A Pool of Blazingly Fast Stacks

Generated by Doxygen 1.9.1

Chapter 1

Namespace Index

1.1 Namespace List

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Namespace Documentation

4.1 internal Namespace Reference

Validity of pre- and post-conditions and any other requirements must be properly checked.

Classes

- class MessageHandler

  Used to handle the optional message provided by the user.
- struct AssertHelper

  Helper class to manage the construction and throwing of the proper exception type.
- class NullStream

  Used like /dev/null for the assertions when compiled in release mode.

4.1.1 Detailed Description

Validity of pre- and post-conditions and any other requirements must be properly checked.

To this aim, in this file a collection of AP_ASSERTs and AP_ERRORs is provided. The assertions (and checks) are performed only when the code is compiled without the -DNDEBUG flag. Assertions are never enough. Put as many assertions as you can without any worry for loss of performance (in release).

Example of usage

AP_ASSERT(condition);

AP_ASSERT(condition) << "optional additional message" << std::endl;

AP_ASSERT_IN_RANGE(element,min,max); // check if element belongs to [min,max]

AP_ASSERT_EQ(a,b); // check if a == b, do not use with floating point numbers

AP_ASSERT_LT(a,b); // check if a < b

AP_ASSERT_LE(a,b); // check if a <= b

AP_ASSERT_GT(a,b); // check if a > b

AP_ASSERT_GE(a,b); // check if a >= b

All the above, by default, throw std::runtime_error.

If you whant to throw your own exception type you can use the macro AP_ASSERT as follows

AP_ASSERT(condition, exception_type);

AP_ASSERT(condition, exception_type) << "optional" << " message" << std::endl;

The only constraint on the exception_type is that it must have a constructor that takes a const std::string& or a const char * (as the std::exceptions). For example

struct my_exception: public std::runtime_error{ using std::runtime_error::runtime_error; // using the same constructors // of the parent };

AP_ASSERT(1>2,my_exception); // it will throw my_exception

If you want/need to create a specific type of assert with all the parameters you want you can do as follows

#define AP_ASSERT_CUSTOM(a,b,c,d) \ *AP_ASSERT*(((a<b) && (c>d)) || (b == c), std::runtime_error) \ << "all " << "what you whant to write to debug " << a << " " << b << " " \ << c << " and " << d << std::endl;

of course you are free to replace std::runtime_error to any exception you like that can be constructed as explained before.

If a condition must be always checked (i.e., also when the code is compiled in release mode), use the AP_ERROR interface

AP_ERROR(condition); // throws an std::rutime_erorr

AP_ERROR(condition) << "optional" << " message" << std::endl;

If you need to throw a particular exception, the syntax and the requirements are the same for the assertions explained above.

AP_ERROR(condition, exception_type); AP_ERROR(condition, exception_type) << "optional" << " message" << std::endl;

The user should use only the above interface. All the rest of this file are technical details and for this reason they are put inside an internal namespace

Chapter 5

Class Documentation

5.1 _stack_iterator< T, N, S_P > Class Template Reference

Class `_stack_iterator`: iterator allowing to navigate stacks in the stack_pool data structure.

`#include <stack_iterator.hpp>`

Public Types

- using **value_type** = T
- using **reference** = value_type &
- using **pointer** = value_type *
- using **difference_type** = std::ptrdiff_t
- using **iterator_category** = std::forward_iterator_tag

Public Member Functions

- _stack_iterator (stack_type x, pool_type *const my_pool)
  Custom constructor: initializes `pool_ptr` and `index` with the passed values.
- ~_stack_iterator () noexcept=default
  Default destructor.
- reference operator* () const
  Dereference operator.
- pointer operator-> () const
  Reference operator.
- _stack_iterator & operator++ ()
  PreIncrement: increments.
- _stack_iterator & operator++ (int)
  PostIncrement: increments.

Private Types

- using **pool_type** = S_P
- using **stack_type** = N

Private Attributes

- pool_type * pool_ptr

  Pointer to stack_pool, will store the passed pool address.
- stack_type index

  Variable of type stack_type that will store the passed index.

Friends

- bool operator== (const _stack_iterator &x, const _stack_iterator &y)

  Equality operator overload.
- bool operator!= (const _stack_iterator &x, const _stack_iterator &y)

  Inequality operator overload.
- std::ostream & operator<< (std::ostream &os, const _stack_iterator &si)

  Put-to operator overload.

