My Project

Generated by Doxygen 1.8.14

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 __debug_new_counter Class Reference

`#include <debug_new.h>`

Public Member Functions

- __debug_new_counter ()
- ~__debug_new_counter ()

4.1.1 Detailed Description

Counter class for on-exit leakage check.

This technique is learnt from *The C++ Programming Language* by Bjarne Stroustup.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 __debug_new_counter()

`__debug_new_counter::__debug_new_counter ( )`

Constructor to increment the count.

4.1.2.2 ~__debug_new_counter()

`__debug_new_counter::~__debug_new_counter ( )`

Destructor to decrement the count. When the count is zero, check_leaks will be called.

The documentation for this class was generated from the following files:

- debug_new.h
- debug_new.cpp

4.2 __debug_new_recorder Class Reference

`#include <debug_new.h>`

Public Member Functions

- __debug_new_recorder (const char *file, int line)
- template<class _Tp >
  _Tp * operator-> * (_Tp *pointer)

4.2.1 Detailed Description

Recorder class to remember the call context.

The idea comes from `Greg Herlihy's post` in comp.lang.c++.moderated.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 __debug_new_recorder()

```
__debug_new_recorder::__debug_new_recorder (
            const char * file,
            int line )  [inline]
```

Constructor to remember the call context. The information will be used in __debug_new_recorder::operator->*.

4.2.3 Member Function Documentation

4.2.3.1 operator-> *()

```
template<class _Tp >
_Tp* __debug_new_recorder::operator->* (
            _Tp * pointer )  [inline]
```

Operator to write the context information to memory. `operator->*` is chosen because it has the right precedence, it is rarely used, and it looks good: so people can tell the special usage more quickly.

The documentation for this class was generated from the following files:

- debug_new.h
- debug_new.cpp

4.3 __nvwa_compile_time_error< bool > Struct Template Reference

The documentation for this struct was generated from the following file:

- static_assert.h

4.4 __nvwa_compile_time_error< true > Struct Template Reference

The documentation for this struct was generated from the following file:

- static_assert.h

4.5 mem_pool_base::_Block_list Struct Reference

`#include <mem_pool_base.h>`

Collaboration diagram for mem_pool_base::_Block_list:

Public Attributes

- _Block_list * **_M_next**

4.5.1 Detailed Description

Structure to store the next available memory block.

The documentation for this struct was generated from the following file:

- mem_pool_base.h

4.6 bool_array Class Reference

`#include <bool_array.h>`

Public Member Functions

- bool_array (unsigned long __size)
- bool create (unsigned long __size)
- void initialize (bool __value)
- _Element operator[ ] (unsigned long __idx)
- bool at (unsigned long __idx) const
- void reset (unsigned long __idx)
- void set (unsigned long __idx)
- unsigned long **size** () const
- unsigned long count () const
- unsigned long count (unsigned long __beg, unsigned long __end) const
- void flip ()

4.6.1 Detailed Description

Class to represent a packed boolean array.

This was first written in April 1995, before I knew of any existing implementation of this kind of classes. Of course, the C++ Standard Template Library now demands an implementation of packed boolean array as 'vector<bool>', but the code here should still be useful for the following three reasons: (1) STL support of MSVC 6 did not implement this specialization (nor did it have a 'bit_vector'); (2) I incorporated some useful member functions from the STL bitset into this 'bool_array', including 'reset', 'set', 'flip', and 'count'; (3) In my tests under MSVC 6 and GCC 2.95.3/3.2.3 my code is really FASTER than vector<bool> or the normal boolean array.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 bool_array()

```
bool_array::bool_array (
            unsigned long __size )  [inline], [explicit]
```

Constructs the packed boolean array with a specific size.

Parameters

__size	size of the array

Exceptions

std::out_of_range	if *__size* equals 0
std::bad_alloc	if memory is insufficient

4.6.3 Member Function Documentation

4.6.3.1 at()

```
bool bool_array::at (
            unsigned long __idx ) const  [inline]
```

Reads the boolean value of an array element via an index.

Parameters

__idx	index of the array element to access

Returns

the boolean value of the accessed array element

Exceptions

std::out_of_range	when the index is too big

4.6.3.2 count() [1/2]

```
unsigned long bool_array::count ( ) const
```

Counts elements with a `true` value.

Returns

the count of `true` elements

4.6.3.3 count() [2/2]

```
unsigned long bool_array::count (
            unsigned long __beg,
            unsigned long __end ) const
```

Counts elements with a `true` value in a specified range.

Parameters

__beg	beginning of the range
__end	end of the range (exclusive)

Returns

the count of `true` elements

4.6.3.4 create()

```
bool bool_array::create (
            unsigned long __size )
```

Creates the packed boolean array with a specific size.

Parameters

__size	size of the array

Returns

`false` if *__size* equals `0` or is too big, or if memory is insufficient; `true` if *__size* has a suitable value and memory allocation is successful.

4.6.3.5 flip()

```
void bool_array::flip ( )
```

Changes all `true` elements to `false`, and `false` ones to `true`.

4.6.3.6 initialize()

```
void bool_array::initialize (
            bool __value )
```

Initializes all array elements to a specific value optimally.

Parameters

__value	the boolean value to assign to all elements

4.6.3.7 operator[]()

```
bool_array::_Element bool_array::operator[] (
            unsigned long __idx )  [inline]
```

Creates a reference to an array element.

Parameters

__idx	index of the array element to access

4.6.3.8 reset()

```
void bool_array::reset (
            unsigned long __idx )  [inline]
```

Resets an array element to `false` via an index.

Parameters

__idx	index of the array element to access

Exceptions

std::out_of_range	when the index is too big

4.6.3.9 set()

```
void bool_array::set (
            unsigned long __idx )  [inline]
```

Sets an array element to `true` via an index.

Parameters

__idx	index of the array element to access

Exceptions

std::out_of_range	when the index is too big

The documentation for this class was generated from the following files:

- bool_array.h
- bool_array.cpp

4.7 class_level_lock< _Host, _RealLock > Class Template Reference

`#include <class_level_lock.h>`

Classes

- class lock

Public Types

- typedef _Host **volatile_type**

4.7.1 Detailed Description

template<class _Host, bool _RealLock = false>
class class_level_lock< _Host, _RealLock >

Helper class for class-level locking. This is the single-threaded implementation.

The documentation for this class was generated from the following file:

- class_level_lock.h

4.8 delete_object Struct Reference

`#include <cont_ptr_utils.h>`

Public Member Functions

- template<typename _Pointer >
  void **operator()** (_Pointer __ptr) const

4.8.1 Detailed Description

Functor to delete objects pointed by a container of pointers.

A typical usage might be like:

```
list<Object*> l;
...
for_each(l.begin(), l.end(), delete_object());
```

The documentation for this struct was generated from the following file:

- cont_ptr_utils.h

4.9 dereference Struct Reference

`#include <cont_ptr_utils.h>`

Public Member Functions

- template<typename _Tp >
  const _Tp & **operator()** (const _Tp *__ptr) const

4.9.1 Detailed Description

Functor to return objects pointed by a container of pointers.

