Nvwa

1.1

Generated by Doxygen 1.8.14

Contents

Chapter 1

Namespace Index

1.1 Namespace List

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

Chapter 5

Namespace Documentation

5.1 nvwa Namespace Reference

Namespace of the nvwa project.

Classes

- class **bad_optional_access**

  Class for bad optional access exception.
- class **basic_mmap_byte_reader**

  Class template to allow iteration over all bytes of a mmappable file.
- class **basic_mmap_line_reader**

  Class template to allow iteration over all lines of a mmappable file.
- class **basic_split_view**

  Class to allow iteration over split items from the input.
- class **bool_array**

  Class to represent a packed boolean array.
- class **breadth_first_iteration**

  Iteration class for breadth-first traversal.
- class **class_level_lock**

  Helper class for class-level locking.
- class **class_level_lock**< **_Host, false** >

  Partial specialization that makes null locking.
- class **debug_new_counter**

  Counter class for on-exit leakage check.
- class **debug_new_recorder**

  Recorder class to remember the call context.
- struct **delete_object**

  Functor to delete objects pointed by a container of pointers.
- class **depth_first_iteration**

  Iteration class for depth-first traversal.
- 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.

- class **fast_mutex**

  Class for non-reentrant fast mutexes.
- class **fast_mutex_autolock**

  *RAII lock class for **fast_mutex** (p. 56).*
- class **fc_queue**

  Class to represent a fixed-capacity queue.
- class **file_line_reader**

  Class to allow iteration over all lines of a text file.
- class **fixed_mem_pool**

  Class template to manipulate a fixed-size memory pool.
- class **in_order_iteration**

  Iteration class for in-order traversal.
- class **istream_line_reader**

  Class to allow iteration over all lines from an input stream.
- class **mem_pool_base**

  Base class for memory pools.
- struct **new_ptr_list_t**

  Structure to store the position information where new occurs.
- class **object_level_lock**

  Helper class for object-level locking.
- class **optional**

  Class for optional values.
- struct **output_object**

  Functor to output objects pointed by a container of pointers.
- struct **smart_ptr**

  Declaration of policy class to generate the smart pointer type.
- struct **smart_ptr**< **_Tp, storage_policy::shared** >

  Partial specialization to get std::shared_ptr.
- struct **smart_ptr**< **_Tp, storage_policy::unique** >

  Partial specialization to get std::unique_ptr.
- class **static_mem_pool**

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

  *Singleton class to maintain a set of existing instantiations of **static_mem_pool** (p. 85).*
- class **tree**

  Basic tree (node) class template that owns all its children.

Typedefs

- typedef void(* **stacktrace_print_callback_t**) (FILE *fp, void **stacktrace)

  Callback type for stack trace printing.
- typedef bool(* **leak_whitelist_callback_t**) (char const *file, int line, void *addr, void **stacktrace)

  Callback type for the leak whitelist function.

Enumerations

- enum **storage_policy** { **storage_policy::unique**, **storage_policy::shared** }

  Policy class for how to store members.

Functions

- int **check_leaks** ()

  Checks for memory leaks.
- int **check_mem_corruption** ()

  Checks for heap corruption.
- static bool **print_position_from_addr** (const void *addr)

  Tries printing the position information from an instruction address.
- static void **print_position** (const void *ptr, int line)

  Prints the position information of a memory operation point.
- static void **print_stacktrace** (void **stacktrace)

  Prints the stack backtrace.
- static bool **is_leak_whitelisted** (**new_ptr_list_t** *ptr)

  Checks whether a leak should be ignored.
- static void * **alloc_mem** (size_t size, const char *file, int line, bool is_array)

  Allocates memory and initializes control data.
- static void **free_pointer** (void *usr_ptr, void *addr, bool is_array)

  Frees memory and adjusts pointers.
- 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) noexcept(noexcept(lhs.↩
  swap(rhs)))

  Exchanges the elements of two queues.
- template<typename _StringType , typename _DelimiterType >
  constexpr **basic_split_view**< _StringType, _DelimiterType > **split** (const _StringType &src, _DelimiterType delimiter) noexcept

  Splits a string (or string_view) into lazy views.
- template<storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY, typename _Tp >
  tree< typename std::decay< _Tp >::type, _Policy >::tree_ptr **create_tree** (_Tp &&value)

  Creates a tree without any children.
- template<storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY, typename _Tp , typename... Args>
  tree< typename std::decay< _Tp >::type, _Policy >::tree_ptr **create_tree** (_Tp &&value, Args &&... args)

  Creates a tree with children.
- template<typename _Fn >
  auto **lift_optional** (_Fn &&f)

  Lifts a function so that it takes optionals and returns an optional.
- template<typename _Fn , typename... _Opt>
  constexpr auto **apply** (_Fn &&f, _Opt &&... args) -> decltype(has_value(args...), **optional**< std::decay_t< decltype(f(std::forward< _Opt >(args).value()...))>>())

  Applies a function to the values of optionals if they are all valid.
- template<typename _Fn , class _Tuple >
  constexpr auto **apply** (_Fn &&f, _Tuple &&t) -> decltype(detail::tuple_apply_impl(std::forward< _Fn >(f), std::forward< _Tuple >(t), std::make_index_sequence< std::tuple_size< std::decay_t< _Tuple >>::value >()))

  Applies the function with all elements of the tuple as arguments.
- template<typename _Fn , typename _T1 , typename _T2 >
  constexpr auto **fmap** (_Fn &&f, const std::pair< _T1, _T2 > &args)

  Applies a function to both elements of a pair, and makes the results a pair.
- template<typename _Fn , typename... _Targs>
  constexpr auto **fmap** (_Fn &&f, const std::tuple< _Targs... > &args)

  Applies a function to all elements of a tuple, and makes the results a tuple.

- template<template< typename, typename > class _OutCont = std::vector, template< typename > class _Alloc = std::allocator, typename _Fn , class _Rng >
  constexpr auto **fmap** (_Fn &&f, _Rng &&inputs) -> decltype(detail::adl_begin(inputs), detail::adl_↩ end(inputs), _OutCont< std::decay_t< decltype(f(∗detail::adl_begin(inputs)))>, _Alloc< std::decay_t< decltype(f(∗detail::adl_begin(inputs)))>>>())

  Applies a function to each element in the input range.
- template<typename _Rs , typename _Fn , typename... _Targs>
  constexpr auto **reduce** (_Fn &&f, const std::tuple< _Targs... > &args, _Rs &&value)

  Applies a function cumulatively to all elements of a tuple.
- template<typename _Fn , class _Rng >
  constexpr auto **reduce** (_Fn &&f, _Rng &&inputs)

  Applies a function cumulatively to elements in the input range.
- template<typename _Rs , typename _Fn , typename _Iter >
  constexpr _Rs && **reduce** (_Fn &&f, _Rs &&value, _Iter begin, _Iter end)

  Applies a function cumulatively to a range.
- template<typename _Rs , typename _Fn , class _Rng >
  constexpr auto **reduce** (_Fn &&f, _Rng &&inputs, _Rs &&initval) -> decltype(f(initval, ∗detail::adl_↩ begin(inputs)))

  Applies a function cumulatively to elements in the input range.
- template<typename _T1 , typename _T2 , typename _Fn >
  constexpr auto **wrap_args_as_pair** (_Fn &&f)

  Makes a two-argument function accept a pair instead.
- template<typename _Tuple , typename _Fn >
  constexpr auto **wrap_args_as_tuple** (_Fn &&f)

  Makes a function accept a tuple as its arguments.
- template<typename _Tp >
  constexpr _Tp **pipeline** (_Tp &&data)

  Returns the data intact to terminate the recursion.
- template<typename _Tp , typename _Fn , typename... _Fargs>
  decltype(auto) constexpr **pipeline** (_Tp &&data, _Fn &&f, _Fargs &&... args)

  Applies the functions in the arguments to the data consecutively.
- auto **compose** ()

  Constructs a function (object) that composes the passed functions.
- template<typename _Fn >
  auto **compose** (_Fn f)

  Constructs a function (object) that composes the passed functions.
- template<typename _Fn , typename... _Fargs>
  auto **compose** (_Fn f, _Fargs... args)

  Constructs a function (object) that composes the passed functions.
- template<typename _Rs , typename _Tp >
  std::function< _Rs(_Tp)> **fix_simple** (std::function< _Rs(std::function< _Rs(_Tp)>, _Tp)> f)

  Generates the fixed point using the simple fixed-point combinator.
- template<typename _Rs , typename _Tp >
  std::function< _Rs(_Tp)> **fix_simple** (std::function< std::function< _Rs(_Tp)>(std::function< _Rs(_↩ Tp)>)> f)

  Generates the fixed point using the simple fixed-point combinator.
- template<typename _Rs , typename _Tp >
  std::function< _Rs(_Tp)> **fix_curry** (std::function< std::function< _Rs(_Tp)>(std::function< _Rs(_Tp)>)> f)

  Generates the fixed point using the Curry-style fixed-point combinator.
- template<typename _Rs , typename... _Targs>
  auto **make_curry** (std::function< _Rs(_Targs...)> f)

  Makes a curried function.

- template<typename _Rs , typename... _Targs>
  auto **make_curry** (_Rs(*f)(_Targs...))

  Makes a curried function.
- template<typename _FnType , typename _Fn >
  auto **make_curry** (_Fn &&f)

  Makes a curried function.

Variables

- bool **new_autocheck_flag** = true

  *Flag to control whether **nvwa::check_leaks*** (p. 16) *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.
- **stacktrace_print_callback_t stacktrace_print_callback** = nullptr

  Pointer to the callback used to print the stack backtrace in case of a memory problem.
- **leak_whitelist_callback_t leak_whitelist_callback** = nullptr

  Pointer to the callback used to filter out false positives from leak reports.
- static **debug_new_counter** **__debug_new_count**

  *Counting object for each file including **debug_new.h*** (p. 106).
- const size_t **PLATFORM_MEM_ALIGNMENT** = sizeof(size_t) * 2

  The platform memory alignment.
- static const unsigned **DEBUG_NEW_MAGIC** = 0x4442474E

  Definition of the constant magic number used for error detection.
- static const int **ALIGNED_LIST_ITEM_SIZE** = **ALIGN**(sizeof(**new_ptr_list_t**))

  The extra memory allocated by `operator new`.
- static **new_ptr_list_t** **new_ptr_list**

  List of all new'd pointers.
- static **fast_mutex** **new_ptr_lock**

  The mutex guard to protect simultaneous access to the pointer list.
- static **fast_mutex** **new_output_lock**

  *The mutex guard to protect simultaneous output to **new_output_fp*** (p. 29).
- static size_t **total_mem_alloc** = 0

  Total memory allocated in bytes.
- const size_t **BUFFER_SIZE** = 256

  Size of buffer.

5.1.1 Detailed Description

Namespace of the nvwa project.

Most functions and global variables are defined in this namespace.

5.1.2 Typedef Documentation

5.1.2.1 leak_whitelist_callback_t

```
typedef bool(* nvwa::leak_whitelist_callback_t) (char const *file, int line, void *addr, void
**stacktrace)
```

Callback type for the leak whitelist function.

file, *address*, and *backtrace* might be null depending on library configuration, platform, and amount of runtime information available. *line* can be 0 when line number info is not available at runtime.

Parameters

file	null-terminated string of the file name
line	line number
addr	address of code where leakage happens
stacktrace	pointer to the stack trace array (null-terminated)

Returns

`true` if the leak should be whitelisted; `false` otherwise

5.1.2.2 stacktrace_print_callback_t

```
typedef void(* nvwa::stacktrace_print_callback_t) (FILE *fp, void **stacktrace)
```

Callback type for stack trace printing.

Parameters

fp	pointer to the output stream
stacktrace	pointer to the stack trace array (null-terminated)

5.1.3 Enumeration Type Documentation

5.1.3.1 storage_policy

```
enum nvwa::storage_policy [strong]
```

Policy class for how to store members.

Enumerator

unique	Members are directly owned.
shared	Members may be shared and passed around.

5.1.4 Function Documentation

5.1.4.1 alloc_mem()

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

Allocates memory and initializes control data.

Parameters

size	size of the required memory block
file	null-terminated string of the file name
line	line number
is_array	boolean value whether this is an array operation

Returns

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

5.1.4.2 apply() [1/2]

```
template<typename _Fn , typename...  _Opt>
constexpr auto nvwa::apply (
            _Fn && f,
            _Opt &&...  args ) -> decltype( has_value(args...),  optional< std::decay_↩
t<decltype(f(std::forward<_Opt>(args).value()...))>>())
```

Applies a function to the values of optionals if they are all valid.

If any of the optionals are invalid, the result is invalid too.

Parameters

f	the function to apply
args	optionals whose values will be passed to *f*

Returns

an optional that either is invalid (if any of the input is invalid) or contains the output of *f*

5.1.4.3 apply() [2/2]

```
template<typename _Fn , class _Tuple >
constexpr auto nvwa::apply (
            _Fn && f,
            _Tuple && t ) -> decltype(detail::tuple_apply_impl( std::forward<_Fn>(f), std↩
::forward<_Tuple>(t), std::make_index_sequence< std::tuple_size<std::decay_t<_Tuple>>↩
::value>()))
```

Applies the function with all elements of the tuple as arguments.

It is exactly like the C++17 std::apply.

Parameters

f	the function to apply
t	the tuple that can expand to the function arguments

Returns

the result of the function applied

5.1.4.4 check_leaks()

```
int nvwa::check_leaks ( )
```

Checks for memory leaks.

Returns

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

5.1.4.5 check_mem_corruption()

```
int nvwa::check_mem_corruption ( )
```

Checks for heap corruption.

Returns

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

5.1.4.6 compose() [1/3]

```
auto nvwa::compose ( )
```

Constructs a function (object) that composes the passed functions.

Returns

the forwarding function

5.1.4.7 compose() [2/3]

```
template<typename _Fn >
auto nvwa::compose (
            _Fn f )
```

Constructs a function (object) that composes the passed functions.

Parameters

f	the function to compose

Returns

the function object that composes the passed function

5.1.4.8 compose() [3/3]

```
template<typename _Fn , typename...  _Fargs>
auto nvwa::compose (
            _Fn f,
            _Fargs...  args )
```

Constructs a function (object) that composes the passed functions.

Parameters

f	the last function to compose
args	the other functions to compose

Returns

the function object that composes the passed functions

5.1.4.9 create_tree() [1/2]

```
template<storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY, typename _Tp >
tree<typename std::decay<_Tp>::type, _Policy>::tree_ptr nvwa::create_tree (
            _Tp && value )
```

Creates a tree without any children.

Parameters

value	the value to assign to the tree node

Returns

the unique_ptr to the newly created tree

5.1.4.10 create_tree() [2/2]

```
template<storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY, typename _Tp , typename...
Args>
tree<typename std::decay<_Tp>::type, _Policy>::tree_ptr nvwa::create_tree (
            _Tp && value,
            Args &&...  args )
```

Creates a tree with children.