5.1.1 Detailed Description

template<typename T, typename N, typename S_P>
class _stack_iterator< T, N, S_P >

Class `_stack_iterator`: iterator allowing to navigate stacks in the stack_pool data structure.

Notice: the class allows to iterate through a single stack at a time!

Template Parameters

T	type of the values carried by each node.
N	stack/index type
S_P	stack_pool type, templating the function on the type it's supposed to work on

5.1.2 Constructor & Destructor Documentation

5.1.2.1 _stack_iterator()

```
template<typename T , typename N , typename S_P >
_stack_iterator< T, N, S_P >::_stack_iterator (
            stack_type x,
            pool_type *const my_pool )  [inline]
```

Custom constructor: initializes `pool_ptr` and `index` with the passed values.

Throws if my_pool is of type pool_type but points to nothing. Don't do that!
Given the previous does not happen, throws if the index is not in the pool, hence larger than `pool_ptr` -> psize() since up to size() all nodes belong to a stack or to free_nodes, hence could be successfully used to build an iterator

Parameters

x	index value
my_pool	pointer to the stack_pool, it's a constant pointer

5.1.3 Member Function Documentation

5.1.3.1 operator*()

```
template<typename T , typename N , typename S_P >
reference _stack_iterator< T, N, S_P >::operator* ( ) const  [inline]
```

Dereference operator.

Given that the object already passed from the constructor so initial index and pool_ptr are fine, there's a problem if the index is end() or reaches it due to the increment of the iterator. See `stack_pool<T,N>::value()`.

Returns

value of the node at index

5.1.3.2 operator++() [1/2]

```
template<typename T , typename N , typename S_P >
_stack_iterator& _stack_iterator< T, N, S_P >::operator++ ( )  [inline]
```

PreIncrement: increments.

Given that the object already passed from the constructor so initial index and pool_ptr are fine, there's a problem if the index is end() or reaches it due to the increment of the iterator. See `stack_pool<T,N>::next()`.

Returns

the incremented iterator

5.1.3.3 operator++() [2/2]

```
template<typename T , typename N , typename S_P >
_stack_iterator& _stack_iterator< T, N, S_P >::operator++ (
            int  )  [inline]
```

PostIncrement: increments.

Exceptions due to `operator++()`.

Returns

the iterator

5.1.3.4 operator->()

```
template<typename T , typename N , typename S_P >
pointer _stack_iterator< T, N, S_P >::operator-> ( ) const  [inline]
```

Reference operator.

Throws if problems from `operator*`

Returns

address of the value of the node at index (pointer)

5.1.4 Friends And Related Function Documentation

5.1.4.1 operator"!=

```
template<typename T , typename N , typename S_P >
bool operator!= (
            const _stack_iterator< T, N, S_P > & x,
            const _stack_iterator< T, N, S_P > & y )  [friend]
```

Inequality operator overload.

Parameters

x	reference to an iterator
y	reference to another iterator

Returns

a boolean: false (iterators at the same node) or true (iterators at different nodes)

5.1.4.2 operator<<

```
template<typename T , typename N , typename S_P >
std::ostream& operator<< (
            std::ostream & os,
            const _stack_iterator< T, N, S_P > & si )  [friend]
```

Put-to operator overload.

Parameters

os	reference to output stream
si	reference to iterator

Returns

output stream

5.1.4.3 operator==

```
template<typename T , typename N , typename S_P >
bool operator== (
            const _stack_iterator< T, N, S_P > & x,
            const _stack_iterator< T, N, S_P > & y )  [friend]
```

Equality operator overload.

Parameters

x	reference to an iterator
y	reference to another iterator

Returns

a boolean: true (iterators at the same node) or false (iterators at different nodes)

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

- stack_iterator.hpp

5.2 internal::AssertHelper< ET > Struct Template Reference

Helper class to manage the construction and throwing of the proper exception type.

`#include <ap_error.hpp>`

Public Member Functions

- void **operator=** (const MessageHandler &m)

5.2.1 Detailed Description

template<typename ET>
struct internal::AssertHelper< ET >

Helper class to manage the construction and throwing of the proper exception type.