A typical usage might be like:

```
vector<Object*> v;
...
transform(v.begin(), v.end(),
          ostream_iterator<Object>(cout, " "),
          dereference());
```

The documentation for this struct was generated from the following file:

- cont_ptr_utils.h

4.10 dereference_less Struct Reference

```
#include <cont_ptr_utils.h>
```

Public Member Functions

- template<typename _Pointer >
  bool **operator()** (_Pointer __ptr1, _Pointer __ptr2) const

4.10.1 Detailed Description

Functor to compare objects pointed by a container of pointers.

```
vector<Object*> v;
...
sort(v.begin(), v.end(), dereference_less());
```

or

```
set<Object*, dereference_less> s;
```

The documentation for this struct was generated from the following file:

- cont_ptr_utils.h

4.11 fast_mutex Class Reference

`#include <fast_mutex.h>`

Public Member Functions

- void **lock** ()
- void **unlock** ()

4.11.1 Detailed Description

Class for non-reentrant fast mutexes. This is the null implementation for single-threaded environments.

The documentation for this class was generated from the following file:

- fast_mutex.h

4.12 fast_mutex_autolock Class Reference

`#include <fast_mutex.h>`

Public Member Functions

- **fast_mutex_autolock** (fast_mutex &__mtx)

4.12.1 Detailed Description

An acquistion-on-initialization lock class based on fast_mutex.

The documentation for this class was generated from the following file:

- fast_mutex.h

4.13 fixed_mem_pool< _Tp > Class Template Reference

`#include <fixed_mem_pool.h>`

Public Types

- typedef class_level_lock< fixed_mem_pool< _Tp > >::lock **lock**

Static Public Member Functions

- static void * allocate ()
- static void deallocate (void *)
- static bool initialize (size_t __size)
- static int deinitialize ()
- static int get_alloc_count ()
- static bool is_initialized ()

Static Protected Member Functions

- static bool bad_alloc_handler ()

4.13.1 Detailed Description

template<class _Tp>
class fixed_mem_pool< _Tp >

Class template to manipulate a fixed-size memory pool. Please notice that only allocate and deallocate are protected by a lock.

Parameters

_Tp	class to use the fixed_mem_pool

4.13.2 Member Function Documentation

4.13.2.1 allocate()

```
template<class _Tp >
void * fixed_mem_pool< _Tp >::allocate ( )  [inline], [static]
```

Allocates a memory block from the memory pool.

Returns

pointer to the allocated memory block

4.13.2.2 bad_alloc_handler()

```
template<class _Tp >
bool fixed_mem_pool< _Tp >::bad_alloc_handler ( )  [static], [protected]
```

Bad allocation handler. Called when there are no memory blocks available in the memory pool. If this function returns `false` (default behaviour if not explicitly specialized), it indicates that it can do nothing and allocate() should return `NULL`; if this function returns `true`, it indicates that it has freed some memory blocks and allocate() should try allocating again.

4.13.2.3 deallocate()

```
template<class _Tp >
void fixed_mem_pool< _Tp >::deallocate (
            void * __block_ptr )  [inline], [static]
```

Deallocates a memory block and returns it to the memory pool.

Parameters

__block_ptr	pointer to the memory block to return

4.13.2.4 deinitialize()

```
template<class _Tp >
int fixed_mem_pool< _Tp >::deinitialize ( )  [static]
```

Deinitializes the memory pool.

Returns

0 if all memory blocks are returned and the memory pool successfully freed; or a non-zero value indicating number of memory blocks still in allocation

4.13.2.5 get_alloc_count()

```
template<class _Tp >
int fixed_mem_pool< _Tp >::get_alloc_count ( )  [inline], [static]
```

Gets the allocation count.

Returns

the number of memory blocks still in allocation

4.13.2.6 initialize()

```
template<class _Tp >
bool fixed_mem_pool< _Tp >::initialize (
            size_t __size )  [static]
```

Initializes the memory pool.

Parameters

__size	number of memory blocks to put in the memory pool

Returns

`true` if successful; `false` if memory insufficient

4.13.2.7 is_initialized()

```
template<class _Tp >
bool fixed_mem_pool< _Tp >::is_initialized ( )  [inline], [static]
```

Is the memory pool initialized?

Returns

`true` if it is successfully initialized; `false` otherwise

The documentation for this class was generated from the following file:

- fixed_mem_pool.h

4.14 class_level_lock< _Host, _RealLock >::lock Class Reference

`#include <class_level_lock.h>`

4.14.1 Detailed Description

template<class _Host, bool _RealLock = false>
class class_level_lock< _Host, _RealLock >::lock

Type that provides locking/unlocking semantics.

The documentation for this class was generated from the following file:

- class_level_lock.h

4.15 object_level_lock< _Host >::lock Class Reference

`#include <object_level_lock.h>`

Public Member Functions

- **lock** (const object_level_lock &__host)
- const object_level_lock * **get_locked_object** () const

4.15.1 Detailed Description

template<class _Host>
class object_level_lock< _Host >::lock

Type that provides locking/unlocking semantics.

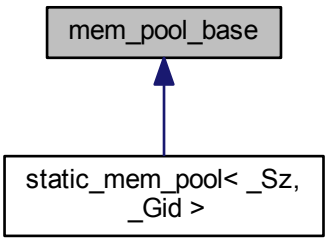
The documentation for this class was generated from the following file:

- object_level_lock.h

4.16 mem_pool_base Class Reference

`#include <mem_pool_base.h>`

Inheritance diagram for mem_pool_base:



Classes

- struct _Block_list

Public Member Functions

- virtual void **recycle** ()=0

Static Public Member Functions

- static void * **alloc_sys** (size_t __size)
- static void **dealloc_sys** (void *__ptr)

4.16.1 Detailed Description

Base class for memory pools.

The documentation for this class was generated from the following files:

- mem_pool_base.h
- mem_pool_base.cpp

4.17 new_ptr_list_t Struct Reference

Collaboration diagram for new_ptr_list_t:

Public Attributes

- new_ptr_list_t * **next**
- new_ptr_list_t * **prev**
- size_t **size**
- union {
  char **file** [_DEBUG_NEW_FILENAME_LEN]
  void * **addr**
  };
- unsigned **line**:31
- unsigned **is_array**:1
- unsigned **magic**

4.17.1 Detailed Description

Structure to store the position information where `new` occurs.

The documentation for this struct was generated from the following file:

- debug_new.cpp

4.18 object_level_lock< _Host > Class Template Reference

`#include <object_level_lock.h>`

Classes

- class lock

Public Types

- typedef _Host **volatile_type**

4.18.1 Detailed Description

template<class _Host>
class object_level_lock< _Host >

Helper class for object-level locking. This is the single-threaded implementation.

