Nvwa Reference Manual

0.8.2

Generated by Doxygen 1.5.1

Tue Dec 30 14:53:05 2008

Contents

1 Nvwa Hierarchical Index **1**

1.1 Nvwa Class Hierarchy 1

2 Nvwa Class Index **3**

2.1 Nvwa Class List 3

3 Nvwa File Index **5**

3.1 Nvwa File List 5

4 Nvwa Class Documentation **7**

4.1 __debug_new_counter Class Reference 7

4.2 __debug_new_recorder Class Reference 9

4.3 __nvwa_compile_time_error< true > Struct Template Reference 11

4.4 bool_array Class Reference 12

4.5 class_level_lock< _Host, _RealLock > Class Template Reference 17

4.6 class_level_lock< _Host, _RealLock >::lock Class Reference 19

4.7 delete_object Struct Reference 20

4.8 dereference Struct Reference 21

4.9 dereference_less Struct Reference 22

4.10 fast_mutex Class Reference 23

4.11 fast_mutex_autolock Class Reference 25

4.12 fixed_mem_pool< _Tp > Class Template Reference 26

4.13 mem_pool_base Class Reference 29

4.14 mem_pool_base::_Block_list Struct Reference 31

4.15 new_ptr_list_t Struct Reference 32

4.16 object_level_lock< _Host > Class Template Reference 34

4.17 object_level_lock< _Host >::lock Class Reference 36

4.18 output_object< _OutputStrm, _StringType > Struct Template Reference 38

4.19 static_mem_pool< _Sz, _Gid > Class Template Reference 40

4.20 static_mem_pool_set Class Reference 43

5 Nvwa File Documentation **45**

5.1 bool_array.cpp File Reference 45
5.2 bool_array.h File Reference 46
5.3 class_level_lock.h File Reference 48
5.4 cont_ptr_utils.h File Reference 49
5.5 debug_new.cpp File Reference 50
5.6 debug_new.h File Reference 60
5.7 fast_mutex.h File Reference 65
5.8 fixed_mem_pool.h File Reference 67
5.9 mem_pool_base.cpp File Reference 70
5.10 mem_pool_base.h File Reference 71
5.11 object_level_lock.h File Reference 72
5.12 pctimer.h File Reference 73
5.13 set_assign.h File Reference 74
5.14 static_assert.h File Reference 76
5.15 static_mem_pool.cpp File Reference 77
5.16 static_mem_pool.h File Reference 78

Chapter 1

Nvwa Hierarchical Index

1.1 Nvwa Class Hierarchy

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

__debug_new_counter . . . 7
__debug_new_recorder . . . 9
__nvwa_compile_time_error< true > . . . 11
bool_array . . . 12
class_level_lock< _Host, _RealLock > . . . 17
class_level_lock< _Host, _RealLock >::lock . . . 19
delete_object . . . 20
dereference . . . 21
dereference_less . . . 22
fast_mutex . . . 23
fast_mutex_autolock . . . 25
fixed_mem_pool< _Tp > . . . 26
mem_pool_base . . . 29
    static_mem_pool< _Sz, _Gid > . . . 40
mem_pool_base::_Block_list . . . 31
new_ptr_list_t . . . 32
object_level_lock< _Host > . . . 34
object_level_lock< _Host >::lock . . . 36
output_object< _OutputStrm, _StringType > . . . 38
static_mem_pool_set . . . 43

Chapter 2

Nvwa Class Index

2.1 Nvwa Class List

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

__debug_new_counter (Counter class for on-exit leakage check) 7
__debug_new_recorder (Recorder class to remember the call context) 9
__nvwa_compile_time_error< true > 11
bool_array (Class to represent a packed boolean array) 12
class_level_lock< _Host, _RealLock > (Helper class for class-level locking) 17
class_level_lock< _Host, _RealLock >::lock (Type that provides locking/unlocking semantics) 19
delete_object (Functor to delete objects pointed by a container of pointers) 20
dereference (Functor to return objects pointed by a container of pointers) 21
dereference_less (Functor to compare objects pointed by a container of pointers) . . . 22
fast_mutex (Class for non-reentrant fast mutexes) 23
fast_mutex_autolock (An acquistion-on-initialization lock class based on fast_mutex) 25
fixed_mem_pool< _Tp > (Class template to manipulate a fixed-size memory pool) . 26
mem_pool_base (Base class for memory pools) 29
mem_pool_base::_Block_list (Structure to store the next available memory block) . . 31
new_ptr_list_t (Structure to store the position information where `new` occurs) 32
object_level_lock< _Host > (Helper class for class-level locking) 34
object_level_lock< _Host >::lock (Type that provides locking/unlocking semantics) . 36
output_object< _OutputStrm, _StringType > (Functor to output objects pointed by a container of pointers) 38
static_mem_pool< _Sz, _Gid > (Singleton class template to manage the allocation/deallocation of memory blocks of one specific size) 40
static_mem_pool_set (Singleton class to maintain a set of existing instantiations of static_mem_pool) 43

Chapter 3

Nvwa File Index

3.1 Nvwa File List

Here is a list of all files with brief descriptions:

bool_array.cpp (Code for class bool_array (packed boolean array)) 45
bool_array.h (Header file for class bool_array (packed boolean array)) 46
class_level_lock.h (In essence Loki ClassLevelLockable re-engineered to use a fast_mutex class) 48
cont_ptr_utils.h (Utility functors for containers of pointers (adapted from Scott Meyers' *Effective STL*)) 49
debug_new.cpp (Implementation of debug versions of new and delete to check leakage) 50
debug_new.h (Header file for checking leaks caused by unmatched new/delete) 60
fast_mutex.h (A fast mutex implementation for POSIX and Win32) 65
fixed_mem_pool.h (Definition of a fixed-size memory pool template for structs/classes) 67
mem_pool_base.cpp (Implementation for the memory pool base) 70
mem_pool_base.h (Header file for the memory pool base) 71
object_level_lock.h (In essence Loki ObjectLevelLockable re-engineered to use a fast_mutex class) 72
pctimer.h (Function to get a high-resolution timer for Win32/Cygwin/Unix) 73
set_assign.h (Definition of template functions set_assign_union and set_assign_difference) 74
static_assert.h (Template class to check validity duing compile time (adapted from Loki)) 76
static_mem_pool.cpp (Non-template and non-inline code for the 'static' memory pool) 77
static_mem_pool.h (Header file for the 'static' memory pool) 78

Chapter 4

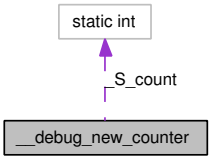
Nvwa Class Documentation

4.1 __debug_new_counter Class Reference

Counter class for on-exit leakage check.

`#include <debug_new.h>`

Collaboration diagram for __debug_new_counter:



Public Member Functions

- __debug_new_counter ()
  Constructor to increment the count.
- ~__debug_new_counter ()
  Destructor to decrement the count.

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 ()

Constructor to increment the count.