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

- ap_error.hpp

5.3 internal::MessageHandler Class Reference

Used to handle the optional message provided by the user.

`#include <ap_error.hpp>`

Public Member Functions

- **MessageHandler** (const MessageHandler &)=delete
- template<typename T >
  MessageHandler & **operator**<< (const T &val)
- template<typename T >
  MessageHandler & **operator**<< (T *const &p)
- MessageHandler & **operator**<< (std::ostream &(*basic_manipulator)(std::ostream &))
- MessageHandler & **operator**<< (const bool b)
- std::string **get_string** () const

Private Attributes

- std::ostringstream **_os**

5.3.1 Detailed Description

Used to handle the optional message provided by the user.

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

- ap_error.hpp

5.4 stack_pool< T, N >::node_t Struct Reference

Class `node_t`, implementing the concept of node of a stack.

Public Member Functions

- node_t (const T &v, N index) noexcept

  Custom constructor taking l-value: initializes `value` and `index` with the passed values.
- node_t (T &&v, N index) noexcept

  Custom constructor taking r-value: initializes `value` and `index` with the passed values.
- ~node_t () noexcept=default

  Default destructor, explicitly = default.

Public Attributes

- T value

  value of type T carried by the node
- N next

  index to the next node, type N

5.4.1 Detailed Description

template<typename T, typename N = std::size_t>
struct stack_pool< T, N >::node_t

Class node_t, implementing the concept of node of a stack.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 node_t() [1/2]

```
template<typename T , typename N = std::size_t>
stack_pool< T, N >::node_t::node_t (
            const T & v,
            N index )  [inline], [noexcept]
```

Custom constructor taking l-value: initializes value and index with the passed values.

Does not throw: upstream checks done by stack_pool<T,N>::_push(), which is the only one able to call it

Parameters

v	const reference to the value the node will carry
index	index of the next node

5.4.2.2 node_t() [2/2]

```
template<typename T , typename N = std::size_t>
stack_pool< T, N >::node_t::node_t (
            T && v,
            N index )  [inline], [noexcept]
```

Custom constructor taking r-value: initializes value and index with the passed values.

Does not throw: upstream checks done by stack_pool<T,N>::_push(), which is the only one able to call it

Parameters

v	r-value, indicating the value the node will carry
index	index of the next node

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

- stack_pool.hpp

5.5 internal::NullStream Class Reference

Used like /dev/null for the assertions when compiled in release mode.

`#include <ap_error.hpp>`

Public Member Functions

- template<typename T >
  NullStream & **operator**<< (const T &)
- NullStream & **operator**<< (std::ostream &(*)(std::ostream &))

5.5.1 Detailed Description

Used like /dev/null for the assertions when compiled in release mode.

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

- ap_error.hpp

5.6 stack_pool< T, N > Class Template Reference

Class `stack_pool`: pool of stacks, data structures compliant with the LastInFirstOut rule.

`#include <stack_pool.hpp>`

Classes

- struct node_t

  Class `node_t`, implementing the concept of node of a stack.

Public Types

- using **iterator** = _stack_iterator< value_type, stack_type, pool_type >
- using **const_iterator** = _stack_iterator< const value_type, stack_type, const pool_type >

Public Member Functions

- stack_pool () noexcept

  Default constructor, sets free_nodes as empty.
- stack_pool (size_type n)

  Custom constructor, reserves n nodes in the pool, sets free_nodes as empty.
- iterator begin (stack_type x)

  Function providing the iterator to the first element of the stack.
- iterator end (stack_type) noexcept

  Function providing the iterator to the last element of the stack.
- const_iterator begin (stack_type x) const

  Overloaded begin function providing a const iterator to the first element of the stack.
- const_iterator end (stack_type) const noexcept

  Overloaded end function providing the iterator to the last element of the stack.
- const_iterator cbegin (stack_type x) const

  Constant begin function providing a const iterator to the first element of the stack.
- const_iterator cend (stack_type) const noexcept

  Constant end function providing the iterator to the last element of the stack.
- stack_type end () const noexcept

  Function providing the proxy index of the end of the stacks.
- stack_type new_stack () noexcept

  Function providing the head of a new empty stack.
- T & value (stack_type x)

  Function providing access to the node value at the given index.
- const T & value (stack_type x) const