Parameters

value	the value to assign to the tree node
args	the unique_ptrs to children of the tree

Returns

the unique_ptr to the newly created tree

5.1.4.11 fix_curry()

```
template<typename _Rs , typename _Tp >
std::function<_Rs(_Tp)> nvwa::fix_curry (
            std::function< std::function< _Rs(_Tp)>(std::function< _Rs(_Tp)>)> f )
```

Generates the fixed point using the Curry-style fixed-point combinator.

This function takes a curried function of one parameter.

Parameters

f	the second-order function to combine with

Returns

the fixed point

5.1.4.12 fix_simple() [1/2]

```
template<typename _Rs , typename _Tp >
std::function<_Rs(_Tp)> nvwa::fix_simple (
            std::function< _Rs(std::function< _Rs(_Tp)>, _Tp)> f )
```

Generates the fixed point using the simple fixed-point combinator.

This function takes a non-curried function of two parameters.

Parameters

f	the second-order function to combine with

Returns

the fixed point

5.1.4.13 fix_simple() [2/2]

```
template<typename _Rs , typename _Tp >
std::function<_Rs(_Tp)> nvwa::fix_simple (
            std::function< std::function< _Rs(_Tp)>(std::function< _Rs(_Tp)>)> f )
```

Generates the fixed point using the simple fixed-point combinator.

This function takes a curried function of one parameter.

Parameters

f	the second-order function to combine with

Returns

the fixed point

5.1.4.14 fmap() [1/3]

```
template<typename _Fn , typename _T1 , typename _T2 >
constexpr auto nvwa::fmap (
            _Fn && f,
            const std::pair< _T1, _T2 > & args )
```

Applies a function to both elements of a pair, and makes the results a pair.

Parameters

f	the function to apply
args	pair of arguments to pass

Returns

pair of results of function invocation

5.1.4.15 fmap() [2/3]

```
template<typename _Fn , typename...  _Targs>
constexpr auto nvwa::fmap (
            _Fn && f,
            const std::tuple< _Targs...  > & args )
```

Applies a function to all elements of a tuple, and makes the results a tuple.

Parameters

f	the function to apply
args	tuple of arguments to pass

Returns

tuple of results of function invocation

5.1.4.16 fmap() [3/3]

```
template<template< typename, typename > class _OutCont = std::vector, template< typename >
class _Alloc = std::allocator, typename _Fn , class _Rng >
constexpr auto nvwa::fmap (
            _Fn && f,
            _Rng && inputs ) -> decltype( detail::adl_begin(inputs), detail::adl_end(inputs),
_OutCont< std::decay_t<decltype(f(*detail::adl_begin(inputs)))>, _Alloc<std::decay_t<decltype(f(*detail↩
::adl_begin(inputs)))>>>())
```

Applies a function to each element in the input range.

This is similar to `std::transform`, but the style is more functional and more suitable for chaining operations.

Parameters

f	the function to apply
inputs	the input range

Precondition

f shall take one argument of the type of the elements in *inputs*, the output container shall support `push_back`, and the input range shall support iteration.

Returns

the container of results

5.1.4.17 free_pointer()

```
static void nvwa::free_pointer (
            void * usr_ptr,
            void * addr,
            bool is_array ) [static]
```

Frees memory and adjusts pointers.

Parameters

usr_ptr	pointer to the previously allocated memory
addr	pointer to the caller
is_array	flag indicating whether it is invoked by a `delete[]` call

5.1.4.18 is_leak_whitelisted()

```
static bool nvwa::is_leak_whitelisted (
             new_ptr_list_t * ptr ) [static]
```

Checks whether a leak should be ignored.

Its runtime performance depends on the callback **nvwa::leak_whitelist_callback** (p. 29).

Parameters

ptr	pointer to a **new_ptr_list_t** (p. 81) struct

Returns

`true` if the leak should be whitelisted; `false` otherwise

5.1.4.19 lift_optional()

```
template<typename _Fn >
auto nvwa::lift_optional (
            _Fn && f )
```

Lifts a function so that it takes optionals and returns an optional.

Parameters

f	function to lift

Returns

the lifted function

5.1.4.20 make_curry() [1/3]

```
template<typename _Rs , typename...  _Targs>
auto nvwa::make_curry (
            std::function< _Rs(_Targs...)> f )
```

Makes a curried function.

The returned function takes one argument at a time, and return a function that takes the next argument until all arguments are exhausted, in which case it returns the final result. This overload takes an std::function and can deduce its argument types and return type.

Parameters

f	the function to be curried as a `std::function`

Returns

the curried function

5.1.4.21 make_curry() [2/3]

```
template<typename _Rs , typename...  _Targs>
auto nvwa::make_curry (
            _Rs(*)(_Targs...)  f )
```

Makes a curried function.

The returned function takes one argument at a time, and return a function that takes the next argument until all arguments are exhausted, in which case it returns the final result. This overload takes a pointer to function and can deduce its argument types and return type.

Parameters

f	the function to be curried as a function pointer

Returns

the curried function

5.1.4.22 make_curry() [3/3]

```
template<typename _FnType , typename _Fn >
auto nvwa::make_curry (
            _Fn && f )
```

Makes a curried function.

The returned function takes one argument at a time, and return a function that takes the next argument until all arguments are exhausted, in which case it returns the final result. This overload takes a generic function object, and the function type must be specified when this function template is instantiated.

Parameters

f	the function to be curried as a function pointer

Returns

the curried function

5.1.4.23 pipeline()

```
template<typename _Tp , typename _Fn , typename...  _Fargs>
decltype(auto) constexpr nvwa::pipeline (
            _Tp && data,
            _Fn && f,
            _Fargs &&...  args )
```

Applies the functions in the arguments to the data consecutively.

Parameters

data	the data to operate on
f	the first function to apply
args	the rest functions to apply

5.1.4.24 print_position()

```
static void nvwa::print_position (
            const void * ptr,
            int line )  [static]
```

Prints the position information of a memory operation point.

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

Parameters

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

5.1.4.25 print_position_from_addr()

```
static bool nvwa::print_position_from_addr (
            const void * addr )  [static]
```

Tries printing the position information from an instruction address.

This is the version that uses *addr2line*.

Parameters

addr	the instruction address to convert and print

Returns

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

5.1.4.26 print_stacktrace()

```
static void nvwa::print_stacktrace (
            void ** stacktrace )  [static]
```

Prints the stack backtrace.

When **nvwa::stacktrace_print_callback** (p. 30) is not null, it is used for printing the stacktrace items. Default implementation of call stack printing is very spartan—only stack frame pointers are printed—but even that output is still useful. Just do address lookup in LLDB etc.

Parameters

stacktrace	pointer to the stack trace array

5.1.4.27 reduce() [1/4]

```
template<typename _Rs , typename _Fn , typename...  _Targs>
constexpr auto nvwa::reduce (
            _Fn && f,
            const std::tuple< _Targs...  > & args,
            _Rs && value )
```

Applies a function cumulatively to all elements of a tuple.

Parameters

f	the function to apply
value	the first argument to be passed to the function
args	the input tuple

Precondition

f shall take one argument of the result type, and one argument of the type of the elements in *args*.

5.1.4.28 reduce() [2/4]

```
template<typename _Fn , class _Rng >
constexpr auto nvwa::reduce (
            _Fn && f,
            _Rng && inputs )
```

Applies a function cumulatively to elements in the input range.

This is similar to `std::accumulate`, but the style is more functional and more suitable for chaining operations.

Parameters

f	the function to apply
inputs	the input range

Precondition

f shall take two arguments of the type of the elements in *inputs*, and the input range shall support iteration.

5.1.4.29 reduce() [3/4]

```
template<typename _Rs , typename _Fn , typename _Iter >
constexpr _Rs&& nvwa::reduce (
            _Fn && f,
            _Rs && value,
            _Iter begin,
            _Iter end )
```

Applies a function cumulatively to a range.

This is similar to `std::accumulate`, but the interface is different, and *f* is allowed to have arguments of different types (the first argument shall have the same type as the function return type). Perfect forwarding allows the result to be a reference type.

Parameters

f	the function to apply
value	the first argument to be passed to *f*
begin	beginning of the range
end	end of the range

Precondition

f shall take one argument of the result type, and one argument of the type of the elements in *inputs*, and the input range shall support iteration.

5.1.4.30 reduce() [4/4]

```
template<typename _Rs , typename _Fn , class _Rng >
constexpr auto nvwa::reduce (
            _Fn && f,
            _Rng && inputs,
            _Rs && initval ) -> decltype(f(initval, *detail::adl_begin(inputs)))
```

Applies a function cumulatively to elements in the input range.

This is similar to `std::accumulate`, but the style is more functional and more suitable for chaining operations. This implementation allows the two arguments of *f* to have different types (the first argument shall have the same type as the function return type). Perfect forwarding allows the result to be a reference type, and it is possible to write code like:

```
// Declaration of the output function
std::ostream& print(std::ostream& os, const my_type&);
...
// Dump all contents of the container_of_my_type to cout
nvwa::reduce(print, container_of_my_type, std::cout);
```

Parameters

f	the function to apply
inputs	the input range
initval	initial value for the cumulative calculation

Precondition

f shall take one argument of the result type, and one argument of the type of the elements in *inputs*, and the input range shall support iteration.

5.1.4.31 split()

```
template<typename _StringType , typename _DelimiterType >
constexpr  basic_split_view<_StringType, _DelimiterType> nvwa::split (
             const _StringType & src,
             _DelimiterType delimiter )  [noexcept]
```

Splits a string (or string_view) into lazy views.

The source input shall remain unchanged when the generated **basic_split_view** (p. 36) is used in anyway.

Parameters

src	the source input to be split
delimiter	delimiter used to split *src*; its type should be the same as that of *src*, or its character type

5.1.4.32 swap() [1/2]

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

Exchanges the content of two bool_arrays.

Parameters

lhs	the first **bool_array** (p. 38) to exchange
rhs	the second **bool_array** (p. 38) to exchange

5.1.4.33 swap() [2/2]

```
template<class _Tp , class _Alloc >
```

```
void nvwa::swap (
              fc_queue< _Tp, _Alloc > & lhs,
              fc_queue< _Tp, _Alloc > & rhs )  [noexcept]
```

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.

5.1.4.34 wrap_args_as_pair()

```
template<typename _T1 , typename _T2 , typename _Fn >
constexpr auto nvwa::wrap_args_as_pair (
             _Fn && f )
```

Makes a two-argument function accept a pair instead.

Parameters

f	a function that accepts two arguments

Returns

a function that accepts a pair

5.1.4.35 wrap_args_as_tuple()

```
template<typename _Tuple , typename _Fn >
constexpr auto nvwa::wrap_args_as_tuple (
             _Fn && f )
```

Makes a function accept a tuple as its arguments.

The tuple shall contain exactly the same type/number of argument as the function needs.

Parameters

f	a function that accepts two arguments

Returns

a function that accepts a pair

5.1.5 Variable Documentation

5.1.5.1 __debug_new_count

```
debug_new_counter nvwa::__debug_new_count [static]
```

Counting object for each file including **debug_new.h** (p. 106).

5.1.5.2 BUFFER_SIZE

```
const size_t nvwa::BUFFER_SIZE = 256
```

Size of buffer.

5.1.5.3 leak_whitelist_callback

```
leak_whitelist_callback_t nvwa::leak_whitelist_callback = nullptr
```

Pointer to the callback used to filter out false positives from leak reports.

A null value means the lack of filtering.

5.1.5.4 new_output_fp

```
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** (p. 107) is `true` and there are a lot of (de)allocations).

5.1.5.5 new_progname

```
const char * nvwa::new_progname = _DEBUG_NEW_PROGNAME
```

Pointer to the program name.

Its initial value is the macro **_DEBUG_NEW_PROGNAME** (p. 99). 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("_");`'.

5.1.5.6 PLATFORM_MEM_ALIGNMENT

```
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** (p. 53) uses it to detect misaligned pointer returned by '`new NonPODType[size]`'.

5.1.5.7 stacktrace_print_callback

```
stacktrace_print_callback_t nvwa::stacktrace_print_callback = nullptr
```

Pointer to the callback used to print the stack backtrace in case of a memory problem.

A null value causes the default stack trace printing routine to be used.

Chapter 6

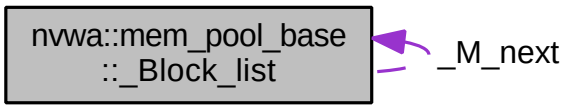
Class Documentation

6.1 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.

6.1.1 Detailed Description

Structure to store the next available memory block.

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

- **mem_pool_base.h**

6.2 nvwa::bool_array::_Element< _Byte_type > Class Template Reference

Class to represent a reference to an array element.

Public Member Functions

- **_Element** (_Byte_type *ptr, **size_type** pos)

  Constructs a reference to an array element.
- bool **operator=** (bool value)

  Assigns a new boolean value to an array element.
- **operator bool** () const

  Reads the boolean value from an array element.

6.2.1 Detailed Description

template<typename _Byte_type>
class nvwa::bool_array::_Element< _Byte_type >

Class to represent a reference to an array element.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 _Element()

```
template<typename _Byte_type >
nvwa::bool_array::_Element< _Byte_type >::  _Element (
            _Byte_type * ptr,
             size_type pos )  [inline]
```

Constructs a reference to an array element.

Parameters

ptr	ptr to the interal boolean data
pos	position of the array element to access

6.2.3 Member Function Documentation

6.2.3.1 operator bool()

```
template<typename _Byte_type >
nvwa::bool_array::_Element< _Byte_type >::operator bool ( ) const  [inline]
```

Reads the boolean value from an array element.

Returns

the boolean value of the accessed array element

6.2.3.2 operator=()

```
template<typename _Byte_type >
bool  nvwa::bool_array::_Element< _Byte_type >::operator= (
             bool value )  [inline]
```

Assigns a new boolean value to an array element.

Parameters

value	the new boolean value

Returns

the assigned boolean value

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

- **bool_array.h**

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

Specializable struct to define the alignment of an object in the **fixed_mem_pool** (p. 69).

```
#include <fixed_mem_pool.h>
```

6.3.1 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** (p. 69).

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

- **fixed_mem_pool.h**

6.4 nvwa::bad_optional_access Class Reference

Class for bad optional access exception.

```
#include <functional.h>
```

Inherits logic_error.

6.4.1 Detailed Description

Class for bad optional access exception.

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

- **functional.h**

6.5 nvwa::basic_mmap_byte_reader< _Tp > Class Template Reference

Class template to allow iteration over all bytes of a mmappable file.

`#include <mmap_byte_reader.h>`

Inherits nvwa::mmap_reader_base.

Classes

- class **iterator**
  Iterator over the bytes.

6.5.1 Detailed Description

template<typename _Tp>
class nvwa::basic_mmap_byte_reader< _Tp >

Class template to allow iteration over all bytes of a mmappable file.

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

- **mmap_byte_reader.h**

6.6 nvwa::basic_mmap_line_reader< _Tp > Class Template Reference

Class template to allow iteration over all lines of a mmappable file.

`#include <mmap_line_reader.h>`

Inherits nvwa::mmap_reader_base.

Classes

- class **iterator**
  Iterator that contains the line content.

Public Types

- enum **strip_type** { **strip_delimiter**, **no_strip_delimiter** }

  Enumeration of whether the delimiter should be stripped.

