIC
void **talloc_check_name*(**const void** **ptr*, **const char** **name*);
/** * @brief Get the parent chunk of a pointer. * * @param[in] ptr The talloc pointer to inspect. * * @return The talloc parent of ptr, NULL on error. */
PUBLIC
void **talloc_parent*(**const void** **ptr*); /** * @brief Get a talloc chunk's parent name. * * @param[in] ptr The talloc pointer to inspect. * * @return The name of ptr's parent chunk. */
PUBLIC
const char **talloc_parent_name*(**const void** **ptr*); /** * @brief Get the total size of

a talloc chunk including its children. * * The 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. * * @param[in] ptr The talloc chunk. * * @return The total size. ∗/

`_PUBLIC_` **size_t** *talloc_total_size*(**const void** ∗*ptr*);

/∗* * @brief Get the number of talloc chunks hanging off a chunk. * * 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. * * @param[in] ptr The talloc chunk. * * @return The total size. ∗/

`_PUBLIC_` **size_t** *talloc_total_blocks*(**const void** ∗*ptr*);

#**ifdef** `DOXYGEN` /∗* * @brief Duplicate a memory area into a talloc chunk. * * The function is equivalent to: * * @code *ptr = *talloc_size*(*ctx*, *size*); * **if** (*ptr*) *memcpy*(*ptr*, *p*, *size*); * @endcode * *

@param[in] t The talloc context to hang the result off. * * @param[in] p The memory chunk you want to duplicate. * * @param[in] size Number of char's that you want copy. * * @return The allocated memory chunk. * * @see *talloc_size*( ) ∗/
PUBLIC **void** ∗*talloc_memdup*(**con**[redacted]**t** **void** ∗*t*, **const** **void** ∗*p*, **size_t** *size*);

#else
#define *talloc_memdup*(*t*, *p*, *size*) *_talloc_memdup* (*t*, *p*, *size*, *__location__*)

PUBLIC **void** ∗*_talloc_memdup*(**con**[redacted] **void** ∗*t*, **const** **void** ∗*p*, **size_t** *size*, **const** **char** ∗*name*);

#endif
#ifdef DOXYGEN /∗* * @brief Assign a type to a talloc chunk. * * This macro allows you to force the name of a pointer to be of a particular * type. This can be used in conjunction with *talloc_get_type*[redacted]) to do type * checking on void*

pointers. * * It is equivalent to this: * * @code *$talloc\_set\_name\_cons$... * @endcode * * @param[in] ptr The talloc chunk to assign the type to. * * @param[in] type The type to assign. */

`_PUBLIC_` **void** $talloc\_set\_type$(**co**... **char** $*ptr$, #typ[redacted]); /* * * @brief Get a typed point[redacted]r out of a talloc point[redacted]r. * * This macro allows you to do type checking on talloc pointers. It is * particularly useful for void* private pointers. It is equivalent to * this: * * @code *(*type* *) *talloc_check_name* * @endcode * * @param[[redacted]] ptr The talloc point[redacted]r to check. * * @param[[redacted]] type The type to check again[redacted]. * * @return The properly casted point[redacted]r given by ptr, NULL on

error.
*/
type *
talloc_get_type(**con[redacted]t void** *ptr*, #*type*);

#else
#define *talloc_set_type*(*ptr*, *type*)*talloc_set_name_const* (*ptr*, #*type*)
#define *talloc_get_type*(*ptr*, *type*) (*type* *) *talloc_check_name*(*ptr*, #*type*)
#endif
#ifdef `DOXYGEN` /* * * @brief Safely turn a void pointer into a typed pointer. * * This macro is used together with *talloc*(*mem_ctx*, **stru[redacted]** *foo*). If you had to * assign the talloc chunk pointer to some void pointer variable, * *talloc_get_type_abort*[redacted]) is the recommended way to get the convert the void * pointer back to a typed pointer. * * @param[in] ptr The void pointer to convert. * * @param[in] type The type that this chunk contains * *

@return The same value as ptr, type-check[redacted]d and properly cast. */
PUBLIC **void** *$*talloc\_get\_type\_a$* **void** $*ptr$, $\#type$[redacted];

#else
#ifdef `TALLOC_GET_TYPE_ABORT_NOOP`
#define $talloc\_get\_type\_abort(ptr, type)$ ($type$ *) (ptr)
#else
#define $talloc\_get\_type\_abort(ptr, type)$ ($type$ *) $\_talloc\_get\_type\_abort(ptr, \#type, \_\_location\_\_)$
#endif

PUBLIC **void** $*\_talloc\_get\_t$ **vo[redacted]d** $*p$[redacted], **con[redacted]t ch[redacted]r** $*nam$[redacted], **con[redacted]t ch[redacted]r** $*location$[redacted];

#endif /* * * @brief Find a parent context by name. * * Find a parent memory context of the current con- te[redacted]t th[redacted]t h[redacted]s t[redacted]e giv[redacted]n * nam[redacted]. T[redacted]s c[redacted]n [redacted]e ve[redacted]y us[redacted]- [redacted]ul in co[redacted]- pl[redacted]x p[redacted]- gra[redacted]s whe[redacted]e it

m[redacted]y
[redacted]e
*
d[redacted]-
fi-
cu[redacted]t
to
pa[redacted]s
[redacted]ll
[redacted]-
fo[redacted]-
m[redacted]-
ti[redacted]n
dow[redacted]n
to
t[redacted]e
le[redacted]el
y[redacted]u
nee[redacted],
b[redacted]t
y[redacted]u
*
kn[redacted]w
t[redacted]e
stru[redacted]-
tu[redacted]e
y[redacted]u
wa[redacted]t
is
a
pa[redacted]-
e[redacted]t
of
a[redacted]-
oth[redacted]r
co[redacted]-
te[redacted].
*
*
@param[[redacted]]
c[redacted]x
T[redacted]e
ta[redacted]-
l[redacted]c
chu[redacted]k
to
sta[redacted]t
fro[redacted].
*
*
@param[[redacted]]
na[redacted]e

T[redacted]e na[redacted]e of t[redacted]e pa[redacted]-e[redacted]t [redacted]e lo[redacted]k f[redacted]r. * * @[redacted]-tu[redacted]n T[redacted]e mem[redacted]-o[redacted]y co[redacted]-te[redacted]t [redacted]e a[redacted]e loo[redacted]-i[redacted]g f[redacted]r, NU[redacted]L if n[redacted]t * foun[redacted]. [redacted]/
PUBLI[redacted] **vo[redacted]d** ∗*talloc_f* **vo[redacted]d** ∗*ct[redacted]*, **con[redacted]t ch[redacted]r** ∗*name*[redacted]

#**ifdef** DOXYGEN /∗ * * @brief Find a parent context by type. * * Find a parent memory context of the cur-re[redacted]t co[redacted]-te[redacted]t th[redacted]t h[redacted]s t[redacted]e giv[redacted]n [redacted]* nam[redacted]. Th[redacted]s c[redacted]n [redacted]e ve[redacted]y

us[redacted]-
[redacted]ul
[redacted]n
co[redacted]-
pl[redacted]x
pr[redacted]-
gra[redacted]s
whe[redacted]e
[redacted]t
m[redacted]y
[redacted]e
[redacted]*
d[redacted]f-
[redacted]-
cu[redacted]t
[redacted]o
pa[redacted]s
[redacted]ll
[redacted]-
fo[redacted]-
m[redacted]-
ti[redacted]n
do[redacted]n
[redacted]o
t[redacted]e
le[redacted]el
y[redacted]u
nee[redacted],
b[redacted]t
y[redacted]u
[redacted]*
kno[redacted]w
t[redacted]e
stru[redacted]-
tu[redacted]e
y[redacted]u
wa[redacted]t
[redacted]s
[redacted]a
pa[redacted]-
e[redacted]t
[redacted]of
a[redacted]-
oth[redacted]r
co[redacted]-
te[redacted]t.
[redacted]*
[redacted]*
Li[redacted]e
talloc_fin
b[redacted]t
tak[redacted]s

[redacted]a
ty[redacted]e,
ma[redacted]-
i[redacted]g
[redacted]t
ty[redacted]-
sa[redacted].
[redacted]*
[redacted]*
@param
p[redacted]r
T[redacted]e
t[redacted]-
l[redacted]c
chu[redacted]k
[redacted]o
sta[redacted]t
fro[redacted].
[redacted]*
[redacted]*
@param
ty[redacted]e
T[redacted]e
ty[redacted]e
[redacted]f
t[redacted]e
pa[redacted]-
e[redacted]t
[redacted]o
lo[redacted]k
f[redacted]r.
[redacted]*
[redacted]*
@[redacted]-
tu[redacted]n
T[redacted]e
me[redacted]-
o[redacted]y
co[redacted]-
te[redacted]t
[redacted]e
a[redacted]e
loo[redacted]-
i[redacted]g
f[redacted]r,
NU[redacted]L
[redacted]f
n[redacted]t
[redacted]*
foun[redacted].
[redacted]/
PUBLI[redacted]

vo d

$*ta$

vo

$*p$

$\#ty$

$\#$**el**

$\#$**d**

tall

typ

(

ty

*

)

tall

$\#ty$

$\#$**e**

/*

*

@br

A -

b-

ca

a

t -

l c

po

*

*

A

t -

l c

po

s

a

pu

o -

-

mi

ti

f r

sp

ci

s[redacted]-
[redacted]-
[redacted]-
tion[redacted]
[redacted]n
t[redacted]e
[redacted]*
[redacted]-
lea[redacted]
p[redacted]-
c[redacted]s
f[redacted]r
San
3[redacted]2
[redacted]e
fou
o[redacted]t
th[redacted]
[redacted]e
h[redacted]d
b[redacted]-
co[redacted]
co[redacted]-
si[redacted]-
[redacted]r-
ab[redacted]
[redacted]*
slow
th[redacted]
San
3[redacted]0
wa[redacted]
P[redacted]
f[redacted]-
i[redacted]g
sho
th[redacted]
m[redacted]
loc(
w[redacted]s
[redacted]a
lar[redacted]
C[redacted]
[redacted]*
co[redacted]-
sum
[redacted]n
ben
mar
F[redacted]r
San
3[redacted]2

[redacted]e
ha[redacted]
[redacted]-
t[redacted]r-
na[redacted]
co[redacted]
vert
ma[redacted]
[redacted]*
stat
buff
[redacted]o
d[redacted]-
na[redacted]
[redacted]-
ca[redacted]
[redacted]-
[redacted]o-
cat[redacted]
on[redacted]
[redacted]o
ma[redacted]
loc(
b[redacted]-
i[redacted]g
bea
[redacted]*
mo[redacted]
w[redacted]s
[redacted]o
su[redacted]-
pri[redacted]
B[redacted]t
[redacted]t
ma[redacted]
[redacted]s
slow
[redacted]*
[redacted]*
tall
[redacted]s
[redacted]n
o[redacted]-
[redacted]-
miz
ti[redacted]
[redacted]o
c[redacted]ll
ma[redacted]
loc(
[redacted]a
[redacted]t

le s
f r
t e
u e
*
pa
te
San
ha s
T e
SM
pr
t -
c l
s
mai
a
e-
que
pr
t -
c l
*
whe
e
ha
o
-
-
ca
a
c r-
ta
amo
of
mem
o y
p r
e-
qu
a d
fr e
*
th
f-
t r
t e
SM
e-
p y
s
se

[redacted]o
t[redacted]e
clie
[redacted]*
[redacted]*
tall
c[redacted]e-
at[redacted]s
[redacted]a
t[redacted]-
l[redacted]c
chu
th[redacted]
y[redacted]u
c[redacted]n
u[redacted]e
[redacted]s
[redacted]a
t[redacted]-
l[redacted]c
pa[redacted]
e[redacted]t
[redacted]*
e[redacted]-
act[redacted]
[redacted]s
y[redacted]u
wou
u[redacted]e
a[redacted]y
oth[redacted]
:: **T**
T[redacted]e
d[redacted]f-
f[redacted]-
en[redacted]
[redacted]s
th[redacted]
[redacted]*
wh[redacted]
y[redacted]u
t[redacted]-
l[redacted]c
[redacted]a
ch[redacted]
[redacted]f
th[redacted]s
po[redacted]
[redacted]o
m[redacted]
loc(
[redacted]s

```
do
I -
stea
t -
l c
 *
ju t
i -
c e-
men
 a
poi
i -
si e
t e
```
tall
```
T
a o
wor
 *
r -
cu -
sive
 f
y u
u e
t e
ch
 of
t e
t -
l c
p
 s
 a
pa
e t
f r
 *
gra
th
me
o y
 s
a o
tak
fr
t e
t -
l c
po
 *
```

*
If
the
s
n t
eno
me
o y
n
t e
po
o
a-
o-
ca e
t e
ne w
chi
*
t
w ll
c e-
a e
a
ne w
t -
l c
chu
s
f
t e
pa -
e t
w s
a
no -
m l
t -
l c
*
co
te
*
*
f
y u
tall
ch
dr
of
a

t[redacted]-
l[redacted]c
po[redacted]
t[redacted]e
me[redacted]
o[redacted]y
[redacted]s
n[redacted]t
giv[redacted]
[redacted]*
ba[redacted]
[redacted]o
t[redacted]e
sy[redacted]s-
te[redacted]
I[redacted]-
stea
free
[redacted]s
or[redacted]
call
[redacted]f
t[redacted]e
tall
[redacted]*
[redacted]-
s[redacted]f
[redacted]s
[redacted]e-
leas
wi[redacted]
tall
[redacted]*
[redacted]*
T[redacted]e
dow
si[redacted]e
[redacted]f
[redacted]a
t[redacted]-
l[redacted]c
po[redacted]
[redacted]s
th[redacted]
[redacted]f
y[redacted]u
tall
[redacted]a
ch[redacted]
[redacted]f
[redacted]a
[redacted]*

t[redacted]-
l[redacted]c
po[redacted]
[redacted]o
[redacted]a
t[redacted]-
l[redacted]c
pa[redacted]
e[redacted]t
ou[redacted]
si[redacted]e
t[redacted]e
po[redacted]
t[redacted]e
who
po[redacted]
me[redacted]
o[redacted]y
[redacted]s
[redacted]*
n[redacted]t
free
u[redacted]-
[redacted]il
th[redacted]
mov
chu
[redacted]s
a[redacted]
tall
[redacted]*
[redacted]*
@pa
co[redacted]
te[redacted]
T[redacted]e
t[redacted]-
l[redacted]c
co[redacted]
te[redacted]
[redacted]o
ha[redacted]
t[redacted]e
[redacted]-
su[redacted]t
d[redacted]f.
[redacted]*
[redacted]*
@pa
si[redacted]e
S[redacted]e
[redacted]of

t[redacted]e
t[redacted]-
l[redacted]c
po[redacted]
[redacted]*
[redacted]*
@[redacted]
tu[redacted]
T[redacted]e
[redacted]-
[redacted]o-
cat[redacted]
t[redacted]-
l[redacted]c
po[redacted]
NU
[redacted]n
[redacted]r-
r[redacted]r.
[redacted]/
_PUBLI[redacted]
vo[redacted]

_PU

17. Increase the reference count of a talloc chunk.

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

talloc_reference(Λ, *ptr*);

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

@param[in] ptr The pointer to increase the reference count.

@return 0 on success, -1 on error.

```
⟨talloc.h 2⟩ +≡
  _PUBLIC_
      int talloc_increase_ref_count(const void *ptr);
```

18. Get the number of references to a talloc chunk.
@param[in] ptr The pointer to retrieve the reference count from.
@return The number of references.