The documentation for this class was generated from the following file:

- object_level_lock.h

4.19 output_object< _OutputStrm, _StringType > Struct Template Reference

`#include <cont_ptr_utils.h>`

Public Member Functions

- **output_object** (_OutputStrm &__outs, const _StringType &__sep)
- template<typename _Tp >
  void **operator()** (const _Tp *__ptr) const

4.19.1 Detailed Description

template<typename _OutputStrm, typename _StringType = const char*>
struct output_object< _OutputStrm, _StringType >

Functor to output objects pointed by a container of pointers.

A typical usage might be like:

```
list<Object*> l;
...
for_each(l.begin(), l.end(), output_object<ostream>(cout, " "));
```

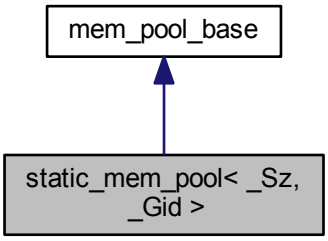
The documentation for this struct was generated from the following file:

- cont_ptr_utils.h

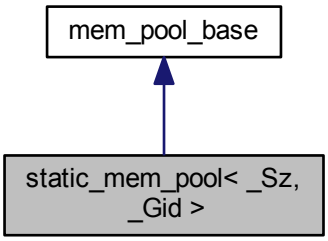
4.20 static_mem_pool< _Sz, _Gid > Class Template Reference

`#include <static_mem_pool.h>`

Inheritance diagram for static_mem_pool< _Sz, _Gid >:



Collaboration diagram for static_mem_pool< _Sz, _Gid >:



Public Member Functions

- void * allocate ()
- void deallocate (void *__ptr)
- virtual void recycle ()

Static Public Member Functions

- static static_mem_pool & instance ()
- static static_mem_pool & instance_known ()

4.20.1 Detailed Description

template<size_t _Sz, int _Gid = -1>
class static_mem_pool< _Sz, _Gid >

Singleton class template to manage the allocation/deallocation of memory blocks of one specific size.

Parameters

_Sz	size of elements in the static_mem_pool
_Gid	group id of a static_mem_pool: if it is negative, simultaneous accesses to this static_mem_pool will be protected from each other; otherwise no protection is given

4.20.2 Member Function Documentation

4.20.2.1 allocate()

```
template<size_t _Sz, int _Gid = -1>
void* static_mem_pool< _Sz, _Gid >::allocate ( )  [inline]
```

Allocates memory and returns its pointer. The template will try to get it from the memory pool first, and request memory from the system if there is no free memory in the pool.

Returns

pointer to allocated memory if successful; `NULL` otherwise

4.20.2.2 deallocate()

```
template<size_t _Sz, int _Gid = -1>
void static_mem_pool< _Sz, _Gid >::deallocate (
            void * __ptr )  [inline]
```

Deallocates memory by putting the memory block into the pool.

Parameters

__ptr	pointer to memory to be deallocated

4.20.2.3 instance()

```
template<size_t _Sz, int _Gid = -1>
static static_mem_pool& static_mem_pool< _Sz, _Gid >::instance ( )  [inline], [static]
```

Gets the instance of the static memory pool. It will create the instance if it does not already exist. Generally this function is now not needed.

Returns

reference to the instance of the static memory pool

See also

instance_known

4.20.2.4 instance_known()

```
template<size_t _Sz, int _Gid = -1>
static static_mem_pool& static_mem_pool< _Sz, _Gid >::instance_known ( )  [inline], [static]
```

Gets the known instance of the static memory pool. The instance must already exist. Generally the static initializer of the template guarantees it.

Returns

reference to the instance of the static memory pool

4.20.2.5 recycle()

```
template<size_t _Sz, int _Gid>
void static_mem_pool< _Sz, _Gid >::recycle ( )  [virtual]
```

Recycles half of the free memory blocks in the memory pool to the system. It is called when a memory request to the system (in other instances of the static memory pool) fails.

Implements mem_pool_base.

The documentation for this class was generated from the following file:

- static_mem_pool.h

4.21 static_mem_pool_set Class Reference

```
#include <static_mem_pool.h>
```

Public Types

- typedef class_level_lock< static_mem_pool_set >::lock **lock**

Public Member Functions

- void recycle ()
- void add (mem_pool_base *__memory_pool_p)

Static Public Member Functions

- static static_mem_pool_set & instance ()

Public Attributes

- __PRIVATE **__pad0__**: ~static_mem_pool_set()

4.21.1 Detailed Description

Singleton class to maintain a set of existing instantiations of static_mem_pool.

4.21.2 Member Function Documentation

4.21.2.1 add()

```
void static_mem_pool_set::add (
            mem_pool_base * __memory_pool_p )
```

Adds a new memory pool to static_mem_pool_set.

Parameters

__memory_pool_p	pointer to the memory pool to add

4.21.2.2 instance()

```
static_mem_pool_set & static_mem_pool_set::instance ( )  [static]
```

Creates the singleton instance of static_mem_pool_set.

Returns

reference to the instance of static_mem_pool_set

4.21.2.3 recycle()

```
void static_mem_pool_set::recycle ( )
```

Asks all static memory pools to recycle unused memory blocks back to the system. The caller should get the lock to prevent other operations to static_mem_pool_set during its execution.

The documentation for this class was generated from the following files:

- static_mem_pool.h
- static_mem_pool.cpp

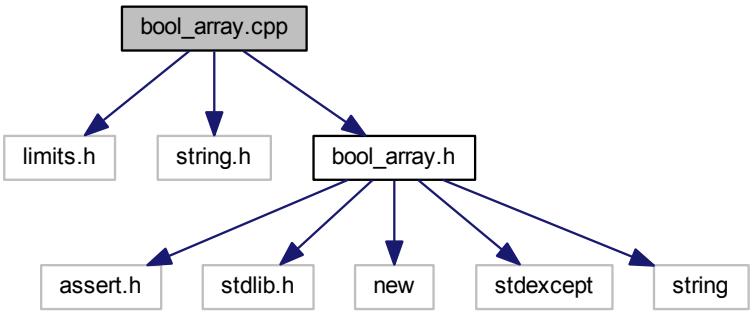
Chapter 5

File Documentation

5.1 bool_array.cpp File Reference

```
#include <limits.h>
#include <string.h>
#include "bool_array.h"
```

Include dependency graph for bool_array.cpp:



5.1.1 Detailed Description

Code for class bool_array (packed boolean array).