Public Member Functions

- bool **read** (_Tp &output, size_t &offset)

  Reads content from the mmaped file.

6.6.1 Detailed Description

template<typename _Tp>
class nvwa::basic_mmap_line_reader< _Tp >

Class template to allow iteration over all lines of a mmappable file.

6.6.2 Member Enumeration Documentation

6.6.2.1 strip_type

```
template<typename _Tp >
enum  nvwa::basic_mmap_line_reader::strip_type
```

Enumeration of whether the delimiter should be stripped.

Enumerator

strip_delimiter	The delimiter should be stripped.
no_strip_delimiter	The delimiter should be retained.

6.6.3 Member Function Documentation

6.6.3.1 read()

```
template<typename _Tp >
bool  nvwa::basic_mmap_line_reader< _Tp >::read (
            _Tp & output,
            size_t & offset )
```

Reads content from the mmaped file.

Parameters

`out`	*output*	object to receive the line
`in,out`	*offset*	offset of reading pos on entry; end offset on exit

Returns

`true` if line content is returned; `false` otherwise

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

- **mmap_line_reader.h**

6.7 nvwa::basic_split_view< _StringType, _DelimiterType > Class Template Reference

Class to allow iteration over split items from the input.

`#include <split.h>`

Classes

- class **iterator**
  Iterator over the split items.

Public Member Functions

- constexpr **basic_split_view** (const string_type &src, delimiter_type delimiter) noexcept
  Constructor.
- std::vector< std::basic_string< char_type > > **to_vector** () const
  Converts the view to a string vector.
- std::vector< std::basic_string_view< char_type > > **to_vector_sv** () const
  Converts the view to a string_view vector.

6.7.1 Detailed Description

template<typename _StringType, typename _DelimiterType>
class nvwa::basic_split_view< _StringType, _DelimiterType >

Class to allow iteration over split items from the input.

Parameters

_StringType	either string or string_view (or something similar)
_DelimiterType	the same type as *_StringType* or its character type (other types may cause unpredictable results)

6.7.2 Constructor & Destructor Documentation

6.7.2.1 basic_split_view()

```
template<typename _StringType, typename _DelimiterType>
constexpr  nvwa::basic_split_view< _StringType, _DelimiterType >::  basic_split_view (
             const string_type & src,
             delimiter_type delimiter )  [inline], [explicit], [noexcept]
```

Constructor.

Parameters

src	the source input to be split
delimiter	delimiter used to split *src*; its type should be the same as that of *src*, or its character type

6.7.3 Member Function Documentation

6.7.3.1 to_vector()

```
template<typename _StringType, typename _DelimiterType>
std::vector<std::basic_string<char_type> >  nvwa::basic_split_view< _StringType, _Delimiter↩
Type >::to_vector ( ) const  [inline]
```

Converts the view to a string vector.

6.7.3.2 to_vector_sv()

```
template<typename _StringType, typename _DelimiterType>
std::vector<std::basic_string_view<char_type> >  nvwa::basic_split_view< _StringType, _↩
DelimiterType >::to_vector_sv ( ) const  [inline]
```

Converts the view to a string_view vector.

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

- **split.h**

6.8 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>
```

6.8.1 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.

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

- **fixed_mem_pool.h**

6.9 nvwa::bool_array Class Reference

Class to represent a packed boolean array.

`#include <bool_array.h>`

Classes

- class **_Element**
  Class to represent a reference to an array element.

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** (*p. 38*).*
- **bool_array** (**size_type size**)
  *Constructs a **bool_array** (*p. 38*) with a specific size.*
- **bool_array** (const void *ptr, **size_type size**)
  *Constructs a **bool_array** (*p. 38*) from a given bitmap.*
- ~**bool_array** ()
  *Destroys the **bool_array** (*p. 38*) 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** (p. 38).*
- **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** (p. 38) 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** (p. 38) 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** (p. 38) with a logical OR.*
- void **copy_to_bitmap** (void *dest, **size_type** begin=0, **size_type** end= **npos**)

  *Copies the **bool_array** (p. 38) 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'.

Private Types

- typedef unsigned char **byte**

  Private definition of byte to avoid polluting the global namespace.

Private Member Functions

- **byte get_8bits** (**size_type** offset, **size_type** end) const

  *Retreive contiguous 8 bits from the **bool_array** (p. 38).*

Static Private Attributes

- static **byte _S_bit_count** [256]

  Array that contains pre-calculated values how many 1-bits there are in a given byte.
- static **byte _S_bit_ordinal** [256]

  Array that contains pre-calculated values which the first 1-bit is for a given byte.

6.9.1 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*** (p. 46), ***count*** (p. 42), and ***find*** (p. 44), 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`** (p. 38)).

6.9.2 Member Typedef Documentation

6.9.2.1 byte

```
typedef unsigned char  nvwa::bool_array::byte  [private]
```

Private definition of byte to avoid polluting the global namespace.

6.9.2.2 size_type

```
typedef unsigned long  nvwa::bool_array::size_type
```

Type of array indices.

6.9.3 Constructor & Destructor Documentation

6.9.3.1 bool_array() [1/3]

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

Constructs a **bool_array** (p. 38) with a specific size.

Parameters

size	size of the array

Exceptions

out_of_range	*size* equals 0
bad_alloc	memory is insufficient

6.9.3.2 bool_array() [2/3]

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

Constructs a **bool_array** (p. 38) 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

6.9.3.3 bool_array() [3/3]

```
nvwa::bool_array::bool_array (
            const  bool_array & rhs )
```

Copy-constructor.

Parameters

rhs	the **bool_array** (p. 38) to copy from

Exceptions

bad_alloc	memory is insufficient

6.9.4 Member Function Documentation

6.9.4.1 at()

```
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

6.9.4.2 copy_to_bitmap()

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

Copies the **bool_array** (p. 38) 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

6.9.4.3 count() [1/2]

```
bool_array::size_type nvwa::bool_array::count ( ) const  [noexcept]
```

Counts elements with a `true` value.

Returns

the count of `true` elements

6.9.4.4 count() [2/2]

```
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

6.9.4.5 create()

```
bool nvwa::bool_array::create (
            size_type size )  [noexcept]
```

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.

6.9.4.6 find() [1/2]

```
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** (p. 50) otherwise

6.9.4.7 find() [2/2]

```
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** (p. 50) otherwise

Exceptions

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

6.9.4.8 find_until()

```
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** (p. 50) otherwise

Exceptions

out_of_range	the range [begin, end) is invalid

6.9.4.9 get_8bits()

```
bool_array::byte nvwa::bool_array::get_8bits (
             size_type offset,
             size_type end ) const  [private]
```

Retreive contiguous 8 bits from the **bool_array** (p. 38).

If fewer than 8 bits are available, the extra bits are undefined.

Parameters

offset	beginning position to retrieve the bits
end	end position beyond whose byte no bits will be taken

6.9.4.10 get_num_bytes_from_bits()

```
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

6.9.4.11 initialize()

```
void nvwa::bool_array::initialize (
            bool value )  [noexcept]
```

Initializes all array elements to a specific value optimally.

Parameters

value	the boolean value to assign to all elements

6.9.4.12 merge_and()

```
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** (p. 38) with a logical AND.

Parameters

rhs	another **bool_array** (p. 38) 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** (p. 38)

Exceptions

out_of_range	bad range for the source or the destination

6.9.4.13 merge_or()

```
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** (p. 38) with a logical OR.

Parameters

rhs	another **bool_array** (p. 38) 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** (p. 38)

Exceptions

out_of_range	bad range for the source or the destination

6.9.4.14 operator=()

```
bool_array & nvwa::bool_array::operator= (
            const  bool_array & rhs )
```

Assignment operator.

Parameters

rhs	the **bool_array** (p. 38) to copy from

Exceptions

bad_alloc	memory is insufficient

6.9.4.15 operator[]() [1/2]

```
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

6.9.4.16 operator[]() [2/2]

```
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

6.9.4.17 reset()

```
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

6.9.4.18 set()

```
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

6.9.4.19 size()

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

Gets the size of the **bool_array** (p. 38).

Returns

the number of bits of the **bool_array** (p. 38)

6.9.4.20 swap()

```
void nvwa::bool_array::swap (
            bool_array & rhs ) [noexcept]
```

Exchanges the content of this **bool_array** (p. 38) with another.

Parameters

rhs	another **bool_array** (p. 38) to exchange content with

6.9.5 Member Data Documentation

6.9.5.1 _S_bit_ordinal

```
bool_array::byte nvwa::bool_array::_S_bit_ordinal [static], [private]
```

Array that contains pre-calculated values which the first 1-bit is for a given byte.

The first element indicates an invalid value (there are only 0-bits).

6.9.5.2 npos

```
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**

6.10 nvwa::breadth_first_iteration< _Tree > Class Template Reference

Iteration class for breadth-first traversal.

```
#include <tree.h>
```

6.10.1 Detailed Description

template<typename _Tree>
class nvwa::breadth_first_iteration< _Tree >

Iteration class for breadth-first traversal.

Mutating (adding or removing children) or removing the part of the tree nodes at the current traversal level has undefined behaviour.

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

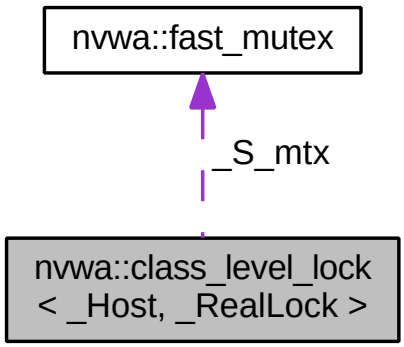
- **tree.h**

6.11 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 >:



Classes

- class **lock**

  Type that provides locking/unlocking semantics.

6.11.1 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** (p. 131) for real usage.

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

- **class_level_lock.h**

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

Partial specialization that makes null locking.

`#include <class_level_lock.h>`

Classes

- class **lock**

  Type that provides locking/unlocking semantics.

6.12.1 Detailed Description

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

Partial specialization that makes null locking.

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

- **class_level_lock.h**

6.13 nvwa::debug_new_counter Class Reference

Counter class for on-exit leakage check.

`#include <debug_new.h>`

Public Member Functions

- **debug_new_counter** ()

  Constructor to increment the count.
- ~**debug_new_counter** ()

  Destructor to decrement the count.

Static Private Attributes

- static int **_S_count** = 0

  Count of source files that use debug_new.

6.13.1 Detailed Description

Counter class for on-exit leakage check.

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

6.13.2 Constructor & Destructor Documentation

6.13.2.1 ~debug_new_counter()

```
nvwa::debug_new_counter::~debug_new_counter ( )
```

Destructor to decrement the count.

When the count is zero, **nvwa::check_leaks** (p. 16) will be called.

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

- **debug_new.h**
- **debug_new.cpp**

6.14 nvwa::debug_new_recorder Class Reference

Recorder class to remember the call context.

```
#include <debug_new.h>
```

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.

Private Member Functions

- void **_M_process** (void *ptr)

  Processes the allocated memory and inserts file/line informatin.

6.14.1 Detailed Description

Recorder class to remember the call context.

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

6.14.2 Constructor & Destructor Documentation

6.14.2.1 debug_new_recorder()

```
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->*.

6.14.3 Member Function Documentation

6.14.3.1 _M_process()

```
void nvwa::debug_new_recorder::_M_process (
            void * usr_ptr )  [private]
```

Processes the allocated memory and inserts file/line informatin.

It will only be done when it can ensure the memory is allocated by one of our operator new variants.

Parameters

usr_ptr	pointer returned by a new-expression

6.14.3.2 operator->*()

```
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**

6.15 nvwa::delete_object Struct Reference

Functor to delete objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

6.15.1 Detailed Description

Functor to delete objects pointed by a container of pointers.

A typical usage might be like:

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

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

- **cont_ptr_utils.h**

6.16 nvwa::depth_first_iteration< _Tree > Class Template Reference

Iteration class for depth-first traversal.

`#include <tree.h>`

6.16.1 Detailed Description

template<typename _Tree>
class nvwa::depth_first_iteration< _Tree >

Iteration class for depth-first traversal.

Mutating (adding or removing children) or removing the part of the tree nodes that have already been traversed has undefined behaviour.

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

- **tree.h**

6.17 nvwa::dereference Struct Reference

Functor to return objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

6.17.1 Detailed Description

Functor to return objects pointed by a container of pointers.

A typical usage might be like:

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

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

- **cont_ptr_utils.h**

6.18 nvwa::dereference_less Struct Reference

Functor to compare objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

6.18.1 Detailed Description

Functor to compare objects pointed by a container of pointers.

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

or

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

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

- **cont_ptr_utils.h**

6.19 nvwa::fast_mutex Class Reference

Class for non-reentrant fast mutexes.

`#include <fast_mutex.h>`

6.19.1 Detailed Description

Class for non-reentrant fast mutexes.

This is the implementation for POSIX threads.

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

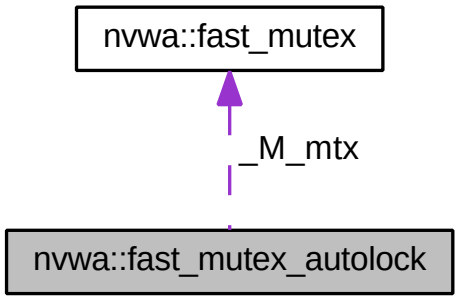
- **fast_mutex.h**

6.20 nvwa::fast_mutex_autolock Class Reference

RAII lock class for **fast_mutex** (p. 56).

`#include <fast_mutex.h>`

Collaboration diagram for nvwa::fast_mutex_autolock:



6.20.1 Detailed Description

RAII lock class for **fast_mutex** (p. 56).

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

- **fast_mutex.h**

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

Class to represent a fixed-capacity queue.

`#include <fc_queue.h>`

Public Member Functions

- **fc_queue** () noexcept(noexcept(allocator_type()))
  Default-constructor that creates an empty queue.
- **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** (**fc_queue** &&rhs) noexcept(noexcept(allocator_type(std::declval< allocator_type &&>())))
  Move-constructor that moves all elements from another queue.
- ~**fc_queue** () noexcept(noexcept(std::declval< _Tp *>() ->~_Tp()))
  Destructor.
- **fc_queue** & **operator=** (const **fc_queue** &rhs)
  Assignment operator that copies all elements from another queue.
- **fc_queue** & **operator=** (**fc_queue** &&rhs) noexcept(noexcept(allocator_type(std::declval< allocator_type &&>())))
  Assignment operator that moves 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.
- template<typename... _Targs>
  void **push** (_Targs &&... args)
  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) noexcept(noexcept(std::swap(std::declval< allocator_type &>(), std::declval< allocator_type &>())))
  Exchanges the elements of two queues.
- allocator_type **get_allocator** () const
  Gets the allocator of the queue.

6.21.1 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).

6.21.2 Constructor & Destructor Documentation

6.21.2.1 fc_queue() [1/4]

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

Default-constructor that creates an empty queue.

It is not very useful, except as the target of an assignment.