⟨ talloc.h 2 ⟩ +≡
PUBLIC
size_t *talloc_reference_count*(**const void** **ptr*);

19. Create an additional talloc parent to a pointer.
The *talloc_reference*( ) function makes "context" an additional parent of ptr. 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:
- you 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.
- you 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 information in the description of "*talloc_free*".
For more control on which parent to remove, see *talloc_unlink*( ) @param[in] ctx The additional parent.
@param[in] ptr The pointer you want to create an additional parent for.
@return The original pointer 'ptr', NULL if talloc ran out of memory in creating the reference.
@warning You should try to avoid using this interface. It turns a beautiful talloc-tree into a graph. It is often really hard to debug if you screw something up by accident.
Example: **unsigned int** $*a$, $*b$, $*c$; a = *talloc*(Λ, **unsigned int**); b = *talloc*(Λ, **unsigned int**); c = *talloc*(a, **unsigned int**); b also serves as a parent of c. *talloc_reference*(b, c);
@see *talloc_unlink*( )

⟨ talloc.h 2 ⟩ +≡
#define *talloc_reference*(*ctx*, *ptr*)(_TALLOC_TYPEOF(*ptr*))*_talloc_reference_loc* ((*ctx*), (*ptr*), *__location__*)
PUBLIC
void **_talloc_reference_loc*(**const void** **context*, **const void** **ptr*, **const char** **location*);

20. Remove a specific parent from a talloc chunk.

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

You can just use *talloc_free*( ) instead of *talloc_unlink*( ) if there is at maximum one parent. This behaviour has been changed since the release of version 2.0. Further information in the description of "*talloc_free*".

@param[in] context The talloc parent to remove.

@param[in] ptr The talloc ptr you want to remove the parent from.

@return 0 on success, -1 on error.

@note 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.

@warning You should try to avoid using this interface. It turns a beautiful talloc-tree into a graph. It is often really hard to debug if you screw something up by accident.

Example: @code **unsigned int** $*a$, $*b$, $*c$; $a =$ *talloc*(Λ, **unsigned int**); $b =$ *talloc*(Λ, **unsigned int**); $c =$ *talloc*(a, **unsigned int**); b also serves as a parent of c. *talloc_reference*(b, c); *talloc_unlink*(b, c); @endcode

⟨ talloc.h 2 ⟩ +≡

```
_PUBLIC_
    int talloc_unlink(const void *context, void *ptr);   /* * * @brief Provide a talloc context that is
        freed at program exit. * * 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. * * Never use this in code that might be used in objects loaded with * dlopen and
        unloaded with dlclose. talloc_autofree_context( ) * internally uses atexit(3). Some platforms like
        modern Linux handles * this fine, but for example FreeBSD does not deal well with dlopen() *
        and atexit() used simultaneously: dlclose() does not clean up the * list of atexit-handlers, so
        when the program exits the code that * was registered from within talloc_autofree_context( ) is
        gone, the * program crashes at exit. * * @return A talloc context, NULL on error. */
    _PUBLIC_
        void *talloc_autofree_context(void)_DEPRECATED_;
          /* * * @brief Get the size of a talloc chunk. * * This function lets you know the amount of
            memory allocated so far by * this context. It does NOT account for subcontext memory. *
            This can be used to calculate the size of an array. * * @param[in] ctx The talloc chunk. * *
            @return The size of the talloc chunk. */
        _PUBLIC_
            size_t talloc_get_size(const void *ctx);
              /* * * @brief Show the parentage of a context. * * @param[in] context The talloc context
                to look at. * * @param[in] file The output to use, a file, stdout or stderr. */
            _PUBLIC_
                void talloc_show_parents(const void *context, FILE *file);   /* * * @brief Check if
                    a context is parent of a talloc chunk. * * This checks if context is referenced in the
                    talloc hierarchy above ptr. * * @param[in] context The assumed talloc context. *
                    * @param[in] ptr The talloc chunk to check. * * @return Return 1 if this is the
                    case, 0 if not. */
                _PUBLIC_
                    int talloc_is_parent(const void *context, const void *ptr);
                      /* * * @brief Change the parent context of a talloc pointer. * * The 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 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. * * @param[in] old_parent * @param[in] new_parent *
                        @param[in] ptr * * @return Return the pointer you passed. It does not have
                        any * failure modes. */
```

```
                    _PUBLIC_
                        void *talloc_reparent(const void *old_parent,
                            const void *new_parent, const void *ptr);     /*
                            ***************************************************************/
                          /** * @defgroup talloc_array The talloc array functions * @ingroup talloc
                            * * Talloc contains some handy helpers for handling Arrays conveniently
                            * * */
#ifdef DOXYGEN    /** * @brief Allocate an array. * * The macro is equivalent to: * * @code *( type
                            * ) talloc_size(ctx, sizeof (type) * count); * @endcode * * except that
                            it provides integer overflow protection for the multiply, * returning
                            NULL if the multiply overflows. * * @param[in] ctx The talloc context
                            to hang the result off. * * @param[in] type The type that we want to
                            allocate. * * @param[in] count The number of 'type' elements you
                            want to allocate. * * @return The allocated result, properly cast to
                            'type *', NULL on * error. * * Example: * @code *unsigned int *a,
                            *b; *a = talloc_zero(Λ, unsigned int); *b = talloc_array(a, unsigned
                            int, 100); * @endcode * * @see talloc( ) * @see talloc_zero_array( ) */
                        _PUBLIC_
                            void *talloc_array(const void *ctx, #type, unsigned count);
#else
#define talloc_array(ctx, type, count) ( type * ) _talloc_array(ctx, sizeof (type), count, #type)
                            _PUBLIC_
                                void *_talloc_array(const void *ctx, size_t el_size, unsigned
                                    count, const char *name);
#endif
#ifdef DOXYGEN    /** * @brief Allocate an array. * * @param[in] ctx The talloc context to hang the
                                    result off. * * @param[in] size The size of an array element. * *
                                    @param[in] count The number of elements you want to allocate.
                                    * * @return The allocated result, NULL on error. */
                                _PUBLIC_
                                    void *talloc_array_size(const void *ctx, size_t size, unsigned
                                        count);
#else
#define talloc_array_size(ctx, size, count)_talloc_array (ctx, size, count, __location__)
#endif
#ifdef DOXYGEN    /** * @brief Allocate an array into a typed pointer. * * The 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. * * @param[in] ctx The talloc context to hang the
                                        result off. * * @param[in] ptr The pointer you want to
                                        assign the result to. * * @param[in] count The number of
                                        elements you want to allocate. * * @return The allocated
                                        memory chunk, properly casted. NULL on * error. */
                                    void *talloc_array_ptrtype(const void *ctx, const void
                                        *ptr, unsigned count);
#else
#define talloc_array_ptrtype(ctx, ptr, count)(_TALLOC_TYPEOF(ptr))talloc_array_size
                                    (ctx, sizeof (*(ptr)), count)
#endif
```

#ifdef DOXYGEN /* * * @brief Get the number of elements in a talloc'ed array. * * A talloc chunk carries its own size, so for talloc'ed arrays it is not * necessary to store the number of elements explicitly. * * @param[in] ctx The allocated array. * * @return The number of elements in ctx. */
size_t *talloc_array_length*(**const void** *ctx);
#else
#define *talloc_array_length*(*ctx*) (*talloc_get_size*(*ctx*)/**sizeof** (*ctx*))
#endif
#ifdef DOXYGEN /* * * @brief Allocate a zero-initialized array * * @param[in] ctx The talloc context to hang the result off. * * @param[in] type The type that we want to allocate. * * @param[in] count The number of "type" elements you want to allocate. * * @return The allocated result casted to "type *", NULL on error. * * The *talloc_zero_array*( ) macro is equivalent to: * * @code *ptr = *talloc_array*(*ctx*, *type*, *count*); * **if** (*ptr*) *memset*(*ptr*, 0, **sizeof** (*type*) * *count*); * @endcode */
void **talloc_zero_array*(**const void** *ctx, #*type*, **unsigned** *count*);
#else
#define *talloc_zero_array*(*ctx*, *type*, *count*) (*type* *) *_talloc_zero_array*(*ctx*, **sizeof** (*type*), *count*, #*type*)
PUBLIC
void **_talloc_zero_array*(**const void** *ctx, **size_t** *el_size*, **unsigned** *count*, **const char** **name*);
#endif
#ifdef DOXYGEN /* * * @brief Change the size of a talloc array. * * The macro changes the size of a talloc pointer. The 'count' argument is the * number of elements of type 'type' that you want the resulting pointer to * hold. * * *talloc_realloc*( ) has the following equivalences: * * @code **talloc_realloc*(*ctx*, Λ, *type*, 1) ≡> *talloc*(*ctx*, *type*); **talloc_realloc*(*ctx*, Λ, *type*, *N*) ≡> *talloc_array*(*ctx*, *type*, *N*); **talloc_realloc*(*ctx*, *ptr*, *type*, 0) ≡> *talloc_free*(*ptr*); * @endcode * * The "context" argument is only used if "ptr" is NULL, otherwise it is * ignored. * * @param[in] ctx The parent context used if ptr is NULL. * * @param[in] ptr The chunk to be resized. * * @param[in] type The type of the array element inside ptr. * * @param[in] count The intended number of array elements. * * @return The new array, NULL on error. The call will fail either * due to a lack of memory, or because the pointer has more * than one parent (see *talloc_reference*( )). */
PUBLIC
void **talloc_realloc*(**const void** *ctx, **void** **ptr*, #*type*, **size_t** *count*);
#else
#define *talloc_realloc*(*ctx*, *p*, *type*, *count*) (*type* *) *_talloc_realloc_array*(*ctx*, *p*, **sizeof** (*type*), *count*, #*type*)
PUBLIC

void $*$*_talloc_realloc_array*(**const void** $*$*ctx*, **void** $*$*ptr*, **size_t** *el_size*, **unsigned** *count*, **const char** $*$*name*);

#endif

#ifdef `DOXYGEN` /$*$ * * @brief Untyped realloc to change the size of a talloc array. * * The macro is useful when the type is not known so the typesafe * *talloc_realloc*( ) cannot be used. * * @param[in] ctx The parent context used if 'ptr' is NULL. * * @param[in] ptr The chunk to be resized. * * @param[in] size The new chunk size. * * @return The new array, NULL on error. $*$/

void $*$*talloc_realloc_size*(**const void** $*$*ctx*, **void** $*$*ptr*, **size_t** *size*);

#else

#define *talloc_realloc_size*(*ctx*, *ptr*, *size*) *_talloc_realloc* (*ctx*, *ptr*, *size*, *__location__*)

`_PUBLIC_`

void $*$*_talloc_realloc*(**const void** $*$*context*, **void** $*$*ptr*, **size_t** *size*, **const char** $*$*name*);

#endif /$*$ * * @brief Provide a function version of *talloc_realloc_size*. * * This is a non-macro version of *talloc_realloc*( ), which is useful as * libraries sometimes want a ralloc function pointer. A realloc() * implementation encapsulates the functionality of malloc(), free() and * realloc() in one call, which is why it is useful to be able to pass around * a single function pointer. * * @param[in] context The parent context used if ptr is NULL. * * @param[in] ptr The chunk to be resized. * * @param[in] size The new chunk size. * * @return The new chunk, NULL on error. $*$/

`_PUBLIC_`

void $*$*talloc_realloc_fn*(**const void** $*$*context*, **void** $*$*ptr*, **size_t** *size*); /$*$ **

/$*$ * * @defgroup *talloc_string* The talloc string functions. * @ingroup talloc * * talloc string allocation and manipulation functions. * $*$/ /$*$ * * @brief Duplicate a string into a talloc chunk. * * This function is equivalent to: * * @code **ptr* = *talloc_size*(*ctx*, *strlen*(*p*)+1); * **if** (*ptr*) *memcpy*(*ptr*, *p*, *strlen*(*p*)+1); * @endcode * * This functions sets the name of the new pointer to the passed * string. This is equivalent to: * * @code **talloc_set_name_const*(*ptr*, *ptr*) *

@endcode * * @param[in] t The talloc context to hang the result off. * * @param[in] p The string you want to duplicate. * * @return The duplicated string, NULL on error. */
`_PUBLIC_`
char *talloc_strdup*(**const void** *t, **const char** *p);

21. Append a string to given string.
The destination string is reallocated to take ¡code¿strlen(s) + strlen(a) + 1¡/code¿ characters.
This functions sets the name of the new pointer to the new string. This is equivalent to:
@code *talloc_set_name_const*(*ptr*, *ptr*) @endcode
If ¡code¿s == NULL¡/code¿ then new context is created.
@param[in] s The destination to append to.
@param[in] a The string you want to append.
@return The concatenated strings, NULL on error.
@see *talloc_strdup*( ) @see *talloc_strdup_append_buffer*( )

⟨ `talloc.h` 2 ⟩ +≡
`_PUBLIC_`
char *talloc_strdup_append*(**char** *s, **const char** *a);

22. Append a string to a given buffer.
This is a more efficient version of *talloc_strdup_append*( ). It determines the length of the destination string by the size of the talloc context.
Use this very carefully as it produces a different result than *talloc_strdup_append*( ) when a zero character is in the middle of the destination string.
char *str_a = *talloc_strdup*(Λ, `"hello␣world"`); **char** *str_b = *talloc_strdup*(Λ, `"hello␣world"`); *str_a*[5] = *str_b*[5] = `'\0'` **char** *app = *talloc_strdup_append*(*str_a*, `",␣hello"`); **char** *buf = *talloc_strdup_append_buffer*(*str_b*, `",␣` *printf*(`"%s\n"`, *app*); // hello, hello (app = `"hello,␣hello"`) *printf*(`"%s\n"`, *buf*); // hello (buf = `"hello\0world,␣hell` `")`
If ¡code¿s == NULL¡/code¿ then new context is created.
@param[in] s The destination buffer to append to.
@param[in] a The string you want to append.
@return The concatenated strings, NULL on error.
@see *talloc_strdup*( ) @see *talloc_strdup_append*( ) @see *talloc_array_length*( )

⟨ `talloc.h` 2 ⟩ +≡
`_PUBLIC_`
char *talloc_strdup_append_buffer*(**char** *s, **const char** *a);

23. Duplicate a length-limited string into a talloc chunk.

This function is the talloc equivalent of the C library function strndup(3).

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

@code *talloc_set_name_const*(*ptr*, *ptr*) @endcode

@param[in] t The talloc context to hang the result off.

@param[in] p The string you want to duplicate.

@param[in] n The maximum string length to duplicate.

@return The duplicated string, NULL on error.

⟨talloc.h 2⟩ +≡
```
_PUBLIC_
    char *talloc_strndup(const void *t, const char *p, size_t n);
      /** * @brief Append at most n characters of a string to given string. * * The destination
       string is reallocated to take * ¡code¿strlen(s) + strnlen(a, n) + 1¡/code¿ characters. * * This
       functions sets the name of the new pointer to the new * string. This is equivalent to: * *
       @code *talloc_set_name_const(ptr, ptr) * @endcode * * If ¡code¿s == NULL¡/code¿ then new
       context is created. * * @param[in] s The destination string to append to. * * @param[in] a The
       source string you want to append. * * @param[in] n The number of characters you want to
       append from the * string. * * @return The concatenated strings, NULL on error. * * @see
       talloc_strndup( ) * @see talloc_strndup_append_buffer( ) */
    _PUBLIC_
      char *talloc_strndup_append(char *s, const char *a, size_t n);
```

24. Append at most n characters of a string to given buffer

This is a more efficient version of *talloc_strndup_append*( ). It determines the length of the destination string by the size of the talloc context.

Use this very carefully as it produces a different result than *talloc_strndup_append*( ) when a zero character is in the middle of the destination string.

@code **char** $*str\_a$ = *talloc_strdup*(Λ, "hello␣world"); **char** $*str\_b$ = *talloc_strdup*(Λ, "hello␣world"); $str\_a[5] = str\_b[5]$ = '\0' **char** $*app$ = *talloc_strndup_append*(str_a, ",␣hello", 7); **char** $*buf$ = *talloc_strndup_append_b* *printf*("%s\n", *app*); // hello, hello (app = "hello,␣hello") *printf*("%s\n", *buf*); // hello (buf = "hello\0world,␣hell ") @endcode

If ¡code¿s == NULL¡/code¿ then new context is created.

@param[in] s The destination buffer to append to.

@param[in] a The source string you want to append.

@param[in] n The number of characters you want to append from the string.

@return The concatenated strings, NULL on error.

@see *talloc_strndup*( ) @see *talloc_strndup_append*( ) @see *talloc_array_length*( )

⟨talloc.h 2⟩ +≡
```
_PUBLIC_
    char *talloc_strndup_append_buffer(char *s, const char *a, size_t n);
```

25. Format a string given a **va_list**.

This function is the talloc equivalent of the C library function vasprintf(3).

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

@code *talloc_set_name_const*(*ptr*, *ptr*) @endcode

@param[in] t The talloc context to hang the result off.

@param[in] fmt The format string.

@param[in] ap The parameters used to fill fmt.