Version

3.1, 2005/08/25

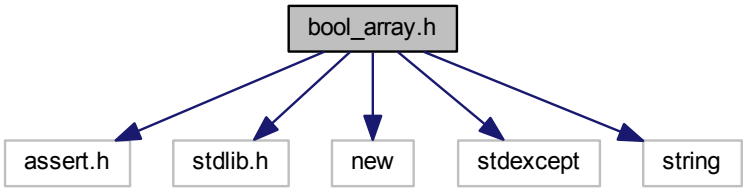
Author

Wu Yongwei

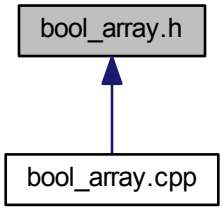
5.2 bool_array.h File Reference

```
#include <assert.h>
#include <stdlib.h>
#include <new>
#include <stdexcept>
#include <string>
```

Include dependency graph for bool_array.h:



This graph shows which files directly or indirectly include this file:



Classes

- class bool_array

Typedefs

- typedef unsigned char **BYTE**

5.2.1 Detailed Description

Header file for class bool_array (packed boolean array).

Version

3.1, 2005/08/25

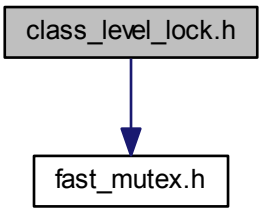
Author

Wu Yongwei

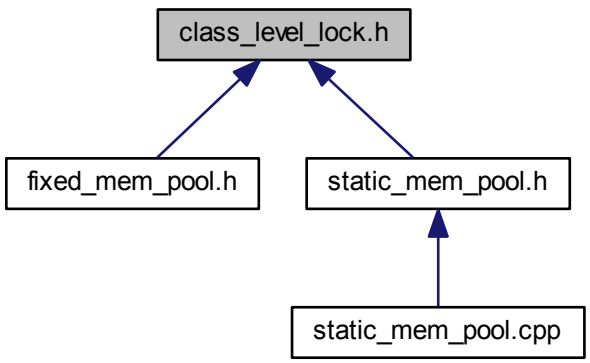
5.3 class_level_lock.h File Reference

```
#include "fast_mutex.h"
```

Include dependency graph for class_level_lock.h:



This graph shows which files directly or indirectly include this file:



Classes

- class class_level_lock< _Host, _RealLock >
- class class_level_lock< _Host, _RealLock >::lock

5.3.1 Detailed Description

In essence Loki ClassLevelLockable re-engineered to use a fast_mutex class.

Version

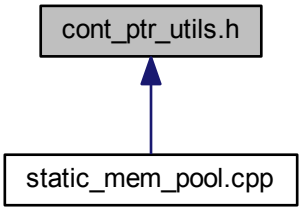
1.13, 2007/12/30

Author

Wu Yongwei

5.4 cont_ptr_utils.h File Reference

This graph shows which files directly or indirectly include this file:



Classes

- struct dereference
- struct dereference_less
- struct delete_object
- struct output_object< _OutputStrm, _StringType >

5.4.1 Detailed Description

Utility functors for containers of pointers (adapted from Scott Meyers' *Effective STL*).

Version

1.4, 2007/09/12

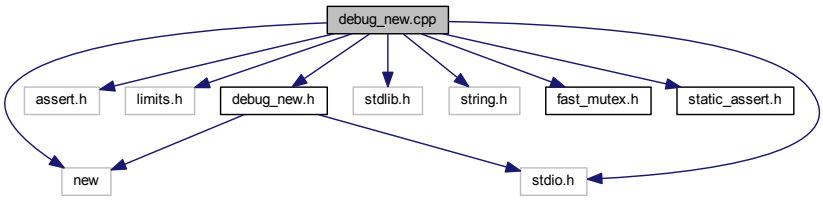
Author

Wu Yongwei

5.5 debug_new.cpp File Reference

```
#include <new>
#include <assert.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fast_mutex.h"
#include "static_assert.h"
#include "debug_new.h"
```

Include dependency graph for debug_new.cpp:



Classes

- struct new_ptr_list_t

Macros

- #define _DEBUG_NEW_ALIGNMENT 16
- #define _DEBUG_NEW_CALLER_ADDRESS NULL
- #define _DEBUG_NEW_ERROR_ACTION abort()
- #define _DEBUG_NEW_FILENAME_LEN 44
- #define _DEBUG_NEW_PROGNAME NULL
- #define _DEBUG_NEW_STD_OPER_NEW 1
- #define _DEBUG_NEW_TAILCHECK 0
- #define _DEBUG_NEW_TAILCHECK_CHAR 0xCC
- #define _DEBUG_NEW_USE_ADDR2LINE 0
- #define _DEBUG_NEW_REDEFINE_NEW 0
- #define align(s) (((s) + _DEBUG_NEW_ALIGNMENT - 1) & ~(_DEBUG_NEW_ALIGNMENT - 1))

Functions

- int check_leaks ()
- int check_mem_corruption ()
- void * **operator new** (size_t size, const char *file, int line)
- void * **operator new[ ]** (size_t size, const char *file, int line)
- void * **operator new** (size_t size) throw (std::bad_alloc)
- void * **operator new[ ]** (size_t size) throw (std::bad_alloc)
- void * **operator new** (size_t size, const std::nothrow_t &) throw ()
- void * **operator new[ ]** (size_t size, const std::nothrow_t &) throw ()

- void **operator delete** (void *pointer) throw ()
- void **operator delete[ ]** (void *pointer) throw ()
- void **operator delete** (void *pointer, const char *file, int line) throw ()
- void **operator delete[ ]** (void *pointer, const char *file, int line) throw ()
- void **operator delete** (void *pointer, const std::nothrow_t &) throw ()
- void **operator delete[ ]** (void *pointer, const std::nothrow_t &) throw ()

Variables

- const unsigned MAGIC = 0x4442474E
- const int ALIGNED_LIST_ITEM_SIZE = align(sizeof(new_ptr_list_t))
- bool new_autocheck_flag = true
- bool new_verbose_flag = false
- FILE * new_output_fp = stderr
- const char * new_progname = _DEBUG_NEW_PROGNAME

5.5.1 Detailed Description

Implementation of debug versions of new and delete to check leakage.

Version

4.12, 2007/12/31

Author

Wu Yongwei

5.5.2 Macro Definition Documentation

5.5.2.1 _DEBUG_NEW_ALIGNMENT

```
#define _DEBUG_NEW_ALIGNMENT 16
```

The alignment requirement of allocated memory blocks. It must be a power of two.

5.5.2.2 _DEBUG_NEW_CALLER_ADDRESS

```
#define _DEBUG_NEW_CALLER_ADDRESS NULL
```

The expression to return the caller address. #print_position will later on use this address to print the position information of memory operation points.

5.5.2.3 _DEBUG_NEW_ERROR_ACTION

```
#define _DEBUG_NEW_ERROR_ACTION abort()
```

The action to take when an error occurs. The default behaviour is to call *abort*, unless _DEBUG_NEW_ERROR↩ _CRASH is defined, in which case a segmentation fault will be triggered instead (which can be useful on platforms like Windows that do not generate a core dump when *abort* is called).