Postcondition

The following conditions will hold:

- **empty()** (p. 62)
- **full()** (p. 63)
- **capacity()** (p. 61) == 0
- **size()** (p. 65) == 0

6.21.2.2 fc_queue() [2/4]

```
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()** (p. 62)
- ! **full()** (p. 63)
- **capacity()** (p. 61) == max_size
- **size()** (p. 65) == 0
- **get_allocator()** (p. 63) == alloc

6.21.2.3 fc_queue() [3/4]

```
template<class _Tp , class _Alloc >
nvwa::fc_queue< _Tp, _Alloc >::  fc_queue (
            const  fc_queue< _Tp, _Alloc > & rhs )
```

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*.

6.21.2.4 fc_queue() [4/4]

```
template<class _Tp , class _Alloc >
nvwa::fc_queue< _Tp, _Alloc >::  fc_queue (
             fc_queue< _Tp, _Alloc > && rhs )  [noexcept]
```

Move-constructor that moves all elements from another queue.

Parameters

rhs	the queue to move from

Postcondition

If the allocator does not throw on move, this queue will have the same elements as the original *rhs*.

6.21.2.5 ~fc_queue()

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

Destructor.

It erases all elements and frees memory.

6.21.3 Member Function Documentation

6.21.3.1 back() [1/2]

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

Gets the last element in the queue.

Precondition

the queue is not empty

Returns

reference to the last element

6.21.3.2 back() [2/2]

```
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.

Precondition

the queue is not empty

Returns

const reference to the last element

6.21.3.3 capacity()

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

Gets the maximum number of allowed elements in the queue.

Returns

the maximum number of allowed elements in the queue

6.21.3.4 contains()

```
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

6.21.3.5 empty()

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

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

Returns

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

6.21.3.6 front() [1/2]

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

Gets the first element in the queue.

Precondition

the queue is not empty

Returns

reference to the first element

6.21.3.7 front() [2/2]

```
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.

Precondition

the queue is not empty

Returns

const reference to the first element

6.21.3.8 full()

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

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

Returns

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

6.21.3.9 get_allocator()

```
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

6.21.3.10 operator=() [1/2]

```
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).

6.21.3.11 operator=() [2/2]

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

Assignment operator that moves all elements from another queue.

Parameters

rhs	the queue to move from

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).

6.21.3.12 pop()

```
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()** (p. 65) is decremented by one, and **full()** (p. 63) is `false`.

6.21.3.13 push()

```
template<class _Tp, class _Alloc = std::allocator<_Tp>>
template<typename...  _Targs>
void  nvwa::fc_queue< _Tp, _Alloc >::push (
            _Targs &&...  args )  [inline]
```

Inserts a new element at the end of the queue.

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

Parameters

args	arguments to construct a new element

Precondition

capacity() (p. 61) > 0

Postcondition

size() (p. 65) <= **capacity()** (p. 61) && **back()** (p. 61) == `value`, unless an exception is thrown, in which case this queue is unchanged (strong exception safety is guaranteed).

6.21.3.14 size()

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

Gets the number of existing elements in the queue.

Returns

the number of existing elements in the queue

6.21.3.15 swap()

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

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.

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

- **fc_queue.h**

6.22 nvwa::file_line_reader Class Reference

Class to allow iteration over all lines of a text file.

`#include <file_line_reader.h>`

Classes

- class **iterator**

  Iterator that contains the line content.

Public Types

- enum **strip_type** { **strip_delimiter**, **no_strip_delimiter** }

  Enumeration of whether the delimiter should be stripped.

Public Member Functions

- **file_line_reader** (FILE *stream, char delimiter='\n', **strip_type** strip= **strip_delimiter**)

  Constructor.
- ~**file_line_reader** ()

  Destructor.
- bool **read** (char *&output, size_t &size, size_t &capacity)

  Reads content from the file stream.

6.22.1 Detailed Description

Class to allow iteration over all lines of a text file.

6.22.2 Member Enumeration Documentation

6.22.2.1 strip_type

`enum` **`nvwa::file_line_reader::strip_type`**

Enumeration of whether the delimiter should be stripped.

Enumerator

strip_delimiter	The delimiter should be stripped.
no_strip_delimiter	The delimiter should be retained.

6.22.3 Constructor & Destructor Documentation

6.22.3.1 file_line_reader()

```
nvwa::file_line_reader::file_line_reader (
            FILE * stream,
            char delimiter = '\n',
             strip_type strip =  strip_delimiter )  [explicit]
```

Constructor.

Parameters

stream	the file stream to read from
delimiter	the delimiter between text 'lines' (default to LF)
strip	enumerator about whether to strip the delimiter

6.22.3.2 ~file_line_reader()

```
nvwa::file_line_reader::~file_line_reader ( )
```

Destructor.

6.22.4 Member Function Documentation

6.22.4.1 read()

```
bool nvwa::file_line_reader::read (
            char *& output,
            size_t & size,
            size_t & capacity )
```

Reads content from the file stream.

If necessary, the receiving buffer will be expanded so that it is big enough to contain all the line content.

Parameters

`in,out`	*output*	initial receiving buffer
`out`	*size*	size of the line
`in,out`	*capacity*	capacity of the initial receiving buffer on entering the function; it can be increased when necessary

Returns

`true` if line content is returned; `false` otherwise

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

- **file_line_reader.h**
- **file_line_reader.cpp**

6.23 nvwa::fixed_mem_pool< _Tp > Class Template Reference

Class template to manipulate a fixed-size memory pool.

`#include <fixed_mem_pool.h>`

Classes

- struct **alignment**

  *Specializable struct to define the alignment of an object in the **fixed_mem_pool** (p. 69).*
- struct **block_size**

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

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.

Static Private Attributes

- static void * **_S_mem_pool_ptr** = nullptr

  Pointer to the allocated chunk of memory.
- static void * **_S_first_avail_ptr** = nullptr

  Pointer to the first available memory block.
- static int **_S_alloc_cnt** = 0

  Count of allocations.

6.23.1 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** (p. 69)

6.23.2 Member Function Documentation

6.23.2.1 allocate()

```
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

6.23.2.2 bad_alloc_handler()

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

Bad allocation handler.

Called when there are no memory blocks available in the memory pool. If this function returns `false` (default behaviour if not explicitly specialized), it indicates that it can do nothing and **allocate()** (p. 70) should return null; if this function returns `true`, it indicates that it has freed some memory blocks and **allocate()** (p. 70) should try allocating again.

6.23.2.3 deallocate()

```
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

6.23.2.4 deinitialize()

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

Deinitializes the memory pool.

Returns

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

6.23.2.5 get_alloc_count()

```
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

6.23.2.6 initialize()

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

Initializes the memory pool.

Parameters

size	number of memory blocks to put in the memory pool

Returns

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

6.23.2.7 is_initialized()

```
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

6.23.3 Member Data Documentation

6.23.3.1 _S_alloc_cnt

```
template<class _Tp >
int  nvwa::fixed_mem_pool< _Tp >::_S_alloc_cnt = 0  [static], [private]
```

Count of allocations.

6.23.3.2 _S_first_avail_ptr

```
template<class _Tp >
void * nvwa::fixed_mem_pool< _Tp >::_S_first_avail_ptr = nullptr  [static], [private]
```

Pointer to the first available memory block.

6.23.3.3 _S_mem_pool_ptr

```
template<class _Tp >
void * nvwa::fixed_mem_pool< _Tp >::_S_mem_pool_ptr = nullptr  [static], [private]
```

Pointer to the allocated chunk of memory.

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

- **fixed_mem_pool.h**

6.24 nvwa::in_order_iteration< _Tree > Class Template Reference

Iteration class for in-order traversal.

`#include <tree.h>`

6.24.1 Detailed Description

template<typename _Tree>
class nvwa::in_order_iteration< _Tree >

Iteration class for in-order traversal.

Mutating (adding or removing children) or removing the part of the tree nodes that have already been traversed has undefined behaviour.

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

- **tree.h**

6.25 nvwa::istream_line_reader Class Reference

Class to allow iteration over all lines from an input stream.

`#include <istream_line_reader.h>`

Classes

- class **iterator**

  Iterator that contains the line content.

6.25.1 Detailed Description

Class to allow iteration over all lines from an input stream.

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

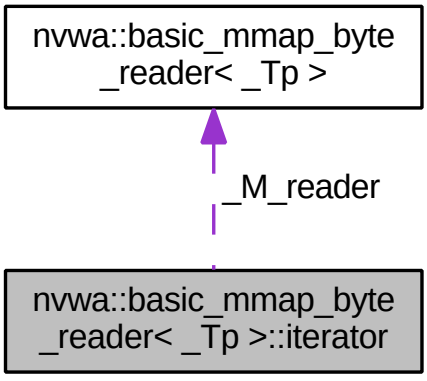
- **istream_line_reader.h**

6.26 nvwa::basic_mmap_byte_reader< _Tp >::iterator Class Reference

Iterator over the bytes.

`#include <mmap_byte_reader.h>`

Collaboration diagram for nvwa::basic_mmap_byte_reader< _Tp >::iterator:



6.26.1 Detailed Description

template<typename _Tp>
class nvwa::basic_mmap_byte_reader< _Tp >::iterator

Iterator over the bytes.

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

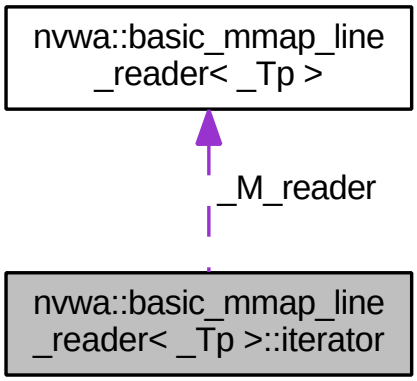
- **mmap_byte_reader.h**

6.27 nvwa::basic_mmap_line_reader< _Tp >::iterator Class Reference

Iterator that contains the line content.

`#include <mmap_line_reader.h>`

Collaboration diagram for nvwa::basic_mmap_line_reader< _Tp >::iterator:



6.27.1 Detailed Description

template<typename _Tp>
class nvwa::basic_mmap_line_reader< _Tp >::iterator

Iterator that contains the line content.

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

- **mmap_line_reader.h**

6.28 nvwa::basic_split_view< _StringType, _DelimiterType >::iterator Class Reference

Iterator over the split items.

`#include <split.h>`

6.28.1 Detailed Description

template<typename _StringType, typename _DelimiterType>
class nvwa::basic_split_view< _StringType, _DelimiterType >::iterator

Iterator over the split items.

The iterator *owns* the content.

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

- **split.h**

6.29 nvwa::istream_line_reader::iterator Class Reference

Iterator that contains the line content.

`#include <istream_line_reader.h>`

6.29.1 Detailed Description

Iterator that contains the line content.

The iterator *owns* the content.

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

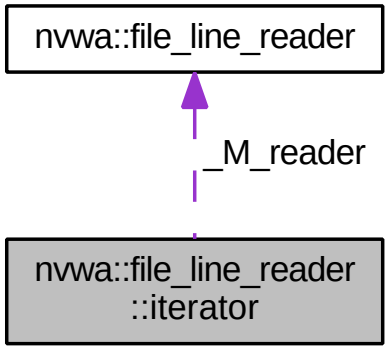
- **istream_line_reader.h**

6.30 nvwa::file_line_reader::iterator Class Reference

Iterator that contains the line content.

`#include <file_line_reader.h>`

Collaboration diagram for nvwa::file_line_reader::iterator:



Public Member Functions

- **iterator** (**file_line_reader** ∗reader)

  Constructs the beginning iterator.
- ∼**iterator** ()

  Destuctor.
- **iterator** (const **iterator** &rhs)

  Copy-constructor.
- **iterator** & **operator=** (const **iterator** &rhs)

  Assignment.
- void **swap** (**iterator** &rhs) noexcept

  Swaps the iterator with another.

6.30.1 Detailed Description

Iterator that contains the line content.

The iterator *owns* the content.

6.30.2 Constructor & Destructor Documentation

6.30.2.1 iterator() [1/2]

```
nvwa::file_line_reader::iterator::iterator (
            file_line_reader ∗ reader ) [explicit]
```

Constructs the beginning iterator.

Parameters

reader	pointer to the **file_line_reader** (p. 66) object

6.30.2.2 ∼iterator()

```
nvwa::file_line_reader::iterator::∼iterator ( )
```

Destuctor.

6.30.2.3 iterator() [2/2]

```
nvwa::file_line_reader::iterator::iterator (
            const  iterator & rhs )
```

Copy-constructor.

The line content will be copied to the newly constructed iterator.

Parameters

rhs	the iterator to copy from

6.30.3 Member Function Documentation

6.30.3.1 operator=()

```
 file_line_reader::iterator & nvwa::file_line_reader::iterator::operator= (
            const  iterator & rhs )
```

Assignment.

The line content will be copied to the newly constructed iterator.

Parameters

rhs	the iterator to copy from

6.30.3.2 swap()

```
void nvwa::file_line_reader::iterator::swap (
             file_line_reader::iterator & rhs )  [noexcept]
```

Swaps the iterator with another.

Parameters

rhs	the iterator to swap with

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

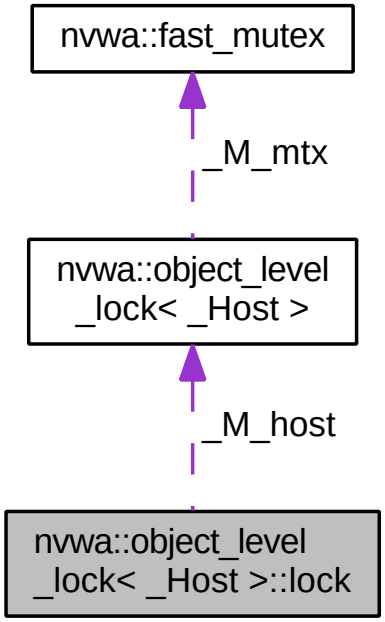
- **file_line_reader.h**
- **file_line_reader.cpp**

6.31 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:



6.31.1 Detailed Description

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

Type that provides locking/unlocking semantics.

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

- **object_level_lock.h**

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

Type that provides locking/unlocking semantics.

`#include <class_level_lock.h>`

6.32.1 Detailed Description

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

Type that provides locking/unlocking semantics.

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

- **class_level_lock.h**

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

Type that provides locking/unlocking semantics.

`#include <class_level_lock.h>`

6.33.1 Detailed Description

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

Type that provides locking/unlocking semantics.

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

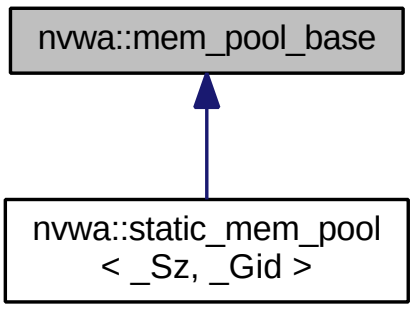
- **class_level_lock.h**

6.34 nvwa::mem_pool_base Class Reference

Base class for memory pools.

`#include <mem_pool_base.h>`

Inheritance diagram for nvwa::mem_pool_base:



Classes

- struct **_Block_list**

  Structure to store the next available memory block.

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.

6.34.1 Detailed Description

Base class for memory pools.

6.34.2 Member Function Documentation

6.34.2.1 alloc_sys()

```
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

6.34.2.2 dealloc_sys()

```
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

6.34.2.3 recycle()

```
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 >** (p. 88).