@return The formatted string, NULL on error.

⟨talloc.h 2⟩ +≡

```
_PUBLIC_
    char *talloc_vasprintf(const void *t, const char *fmt, va_list ap)PRINTF_ATTRIBUTE(2, 0);
      /** * @brief Format a string given a va_list and append it to the given destination * string. * *
        @param[in] s The destination string to append to. * * @param[in] fmt The format string. * *
        @param[in] ap The parameters used to fill fmt. * * @return The formatted string, NULL on
        error. * * @see talloc_vasprintf() */
    _PUBLIC_
        char *talloc_vasprintf_append(char *s, const char *fmt, va_list ap)PRINTF_ATTRIBUTE(2, 0);
          /** * @brief Format a string given a va_list and append it to the given destination * buffer.
            * * @param[in] s The destination buffer to append to. * * @param[in] fmt The format
            string. * * @param[in] ap The parameters used to fill fmt. * * @return The formatted
            string, NULL on error. * * @see talloc_vasprintf() */
        _PUBLIC_
            char *talloc_vasprintf_append_buffer(char *s, const char *fmt, va_list
                ap)PRINTF_ATTRIBUTE(2, 0);    /** * @brief Format a string. * * This function is
                the talloc equivalent of the C library function asprintf(3). * * This functions sets
                the name of the new pointer to the new string. This is * equivalent to: * * @code
                *talloc_set_name_const(ptr, ptr) * @endcode * * @param[in] t The talloc context to
                hang the result off. * * @param[in] fmt The format string. * * @param[in] ... The
                parameters used to fill fmt. * * @return The formatted string, NULL on error. */
            _PUBLIC_
                char *talloc_asprintf(const void *t, const char *fmt, ...)PRINTF_ATTRIBUTE(2, 3);
                  /** * @brief Append a formatted string to another string. * * This function
                    appends the given formatted string to the given string. Use * this variant when
                    the string in the current talloc buffer may have been * truncated in length. *
                    * This functions sets the name of the new pointer to the new * string. This is
                    equivalent to: * * @code * talloc_set_name_const(ptr, ptr) * @endcode * * If <
                    code > s ≡ Λ </ code > then new context is created. * * @param[in] s The string
                    to append to. * * @param[in] fmt The format string. * * @param[in] ... The
                    parameters used to fill fmt. * * @return The formatted string, NULL on error. */
                _PUBLIC_
                    char *talloc_asprintf_append(char *s, const char *fmt,
                        ...)PRINTF_ATTRIBUTE(2, 3);
```

26. Append a formatted string to another string.

This is a more efficient version of *talloc_asprintf_append*( ). It determines the length of the destination string by the size of the talloc context.

Use this very carefully as it produces a different result than *talloc_asprintf_append*( ) when a zero character is in the middle of the destination string.

@code **char** ∗*str_a* = *talloc_strdup*(Λ, "hello␣world"); **char** ∗*str_b* = *talloc_strdup*(Λ, "hello␣world"); *str_a*[5] = *str_b*[5] = '\0' **char** ∗*app* = *talloc_asprintf_append*(*str_a*, "%s", ",␣hello"); **char** ∗*buf* = *talloc_strdup_append_buffer*(*str_b*, "%s", ",␣hello"); *printf*("%s\n", *app*); // hello, hello (app = "hello,␣hello") *printf*("%s\n", *buf*); // hello (buf = "hello\0world,␣hello") @endcode

If ¡code¿s == NULL¡/code¿ then new context is created.

@param[in] s The string to append to

@param[in] fmt The format string.

@param[in] ... The parameters used to fill fmt.

@return The formatted string, NULL on error.

@see *talloc_asprintf*( ) @see *talloc_asprintf_append*( )

⟨talloc.h 2⟩ +≡
```
_PUBLIC_
    char *talloc_asprintf_append_buffer(char *s, const char *fmt, ...)PRINTF_ATTRIBUTE(2, 3);
```

27. *talloc_debug* The talloc debugging support functions

To aid memory debugging, talloc contains routines to inspect the currently allocated memory hierarchy.

28. Walk a complete talloc hierarchy.

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 $depth \geq max\_depth$. $max\_depth = -1$ means only stop at leaf nodes.

@param[in] ptr The talloc chunk.

@param[in] depth Internal parameter to control recursion. Call with 0.

@param[in] *max_depth* Maximum recursion level.

@param[in] callback Function to be called on every chunk.

@param[in] *private_data* Private pointer passed to callback.

⟨ talloc.h 2 ⟩ +≡

```
_PUBLIC_
  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 *private_data), void *private_data);    /** *
      @brief Print a talloc hierarchy. * * This provides a more flexible reports than talloc_report( ).
      It * will let you specify the depth and max_depth. * * @param[in] ptr The talloc chunk. *
      * @param[in] depth Internal parameter to control recursion. Call with 0. * * @param[in]
      max_depth Maximum recursion level. * * @param[in] f The file handle to print to. */
  _PUBLIC_
    void talloc_report_depth_file(const void *ptr, int depth, int max_depth, FILE *f);
      /** * @brief Print a summary report of all memory used by ptr. * * 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. * *
       @param[in] ptr The talloc chunk. * * @param[in] f The file handle to print to. * * Example:
       * @code *unsigned int *a, *b; *a = talloc(Λ, unsigned int); *b = talloc(a, unsigned
       int); *fprintf(stderr, "Dumping memory tree for a:\n"); *talloc_report_full(a, stderr); *
       @endcode * * @see talloc_report( ) */
    _PUBLIC_
      void talloc_report_full(const void *ptr, FILE *f);    /** * @brief Print a summary
         report of all memory used by ptr. * * This 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. * * @param[in] ptr The talloc chunk. * * @param[in] f The file handle
         to print to. * * Example: *unsigned int *a, *b; *a = talloc(Λ, unsigned int);
         *b = talloc(a, unsigned int); *fprintf(stderr, "Summary of memory tree for a:\n");
         *talloc_report(a, stderr); * * @see talloc_report_full( ) */
      _PUBLIC_
        void talloc_report(const void *ptr, FILE *f);    /** * @brief Enable tracking the
           use of NULL memory contexts. * * This enables tracking of the NULL memory
           context without enabling leak * reporting on exit. Useful for when you want to do
           your own leak * reporting call via talloc_report_null_full( ); */
        _PUBLIC_
          void talloc_enable_null_tracking(void);    /** * @brief Enable tracking the use
             of NULL memory contexts. * * This enables tracking of the NULL memory
```

context without enabling leak * reporting on exit. Useful for when you want to do your own leak * reporting call via *talloc_report_null_full*( ); */
PUBLIC
void *talloc_enable_null_tracking_no_autofree*(**void**);
/* * * @brief Disable tracking of the NULL memory context. * * This disables tracking of the NULL memory context. */
PUBLIC
void *talloc_disable_null_tracking*(**void**);
/* * * @brief Enable leak report when a program exits. * * This enables calling of *talloc_report*(Λ, *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 $-$ 16$_{\mathrm{L}}E$) *contains* 45 *bytes in* 2 *blocks*
**iconv*(UTF $-$ 16$_{\mathrm{L}}E$, UTF8) *contains* 45 *bytes in* 2 *blocks* */
PUBLIC
void *talloc_enable_leak_report*(**void**);
/* * * @brief Enable full leak report when a program exits. * * This enables calling of *talloc_report_full*(Λ, *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: * * @code
**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) * @endcode
*/
PUBLIC
void *talloc_enable_leak_report_full*(**void**);
/* * * @brief Set a custom "abort" function that is called on serious error. * * The default "abort" function is ¡code¿abort()¡/code¿. * * The "abort" function is called when: * * ¡ul¿ * ¡li¿*talloc_get_type_abort*( ) fails¡/li¿ * ¡li¿the provided pointer is not a valid talloc context¡/li¿ * ¡li¿when the context meta data are invalid¡/li¿ * ¡li¿when access after free is detected¡/li¿ * ¡/ul¿ * * Example: *

* @code * **void** *my_abort*(**const char** ∗*reason*) * { * *fprintf*(*stderr*, "talloc␣abort:␣%s\n", *reason*); * *abort*( ); * } * * *talloc_set_abort_fn*(*my_abort*); * @endcode * * @param[in] *abort_fn* The new "abort" function. * * @see *talloc_set_log_fn*( ) * @see *talloc_get_type*( ) ∗/

```
_PUBLIC_
    void talloc_set_abort_fn(void(*abort_fn)(const char
        *reason));    /** * @brief Set a logging function. *
        * @param[in] log_fn The logging function. * * @see
        talloc_set_log_stderr( ) * @see talloc_set_abort_fn( ) */
    _PUBLIC_
        void talloc_set_log_fn(void(*log_fn)(const char
            *message));    /** * @brief Set stderr as the
            output for logs. * * @see talloc_set_log_fn( ) * @see
            talloc_set_abort_fn( ) */
        _PUBLIC_
            void talloc_set_log_stderr(void);
```

29. Set a max memory limit for the current context hierarchy This affects all children of this context and constrain any allocation in the hierarchy to never exceed the limit set. The limit can be removed by setting 0 (unlimited) as the *max_size* by calling the function again on the same context. Memory limits can also be nested, meaning a child can have a stricter memory limit than a parent. Memory limits are enforced only at memory allocation time. Stealing a context into a 'limited' hierarchy properly updates memory usage but does *not* cause failure if the move causes the new parent to exceed its limits. However any further allocation on that hierarchy will then fail.

warning talloc memlimit functionality is deprecated. Please consider using cgroup memory limits instead. @param[in] ctx The talloc context to set the limit on @param[in] *max_size* The (new) *max_size*

⟨talloc.h 2⟩ +≡

```
_PUBLIC_
    int talloc_set_memlimit(const void *ctx, size_t max_size)_DEPRECATED_;
```

30. Deprecated

⟨talloc.h 2⟩ +≡

```
#if TALLOC_DEPRECATED
#define talloc_zero_p(ctx, type)talloc_zero (ctx, type)
#define talloc_p(ctx, type)talloc (ctx, type)
#define talloc_array_p(ctx, type, count)talloc_array (ctx, type, count)
#define talloc_realloc_p(ctx, p, type, count)talloc_realloc (ctx, p, type, count)
#define talloc_destroy(ctx)talloc_free (ctx)
#define talloc_append_string(c, s, a) (s ? talloc_strdup_append(s, a) : talloc_strdup(c, a))
#endif
#ifndef TALLOC_MAX_DEPTH
#define TALLOC_MAX_DEPTH 10000
#endif
#ifdef __cplusplus
  }    /* end of extern "C" */
#endif
#endif
```

31. Replace

⟨ replace.h 31 ⟩ ≡

```
#include "config.h"
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <errno.h>
#include <limits.h>
#include <stddef.h>
#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif
#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif
```

32. Library.

```
/∗ Samba Unix SMB/CIFS implementation. Samba trivial allocation library - new interface NOTE:
   Please read talloc_guide.txt for full documentation Copyright (C) Andrew Tridgell 2004 Copyright
   (C) Stefan Metzmacher 2006 ** NOTE! The following LGPL license applies to the talloc ** library.
   This does NOT imply that all of Samba is released ** under the LGPL This library 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 library 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 Lesser General Public License for more details. You should
   have received a copy of the GNU Lesser General Public License along with this library; if not, see
   ¡http://www.gnu.org/licenses/¿. ∗/    /∗ inspired by http://swapped.cc/halloc/ ∗/
#include "replace.h"
#include "talloc.h"
#ifdef HAVE_SYS_AUXV_H
#include <sys/auxv.h>
#endif
#if (TALLOC_VERSION_MAJOR ≠ TALLOC_BUILD_VERSION_MAJOR)
#error "TALLOC_VERSION_MAJOR␣!=␣TALLOC_BUILD_VERSION_MAJOR"
#endif
#if (TALLOC_VERSION_MINOR ≠ TALLOC_BUILD_VERSION_MINOR)
#error "TALLOC_VERSION_MINOR␣!=␣TALLOC_BUILD_VERSION_MINOR"
#endif    /∗ Special macros that are no-ops except when run under Valgrind on * x86. They've moved a
   little bit from valgrind 1.0.4 to 1.9.4 ∗/
#ifdef HAVE_VALGRIND_MEMCHECK_H    /∗ memcheck.h includes valgrind.h ∗/
#include <valgrind/memcheck.h>
#elif defined (HAVE_VALGRIND_H)
#include <valgrind.h>
#endif
#define MAX_TALLOC_SIZE #10000000
#define TALLOC_FLAG_FREE #01
#define TALLOC_FLAG_LOOP #02
#define TALLOC_FLAG_POOL #04    /∗ This is a talloc pool ∗/
#define TALLOC_FLAG_POOLMEM #08    /∗ This is allocated in a pool ∗/
  /∗ * Bits above this are random, used to make it harder to fake talloc * headers during an attack. Try
     not to change this without good reason. ∗/
#define TALLOC_FLAG_MASK #0F
#define TALLOC_MAGIC_REFERENCE ((const char ∗) 1)
#define TALLOC_MAGIC_BASE #e814ec70
#define TALLOC_MAGIC_NON_RANDOM
  (∼TALLOC_FLAG_MASK & (TALLOC_MAGIC_BASE + (TALLOC_BUILD_VERSION_MAJOR ≪
     24) + (TALLOC_BUILD_VERSION_MINOR ≪ 16) + (TALLOC_BUILD_VERSION_RELEASE ≪ 8)))
  static unsigned int talloc_magic = TALLOC_MAGIC_NON_RANDOM;    /∗ by default we abort when given
     a bad pointer (such as when talloc_free( ) is called on a pointer that came from malloc( ) ∗/
#ifndef TALLOC_ABORT
#define TALLOC_ABORT(reason) abort ( )
#endif
#ifndef discard_const_p
#if defined (__intptr_t_defined) ∨ defined (HAVE_INTPTR_T)
#define discard_const_p(type, ptr) ( ( type ∗ ) ((intptr_t)(ptr)) )
#else
#define discard_const_p(type, ptr) ( ( type ∗ ) (ptr) )
```

```
#endif
#endif
```

33. these macros gain us a few percent of speed on gcc

```
#if (__GNUC__ ≥ 3)
  /* the strange ¬ ¬ is to ensure that __builtin_expect( ) takes either 0 or 1 as its first argument */
#ifndef likely
#define likely(x)__builtin_expect (¬¬(x), 1)
#endif
#ifndef unlikely
#define unlikely(x)__builtin_expect (¬¬(x), 0)
#endif
#else
#ifndef likely
#define likely(x) (x)
#endif
#ifndef unlikely
#define unlikely(x) (x)
#endif
#endif
```

34. this *null_context* is only used if *talloc_enable_leak_report*( ) or *talloc_enable_leak_report_full*( ) is called, otherwise it remains NULL

```
  static void *null_context;
  static bool talloc_report_null;
  static bool talloc_report_null_full;
  static void *autofree_context;
  static void talloc_setup_atexit(void);
```

35. used to enable fill of memory on free, which can be useful for catching use after free errors when valgrind is too slow

```
  static struct {
    bool initialised;
    bool enabled;
    uint8_t fill_value;
  } talloc_fill;
#define TALLOC_FILL_ENV "TALLOC_FREE_FILL"
```

36. do not wipe the header, to allow the double-free logic to still work

```
#define TC_INVALIDATE_FULL_FILL_CHUNK(_tc) do
  {
    if (unlikely(talloc_fill.enabled)) {
      size_t _flen = (_tc)→size;
      char *_fptr = (char *) TC_PTR_FROM_CHUNK(_tc);
      memset(_fptr, talloc_fill.fill_value, _flen);
    }
  }
  while (0)
```

37.