4.1.2.2 __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:

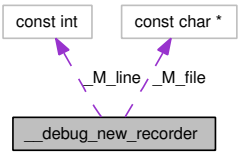
- debug_new.h
- debug_new.cpp

4.2 __debug_new_recorder Class Reference

Recorder class to remember the call context.

`#include <debug_new.h>`

Collaboration diagram for __debug_new_recorder:



Public Member Functions

- __debug_new_recorder (const char *file, int line)

  Constructor to remember the call context.

- template<class _Tp> _Tp * operator->* (_Tp *pointer)

  Operator to write the context information to memory.

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 (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 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< true > Struct Template Reference

`#include <static_assert.h>`

template<> struct __nvwa_compile_time_error< true >

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

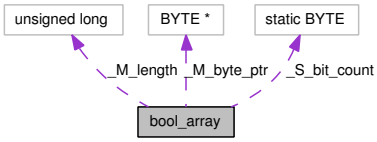
- static_assert.h

4.4 bool_array Class Reference

Class to represent a packed boolean array.

`#include <bool_array.h>`

Collaboration diagram for bool_array:



Public Member Functions

- bool_array ()
- bool_array (unsigned long __size)

  Constructs the packed boolean array with a specific size.

- ~bool_array ()
- bool create (unsigned long __size)

  Creates the packed boolean array with a specific size.

- void initialize (bool __value)

  Initializes all array elements to a specific value optimally.

- _Element operator[] (unsigned long __idx)

  Creates a reference to an array element.

- bool at (unsigned long __idx) const

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

- void reset (unsigned long __idx)

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

- void set (unsigned long __idx)

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

- unsigned long size () const
- unsigned long count () const

  Counts elements with a `true` *value.*

- unsigned long count (unsigned long __beg, unsigned long __end) const

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

- void flip ()

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

Classes

- class **_Element**

  Class to represent a reference to an array element.

4.4.1 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.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 bool_array::bool_array () [inline]

4.4.2.2 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

4.4.2.3 bool_array::~bool_array () [inline]

4.4.3 Member Function Documentation

4.4.3.1 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.

4.4.3.2 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

4.4.3.3 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

4.4.3.4 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

4.4.3.5 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

4.4.3.6 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

4.4.3.7 unsigned long bool_array::size () const [inline]

4.4.3.8 unsigned long bool_array::count () const

Counts elements with a `true` value.

Returns:

the count of `true` elements

4.4.3.9 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

4.4.3.10 void bool_array::flip ()

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

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

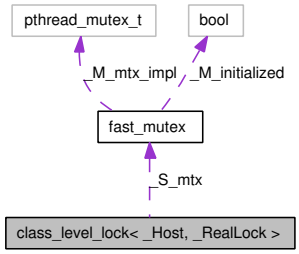
- bool_array.h
- bool_array.cpp

4.5 class_level_lock< _Host, _RealLock > Class Template Reference

Helper class for class-level locking.

`#include <class_level_lock.h>`

Collaboration diagram for class_level_lock< _Host, _RealLock >:



Public Types

- typedef volatile _Host volatile_type

Friends

- class lock

Classes

- class lock

  Type that provides locking/unlocking semantics.

4.5.1 Detailed Description

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

Helper class for class-level locking.

This is the multi-threaded implementation. The main departure from Loki ClassLevelLockable is that there is an additional template parameter which can make the lock not lock at all even in multi-threaded environments. See static_mem_pool.h for real usage.

4.5.2 Member Typedef Documentation

4.5.2.1 template<class _Host, bool _RealLock = true> typedef volatile _Host class_level_lock< _Host, _RealLock >::volatile_type

4.5.3 Friends And Related Function Documentation

4.5.3.1 template<class _Host, bool _RealLock = true> friend class lock [friend]

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

- class_level_lock.h

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

Type that provides locking/unlocking semantics.

`#include <class_level_lock.h>`

Public Member Functions

- lock ()
- ~lock ()

4.6.1 Detailed Description

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

Type that provides locking/unlocking semantics.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 template<class _Host, bool _RealLock = true> class_level_lock< _Host, _RealLock >::lock::lock () [inline]

4.6.2.2 template<class _Host, bool _RealLock = true> class_level_lock< _Host, _RealLock >::lock::~lock () [inline]

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

- class_level_lock.h

4.7 delete_object Struct Reference

Functor to delete objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

Public Member Functions

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

4.7.1 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());
```

4.7.2 Member Function Documentation

4.7.2.1 template<typename _Pointer> void delete_object::operator() (_Pointer __ptr) const [inline]

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

- cont_ptr_utils.h

4.8 dereference Struct Reference

Functor to return objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

Public Member Functions

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

4.8.1 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());
```

4.8.2 Member Function Documentation

4.8.2.1 template<typename _Tp> const _Tp& dereference::operator() (const _Tp * __ptr) const [inline]