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

- **mem_pool_base.h**
- **mem_pool_base.cpp**

6.35 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.
- unsigned **line**:31

  Line number of the caller; or 0.
- unsigned **is_array**:1

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

  Pointer to stack trace information.
- unsigned **magic**

  Magic number for error detection.
- char **file** [**_DEBUG_NEW_FILENAME_LEN**]

  File name of the caller.
- void * **addr**

  Address of the caller to new.

6.35.1 Detailed Description

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

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

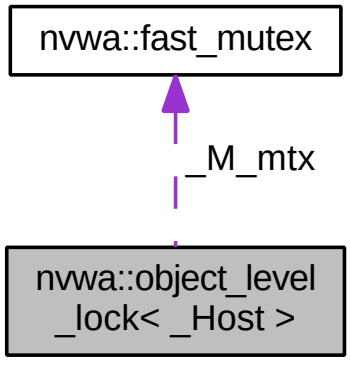
- **debug_new.cpp**

6.36 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 >:



Classes

- class **lock**

  Type that provides locking/unlocking semantics.

6.36.1 Detailed Description

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

Helper class for object-level locking.

This is the multi-threaded implementation.

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

- **object_level_lock.h**

6.37 nvwa::optional< _Tp > Class Template Reference

Class for optional values.

`#include <functional.h>`

Inherits nvwa::optional_base< _Tp, bool >.

6.37.1 Detailed Description

template<typename _Tp>
class nvwa::optional< _Tp >

Class for optional values.

It was initially modelled after the Maybe type in Haskell, but the interface was later changed to be more like (but not fully conformant to) the C++17 std::optional.

Parameters

_Tp	the optional type to store

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

- **functional.h**

6.38 nvwa::output_object< _OutputStrm, _StringType > Struct Template Reference

Functor to output objects pointed by a container of pointers.

`#include <cont_ptr_utils.h>`

6.38.1 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, " "));
```

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

- **cont_ptr_utils.h**

6.39 nvwa::smart_ptr< _Tp, _Policy > Struct Template Reference

Declaration of policy class to generate the smart pointer type.

`#include <tree.h>`

6.39.1 Detailed Description

template<typename _Tp, storage_policy _Policy>
struct nvwa::smart_ptr< _Tp, _Policy >

Declaration of policy class to generate the smart pointer type.

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

- **tree.h**

6.40 nvwa::smart_ptr< _Tp, storage_policy::shared > Struct Template Reference

Partial specialization to get std::shared_ptr.

`#include <tree.h>`

6.40.1 Detailed Description

template<typename _Tp>
struct nvwa::smart_ptr< _Tp, storage_policy::shared >

Partial specialization to get std::shared_ptr.

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

- **tree.h**

6.41 nvwa::smart_ptr< _Tp, storage_policy::unique > Struct Template Reference

Partial specialization to get std::unique_ptr.

`#include <tree.h>`

6.41.1 Detailed Description

template<typename _Tp>
struct nvwa::smart_ptr< _Tp, storage_policy::unique >

Partial specialization to get std::unique_ptr.

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

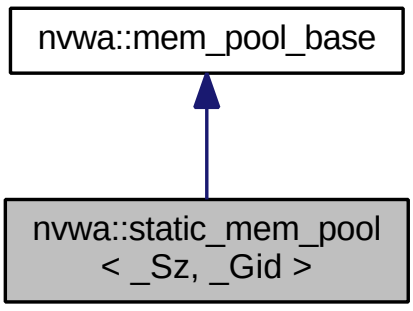
- **tree.h**

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

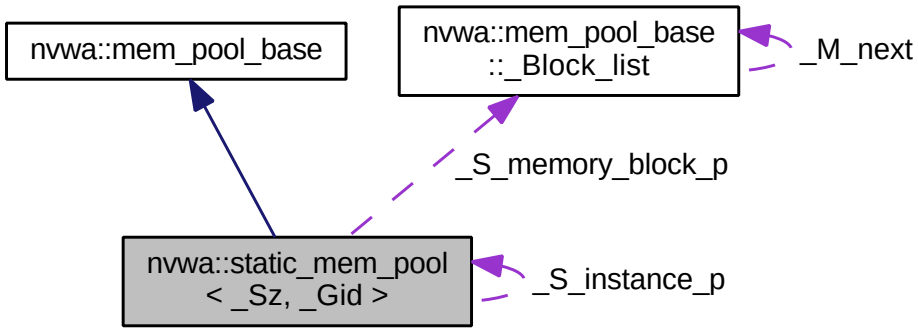
Singleton class template to manage the allocation/deallocation of memory blocks of one specific size.

`#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** () override

  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.

6.42.1 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** (p. 85)
_Gid	group ID of a **static_mem_pool** (p. 85): if it is negative, simultaneous accesses to this **static_mem_pool** (p. 85) will be protected from each other; otherwise no protection is given

6.42.2 Member Function Documentation

6.42.2.1 allocate()

```
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

6.42.2.2 deallocate()

```
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

6.42.2.3 instance()

```
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 (p. 87)

6.42.2.4 instance_known()

```
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

6.42.2.5 recycle()

```
template<size_t _Sz, int _Gid>
void nvwa::static_mem_pool< _Sz, _Gid >::recycle ( ) [override], [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** (p. 81).

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

- **static_mem_pool.h**

6.43 nvwa::static_mem_pool_set Class Reference

Singleton class to maintain a set of existing instantiations of **static_mem_pool** (p. 85).

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

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** (p. 88).*

Static Public Member Functions

- static **static_mem_pool_set** & **instance** ()

  *Gets the singleton instance of **nvwa::static_mem_pool_set** (p. 88).*

6.43.1 Detailed Description

Singleton class to maintain a set of existing instantiations of **static_mem_pool** (p. 85).

6.43.2 Member Function Documentation

6.43.2.1 add()

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

Adds a new memory pool to **nvwa::static_mem_pool_set** (p. 88).

Parameters

memory_pool_p	pointer to the memory pool to add

6.43.2.2 instance()

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

Gets the singleton instance of **nvwa::static_mem_pool_set** (p. 88).

The instance will be created on the first invocation.

Returns

reference to the instance of **nvwa::static_mem_pool_set** (p. 88)

6.43.2.3 recycle()

```
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** (p. 88) during its execution.

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

- **static_mem_pool.h**
- **static_mem_pool.cpp**

6.44 nvwa::tree< _Tp, _Policy > Class Template Reference

Basic tree (node) class template that owns all its children.

`#include <tree.h>`

6.44.1 Detailed Description

template<typename _Tp, storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY>
class nvwa::tree< _Tp, _Policy >

Basic tree (node) class template that owns all its children.

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

- **tree.h**

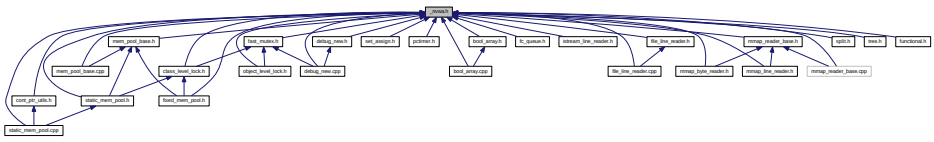
Chapter 7

File Documentation

7.1 _nvwa.h File Reference

Common definitions for preprocessing.

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



Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.1.1 Detailed Description

Common definitions for preprocessing.

Date

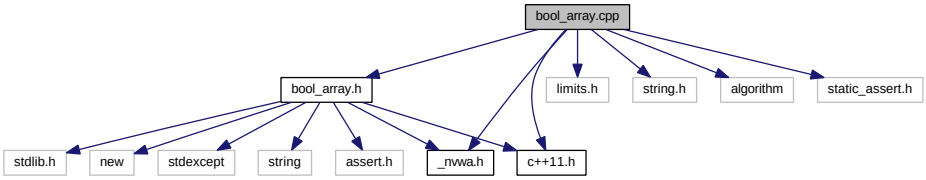
2015-10-28

7.2 bool_array.cpp File Reference

Code for class bool_array (packed boolean array).

```
#include "bool_array.h"
#include <limits.h>
#include <string.h>
#include <algorithm>
#include "_nvwa.h"
#include "c++11.h"
#include "static_assert.h"
```

Include dependency graph for bool_array.cpp:



Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.2.1 Detailed Description

Code for class bool_array (packed boolean array).

Date

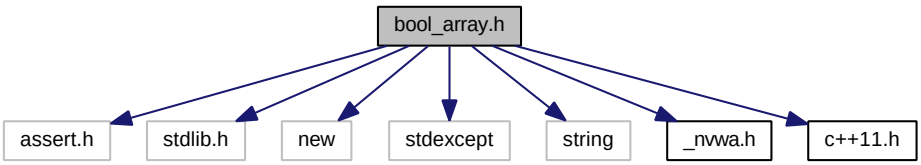
2017-09-09

7.3 bool_array.h File Reference

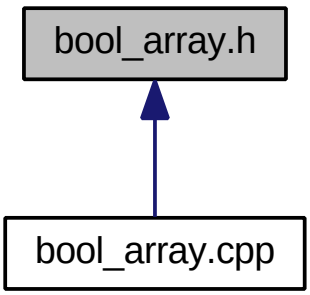
Header file for class bool_array (packed boolean array).

```
#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:



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.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Functions

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

  Exchanges the content of two bool_arrays.

7.3.1 Detailed Description

Header file for class bool_array (packed boolean array).

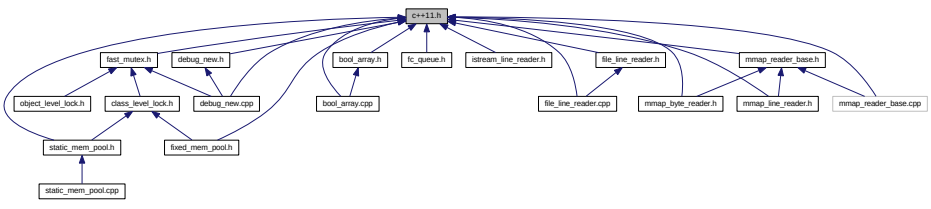
Date

2014-11-29

7.4 c++11.h File Reference

C++11 feature detection macros and workarounds.

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



7.4.1 Detailed Description

C++11 feature detection macros and workarounds.

Date

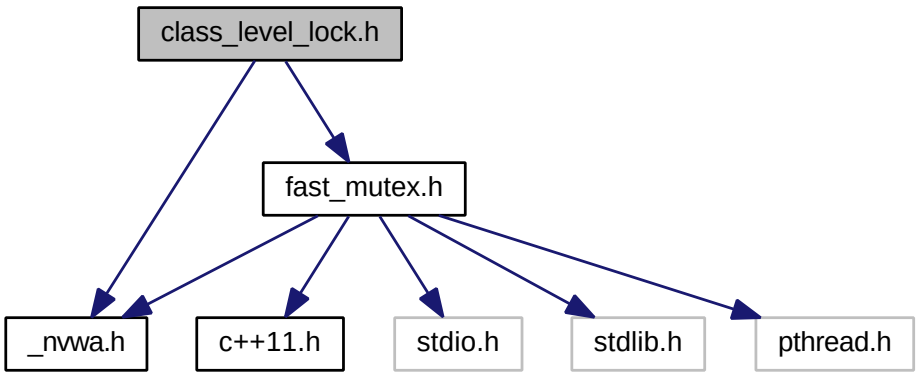
2017-04-03

7.5 class_level_lock.h File Reference

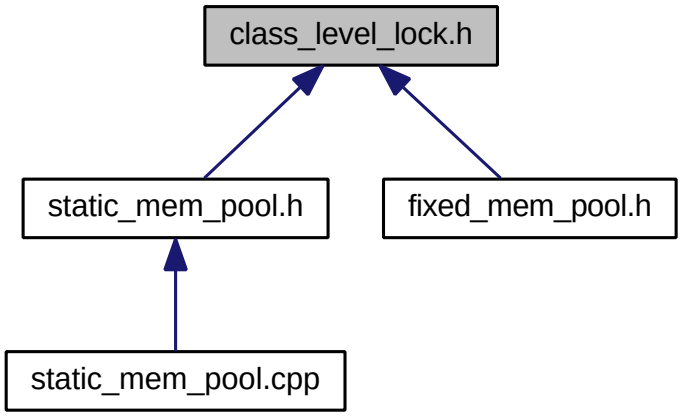
In essence Loki ClassLevelLockable re-engineered to use a fast_mutex class.

```
#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:



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.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.5.1 Detailed Description

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

Date

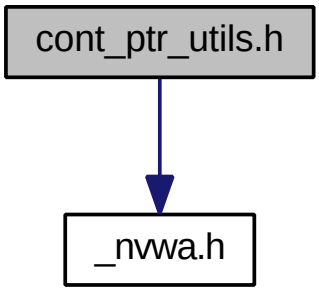
2017-11-25

7.6 cont_ptr_utils.h File Reference

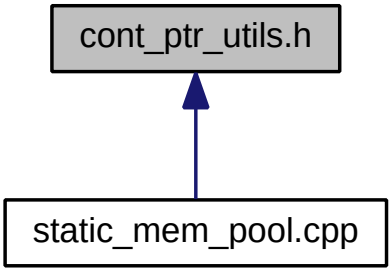
Utility functors for containers of pointers (adapted from Scott Meyers' *Effective STL*).

