Nvwa Reference Manual
1.0

Generated by Doxygen 1.5.2

Tue Dec 31 19:21:18 2013

Contents

Chapter 1

Nvwa Namespace Index

1.1 Nvwa Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Nvwa Hierarchical Index

2.1 Nvwa Class Hierarchy

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

Chapter 3

Nvwa Class Index

3.1 Nvwa Class List

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

Chapter 4

Nvwa File Index

4.1 Nvwa File List

Here is a list of all files with brief descriptions:

Chapter 5

Nvwa Namespace Documentation

5.1 nvwa Namespace Reference

Classes

- class fast_mutex

  Class for non-reentrant fast mutexes.

- class fast_mutex_autolock

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

- class class_level_lock

  Helper class for class-level locking.

- class class_level_lock< _Host, false >

  Partial specialization that makes null locking.

- class object_level_lock

  Helper class for object-level locking.

- class debug_new_recorder

  Recorder class to remember the call context.

- class debug_new_counter

  Counter class for on-exit leakage check.

- struct new_ptr_list_t

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

- class mem_pool_base

  Base class for memory pools.

- class static_mem_pool_set

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

- class static_mem_pool

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

- class fixed_mem_pool

  Class template to manipulate a fixed-size memory pool.

- 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

  Functor to output objects pointed by a container of pointers.

- class bool_array

  Class to represent a packed boolean array.

- class fc_queue

  Class to represent a fixed-capacity queue.

Typedefs

- typedef double pctimer_t

Functions

- int check_leaks ()

  Checks for memory leaks.

- int check_mem_corruption ()

  Checks for heap corruption.

- 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)

- pctimer_t pctimer (void)
- void swap (bool_array &lhs, bool_array &rhs) noexcept

  Exchanges the content of two bool_arrays.

- template<class _Tp, class _Alloc>
  void swap (fc_queue< _Tp, _Alloc > &lhs, fc_queue< _Tp, _Alloc > &rhs)

  Exchanges the elements of two queues.

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.

- const size_t PLATFORM_MEM_ALIGNMENT = sizeof(size_t) ∗ 2

  The platform memory alignment.

- 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.

5.1.1 Typedef Documentation

5.1.1.1 typedef double nvwa::pctimer_t

5.1.2 Function Documentation

5.1.2.1 int nvwa::check_leaks ()

Checks for memory leaks.

Returns:

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

5.1.2.2 int nvwa::check_mem_corruption ()

Checks for heap corruption.

Returns:

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

5.1.2.3 pctimer_t nvwa::pctimer (void) `[inline]`

5.1.2.4 template<class _Container, class _InputIter, class _Compare> _Container& nvwa::set_assign_difference (_Container & *dest*, _InputIter *first*, _InputIter *last*, _Compare *comp*) `[inline]`

5.1.2.5 template<class _Container, class _InputIter> _Container& nvwa::set_assign_difference (_Container & *dest*, _InputIter *first*, _InputIter *last*) `[inline]`

5.1.2.6 template<class _Container, class _InputIter, class _Compare> _Container& nvwa::set_assign_union (_Container & *dest*, _InputIter *first*, _InputIter *last*, _Compare *comp*) `[inline]`

5.1.2.7 template<class _Container, class _InputIter> _Container& nvwa::set_assign_union (_Container & *dest*, _InputIter *first*, _InputIter *last*) `[inline]`

5.1.2.8 template<class _Tp, class _Alloc> void nvwa::swap (fc_queue< _Tp, _Alloc > & *lhs*, fc_queue< _Tp, _Alloc > & *rhs*) `[inline]`

Exchanges the elements of two queues.

Parameters:

lhs the first queue to exchange

rhs the second queue to exchange

Postcondition:

If swapping the allocators does not throw, *lhs* will be swapped with *rhs*. If swapping the allocators throws with strong exception safety guarantee, this function will also provide such guarantee.

5.1.2.9 void nvwa::swap (bool_array & *lhs*, bool_array & *rhs*) `[inline]`

Exchanges the content of two bool_arrays.

Parameters:

lhs the first bool_array to exchange

rhs the second bool_array to exchange

5.1.3 Variable Documentation

5.1.3.1 bool nvwa::new_autocheck_flag = true

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

5.1.3.2 bool nvwa::new_autocheck_flag

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

5.1.3.3 FILE* nvwa::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).

5.1.3.4 FILE∗ nvwa::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).

5.1.3.5 const char∗ nvwa::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("_");`'.

5.1.3.6 const char∗ nvwa::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("_");`'.

5.1.3.7 bool nvwa::new_verbose_flag = false

Flag to control whether verbose messages are output.

5.1.3.8 bool nvwa::new_verbose_flag

Flag to control whether verbose messages are output.

5.1.3.9 const size_t nvwa::PLATFORM_MEM_ALIGNMENT = sizeof(size_t) ∗ 2

The platform memory alignment.

The current value works well in platforms I have tested: Windows XP, Windows 7 x64, and Mac OS X Leopard. It may be smaller than the real alignment, but must be bigger than `sizeof(size_t)` for it work. nvwa::debug_new_recorder uses it to detect misaligned pointer returned by '`new NonPODType[size]`'.

Chapter 6

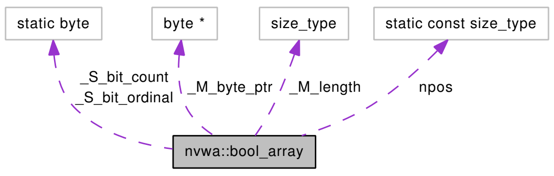
Nvwa Class Documentation

6.1 nvwa::bool_array Class Reference

Class to represent a packed boolean array.

`#include <bool_array.h>`

Collaboration diagram for nvwa::bool_array:



Public Types

- typedef unsigned long size_type

    Type of array indices.

- typedef _Element< byte > reference

    Type of reference.

- typedef _Element< const byte > const_reference

    Type of const reference.

Public Member Functions

- bool_array () noexcept

    Constructs an empty bool_array.

- bool_array (size_type size)

  Constructs a bool_array with a specific size.

- bool_array (const void *ptr, size_type size)

  Constructs a bool_array from a given bitmap.

- ~bool_array ()

  Destroys the bool_array and releases memory.

- bool_array (const bool_array &rhs)

  Copy-constructor.

- bool_array & operator= (const bool_array &rhs)

  Assignment operator.

- bool create (size_type size) noexcept

  Creates the packed boolean array with a specific size.

- void initialize (bool value) noexcept

  Initializes all array elements to a specific value optimally.

- reference operator[] (size_type pos)

  Creates a reference to an array element.

- const_reference operator[] (size_type pos) const

  Creates a const reference to an array element.

- bool at (size_type pos) const

  Reads the boolean value of an array element at a specified position.

- void reset (size_type pos)

  Resets an array element to `false` *at a specified position.*

- void set (size_type pos)

  Sets an array element to `true` *at a specified position.*

- size_type size () const noexcept

  Gets the size of the bool_array.

- size_type count () const noexcept

  Counts elements with a `true` *value.*

- size_type count (size_type begin, size_type end=npos) const

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

- size_type find (bool value, size_type offset=0) const

  Searches for the specified boolean value.

- size_type find (bool value, size_type offset, size_type count) const

  Searches for the specified boolean value.

- size_type find_until (bool value, size_type begin, size_type end) const

  Searches for the specified boolean value.

- void flip () noexcept

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

- void swap (bool_array &rhs) noexcept

  Exchanges the content of this bool_array with another.

- void merge_and (const bool_array &rhs, size_type begin=0, size_type end=npos, size_type offset=0)

  Merges elements of another bool_array with a logical AND.

- void merge_or (const bool_array &rhs, size_type begin=0, size_type end=npos, size_type offset=0)

  Merges elements of another bool_array with a logical OR.

- void copy_to_bitmap (void ∗dest, size_type begin=0, size_type end=npos)

  Copies the bool_array content as bitmap to a specified buffer.

Static Public Member Functions

- static size_t get_num_bytes_from_bits (size_type num_bits)

  Converts the number of bits to number of bytes.

Static Public Attributes

- static const size_type npos = (size_type)-1

  Constant representing 'not found'.

Classes

- class **_Element**

  Class to represent a reference to an array element.

6.1.1 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 reasons:

1. Some compilers (like MSVC 6) did not implement this specialization (and they may not have a `bit_vector` either);
2. I included some additional member functions, like *initialize*, *count*, and *find*, which should be useful;
3. My tests show that the code here is significantly FASTER than `vector<bool>` (and the normal boolean array) under MSVC versions 6/8/9 and GCC versions before 4.3 (while the `vector<bool>` implementations of MSVC 7.1 and GCC 4.3 have performance similar to that of `bool_array`).

6.1.2 Member Typedef Documentation

6.1.2.1 typedef unsigned long nvwa::bool_array::size_type

Type of array indices.

6.1.2.2 typedef _Element<byte> nvwa::bool_array::reference

Type of reference.

6.1.2.3 typedef _Element<const byte> nvwa::bool_array::const_reference

Type of const reference.

6.1.3 Constructor & Destructor Documentation

6.1.3.1 nvwa::bool_array::bool_array () `[inline]`

Constructs an empty bool_array.

6.1.3.2 nvwa::bool_array::bool_array (size_type *size*) `[explicit]`

Constructs a bool_array with a specific size.

Parameters:

size size of the array

Exceptions:

out_of_range *size* equals `0`

bad_alloc memory is insufficient

6.1.3.3 nvwa::bool_array::bool_array (const void ∗ *ptr*, size_type *size*)

Constructs a bool_array from a given bitmap.

Parameters:

ptr pointer to a bitmap
size size of the array

Exceptions:

out_of_range *size* equals `0`
bad_alloc memory is insufficient

6.1.3.4 nvwa::bool_array::∼bool_array () `[inline]`

Destroys the bool_array and releases memory.

6.1.3.5 nvwa::bool_array::bool_array (const bool_array & *rhs*)

Copy-constructor.

Parameters:

rhs the bool_array to copy from

Exceptions:

bad_alloc memory is insufficient

6.1.4 Member Function Documentation

6.1.4.1 bool_array & nvwa::bool_array::operator= (const bool_array & *rhs*)

Assignment operator.