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

- cont_ptr_utils.h

4.9 dereference_less Struct Reference

Functor to compare objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

Public Member Functions

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

4.9.1 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;
```

4.9.2 Member Function Documentation

4.9.2.1 template<typename _Pointer> bool dereference_less::operator() (_Pointer __*ptr1*, _Pointer __*ptr2*) const [inline]

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

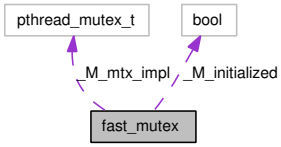
- cont_ptr_utils.h

4.10 fast_mutex Class Reference

Class for non-reentrant fast mutexes.

`#include <fast_mutex.h>`

Collaboration diagram for fast_mutex:



Public Member Functions

- fast_mutex ()
- ~fast_mutex ()
- void lock ()
- void unlock ()

4.10.1 Detailed Description

Class for non-reentrant fast mutexes.

This is the implementation for POSIX threads.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 fast_mutex::fast_mutex () [inline]

4.10.2.2 fast_mutex::~fast_mutex () [inline]

4.10.3 Member Function Documentation

4.10.3.1 void fast_mutex::lock () [inline]

4.10.3.2 void fast_mutex::unlock () [inline]

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

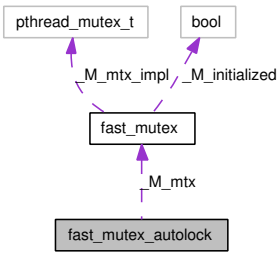
- fast_mutex.h

4.11 fast_mutex_autolock Class Reference

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

`#include <fast_mutex.h>`

Collaboration diagram for fast_mutex_autolock:



Public Member Functions

- fast_mutex_autolock (fast_mutex &__mtx)
- ~fast_mutex_autolock ()

4.11.1 Detailed Description

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

4.11.2 Constructor & Destructor Documentation

4.11.2.1 fast_mutex_autolock::fast_mutex_autolock (fast_mutex & __mtx) [inline, explicit]

4.11.2.2 fast_mutex_autolock::~fast_mutex_autolock () [inline]

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

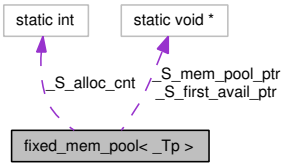
- fast_mutex.h

4.12 fixed_mem_pool< _Tp > Class Template Reference

Class template to manipulate a fixed-size memory pool.

`#include <fixed_mem_pool.h>`

Collaboration diagram for fixed_mem_pool< _Tp >:



Public Types

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

Static Public Member Functions

- static void * allocate ()

  Allocates a memory block from the memory pool.

- static void deallocate (void *)

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

- static bool initialize (size_t __size)

  Initializes the memory pool.

- static int deinitialize ()

  Deinitializes the memory pool.

- static int get_alloc_count ()

  Gets the allocation count.

- static bool is_initialized ()

  Is the memory pool initialized?

Static Protected Member Functions

- static bool bad_alloc_handler ()

  Bad allocation handler.

4.12.1 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

4.12.2 Member Typedef Documentation

4.12.2.1 template<class _Tp> typedef class_level_lock<fixed_mem_pool<_Tp> >::lock fixed_mem_pool< _Tp >::lock

4.12.3 Member Function Documentation

4.12.3.1 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

4.12.3.2 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

4.12.3.3 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

4.12.3.4 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

4.12.3.5 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

4.12.3.6 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

4.12.3.7 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.

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

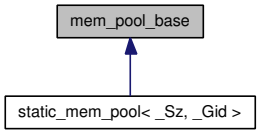
- fixed_mem_pool.h

4.13 mem_pool_base Class Reference

Base class for memory pools.

`#include <mem_pool_base.h>`

Inheritance diagram for mem_pool_base:



Public Member Functions

- virtual ~mem_pool_base ()
- virtual void recycle ()=0

Static Public Member Functions

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

Classes

- struct _Block_list

  Structure to store the next available memory block.

4.13.1 Detailed Description

Base class for memory pools.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 mem_pool_base::~mem_pool_base () [virtual]

4.13.3 Member Function Documentation

4.13.3.1 virtual void mem_pool_base::recycle () [pure virtual]

Implemented in static_mem_pool< _Sz, _Gid >.

4.13.3.2 void * mem_pool_base::alloc_sys (size_t __size) [static]

4.13.3.3 void mem_pool_base::dealloc_sys (void * __ptr) [static]

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

- mem_pool_base.h
- mem_pool_base.cpp

4.14 mem_pool_base::_Block_list Struct Reference

Structure to store the next available memory block.

`#include <mem_pool_base.h>`

Collaboration diagram for mem_pool_base::_Block_list:

Public Attributes

- _Block_list * _M_next

4.14.1 Detailed Description

Structure to store the next available memory block.

4.14.2 Member Data Documentation

4.14.2.1 _Block_list* mem_pool_base::_Block_list::_M_next

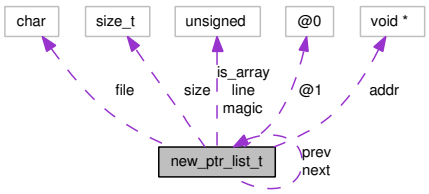
The documentation for this struct was generated from the following file:

- mem_pool_base.h

4.15 new_ptr_list_t Struct Reference

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

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

4.15.1 Detailed Description

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

4.15.2 Member Data Documentation

4.15.2.1 new_ptr_list_t* new_ptr_list_t::next

4.15.2.2 new_ptr_list_t* new_ptr_list_t::prev

4.15.2.3 size_t new_ptr_list_t::size

4.15.2.4 char new_ptr_list_t::file[_DEBUG_NEW_FILENAME_LEN]

4.15.2.5 void* new_ptr_list_t::addr

4.15.2.6 union { ... }

4.15.2.7 unsigned new_ptr_list_t::line

4.15.2.8 unsigned new_ptr_list_t::is_array

4.15.2.9 unsigned new_ptr_list_t::magic

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

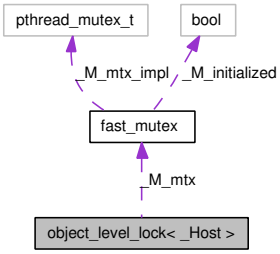
- debug_new.cpp

4.16 object_level_lock< _Host > Class Template Reference

Helper class for class-level locking.

`#include <object_level_lock.h>`

Collaboration diagram for object_level_lock< _Host >:



Public Types

- typedef volatile _Host volatile_type

Friends

- class lock

Classes

- class lock

  Type that provides locking/unlocking semantics.

4.16.1 Detailed Description

template<class _Host> class object_level_lock< _Host >

Helper class for class-level locking.

This is the multi-threaded implementation.

4.16.2 Member Typedef Documentation

4.16.2.1 template<class _Host> typedef volatile _Host object_level_lock< _Host >::volatile_type

4.16.3 Friends And Related Function Documentation

4.16.3.1 template<class _Host> friend class lock [friend]

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

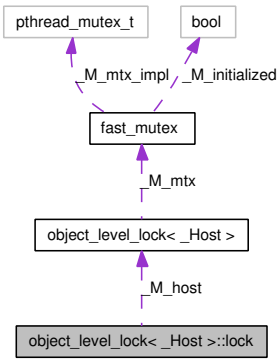
- object_level_lock.h

4.17 object_level_lock< _Host >::lock Class Reference

Type that provides locking/unlocking semantics.

`#include <object_level_lock.h>`

Collaboration diagram for object_level_lock< _Host >::lock:



Public Member Functions

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

4.17.1 Detailed Description

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

Type that provides locking/unlocking semantics.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 template<class _Host> object_level_lock< _Host >::lock::lock (const object_level_lock & __host) [inline, explicit]

4.17.2.2 template<class _Host> object_level_lock< _Host >::lock::~lock () [inline]

4.17.3 Member Function Documentation

4.17.3.1 template<class _Host> const object_level_lock* object_level_lock< _Host >::lock::get_locked_object () const [inline]

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

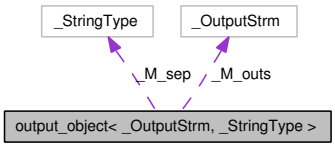
- object_level_lock.h

4.18 output_object< _OutputStrm, _StringType > Struct Template Reference

Functor to output objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

Collaboration diagram for output_object< _OutputStrm, _StringType >:



Public Member Functions

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

4.18.1 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, " "));
```

4.18.2 Constructor & Destructor Documentation

4.18.2.1 template<typename _OutputStrm, typename _StringType = const char*> output_object< _OutputStrm, _StringType >::output_object (_OutputStrm & __outs, const _StringType & __sep) [inline]

4.18.3 Member Function Documentation

4.18.3.1 template<typename _OutputStrm, typename _StringType = const char*> template<typename _Tp> void output_object< _OutputStrm, _StringType >::operator() (const _Tp * __ptr) const [inline]

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

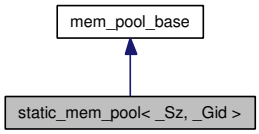
- cont_ptr_utils.h

4.19 static_mem_pool< _Sz, _Gid > Class Template Reference

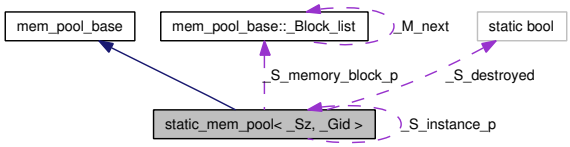
Singleton class template to manage the allocation/deallocation of memory blocks of one specific size.