  Constant function providing access to the node value at the given index.
- stack_type & next (stack_type x)

  Function providing access to the node next element at the given index.
- const stack_type & next (stack_type x) const

  Constant function providing access to the node next element at the given index.
- void reserve (size_type n)

  Function allowing to reserve n nodes in an already present pool.
- size_type capacity () const noexcept

  Function allowing to assess the current capacity of the pool.
- size_type psize () const noexcept

  Function allowing to assess the current size of the pool.
- bool empty (stack_type x) const noexcept

  Function allowing to assess whether the given stack is empty or not.
- stack_type push (const T &val, stack_type head)

  Function taking l-value references able to add a node to the stack.
- stack_type push (T &&val, stack_type head)

  Function taking r-value references able to add a node to the stack.
- stack_type pop (stack_type x)

  Function that deleted the first node of the given stack.
- stack_type free_stack (stack_type x)

  Function that deletes an entire stack.
- size_type ssize (stack_type x) const

  Function allowing to assess the size of the given stack.
- value_type & reach (stack_type x, stack_type m)

  *Function allowing to reach the value of the **mth** node in the stack.*
- const value_type & reach (stack_type x, stack_type m) const

  *Constant function allowing to reach the value of the **mth** node in the stack.*
- void print_stack (stack_type x) const

  Constant function printing the value of each node in the stack.

Private Types

- using **value_type** = T
- using **stack_type** = N
- using **size_type** = typename std::vector< node_t >::size_type
- using **pool_type** = stack_pool< value_type, stack_type >

Private Member Functions

- node_t & node (stack_type x) noexcept

  Function providing the node at a given index.
- const node_t & node (stack_type x) const noexcept

  Constant function providing the node at a given index.
- template<typename V >
  stack_type _push (V &&val, stack_type head)

  Templated auxiliary function allowing to add 1 node using the public method push.
- void _new_first (stack_type &s1, stack_type &s2)

  Auxiliary function allowing to transfer the ownership of the first node from a stack to another.
- stack_type & _last_jump (stack_type x) noexcept

  Auxiliary function allowing to easily reach the `next` element of the last node of the passed stack.

Private Attributes

- std::vector< node_t > pool

  std::vector of nodes, the support of the pool.
- stack_type free_nodes

  Stack of free nodes.

5.6.1 Detailed Description

template<typename T, typename N = std::size_t>
class stack_pool< T, N >

Class `stack_pool`: pool of stacks, data structures compliant with the LastInFirstOut rule.

A stack of nodes is a data structure implementing the LastInFirstOut rule, stating that the last added element (node) will also be the first one removed. The only allowed insertions/removals occur in fact in the front of the stack, and are implemented through the `push()` and `pop()` methods.
The proposed implementation employs an std::vector as support of the pool, exploiting its indexing to provide a simple yet effective identification method for nodes and stacks: each node will be identified by its index on the vector + 1, each stack by its first node's index, referred to as head.
The nodes themselves are objects of type node: nested in the `stack_pool` it's in fact implemented the class representing the concept of node, `node_t`, a simple structure carrying a value and the index of the next node. The nested implementation of the class `node_t` was the most sensible choice due to its templates being the same of `stack_pool`, along with its reduced size and close connection with the latter.
Going back to templates, `stack_pool` has two templates, allowing the user to choose the desired type for both the values carried by the nodes and their indexes.
Also, the aim is building a blazingly fast data structure and to this end the implementations tries to mitigate two common bottlenecks caused by the slow, slow memory: the allocation of the elements one by one and the distance between them. The first issue is mitigated by the employment of methods allowing to reserve a certain memory region upfront, the second by the very use of `std::vector<node_t>`, allowing to keep the nodes organized

and close to each other.
But why, as I spoiled before, the nodes are indexes as their real index on the node + 1? The answer is simple: convenience. This indexing system in fact allows to use index 0 (not a real index in the vector, would be -1) as proxy for the end of the stack: if the next index is 0 the current node is the last one, if the head is 0 the stack is empty.
How does the implementation deal with stacks resizing? When the size increases, hence when nodes are added, the std::vector takes care of the possible need to increase its capacity; when the size decreases, so when nodes are removed, the std::vector slots previously owned by the shrinked stack are not left unused: they are added to a stack of free nodes (free indexes) which will to be occupied by the next newly added nodes. Free nodes have the priority over never used std::vector slots, which will begin to be filled only when no more free nodes result available.