Parameters:

rhs the bool_array to copy from

Exceptions:

bad_alloc memory is insufficient

6.1.4.2 bool nvwa::bool_array::create (size_type *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.

6.1.4.3 void nvwa::bool_array::initialize (bool *value*)

Initializes all array elements to a specific value optimally.

Parameters:

value the boolean value to assign to all elements

6.1.4.4 bool_array::reference nvwa::bool_array::operator[] (size_type *pos*) `[inline]`

Creates a reference to an array element.

Parameters:

pos position of the array element to access

Returns:

reference to the specified element

6.1.4.5 bool_array::const_reference nvwa::bool_array::operator[] (size_type *pos*) const `[inline]`

Creates a const reference to an array element.

Parameters:

pos position of the array element to access

Returns:

const reference to the specified element

6.1.4.6 bool nvwa::bool_array::at (size_type *pos*) const `[inline]`

Reads the boolean value of an array element at a specified position.

Parameters:

pos position of the array element to access

Returns:

the boolean value of the accessed array element

Exceptions:

out_of_range *pos* is greater than the size of the array

6.1.4.7 void nvwa::bool_array::reset (size_type *pos*) `[inline]`

Resets an array element to `false` at a specified position.

Parameters:

pos position of the array element to access

Exceptions:

out_of_range *pos* is greater than the size of the array

6.1.4.8 void nvwa::bool_array::set (size_type *pos*) `[inline]`

Sets an array element to `true` at a specified position.

Parameters:

pos position of the array element to access

Exceptions:

out_of_range *pos* is greater than the size of the array

6.1.4.9 bool_array::size_type nvwa::bool_array::size () const `[inline]`

Gets the size of the bool_array.

Returns:

the number of bits of the bool_array

6.1.4.10 bool_array::size_type nvwa::bool_array::count () const

Counts elements with a `true` value.

Returns:

the count of `true` elements

6.1.4.11 bool_array::size_type nvwa::bool_array::count (size_type *begin*, size_type *end* = npos) const

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

Parameters:

begin beginning of the range

end end of the range (exclusive)

Returns:

the count of `true` elements

Exceptions:

out_of_range the range [begin, end) is invalid

6.1.4.12 bool_array::size_type nvwa::bool_array::find (bool *value*, size_type *offset* = 0) const [inline]

Searches for the specified boolean value.

This function seaches from the specified position (default to beginning) to the end.

Parameters:

offset the position at which the search is to begin

value the boolean value to find

Returns:

position of the first value found if successful; `npos` otherwise

6.1.4.13 bool_array::size_type nvwa::bool_array::find (bool *value*, size_type *offset*, size_type *count*) const [inline]

Searches for the specified boolean value.

This function accepts a range expressed in {position, count}.

Parameters:

offset the position at which the search is to begin

count the number of bits to search

value the boolean value to find

Returns:

position of the first value found if successful; `npos` otherwise

Exceptions:

out_of_range *offset* and/or *count* is too big

6.1.4.14 bool_array::size_type nvwa::bool_array::find_until (bool *value*, size_type *begin*, size_type *end*) const

Searches for the specified boolean value.

This function accepts a range expressed in [begin, end).

Parameters:

begin the position at which the search is to begin

end the end position (exclusive) to stop searching

value the boolean value to find

Returns:

position of the first value found if successful; `npos` otherwise

Exceptions:

out_of_range the range [begin, end) is invalid

6.1.4.15 void nvwa::bool_array::flip ()

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

6.1.4.16 void nvwa::bool_array::swap (bool_array & *rhs*)

Exchanges the content of this bool_array with another.

Parameters:

rhs another bool_array to exchange content with

6.1.4.17 void nvwa::bool_array::merge_and (const bool_array & *rhs*, size_type *begin* = 0, size_type *end* = npos, size_type *offset* = 0)

Merges elements of another bool_array with a logical AND.

Parameters:

rhs another bool_array to merge

begin beginning of the range in *rhs*

end end of the range (exclusive) in *rhs*

offset position to merge in this bool_array

Exceptions:

out_of_range bad range for the source or the destination

6.1.4.18 void nvwa::bool_array::merge_or (const bool_array & *rhs*, size_type *begin* = 0, size_type *end* = npos, size_type *offset* = 0)

Merges elements of another bool_array with a logical OR.

Parameters:

rhs another bool_array to merge

begin beginning of the range in *rhs*

end end of the range (exclusive) in *rhs*

offset position to merge in this bool_array

Exceptions:

out_of_range bad range for the source or the destination

6.1.4.19 void nvwa::bool_array::copy_to_bitmap (void * *dest*, size_type *begin* = 0, size_type *end* = npos)

Copies the bool_array content as bitmap to a specified buffer.

The caller needs to ensure the destination buffer is big enough.

Parameters:

dest address of the destination buffer

begin beginning of the range

end end of the range (exclusive)

Exceptions:

out_of_range bad range for the source or the destination

6.1.4.20 size_t nvwa::bool_array::get_num_bytes_from_bits (size_type *num_bits*) [inline, static]

Converts the number of bits to number of bytes.

Parameters:

num_bits number of bits

Returns:

number of bytes needed to store *num_bits* bits

6.1.5 Member Data Documentation

6.1.5.1 const size_type nvwa::bool_array::npos = (size_type)-1 [static]

Constant representing 'not found'.

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

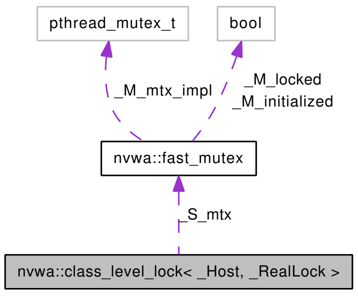
- bool_array.h
- bool_array.cpp

6.2 nvwa::class_level_lock< _Host, _RealLock > Class Template Reference

Helper class for class-level locking.

`#include <class_level_lock.h>`

Collaboration diagram for nvwa::class_level_lock< _Host, _RealLock >:



Public Types

- typedef volatile _Host volatile_type

Friends

- class lock

Classes

- class lock
  Type that provides locking/unlocking semantics.

6.2.1 Detailed Description

template<class _Host, bool _RealLock = true> class nvwa::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.

6.2.2 Member Typedef Documentation

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

6.2.3 Friends And Related Function Documentation

6.2.3.1 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

6.3 nvwa::class_level_lock< _Host, _RealLock >::lock Class Reference

Type that provides locking/unlocking semantics.

`#include <class_level_lock.h>`

Public Member Functions

- lock ()
- ~lock ()

6.3.1 Detailed Description

template<class _Host, bool _RealLock = true> class nvwa::class_level_lock< _-Host, _RealLock >::lock

Type that provides locking/unlocking semantics.

6.3.2 Constructor & Destructor Documentation

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

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

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

- class_level_lock.h

6.4 nvwa::class_level_lock< _Host, false > Class Template Reference

Partial specialization that makes null locking.

`#include <class_level_lock.h>`

Public Types

- typedef _Host volatile_type

Classes

- class lock

  Type that provides locking/unlocking semantics.

6.4.1 Detailed Description

template<class _Host> class nvwa::class_level_lock< _Host, false >

Partial specialization that makes null locking.

6.4.2 Member Typedef Documentation

6.4.2.1 template<class _Host> typedef _Host nvwa::class_level_lock< _Host, false >::volatile_type

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

- class_level_lock.h

6.5 nvwa::class_level_lock< _Host, false >::lock Class Reference

Type that provides locking/unlocking semantics.

`#include <class_level_lock.h>`

Public Member Functions

- lock ()

6.5.1 Detailed Description

template<class _Host> class nvwa::class_level_lock< _Host, false >::lock

Type that provides locking/unlocking semantics.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 template<class _Host> nvwa::class_level_lock< _Host, false >::lock::lock () `[inline]`

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

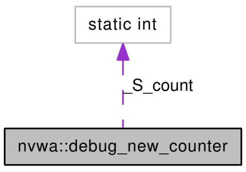
- class_level_lock.h

6.6 nvwa::debug_new_counter Class Reference

Counter class for on-exit leakage check.

`#include <debug_new.h>`

Collaboration diagram for nvwa::debug_new_counter:



Public Member Functions

- debug_new_counter ()

    Constructor to increment the count.

- ~debug_new_counter ()

    Destructor to decrement the count.

6.6.1 Detailed Description

Counter class for on-exit leakage check.

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

6.6.2 Constructor & Destructor Documentation

6.6.2.1 nvwa::debug_new_counter::debug_new_counter ()

Constructor to increment the count.

6.6.2.2 nvwa::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

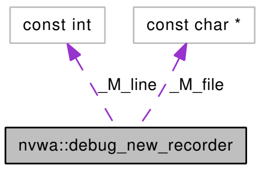
- debug_new.cpp

6.7 nvwa::debug_new_recorder Class Reference

Recorder class to remember the call context.

`#include <debug_new.h>`

Collaboration diagram for nvwa::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 ∗ptr)

  Operator to write the context information to memory.

6.7.1 Detailed Description

Recorder class to remember the call context.

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

6.7.2 Constructor & Destructor Documentation

6.7.2.1 nvwa::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->∗.