`#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 ()

  Allocates memory and returns its pointer.

- void deallocate (void *__ptr)

  Deallocates memory by putting the memory block into the pool.

- virtual void recycle ()

  Recycles half of the free memory blocks in the memory pool to the system.

Static Public Member Functions

- static static_mem_pool & instance ()

  Gets the instance of the static memory pool.

- static static_mem_pool & instance_known ()

  Gets the known instance of the static memory pool.

4.19.1 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

4.19.2 Member Function Documentation

4.19.2.1 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

4.19.2.2 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

4.19.2.3 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

4.19.2.4 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

4.19.2.5 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:

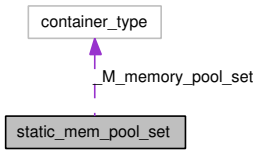
- static_mem_pool.h

4.20 static_mem_pool_set Class Reference

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

`#include <static_mem_pool.h>`

Collaboration diagram for static_mem_pool_set:



Public Types

- typedef class_level_lock< static_mem_pool_set >::lock lock

Public Member Functions

- void recycle ()
  Asks all static memory pools to recycle unused memory blocks back to the system.
- void add (mem_pool_base *__memory_pool_p)
  Adds a new memory pool to static_mem_pool_set.

Static Public Member Functions

- static static_mem_pool_set & instance ()
  Creates the singleton instance of static_mem_pool_set.

4.20.1 Detailed Description

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

4.20.2 Member Typedef Documentation

4.20.2.1 typedef class_level_lock<static_mem_pool_set>::lock static_mem_pool_set::lock

4.20.3 Member Function Documentation

4.20.3.1 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

4.20.3.2 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.

4.20.3.3 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

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

- static_mem_pool.h
- static_mem_pool.cpp

Chapter 5

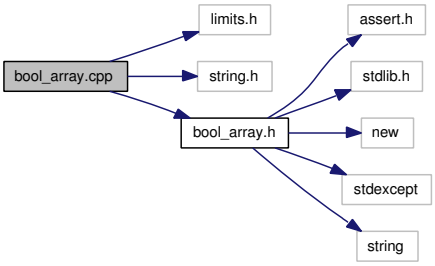
Nvwa File Documentation

5.1 bool_array.cpp File Reference

Code for class bool_array (packed boolean array).

```
#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:

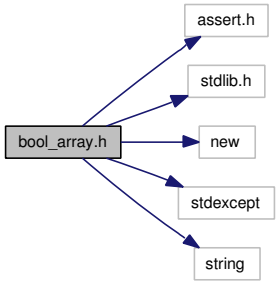
Wu Yongwei

5.2 bool_array.h File Reference

Header file for class bool_array (packed boolean array).

```
#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

  Class to represent a packed boolean array.

- class **bool_array::_Element**

  Class to represent a reference to an array element.

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

5.2.2 Typedef Documentation

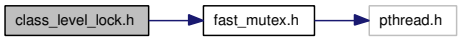
5.2.2.1 typedef unsigned char BYTE

5.3 class_level_lock.h File Reference

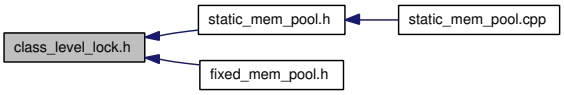
In essence Loki ClassLevelLockable re-engineered to use a fast_mutex class.

`#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 >
  Helper class for class-level locking.
- class class_level_lock< _Host, _RealLock >::lock
  Type that provides locking/unlocking semantics.

5.3.1 Detailed Description

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

Version:

1.13, 2007/12/30

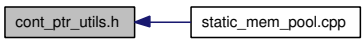
Author:

Wu Yongwei

5.4 cont_ptr_utils.h File Reference

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

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



Classes

- struct dereference

  Functor to return objects pointed by a container of pointers.

- struct dereference_less

  Functor to compare objects pointed by a container of pointers.

- struct delete_object

  Functor to delete objects pointed by a container of pointers.

- struct output_object< _OutputStrm, _StringType >

  Functor to output objects pointed by a container of pointers.

5.4.1 Detailed Description

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

Version:

1.4, 2007/09/12

Author:

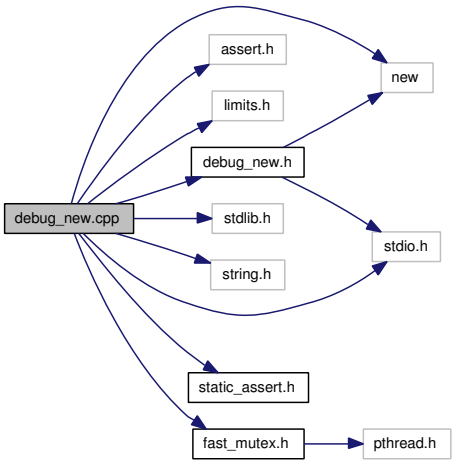
Wu Yongwei

5.5 debug_new.cpp File Reference

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

```
#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

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

Defines

- #define _DEBUG_NEW_ALIGNMENT 16

  The alignment requirement of allocated memory blocks.

- #define _DEBUG_NEW_CALLER_ADDRESS __builtin_return_address(0)

The expression to return the caller address.

- #define _DEBUG_NEW_ERROR_ACTION abort()

  The action to take when an error occurs.

- #define _DEBUG_NEW_FILENAME_LEN 44

  The length of file name stored if greater than zero.

- #define _DEBUG_NEW_PROGNAME NULL

  The program (executable) name to be set at compile time.

- #define _DEBUG_NEW_STD_OPER_NEW 1

  Macro to indicate whether the standard-conformant behaviour of `operator new` *is wanted.*

- #define _DEBUG_NEW_TAILCHECK 0

  Macro to indicate whether a writing-past-end check will be performed.

- #define _DEBUG_NEW_TAILCHECK_CHAR 0xCC

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

- #define _DEBUG_NEW_USE_ADDR2LINE 1

  Whether to use addr2line *to convert a caller address to file/line information.*

- #define _DEBUG_NEW_REDEFINE_NEW 0

  Macro to indicate whether redefinition of `new` *is wanted.*

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

  Gets the aligned value of memory block size.

Functions

- static bool print_position_from_addr (const void *addr)

  Tries printing the position information from an instruction address.

- static void print_position (const void *ptr, int line)

  Prints the position information of a memory operation point.

- static void * alloc_mem (size_t size, const char *file, int line, bool is_array)

  Allocates memory and initializes control data.

- static void free_pointer (void *pointer, void *addr, bool is_array)

  Frees memory and adjusts pointers.

- int check_leaks ()

  Checks for memory leaks.

- int check_mem_corruption ()

  Checks for heap 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

  Magic number for error detection.

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

  The extra memory allocated by `operator new`.

- static new_ptr_list_t new_ptr_list

  List of all new'd pointers.

- static fast_mutex new_ptr_lock

  The mutex guard to protect simultaneous access to the pointer list.

- static fast_mutex new_output_lock

  The mutex guard to protect simultaneous output to new_output_fp.

- static size_t total_mem_alloc = 0

  Total memory allocated in bytes.

- bool new_autocheck_flag = true

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

- bool new_verbose_flag = false

  Flag to control whether verbose messages are output.

- FILE * new_output_fp = stderr

  Pointer to the output stream.

- const char * new_progname = _DEBUG_NEW_PROGNAME

  Pointer to the program name.

5.5.1 Detailed Description

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

Version:

4.14, 2008/10/20

Author:

Wu Yongwei

5.5.2 Define Documentation

5.5.2.1 #define _DEBUG_NEW_ALIGNMENT 16

The alignment requirement of allocated memory blocks.

It must be a power of two.

5.5.2.2 #define _DEBUG_NEW_CALLER_ADDRESS __builtin_return_address(0)

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 #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 #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 #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_PROGNAME=\"a.exe\"` in the Windows command prompt.

5.5.2.6 #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 #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 #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 #define _DEBUG_NEW_TAILCHECK_CHAR 0xCC

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

5.5.2.10 #define _DEBUG_NEW_USE_ADDR2LINE 1

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 #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 static void* alloc_mem (size_t *size*, const char * *file*, int *line*, bool *is_array*) [static]

Allocates memory and initializes control data.

Parameters:

size size of the required memory block

file null-terminated string of the file name

line line number

is_array boolean value whether this is an array operation

Returns:

pointer to the user-requested memory area; `NULL` if memory allocation is not successful

5.5.3.2 int check_leaks ()

Checks for memory leaks.

Returns:

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

5.5.3.3 int check_mem_corruption ()

Checks for heap corruption.

Returns:

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

5.5.3.4 static void free_pointer (void * *pointer*, void * *addr*, bool *is_array*) [static]

Frees memory and adjusts pointers.

Parameters:

pointer pointer to delete

addr pointer to the caller

is_array flag indicating whether it is invoked by a `delete[]` call

5.5.3.5 void operator delete (void * *pointer*, const std::nothrow_t &) throw ()

5.5.3.6 void operator delete (void * *pointer*, const char * *file*, int *line*) throw ()

5.5.3.7 void operator delete (void * *pointer*) throw ()

5.5.3.8 void operator delete[] (void * *pointer*, const std::nothrow_t &) throw ()

5.5.3.9 void operator delete[] (void * *pointer*, const char * *file*, int *line*) throw ()

5.5.3.10 void operator delete[] (void * *pointer*) throw ()

5.5.3.11 void* operator new (size_t *size*, const std::nothrow_t &) throw ()

5.5.3.12 void* operator new (size_t *size*) throw (std::bad_alloc)

5.5.3.13 void* operator new (size_t *size*, const char * *file*, int *line*)

5.5.3.14 void* operator new[] (size_t *size*, const std::nothrow_t &) throw ()

5.5.3.15 void* operator new[] (size_t *size*) throw (std::bad_alloc)

5.5.3.16 void* operator new[] (size_t *size*, const char * *file*, int *line*)

5.5.3.17 static void print_position (const void * *ptr*, int *line*) [static]

Prints the position information of a memory operation point.

When `_DEBUG_NEW_USE_ADDR2LINE` is defined to a non-zero value, this function will try to convert a given caller address to file/line information with *addr2line*.

Parameters:

ptr source file name if *line* is non-zero; caller address otherwise

line source line number if non-zero; indication that *ptr* is the caller address otherwise

5.5.3.18 static bool print_position_from_addr (const void * *addr*) [static]

Tries printing the position information from an instruction address.

This is the version that uses *addr2line*.

Parameters:

addr the instruction address to convert and print

Returns:

`true` if the address is converted successfully (and the result is printed); `false` if no useful information is got (and nothing is printed)

5.5.4 Variable Documentation

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

The extra memory allocated by `operator new`.

5.5.4.2 const unsigned MAGIC = 0x4442474E

Magic number for error detection.

5.5.4.3 bool new_autocheck_flag = true

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

5.5.4.4 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 fast_mutex new_output_lock [static]

The mutex guard to protect simultaneous output to new_output_fp.

5.5.4.6 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("_");`'.

5.5.4.7 new_ptr_list_t new_ptr_list [static]

Initial value:

```
{
    &new_ptr_list,
    &new_ptr_list,
    0,
    {


        ""