```
#include "_nvwa.h"
```

Include dependency graph for cont_ptr_utils.h:



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



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.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.6.1 Detailed Description

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

Date

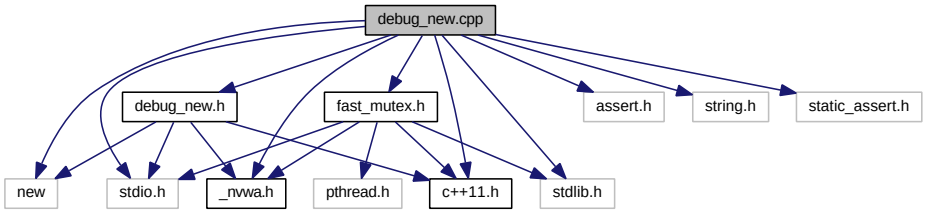
2013-10-06

7.7 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:



Classes

- struct **nvwa::new_ptr_list_t**

  Structure to store the position information where new occurs.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Macros

- #define **_DEBUG_NEW_REDEFINE_NEW** 0

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

  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 **ALIGN**(s) (((s) + **_DEBUG_NEW_ALIGNMENT** - 1) & ~(**_DEBUG_NEW_ALIGNMENT** - 1))

  Gets the aligned value of memory block size.
- #define **_DEBUG_NEW_REMEMBER_STACK_TRACE** 0

  Macro to indicate whether stack traces of allocations should be included in NVWA allocation information and be printed while leaks are reported.

Functions

- static bool **nvwa::print_position_from_addr** (const void *addr)

  Tries printing the position information from an instruction address.
- static void **nvwa::print_position** (const void *ptr, int line)

  Prints the position information of a memory operation point.
- static void **nvwa::print_stacktrace** (void **stacktrace)

  Prints the stack backtrace.
- static bool **nvwa::is_leak_whitelisted** (new_ptr_list_t *ptr)

  Checks whether a leak should be ignored.
- static void * **nvwa::alloc_mem** (size_t size, const char *file, int line, bool is_array)

  Allocates memory and initializes control data.
- static void **nvwa::free_pointer** (void *usr_ptr, void *addr, bool is_array)

  Frees memory and adjusts pointers.
- 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.
- static const unsigned **nvwa::DEBUG_NEW_MAGIC** = 0x4442474E

  Definition of the constant magic number used for error detection.
- static const int **nvwa::ALIGNED_LIST_ITEM_SIZE** = **ALIGN**(sizeof(new_ptr_list_t))

  The extra memory allocated by `operator new`.
- static new_ptr_list_t **nvwa::new_ptr_list**

  List of all new'd pointers.
- static fast_mutex **nvwa::new_ptr_lock**

  The mutex guard to protect simultaneous access to the pointer list.
- static fast_mutex **nvwa::new_output_lock**

  *The mutex guard to protect simultaneous output to **new_output_fp** (p. 29).*
- static size_t **nvwa::total_mem_alloc** = 0

  Total memory allocated in bytes.

7.7.1 Detailed Description

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

Date

2017-09-09

7.7.2 Macro Definition Documentation

7.7.2.1 _DEBUG_NEW_ALIGNMENT

```
#define _DEBUG_NEW_ALIGNMENT 16
```

The alignment requirement of allocated memory blocks.

It must be a power of two.

7.7.2.2 _DEBUG_NEW_CALLER_ADDRESS

```
#define _DEBUG_NEW_CALLER_ADDRESS __builtin_return_address(0)
```

The expression to return the caller address.

nvwa::print_position (p. 24) will later on use this address to print the position information of memory operation points.

7.7.2.3 _DEBUG_NEW_ERROR_ACTION

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

The action to take when an error occurs.

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

7.7.2.4 _DEBUG_NEW_FILENAME_LEN

```
#define _DEBUG_NEW_FILENAME_LEN 44
```

The length of file name stored if greater than zero.

If it is zero, only a const char pointer will be stored. Currently the default 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 (w/o stack backtrace).

7.7.2.5 _DEBUG_NEW_PROGNAME

```
#define _DEBUG_NEW_PROGNAME _NULLPTR
```

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

It is better to assign the full program path to **nvwa::new_progname** (p. 29) 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.

7.7.2.6 _DEBUG_NEW_REDEFINE_NEW

```
#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
```

Here it is defined to `0` to disable the redefinition of `new`.

7.7.2.7 _DEBUG_NEW_REMEMBER_STACK_TRACE

```
#define _DEBUG_NEW_REMEMBER_STACK_TRACE 0
```

Macro to indicate whether stack traces of allocations should be included in NVWA allocation information and be printed while leaks are reported.

Useful to pinpoint leaks caused by strdup and other custom allocation functions. It is also very helpful in filtering out false positives caused by internal STL/C runtime operations. It is off by default because it is quite memory heavy. Set it to `1` to make all allocations have stack trace attached, or set to `2` to make only allocations that lack calling source code information (file and line) have the stack trace attached.

7.7.2.8 _DEBUG_NEW_STD_OPER_NEW

```
#define _DEBUG_NEW_STD_OPER_NEW 1
```

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

It is on by default now, but the user may set it to `0` to revert to the old behaviour.

7.7.2.9 _DEBUG_NEW_TAILCHECK

```
#define _DEBUG_NEW_TAILCHECK 0
```

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

Define it to a positive integer as the number of padding bytes at the end of a memory block for checking.

7.7.2.10 _DEBUG_NEW_USE_ADDR2LINE

```
#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.

7.7.3 Function Documentation

7.7.3.1 operator delete() [1/3]

```
void operator delete (
            void * ptr )  [noexcept]
```

Deallocates memory.

Parameters

ptr	pointer to the previously allocated memory

7.7.3.2 operator delete() [2/3]

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

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

7.7.3.3 operator delete() [3/3]

```
void operator delete (
            void * ptr,
            const std::nothrow_t &  )  [noexcept]
```

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

7.7.3.4 operator delete[]() [1/3]

```
void operator delete[ ] (
            void * ptr )  [noexcept]
```

Deallocates array memory.

Parameters

ptr	pointer to the previously allocated memory

7.7.3.5 operator delete[]() [2/3]

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

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

7.7.3.6 operator delete[]() [3/3]

```
void operator delete[] (
            void * ptr,
            const std::nothrow_t &  )  [noexcept]
```

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

7.7.3.7 operator new() [1/3]

```
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** (p. 100) is 0)

Exceptions

bad_alloc	memory is insufficient (**_DEBUG_NEW_STD_OPER_NEW** (p. 100) is 1)

7.7.3.8 operator new() [2/3]

```
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** (p. 100) is 0)

Exceptions

bad_alloc	memory is insufficient (**_DEBUG_NEW_STD_OPER_NEW** (p. 100) is 1)

7.7.3.9 operator new() [3/3]

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

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

7.7.3.10 operator new[]() [1/3]

```
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** (p. 100) is 0)

Exceptions

bad_alloc	memory is insufficient (**_DEBUG_NEW_STD_OPER_NEW** (p. 100) is 1)

7.7.3.11 operator new[]() [2/3]

```
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** (p. 100) is 0)

Exceptions

bad_alloc	memory is insufficient (**_DEBUG_NEW_STD_OPER_NEW** (p. 100) is 1)

7.7.3.12 operator new[]() [3/3]

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

Allocates array memory with no-throw guarantee.

Parameters

size	size of the required memory block

Returns

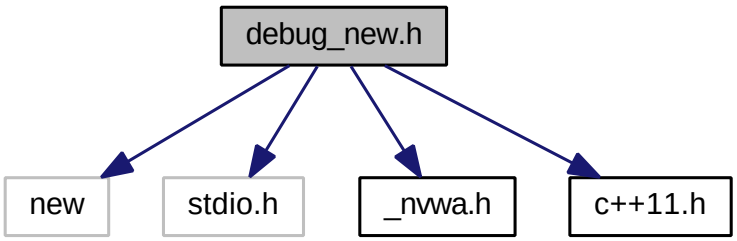
pointer to the memory allocated; or null if memory is insufficient

7.8 debug_new.h File Reference

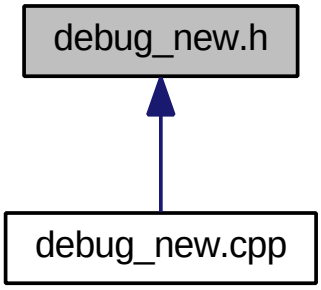
Header file for checking leaks caused by unmatched new/delete.

```
#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:



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.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Macros

- #define **_DEBUG_NEW_TYPE** 1

  *Macro to indicate which variant of **DEBUG_NEW** (p. 108) is wanted.*
- #define **DEBUG_NEW** NVWA::debug_new_recorder(__FILE__, __LINE__) ->* new

  Macro to catch file/line information on allocation.

Typedefs

- typedef void(∗ **nvwa::stacktrace_print_callback_t**) (FILE ∗fp, void ∗∗stacktrace)

  Callback type for stack trace printing.
- typedef bool(∗ **nvwa::leak_whitelist_callback_t**) (char const ∗file, int line, void ∗addr, void ∗∗stacktrace)

  Callback type for the leak whitelist function.

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** = true

  *Flag to control whether **nvwa::check_leaks*** *(p. 16)* *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.
- stacktrace_print_callback_t **nvwa::stacktrace_print_callback** = nullptr

  Pointer to the callback used to print the stack backtrace in case of a memory problem.
- leak_whitelist_callback_t **nvwa::leak_whitelist_callback** = nullptr

  Pointer to the callback used to filter out false positives from leak reports.
- static debug_new_counter **nvwa::__debug_new_count**

  *Counting object for each file including **debug_new.h*** *(p. 106)**.*

7.8.1 Detailed Description

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

Date

2017-09-01

7.8.2 Macro Definition Documentation

7.8.2.1 _DEBUG_NEW_TYPE

```
#define _DEBUG_NEW_TYPE 1
```

Macro to indicate which variant of **DEBUG_NEW** (p. 108) 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** (p. 107) 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`.

7.8.2.2 DEBUG_NEW

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

Macro to catch file/line information on allocation.

If **_DEBUG_NEW_REDEFINE_NEW** (p. 99) is `0`, one can use this macro directly; otherwise `new` will be defined to it, and one must use `new` instead.

7.8.3 Function Documentation

7.8.3.1 operator delete()

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

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

7.8.3.2 operator delete[]()

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

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

7.8.3.3 operator new()

```
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** (p. 100) is 0)

Exceptions

bad_alloc	memory is insufficient (**_DEBUG_NEW_STD_OPER_NEW** (p. 100) is 1)

7.8.3.4 operator new[]()

```
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** (p. 100) is 0)

Exceptions

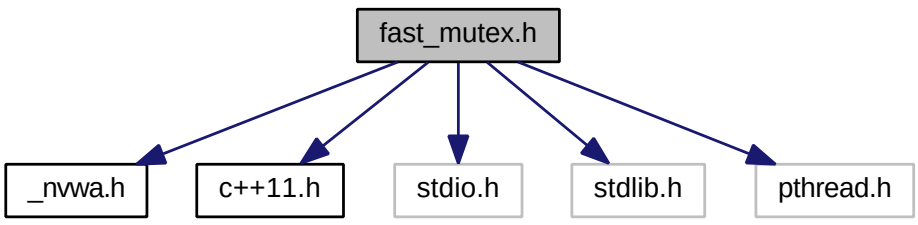
bad_alloc	memory is insufficient (**_DEBUG_NEW_STD_OPER_NEW** (p. 100) is 1)

7.9 fast_mutex.h File Reference

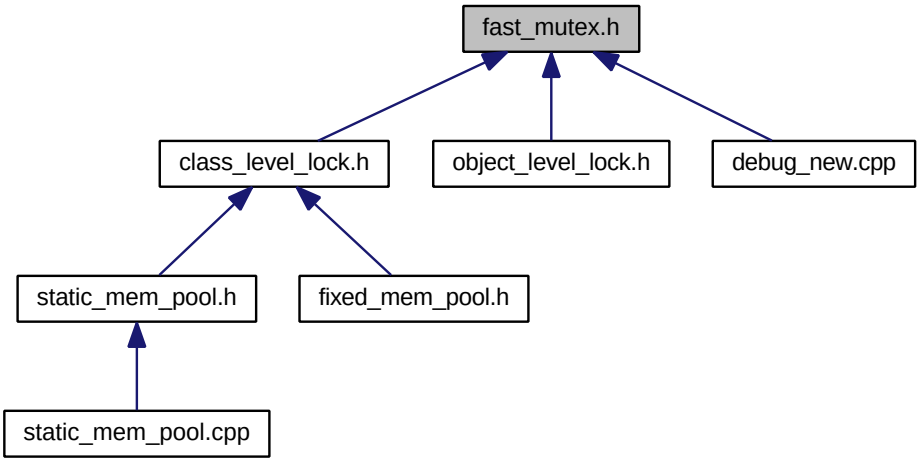
A fast mutex implementation for POSIX, Win32, and modern C++.

```
#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:



Classes

- class **nvwa::fast_mutex**

  Class for non-reentrant fast mutexes.
- class **nvwa::fast_mutex_autolock**

  *RAII lock class for **fast_mutex** (*p. 56*).*

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Macros

- #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.

7.9.1 Detailed Description

A fast mutex implementation for POSIX, Win32, and modern C++.

Date

2017-09-01

7.9.2 Macro Definition Documentation

7.9.2.1 __VOLATILE

```
#define __VOLATILE volatile
```

Macro alias to 'volatile' semantics.

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

7.9.2.2 _FAST_MUTEX_ASSERT

```
#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).

7.9.2.3 _FAST_MUTEX_CHECK_INITIALIZATION

```
#define _FAST_MUTEX_CHECK_INITIALIZATION 1
```

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

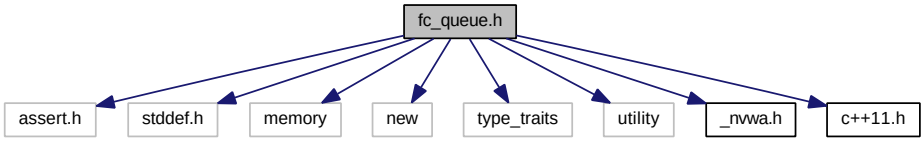
Defining it to a non-zero value will enable the check, so that the construction/destruction of a static object using a static fast_mutex not yet constructed or already destroyed will work (with lock/unlock operations ignored). Defining it to zero will disable to check.

7.10 fc_queue.h File Reference

Definition of a fixed-capacity queue.

```
#include <assert.h>
#include <stddef.h>
#include <memory>
#include <new>
#include <type_traits>
#include <utility>
#include "_nvwa.h"
#include "c++11.h"
```

Include dependency graph for fc_queue.h:



Classes

- class **nvwa::fc_queue**< _Tp, _Alloc >

  Class to represent a fixed-capacity queue.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Functions

- template<class _Tp , class _Alloc >
  void **nvwa::swap** (fc_queue< _Tp, _Alloc > &lhs, fc_queue< _Tp, _Alloc > &rhs) noexcept(noexcept(lhs.↩
  swap(rhs)))

  Exchanges the elements of two queues.

7.10.1 Detailed Description

Definition of a fixed-capacity queue.

Date

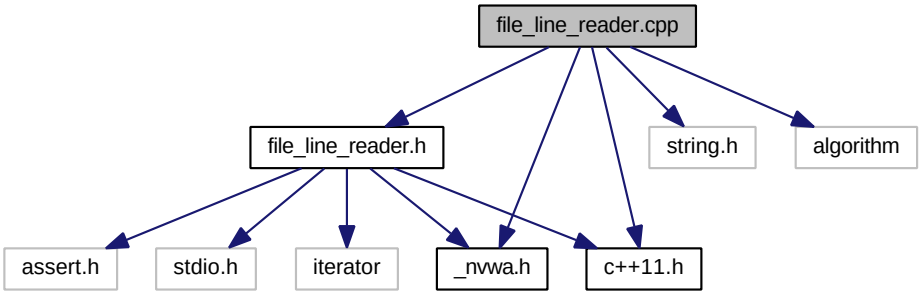
2017-06-10

7.11 file_line_reader.cpp File Reference

Code for file_line_reader, an easy-to-use line-based file reader.