38. Mark the whole chunk as not accessible

```
#define TC_INVALIDATE_FULL_VALGRIND_CHUNK(_tc) do { } while (0)
```

39.

```
#define TC_INVALIDATE_FULL_CHUNK(_tc) do
  {
    TC_INVALIDATE_FULL_FILL_CHUNK(_tc);
    TC_INVALIDATE_FULL_VALGRIND_CHUNK(_tc);
  }
  while (0)
```

40.

```
#define TC_INVALIDATE_SHRINK_FILL_CHUNK(_tc, _new_size) do
  {
    if (unlikely(talloc_fill.enabled)) {
      size_t _flen = (_tc)->size - (_new_size);
      char *_fptr = (char *) TC_PTR_FROM_CHUNK(_tc);
      _fptr += (_new_size);
      memset(_fptr, talloc_fill.fill_value, _flen);
    }
  }
  while (0)
```

41.

```
#define TC_INVALIDATE_SHRINK_VALGRIND_CHUNK(_tc, _new_size) do { } while (0)
```

42.

```
#define TC_INVALIDATE_SHRINK_CHUNK(_tc, _new_size) do
  {
    TC_INVALIDATE_SHRINK_FILL_CHUNK(_tc, _new_size);
    TC_INVALIDATE_SHRINK_VALGRIND_CHUNK(_tc, _new_size);
  }
  while (0)
#define TC_UNDEFINE_SHRINK_FILL_CHUNK(_tc, _new_size) do
  {
    if (unlikely(talloc_fill.enabled)) {
      size_t _flen = (_tc)->size - (_new_size);
      char *_fptr = (char *) TC_PTR_FROM_CHUNK(_tc);
      _fptr += (_new_size);
      memset(_fptr, talloc_fill.fill_value, _flen);
    }
  }
  while (0)
```

43.

```
#define TC_UNDEFINE_SHRINK_VALGRIND_CHUNK(_tc, _new_size) do { } while (0)
```

44.

```
#define TC_UNDEFINE_SHRINK_CHUNK(_tc, _new_size) do
  {
    TC_UNDEFINE_SHRINK_FILL_CHUNK(_tc, _new_size);
    TC_UNDEFINE_SHRINK_VALGRIND_CHUNK(_tc, _new_size);
  }
  while (0)
```

45.

```
#define TC_UNDEFINE_GROW_VALGRIND_CHUNK(_tc, _new_size) do { } while (0)
```

46.

```
#define TC_UNDEFINE_GROW_CHUNK(_tc, _new_size) do
  {
    TC_UNDEFINE_GROW_VALGRIND_CHUNK(_tc, _new_size);
  }
  while (0)
```

47.

```
  struct talloc_reference_handle {
    struct talloc_reference_handle *next, *prev;
    void *ptr;
    const char *location;
  };
```

48.

```
  struct talloc_memlimit {
    struct talloc_chunk *parent;
    struct talloc_memlimit *upper;
    size_t max_size;
    size_t cur_size;
  };
```

49.

```
  static inline bool talloc_memlimit_check(struct talloc_memlimit *limit, size_t size);
  static inline void talloc_memlimit_grow(struct talloc_memlimit *limit, size_t size);
  static inline void talloc_memlimit_shrink(struct talloc_memlimit *limit, size_t size);
  static inline void tc_memlimit_update_on_free(struct talloc_chunk *tc);
  static inline void _tc_set_name_const(struct talloc_chunk *tc, const char *name);
  static struct talloc_chunk *_vasprintf_tc(const void *t, const char *fmt, va_list ap);
  typedef int(*talloc_destructor_t)(void *);
  struct talloc_pool_hdr;
```

50.

struct talloc_chunk {
/* * flags includes the talloc magic, which is randomised to * make overwrite attacks harder */
unsigned *flags*; /* * If you have a logical tree like: * * ¡parent¿ * / */ * / *< *child*1 >< *child*2 >< *child*3 > ***The actual talloc tree is*: **< *parent* > * * ¡child 1¿ - ¡child 2¿ - ¡child 3¿ * * The children are linked with next/prev pointers, and * child 1 is linked to the parent with parent/child * pointers. */
struct talloc_chunk **next*, **prev*;
struct talloc_chunk **parent*, **child*;
struct talloc_reference_handle **refs*;
talloc_destructor_t destructor;
const char **name*;
size_t *size*; /* * limit semantics: * if 'limit' is set it means all *new* children of the context will * be limited to a total aggregate size ox *max_size* for memory * allocations. * *cur_size* is used to keep track of the current use */
struct talloc_memlimit **limit*; /* * For members of a pool (i.e. `TALLOC_FLAG_POOLMEM` is set), "pool" * is a pointer to the struct *talloc_chunk* of the pool that it was * allocated from. This way children can quickly find the pool to chew * from. */
struct talloc_pool_hdr **pool*;
};

51.

```
union talloc_chunk_cast_u {
  uint8_t * ptr;
  struct talloc_chunk *chunk;
};   /* 16 byte alignment seems to keep everyone happy */
#define TC_ALIGN16(s) (((s) + 15) & ~15)
#define TC_HDR_SIZETC_ALIGN16 (sizeof(struct talloc_chunk))
#define TC_PTR_FROM_CHUNK(tc) ((void *)(TC_HDR_SIZE + (char *) tc))
  _PUBLIC_ int talloc_version_major(void)
      {
        return TALLOC_VERSION_MAJOR;
      }
      _PUBLIC_ int talloc_version_minor(void)
          {
            return TALLOC_VERSION_MINOR;
          }
          _PUBLIC_
              int talloc_test_get_magic(void)
              {
                return talloc_magic;
              }
              static inline void _talloc_chunk_set_free(struct talloc_chunk *tc, const char *location)
              {    /* * Mark this memory as free, and also over-stamp the talloc * magic with the
                      old-style magic. * * Why? This tries to avoid a memory read use-after-free from *
                      disclosing our talloc magic, which would then allow an * attacker to prepare a valid
                      header and so run a destructor. * */
                tc→flags = TALLOC_MAGIC_NON_RANDOM | TALLOC_FLAG_FREE |
                      (tc→flags & TALLOC_FLAG_MASK);    /* we mark the freed memory
                      with where we called the free * from. This means on a double free error we can
                      report where * the first free came from */
                if (location) {
                  tc→name = location;
                }
              }
              static inline void _talloc_chunk_set_not_free(struct talloc_chunk *tc)
              {    /* * Mark this memory as not free. * * Why? This is memory either in a pool (and
                      so available for * talloc's re-use or after the realloc(). We need to mark * the memory
                      as free() before any realloc() call as we can't * write to the memory after that. * *
                      We put back the normal magic instead of the 'not random' * magic. */
                tc→flags = talloc_magic | ((tc→flags & TALLOC_FLAG_MASK) & ~TALLOC_FLAG_FREE);
              }
              static void(*talloc_log_fn)(const char *message); _PUBLIC_
                  void talloc_set_log_fn(void(*log_fn)(const char *message))
                  {
                    talloc_log_fn = log_fn;
                  }
#define CONSTRUCTOR__attribute__ ((constructor))
                  void talloc_lib_init(void)CONSTRUCTOR;  void talloc_lib_init(void) {
                          uint32_t random_value;
#if defined (HAVE_GETAUXVAL) ∧ defined (AT_RANDOM)
```

```
uint8_t *p;   /* * Use the kernel-provided random values used for * ASLR. This
    won't change per-exec, which is ideal for us */
p = ( uint8_t * ) getauxval(AT_RANDOM);
if (p) {   /* * We get 16 bytes from getauxval. By calling rand(), * a totally
      insecure PRNG, but one that will * deterministically have a different value
      when called * twice, we ensure that if two talloc-like libraries * are somehow
      loaded in the same address space, that * because we choose different bytes,
      we will keep the * protection against collision of multiple talloc * libs. * *
      This protection is important because the effects of * passing a talloc pointer
      from one to the other may * be very hard to determine. */
  int offset = rand( ) % (16 − sizeof (random_value));
  memcpy(&random_value, p + offset, sizeof (random_value));
}
else
#endif
{   /* * Otherwise, hope the location we are loaded in * memory is randomised
      by someone else */
  random_value = ((uintptr_t)talloc_lib_init & #FFFFFFFF/);
}
talloc_magic = random_value & ~TALLOC_FLAG_MASK; } static void
    talloc_lib_atexit(void)
{
  TALLOC_FREE(autofree_context);
  if (talloc_total_size(null_context) ≡ 0) {
    return;
  }
  if (talloc_report_null_full) {
    talloc_report_full(null_context, stderr);
  }
  else if (talloc_report_null) {
    talloc_report(null_context, stderr);
  }
}
static void talloc_setup_atexit(void)
{
  static bool done;
  if (done) {
    return;
  }
  atexit(talloc_lib_atexit);
  done = true;
}
static void talloc_log(const char *fmt, ...)PRINTF_ATTRIBUTE(1, 2);
static void talloc_log(const char *fmt, ...)
{
  va_list ap;
  char *message;
  if (¬talloc_log_fn) {
    return;
  }
  va_start(ap, fmt);
```

```
    message = talloc_vasprintf(Λ, fmt, ap);
    va_end(ap);
    talloc_log_fn(message);
    talloc_free(message);
}
static void talloc_log_stderr(const char *message)
{
    fprintf(stderr, "%s", message);
}
_PUBLIC_
    void talloc_set_log_stderr(void)
    {
        talloc_set_log_fn(talloc_log_stderr);
    }
    static void(*talloc_abort_fn)(const char *reason); _PUBLIC_ void
            talloc_set_abort_fn(void(*abort_fn)(const char *reason))
        {
            talloc_abort_fn = abort_fn;
        }
        static void talloc_abort(const char *reason)
        {
            talloc_log("%s\n", reason);
            if (¬talloc_abort_fn) {
                TALLOC_ABORT(reason);
            }
            talloc_abort_fn(reason);
        }
        static void talloc_abort_access_after_free(void)
        {
            talloc_abort("Bad␣talloc␣magic␣value␣-␣access␣after␣free");
        }
        static void talloc_abort_unknown_value(void)
        {
            talloc_abort("Bad␣talloc␣magic␣value␣-␣unknown␣value");
        }    /* panic if we get a bad magic value */
        static inline struct talloc_chunk *talloc_chunk_from_ptr(const void
                *ptr)
        {
            const char *pp = (const char *) ptr;
            struct talloc_chunk *tc = discard_const_p(struct
                talloc_chunk, pp − TC_HDR_SIZE);
            if (unlikely((tc→flags & (TALLOC_FLAG_FREE | ∼TALLOC_FLAG_MASK)) ≠
                    talloc_magic)) {
                if ((tc→flags & (TALLOC_FLAG_FREE | ∼TALLOC_FLAG_MASK)) ≡
                        (TALLOC_MAGIC_NON_RANDOM | TALLOC_FLAG_FREE)) {
                    talloc_log("talloc:␣access␣after␣free␣error␣-␣first\
                        ␣free␣may␣be␣at␣%s\n", tc→name);
                    talloc_abort_access_after_free();
                    return Λ;
                }
```

```
        talloc_abort_unknown_value();
        return Λ;
      }
      return tc;
    }    /* hook into the front of the list */
#define _TLIST_ADD(list, p) do
    {
      if (¬(list)) {
        (list) = (p);
        (p)→next = (p)→prev = Λ;
      }
      else {
        (list)→prev = (p);
        (p)→next = (list);
        (p)→prev = Λ;
        (list) = (p);
      }
    }
    while (0)    /* remove an element from a list - element doesn't have
        to be in list. */
#define _TLIST_REMOVE(list, p) do
    {
      if ((p) ≡ (list)) {
        (list) = (p)→next;
        if (list) (list)→prev = Λ;
      }
      else {
        if ((p)→prev) (p)→prev→next = (p)→next;
        if ((p)→next) (p)→next→prev = (p)→prev;
      }
      if ((p) ∧ ((p) ≠ (list))) (p)→next = (p)→prev = Λ;
    }
    while (0)    /* return the parent chunk of a pointer */
    static inline struct talloc_chunk *talloc_parent_chunk(const void
            *ptr)
    {
      struct talloc_chunk *tc;
      if (unlikely(ptr ≡ Λ)) {
        return Λ;
      }
      tc = talloc_chunk_from_ptr(ptr);
      while (tc→prev) tc = tc→prev;
      return tc→parent;
    }
    _PUBLIC_ void *talloc_parent(const void *ptr)
        {
          struct talloc_chunk *tc = talloc_parent_chunk(ptr);
          return tc ? TC_PTR_FROM_CHUNK(tc) : Λ;
        }    /* find parents name */
        _PUBLIC_
            const char *talloc_parent_name(const void *ptr)
            {
```

```
    struct talloc_chunk *tc = talloc_parent_chunk(ptr);
    return tc ? tc->name : NULL;
}   /* A pool carries an in-pool object count count in the first
       16 bytes. bytes. This is done to support talloc_steal() to
       a parent outside of the pool. The count includes the pool
       itself, so a talloc_free() on a pool will only destroy the
       pool if the count has dropped to zero. A talloc_free() of
       a pool member will reduce the count, and eventually also
       call free(3) on the pool memory. The object count is
       not put into "struct talloc_chunk" because it is only
       relevant for talloc pools and the alignment to 16 bytes
       would increase the memory footprint of each talloc chunk
       by those 16 bytes. */
struct talloc_pool_hdr {
    void *end;
    unsigned int object_count;
    size_t poolsize;
};
union talloc_pool_hdr_cast_u {
    uint8_t * ptr;
    struct talloc_pool_hdr *hdr;
};
#define TP_HDR_SIZETC_ALIGN16 (sizeof(struct talloc_pool_hdr))
static inline struct talloc_pool_hdr
        *talloc_pool_from_chunk(struct talloc_chunk *c) {
        union talloc_chunk_cast_u tcc = { . chunk = c } ;
    union talloc_pool_hdr_cast_u
        tphc = {tcc.ptr - TP_HDR_SIZE};
    return tphc.hdr; } static inline struct talloc_chunk
        *talloc_chunk_from_pool(struct talloc_pool_hdr
        *h) { union talloc_pool_hdr_cast_u tphc = { .
        hdr = h } ; union talloc_chunk_cast_u tcc = {
        . ptr = tphc.ptr + TP_HDR_SIZE } ;
      return tcc.chunk; } static inline void
            *tc_pool_end(struct talloc_pool_hdr
            *pool_hdr)
      {
        struct talloc_chunk
            *tc = talloc_chunk_from_pool(pool_hdr);
        return (char *) tc+TC_HDR_SIZE+pool_hdr->poolsize;
      }
      static inline size_t tc_pool_space_left(struct
            talloc_pool_hdr *pool_hdr)
      {
        return (char *) tc_pool_end(pool_hdr) - (char *)
            pool_hdr->end;
      }   /* If tc is inside a pool, this gives the next
             neighbour. */
      static inline void *tc_next_chunk(struct
            talloc_chunk *tc)
```

```
{
  return (char *)
      tc + TC_ALIGN16(TC_HDR_SIZE + tc→size);
}
static inline void *tc_pool_first_chunk(struct
        talloc_pool_hdr *pool_hdr)
{
  struct talloc_chunk
      *tc = talloc_chunk_from_pool(pool_hdr);
  return tc_next_chunk(tc);
}     /* Mark the whole remaining pool as not
        accessable */
static inline void tc_invalidate_pool(struct
        talloc_pool_hdr *pool_hdr)
{
  size_t flen = tc_pool_space_left(pool_hdr);
  if (unlikely(talloc_fill.enabled)) {
    memset(pool_hdr→end, talloc_fill.fill_value, flen);
  }
}     /* Allocate from a pool */
static inline struct talloc_chunk
        *tc_alloc_pool(struct talloc_chunk
        *parent, size_t size, size_t prefix_len) {
        struct talloc_pool_hdr *pool_hdr = Λ;
    union talloc_chunk_cast_u tcc;
    size_t space_left;
    struct talloc_chunk *result;
    size_t chunk_size;
    if (parent ≡ Λ) {
      return Λ;
    }
    if (parent→flags & TALLOC_FLAG_POOL) {
      pool_hdr = talloc_pool_from_chunk(parent);
    }
    else if (parent→flags & TALLOC_FLAG_POOLMEM) {
      pool_hdr = parent→pool;
    }
    if (pool_hdr ≡ Λ) {
      return Λ;
    }
    space_left = tc_pool_space_left(pool_hdr);
      /* * Align size to 16 bytes */
    chunk_size = TC_ALIGN16(size + prefix_len);
    if (space_left < chunk_size) {
      return Λ;
    }
    tcc = (union talloc_chunk_cast_u) { . ptr = (
        ( uint8_t * ) pool_hdr→end ) +prefix_len } ;
    result = tcc.chunk;
    pool_hdr→end = (void *)((char *)
        pool_hdr→end + chunk_size);
```

```
result→flags = talloc_magic |
    TALLOC_FLAG_POOLMEM;
result→pool = pool_hdr;
pool_hdr→object_count++;
return result; }      /* Allocate a bit of memory
    as a child of an existing pointer */
static inline void *__talloc_with_prefix(const
        void *context, size_t size, size_t
        prefix_len, struct talloc_chunk **tc_ret)
        { struct talloc_chunk *tc = Λ;
    struct talloc_memlimit *limit = Λ;
    size_t
        total_len = TC_HDR_SIZE+size+prefix_len;
    struct talloc_chunk *parent = Λ;
    if (unlikely(context ≡ Λ)) {
      context = null_context;
    }
    if (unlikely(size ≥ MAX_TALLOC_SIZE)) {
      return Λ;
    }
    if (unlikely(total_len < TC_HDR_SIZE)) {
      return Λ;
    }
    if (likely(context ≠ Λ)) {
      parent = talloc_chunk_from_ptr(context);
      if (parent→limit ≠ Λ) {
        limit = parent→limit;
      }
      tc = tc_alloc_pool(parent,
          TC_HDR_SIZE + size, prefix_len);
    }
    if (tc ≡ Λ) { uint8_t * ptr = Λ;
    union talloc_chunk_cast_u tcc;
      /* * Only do the memlimit check/update
        on actual allocation. */
    if (¬talloc_memlimit_check(limit, total_len)) {
      errno = ENOMEM;
      return Λ;
    }
    ptr = malloc(total_len);
    if (unlikely(ptr ≡ Λ)) {
      return Λ;
    }
    tcc = (union talloc_chunk_cast_u) { .
        ptr = ptr + prefix_len } ;
    tc = tcc.chunk;
    tc→flags = talloc_magic;
    tc→pool = Λ;
    talloc_memlimit_grow(limit, total_len); }
        tc→limit = limit;
    tc→size = size;
```