Templates guidelines: the class has been designed with N being an unsigned integral type in mind, since indicating indexes this makes the most sense (there is no type check: be kind to yourself, don't use clearly unsuitable types!).
Notice: the choice of different types will impact the class in the following ways:

- small types: better performances; many methods are implemented by passing arguments by value, which is cheaper then passing references if the ints are small.
- large types: larger pool, since the correct implementation of the pool is possible as long as there are enough indexes to represent the nodes

Template Parameters

T	type of the values carried by each node
N	stack/index type

5.6.2 Constructor & Destructor Documentation

5.6.2.1 stack_pool()

```
template<typename T , typename N = std::size_t>
stack_pool< T, N >::stack_pool (
            size_type n )  [inline], [explicit]
```

Custom constructor, reserves n nodes in the pool, sets free_nodes as empty.

Notice, the nodes are reserved but not constructed. Reserving nodes allows to avoid reallocation each time the capacity of the vector is reached.
`std::vector<T>::reserve(...)` throws in case insufficient memory is available

Parameters

n	number of nodes to reserve

5.6.3 Member Function Documentation

5.6.3.1 _last_jump()

```
template<typename T , typename N = std::size_t>
stack_type& stack_pool< T, N >::_last_jump (
            stack_type x )  [inline], [private], [noexcept]
```

Auxiliary function allowing to easily reach the `next` element of the last node of the passed stack.

See `supplementary_materials`
The function does not throw since it's only passed the head of non-empty free_nodes:

- `x` is not equal to `end()`
- `x` is not larger than `psize()` Notice, the function does not modify the object itself, but it's tailored to return a value so that it can be modified, hence the const qualification would not suit the intents of the function.

Parameters

x	head of a stack, index

Returns

reference to the last node's `next()` element, always equal to `end()`

5.6.3.2 _new_first()

```
template<typename T , typename N = std::size_t>
void stack_pool< T, N >::_new_first (
            stack_type & s1,
            stack_type & s2 )  [inline], [private]
```

Auxiliary function allowing to transfer the ownership of the first node from a stack to another.

The first node of `stack2` (head `s2`) becomes the new first node (head `s1`) of `stack1`; to do that the following movements occur "in parallel":

- s1 becomes s2
- s2 becomes next(s2)
- next(s2) becomes s1 See `supplementary_materials`
  The function throws through `next()` if s2 is equal to `end()` or larger than `psize()` since there are no nodes there.
  Throws "de novo" if s1 is larger than `psize()`; no problem if s1 is equal to `end()`
  When the function is accessed through _push, we can be sure that at least the first condition is respected since it's accessed if !(empty(free_nodes))

Template Parameters

V	deduced from the arguments passed to the function

Parameters

val	universal reference to the value of the new node
head	index that will become the next of the new node

Returns

the new head of the stack hence index of the new node

5.6.3.3 _push()

```
template<typename T , typename N = std::size_t>
template<typename V >
stack_type stack_pool< T, N >::_push (
            V && val,
            stack_type head )  [inline], [private]
```

Templated auxiliary function allowing to add 1 node using the public method push.

The function can take both r-values and r-value while being able to correctly identify and treat them.
If free_nodes it's empty the function proceeds to add the node at the end of the pool.
If free_nodes it's not empty: the first node of free_nodes becomes the new node and free_nodes shrinks, while the next element of the already gone first element becomes its new head. See supplementary_materials.
The function throws through push_back and through _new_first (in case
head is equal to end() or larger than psize())

Parameters

V	deduced from the arguments passed to the function
val	universal reference to the value of the new node
head	current head of the stack, index that will become the next of the new node

Returns

the new head of the stack hence index of the new node

5.6.3.4 begin() [1/2]

```
template<typename T , typename N = std::size_t>
iterator stack_pool< T, N >::begin (
            stack_type x )  [inline]
```

Function providing the iterator to the first element of the stack.