```
#include "file_line_reader.h"
#include <string.h>
#include "_nvwa.h"
#include "c++11.h"
#include <algorithm>
```

Include dependency graph for file_line_reader.cpp:



Namespaces

- **nvwa**

  Namespace of the nvwa project.

Variables

- const size_t **nvwa::BUFFER_SIZE** = 256

  Size of buffer.

7.11.1 Detailed Description

Code for file_line_reader, an easy-to-use line-based file reader.

Date

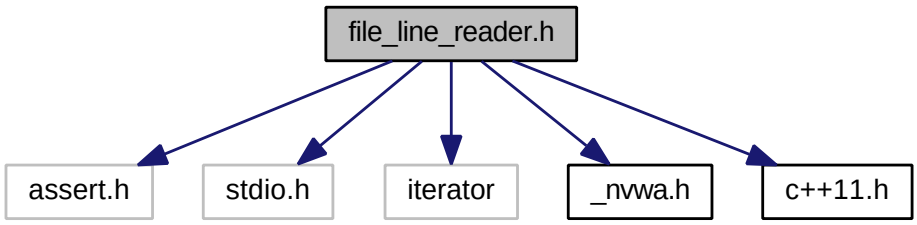
2017-09-09

7.12 file_line_reader.h File Reference

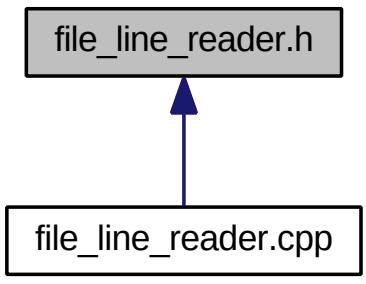
Header file for file_line_reader, an easy-to-use line-based file reader.

```
#include <assert.h>
#include <stdio.h>
#include <iterator>
#include "_nvwa.h"
#include "c++11.h"
```

Include dependency graph for file_line_reader.h:



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



Classes

- class **nvwa::file_line_reader**

  Class to allow iteration over all lines of a text file.
- class **nvwa::file_line_reader::iterator**

  Iterator that contains the line content.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.12.1 Detailed Description

Header file for file_line_reader, an easy-to-use line-based file reader.

Date

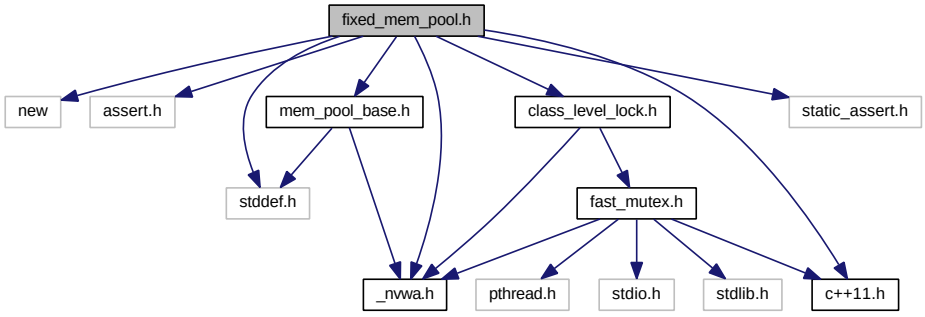
2017-03-23

7.13 fixed_mem_pool.h File Reference

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

```
#include <new>
#include <assert.h>
#include <stddef.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:



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** (p. 69).*
- struct **nvwa::fixed_mem_pool< _Tp >::block_size**

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

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Macros

- #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.

7.13.1 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** (p. 116),
  - **DECLARE_FIXED_MEM_POOL__NOTHROW** (p. 117), or
  - **DECLARE_FIXED_MEM_POOL__THROW_NOCHECK** (p. 118)

  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

2014-11-29

7.13.2 Macro Definition Documentation

7.13.2.1 DECLARE_FIXED_MEM_POOL

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

Value:

```
public: \
    static void* operator new(size_t size) \
    { \
        assert(size == sizeof(_Cls)); \
        if (void* ptr = NVWA::fixed_mem_pool<_Cls>::allocate()) \
            return ptr; \
        else \
            throw std::bad_alloc(); \
    } \
    static void  operator delete(void* ptr) \
    { \
        if (ptr != _NULLPTR) \
            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 (p. 118), which, too, defines an **operator new** that will never return null, but requires more discipline on the programmer's side.

7.13.2.2 DECLARE_FIXED_MEM_POOL__NOTHROW

```
#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 != _NULLPTR) \
            NVWA::fixed_mem_pool<_Cls>::deallocate(ptr); \
    }
```

Declares the nothrow allocation and deallocation functions.

Parameters

_Cls	class to use the fixed_mem_pool

7.13.2.3 DECLARE_FIXED_MEM_POOL__THROW_NOCHECK

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

Value:

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

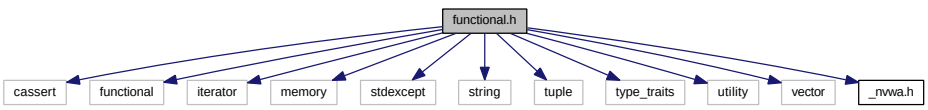
_Cls	class to use the fixed_mem_pool

7.14 functional.h File Reference

Utility templates for functional programming style.

```
#include <cassert>
#include <functional>
#include <iterator>
#include <memory>
#include <stdexcept>
#include <string>
#include <tuple>
#include <type_traits>
#include <utility>
#include <vector>
#include "_nvwa.h"
```

Include dependency graph for functional.h:



Classes

- class **nvwa::bad_optional_access**

  Class for bad optional access exception.
- class **nvwa::optional**< _**Tp** >

  Class for optional values.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Functions

- template<typename _Fn >
  auto **nvwa::lift_optional** (_Fn &&f)

  Lifts a function so that it takes optionals and returns an optional.
- template<typename _Fn , typename... _Opt>
  constexpr auto **nvwa::apply** (_Fn &&f, _Opt &&... args) -> decltype(has_value(args...), optional< std↩
  ::decay_t< decltype(f(std::forward< _Opt >(args).value()...))>>())

  Applies a function to the values of optionals if they are all valid.
- template<typename _Fn , class _Tuple >
  constexpr auto **nvwa::apply** (_Fn &&f, _Tuple &&t) -> decltype(detail::tuple_apply_impl(std::forward< _Fn
  >(f), std::forward< _Tuple >(t), std::make_index_sequence< std::tuple_size< std::decay_t< _Tuple >>↩
  ::value >()))

  Applies the function with all elements of the tuple as arguments.
- template<typename _Fn , typename _T1 , typename _T2 >
  constexpr auto **nvwa::fmap** (_Fn &&f, const std::pair< _T1, _T2 > &args)

  Applies a function to both elements of a pair, and makes the results a pair.
- template<typename _Fn , typename... _Targs>
  constexpr auto **nvwa::fmap** (_Fn &&f, const std::tuple< _Targs... > &args)

  Applies a function to all elements of a tuple, and makes the results a tuple.
- template<template< typename, typename > class _OutCont = std::vector, template< typename > class _Alloc = std::allocator, type-
  name _Fn , class _Rng >
  constexpr auto **nvwa::fmap** (_Fn &&f, _Rng &&inputs) -> decltype(detail::adl_begin(inputs), detail↩
  ::adl_end(inputs), _OutCont< std::decay_t< decltype(f(∗detail::adl_begin(inputs)))>, _Alloc< std::decay_t<
  decltype(f(∗detail::adl_begin(inputs)))>>>())

  Applies a function to each element in the input range.
- template<typename _Rs , typename _Fn , typename... _Targs>
  constexpr auto **nvwa::reduce** (_Fn &&f, const std::tuple< _Targs... > &args, _Rs &&value)

  Applies a function cumulatively to all elements of a tuple.
- template<typename _Fn , class _Rng >
  constexpr auto **nvwa::reduce** (_Fn &&f, _Rng &&inputs)

  Applies a function cumulatively to elements in the input range.
- template<typename _Rs , typename _Fn , typename _Iter >
  constexpr _Rs && **nvwa::reduce** (_Fn &&f, _Rs &&value, _Iter begin, _Iter end)

  Applies a function cumulatively to a range.
- template<typename _Rs , typename _Fn , class _Rng >
  constexpr auto **nvwa::reduce** (_Fn &&f, _Rng &&inputs, _Rs &&initval) -> decltype(f(initval, ∗detail::adl_↩
  begin(inputs)))

  Applies a function cumulatively to elements in the input range.

- template<typename _T1 , typename _T2 , typename _Fn >
  constexpr auto **nvwa::wrap_args_as_pair** (_Fn &&f)

  Makes a two-argument function accept a pair instead.
- template<typename _Tuple , typename _Fn >
  constexpr auto **nvwa::wrap_args_as_tuple** (_Fn &&f)

  Makes a function accept a tuple as its arguments.
- template<typename _Tp >
  constexpr _Tp **nvwa::pipeline** (_Tp &&data)

  Returns the data intact to terminate the recursion.
- template<typename _Tp , typename _Fn , typename... _Fargs>
  decltype(auto) constexpr **nvwa::pipeline** (_Tp &&data, _Fn &&f, _Fargs &&... args)

  Applies the functions in the arguments to the data consecutively.
- auto **nvwa::compose** ()

  Constructs a function (object) that composes the passed functions.
- template<typename _Fn >
  auto **nvwa::compose** (_Fn f)

  Constructs a function (object) that composes the passed functions.
- template<typename _Fn , typename... _Fargs>
  auto **nvwa::compose** (_Fn f, _Fargs... args)

  Constructs a function (object) that composes the passed functions.
- template<typename _Rs , typename _Tp >
  std::function< _Rs(_Tp)> **nvwa::fix_simple** (std::function< _Rs(std::function< _Rs(_Tp)>, _Tp)> f)

  Generates the fixed point using the simple fixed-point combinator.
- template<typename _Rs , typename _Tp >
  std::function< _Rs(_Tp)> **nvwa::fix_simple** (std::function< std::function< _Rs(_Tp)>(std::function< _Rs(_Tp)>)> f)

  Generates the fixed point using the simple fixed-point combinator.
- template<typename _Rs , typename _Tp >
  std::function< _Rs(_Tp)> **nvwa::fix_curry** (std::function< std::function< _Rs(_Tp)>(std::function< _Rs(_Tp)>)> f)

  Generates the fixed point using the Curry-style fixed-point combinator.
- template<typename _Rs , typename... _Targs>
  auto **nvwa::make_curry** (std::function< _Rs(_Targs...)> f)

  Makes a curried function.
- template<typename _Rs , typename... _Targs>
  auto **nvwa::make_curry** (_Rs(*f)(_Targs...))

  Makes a curried function.
- template<typename _FnType , typename _Fn >
  auto **nvwa::make_curry** (_Fn &&f)

  Makes a curried function.

7.14.1 Detailed Description

Utility templates for functional programming style.

Using this file requires a C++14-compliant compiler.

Date

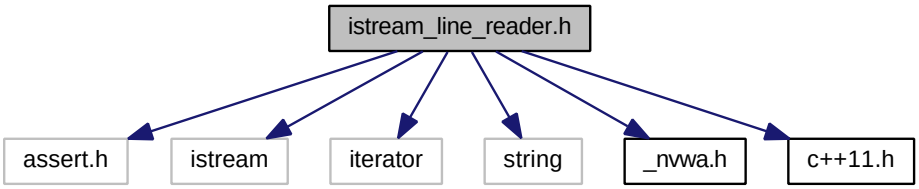
2018-01-02

7.15 istream_line_reader.h File Reference

Header file for istream_line_reader, an easy-to-use line-based istream reader.

```
#include <assert.h>
#include <istream>
#include <iterator>
#include <string>
#include "_nvwa.h"
#include "c++11.h"
```

Include dependency graph for istream_line_reader.h:



Classes

- class **nvwa::istream_line_reader**

  Class to allow iteration over all lines from an input stream.
- class **nvwa::istream_line_reader::iterator**

  Iterator that contains the line content.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.15.1 Detailed Description

Header file for istream_line_reader, an easy-to-use line-based istream reader.

This class allows using istreams in a Pythonic way (if your compiler supports C++11 or later), e.g.:

```
for (auto& line : nvwa::istream_line_reader(std::cin)) {
    // Process line
}
```

This code can be used without C++11, but using it would be less convenient then.

It was first published in a `blog`, and has since been modified to satisfy the `InputIterator` concept, along with other minor changes.

Date

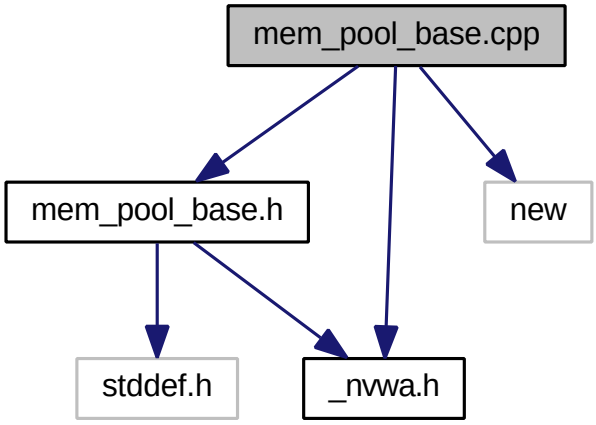
2017-03-23

7.16 mem_pool_base.cpp File Reference

Implementation for the memory pool base.

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

Include dependency graph for mem_pool_base.cpp:



Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.16.1 Detailed Description

Implementation for the memory pool base.

Date

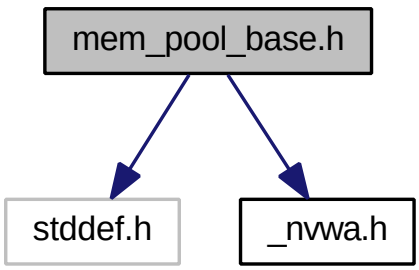
2017-09-09

7.17 mem_pool_base.h File Reference

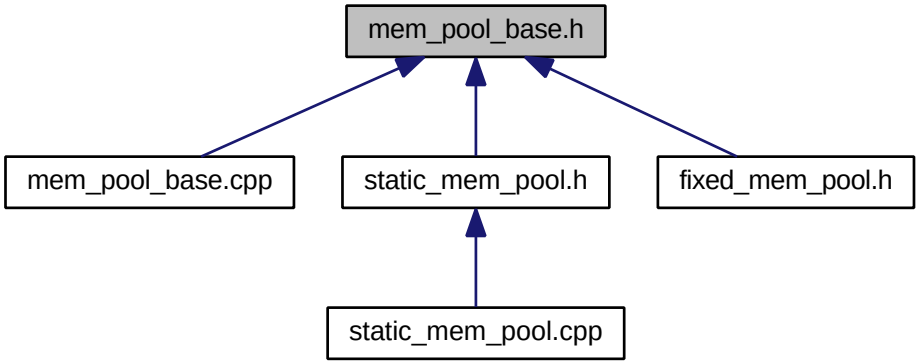
Header file for the memory pool base.

```
#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:



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.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.17.1 Detailed Description

Header file for the memory pool base.