5.5.2.4 _DEBUG_NEW_FILENAME_LEN

```
#define _DEBUG_NEW_FILENAME_LEN 44
```

The length of file name stored if greater than zero. If it is zero, only a const char pointer will be stored. Currently the default behaviour is to copy the file name, because I found that the exit leakage check cannot access the address of the file name sometimes (in my case, a core dump will occur when trying to access the file name in a shared library after a SIGINT). The current default value makes the size of new_ptr_list_t 64 on 32-bit platforms.

5.5.2.5 _DEBUG_NEW_PROGNAME

```
#define _DEBUG_NEW_PROGNAME NULL
```

The program (executable) name to be set at compile time. It is better to assign the full program path to new_progname in *main* (at run time) than to use this (compile-time) macro, but this macro serves well as a quick hack. Note also that double quotation marks need to be used around the program name, i.e., one should specify a command-line option like -D_DEBUG_NEW_PROGNAME="a.out" in *bash*, or -D_DEBUG_NEW_PROGNA↩ ME="a.exe" in the Windows command prompt.

5.5.2.6 _DEBUG_NEW_REDEFINE_NEW

```
#define _DEBUG_NEW_REDEFINE_NEW 0
```

Macro to indicate whether redefinition of new is wanted. Here it is defined to 0 to disable the redefinition of new.

5.5.2.7 _DEBUG_NEW_STD_OPER_NEW

```
#define _DEBUG_NEW_STD_OPER_NEW 1
```

Macro to indicate whether the standard-conformant behaviour of operator new is wanted. It is on by default now, but the user may set it to 0 to revert to the old behaviour.

5.5.2.8 _DEBUG_NEW_TAILCHECK

```
#define _DEBUG_NEW_TAILCHECK 0
```

Macro to indicate whether a writing-past-end check will be performed. Define it to a positive integer as the number of padding bytes at the end of a memory block for checking.

5.5.2.9 _DEBUG_NEW_TAILCHECK_CHAR

```
#define _DEBUG_NEW_TAILCHECK_CHAR 0xCC
```

Value of the padding bytes at the end of a memory block.

5.5.2.10 _DEBUG_NEW_USE_ADDR2LINE

```
#define _DEBUG_NEW_USE_ADDR2LINE 0
```

Whether to use *addr2line* to convert a caller address to file/line information. Defining it to a non-zero value will enable the conversion (automatically done if GCC is detected). Defining it to zero will disable the conversion.

5.5.2.11 align

```
#define align(
              s ) (((s) + _DEBUG_NEW_ALIGNMENT - 1) & ~(_DEBUG_NEW_ALIGNMENT - 1))
```

Gets the aligned value of memory block size.

5.5.3 Function Documentation

5.5.3.1 check_leaks()

```
int check_leaks ( )
```

Checks for memory leaks.

Returns

zero if no leakage is found; the number of leaks otherwise

5.5.3.2 check_mem_corruption()

```
int check_mem_corruption ( )
```

Checks for heap corruption.

Returns

zero if no problem is found; the number of found memory corruptions otherwise

5.5.4 Variable Documentation

5.5.4.1 ALIGNED_LIST_ITEM_SIZE

```
const int ALIGNED_LIST_ITEM_SIZE = align(sizeof(new_ptr_list_t))
```

The extra memory allocated by `operator new`.

5.5.4.2 MAGIC

```
const unsigned MAGIC = 0x4442474E
```

Magic number for error detection.

5.5.4.3 new_autocheck_flag

```
bool new_autocheck_flag = true
```

Flag to control whether check_leaks will be automatically called on program exit.

5.5.4.4 new_output_fp

```
FILE* new_output_fp = stderr
```

Pointer to the output stream. The default output is *stderr*, and one may change it to a user stream if needed (say, new_verbose_flag is `true` and there are a lot of (de)allocations).

5.5.4.5 new_progname

```
const char* new_progname = _DEBUG_NEW_PROGNAME
```

Pointer to the program name. Its initial value is the macro _DEBUG_NEW_PROGNAME. You should try to assign the program path to it early in your application. Assigning `argv[0]` to it in *main* is one way. If you use *bash* or *ksh* (or similar), the following statement is probably what you want: '`new_progname = getenv("_");`'.

5.5.4.6 new_verbose_flag

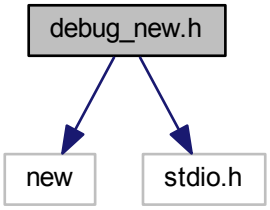
```
bool new_verbose_flag = false
```

Flag to control whether verbose messages are output.

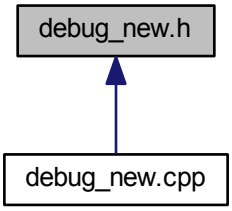
5.6 debug_new.h File Reference

```
#include <new>
#include <stdio.h>
```

Include dependency graph for debug_new.h:



This graph shows which files directly or indirectly include this file:



Classes

- class __debug_new_recorder
- class __debug_new_counter

Macros

- #define HAVE_PLACEMENT_DELETE 1
- #define _DEBUG_NEW_REDEFINE_NEW 1
- #define DEBUG_NEW __debug_new_recorder(__FILE__, __LINE__) ->* new
- #define **new** DEBUG_NEW

Functions

- int check_leaks ()
- int check_mem_corruption ()
- void * **operator new** (size_t size, const char *file, int line)
- void * **operator new[ ]** (size_t size, const char *file, int line)
- void **operator delete** (void *pointer, const char *file, int line) throw ()
- void **operator delete[ ]** (void *pointer, const char *file, int line) throw ()

Variables

- bool new_autocheck_flag
- bool new_verbose_flag
- FILE * new_output_fp
- const char * new_progname

5.6.1 Detailed Description

Header file for checking leaks caused by unmatched new/delete.

Version

4.4, 2007/12/31

Author

Wu Yongwei

5.6.2 Macro Definition Documentation

5.6.2.1 _DEBUG_NEW_REDEFINE_NEW

```
#define _DEBUG_NEW_REDEFINE_NEW 1
```

Macro to indicate whether redefinition of `new` is wanted. If one wants to define one's own `operator new`, to call `operator new` directly, or to call placement `new`, it should be defined to `0` to alter the default behaviour. Unless, of course, one is willing to take the trouble to write something like:

```
# ifdef new
#   define _NEW_REDEFINED
#   undef new
# endif

// Code that uses new is here

# ifdef _NEW_REDEFINED
#   ifdef DEBUG_NEW
#     define new DEBUG_NEW
#   endif
#   undef _NEW_REDEFINED
# endif
```

5.6.2.2 DEBUG_NEW

```
#define DEBUG_NEW __debug_new_recorder(__FILE__, __LINE__) ->* new
```

Macro to catch file/line information on allocation. If _DEBUG_NEW_REDEFINE_NEW is `0`, one can use this macro directly; otherwise `new` will be defined to it, and one must use `new` instead.

5.6.2.3 HAVE_PLACEMENT_DELETE

```
#define HAVE_PLACEMENT_DELETE 1
```

Macro to indicate whether placement delete operators are supported on a certain compiler. Some compilers, like Borland C++ Compiler 5.5.1 and Digital Mars Compiler 8.42, do not support them, and the user must define this macro to `0` to make the program compile. Also note that in that case memory leakage will occur if an exception is thrown in the initialization (constructor) of a dynamically created object.