Easy way to obtain the iterator to the first element, without the need of coding its instantiation.
Keyword this is used to point at the object, in order to maintain the connection between the pool and the iterator.
Throws through the constructor of _stack_iterator<> if the given index is larger than `psize()`, hence not a plausible index for any stack

Parameters

x	head of the stack

Returns

iterator to the first element

5.6.3.5 begin() [2/2]

```
template<typename T , typename N = std::size_t>
const_iterator stack_pool< T, N >::begin (
            stack_type x ) const  [inline]
```

Overloaded begin function providing a const iterator to the first element of the stack.

Easy way to obtain the iterator to the first element, without the need of coding its instantiation.
Keyword `this` is used to point at the object, in order to maintain the connection between the pool and the iterator.
Throws through the constructor of _stack_iterator<> if the given index is larger than `psize()`, hence not a plausible index for any stack

Parameters

x	head of the stack

Returns

const iterator to the first element

5.6.3.6 capacity()

```
template<typename T , typename N = std::size_t>
size_type stack_pool< T, N >::capacity ( ) const  [inline], [noexcept]
```

Function allowing to assess the current capacity of the pool.

It does not throw since `std::vector<T>::capacity()` is no-throw guaranteed.

Returns

the pool capacity

5.6.3.7 cbegin()

```
template<typename T , typename N = std::size_t>
const_iterator stack_pool< T, N >::cbegin (
            stack_type x ) const  [inline]
```

Constant begin function providing a const iterator to the first element of the stack.

Easy way to obtain the iterator to the first element, without the need of coding its instantiation.
Keyword `this` is used to point at the object, in order to maintain the connection between the pool and the iterator.
Throws through the constructor of _stack_iterator<> if the given index is larger than `psize()`, hence not a plausible index for any stack

Parameters

x	head of the stack

Returns

const iterator to the first element

5.6.3.8 cend()

```
template<typename T , typename N = std::size_t>
const_iterator stack_pool< T, N >::cend (
            stack_type  ) const  [inline], [noexcept]
```

Constant end function providing the iterator to the last element of the stack.

Since the proxy end of the stack is simply the zero element, the function returns an iterator to `end()`; no parameter is passed since any passed index would remain unused, in this way the warnings are avoided and there's no throwing risk.
Keyword `this` is used to point at the object, in order to maintain the connection between the pool and the iterator.

Returns

const iterator to the proxy last element

5.6.3.9 empty()

```
template<typename T , typename N = std::size_t>
bool stack_pool< T, N >::empty (
            stack_type x ) const  [inline], [noexcept]
```

Function allowing to assess whether the given stack is empty or not.

In order to check if a stack is empty or not it's enough to check if the head is equal to `end()`

Returns

true if the stack is empty, false if it's not

5.6.3.10 end() [1/3]

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::end ( ) const  [inline], [noexcept]
```

Function providing the proxy index of the end of the stacks.

Returns

element 0 casted in the correct way to `stack_type`

5.6.3.11 end() [2/3]

```
template<typename T , typename N = std::size_t>
const_iterator stack_pool< T, N >::end (
            stack_type  ) const  [inline], [noexcept]
```

Overloaded end function providing the iterator to the last element of the stack.

Since the proxy end of the stack is simply the zero element, the function returns an iterator to `end()`; no parameter is passed since any passed index would remain unused, in this way the warnings are avoided and there's no throwing risk.
Keyword `this` is used to point at the object, in order to maintain the connection between the pool and the iterator.

Returns

const iterator to the proxy last element

5.6.3.12 end() [3/3]

```
template<typename T , typename N = std::size_t>
iterator stack_pool< T, N >::end (
            stack_type  )  [inline], [noexcept]
```

Function providing the iterator to the last element of the stack.

Since the proxy end of the stack is simply the zero element, the function returns an iterator to `end()`; no parameter is passed since any passed index would remain unused, in this way the warnings are avoided and there's no throwing risk.
Keyword `this` is used to point at the object, in order to maintain the connection between the pool and the iterator.

Returns

iterator to the proxy last element

5.6.3.13 free_stack()

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::free_stack (
            stack_type x )  [inline]
```

Function that deletes an entire stack.

The function takes the head of the stack as a parameter and sets the stack to end(). **Be careful**, if an index different from a head is supplied to the function the portion of stack up to the pointed node will be deleted!
The removed stack is added to free_nodes.