6.7.3 Member Function Documentation

6.7.3.1 template<class _Tp> _Tp* nvwa::debug_new_recorder::operator->* (_Tp * *ptr*) `[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

6.8 nvwa::delete_object Struct Reference

Functor to delete objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

Public Member Functions

- template<typename _Tp>
  void operator() (_Tp *ptr) const

6.8.1 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());
```

6.8.2 Member Function Documentation

6.8.2.1 template<typename _Tp> void nvwa::delete_object::operator() (_Tp * *ptr*) const [inline]

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

- cont_ptr_utils.h

6.9 nvwa::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

6.9.1 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());
```

6.9.2 Member Function Documentation

6.9.2.1 template<typename _Tp> const _Tp& nvwa::dereference::operator() (const _Tp * *ptr*) const `[inline]`

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

- cont_ptr_utils.h

6.10 nvwa::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() (const _Pointer &ptr1, const _Pointer &ptr2) const

6.10.1 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;
```

6.10.2 Member Function Documentation

6.10.2.1 template<typename _Pointer> bool nvwa::dereference_less::operator() (const _Pointer & *ptr1*, const _Pointer & *ptr2*) const [inline]

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

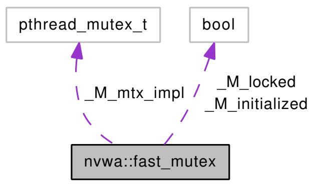
- cont_ptr_utils.h

6.11 nvwa::fast_mutex Class Reference

Class for non-reentrant fast mutexes.

`#include <fast_mutex.h>`

Collaboration diagram for nvwa::fast_mutex:



Public Member Functions

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

6.11.1 Detailed Description

Class for non-reentrant fast mutexes.

This is the implementation for POSIX threads.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 nvwa::fast_mutex::fast_mutex () `[inline]`

6.11.2.2 nvwa::fast_mutex::~fast_mutex () `[inline]`

6.11.3 Member Function Documentation

6.11.3.1 void nvwa::fast_mutex::lock () `[inline]`

6.11.3.2 void nvwa::fast_mutex::unlock () `[inline]`

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

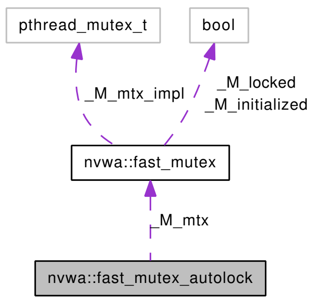
- fast_mutex.h

6.12 nvwa::fast_mutex_autolock Class Reference

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

`#include <fast_mutex.h>`

Collaboration diagram for nvwa::fast_mutex_autolock:



Public Member Functions

- fast_mutex_autolock (fast_mutex &mtx)
- ~fast_mutex_autolock ()

6.12.1 Detailed Description

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

6.12.2 Constructor & Destructor Documentation

6.12.2.1 nvwa::fast_mutex_autolock::fast_mutex_autolock (fast_mutex & *mtx*) `[inline, explicit]`

6.12.2.2 nvwa::fast_mutex_autolock::~fast_mutex_autolock () `[inline]`

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

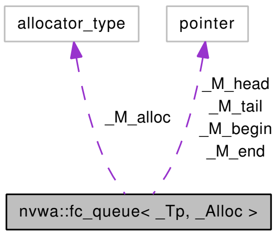
- fast_mutex.h

6.13 nvwa::fc_queue< _Tp, _Alloc > Class Template Reference

Class to represent a fixed-capacity queue.

`#include <fc_queue.h>`

Collaboration diagram for nvwa::fc_queue< _Tp, _Alloc >:



Public Types

- typedef _Tp value_type
- typedef _Alloc allocator_type
- typedef size_t size_type
- typedef value_type ∗ pointer
- typedef const value_type ∗ const_pointer
- typedef value_type & reference
- typedef const value_type & const_reference

Public Member Functions

- fc_queue (size_type max_size, const allocator_type &alloc=allocator_type())

  Constructor that creates the queue with a maximum size (capacity).

- fc_queue (const fc_queue &rhs)

  Copy-constructor that copies all elements from another queue.

- ~fc_queue ()

  Destructor.

- fc_queue & operator= (const fc_queue &rhs)

  Assignment operator that copies all elements from another queue.

- bool empty () const noexcept

  Checks whether the queue is empty (containing no elements).

- bool full () const noexcept

  Checks whether the queue is full (containing the maximum allowed elements).

- size_type capacity () const noexcept

  Gets the maximum number of allowed elements in the queue.

- size_type size () const noexcept

  Gets the number of existing elements in the queue.

- reference front ()

  Gets the first element in the queue.

- const_reference front () const

  Gets the first element in the queue.

- reference back ()

  Gets the last element in the queue.

- const_reference back () const

  Gets the last element in the queue.

- void push (const value_type &value)

  Inserts a new element at the end of the queue.

- void pop ()

  Discards the first element in the queue.

- bool contains (const value_type &value) const

  Checks whether the queue contains a specific element.

- void swap (fc_queue &rhs)

  Exchanges the elements of two queues.

- allocator_type get_allocator () const

  Gets the allocator of the queue.

Protected Member Functions

- pointer increment (pointer ptr) const noexcept
- pointer decrement (pointer ptr) const noexcept
- void construct (void ∗ptr, const _Tp &value)
- void destroy (void ∗ptr)

Protected Attributes

- pointer _M_head
- pointer _M_tail
- pointer _M_begin
- pointer _M_end
- allocator_type _M_alloc

6.13.1 Detailed Description

template<class _Tp, class _Alloc = std::allocator<_Tp>> class nvwa::fc_queue< _Tp, _Alloc >

Class to represent a fixed-capacity queue.

This class has an interface close to `std::queue`, but it allows very efficient and lockless one-producer, one-consumer access, as long as the producer does not try to queue an element when the queue is already full.

Parameters:

_***Tp*** the type of elements in the queue

_***Alloc*** allocator to use for memory management

Precondition:

_*Tp* shall be `CopyConstructible` and `Destructible`, and _*Alloc* shall meet the allocator requirements (Table 28 in the C++11 spec).

6.13.2 Member Typedef Documentation

6.13.2.1 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef _Tp nvwa::fc_queue< _Tp, _Alloc >::value_type

6.13.2.2 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef _Alloc nvwa::fc_queue< _Tp, _Alloc >::allocator_type

6.13.2.3 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef size_t nvwa::fc_queue< _Tp, _Alloc >::size_type

6.13.2.4 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef value_type* nvwa::fc_queue< _Tp, _Alloc >::pointer

6.13.2.5 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef const value_type* nvwa::fc_queue< _Tp, _Alloc >::const_pointer

6.13.2.6 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef value_type& nvwa::fc_queue< _Tp, _Alloc >::reference

6.13.2.7 template<class _Tp, class _Alloc = std::allocator<_Tp>> typedef const value_type& nvwa::fc_queue< _Tp, _Alloc >::const_reference

6.13.3 Constructor & Destructor Documentation

6.13.3.1 template<class _Tp, class _Alloc = std::allocator<_Tp>> nvwa::fc_queue< _Tp, _Alloc >::fc_queue (size_type *max_size*, const allocator_type & *alloc* = allocator_type()) `[inline, explicit]`

Constructor that creates the queue with a maximum size (capacity).

Parameters:

max_size the maximum size allowed
alloc the allocator to use

Precondition:

max_size shall be not be zero

Postcondition:

Unless memory allocation throws an exception, this queue will be constructed with the specified maximum size, and the following conditions will hold:

- `empty()`
- `! full()`
- `capacity() == max_size`
- `size() == 0`
- `get_allocator() == alloc`

6.13.3.2 template<class _Tp, class _Alloc> nvwa::fc_queue< _Tp, _Alloc >::fc_queue (const fc_queue< _Tp, _Alloc > & *rhs*) `[inline]`

Copy-constructor that copies all elements from another queue.

Parameters:

rhs the queue to copy

Postcondition:

If copy-construction is successful (no exception is thrown during memory allocation and element copy), this queue will have the same elements as *rhs*.

6.13.3.3 template<class _Tp, class _Alloc = std::allocator<_Tp>> nvwa::fc_queue< _Tp, _Alloc >::~fc_queue () [inline]

Destructor.

It erases all elements and frees memory.

6.13.4 Member Function Documentation

6.13.4.1 template<class _Tp, class _Alloc = std::allocator<_Tp>> fc_queue& nvwa::fc_queue< _Tp, _Alloc >::operator= (const fc_queue< _Tp, _Alloc > & *rhs*) [inline]

Assignment operator that copies all elements from another queue.

Parameters:

rhs the queue to copy

Postcondition:

If assignment is successful (no exception is thrown during memory allocation and element copy), this queue will have the same elements as *rhs*. Otherwise this queue is unchanged (strong exception safety is guaranteed).

6.13.4.2 template<class _Tp, class _Alloc = std::allocator<_Tp>> bool nvwa::fc_queue< _Tp, _Alloc >::empty () const [inline]

Checks whether the queue is empty (containing no elements).

Returns:

`true` if it is empty; `false` otherwise

6.13.4.3 template<class _Tp, class _Alloc = std::allocator<_Tp>> bool nvwa::fc_queue< _Tp, _Alloc >::full () const [inline]

Checks whether the queue is full (containing the maximum allowed elements).

Returns:

`true` if it is full; `false` otherwise

6.13.4.4 template<class _Tp, class _Alloc = std::allocator<_Tp>> size_type nvwa::fc_queue< _Tp, _Alloc >::capacity () const `[inline]`

Gets the maximum number of allowed elements in the queue.

Returns:

the maximum number of allowed elements in the queue

6.13.4.5 template<class _Tp, class _Alloc = std::allocator<_Tp>> size_type nvwa::fc_queue< _Tp, _Alloc >::size () const `[inline]`

Gets the number of existing elements in the queue.

Returns:

the number of existing elements in the queue

6.13.4.6 template<class _Tp, class _Alloc = std::allocator<_Tp>> reference nvwa::fc_queue< _Tp, _Alloc >::front () `[inline]`

Gets the first element in the queue.

Returns:

reference to the first element

6.13.4.7 template<class _Tp, class _Alloc = std::allocator<_Tp>> const_reference nvwa::fc_queue< _Tp, _Alloc >::front () const `[inline]`

Gets the first element in the queue.

Returns:

const reference to the first element

6.13.4.8 template<class _Tp, class _Alloc = std::allocator<_Tp>> reference nvwa::fc_queue< _Tp, _Alloc >::back () `[inline]`

Gets the last element in the queue.

Returns:

reference to the last element

6.13.4.9 template<class _Tp, class _Alloc = std::allocator<_Tp>> const_reference nvwa::fc_queue< _Tp, _Alloc >::back () const `[inline]`

Gets the last element in the queue.

Returns:

const reference to the last element

6.13.4.10 template<class _Tp, class _Alloc = std::allocator<_Tp>> void nvwa::fc_queue< _Tp, _Alloc >::push (const value_type & *value*) `[inline]`

Inserts a new element at the end of the queue.

The first element will be discarded if the queue is full.

Parameters:

value the value to be inserted

Postcondition:

`size() <= capacity() && back() == value`, unless an exception is thrown, in which case this queue is unchanged (strong exception safety is guaranteed).

6.13.4.11 template<class _Tp, class _Alloc = std::allocator<_Tp>> void nvwa::fc_queue< _Tp, _Alloc >::pop () `[inline]`

Discards the first element in the queue.

Precondition:

This queue is not empty.

Postcondition:

One element is discarded at the front, `size()` is decremented by one, and `full()` is `false`.

6.13.4.12 template<class _Tp, class _Alloc = std::allocator<_Tp>> bool nvwa::fc_queue< _Tp, _Alloc >::contains (const value_type & *value*) const `[inline]`

Checks whether the queue contains a specific element.

Parameters:

value the value to be compared

Precondition:

`value_type` shall be `EqualityComparable`.

Returns:

`true` if found; `false` otherwise

6.13.4.13 template<class _Tp, class _Alloc = std::allocator<_Tp>> void nvwa::fc_queue< _Tp, _Alloc >::swap (fc_queue< _Tp, _Alloc > & *rhs*) `[inline]`

Exchanges the elements of two queues.

Parameters:

rhs the queue to exchange with

Postcondition:

If swapping the allocators does not throw, `*this` will be swapped with *rhs*. If swapping the allocators throws with strong exception safety guarantee, this function will also provide such guarantee.

6.13.4.14 template<class _Tp, class _Alloc = std::allocator<_Tp>> allocator_type nvwa::fc_queue< _Tp, _Alloc >::get_allocator () const `[inline]`

Gets the allocator of the queue.

Returns:

the allocator of the queue

6.13.4.15 template<class _Tp, class _Alloc = std::allocator<_Tp>> pointer nvwa::fc_queue< _Tp, _Alloc >::increment (pointer *ptr*) const `[inline, protected]`

6.13.4.16 template<class _Tp, class _Alloc = std::allocator<_Tp>> pointer nvwa::fc_queue< _Tp, _Alloc >::decrement (pointer *ptr*) const `[inline, protected]`

6.13.4.17 template<class _Tp, class _Alloc = std::allocator<_Tp>> void nvwa::fc_queue< _Tp, _Alloc >::construct (void ∗ *ptr*, const _Tp & *value*) `[inline, protected]`

6.13.4.18 template<class _Tp, class _Alloc = std::allocator<_Tp>> void nvwa::fc_queue< _Tp, _Alloc >::destroy (void ∗ *ptr*) `[inline, protected]`

6.13.5 Member Data Documentation

6.13.5.1 template<class _Tp, class _Alloc = std::allocator<_Tp>> pointer nvwa::fc_queue< _Tp, _Alloc >::_M_head `[protected]`

6.13.5.2 template<class _Tp, class _Alloc = std::allocator<_Tp>> pointer nvwa::fc_queue< _Tp, _Alloc >::_M_tail `[protected]`

6.13.5.3 template<class _Tp, class _Alloc = std::allocator<_Tp>> pointer nvwa::fc_queue< _Tp, _Alloc >::_M_begin `[protected]`

6.13.5.4 template<class _Tp, class _Alloc = std::allocator<_Tp>> pointer nvwa::fc_queue< _Tp, _Alloc >::_M_end `[protected]`

6.13.5.5 template<class _Tp, class _Alloc = std::allocator<_Tp>> allocator_type nvwa::fc_queue< _Tp, _Alloc >::_M_alloc `[protected]`

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

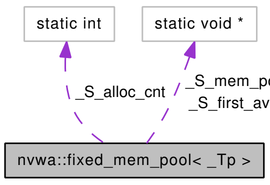
- fc_queue.h

6.14 nvwa::fixed_mem_pool< _Tp > Class Template Reference

Class template to manipulate a fixed-size memory pool.

`#include <fixed_mem_pool.h>`

Collaboration diagram for nvwa::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.

Classes

- struct alignment

  Specializable struct to define the alignment of an object in the fixed_mem_pool.

- struct block_size

  Struct to calculate the block size based on the (specializable) alignment value.

6.14.1 Detailed Description

template<class _Tp> class nvwa::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

6.14.2 Member Typedef Documentation

6.14.2.1 template<class _Tp> typedef class_level_lock<fixed_mem_pool<_Tp> >::lock nvwa::fixed_mem_pool< _Tp >::lock

6.14.3 Member Function Documentation

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

Allocates a memory block from the memory pool.

Returns:

pointer to the allocated memory block

6.14.3.2 template<class _Tp> void nvwa::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

6.14.3.3 template<class _Tp> bool nvwa::fixed_mem_pool< _Tp >::initialize (size_t *size*) `[inline, 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

6.14.3.4 template<class _Tp> int nvwa::fixed_mem_pool< _Tp >::deinitialize () `[inline, 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

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

Gets the allocation count.

Returns:

the number of memory blocks still in allocation

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

Is the memory pool initialized?

Returns:

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

6.14.3.7 template<class _Tp> bool nvwa::fixed_mem_pool< _Tp >::bad_alloc_handler () `[inline, 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:

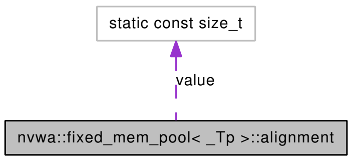
- fixed_mem_pool.h

6.15 nvwa::fixed_mem_pool< _Tp >::alignment Struct Reference

Specializable struct to define the alignment of an object in the fixed_mem_pool.

`#include <fixed_mem_pool.h>`

Collaboration diagram for nvwa::fixed_mem_pool< _Tp >::alignment:



Static Public Attributes

- static const size_t value = MEM_POOL_ALIGNMENT

6.15.1 Detailed Description

template<class _Tp> struct nvwa::fixed_mem_pool< _Tp >::alignment

Specializable struct to define the alignment of an object in the fixed_mem_pool.

6.15.2 Member Data Documentation

6.15.2.1 template<class _Tp> const size_t nvwa::fixed_mem_pool< _Tp >::alignment::value = MEM_POOL_ALIGNMENT `[static]`

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

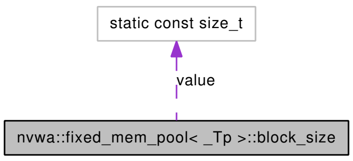
- fixed_mem_pool.h

6.16 nvwa::fixed_mem_pool< _Tp >::block_size Struct Reference

Struct to calculate the block size based on the (specializable) alignment value.

`#include <fixed_mem_pool.h>`

Collaboration diagram for nvwa::fixed_mem_pool< _Tp >::block_size:



Static Public Attributes

- static const size_t value

6.16.1 Detailed Description

template<class _Tp> struct nvwa::fixed_mem_pool< _Tp >::block_size

Struct to calculate the block size based on the (specializable) alignment value.

6.16.2 Member Data Documentation

6.16.2.1 template<class _Tp> const size_t nvwa::fixed_mem_pool< _Tp >::block_size::value [static]

Initial value:

```
(sizeof(_Tp) + fixed_mem_pool<_Tp>::alignment::value - 1)
           & ~(fixed_mem_pool<_Tp>::alignment::value - 1)
```

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

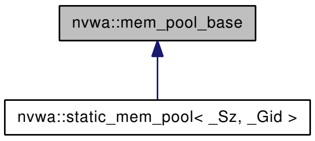
- fixed_mem_pool.h

6.17 nvwa::mem_pool_base Class Reference

Base class for memory pools.

`#include <mem_pool_base.h>`

Inheritance diagram for nvwa::mem_pool_base:



Public Member Functions

- virtual ~mem_pool_base ()

    Empty base destructor.

- virtual void recycle ()=0

    Recycles unused memory from memory pools.

Static Public Member Functions

- static void ∗ alloc_sys (size_t size)

    Allocates memory from the run-time system.

- static void dealloc_sys (void ∗ptr)

    Frees memory and returns it to the run-time system.

Classes

- struct _Block_list

    Structure to store the next available memory block.

6.17.1 Detailed Description

Base class for memory pools.

6.17.2 Constructor & Destructor Documentation

6.17.2.1 nvwa::mem_pool_base::~mem_pool_base () `[virtual]`

Empty base destructor.

6.17.3 Member Function Documentation

6.17.3.1 void nvwa::mem_pool_base::recycle () `[pure virtual]`

Recycles unused memory from memory pools.

It is an interface and needs to be implemented in subclasses.

Implemented in nvwa::static_mem_pool< _Sz, _Gid >.

6.17.3.2 void * nvwa::mem_pool_base::alloc_sys (size_t *size*) `[static]`

Allocates memory from the run-time system.

Parameters:

size size of the memory to allocate in bytes

Returns:

pointer to allocated memory block if successful; or `NULL` if memory allocation fails

6.17.3.3 void nvwa::mem_pool_base::dealloc_sys (void * *ptr*) `[static]`

Frees memory and returns it to the run-time system.

Parameters:

ptr pointer to the memory block previously allocated

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

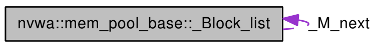
- mem_pool_base.h
- mem_pool_base.cpp

6.18 nvwa::mem_pool_base::_Block_list Struct Reference

Structure to store the next available memory block.

`#include <mem_pool_base.h>`

Collaboration diagram for nvwa::mem_pool_base::_Block_list:



Public Attributes

- _Block_list * _M_next

  Pointer to the next memory block.

6.18.1 Detailed Description

Structure to store the next available memory block.

6.18.2 Member Data Documentation

6.18.2.1 _Block_list∗ nvwa::mem_pool_base::_Block_list::_M_next

Pointer to the next memory block.

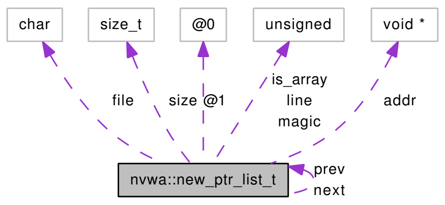
The documentation for this struct was generated from the following file:

- mem_pool_base.h

6.19 nvwa::new_ptr_list_t Struct Reference

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

Collaboration diagram for nvwa::new_ptr_list_t:



Public Attributes

- new_ptr_list_t ∗ next

  Pointer to the next memory block.

- new_ptr_list_t ∗ prev

  Pointer to the previous memory block.

- size_t size

  Size of the memory block.

- union {
  char file [_DEBUG_NEW_FILENAME_LEN]
  File name of the caller.
  void ∗ addr
  Address of the caller to `new`*.*
  };

- unsigned line:31

  Line number of the caller; or `0`*.*

- unsigned is_array:1

  Non-zero iff new[] *is used.*

- unsigned magic

  Magic number for error detection.

6.19.1 Detailed Description

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

6.19.2 Member Data Documentation

6.19.2.1 new_ptr_list_t* nvwa::new_ptr_list_t::next

Pointer to the next memory block.

6.19.2.2 new_ptr_list_t* nvwa::new_ptr_list_t::prev

Pointer to the previous memory block.

6.19.2.3 size_t nvwa::new_ptr_list_t::size

Size of the memory block.

6.19.2.4 char nvwa::new_ptr_list_t::file[_DEBUG_NEW_FILENAME_LEN]

File name of the caller.

6.19.2.5 void* nvwa::new_ptr_list_t::addr

Address of the caller to *new*.

6.19.2.6 union { ... }

6.19.2.7 unsigned nvwa::new_ptr_list_t::line

Line number of the caller; or `0`.

6.19.2.8 unsigned nvwa::new_ptr_list_t::is_array

Non-zero iff *new[]* is used.

6.19.2.9 unsigned nvwa::new_ptr_list_t::magic

Magic number for error detection.

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

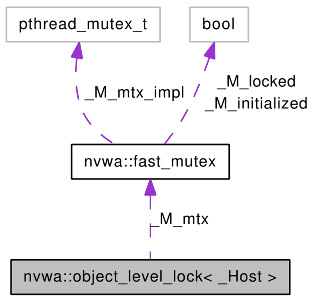
- debug_new.cpp

6.20 nvwa::object_level_lock< _Host > Class Template Reference

Helper class for object-level locking.

`#include <object_level_lock.h>`

Collaboration diagram for nvwa::object_level_lock< _Host >:



Public Types

- typedef volatile _Host volatile_type

Friends

- class lock

Classes

- class lock

  Type that provides locking/unlocking semantics.

6.20.1 Detailed Description

template<class _Host> class nvwa::object_level_lock< _Host >

Helper class for object-level locking.

This is the multi-threaded implementation.

6.20.2 Member Typedef Documentation

6.20.2.1 template<class _Host> typedef volatile _Host nvwa::object_level_lock< _Host >::volatile_type

6.20.3 Friends And Related Function Documentation

6.20.3.1 template<class _Host> friend class lock `[friend]`

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

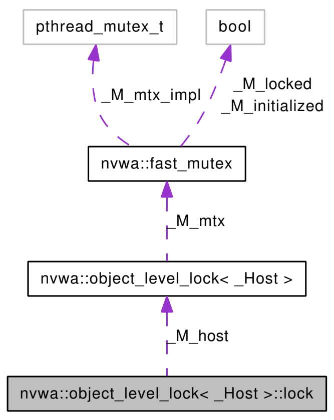
- object_level_lock.h

6.21 nvwa::object_level_lock< _Host >::lock Class Reference

Type that provides locking/unlocking semantics.

`#include <object_level_lock.h>`

Collaboration diagram for nvwa::object_level_lock< _Host >::lock:



Public Member Functions

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

6.21.1 Detailed Description

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

Type that provides locking/unlocking semantics.

6.21.2 Constructor & Destructor Documentation

6.21.2.1 template<class _Host> nvwa::object_level_lock< _Host >::lock::lock (const object_level_lock & *host*) `[inline, explicit]`

6.21.2.2 template<class _Host> nvwa::object_level_lock< _Host >::lock::~lock () `[inline]`

6.21.3 Member Function Documentation

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

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

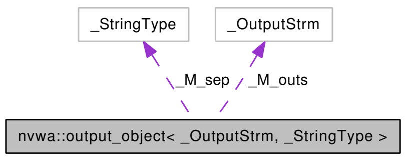
- object_level_lock.h

6.22 nvwa::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 nvwa::output_object< _OutputStrm, _StringType >:



Public Member Functions

- output_object (_OutputStrm &outs, const _StringType &sep)
- template<typename _Pointer>
  void operator() (const _Pointer &ptr) const

6.22.1 Detailed Description

template<typename _OutputStrm, typename _StringType = const char∗> struct nvwa::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, " "));
```