5.6.3 Function Documentation

5.6.3.1 check_leaks()

```
int check_leaks ( )
```

Checks for memory leaks.

Returns

zero if no leakage is found; the number of leaks otherwise

5.6.3.2 check_mem_corruption()

```
int check_mem_corruption ( )
```

Checks for heap corruption.

Returns

zero if no problem is found; the number of found memory corruptions otherwise

5.6.4 Variable Documentation

5.6.4.1 new_autocheck_flag

```
bool new_autocheck_flag
```

Flag to control whether check_leaks will be automatically called on program exit.

5.6.4.2 new_output_fp

```
FILE* new_output_fp
```

Pointer to the output stream. The default output is *stderr*, and one may change it to a user stream if needed (say, new_verbose_flag is `true` and there are a lot of (de)allocations).

5.6.4.3 new_progname

```
const char* new_progname
```

Pointer to the program name. Its initial value is the macro _DEBUG_NEW_PROGNAME. You should try to assign the program path to it early in your application. Assigning `argv[0]` to it in *main* is one way. If you use *bash* or *ksh* (or similar), the following statement is probably what you want: '`new_progname = getenv("_");`'.

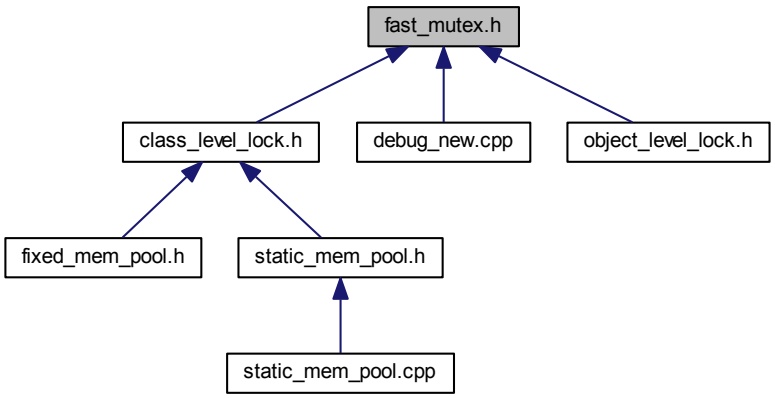
5.6.4.4 new_verbose_flag

```
bool new_verbose_flag
```

Flag to control whether verbose messages are output.

5.7 fast_mutex.h File Reference

This graph shows which files directly or indirectly include this file:



Classes

- class fast_mutex
- class fast_mutex_autolock

Macros

- #define **_NOTHREADS**
- #define _FAST_MUTEX_CHECK_INITIALIZATION 1
- #define _FAST_MUTEX_ASSERT(_Expr, _Msg) ((void)0)
- #define __VOLATILE

5.7.1 Detailed Description

A fast mutex implementation for POSIX and Win32.

Version

1.18, 2005/05/06

Author

Wu Yongwei

5.7.2 Macro Definition Documentation

5.7.2.1 __VOLATILE

```
#define __VOLATILE
```

Macro alias to 'volatile' semantics. Here it is not truly volatile since it is in a single-threaded environment.

5.7.2.2 _FAST_MUTEX_ASSERT

```
#define _FAST_MUTEX_ASSERT(
              _Expr,
              _Msg ) ((void)0)
```

Macro for fast_mutex assertions. Fake version (for release mode).

5.7.2.3 _FAST_MUTEX_CHECK_INITIALIZATION

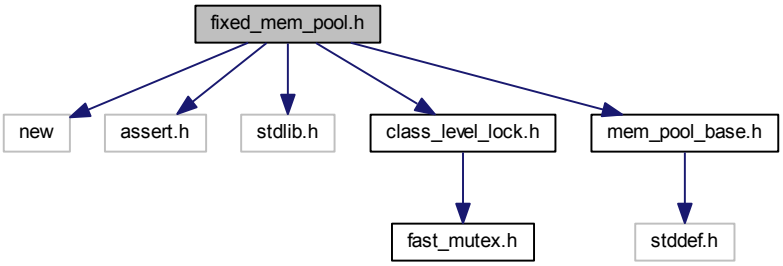
```
#define _FAST_MUTEX_CHECK_INITIALIZATION 1
```

Macro to control whether to check for initialization status for each lock/unlock operation. Defining it to a non-zero value will enable the check, so that the construction/destruction of a static object using a static fast_mutex not yet constructed or already destroyed will work (with lock/unlock operations ignored). Defining it to zero will disable to check.

5.8 fixed_mem_pool.h File Reference

```
#include <new>
#include <assert.h>
#include <stdlib.h>
#include "class_level_lock.h"
#include "mem_pool_base.h"
```

Include dependency graph for fixed_mem_pool.h:



Classes

- class fixed_mem_pool< _Tp >

Macros

- #define MEM_POOL_ALIGNMENT 4
- #define DECLARE_FIXED_MEM_POOL(_Cls)
- #define DECLARE_FIXED_MEM_POOL__NOTHROW(_Cls)
- #define DECLARE_FIXED_MEM_POOL__THROW_NOCHECK(_Cls)

5.8.1 Detailed Description

Definition of a fixed-size memory pool template for structs/classes. This is a easy-to-use class template for pre-allocated memory pools. The client side needs to do the following things:

- Use one of the macros DECLARE_FIXED_MEM_POOL, DECLARE_FIXED_MEM_POOL__NOTHROW, and DECLARE_FIXED_MEM_POOL__THROW_NOCHECK at the end of the class (say, `class` *_Cls*) definitions
- Call fixed_mem_pool<_Cls>::initialize at the beginning of the program
- Optionally, specialize fixed_mem_pool<_Cls>::bad_alloc_handler to change the behaviour when all memory blocks are allocated
- Optionally, call fixed_mem_pool<_Cls>::deinitialize at exit of the program to check for memory leaks
- Optionally, call fixed_mem_pool<_Cls>::get_alloc_count to check memory usage when the program is running

Version

1.14, 2005/09/19

Author

Wu Yongwei

5.8.2 Macro Definition Documentation

5.8.2.1 DECLARE_FIXED_MEM_POOL

```
#define DECLARE_FIXED_MEM_POOL(
              _Cls )
```

Value:

```
public: \
    static void* operator new(size_t __size) \
    { \
        assert(__size == sizeof(_Cls)); \
        if (void* __ptr = fixed_mem_pool<_Cls>::allocate()) \
            return __ptr; \
        else \
            throw std::bad_alloc(); \
    } \
    static void  operator delete(void* __ptr) \
    { \
        if (__ptr != NULL) \
            fixed_mem_pool<_Cls>::deallocate(__ptr); \
    }
```

Declares the normal (exceptionable) overload of **operator new** and **operator delete**.