Date

2013-10-06

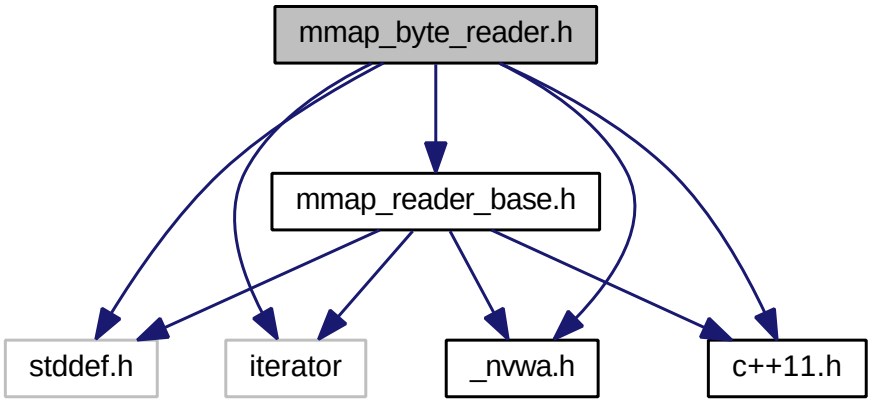
7.18 mmap_byte_reader.h File Reference

Header file for mmap_byte_reader, an easy-to-use byte-based file reader.

```
#include <stddef.h>
#include <iterator>
#include "_nvwa.h"
#include "c++11.h"
```

```
#include "mmap_reader_base.h"
```

Include dependency graph for mmap_byte_reader.h:



Classes

- class **nvwa::basic_mmap_byte_reader**< **_Tp** >

  Class template to allow iteration over all bytes of a mmappable file.
- class **nvwa::basic_mmap_byte_reader**< **_Tp** >**::iterator**

  Iterator over the bytes.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.18.1 Detailed Description

Header file for mmap_byte_reader, an easy-to-use byte-based file reader.

It is implemented with memory-mapped file APIs.

Date

2017-09-12

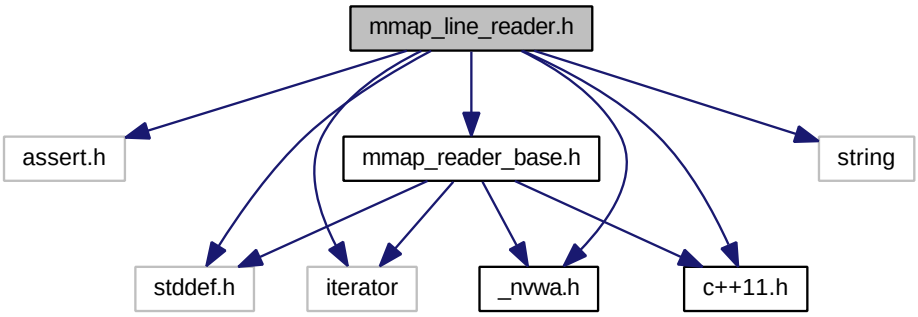
7.19 mmap_line_reader.h File Reference

Header file for mmap_line_reader and mmap_line_reader_sv, easy-to-use line-based file readers.

```
#include <assert.h>
#include <stddef.h>
#include <iterator>
#include "_nvwa.h"
#include "c++11.h"
#include "mmap_reader_base.h"
```

```
#include <string>
```

Include dependency graph for mmap_line_reader.h:



Classes

- class **nvwa::basic_mmap_line_reader**< _Tp >

  Class template to allow iteration over all lines of a mmappable file.
- class **nvwa::basic_mmap_line_reader**< _Tp >**::iterator**

  Iterator that contains the line content.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.19.1 Detailed Description

Header file for mmap_line_reader and mmap_line_reader_sv, easy-to-use line-based file readers.

It is implemented with memory-mapped file APIs.

Date

2017-09-10

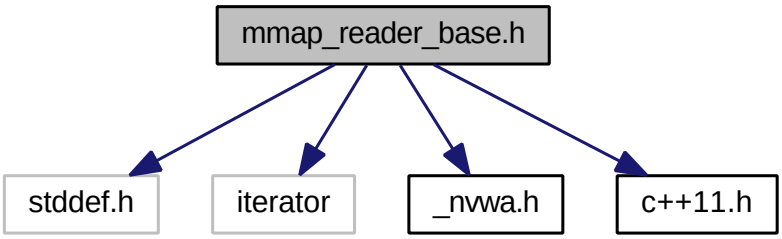
7.20 mmap_reader_base.h File Reference

Header file for mmap_reader_base, common base for mmap-based file readers.

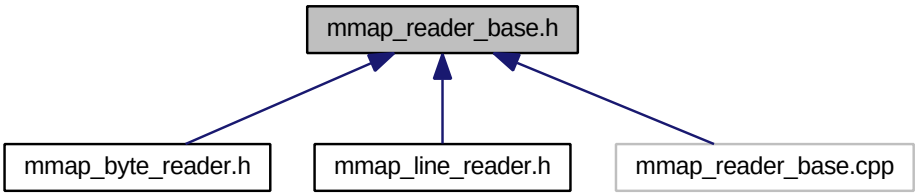
```
#include <stddef.h>
#include <iterator>
#include "_nvwa.h"
```

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

Include dependency graph for mmap_reader_base.h:



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



Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.20.1 Detailed Description

Header file for mmap_reader_base, common base for mmap-based file readers.

It currently supports POSIX and Win32.

Date

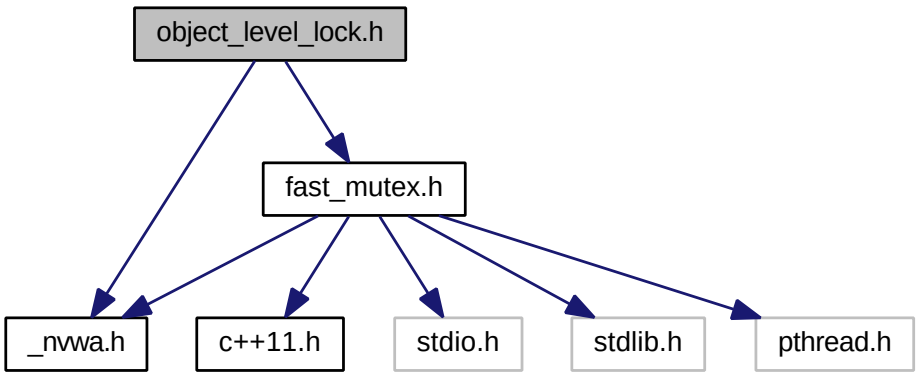
2017-09-12

7.21 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:



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.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.21.1 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.

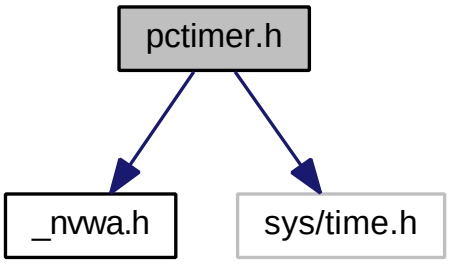
Date

2013-03-01

7.22 pctimer.h File Reference

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

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



Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.22.1 Detailed Description

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

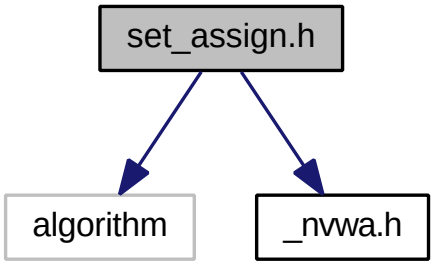
Date

2015-09-20

7.23 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

- **nvwa**

  Namespace of the nvwa project.

7.23.1 Detailed Description

Definition of template functions set_assign_union and set_assign_difference.

Date

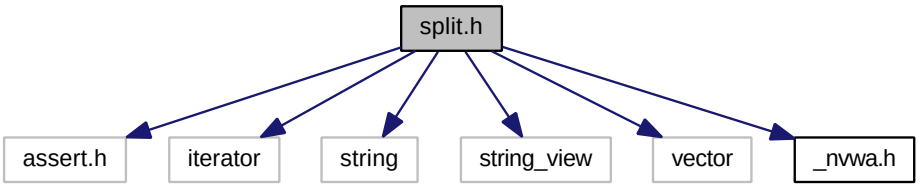
2013-03-01

7.24 split.h File Reference

Header file for an efficient, lazy split function, when using ranges seems an overkill.

```
#include <assert.h>
#include <iterator>
#include <string>
#include <string_view>
#include <vector>
#include "_nvwa.h"
```

Include dependency graph for split.h:



Classes

- class **nvwa::basic_split_view**< **_StringType, _DelimiterType** >

  Class to allow iteration over split items from the input.
- class **nvwa::basic_split_view**< **_StringType, _DelimiterType** >**::iterator**

  Iterator over the split items.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Functions

- template<typename _StringType , typename _DelimiterType >
  constexpr basic_split_view< _StringType, _DelimiterType > **nvwa::split** (const _StringType &src, _↩ DelimiterType delimiter) noexcept

  Splits a string (or string_view) into lazy views.

7.24.1 Detailed Description

Header file for an efficient, lazy split function, when using ranges seems an overkill.

It allows using split in a Pythonic way, e.g.:

```
for (auto& word : nvwa::split(a_long_string_or_string_view, ' ')) {
    // Process word
}
```

Date

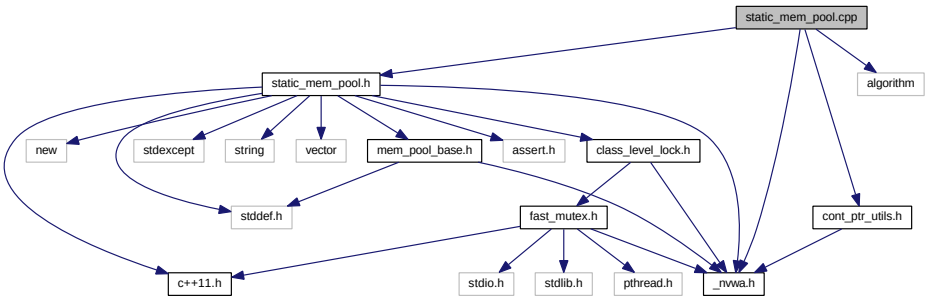
2017-12-31

7.25 static_mem_pool.cpp File Reference

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

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

Include dependency graph for static_mem_pool.cpp:



Namespaces

- **nvwa**

  Namespace of the nvwa project.

7.25.1 Detailed Description

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

Date

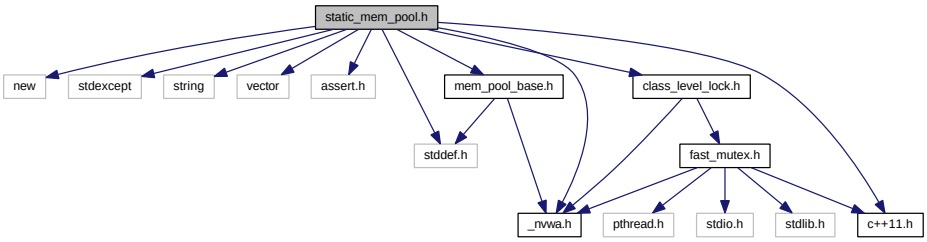
2017-09-09

7.26 static_mem_pool.h File Reference

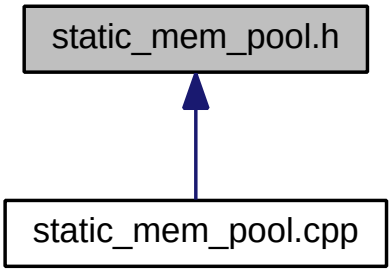
Header file for the 'static' memory pool.

```
#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:



Classes

- class **nvwa::static_mem_pool_set**

  *Singleton class to maintain a set of existing instantiations of **static_mem_pool** (p. 85).*
- class **nvwa::static_mem_pool< _Sz, _Gid >**

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

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Macros

- #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.

7.26.1 Detailed Description

Header file for the 'static' memory pool.

Date

2014-11-29

7.26.2 Macro Definition Documentation

7.26.2.1 DECLARE_STATIC_MEM_POOL

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

Value:

```
public: \
    static void* operator new(size_t size) \
    { \
        assert(size == sizeof(_Cls)); \
        void* ptr; \
        ptr = NVWA::static_mem_pool<sizeof(_Cls)>:: \
                                instance_known().allocate(); \
        if (ptr == _NULLPTR) \
            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 (p. 133)
DECLARE_STATIC_MEM_POOL_GROUPED (p. 133)
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW (p. 134)

7.26.2.2 DECLARE_STATIC_MEM_POOL__NOTHROW

```
#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 (p. 132)
DECLARE_STATIC_MEM_POOL_GROUPED (p. 133)
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW (p. 134)

7.26.2.3 DECLARE_STATIC_MEM_POOL_GROUPED

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

Value:

```
public: \
    static void* operator new(size_t size) \
    { \
        assert(size == sizeof(_Cls)); \
        void* ptr; \
        ptr = NVWA::static_mem_pool<sizeof(_Cls), (_Gid)>:: \
                                instance_known().allocate(); \
        if (ptr == _NULLPTR) \
            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 (p. 132)
DECLARE_STATIC_MEM_POOL__NOTHROW (p. 133)
DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW (p. 134)

7.26.2.4 DECLARE_STATIC_MEM_POOL_GROUPED__NOTHROW

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

Value:

```
public: \
    static void* operator new(size_t size) _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

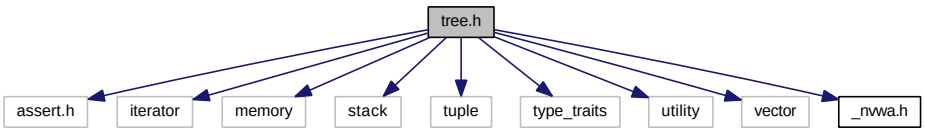
DECLARE_STATIC_MEM_POOL (p. 132)
DECLARE_STATIC_MEM_POOL__NOTHROW (p. 133)
DECLARE_STATIC_MEM_POOL_GROUPED (p. 133)

7.27 tree.h File Reference

A generic tree class template and the traversal utilities.

```
#include <assert.h>
#include <iterator>
#include <memory>
#include <stack>
#include <tuple>
#include <type_traits>
#include <utility>
#include <vector>
#include "_nvwa.h"
```

Include dependency graph for tree.h:



Classes

- struct **nvwa::smart_ptr**< **_Tp, _Policy** >
  Declaration of policy class to generate the smart pointer type.
- struct **nvwa::smart_ptr**< **_Tp, storage_policy::unique** >
  Partial specialization to get std::unique_ptr.
- struct **nvwa::smart_ptr**< **_Tp, storage_policy::shared** >
  Partial specialization to get std::shared_ptr.
- class **nvwa::tree**< **_Tp, _Policy** >
  Basic tree (node) class template that owns all its children.
- class **nvwa::breadth_first_iteration**< **_Tree** >
  Iteration class for breadth-first traversal.
- class **nvwa::depth_first_iteration**< **_Tree** >
  Iteration class for depth-first traversal.
- class **nvwa::in_order_iteration**< **_Tree** >
  Iteration class for in-order traversal.

Namespaces

- **nvwa**

  Namespace of the nvwa project.

Enumerations

- enum **nvwa::storage_policy** { **nvwa::storage_policy::unique**, **nvwa::storage_policy::shared** }

  Policy class for how to store members.

Functions

- template<storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY, typename _Tp >
  tree< typename std::decay< _Tp >::type, _Policy >::tree_ptr **nvwa::create_tree** (_Tp &&value)

  Creates a tree without any children.
- template<storage_policy _Policy = NVWA_TREE_DEFAULT_STORAGE_POLICY, typename _Tp , typename... Args>
  tree< typename std::decay< _Tp >::type, _Policy >::tree_ptr **nvwa::create_tree** (_Tp &&value, Args &&... args)

  Creates a tree with children.

7.27.1 Detailed Description

A generic tree class template and the traversal utilities.

Using this file requires a C++11-compliant compiler.

Date

2017-05-14

Index