    },
    0,
    0,
    MAGIC
}
```

List of all new'd pointers.

5.5.4.8 fast_mutex new_ptr_lock [static]

The mutex guard to protect simultaneous access to the pointer list.

5.5.4.9 bool new_verbose_flag = false

Flag to control whether verbose messages are output.

5.5.4.10 size_t total_mem_alloc = 0 [static]

Total memory allocated in bytes.

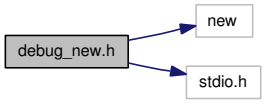
5.6 debug_new.h File Reference

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

`#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
  Recorder class to remember the call context.
- class __debug_new_counter
  Counter class for on-exit leakage check.

Defines

- #define HAVE_PLACEMENT_DELETE 1
  Macro to indicate whether placement delete operators are supported on a certain compiler.
- #define _DEBUG_NEW_REDEFINE_NEW 1
  Macro to indicate whether redefinition of `new` *is wanted.*
- #define DEBUG_NEW __debug_new_recorder(__FILE__, __LINE__) ->* new
  Macro to catch file/line information on allocation.
- #define new DEBUG_NEW

Functions

- int check_leaks ()
  Checks for memory leaks.

- int check_mem_corruption ()

  Checks for heap 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

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

- bool new_verbose_flag

  Flag to control whether verbose messages are output.

- FILE * new_output_fp

  Pointer to the output stream.

- const char * new_progname

  Pointer to the program name.

- static __debug_new_counter __debug_new_count

  Counting object for each file including debug_new.h.

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 Define Documentation

5.6.2.1 #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 #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 #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.

5.6.2.4 #define new DEBUG_NEW

5.6.3 Function Documentation

5.6.3.1 int check_leaks ()

Checks for memory leaks.

Returns:

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

5.6.3.2 int check_mem_corruption ()

Checks for heap corruption.

Returns:

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

5.6.3.3 void operator delete (void * *pointer*, const char * *file*, int *line*) throw ()

5.6.3.4 void operator delete[] (void * *pointer*, const char * *file*, int *line*) throw ()

5.6.3.5 void* operator new (size_t *size*, const char * *file*, int *line*)

5.6.3.6 void* operator new[] (size_t *size*, const char * *file*, int *line*)

5.6.4 Variable Documentation

5.6.4.1 __debug_new_counter __debug_new_count [static]

Counting object for each file including debug_new.h.

5.6.4.2 bool new_autocheck_flag

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

5.6.4.3 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).

5.6.4.4 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("_");`'.

5.6.4.5 bool new_verbose_flag

Flag to control whether verbose messages are output.

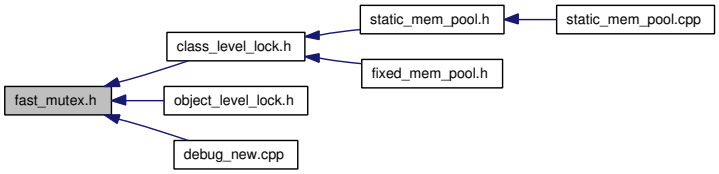
5.7 fast_mutex.h File Reference

A fast mutex implementation for POSIX and Win32.

`#include <pthread.h>`

Include dependency graph for fast_mutex.h:

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



Classes

- class fast_mutex
  Class for non-reentrant fast mutexes.
- class fast_mutex_autolock
  An acquistion-on-initialization lock class based on fast_mutex.

Defines

- #define _FAST_MUTEX_CHECK_INITIALIZATION 1
  Macro to control whether to check for initialization status for each lock/unlock operation.
- #define _FAST_MUTEX_ASSERT(_Expr, _Msg) ((void)0)
  Macro for fast_mutex assertions.
- #define __VOLATILE volatile
  Macro alias to 'volatile' semantics.

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 Define Documentation

5.7.2.1 #define __VOLATILE volatile

Macro alias to 'volatile' semantics.

Here it is truly volatile since it is in a multi-threaded (POSIX threads) environment.

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

Macro for fast_mutex assertions.

Fake version (for release mode).

5.7.2.3 #define _FAST_MUTEX_CHECK_INITIALIZATION 1

Macro to control whether to check for initialization status for each lock/unlock operation.

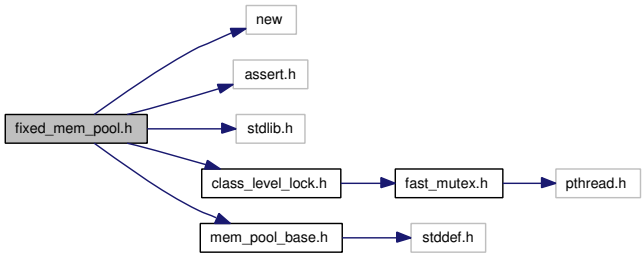
Defining it to a non-zero value will enable the check, so that the construction/destruction of a static object using a static fast_mutex not yet constructed or already destroyed will work (with lock/unlock operations ignored). Defining it to zero will disable to check.

5.8 fixed_mem_pool.h File Reference

Definition of a fixed-size memory pool template for structs/classes.

```
#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 >
  Class template to manipulate a fixed-size memory pool.

Defines

- #define MEM_POOL_ALIGNMENT 4
  Defines the alignment of memory blocks.
- #define DECLARE_FIXED_MEM_POOL(_Cls)
  Declares the normal (exceptionable) overload of **operator new** *and* **operator delete**.
- #define DECLARE_FIXED_MEM_POOL__NOTHROW(_Cls)
  Declares the non-exceptionable overload of **operator new** *and* **operator delete**.
- #define DECLARE_FIXED_MEM_POOL__THROW_NOCHECK(_Cls)
  Declares the exceptionable, non-checking overload of **operator new** *and* **operator delete**.

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_N 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 Define Documentation

5.8.2.1 #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 #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 #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 #define MEM_POOL_ALIGNMENT 4

Defines the alignment of memory blocks.

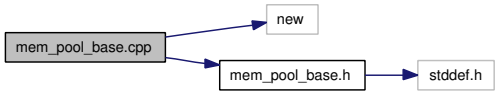
5.9 mem_pool_base.cpp File Reference

Implementation for the memory pool base.

```
#include <new>
```

```
#include "mem_pool_base.h"
```

Include dependency graph for mem_pool_base.cpp:



Defines

- #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

5.9.2 Define Documentation

5.9.2.1 #define _MEM_POOL_ALLOCATE(_Sz) ::operator new((_Sz), std::nothrow)

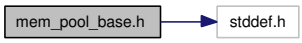
5.9.2.2 #define _MEM_POOL_DEALLOCATE(_Ptr) ::operator delete(_Ptr)

5.10 mem_pool_base.h File Reference

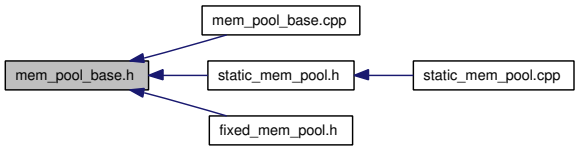
Header file for the memory pool base.

`#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

  Base class for memory pools.

- struct mem_pool_base::_Block_list

  Structure to store the next available memory block.

5.10.1 Detailed Description

Header file for the memory pool base.

Version:

1.1, 2004/07/26

Author:

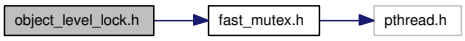
Wu Yongwei

5.11 object_level_lock.h File Reference

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

`#include "fast_mutex.h"`

Include dependency graph for object_level_lock.h:



Classes

- class object_level_lock< _Host >

  Helper class for class-level locking.

- class object_level_lock< _Host >::lock

  Type that provides locking/unlocking semantics.

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

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

`#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

5.12.2 Typedef Documentation

5.12.2.1 typedef double pctimer_t

5.12.3 Function Documentation

5.12.3.1 __inline pctimer_t pctimer (void)

5.13 set_assign.h File Reference

Definition of template functions set_assign_union and set_assign_difference.

`#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

5.13.2 Function Documentation

5.13.2.1 template<class _Container, class _InputIter, class _Compare> _Container& set_assign_difference (_Container & __*dest*, _InputIter __*first*, _InputIter __*last*, _Compare __*comp*)

5.13.2.2 template<class _Container, class _InputIter> _Container& set_assign_difference (_Container & __*dest*, _InputIter __*first*, _InputIter __*last*)

5.13.2.3 template<class _Container, class _InputIter, class _Compare> _Container& set_assign_union (_Container & __*dest*, _InputIter __*first*, _InputIter __*last*, _Compare __*comp*)

5.13.2.4 template<class _Container, class _InputIter> _Container& set_assign_union (_Container & __*dest*, _InputIter __*first*, _InputIter __*last*)

5.14 static_assert.h File Reference

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

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

Classes

- struct __nvwa_compile_time_error< true >

Defines

- #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 Define Documentation

5.14.2.1 #define STATIC_ASSERT(_Expr, _Msg)

Value:

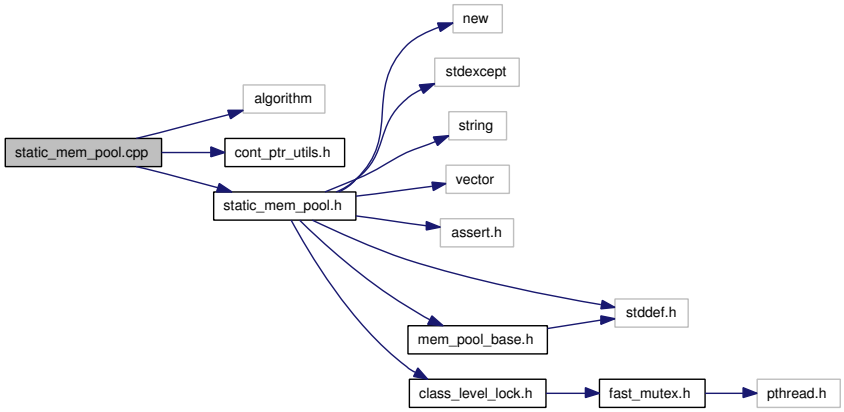
```
{ \
        __nvwa_compile_time_error<((_Expr) != 0)> ERROR_##_Msg; \
        (void)ERROR_##_Msg; \
    }
```

5.15 static_mem_pool.cpp File Reference

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

```
#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:

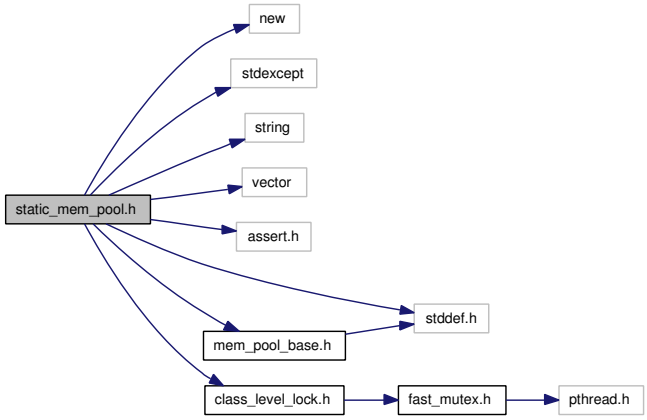
Wu Yongwei

5.16 static_mem_pool.h File Reference

Header file for the 'static' memory pool.

```
#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

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

- class static_mem_pool< _Sz, _Gid >

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

Defines

- #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_MEM_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 Define Documentation

5.16.2.1 #define __PRIVATE private

5.16.2.2 #define _STATIC_MEM_POOL_TRACE(_Lck, _Msg) ((void)0)

5.16.2.3 #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); \
    }
```

5.16.2.4 #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); \
    }
```

5.16.2.5 #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); \
    }
```

5.16.2.6 #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); \
    }
```

5.16.2.7 #define PREPARE_STATIC_MEM_POOL(_Cls) std::cerr << "PREPARE_STATIC_MEM_POOL is obsolete!\n";

5.16.2.8 #define PREPARE_STATIC_MEM_POOL_GROUPED(_Cls, _Gid) std::cerr << "PREPARE_STATIC_MEM_POOL_GROUPED is obsolete!\n";

Index

~__debug_new_counter
    __debug_new_counter, 8
~bool_array
    bool_array, 13
~fast_mutex
    fast_mutex, 23
~fast_mutex_autolock
    fast_mutex_autolock, 25
~lock
    class_level_lock::lock, 19
    object_level_lock::lock, 36
~mem_pool_base
    mem_pool_base, 29
_DEBUG_NEW_ALIGNMENT
    debug_new.cpp, 53
_DEBUG_NEW_CALLER_ADDRESS
    debug_new.cpp, 53
_DEBUG_NEW_ERROR_ACTION
    debug_new.cpp, 53
_DEBUG_NEW_FILENAME_LEN
    debug_new.cpp, 53
_DEBUG_NEW_PROGNAME
    debug_new.cpp, 53
_DEBUG_NEW_REDEFINE_NEW
    debug_new.cpp, 54
    debug_new.h, 61
_DEBUG_NEW_STD_OPER_NEW
    debug_new.cpp, 54
_DEBUG_NEW_TAILCHECK
    debug_new.cpp, 54
_DEBUG_NEW_TAILCHECK_CHAR
    debug_new.cpp, 54
_DEBUG_NEW_USE_ADDR2LINE
    debug_new.cpp, 54
_FAST_MUTEX_ASSERT
    fast_mutex.h, 66
_FAST_MUTEX_CHECK_-
INITIALIZATION
    fast_mutex.h, 66
_MEM_POOL_ALLOCATE
    mem_pool_base.cpp, 70
_MEM_POOL_DEALLOCATE
    mem_pool_base.cpp, 70
_M_next
    mem_pool_base::_Block_list, 31
_STATIC_MEM_POOL_TRACE
    static_mem_pool.h, 79
__PRIVATE
    static_mem_pool.h, 79
__VOLATILE
    fast_mutex.h, 66
__debug_new_count
    debug_new.h, 63
__debug_new_counter, 7
    ~__debug_new_counter, 8
    __debug_new_counter, 7
__debug_new_recorder, 9
    __debug_new_recorder, 9
    operator-> ∗, 9
__nvwa_compile_time_error< true >, 11

add
    static_mem_pool_set, 44
addr
    new_ptr_list_t, 33
align
    debug_new.cpp, 54
ALIGNED_LIST_ITEM_SIZE
    debug_new.cpp, 57
alloc_mem
    debug_new.cpp, 54
alloc_sys
    mem_pool_base, 30
allocate
    fixed_mem_pool, 27
    static_mem_pool, 41
at
    bool_array, 14

bad_alloc_handler
    fixed_mem_pool, 28
bool_array, 12
    ~bool_array, 13
    at, 14
    bool_array, 13
    count, 15
    create, 13
    flip, 15
    initialize, 14
    operator[], 14

reset, 14
set, 15
size, 15
bool_array.cpp, 45
bool_array.h, 46
BYTE, 47
BYTE
bool_array.h, 47

check_leaks
debug_new.cpp, 55
debug_new.h, 62
check_mem_corruption
debug_new.cpp, 55
debug_new.h, 62
class_level_lock, 17
lock, 18
volatile_type, 17
class_level_lock.h, 48
class_level_lock::lock, 19
∼lock, 19
lock, 19
cont_ptr_utils.h, 49
count
bool_array, 15
create
bool_array, 13