6.22.2 Constructor & Destructor Documentation

6.22.2.1 template<typename _OutputStrm, typename _StringType = const char∗> nvwa::output_object< _OutputStrm, _StringType >::output_object (_OutputStrm & *outs*, const _StringType & *sep*) [inline]

6.22.3 Member Function Documentation

6.22.3.1 template<typename _OutputStrm, typename _StringType = const char*> template<typename _Pointer> void nvwa::output_object< _OutputStrm, _StringType >::operator() (const _Pointer & *ptr*) const `[inline]`

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

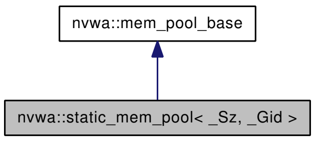
- cont_ptr_utils.h

6.23 nvwa::static_mem_pool< _Sz, _Gid > Class Template Reference

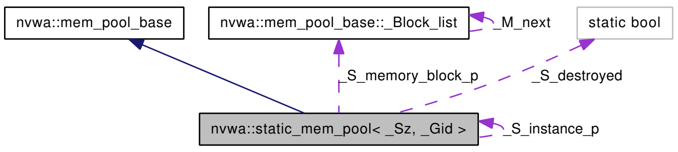
Singleton class template to manage the allocation/deallocation of memory blocks of one specific size.

`#include <static_mem_pool.h>`

Inheritance diagram for nvwa::static_mem_pool< _Sz, _Gid >:



Collaboration diagram for nvwa::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.

6.23.1 Detailed Description

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

6.23.2 Member Function Documentation

6.23.2.1 template<size_t _Sz, int _Gid = -1> static static_mem_pool& nvwa::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

6.23.2.2 template<size_t _Sz, int _Gid = -1> static static_mem_pool& nvwa::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

6.23.2.3 template<size_t _Sz, int _Gid = -1> void* nvwa::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

6.23.2.4 template<size_t _Sz, int _Gid = -1> void nvwa::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

6.23.2.5 template<size_t _Sz, int _Gid> void nvwa::static_mem_pool< _Sz, _Gid >::recycle () `[inline, 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 nvwa::mem_pool_base.

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

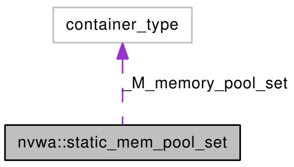
- static_mem_pool.h

6.24 nvwa::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 nvwa::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 nvwa::static_mem_pool_set.

Static Public Member Functions

- static static_mem_pool_set & instance ()

  Gets the singleton instance of nvwa::static_mem_pool_set.

6.24.1 Detailed Description

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

6.24.2 Member Typedef Documentation

6.24.2.1 typedef class_level_lock<static_mem_pool_set>::lock nvwa::static_mem_pool_set::lock

6.24.3 Member Function Documentation

6.24.3.1 static_mem_pool_set & nvwa::static_mem_pool_set::instance () [static]

Gets the singleton instance of nvwa::static_mem_pool_set.

The instance will be created on the first invocation.

Returns:

reference to the instance of nvwa::static_mem_pool_set

6.24.3.2 void nvwa::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 nvwa::static_mem_pool_set during its execution.

6.24.3.3 void nvwa::static_mem_pool_set::add (mem_pool_base * *memory_pool_p*)

Adds a new memory pool to nvwa::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 7

Nvwa File Documentation

7.1 bool_array.cpp File Reference

Code for class bool_array (packed boolean array).

`#include <limits.h>`

`#include <string.h>`

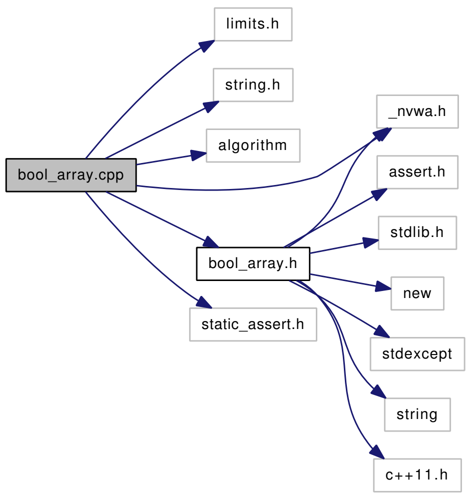
`#include <algorithm>`

`#include "_nvwa.h"`

`#include "bool_array.h"`

`#include "static_assert.h"`

Include dependency graph for bool_array.cpp:



Namespaces

- namespace nvwa

7.1.1 Detailed Description

Code for class bool_array (packed boolean array).

Date:

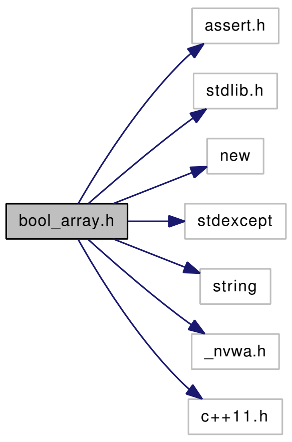
2013-03-01

7.2 bool_array.h File Reference

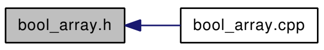
Header file for class bool_array (packed boolean array).

```
#include <assert.h>
#include <stdlib.h>
#include <new>
#include <stdexcept>
#include <string>
#include "_nvwa.h"
#include "c++11.h"
```

Include dependency graph for bool_array.h:



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



Namespaces

- namespace nvwa

Classes

- class nvwa::bool_array

  Class to represent a packed boolean array.

- class **nvwa::bool_array::_Element< _Byte_type >**

  Class to represent a reference to an array element.

Functions

- void nvwa::swap (bool_array &lhs, bool_array &rhs) noexcept

    Exchanges the content of two bool_arrays.

7.2.1 Detailed Description

Header file for class bool_array (packed boolean array).

Date:

2013-10-06

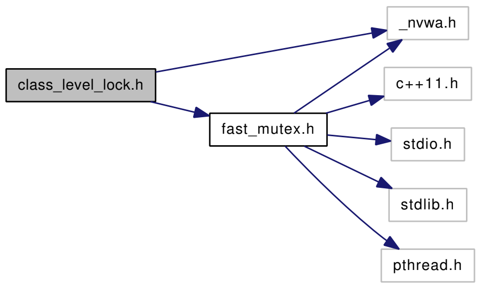
7.3 class_level_lock.h File Reference

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

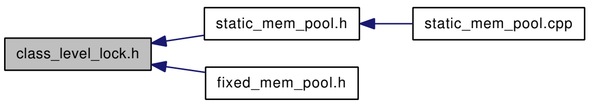
`#include "fast_mutex.h"`

`#include "_nvwa.h"`

Include dependency graph for class_level_lock.h:



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



Namespaces

- namespace nvwa

Classes

- class nvwa::class_level_lock< _Host, _RealLock >

    Helper class for class-level locking.