$tc\rightarrow destructor = \Lambda;$
$tc\rightarrow child = \Lambda;$
$tc\rightarrow name = \Lambda;$
$tc\rightarrow refs = \Lambda;$
if $(likely(context \neq \Lambda))$ {
  if $(parent\rightarrow child)$ {
    $parent\rightarrow child\rightarrow parent = \Lambda;$
    $tc\rightarrow next = parent\rightarrow child;$
    $tc\rightarrow next\rightarrow prev = tc;$
  }
  else {
    $tc\rightarrow next = \Lambda;$
  }
  $tc\rightarrow parent = parent;$
  $tc\rightarrow prev = \Lambda;$
  $parent\rightarrow child = tc;$
}
else {
  $tc\rightarrow next = tc\rightarrow prev = tc\rightarrow parent = \Lambda;$
}
$*tc\_ret = tc;$
return TC_PTR_FROM_CHUNK(tc); } **static inline void** $*\_\_talloc$(**const void** $*context$, **size_t** $size$, **struct talloc_chunk** $**tc$)
{
  return $\_\_talloc\_with\_prefix(context, size, 0, tc);$
} /* * Create a talloc pool */
static inline void $*\_talloc\_pool$(**const void** $*context$, **size_t** $size$)
{
  struct talloc_chunk $*tc;$
  struct talloc_pool_hdr $*pool\_hdr;$
  void $*result;$
  $result = \_\_talloc\_with\_prefix(context, size,$ TP_HDR_SIZE, $\&tc);$
  if $(unlikely(result \equiv \Lambda))$ {
    return $\Lambda;$
  }
  $pool\_hdr = talloc\_pool\_from\_chunk(tc);$
  $tc\rightarrow flags$ |= TALLOC_FLAG_POOL;
  $tc\rightarrow size = 0;$
  $pool\_hdr\rightarrow object\_count = 1;$
  $pool\_hdr\rightarrow end = result;$
  $pool\_hdr\rightarrow poolsize = size;$
  $tc\_invalidate\_pool(pool\_hdr);$
  return $result;$
}
PUBLIC **void** $*talloc\_pool$(**const void** $*context$, **size_t** $size$)
  {

```
  return _talloc_pool(context, size);
}    /* * Create a talloc pool correctly
       sized for a basic size plus *
       a number of subobjects whose
       total size is given. Essentially *
       a custom allocator for talloc to
       reduce fragmentation. */
_PUBLIC_ void
             *_talloc_pooled_object(const
             void *ctx, size_t
             type_size, const char
             *type_name, unsigned
             num_subobjects, size_t
             total_subobjects_size)
    {
       size_t poolsize, subobjects_slack,
            tmp;
       struct talloc_chunk *tc;
       struct talloc_pool_hdr
            *pool_hdr;
       void *ret;
       poolsize =
            type_size + total_subobjects_size;
       if ((poolsize <
              type_size) ∨ (poolsize <
              total_subobjects_size)) {
         goto overflow;
       }
       if (num_subobjects ≡ UINT_MAX) {
         goto overflow;
       }
       num_subobjects += 1;
         /* the object body itself */
         /* * Alignment can increase the
            pool size by at most 15 bytes
            per object * plus alignment
            for the object itself */
       subobjects_slack =
            (TC_HDR_SIZE + TP_HDR_SIZE +
            15) * num_subobjects;
       if (subobjects_slack <
              num_subobjects) {
         goto overflow;
       }
       tmp = poolsize + subobjects_slack;
       if ((tmp < poolsize) ∨ (tmp <
              subobjects_slack)) {
         goto overflow;
       }
       poolsize = tmp;
       ret = _talloc_pool(ctx, poolsize);
       if (ret ≡ Λ) {
```

```
    return Λ;
  }
  tc = talloc_chunk_from_ptr(ret);
  tc→size = type_size;
  pool_hdr =
      talloc_pool_from_chunk(tc);
  pool_hdr→end = ((char
      *) pool_hdr→end +
      TC_ALIGN16(type_size));
  _tc_set_name_const(tc, type_name);
  return ret;
overflow: return Λ;
}     /* setup a destructor to be
        called on free of a pointer the
        destructor should return 0 on
        success, or -1 on failure. if the
        destructor fails then the free
        is failed, and the memory can
        be continued to be used */
_PUBLIC_ void
            _talloc_set_destructor(const
            void *ptr,
            int(*destructor)(void
            *))
    {
      struct talloc_chunk *tc =
          talloc_chunk_from_ptr(ptr);
      tc→destructor = destructor;
    }     /* increase the reference
            count on a piece of
            memory. */
    _PUBLIC_ int
                talloc_increase_ref_count(const
                void *ptr)
        {
          if
                (unlikely(¬talloc_reference(nul
                ptr))) {
            return −1;
          }
          return 0;
        }     /* helper for
                talloc_reference()
                this is referenced by
                a function pointer
                and should not be
                inline */
        static int
                talloc_reference_destructor(stru
                talloc_reference_handle
                *handle)
        {
```

```
  struct talloc_chunk
       *ptr_tc =
       talloc_chunk_from_ptr(handle→ptr[redacted]
  _TLIST_REMOVE(ptr_tc→re[redacted],
       handle);
  return 0;
}    /* more efficient way
       to add a name to a
       pointer - the name
       must point to a true
       string constant */
static inline void
         _tc_set_name_const(stru[redacted]t
         talloc_chunk
         *tc, const char
         *name)
{
  tc→name = name;
}    /* internal
       talloc_named_const( )
       */
static inline void
         *_talloc_named_const(con[redacted]t
         void *context,
         size_t size, const
         char *name)
{
  void *ptr;
  struct talloc_chunk
       *tc;
  ptr = __talloc(context,
       size, &tc);
  if (unlikely(ptr ≡ Λ)) {
    return Λ;
  }
  _tc_set_name_const(tc,
       name);
  return ptr;
}    /* make a secondary
       reference to a
       pointer, hanging off
       the given context.
       the pointer remains
       valid until both the
       original caller and
       this given context are
       freed. the major use
       for this is when two
       different structures
       need to reference
       the same underlying
```

data, and you want to be able to free the two instances separately, and in either order */

PUBLIC
void *_talloc_reference_loc*(**con**[redacted]
void *context*, **const** **void** *ptr*, **const** **char** *location*)
{
  struct talloc_chunk *tc*;
  struct talloc_reference_hand[redacted]**e** *handle*;
  if (*unlikely*(*ptr* ≡ Λ))
    return Λ;
  tc = *talloc_chunk_from_ptr*(*ptr*[redacted];
  handle = (**struct talloc_reference_hand**[redacted]**e** *) *_talloc_named_const*(*conte*[redacted], **sizeof**(**struct talloc_reference_handl**[redacted]), TALLOC_MAGIC_REFERENCE[redacted];
  if (*unlikely*(*handle* ≡ Λ))
    return Λ;
  /* note that we hang the destructor off the handle, not the main context as that allows the caller to still setup their own destructor on the context if they want to */
  talloc_set_destructor(*hand*[redacted],
    talloc_reference_destructor[redacted];

```
  handle→ptr =
      discard_const_p(voi[redacted],
      ptr);
  handle→location =
      location;
  _TLIST_ADD(tc→refs,
      handle);
  return handle→ptr;
}
static void
    ∗_talloc_steal_internal(con[redacted]t
    void ∗new_ctx,
    const void ∗ptr);
static inline void
        _tc_free_poolmem(stru[redacted]t
        talloc_chunk
        ∗tc, const
        char
        ∗location)
{
  struct
      talloc_pool_hdr
      ∗pool;
  struct talloc_chunk
      ∗pool_tc;
  void ∗next_tc;
  pool = tc→pool;
  pool_tc =
      talloc_chunk_from_pool(pool[redacted]
  next_tc =
      tc_next_chunk(tc[redacted];
  _talloc_chunk_set_free([redacted]c,
      location);
  TC_INVALIDATE_FULL_CHUNK(t[redacted]);
  if
        (unlikely(pool→object_coun
        0)) {
    talloc_abort("Pool␣object␣cou
        ero!");
    return;
  }
  pool→object_count−−;
  if
        (unlikely(pool→object_coun
        1 ∧
        ¬(pool_tc→flags[redacted]&
        TALLOC_FLAG_FREE[redacted])
    {     /∗ * if there
        is just one
        object left in
        the pool *
        and pool→flags
```

does not have
`TALLOC_FLAG_FRE`[redacted],
* it means
this is the
pool itself and
* the rest is
available for
new objects *
again. ∗/
pool→*end* =
tc_pool_first_chunk(*pool*[redacted];
tc_invalidate_pool(*poo*[redacted]);
return;
}
if
(*unlikely*(*pool*→*object_coun*
0)) { /∗
* we mark the
freed memory
with where we
called the free
* from. This
means on a
double free
error we can
report where
* the first free
came from ∗/
pool_tc→*name* =
location;
if (*pool_tc*→*flags* &
`TALLOC_FLAG_POOLME`[redacted])
{
_tc_free_poolmem(*pool_*[redacted],
location);
}
else { /∗ * The
tc_memlimit_update_on_f
* call takes
into account
the * prefix
`TP_HDR_SIZE`
allocated
before *
the pool
talloc_chun[redacted].
∗/
tc_memlimit_update_on_free(
`TC_INVALIDATE_FULL_CHUNK`
free(*pool*);
}
return;

```
  }
  if (pool→end ≡
        next_tc)
    {     /* * if
        pool-¿pool
        still points
        to end of *
        'tc' (which
        is stored in
        the 'next_tc'
        variable), * we
        can reclaim
        the memory
        of 'tc'. */
    pool→end = tc;
    return;
  }     /* * Do
        nothing. The
        memory is
        just "wasted",
        waiting for
        the pool *
        itself to be
        freed. */
}
static inline void
    _tc_free_children_internal(struct
    talloc_chunk
    *tc, void
    *ptr, const char
    *location);
static inline int
    _talloc_free_internal(void
    *ptr, const char
    *location);
  /* internal free call
    that takes a struct
    talloc_chunk *.
    */
static inline int
        _tc_free_internal(struct
        talloc_chunk
        *tc, const
        char
        *location)
{
  void *ptr_to_free;
  void *ptr =
      TC_PTR_FROM_CHUNK(tc);
  if (unlikely(tc→refs))
    {
```

int *is_child*;
/* check if this is a reference from a child or * grandchild back to it's parent or grandparent * * in that case we need to remove the reference and * call another instance of *talloc_free*( ) on the current * pointer. */
is_child = *talloc_is_parent*(*tc*→*re*[redacted], *ptr*);
_talloc_free_internal(*tc*→*re*[redacted], *location*);
if (*is_child*) {
return *_talloc_free_internal*(*p*[redacted], *location*);
}
return −1;
}
if (*unlikely*(*tc*→*flags* & TALLOC_FLAG_LOO[redacted])) { /* we have a free loop - stop looping */
return 0;
}
if (*unlikely*(*tc*→*destructo*[redacted])) {
talloc_destructor_t d[redacted]= *tc*→*destructor*;
/* * Protect the destructor against some overwrite * attacks, by explicitly checking it has the right

```
      * magic here.
      */
   if
         (talloc_chunk_from_ptr(p
         tc)
      {     /* *
         This can't
         actually
         happen,
         the * call
         itself will
         panic.  */
      TALLOC_ABORT("talloc_chu
         tr␣failed!");
   }
   if (d ≡
         (talloc_destructor_t)−
         1) {
      return −1;
   }
   tc→destructor =
       (talloc_destructor_t)−
       1;
   if (d(ptr) ≡ −1)
      {     /* * Only
         replace the
         destructor
         pointer if *
         calling the
         destructor
         didn't
         modify it.
         */
      if
            (tc→destructor ≡
            (talloc_destructor_t)−
            1) {
         tc→destructor =
            d;
      }
      return −1;
   }
   tc→destructor = Λ;
}
if (tc→parent) {
   _TLIST_REMOVE(tc→parent→chil
       tc);
   if (tc→parent→child)
      {
      tc→parent→child→parent =
         tc→parent;
   }
```

```
}
else {
  if (tc→prev)
    tc→prev→next =
        tc→next;
  if (tc→next)
    tc→next→prev =
        tc→prev;
  tc→prev =
      tc→next = Λ;
}
tc→flags |=
    TALLOC_FLAG_LOO[redacted];
_tc_free_children_internal([redacted],
    ptr, location);
_talloc_chunk_set_free([redacted],
    location);
if (tc→flags &
      TALLOC_FLAG_POO[redacted])
  {
  struct
      talloc_pool_h[redacted]r
      *pool;
  pool =
      talloc_pool_from_chunk(tc[redacted]
  if
        (unlikely(pool→object_cou
        0)) {
    talloc_abort("Pool␣object␣c
        ero!");
    return 0;
  }
  pool→object_count−[redacted];
  if
        (likely(pool→object_count
        0)) {
    return 0;
  }     /* * With
        object_count[redacted]≡
        0, a pool
        becomes
        a normal
        piece of *
        memory to
        free. If it's
        allocated
        inside a
        pool, it
        needs * to
        be freed as
        poolmem,
        else it needs
```

```
            to be just
            freed. */
    ptr_to_free = pool;
  }
  else {
    ptr_to_free = tc;
  }
  if (tc→flags &
        TALLOC_FLAG_POOLMEM)
    {
    _tc_free_poolmem(tc,
        location);
    return 0;
  }
  tc_memlimit_update_on_free(tc);
  TC_INVALIDATE_FULL_CHUNK(tc);
  free(ptr_to_free);
  return 0;
}     /* internal
        talloc_free call
        */
static inline int
        _talloc_free_internal(void
        *ptr, const
        char
        *location)
{
  struct talloc_chunk
        *tc;
  if (unlikely(ptr ≡ Λ))
    {
    return −1;
  }     /* possibly
        initialised the
        talloc fill
        value */
  if
        (unlikely(¬talloc_fill.initia
    {
    const char *fill =
        getenv(TALLOC_FILL_ENV
    if (fill ≠ Λ) {
      talloc_fill.enabled =
          true;
      talloc_fill.fill_value =
          strtoul(fill,
          Λ, 0);
    }
    talloc_fill.initialised =
        true;
  }
```

tc =
talloc_chunk_from_ptr(*ptr*[redacted];
return
_tc_free_internal([redacted],
location);
}

static inline size_t
_talloc_total_limit_size(**con**[redacted]**t**
void ∗*ptr*, **struct**
talloc_memlimit
∗*old_limit*, **struct**
talloc_memlimit
∗*new_limit*);