dealloc_sys
mem_pool_base, 30
deallocate
fixed_mem_pool, 27
static_mem_pool, 42
DEBUG_NEW
debug_new.h, 62
debug_new.cpp, 50
_DEBUG_NEW_ALIGNMENT, 53
_DEBUG_NEW_CALLER_ADDRESS, 53
_DEBUG_NEW_ERROR_ACTION, 53
_DEBUG_NEW_FILENAME_LEN, 53
_DEBUG_NEW_PROGNAME, 53
_DEBUG_NEW_REDEFINE_NEW, 54
_DEBUG_NEW_STD_OPER_NEW, 54
_DEBUG_NEW_TAILCHECK, 54
_DEBUG_NEW_TAILCHECK_CHAR, 54
_DEBUG_NEW_USE_ADDR2LINE, 54
align, 54
ALIGNED_LIST_ITEM_SIZE, 57
alloc_mem, 54
check_leaks, 55
check_mem_corruption, 55
free_pointer, 55
MAGIC, 57
new_autocheck_flag, 57
new_output_fp, 57
new_output_lock, 58
new_progname, 58
new_ptr_list, 58
new_ptr_lock, 58
new_verbose_flag, 58
operator delete, 55, 56
operator delete[], 56
operator new, 56
operator new[], 56
print_position, 56
print_position_from_addr, 57
total_mem_alloc, 58
debug_new.h, 60
_DEBUG_NEW_REDEFINE_NEW, 61
__debug_new_count, 63
check_leaks, 62
check_mem_corruption, 62
DEBUG_NEW, 62
HAVE_PLACEMENT_DELETE, 62
new, 62
new_autocheck_flag, 63
new_output_fp, 63
new_progname, 63
new_verbose_flag, 63
operator delete, 63
operator delete[], 63
operator new, 63
operator new[], 63
DECLARE_FIXED_MEM_POOL
fixed_mem_pool.h, 68
DECLARE_FIXED_MEM_POOL__NOTHROW
fixed_mem_pool.h, 69
DECLARE_FIXED_MEM_POOL__THROW_NOCHECK
fixed_mem_pool.h, 69
DECLARE_STATIC_MEM_POOL
static_mem_pool.h, 79
DECLARE_STATIC_MEM_POOL__NOTHROW
static_mem_pool.h, 80
DECLARE_STATIC_MEM_POOL_GROUPED
static_mem_pool.h, 80
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW
static_mem_pool.h, 80
deinitialize
fixed_mem_pool, 28
delete_object, 20

operator(), 20
dereference, 21
operator(), 21
dereference_less, 22
operator(), 22

fast_mutex, 23
~fast_mutex, 23
fast_mutex, 23
lock, 23
unlock, 23
fast_mutex.h, 65
_FAST_MUTEX_ASSERT, 66
_FAST_MUTEX_CHECK_-INITIALIZATION, 66
__VOLATILE, 66
fast_mutex_autolock, 25
~fast_mutex_autolock, 25
fast_mutex_autolock, 25
file
new_ptr_list_t, 33
fixed_mem_pool, 26
allocate, 27
bad_alloc_handler, 28
deallocate, 27
deinitialize, 28
get_alloc_count, 28
initialize, 27
is_initialized, 28
lock, 27
fixed_mem_pool.h, 67
DECLARE_FIXED_MEM_POOL, 68
DECLARE_FIXED_MEM_POOL__-NOTHROW, 69
DECLARE_FIXED_MEM_POOL__-THROW_NOCHECK, 69
MEM_POOL_ALIGNMENT, 69
flip
bool_array, 15
free_pointer
debug_new.cpp, 55

get_alloc_count
fixed_mem_pool, 28
get_locked_object
object_level_lock::lock, 37

HAVE_PLACEMENT_DELETE
debug_new.h, 62

initialize
bool_array, 14
fixed_mem_pool, 27
instance
static_mem_pool, 41
static_mem_pool_set, 44
instance_known
static_mem_pool, 41
is_array
new_ptr_list_t, 33
is_initialized
fixed_mem_pool, 28

line
new_ptr_list_t, 33
lock
class_level_lock, 18
class_level_lock::lock, 19
fast_mutex, 23
fixed_mem_pool, 27
object_level_lock, 35
object_level_lock::lock, 36
static_mem_pool_set, 43

MAGIC
debug_new.cpp, 57
magic
new_ptr_list_t, 33
MEM_POOL_ALIGNMENT
fixed_mem_pool.h, 69
mem_pool_base, 29
~mem_pool_base, 29
alloc_sys, 30
dealloc_sys, 30
recycle, 29
mem_pool_base.cpp, 70
_MEM_POOL_ALLOCATE, 70
_MEM_POOL_DEALLOCATE, 70
mem_pool_base.h, 71
mem_pool_base::_Block_list, 31
_M_next, 31

new
debug_new.h, 62
new_autocheck_flag
debug_new.cpp, 57
debug_new.h, 63
new_output_fp
debug_new.cpp, 57
debug_new.h, 63
new_output_lock
debug_new.cpp, 58
new_progname
debug_new.cpp, 58
debug_new.h, 63
new_ptr_list
debug_new.cpp, 58
new_ptr_list_t, 32

addr, 33
file, 33
is_array, 33
line, 33
magic, 33
next, 32
prev, 32
size, 32
new_ptr_lock
debug_new.cpp, 58
new_verbose_flag
debug_new.cpp, 58
debug_new.h, 63
next
new_ptr_list_t, 32

object_level_lock, 34
lock, 35
volatile_type, 34
object_level_lock.h, 72
object_level_lock::lock, 36
∼lock, 36
get_locked_object, 37
lock, 36
operator delete
debug_new.cpp, 55, 56
debug_new.h, 63
operator delete[]
debug_new.cpp, 56
debug_new.h, 63
operator new
debug_new.cpp, 56
debug_new.h, 63
operator new[]
debug_new.cpp, 56
debug_new.h, 63
operator()
delete_object, 20
dereference, 21
dereference_less, 22
output_object, 38
operator-> ∗
__debug_new_recorder, 9
operator[]
bool_array, 14
output_object, 38
operator(), 38
output_object, 38

pctimer
pctimer.h, 73
pctimer.h, 73
pctimer, 73
pctimer_t, 73
pctimer_t
pctimer.h, 73
PREPARE_STATIC_MEM_POOL
static_mem_pool.h, 81
PREPARE_STATIC_MEM_POOL_-
GROUPED
static_mem_pool.h, 81
prev
new_ptr_list_t, 32
print_position
debug_new.cpp, 56
print_position_from_addr
debug_new.cpp, 57

recycle
mem_pool_base, 29
static_mem_pool, 42
static_mem_pool_set, 44
reset
bool_array, 14

set
bool_array, 15
set_assign.h, 74
set_assign_difference, 74
set_assign_union, 75
set_assign_difference
set_assign.h, 74
set_assign_union
set_assign.h, 75
size
bool_array, 15
new_ptr_list_t, 32
STATIC_ASSERT
static_assert.h, 76
static_assert.h, 76
STATIC_ASSERT, 76
static_mem_pool, 40
allocate, 41
deallocate, 42
instance, 41
instance_known, 41
recycle, 42
static_mem_pool.cpp, 77
static_mem_pool.h, 78
_STATIC_MEM_POOL_TRACE, 79
__PRIVATE, 79
DECLARE_STATIC_MEM_POOL, 79
DECLARE_STATIC_MEM_POOL__-
NOTHROW, 80
DECLARE_STATIC_MEM_POOL_-
GROUPED, 80
DECLARE_STATIC_MEM_POOL_-
GROUPED__NOTHROW, 80

PREPARE_STATIC_MEM_POOL, 81
PREPARE_STATIC_MEM_POOL_-
GROUPED, 81
static_mem_pool_set, 43
add, 44
instance, 44
lock, 43
recycle, 44

total_mem_alloc
debug_new.cpp, 58

unlock
fast_mutex, 23

volatile_type
class_level_lock, 17
object_level_lock, 34