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

    Type that provides locking/unlocking semantics.

- class nvwa::class_level_lock< _Host, false >

    Partial specialization that makes null locking.

- class nvwa::class_level_lock< _Host, false >::lock

    Type that provides locking/unlocking semantics.

Defines

- #define HAVE_CLASS_TEMPLATE_PARTIAL_SPECIALIZATION 1

7.3.1 Detailed Description

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

Date:

2013-03-04

7.3.2 Define Documentation

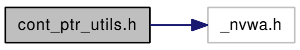
7.3.2.1 #define HAVE_CLASS_TEMPLATE_PARTIAL_SPECIALIZATION 1

7.4 cont_ptr_utils.h File Reference

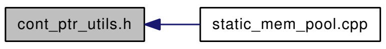
Utility functors for containers of pointers (adapted from Scott Meyers' *Effective STL*).

`#include "_nvwa.h"`

Include dependency graph for cont_ptr_utils.h:



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



Namespaces

- namespace nvwa

Classes

- struct nvwa::dereference

  Functor to return objects pointed by a container of pointers.

- struct nvwa::dereference_less

  Functor to compare objects pointed by a container of pointers.

- struct nvwa::delete_object

  Functor to delete objects pointed by a container of pointers.

- struct nvwa::output_object< _OutputStrm, _StringType >

  Functor to output objects pointed by a container of pointers.

7.4.1 Detailed Description

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

Date:

2013-10-06

7.5 debug_new.cpp File Reference

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

`#include <new>`

`#include <assert.h>`

`#include <stdio.h>`

`#include <stdlib.h>`

`#include <string.h>`

`#include "_nvwa.h"`

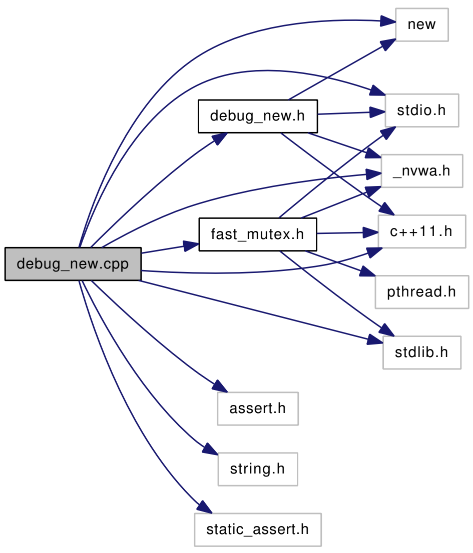
`#include "c++11.h"`

`#include "fast_mutex.h"`

`#include "static_assert.h"`

`#include "debug_new.h"`

Include dependency graph for debug_new.cpp:



Namespaces

- namespace nvwa

Classes

- struct nvwa::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

- int nvwa::check_leaks ()

  Checks for memory leaks.

- int nvwa::check_mem_corruption ()

  Checks for heap corruption.

- void ∗ operator new (size_t size, const char ∗file, int line)

  Allocates memory with file/line information.

- void ∗ operator new[] (size_t size, const char ∗file, int line)

  Allocates array memory with file/line information.

- void ∗ operator new (size_t size) throw (std::bad_alloc)

  Allocates memory without file/line information.

- void ∗ operator new[ ] (size_t size) throw (std::bad_alloc)

  Allocates array memory without file/line information.

- void ∗ operator new (size_t size, const std::nothrow_t &) noexcept

  Allocates memory with no-throw guarantee.

- void ∗ operator new[ ] (size_t size, const std::nothrow_t &) noexcept

  Allocates array memory with no-throw guarantee.

- void operator delete (void ∗ptr) noexcept

  Deallocates memory.

- void operator delete[ ] (void ∗ptr) noexcept

  Deallocates array memory.

- void operator delete (void ∗ptr, const char ∗file, int line) noexcept

  Placement deallocation function.

- void operator delete[ ] (void ∗ptr, const char ∗file, int line) noexcept

  Placement deallocation function.

- void operator delete (void ∗ptr, const std::nothrow_t &) noexcept

  Placement deallocation function.

- void operator delete[ ] (void ∗ptr, const std::nothrow_t &) noexcept

  Placement deallocation function.

Variables

- const size_t nvwa::PLATFORM_MEM_ALIGNMENT = sizeof(size_t) ∗ 2

  The platform memory alignment.

- bool nvwa::new_autocheck_flag = true

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

- bool nvwa::new_verbose_flag = false

  Flag to control whether verbose messages are output.

- FILE ∗ nvwa::new_output_fp = stderr

  Pointer to the output stream.

- const char ∗ nvwa::new_progname = _DEBUG_NEW_PROGNAME

  Pointer to the program name.

7.5.1 Detailed Description

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

Date:

2013-12-31

7.5.2 Define Documentation

7.5.2.1 #define _DEBUG_NEW_ALIGNMENT 16

The alignment requirement of allocated memory blocks.

It must be a power of two.

7.5.2.2 #define _DEBUG_NEW_CALLER_ADDRESS __builtin_return_-address(0)

The expression to return the caller address.

nvwa::print_position will later on use this address to print the position information of memory operation points.

7.5.2.3 #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).

7.5.2.4 #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 value is non-zero (thus to copy the file name) on non-Windows platforms, because I once found that the exit leakage check could not access the address of the file name on Linux (in my case, a core dump occurred when check_leaks tried to access the file name in a shared library after a `SIGINT`). This value makes the size of new_ptr_list_t `64` on non-Windows 32-bit platforms.

7.5.2.5 #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 nvwa::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.

7.5.2.6 #define _DEBUG_NEW_REDEFINE_NEW 0

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

If one wants to define one's own `operator new`, or to call `operator new` directly, 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
```

7.5.2.7 #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.

7.5.2.8 #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.

7.5.2.9 #define _DEBUG_NEW_TAILCHECK_CHAR 0xCC

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

7.5.2.10 #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.

7.5.2.11 #define ALIGN(s) (((s) + _DEBUG_NEW_ALIGNMENT - 1) & ~(_DEBUG_NEW_ALIGNMENT - 1))

Gets the aligned value of memory block size.

7.5.3 Function Documentation

7.5.3.1 void operator delete (void ∗ *ptr*, const std::nothrow_t &)

Placement deallocation function.

For details, please check Section 5.3.4 of the C++ 1998 or 2011 Standard.

Parameters:

ptr	pointer to the previously allocated memory

7.5.3.2 void operator delete (void ∗ *ptr*, const char ∗ *file*, int *line*)

Placement deallocation function.

For details, please check Section 5.3.4 of the C++ 1998 or 2011 Standard.

Parameters:

ptr	pointer to the previously allocated memory
file	null-terminated string of the file name
line	line number

See also:

`http://www.csci.csusb.edu/dick/c++std/cd2/expr.html#expr.new`
`http://wyw.dcweb.cn/leakage.htm`

7.5.3.3 void operator delete (void ∗ *ptr*)

Deallocates memory.

Parameters:

ptr	pointer to the previously allocated memory

7.5.3.4 void operator delete[] (void ∗ *ptr*, const std::nothrow_t &)

Placement deallocation function.

For details, please check Section 5.3.4 of the C++ 1998 or 2011 Standard.

Parameters:

ptr	pointer to the previously allocated memory

7.5.3.5 void operator delete[] (void * *ptr*, const char * *file*, int *line*)

Placement deallocation function.

For details, please check Section 5.3.4 of the C++ 1998 or 2011 Standard.

Parameters:

ptr pointer to the previously allocated memory
file null-terminated string of the file name
line line number

7.5.3.6 void operator delete[] (void * *ptr*)

Deallocates array memory.

Parameters:

ptr pointer to the previously allocated memory

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

Allocates memory with no-throw guarantee.

Parameters:

size size of the required memory block

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient

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

Allocates memory without file/line information.

Parameters:

size size of the required memory block

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 0)

Exceptions:

bad_alloc memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 1)

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

Allocates memory with file/line information.

Parameters:

size size of the required memory block

file null-terminated string of the file name

line line number

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 0)

Exceptions:

bad_alloc memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 1)

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

Allocates array memory with no-throw guarantee.

Parameters:

size size of the required memory block

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient

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

Allocates array memory without file/line information.

Parameters:

size size of the required memory block

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 0)

Exceptions:

bad_alloc memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 1)

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

Allocates array memory with file/line information.

Parameters:

size size of the required memory block

file null-terminated string of the file name

line line number

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 0)

Exceptions:

bad_alloc memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 1)

7.6 debug_new.h File Reference

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

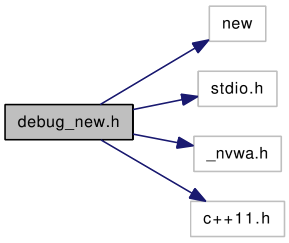
`#include <new>`

`#include <stdio.h>`

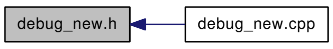
`#include "_nvwa.h"`

`#include "c++11.h"`

Include dependency graph for debug_new.h:



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



Namespaces

- namespace nvwa

Classes

- class nvwa::debug_new_recorder

  Recorder class to remember the call context.

- class nvwa::debug_new_counter

  Counter class for on-exit leakage check.

Defines

- #define _DEBUG_NEW_TYPE 1

  Macro to indicate which variant of DEBUG_NEW is wanted.

- #define DEBUG_NEW NVWA::debug_new_recorder(__FILE__, __LINE__) ->* new

  Macro to catch file/line information on allocation.

Functions

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

  Allocates memory with file/line information.
- void * operator new[] (size_t size, const char *file, int line)

  Allocates array memory with file/line information.
- void operator delete (void *ptr, const char *file, int line) noexcept

  Placement deallocation function.
- void operator delete[] (void *ptr, const char *file, int line) noexcept

  Placement deallocation function.
- int nvwa::check_leaks ()

  Checks for memory leaks.
- int nvwa::check_mem_corruption ()

  Checks for heap corruption.

Variables

- bool nvwa::new_autocheck_flag

  Flag to control whether check_leaks will be automatically called on program exit.
- bool nvwa::new_verbose_flag

  Flag to control whether verbose messages are output.
- FILE * nvwa::new_output_fp

  Pointer to the output stream.
- const char * nvwa::new_progname

  Pointer to the program name.

7.6.1 Detailed Description

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

Date:

2013-10-06

7.6.2 Define Documentation

7.6.2.1 #define _DEBUG_NEW_TYPE 1

Macro to indicate which variant of DEBUG_NEW is wanted.

The default value `1` allows the use of placement new (like `new(std::nothrow)`), but the verbose output (when nvwa::new_verbose_flag is `true`) looks worse than some older versions (no file/line information for allocations). Define it to `2` to revert to the old behaviour that records file and line information directly on the call to `operator new`.

7.6.2.2 #define DEBUG_NEW NVWA::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.

7.6.3 Function Documentation

7.6.3.1 void operator delete (void ∗ *ptr*, const char ∗ *file*, int *line*)

Placement deallocation function.

For details, please check Section 5.3.4 of the C++ 1998 or 2011 Standard.

Parameters:

ptr pointer to the previously allocated memory

file null-terminated string of the file name

line line number

See also:

`http://www.csci.csusb.edu/dick/c++std/cd2/expr.html#expr.new`
`http://wyw.dcweb.cn/leakage.htm`

7.6.3.2 void operator delete[] (void ∗ *ptr*, const char ∗ *file*, int *line*)

Placement deallocation function.

For details, please check Section 5.3.4 of the C++ 1998 or 2011 Standard.

Parameters:

ptr pointer to the previously allocated memory

file null-terminated string of the file name

line line number

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

Allocates memory with file/line information.

Parameters:

size size of the required memory block

file null-terminated string of the file name

line line number

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 0)

Exceptions:

bad_alloc memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 1)

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

Allocates array memory with file/line information.

Parameters:

size size of the required memory block

file null-terminated string of the file name

line line number

Returns:

pointer to the memory allocated; or `NULL` if memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 0)

Exceptions:

bad_alloc memory is insufficient (_DEBUG_NEW_STD_OPER_NEW is 1)

7.7 fast_mutex.h File Reference

A fast mutex implementation for POSIX and Win32.

`#include "_nvwa.h"`

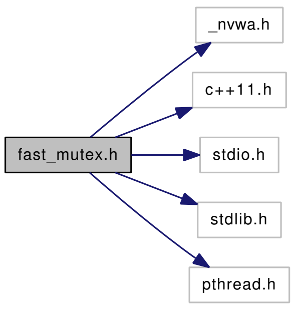
`#include "c++11.h"`

`#include <stdio.h>`

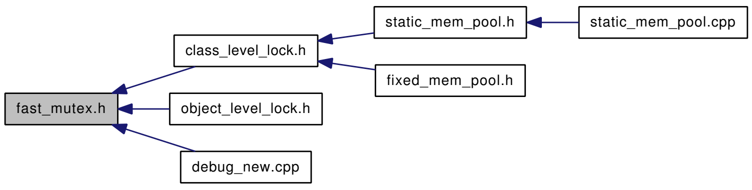
`#include <stdlib.h>`

`#include <pthread.h>`

Include dependency graph for fast_mutex.h:



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



Namespaces

- namespace nvwa

Classes

- class nvwa::fast_mutex

    Class for non-reentrant fast mutexes.

- class nvwa::fast_mutex_autolock

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

Defines

- #define NVWA_USE_CXX11_MUTEX 0
- #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)

    Macro for fast_mutex assertions.

- #define __VOLATILE volatile

    Macro alias to 'volatile' semantics.

7.7.1 Detailed Description

A fast mutex implementation for POSIX and Win32.

Date:

2013-08-02

7.7.2 Define Documentation

7.7.2.1 #define __VOLATILE volatile

Macro alias to 'volatile' semantics.

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

7.7.2.2 #define _FAST_MUTEX_ASSERT(_Expr, _Msg)

Value:

```
if (!(_Expr)) { \
            fprintf(stderr, "fast_mutex::%s\n", _Msg); \
            abort(); \
        }
```

Macro for fast_mutex assertions.

Real version (for debug mode).

7.7.2.3 #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.

7.7.2.4 #define NVWA_USE_CXX11_MUTEX 0

7.8 fc_queue.h File Reference

Definition of a fixed-capacity queue.

`#include <assert.h>`

`#include <stddef.h>`

`#include <memory>`

`#include <new>`

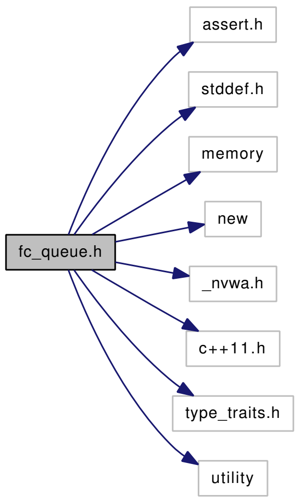
`#include "_nvwa.h"`

`#include "c++11.h"`

`#include "type_traits.h"`

`#include <utility>`

Include dependency graph for fc_queue.h:



Namespaces

- namespace nvwa

Classes

- class nvwa::fc_queue< _Tp, _Alloc >

    Class to represent a fixed-capacity queue.

Functions

- template<class _Tp, class _Alloc>

void nvwa::swap (fc_queue< _Tp, _Alloc > &lhs, fc_queue< _Tp, _Alloc > &rhs)

Exchanges the elements of two queues.

7.8.1 Detailed Description

Definition of a fixed-capacity queue.

Date:

2013-10-07

7.9 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 "_nvwa.h"`

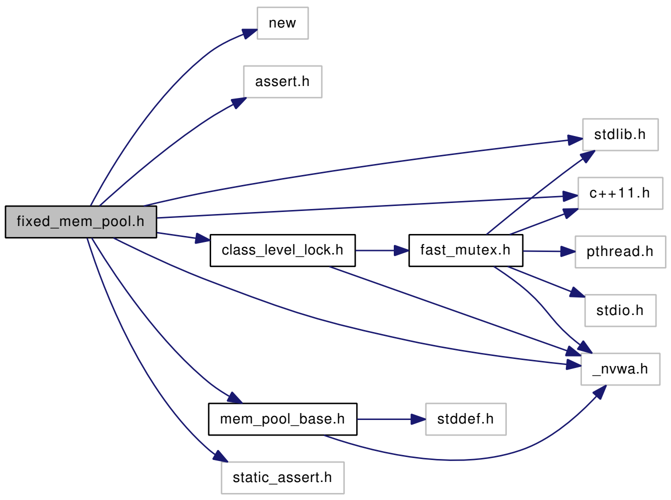
`#include "c++11.h"`

`#include "class_level_lock.h"`

`#include "mem_pool_base.h"`

`#include "static_assert.h"`

Include dependency graph for fixed_mem_pool.h:



Namespaces

- namespace nvwa

Classes

- class nvwa::fixed_mem_pool< _Tp >

  Class template to manipulate a fixed-size memory pool.

- struct nvwa::fixed_mem_pool< _Tp >::alignment

  Specializable struct to define the alignment of an object in the fixed_mem_pool.

- struct nvwa::fixed_mem_pool< _Tp >::block_size

Struct to calculate the block size based on the (specializable) alignment value.

Defines

- #define MEM_POOL_ALIGNMENT sizeof(void∗)
  Defines the alignment of memory blocks.
- #define DECLARE_FIXED_MEM_POOL(_Cls)
  Declares the normal (throwing) allocation and deallocation functions.
- #define DECLARE_FIXED_MEM_POOL__NOTHROW(_Cls)
  Declares the nothrow allocation and deallocation functions.
- #define DECLARE_FIXED_MEM_POOL__THROW_NOCHECK(_Cls)
  Declares the throwing, non-checking allocation and deallocation functions.

7.9.1 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, or
  - DECLARE_FIXED_MEM_POOL__THROW_NOCHECK

  at the end of the class (say, `class` _*Cls*) definitions.
- Optionally, specialize fixed_mem_pool::alignment to change the alignment value for this specific type.
- Optionally, specialize fixed_mem_pool::bad_alloc_handler to change the behaviour when all memory blocks are allocated.
- Call fixed_mem_pool<_Cls>::initialize at the beginning of the program.
- 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.

Date:

2013-10-06

7.9.2 Define Documentation

7.9.2.1 #define DECLARE_FIXED_MEM_POOL(_Cls)

Value:

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

Declares the normal (throwing) allocation and deallocation functions.

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.

7.9.2.2 #define DECLARE_FIXED_MEM_POOL__NOTHROW(_Cls)

Value:

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

Declares the nothrow allocation and deallocation functions.

Parameters:

_*Cls* class to use the fixed_mem_pool

7.9.2.3 #define DECLARE_FIXED_MEM_POOL__THROW_NOCHECK(_-Cls)

Value:

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

Declares the throwing, non-checking allocation and deallocation functions.

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

7.9.2.4 #define MEM_POOL_ALIGNMENT sizeof(void*)

Defines the alignment of memory blocks.

7.10 mem_pool_base.cpp File Reference

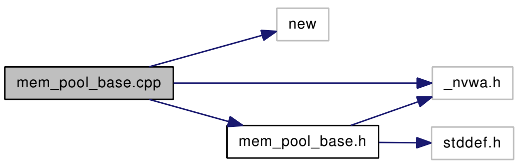
Implementation for the memory pool base.

`#include <new>`

`#include "_nvwa.h"`

`#include "mem_pool_base.h"`

Include dependency graph for mem_pool_base.cpp:



Namespaces

- namespace nvwa

Defines

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

7.10.1 Detailed Description

Implementation for the memory pool base.

Date:

2013-10-06

7.10.2 Define Documentation

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

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

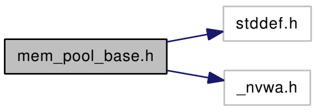
7.11 mem_pool_base.h File Reference

Header file for the memory pool base.

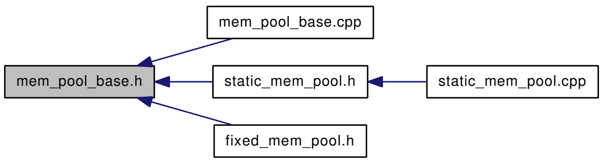
`#include <stddef.h>`

`#include "_nvwa.h"`

Include dependency graph for mem_pool_base.h:



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



Namespaces

- namespace nvwa

Classes

- class nvwa::mem_pool_base

  Base class for memory pools.

- struct nvwa::mem_pool_base::_Block_list

  Structure to store the next available memory block.

7.11.1 Detailed Description

Header file for the memory pool base.

Date:

2013-10-06

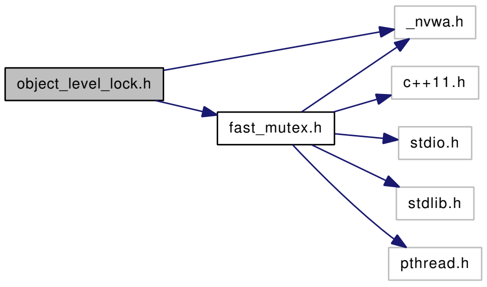
7.12 object_level_lock.h File Reference

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

`#include "fast_mutex.h"`

`#include "_nvwa.h"`

Include dependency graph for object_level_lock.h:



Namespaces

- namespace nvwa

Classes

- class nvwa::object_level_lock< _Host >

  Helper class for object-level locking.

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

  Type that provides locking/unlocking semantics.

7.12.1 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.

Date:

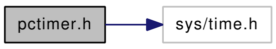
2013-03-01

7.13 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:



Namespaces

- namespace nvwa

Typedefs

- typedef double nvwa::pctimer_t

Functions

- pctimer_t nvwa::pctimer (void)

7.13.1 Detailed Description

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

Date:

2013-03-01

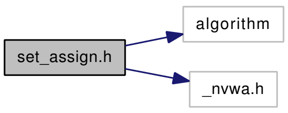
7.14 set_assign.h File Reference

Definition of template functions set_assign_union and set_assign_difference.

`#include <algorithm>`

`#include "_nvwa.h"`

Include dependency graph for set_assign.h:



Namespaces

- namespace nvwa

Functions

- template<class _Container, class _InputIter>
  _Container & nvwa::set_assign_union (_Container &dest, _InputIter first, _InputIter last)
- template<class _Container, class _InputIter, class _Compare>
  _Container & nvwa::set_assign_union (_Container &dest, _InputIter first, _InputIter last, _Compare comp)
- template<class _Container, class _InputIter>
  _Container & nvwa::set_assign_difference (_Container &dest, _InputIter first, _InputIter last)
- template<class _Container, class _InputIter, class _Compare>
  _Container & nvwa::set_assign_difference (_Container &dest, _InputIter first, _InputIter last, _Compare comp)

7.14.1 Detailed Description

Definition of template functions set_assign_union and set_assign_difference.

Date:

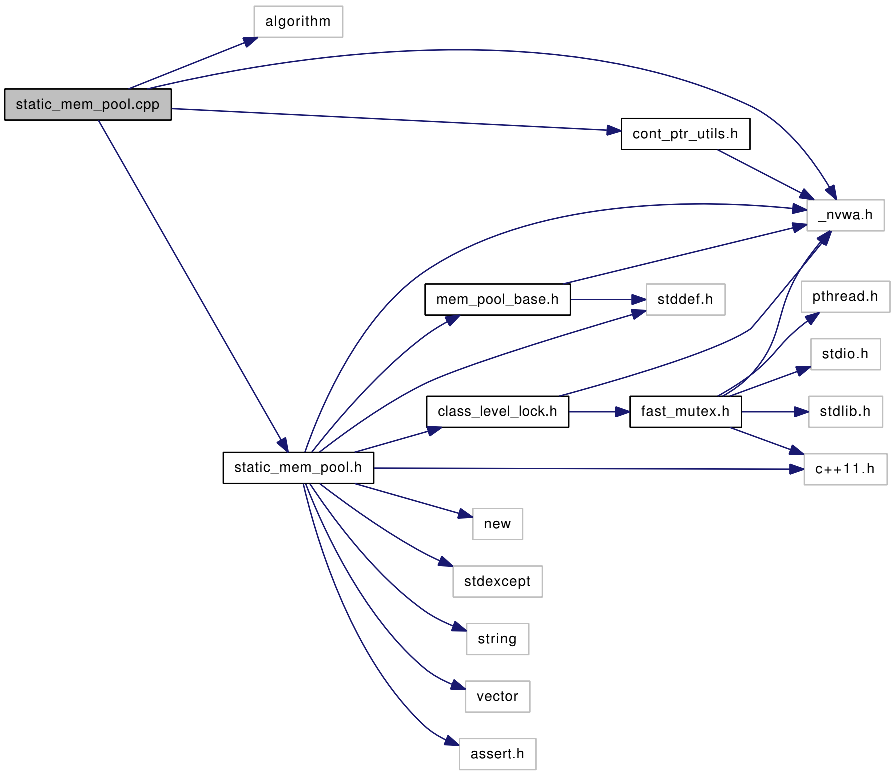
2013-03-01

7.15 static_mem_pool.cpp File Reference

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

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

Include dependency graph for static_mem_pool.cpp:



Namespaces

- namespace nvwa

7.15.1 Detailed Description

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

Date:

2013-03-01

7.16 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>`

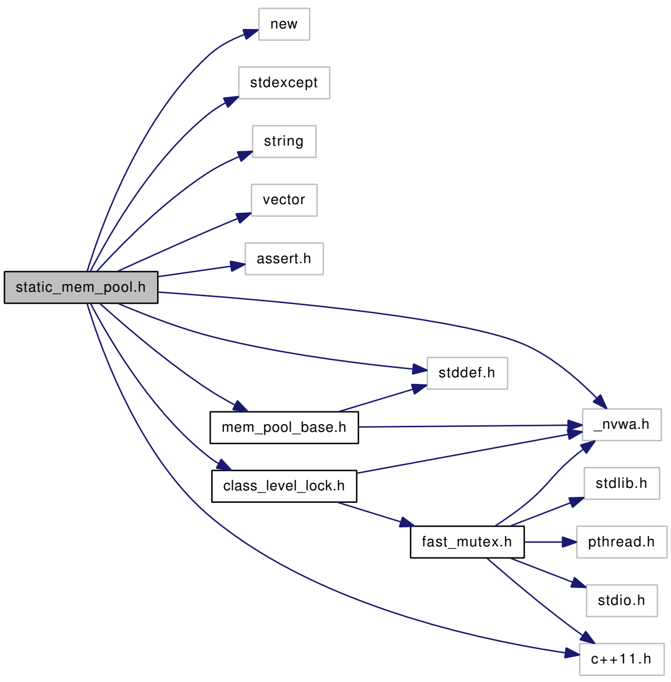
`#include "_nvwa.h"`

`#include "c++11.h"`

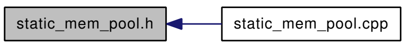
`#include "class_level_lock.h"`

`#include "mem_pool_base.h"`

Include dependency graph for static_mem_pool.h:



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



Namespaces

- namespace nvwa

Classes

- class nvwa::static_mem_pool_set

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

- class nvwa::static_mem_pool< _Sz, _Gid >

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

Defines

- #define _STATIC_MEM_POOL_TRACE(_Lck, _Msg) ((void)0)
- #define DECLARE_STATIC_MEM_POOL(_Cls)

  Declares the normal (throwing) allocation and deallocation functions.

- #define DECLARE_STATIC_MEM_POOL__NOTHROW(_Cls)

  Declares the nothrow allocation and deallocation functions.

- #define DECLARE_STATIC_MEM_POOL_GROUPED(_Cls, _Gid)

  Declares the normal (throwing) allocation and deallocation functions.

- #define DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW(_Cls, _Gid)

  Declares the nothrow allocation and deallocation functions.

7.16.1 Detailed Description

Header file for the 'static' memory pool.

Date:

2013-10-06

7.16.2 Define Documentation

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

7.16.2.2 #define DECLARE_STATIC_MEM_POOL(_Cls)

Value:

```
public: \
    static void* operator new(size_t size) \
    { \
        assert(size == sizeof(_Cls)); \
        void* ptr; \
        ptr = NVWA::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) \
            NVWA::static_mem_pool<sizeof(_Cls)>:: \
                           instance_known().deallocate(ptr); \
    }
```

Declares the normal (throwing) allocation and deallocation functions.

This macro uses the default group.

Parameters:

_*Cls* class to use the static_mem_pool

See also:

DECLARE_STATIC_MEM_POOL__NOTHROW
DECLARE_STATIC_MEM_POOL_GROUPED
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW

7.16.2.3 #define DECLARE_STATIC_MEM_POOL__NOTHROW(_Cls)

Value:

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

Declares the nothrow allocation and deallocation functions.

This macro uses the default group.

Parameters:

_*Cls* class to use the static_mem_pool

See also:

DECLARE_STATIC_MEM_POOL
DECLARE_STATIC_MEM_POOL_GROUPED
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW

7.16.2.4 #define DECLARE_STATIC_MEM_POOL_GROUPED(_Cls, _Gid)

Value:

```
public: \
    static void* operator new(size_t size) \
    { \
        assert(size == sizeof(_Cls)); \
        void* ptr; \
        ptr = NVWA::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) \
            NVWA::static_mem_pool<sizeof(_Cls), (_Gid)>:: \
                           instance_known().deallocate(ptr); \
    }
```

Declares the normal (throwing) allocation and deallocation functions.

Users need to specify a group ID.

Parameters:

_*Cls* class to use the static_mem_pool

_*Gid* group ID (negative to protect multi-threaded access)

See also:

DECLARE_STATIC_MEM_POOL
DECLARE_STATIC_MEM_POOL__NOTHROW
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW

7.16.2.5 #define DECLARE_STATIC_MEM_POOL_GROUPED__-NOTHROW(_Cls, _Gid)

Value:

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

Declares the nothrow allocation and deallocation functions.

Users need to specify a group ID.

Parameters:

_ ***Cls*** class to use the static_mem_pool

_ ***Gid*** group ID (negative to protect multi-threaded access)

See also:

DECLARE_STATIC_MEM_POOL
DECLARE_STATIC_MEM_POOL__NOTHROW
DECLARE_STATIC_MEM_POOL_GROUPED

Index