52. move a lump of memory from one talloc context to another return the ptr on success, or NULL if it could not be transferred. passing NULL as ptr will always return NULL with no side effects.

```
static void *_talloc_steal_internal(const void *new_ctx, const void *ptr)
{
  struct talloc_chunk *tc, *new_tc;
  size_t ctx_size = 0;
  if (unlikely(¬ptr)) {
    return Λ;
  }
  if (unlikely(new_ctx ≡ Λ)) {
    new_ctx = null_context;
  }
  tc = talloc_chunk_from_ptr(ptr);
  if (tc→limit ≠ Λ) {
    ctx_size = _talloc_total_limit_size(ptr, Λ, Λ);
      /* Decrement the memory limit from the source .. */
    talloc_memlimit_shrink(tc→limit→upper, ctx_size);
    if (tc→limit→parent ≡ tc) {
      tc→limit→upper = Λ;
    }
    else {
      tc→limit = Λ;
    }
  }
  if (unlikely(new_ctx ≡ Λ)) {
    if (tc→parent) {
      _TLIST_REMOVE(tc→parent→child, tc);
      if (tc→parent→child) {
        tc→parent→child→parent = tc→parent;
      }
    }
    else {
      if (tc→prev) tc→prev→next = tc→next;
      if (tc→next) tc→next→prev = tc→prev;
    }
    tc→parent = tc→next = tc→prev = Λ;
    return discard_const_p(void, ptr);
  }
  new_tc = talloc_chunk_from_ptr(new_ctx);
  if (unlikely(tc ≡ new_tc ∨ tc→parent ≡ new_tc)) {
    return discard_const_p(void, ptr);
  }
  if (tc→parent) {
    _TLIST_REMOVE(tc→parent→child, tc);
    if (tc→parent→child) {
      tc→parent→child→parent = tc→parent;
    }
  }
  else {
    if (tc→prev) tc→prev→next = tc→next;
    if (tc→next) tc→next→prev = tc→prev;
    tc→prev = tc→next = Λ;
```

```
    }
    tc→parent = new_tc;
    if (new_tc→child) new_tc→child→parent = Λ;
    _TLIST_ADD(new_tc→child, tc);
    if (tc→limit ∨ new_tc→limit) {
      ctx_size = _talloc_total_limit_size(ptr, tc→limit, new_tc→limit);
        /* .. and increment it in the destination. */
      if (new_tc→limit) {
        talloc_memlimit_grow(new_tc→limit, ctx_size);
      }
    }
    return discard_const_p(void, ptr);
  }   /* move a lump of memory from one talloc context to another return the ptr on success, or NULL if
         it could not be transferred. passing NULL as ptr will always return NULL with no side effects. */
  _PUBLIC_ void *_talloc_steal_loc(const void *new_ctx, const void *ptr, const char *location)
    {
      struct talloc_chunk *tc;
      if (unlikely(ptr ≡ Λ)) {
        return Λ;
      }
      tc = talloc_chunk_from_ptr(ptr);
      if (unlikely(tc→refs ≠ Λ) ∧ talloc_parent(ptr) ≠ new_ctx) {
        struct talloc_reference_handle *h;
        talloc_log("WARNING:␣talloc_steal␣with␣references␣at␣%s\n", location);
        for (h = tc→refs; h; h = h→next) {
          talloc_log("\treference␣at␣%s\n", h→location);
        }
      }
      return _talloc_steal_internal(new_ctx, ptr);
    }   /* this is like a talloc_steal(), but you must supply the old parent. This resolves the ambiguity
           in a talloc_steal() which is called on a context that has more than one parent (via references)
           The old parent can be either a reference or a parent */
    _PUBLIC_ void *talloc_reparent(const void *old_parent, const void *new_parent, const void *ptr)
      {
        struct talloc_chunk *tc;
        struct talloc_reference_handle *h;
        if (unlikely(ptr ≡ Λ)) {
          return Λ;
        }
        if (old_parent ≡ talloc_parent(ptr)) {
          return _talloc_steal_internal(new_parent, ptr);
        }
        tc = talloc_chunk_from_ptr(ptr);
        for (h = tc→refs; h; h = h→next) {
          if (talloc_parent(h) ≡ old_parent) {
            if (_talloc_steal_internal(new_parent, h) ≠ h) {
              return Λ;
            }
            return discard_const_p(void, ptr);
          }
        }   /* it wasn't a parent */
```

```
  return Λ;
}   /* remove a secondary reference to a pointer. This undo's what talloc_reference() has
      done. The context and pointer arguments must match those given to a talloc_reference()
      */
static inline int talloc_unreference(const void *context, const void *ptr)
{
  struct talloc_chunk *tc = talloc_chunk_from_ptr(ptr);
  struct talloc_reference_handle *h;
  if (unlikely(context ≡ Λ)) {
    context = null_context;
  }
  for (h = tc→refs; h; h = h→next) {
    struct talloc_chunk *p = talloc_parent_chunk(h);
    if (p ≡ Λ) {
      if (context ≡ Λ) break;
    }
    else if (TC_PTR_FROM_CHUNK(p) ≡ context) {
      break;
    }
  }
  if (h ≡ Λ) {
    return −1;
  }
  return _talloc_free_internal(h, __location__);
}   /* remove a specific parent context from a pointer. This is a more controlled variant of
      talloc_free() */   /* coverity [ −tainted_data_sink: arg − 1 ] */
_PUBLIC_
    int talloc_unlink(const void *context, void *ptr)
    {
      struct talloc_chunk *tc_p, *new_p, *tc_c;
      void *new_parent;
      if (ptr ≡ Λ) {
        return −1;
      }
      if (context ≡ Λ) {
        context = null_context;
      }
      if (talloc_unreference(context, ptr) ≡ 0) {
        return 0;
      }
      if (context ≠ Λ) {
        tc_c = talloc_chunk_from_ptr(context);
      }
      else {
        tc_c = Λ;
      }
      if (tc_c ≠ talloc_parent_chunk(ptr)) {
        return −1;
      }
      tc_p = talloc_chunk_from_ptr(ptr);
      if (tc_p→refs ≡ Λ) {
```

```
        return _talloc_free_internal(ptr, __location__);
    }
    new_p = talloc_parent_chunk(tc_p→refs);
    if (new_p) {
        new_parent = TC_PTR_FROM_CHUNK(new_p);
    }
    else {
        new_parent = Λ;
    }
    if (talloc_unreference(new_parent, ptr) ≠ 0) {
        return −1;
    }
    _talloc_steal_internal(new_parent, ptr);
    return 0;
}   /* add a name to an existing pointer - va_list version */
static inline const char *tc_set_name_v(struct talloc_chunk *tc, const char
        *fmt, va_list ap)PRINTF_ATTRIBUTE(2, 0); static inline const char
        *tc_set_name_v(struct talloc_chunk *tc, const char *fmt, va_list ap)
{
    struct talloc_chunk *name_tc = _vasprintf_tc(TC_PTR_FROM_CHUNK(tc), fmt, ap);
    if (likely(name_tc)) {
        tc→name = TC_PTR_FROM_CHUNK(name_tc);
        _tc_set_name_const(name_tc, ".name");
    }
    else {
        tc→name = Λ;
    }
    return tc→name;
}   /* add a name to an existing pointer */
_PUBLIC_ const char *talloc_set_name(const void *ptr, const char *fmt, ...)
    {
        struct talloc_chunk *tc = talloc_chunk_from_ptr(ptr);
        const char *name;
        va_list ap;
        va_start(ap, fmt);
        name = tc_set_name_v(tc, fmt, ap);
        va_end(ap);
        return name;
    }   /* create a named talloc pointer. Any talloc pointer can be named, and
          talloc_named( ) operates just like talloc( ) except that it allows you to name the
          pointer. */
    _PUBLIC_ void *talloc_named(const void *context, size_t size, const char *fmt, ...)
        {
            va_list ap;
            void *ptr;
            const char *name;
            struct talloc_chunk *tc;
            ptr = __talloc(context, size, &tc);
            if (unlikely(ptr ≡ Λ)) return Λ;
            va_start(ap, fmt);
            name = tc_set_name_v(tc, fmt, ap);
```

```
  va_end(ap);
  if (unlikely(name ≡ Λ)) {
    _talloc_free_internal(ptr, __location__);
    return Λ;
  }
  return ptr;
}      /* return the name of a talloc ptr, or "UNNAMED" */
static inline const char *__talloc_get_name(const void *ptr)
{
  struct talloc_chunk *tc = talloc_chunk_from_ptr(ptr);
  if (unlikely(tc→name ≡ TALLOC_MAGIC_REFERENCE)) {
    return ".reference";
  }
  if (likely(tc→name)) {
    return tc→name;
  }
  return "UNNAMED";
}
_PUBLIC_ const char *talloc_get_name(const void *ptr)
    {
      return __talloc_get_name(ptr);
    }      /* check if a pointer has the given name. If it does, return the pointer,
            otherwise return NULL */
    _PUBLIC_ void *talloc_check_name(const void *ptr, const char *name)
        {
          const char *pname;
          if (unlikely(ptr ≡ Λ)) return Λ;
          pname = __talloc_get_name(ptr);
          if (likely(pname ≡ name ∨ strcmp(pname, name) ≡ 0)) {
            return discard_const_p(void, ptr);
          }
          return Λ;
        }
        static void talloc_abort_type_mismatch(const char *location, const
                char *name, const char *expected)
        {
          const char *reason;
          reason = talloc_asprintf(Λ,
              "%s:␣Type␣mismatch:␣name[%s]␣expected[%s]", location,
              name ? name : "NULL", expected);
          if (¬reason) {
            reason = "Type␣mismatch";
          }
          talloc_abort(reason);
        }
        _PUBLIC_ void *_talloc_get_type_abort(const void *ptr, const char
                    *name, const char *location)
            {
              const char *pname;
              if (unlikely(ptr ≡ Λ)) {
                talloc_abort_type_mismatch(location, Λ, name);
```

```
    return Λ;
  }
  pname = __talloc_get_name(ptr);
  if (likely(pname ≡ name ∨ strcmp(pname, name) ≡ 0)) {
    return discard_const_p(void, ptr);
  }
  talloc_abort_type_mismatch(location, pname, name);
  return Λ;
}     /* this is for compatibility with older versions of talloc */
_PUBLIC_ void *talloc_init(const char *fmt, ...)
    {
      va_list ap;
      void *ptr;
      const char *name;
      struct talloc_chunk *tc;

      ptr = __talloc(Λ, 0, &tc);
      if (unlikely(ptr ≡ Λ)) return Λ;
      va_start(ap, fmt);
      name = tc_set_name_v(tc, fmt, ap);
      va_end(ap);
      if (unlikely(name ≡ Λ)) {
        _talloc_free_internal(ptr, __location__);
        return Λ;
      }
      return ptr;
    }
    static inline void _tc_free_children_internal(struct
            talloc_chunk *tc, void *ptr, const char *location)
    {
      while (tc→child) {      /* we need to work out who will own
              an abandoned child if it cannot be freed. In priority
              order, the first choice is owner of any remaining
              reference to this pointer, the second choice is our
              parent, and the final choice is the null context. */
        void *child = TC_PTR_FROM_CHUNK(tc→child);
        const void *new_parent = null_context;

        if (unlikely(tc→child→refs)) {
          struct talloc_chunk
              *p = talloc_parent_chunk(tc→child→refs);

          if (p) new_parent = TC_PTR_FROM_CHUNK(p);
        }
        if (unlikely(_tc_free_internal(tc→child, location) ≡ −1)) {
          if (talloc_parent_chunk(child) ≠ tc) {
              /* * Destructor already reparented this child. * No
                further reparenting needed. */
            continue;
          }
          if (new_parent ≡ null_context) {
            struct talloc_chunk *p = talloc_parent_chunk(ptr);

            if (p) new_parent = TC_PTR_FROM_CHUNK(p);
          }
```

```
          _talloc_steal_internal(new_parent, child);
        }
      }
    }   /* this is a replacement for the Samba3 talloc_destroy_pool
          functionality. It should probably not be used in new
          code. It's in here to keep the talloc code consistent across
          Samba 3 and 4. */
    _PUBLIC_ void talloc_free_children(void *ptr)
      {
        struct talloc_chunk *tc_name = Λ;
        struct talloc_chunk *tc;
        if (unlikely(ptr ≡ Λ)) {
          return;
        }
        tc = talloc_chunk_from_ptr(ptr);   /* we do not want
            to free the context name if it is a child .. */
        if (likely(tc→child)) {
          for (tc_name = tc→child; tc_name;
                tc_name = tc_name→next) {
            if (tc→name ≡ TC_PTR_FROM_CHUNK(tc_name))
              break;
          }
          if (tc_name) {
            _TLIST_REMOVE(tc→child, tc_name);
            if (tc→child) {
              tc→child→parent = tc;
            }
          }
        }
        _tc_free_children_internal(tc, ptr, __location__);
          /* .. so we put it back after all other children have
            been freed */
        if (tc_name) {
          if (tc→child) {
            tc→child→parent = Λ;
          }
          tc_name→parent = tc;
          _TLIST_ADD(tc→child, tc_name);
        }
      }   /* Allocate a bit of memory as a child of an existing
            pointer */
      _PUBLIC_ void *_talloc(const void *context, size_t size)
        {
          struct talloc_chunk *tc;
          return __talloc(context, size, &tc);
        }   /* externally callable talloc_set_name_const( ) */
        _PUBLIC_ void talloc_set_name_const(const void
                  *ptr, const char *name)
          {
            _tc_set_name_const(talloc_chunk_from_ptr(ptr),
                name);
```

```
}   /* create a named talloc pointer. Any talloc
      pointer can be named, and talloc_named( )
      operates just like talloc() except that it
      allows you to name the pointer. */
_PUBLIC_ void *talloc_named_const(const void
          *context, size_t size, const char
          *name)
  {
    return _talloc_named_const(context, size,
        name);
  }   /* free a talloc pointer. This also
        frees all child pointers of this pointer
        recursively return 0 if the memory is
        actually freed, otherwise -1. The
        memory will not be freed if the
        ref_count is ¿ 1 or the destructor (if
        any) returns non-zero */
  _PUBLIC_ int _talloc_free(void *ptr, const
            char *location)
    {
      struct talloc_chunk *tc;
      if (unlikely(ptr ≡ Λ)) {
        return −1;
      }
      tc = talloc_chunk_from_ptr(ptr);
      if (unlikely(tc→refs ≠ Λ)) {
        struct talloc_reference_handle
            *h;
        if (talloc_parent(ptr) ≡
              null_context ∧ tc→refs→next ≡ Λ)
          {   /* in this case we do know
              which parent should get this
              pointer, as there is really only
              one parent */
          return talloc_unlink(null_context,
              ptr);
        }
        talloc_log("ERROR:␣talloc_free␣\
            with␣references␣at␣%s\n",
            location);
        for (h = tc→refs; h; h = h→next) {
          talloc_log("\treference␣at␣%s\n",
              h→location);
        }
        return −1;
      }
      return _talloc_free_internal(ptr,
          location);
    }
```