Parameters

_Cls	class to use the fixed_mem_pool

See also

DECLARE_FIXED_MEM_POOL__THROW_NOCHECK, which, too, defines an **operator new** that will never return NULL, but requires more discipline on the programmer's side.

5.8.2.2 DECLARE_FIXED_MEM_POOL__NOTHROW

```
#define DECLARE_FIXED_MEM_POOL__NOTHROW(
              _Cls )
```

Value:

```
public: \
    static void* operator new(size_t __size) throw() \
    { \
        assert(__size == sizeof(_Cls)); \
        return fixed_mem_pool<_Cls>::allocate(); \
    } \
    static void  operator delete(void* __ptr) \
    { \
        if (__ptr != NULL) \
            fixed_mem_pool<_Cls>::deallocate(__ptr); \
    }
```

Declares the non-exceptionable overload of **operator new** and **operator delete**.

Parameters

_Cls	class to use the fixed_mem_pool

5.8.2.3 DECLARE_FIXED_MEM_POOL__THROW_NOCHECK

```
#define DECLARE_FIXED_MEM_POOL__THROW_NOCHECK(
              _Cls )
```

Value:

```
public: \
    static void* operator new(size_t __size) \
    { \
        assert(__size == sizeof(_Cls)); \
        return fixed_mem_pool<_Cls>::allocate(); \
    } \
    static void  operator delete(void* __ptr) \
    { \
        if (__ptr != NULL) \
            fixed_mem_pool<_Cls>::deallocate(__ptr); \
    }
```

Declares the exceptionable, non-checking overload of **operator new** and **operator delete**.

N.B. Using this macro *requires* users to explicitly specialize fixed_mem_pool::bad_alloc_handler so that it shall never return `false` (it may throw exceptions, say, `std::bad_alloc`, or simply abort). Otherwise a segmentation fault might occur (instead of returning a `NULL` pointer).

Parameters

_Cls	class to use the fixed_mem_pool

5.8.2.4 MEM_POOL_ALIGNMENT

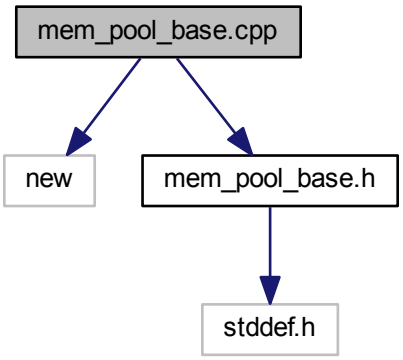
```
#define MEM_POOL_ALIGNMENT 4
```

Defines the alignment of memory blocks.

5.9 mem_pool_base.cpp File Reference

```
#include <new>
#include "mem_pool_base.h"
```

Include dependency graph for mem_pool_base.cpp:



Macros

- #define **_MEM_POOL_ALLOCATE**(_Sz) ::operator new((_Sz), std::nothrow)
- #define **_MEM_POOL_DEALLOCATE**(_Ptr) ::operator delete(_Ptr)

5.9.1 Detailed Description

Implementation for the memory pool base.

Version

1.2, 2004/07/26

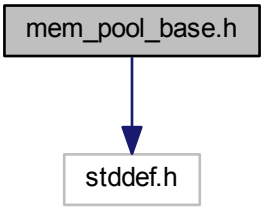
Author

Wu Yongwei

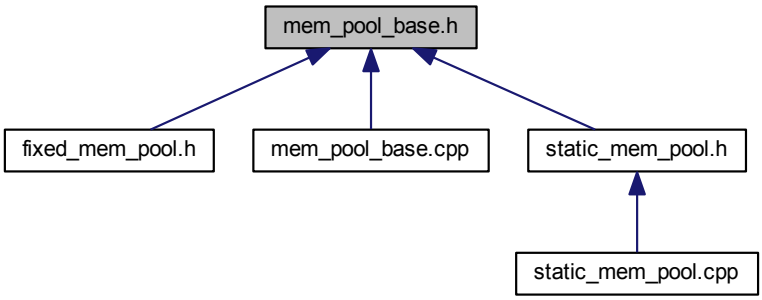
5.10 mem_pool_base.h File Reference

```
#include <stddef.h>
```

Include dependency graph for mem_pool_base.h:



This graph shows which files directly or indirectly include this file:



Classes

- class mem_pool_base
- struct mem_pool_base::_Block_list

5.10.1 Detailed Description

Header file for the memory pool base.

Version

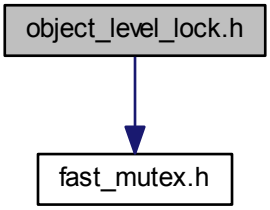
1.1, 2004/07/26

Author

Wu Yongwei

5.11 object_level_lock.h File Reference

`#include "fast_mutex.h"`
Include dependency graph for object_level_lock.h:



Classes

- class object_level_lock< _Host >
- class object_level_lock< _Host >::lock

5.11.1 Detailed Description

In essence Loki ObjectLevelLockable re-engineered to use a fast_mutex class. Check also Andrei Alexandrescu's article "Multithreading and the C++ Type System" for the ideas behind.

Version

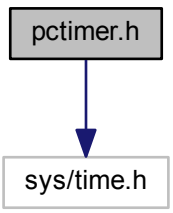
1.4, 2004/05/09

Author

Wu Yongwei

5.12 pctimer.h File Reference

`#include <sys/time.h>`
Include dependency graph for pctimer.h:



Typedefs

- typedef double **pctimer_t**

Functions

- __inline pctimer_t **pctimer** (void)

5.12.1 Detailed Description

Function to get a high-resolution timer for Win32/Cygwin/Unix.

Version

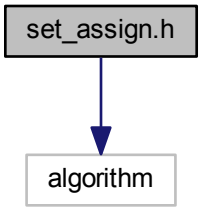
1.6, 2004/08/02

Author

Wu Yongwei

5.13 set_assign.h File Reference

`#include <algorithm>`
Include dependency graph for set_assign.h:



Functions

- template<class _Container , class _InputIter >
  _Container & **set_assign_union** (_Container &__dest, _InputIter __first, _InputIter __last)
- template<class _Container , class _InputIter , class _Compare >
  _Container & **set_assign_union** (_Container &__dest, _InputIter __first, _InputIter __last, _Compare __comp)
- template<class _Container , class _InputIter >
  _Container & **set_assign_difference** (_Container &__dest, _InputIter __first, _InputIter __last)
- template<class _Container , class _InputIter , class _Compare >
  _Container & **set_assign_difference** (_Container &__dest, _InputIter __first, _InputIter __last, _Compare __comp)

5.13.1 Detailed Description

Definition of template functions set_assign_union and set_assign_difference.