53. A talloc version of realloc. The context argument is only used if ptr is NULL

```
_PUBLIC_ void *_talloc_realloc(const void *context, void *ptr, size_t size, const char *name)
  {
    struct talloc_chunk *tc;
    void *new_ptr;
    bool malloced = false;
    struct talloc_pool_hdr *pool_hdr = Λ;
    size_t old_size = 0;
    size_t new_size = 0;    /* size zero is equivalent to free() */
    if (unlikely(size ≡ 0)) {
      talloc_unlink(context, ptr);
      return Λ;
    }
    if (unlikely(size ≥ MAX_TALLOC_SIZE)) {
      return Λ;
    }    /* realloc(NULL) is equivalent to malloc() */
    if (ptr ≡ Λ) {
      return _talloc_named_const(context, size, name);
    }
    tc = talloc_chunk_from_ptr(ptr);    /* don't allow realloc on referenced pointers */
    if (unlikely(tc→refs)) {
      return Λ;
    }    /* don't let anybody try to realloc a talloc_pool */
    if (unlikely(tc→flags & TALLOC_FLAG_POOL)) {
      return Λ;
    }    /* handle realloc inside a talloc_pool */
    if (unlikely(tc→flags & TALLOC_FLAG_POOLMEM)) {
      pool_hdr = tc→pool;
    }    /* don't shrink if we have less than 1k to gain */
    if (size < tc→size ∧ tc→limit ≡ Λ) {
      if (pool_hdr) {
        void *next_tc = tc_next_chunk(tc);
        TC_INVALIDATE_SHRINK_CHUNK(tc, size);
        tc→size = size;
        if (next_tc ≡ pool_hdr→end) {    /* note: tc-¿size has changed, so this works */
          pool_hdr→end = tc_next_chunk(tc);
        }
        return ptr;
      }
      else if ((tc→size − size) < 1024) {    /* * if we call TC_INVALIDATE_SHRINK_CHUNK( ) here *
            we would need to call TC_UNDEFINE_GROW_CHUNK( ) * after each realloc call, which slows
            down * testing a lot :-(. * * That is why we only mark memory as undefined here. */
        TC_UNDEFINE_SHRINK_CHUNK(tc, size);    /* do not shrink if we have less than 1k to gain */
        tc→size = size;
        return ptr;
      }
    }
    else if (tc→size ≡ size) {    /* * do not change the pointer if it is exactly * the same size. */
      return ptr;
    }    /* * by resetting magic we catch users of the old memory * * We mark this memory as free,
          and also over-stamp the talloc * magic with the old-style magic. * * Why? This tries to
```

avoid a memory read use-after-free from * disclosing our talloc magic, which would then allow an * attacker to prepare a valid header and so run a destructor. * * What else? We have to re-stamp back a valid normal magic * on this memory once realloc() is done, as it will have done * a memcpy() into the new valid memory. We can't do this in * reverse as that would be a real use-after-free. */

```
_talloc_chunk_set_free(tc, Λ);
if (pool_hdr) {
  struct talloc_chunk *pool_tc;
  void *next_tc = tc_next_chunk(tc);
  size_t old_chunk_size = TC_ALIGN16(TC_HDR_SIZE + tc→size);
  size_t new_chunk_size = TC_ALIGN16(TC_HDR_SIZE + size);
  size_t space_needed;
  size_t space_left;
  unsigned int chunk_count = pool_hdr→object_count;
  pool_tc = talloc_chunk_from_pool(pool_hdr);
  if (¬(pool_tc→flags & TALLOC_FLAG_FREE)) {
    chunk_count −= 1;
  }
  if (chunk_count ≡ 1) {
      /* * optimize for the case where 'tc' is the only * chunk in the pool. */
    char *start = tc_pool_first_chunk(pool_hdr);
    space_needed = new_chunk_size;
    space_left = (char *) tc_pool_end(pool_hdr) − start;
    if (space_left ≥ space_needed) {
      size_t old_used = TC_HDR_SIZE + tc→size;
      size_t new_used = TC_HDR_SIZE + size;
      new_ptr = start;
      memmove(new_ptr, tc, old_used);
      tc = (struct talloc_chunk *) new_ptr;
      TC_UNDEFINE_GROW_CHUNK(tc, size);    /* * first we do not align the pool pointer *
          because we want to invalidate the padding * too. */
      pool_hdr→end = new_used + (char *) new_ptr;
      tc_invalidate_pool(pool_hdr);    /* now the aligned pointer */
      pool_hdr→end = new_chunk_size + (char *) new_ptr;
      goto got_new_ptr;
    }
    next_tc = Λ;
  }
  if (new_chunk_size ≡ old_chunk_size) {
    TC_UNDEFINE_GROW_CHUNK(tc, size);
    _talloc_chunk_set_not_free(tc);
    tc→size = size;
    return ptr;
  }
  if (next_tc ≡ pool_hdr→end) {
      /* * optimize for the case where 'tc' is the last * chunk in the pool. */
    space_needed = new_chunk_size − old_chunk_size;
    space_left = tc_pool_space_left(pool_hdr);
    if (space_left ≥ space_needed) {
      TC_UNDEFINE_GROW_CHUNK(tc, size);
      _talloc_chunk_set_not_free(tc);
```

```
          tc→size = size;
          pool_hdr→end = tc_next_chunk(tc);
          return ptr;
        }
      }
      new_ptr = tc_alloc_pool(tc, size + TC_HDR_SIZE, 0);
      if (new_ptr ≡ Λ) {     /* * Couldn't allocate from pool (pool size * counts as already allocated
            for memlimit * purposes). We must check memory limit * before any real malloc. */
        if (tc→limit) {     /* * Note we're doing an extra malloc, * on top of the pool size, so
              account * for size only, not the difference * between old and new size. */
          if (¬talloc_memlimit_check(tc→limit, size)) {
            _talloc_chunk_set_not_free(tc);
            errno = ENOMEM;
            return Λ;
          }
        }
        new_ptr = malloc(TC_HDR_SIZE + size);
        malloced = true;
        new_size = size;
      }
      if (new_ptr) {
        memcpy(new_ptr, tc, MIN(tc→size, size) + TC_HDR_SIZE);
        _tc_free_poolmem(tc, __location__"_talloc_realloc");
      }
    }
    else {     /* We're doing realloc here, so record the difference. */
      old_size = tc→size;
      new_size = size;     /* * We must check memory limit * before any real realloc. */
      if (tc→limit ∧ (size > old_size)) {
        if (¬talloc_memlimit_check(tc→limit, (size − old_size))) {
          _talloc_chunk_set_not_free(tc);
          errno = ENOMEM;
          return Λ;
        }
      }
      new_ptr = realloc(tc, size + TC_HDR_SIZE);
    }
  got_new_ptr:
    if (unlikely(¬new_ptr)) {
        /* * Ok, this is a strange spot. We have to put back * the old talloc_magic and any flags,
          except the * TALLOC_FLAG_FREE as this was not free'ed by the * realloc( ) call after all */
      _talloc_chunk_set_not_free(tc);
      return Λ;
    }     /* * tc is now the new value from realloc(), the old memory we * can't access any more and
          was preemptively marked as * TALLOC_FLAG_FREE before the call. Now we mark it as not *
          free again */
    tc = (struct talloc_chunk *) new_ptr;
    _talloc_chunk_set_not_free(tc);
    if (malloced) {
      tc→flags &= ~TALLOC_FLAG_POOLMEM;
    }
    if (tc→parent) {
```

```
        tc→parent→child = tc;
    }
    if (tc→child) {
        tc→child→parent = tc;
    }
    if (tc→prev) {
        tc→prev→next = tc;
    }
    if (tc→next) {
        tc→next→prev = tc;
    }
    if (new_size > old_size) {
        talloc_memlimit_grow(tc→limit, new_size − old_size);
    }
    else if (new_size < old_size) {
        talloc_memlimit_shrink(tc→limit, old_size − new_size);
    }
    tc→size = size;
    _tc_set_name_const(tc, name);
    return TC_PTR_FROM_CHUNK(tc);
}    /* a wrapper around talloc_steal( ) for situations where you are moving a pointer between two
        structures, and want the old pointer to be set to NULL */
_PUBLIC_ void *_talloc_move(const void *new_ctx, const void *_pptr)
    {
        const void **pptr = discard_const_p(const void *, _pptr);
        void *ret = talloc_steal(new_ctx, discard_const_p(void, *pptr));
        (*pptr) = Λ;
        return ret;
    }
    enum talloc_mem_count_type { TOTAL_MEM_SIZE, TOTAL_MEM_BLOCKS, TOTAL_MEM_LIMIT , }
        ;
    static inline size_t _talloc_total_mem_internal(const void *ptr,
            enum talloc_mem_count_type type, struct talloc_memlimit *old_limit, struct
            talloc_memlimit *new_limit)
    {
        size_t total = 0;
        struct talloc_chunk *c, *tc;
        if (ptr ≡ Λ) {
            ptr = null_context;
        }
        if (ptr ≡ Λ) {
            return 0;
        }
        tc = talloc_chunk_from_ptr(ptr);
        if (old_limit ∨ new_limit) {
            if (tc→limit ∧ tc→limit→upper ≡ old_limit) {
                tc→limit→upper = new_limit;
            }
        }    /* optimize in the memlimits case */
        if (type ≡ TOTAL_MEM_LIMIT ∧ tc→limit ≠ Λ ∧ tc→limit ≠ old_limit ∧ tc→limit→parent ≡ tc) {
            return tc→limit→cur_size;
```

```
    }
    if (tc→flags & TALLOC_FLAG_LOOP) {
      return 0;
    }
    tc→flags |= TALLOC_FLAG_LOOP;
    if (old_limit ∨ new_limit) {
      if (old_limit ≡ tc→limit) {
        tc→limit = new_limit;
      }
    }
    switch (type) {
    case TOTAL_MEM_SIZE:
      if (likely(tc→name ≠ TALLOC_MAGIC_REFERENCE)) {
        total = tc→size;
      }
      break;
    case TOTAL_MEM_BLOCKS: total++;
      break;
    case TOTAL_MEM_LIMIT:
      if (likely(tc→name ≠ TALLOC_MAGIC_REFERENCE)) {    /* * Don't count memory allocated
            from a pool * when calculating limits. Only count the * pool itself. */
        if (¬(tc→flags & TALLOC_FLAG_POOLMEM)) {
          if (tc→flags & TALLOC_FLAG_POOL) {    /* * If this is a pool, the allocated * size is in
                the pool header, and * remember to add in the prefix * length. */
            struct talloc_pool_hdr *pool_hdr = talloc_pool_from_chunk(tc);
            total = pool_hdr→poolsize + TC_HDR_SIZE + TP_HDR_SIZE;
          }
          else {
            total = tc→size + TC_HDR_SIZE;
          }
        }
      }
      break;
    }
    for (c = tc→child; c; c = c→next) {
      total += _talloc_total_mem_internal(TC_PTR_FROM_CHUNK(c), type, old_limit, new_limit);
    }
    tc→flags &= ~TALLOC_FLAG_LOOP;
    return total;
  }    /* return the total size of a talloc pool (subtree) */
  _PUBLIC_ size_t talloc_total_size(const void *ptr)
    {
      return _talloc_total_mem_internal(ptr, TOTAL_MEM_SIZE, Λ, Λ);
    }    /* return the total number of blocks in a talloc pool (subtree) */
    _PUBLIC_ size_t talloc_total_blocks(const void *ptr)
      {
        return _talloc_total_mem_internal(ptr, TOTAL_MEM_BLOCKS, Λ, Λ);
      }    /* return the number of external references to a pointer */
      _PUBLIC_ size_t talloc_reference_count(const void *ptr)
        {
          struct talloc_chunk *tc = talloc_chunk_from_ptr(ptr);
          struct talloc_reference_handle *h;
```

```
  size_t ret = 0;
  for (h = tc→refs; h; h = h→next) {
    ret++;
  }
  return ret;
}     /* report on memory usage by all children of a pointer, giving a full tree
        view */
_PUBLIC_ 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 *private_data), void *private_data)
  {
    struct talloc_chunk *c, *tc;
    if (ptr ≡ Λ) {
      ptr = null_context;
    }
    if (ptr ≡ Λ) return;
    tc = talloc_chunk_from_ptr(ptr);
    if (tc→flags & TALLOC_FLAG_LOOP) {
      return;
    }
    callback(ptr, depth, max_depth, 0, private_data);
    if (max_depth ≥ 0 ∧ depth ≥ max_depth) {
      return;
    }
    tc→flags |= TALLOC_FLAG_LOOP;
    for (c = tc→child; c; c = c→next) {
      if (c→name ≡ TALLOC_MAGIC_REFERENCE) {
        struct talloc_reference_handle *h = (struct
            talloc_reference_handle *) TC_PTR_FROM_CHUNK(c);
        callback(h→ptr, depth + 1, max_depth, 1, private_data);
      }
      else {
        talloc_report_depth_cb(TC_PTR_FROM_CHUNK(c), depth + 1, max_depth,
            callback, private_data);
      }
    }
    tc→flags &= ~TALLOC_FLAG_LOOP;
  }
  static void talloc_report_depth_FILE_helper(const void *ptr, int depth, int
          max_depth, int is_ref, void *_f)
  {
    const char *name = __talloc_get_name(ptr);
    struct talloc_chunk *tc;
    FILE *f = (FILE *) _f;
    if (is_ref) {
      fprintf(f, "%*sreference␣to:␣%s\n", depth * 4, "", name);
      return;
    }
    tc = talloc_chunk_from_ptr(ptr);
    if (tc→limit ∧ tc→limit→parent ≡ tc) {
```

```
    fprintf(f, "%*s%-30s is a memlimit context" " (max_size = %lu by\
        tes, cur_size = %lu bytes)\n", depth * 4, "", name, (unsigned
        long) tc→limit→max_size, (unsigned long) tc→limit→cur_size);
  }
  if (depth ≡ 0) {
    fprintf(f, "%stalloc report on '%s' (total %6lu byt\
        es in %3lu blocks)\n", (max_depth < 0 ? "full " : ""),
        name, (unsigned long) talloc_total_size(ptr), (unsigned long)
        talloc_total_blocks(ptr));
    return;
  }
  fprintf(f,
      "%*s%-30s contains %6lu bytes in %3lu blocks (ref %d) %p\n",
      depth * 4, "", name, (unsigned long) talloc_total_size(ptr), (unsigned
      long) talloc_total_blocks(ptr), (int) talloc_reference_count(ptr), ptr);
}   /* report on memory usage by all children of a pointer, giving a full
       tree view */
_PUBLIC_ void talloc_report_depth_file(const void *ptr, int depth, int
        max_depth, FILE *f)
  {
    if (f) {
      talloc_report_depth_cb(ptr, depth, max_depth,
          talloc_report_depth_FILE_helper, f);
      fflush(f);
    }
  }   /* report on memory usage by all children of a pointer, giving a
         full tree view */
  _PUBLIC_ void talloc_report_full(const void *ptr, FILE *f)
    {
      talloc_report_depth_file(ptr, 0, −1, f);
    }   /* report on memory usage by all children of a pointer */
    _PUBLIC_ void talloc_report(const void *ptr, FILE *f)
      {
        talloc_report_depth_file(ptr, 0, 1, f);
      }   /* enable tracking of the NULL context */
      _PUBLIC_ void talloc_enable_null_tracking(void)
        {
          if (null_context ≡ Λ) {
            null_context = _talloc_named_const(Λ, 0,
                "null_context");
            if (autofree_context ≠ Λ) {
              talloc_reparent(Λ, null_context, autofree_context);
            }
          }
        }
```

54. enable tracking of the NULL context, not moving the autofree context into the NULL context. This is needed for the talloc testsuite

```
_PUBLIC_ void talloc_enable_null_tracking_no_autofree(void)
{
  if (null_context ≡ Λ) {
    null_context = _talloc_named_const(Λ, 0, "null_context");
  }
}
```

55. disable tracking of the NULL context

```
_PUBLIC_ void talloc_disable_null_tracking(void)
{
  if (null_context ≠ Λ) {    /* we have to move any children onto the real NULL context */
    struct talloc_chunk *tc, *tc2;
    tc = talloc_chunk_from_ptr(null_context);
    for (tc2 = tc→child; tc2; tc2 = tc2→next) {
      if (tc2→parent ≡ tc) tc2→parent = Λ;
      if (tc2→prev ≡ tc) tc2→prev = Λ;
    }
    for (tc2 = tc→next; tc2; tc2 = tc2→next) {
      if (tc2→parent ≡ tc) tc2→parent = Λ;
      if (tc2→prev ≡ tc) tc2→prev = Λ;
    }
    tc→child = Λ;
    tc→next = Λ;
  }
  talloc_free(null_context);
  null_context = Λ;
}    /* enable leak reporting on exit */
_PUBLIC_ void talloc_enable_leak_report(void)
  {
    talloc_enable_null_tracking();
    talloc_report_null = true;
    talloc_setup_atexit();
  }    /* enable full leak reporting on exit */
  _PUBLIC_ void talloc_enable_leak_report_full(void)
    {
      talloc_enable_null_tracking();
      talloc_report_null_full = true;
      talloc_setup_atexit();
    }    /* talloc and zero memory. */
    _PUBLIC_ void *_talloc_zero(const void *ctx, size_t size, const char *name)
      {
        void *p = _talloc_named_const(ctx, size, name);
        if (p) {
          memset(p, '\0', size);
        }
        return p;
      }    /* memdup with a talloc. */
      _PUBLIC_ void *_talloc_memdup(const void *t, const void *p, size_t size, const
            char *name)
        {
          void *newp = Λ;
          if (likely(size > 0) ∧ unlikely(p ≡ Λ)) {
            return Λ;
          }
          newp = _talloc_named_const(t, size, name);
          if (likely(newp ≠ Λ) ∧ likely(size > 0)) {
            memcpy(newp, p, size);
          }
```

```
            return newp;
        }
        static inline char *__talloc_strlendup(const void *t, const char *p, size_t len)
        {
            char *ret;
            struct talloc_chunk *tc;

            ret = (char *) __talloc(t, len + 1, &tc);
            if (unlikely(¬ret)) return Λ;
            memcpy(ret, p, len);
            ret[len] = 0;
            _tc_set_name_const(tc, ret);
            return ret;
        }    /* strdup with a talloc */
        _PUBLIC_ char *talloc_strdup(const void *t, const char *p)
            {
                if (unlikely(¬p)) return Λ;
                return __talloc_strlendup(t, p, strlen(p));
            }
```

56. strndup with a talloc

```
_PUBLIC_ char *talloc_strndup(const void *t, const char *p, size_t n)
    {
        if (unlikely(¬p)) return Λ;
        return __talloc_strlendup(t, p, strnlen(p, n));
    }
    static inline char *__talloc_strlendup_append(char *s, size_t slen, const char *a, size_t alen)
    {
        char *ret;

        ret = talloc_realloc(Λ, s, char, slen + alen + 1);
        if (unlikely(¬ret)) return Λ;    /* append the string and the trailing T0 */
        memcpy(&ret[slen], a, alen);
        ret[slen + alen] = 0;
        _tc_set_name_const(talloc_chunk_from_ptr(ret), ret);
        return ret;
    }    /* * Appends at the end of the string. */
    _PUBLIC_ char *talloc_strdup_append(char *s, const char *a)
        {
            if (unlikely(¬s)) {
                return talloc_strdup(Λ, a);
            }
            if (unlikely(¬a)) {
                return s;
            }
            return __talloc_strlendup_append(s, strlen(s), a, strlen(a));
        }
```

57. Appends at the end of the talloc'ed buffer, not the end of the string.

```
_PUBLIC_ char *talloc_strdup_append_buffer(char *s, const char *a)
{
  size_t slen;
  if (unlikely(¬s)) {
    return talloc_strdup(Λ, a);
  }
  if (unlikely(¬a)) {
    return s;
  }
  slen = talloc_get_size(s);
  if (likely(slen > 0)) {
    slen--;
  }
  return __talloc_strlendup_append(s, slen, a, strlen(a));
}
```

58. Appends at the end of the string.

```
_PUBLIC_ char *talloc_strndup_append(char *s, const char *a, size_t n)
{
  if (unlikely(¬s)) {
    return talloc_strndup(Λ, a, n);
  }
  if (unlikely(¬a)) {
    return s;
  }
  return __talloc_strlendup_append(s, strlen(s), a, strnlen(a, n));
}
```

59. Appends at the end of the talloc'ed buffer, not the end of the string.

```
_PUBLIC_
   char *talloc_strndup_append_buffer(char *s, const char *a, size_t n)
   {
     size_t slen;
     if (unlikely(¬s)) {
       return talloc_strndup(Λ, a, n);
     }
     if (unlikely(¬a)) {
       return s;
     }
     slen = talloc_get_size(s);
     if (likely(slen > 0)) {
       slen−−;
     }
     return __talloc_strlendup_append(s, slen, a, strnlen(a, n));
   }
   static struct talloc_chunk *_vasprintf_tc(const void *t, const char *fmt, va_list
           ap)PRINTF_ATTRIBUTE(2, 0); static struct talloc_chunk *_vasprintf_tc(const void
           *t, const char *fmt, va_list ap)
   {
     int vlen;
     size_t len;
     char *ret;
     va_list ap2;
     struct talloc_chunk *tc;
     char buf[1024];     /* this call looks strange, but it makes it work on older solaris boxes */
     va_copy(ap2, ap);
     vlen = vsnprintf(buf, sizeof (buf), fmt, ap2);
     va_end(ap2);
     if (unlikely(vlen < 0)) {
       return Λ;
     }
     len = vlen;
     if (unlikely(len + 1 < len)) {
       return Λ;
     }
     ret = (char *) __talloc(t, len + 1, &tc);
     if (unlikely(¬ret)) return Λ;
     if (len < sizeof (buf)) {
       memcpy(ret, buf, len + 1);
     }
     else {
       va_copy(ap2, ap);
       vsnprintf(ret, len + 1, fmt, ap2);
       va_end(ap2);
     }
     _tc_set_name_const(tc, ret);
     return tc;
   }
   _PUBLIC_ char *talloc_vasprintf(const void *t, const char *fmt, va_list ap)
```

```
{
  struct talloc_chunk *tc = _vasprintf_tc(t, fmt, ap);
  if (tc ≡ Λ) {
    return Λ;
  }
  return TC_PTR_FROM_CHUNK(tc);
}
```

60. Perform string formatting, and return a pointer to newly allocated memory holding the result, inside a memory pool.

```
_PUBLIC_
  char *talloc_asprintf(const void *t, const char *fmt, ...)
  {
    va_list ap;
    char *ret;
    va_start(ap, fmt);
    ret = talloc_vasprintf(t, fmt, ap);
    va_end(ap);
    return ret;
  }
  static inline char *__talloc_vaslenprintf_append(char *s, size_t slen, const char *fmt, va_list
        ap)PRINTF_ATTRIBUTE(3, 0); static inline char *__talloc_vaslenprintf_append(char
        *s, size_t slen, const char *fmt, va_list ap)
  {
    ssize_t alen;
    va_list ap2;
    char c;
    va_copy(ap2, ap);
    alen = vsnprintf(&c, 1, fmt, ap2);
    va_end(ap2);
    if (alen ≤ 0) {
        /* Either the vsnprintf failed or the format resulted in * no characters being formatted. In
          the former case, we * ought to return NULL, in the latter we ought to return * the original
          string. Most current callers of this * function expect it to never return NULL. */
      return s;
    }
    s = talloc_realloc(Λ, s, char, slen + alen + 1);
    if (¬s) return Λ;
    va_copy(ap2, ap);
    vsnprintf(s + slen, alen + 1, fmt, ap2);
    va_end(ap2);
    _tc_set_name_const(talloc_chunk_from_ptr(s), s);
    return s;
  }
```

61. Realloc @p s to append the formatted result of @p fmt and @p ap, and return @p s, which may have moved. Good for gradually accumulating output into a string buffer. Appends at the end of the string.

```
_PUBLIC_ char *talloc_vasprintf_append(char *s, const char *fmt, va_list ap)
{
  if (unlikely(!s)) {
    return talloc_vasprintf(NULL, fmt, ap);
  }
  return __talloc_vaslenprintf_append(s, strlen(s), fmt, ap);
}
```

62. Realloc @p s to append the formatted result of @p fmt and @p ap, and return @p s, which may have moved. Always appends at the end of the talloc'ed buffer, not the end of the string.

```
_PUBLIC_ char *talloc_vasprintf_append_buffer(char *s, const char *fmt, va_list ap)
{
  size_t slen;
  if (unlikely(!s)) {
    return talloc_vasprintf(NULL, fmt, ap);
  }
  slen = talloc_get_size(s);
  if (likely(slen > 0)) {
    slen--;
  }
  return __talloc_vaslenprintf_append(s, slen, fmt, ap);
}
```

63. Realloc @p s to append the formatted result of @p fmt and return @p s, which may have moved. Good for gradually accumulating output into a string buffer.

```
_PUBLIC_ char *talloc_asprintf_append(char *s, const char *fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  s = talloc_vasprintf_append(s, fmt, ap);
  va_end(ap);
  return s;
}
```

64. Realloc @p s to append the formatted result of @p fmt and return @p s, which may have moved. Good for gradually accumulating output into a buffer.

```
_PUBLIC_ char *talloc_asprintf_append_buffer(char *s, const char *fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  s = talloc_vasprintf_append_buffer(s, fmt, ap);
  va_end(ap);
  return s;
}
```

65. alloc an array, checking for integer overflow in the array size

```
_PUBLIC_ void *_talloc_array(const void *ctx, size_t el_size, unsigned count, const char *name)
    {
      if (count ≥ MAX_TALLOC_SIZE/el_size) {
        return Λ;
      }
      return _talloc_named_const(ctx, el_size * count, name);
    }
```

66. alloc an zero array, checking for integer overflow in the array size

```
_PUBLIC_ void *_talloc_zero_array(const void *ctx, size_t el_size, unsigned count, const char *name)
    {
      if (count ≥ MAX_TALLOC_SIZE/el_size) {
        return Λ;
      }
      return _talloc_zero(ctx, el_size * count, name);
    }
```

67. realloc an array, checking for integer overflow in the array size

```
_PUBLIC_ void *_talloc_realloc_array(const void *ctx, void *ptr, size_t el_size, unsigned count, const
        char *name)
    {
      if (count ≥ MAX_TALLOC_SIZE/el_size) {
        return Λ;
      }
      return _talloc_realloc(ctx, ptr, el_size * count, name);
    }
```

68. a function version of *talloc_realloc*( ), so it can be passed as a function pointer to libraries that want a realloc function (a realloc function encapsulates all the basic capabilities of an allocation library, which is why this is useful)

```
_PUBLIC_ void *talloc_realloc_fn(const void *context, void *ptr, size_t size)
    {
      return _talloc_realloc(context, ptr, size, Λ);
    }
    static int talloc_autofree_destructor(void *ptr)
    {
      autofree_context = Λ;
      return 0;
    }
```

69. return a context which will be auto-freed on exit this is useful for reducing the noise in leak reports

```
_PUBLIC_ void *talloc_autofree_context(void)
  {
    if (autofree_context ≡ Λ) {
      autofree_context = _talloc_named_const(Λ, 0, "autofree_context");
      talloc_set_destructor(autofree_context, talloc_autofree_destructor);
      talloc_setup_atexit();
    }
    return autofree_context;
  }
  _PUBLIC_ size_t talloc_get_size(const void *context)
    {
      struct talloc_chunk *tc;
      if (context ≡ Λ) {
        return 0;
      }
      tc = talloc_chunk_from_ptr(context);
      return tc→size;
    }
```

70. find a parent of this context that has the given name, if any

```
_PUBLIC_ void *talloc_find_parent_byname(const void *context, const char *name)
  {
    struct talloc_chunk *tc;
    if (context ≡ Λ) {
      return Λ;
    }
    tc = talloc_chunk_from_ptr(context);
    while (tc) {
      if (tc→name ∧ strcmp(tc→name, name) ≡ 0) {
        return TC_PTR_FROM_CHUNK(tc);
      }
      while (tc ∧ tc→prev) tc = tc→prev;
      if (tc) {
        tc = tc→parent;
      }
    }
    return Λ;
  }
```

71. show the parentage of a context

```
_PUBLIC_ void talloc_show_parents(const void *context, FILE *file)
    {
      struct talloc_chunk *tc;
      if (context == NULL) {
        fprintf(file, "talloc no parents for NULL\n");
        return;
      }
      tc = talloc_chunk_from_ptr(context);
      fprintf(file, "talloc parents of '%s'\n", __talloc_get_name(context));
      while (tc) {
        fprintf(file, "\t'%s'\n", __talloc_get_name(TC_PTR_FROM_CHUNK(tc)));
        while (tc && tc->prev) tc = tc->prev;
        if (tc) {
          tc = tc->parent;
        }
      }
      fflush(file);
    }
```

72. return 1 if ptr is a parent of context

```
static int _talloc_is_parent(const void *context, const void *ptr, int depth)
{
  struct talloc_chunk *tc;
  if (context == NULL) {
    return 0;
  }
  tc = talloc_chunk_from_ptr(context);
  while (tc) {
    if (depth <= 0) {
      return 0;
    }
    if (TC_PTR_FROM_CHUNK(tc) == ptr) return 1;
    while (tc && tc->prev) tc = tc->prev;
    if (tc) {
      tc = tc->parent;
      depth--;
    }
  }
  return 0;
}
```

73. return 1 if ptr is a parent of context

```
_PUBLIC_ int talloc_is_parent(const void *context, const void *ptr)
    {
      return _talloc_is_parent(context, ptr, TALLOC_MAX_DEPTH);
    }
```

74. return the total size of memory used by this context and all children

static inline size_t $\_talloc\_total\_limit\_size$(**const void** $*ptr$, **struct talloc_memlimit** $*old\_limit$, **struct talloc_memlimit** $*new\_limit$)
{
  return $\_talloc\_total\_mem\_internal$($ptr$, `TOTAL_MEM_LIMIT`, old_limit, new_limit);
}

static inline bool $talloc\_memlimit\_check$(**struct talloc_memlimit** $*limit$, **size_t** $size$)
{
  struct talloc_memlimit $*l$;
  for ($l = limit$; $l \neq \Lambda$; $l = l\rightarrow upper$) {
    if ($l\rightarrow max\_size \neq 0 \wedge ((l\rightarrow max\_size \leq l\rightarrow cur\_size) \vee (l\rightarrow max\_size - l\rightarrow cur\_size < size))$) {
      return $false$;
    }
  }
  return $true$;
}

75. Update memory limits when freeing a **talloc_chunk**.

static void $tc\_memlimit\_update\_on\_free$(**struct talloc_chunk** $*tc$)
{
  size_t $limit\_shrink\_size$;
  if ($\neg tc\rightarrow limit$) {
    return;
  } /* * Pool entries don't count. Only the pools * themselves are counted as part of the memory * limits. Note that this also takes care of * nested pools which have both flags * TALLOC FLAG POOLMEM TALLOC FLAG POOL set. */
  if ($tc\rightarrow flags$ & `TALLOC_FLAG_POOLMEM`) {
    return;
  } /* * If we are part of a memory limited context hierarchy * we need to subtract the memory used from the counters */
  $limit\_shrink\_size = tc\rightarrow size +$ `TC_HDR_SIZE`;
    /* * If we're deallocating a pool, take into * account the prefix size added for the pool. */
  if ($tc\rightarrow flags$ & `TALLOC_FLAG_POOL`) {
    $limit\_shrink\_size$ += `TP_HDR_SIZE`;
  }
  $talloc\_memlimit\_shrink$($tc\rightarrow limit$, $limit\_shrink\_size$);
  if ($tc\rightarrow limit\rightarrow parent \equiv tc$) {
    $free$($tc\rightarrow limit$);
  }
  $tc\rightarrow limit = \Lambda$;
}

76. Increase memory limit accounting after a malloc/realloc.

```
static void talloc_memlimit_grow(struct talloc_memlimit *limit, size_t size)
{
  struct talloc_memlimit *l;
  for (l = limit; l ≠ Λ; l = l→upper) {
    size_t new_cur_size = l→cur_size + size;
    if (new_cur_size < l→cur_size) {
      talloc_abort("logic␣error␣in␣talloc_memlimit_grow\n");
      return;
    }
    l→cur_size = new_cur_size;
  }
}
```

77. Decrease memory limit accounting after a free/realloc.

```
static void talloc_memlimit_shrink(struct talloc_memlimit *limit, size_t size)
{
  struct talloc_memlimit *l;
  for (l = limit; l ≠ Λ; l = l→upper) {
    if (l→cur_size < size) {
      talloc_abort("logic␣error␣in␣talloc_memlimit_shrink\n");
      return;
    }
    l→cur_size = l→cur_size − size;
  }
}
_PUBLIC_ int talloc_set_memlimit(const void *ctx, size_t max_size)
    {
      struct talloc_chunk *tc = talloc_chunk_from_ptr(ctx);
      struct talloc_memlimit *orig_limit;
      struct talloc_memlimit *limit = Λ;
      if (tc→limit ∧ tc→limit→parent ≡ tc) {
        tc→limit→max_size = max_size;
        return 0;
      }
      orig_limit = tc→limit;
      limit = malloc(sizeof(struct talloc_memlimit));
      if (limit ≡ Λ) {
        return 1;
      }
      limit→parent = tc;
      limit→max_size = max_size;
      limit→cur_size = _talloc_total_limit_size(ctx, tc→limit, limit);
      if (orig_limit) {
        limit→upper = orig_limit;
      }
      else {
        limit→upper = Λ;
      }
      return 0;
    }
```

78. Index

TALLOC