Version

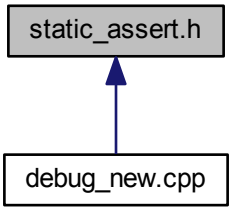
1.5, 2004/07/26

Author

Wu Yongwei

5.14 static_assert.h File Reference

This graph shows which files directly or indirectly include this file:



Classes

- struct __nvwa_compile_time_error< bool >
- struct __nvwa_compile_time_error< true >

Macros

- #define **STATIC_ASSERT**(_Expr, _Msg)

5.14.1 Detailed Description

Template class to check validity duing compile time (adapted from Loki).

Version

1.2, 2005/11/22

Author

Wu Yongwei

5.14.2 Macro Definition Documentation

5.14.2.1 STATIC_ASSERT

```
#define STATIC_ASSERT(
              _Expr,
              _Msg )
```

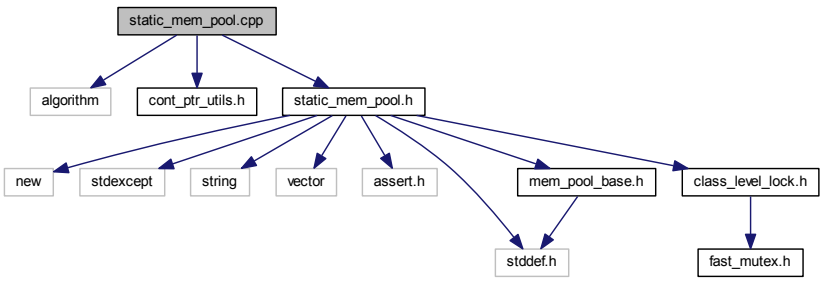
Value:

```
{ \
        __nvwa_compile_time_error<((_Expr) != 0)> ERROR_##_Msg; \
        (void)ERROR_##_Msg; \
    }
```

5.15 static_mem_pool.cpp File Reference

```
#include <algorithm>
#include "cont_ptr_utils.h"
#include "static_mem_pool.h"
```

Include dependency graph for static_mem_pool.cpp:



5.15.1 Detailed Description

Non-template and non-inline code for the 'static' memory pool.

Version

1.7, 2006/08/26

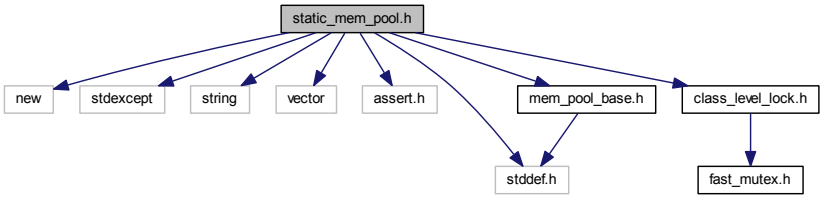
Author

Wu Yongwei

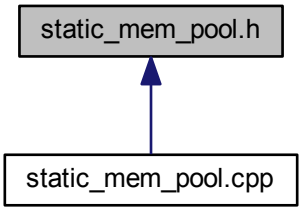
5.16 static_mem_pool.h File Reference

```
#include <new>
#include <stdexcept>
#include <string>
#include <vector>
#include <assert.h>
#include <stddef.h>
#include "class_level_lock.h"
#include "mem_pool_base.h"
```

Include dependency graph for static_mem_pool.h:



This graph shows which files directly or indirectly include this file:



Classes

- class static_mem_pool_set
- class static_mem_pool< _Sz, _Gid >

Macros

- #define **__PRIVATE** private
- #define **_STATIC_MEM_POOL_TRACE**(_Lck, _Msg) ((void)0)
- #define **DECLARE_STATIC_MEM_POOL**(_Cls)
- #define **DECLARE_STATIC_MEM_POOL__NOTHROW**(_Cls)
- #define **DECLARE_STATIC_MEM_POOL_GROUPED**(_Cls, _Gid)
- #define **DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW**(_Cls, _Gid)
- #define **PREPARE_STATIC_MEM_POOL**(_Cls) std::cerr << "PREPARE_STATIC_MEM_POOL is obsolete!\n";
- #define **PREPARE_STATIC_MEM_POOL_GROUPED**(_Cls, _Gid) std::cerr << "PREPARE_STATIC_M↩EM_POOL_GROUPED is obsolete!\n";

5.16.1 Detailed Description

Header file for the 'static' memory pool.

Version

1.20, 2007/10/20

Author

Wu Yongwei

5.16.2 Macro Definition Documentation

5.16.2.1 DECLARE_STATIC_MEM_POOL

```
#define DECLARE_STATIC_MEM_POOL(
              _Cls )
```

Value:

```
public: \
    static void* operator new(size_t __size) \
    { \
        assert(__size == sizeof(_Cls)); \
        void* __ptr; \
        __ptr = static_mem_pool<sizeof(_Cls)>:: \
                                instance_known().allocate(); \
        if (__ptr == NULL) \
            throw std::bad_alloc(); \
        return __ptr; \
    } \
    static void operator delete(void* __ptr) \
    { \
        if (__ptr) \
            static_mem_pool<sizeof(_Cls)>:: \
                        instance_known().deallocate(__ptr); \
    }
```

5.16.2.2 DECLARE_STATIC_MEM_POOL__NOTHROW

```
#define DECLARE_STATIC_MEM_POOL__NOTHROW(
              _Cls )
```

Value:

```
public: \
    static void* operator new(size_t __size) throw() \
    { \
        assert(__size == sizeof(_Cls)); \
        return static_mem_pool<sizeof(_Cls)>:: \
                            instance_known().allocate(); \
    } \
    static void operator delete(void* __ptr) \
    { \
        if (__ptr) \
            static_mem_pool<sizeof(_Cls)>:: \
                        instance_known().deallocate(__ptr); \
    }
```

5.16.2.3 DECLARE_STATIC_MEM_POOL_GROUPED

```
#define DECLARE_STATIC_MEM_POOL_GROUPED(
              _Cls,
              _Gid )
```

Value:

```
public: \
    static void* operator new(size_t __size) \
    { \
        assert(__size == sizeof(_Cls)); \
        void* __ptr; \
        __ptr = static_mem_pool<sizeof(_Cls), (_Gid)>:: \
                                instance_known().allocate(); \
        if (__ptr == NULL) \
            throw std::bad_alloc(); \
        return __ptr; \
    } \
    static void operator delete(void* __ptr) \
    { \
        if (__ptr) \
            static_mem_pool<sizeof(_Cls), (_Gid)>:: \
                            instance_known().deallocate(__ptr); \
    }
```

5.16.2.4 DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW

```
#define DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW(
              _Cls,
              _Gid )
```

Value:

```
public: \
    static void* operator new(size_t __size) throw() \
    { \
        assert(__size == sizeof(_Cls)); \
        return static_mem_pool<sizeof(_Cls), (_Gid)>:: \
                               instance_known().allocate(); \
    } \
    static void operator delete(void* __ptr) \
    { \
        if (__ptr) \
            static_mem_pool<sizeof(_Cls), (_Gid)>:: \
                            instance_known().deallocate(__ptr); \
    }
```

Index