- If free_nodes is empty, the function simply works by assigning it to head before setting the latter to `end()`
- If free_nodes is not empty, the auxiliary function `_last_jump()` is called, returning the next element of the last node and assigning it to head, before setting the latter to `end()`; the choice of reaching the end of free_nodes instead of the one of the given stack and do the opposite procedure is due to the fact that free_nodes is bound to shrink, while stacks to increase, so in the majority of the cases free_nodes should be shorter than the stacks, which is relevant since we have to travel up to its last node. See `supplementary_materials`
  The function throws if `x` is larger than `psize()` since assigning to free nodes an index not pointing to nodes would not makes sense and would cause troubles with free_nodes.

Parameters

head	of the stack, index of the stack to remove

Returns

the new head of the freed stack, always `end()`

5.6.3.14 new_stack()

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::new_stack ( )  [inline], [noexcept]
```

Function providing the head of a new empty stack.

Returns

head of the empty new stack which is always `end()`

5.6.3.15 next() [1/2]

```
template<typename T , typename N = std::size_t>
stack_type& stack_pool< T, N >::next (
            stack_type x )  [inline]
```

Function providing access to the node next element at the given index.

The function throws if the given index is equal to `end()` or larger than `psize()`, since at these indexes there is no next at all

Parameters

x	index of a node

Returns

reference to the next index of the node identified by `x`

5.6.3.16 next() [2/2]

```
template<typename T , typename N = std::size_t>
const stack_type& stack_pool< T, N >::next (
            stack_type x ) const  [inline]
```

Constant function providing access to the node next element at the given index.

The function throws if the given index is equal to `end()` or larger than `psize()`, since at these indexes there is no next at all

Parameters

x	index of a node

Returns

constant reference to the next index of the node identified by `x`

5.6.3.17 node() [1/2]

```
template<typename T , typename N = std::size_t>
const node_t& stack_pool< T, N >::node (
            stack_type x ) const  [inline], [private], [noexcept]
```

Constant function providing the node at a given index.

Parameters

x	"stack index" of a node, which is real index + 1

Returns

the node at the correct index

5.6.3.18 node() [2/2]

```
template<typename T , typename N = std::size_t>
node_t& stack_pool< T, N >::node (
            stack_type x )  [inline], [private], [noexcept]
```

Function providing the node at a given index.

Parameters

x	"stack index" of a node, which is real index + 1

Returns

the node at the correct index

5.6.3.19 pop()

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::pop (
            stack_type x )  [inline]
```

Function that deleted the first node of the given stack.

The function takes the head of the stack as a parameter and pops the first element. **Be careful**, if an index different from a head is supplied to the function a node imbetween the stack will be deleted!
The removed node is added to free_nodes.
The function works by assigning to free_nodes the original head, to the original head the original next element of the first node, and to the latter the original free_nodes. See `supplementary_materials`
Calls the auxiliary function `_new_first` and throws through it.

Parameters

head	of the stack, index of the node to be removed

Returns

the new head of the stack after removing the first node

5.6.3.20 print_stack()

```
template<typename T , typename N = std::size_t>
void stack_pool< T, N >::print_stack (
            stack_type x ) const  [inline]
```

Constant function printing the value of each node in the stack.

Be careful, if an intermediate index is passed instead of The function throws through `next()`: if `x` is larger than `psize()` the operator++ of the iterator calls `next()` which fails if there is no next element.

Parameters

x	stack index

5.6.3.21 psize()

```
template<typename T , typename N = std::size_t>
size_type stack_pool< T, N >::psize ( ) const  [inline], [noexcept]
```

Function allowing to assess the current size of the pool.

Notice that the size of the pool is the sum of the nodes in the stacks and in free_nodes.
It does not throw since `std::vector<T>::size()` is no-throw guaranteed.

Returns

the pool capacity

5.6.3.22 push() [1/2]

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::push (
            const T & val,
            stack_type head )  [inline]
```

Function taking l-value references able to add a node to the stack.

Calls the auxiliary function `_push()`, and throws through it.

Parameters

val	constant reference to the value of the new node
head	current head of the stack, will be the next of the new node

Returns

the new head of the stack after adding the new node

5.6.3.23 push() [2/2]

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::push (
```

```
            T && val,
            stack_type head )  [inline]
```

Function taking r-value references able to add a node to the stack.

Calls the auxiliary function `_push()`, and throws through it.

Parameters

val	r-value reference to the value of the new node
head	current head of the stack, will be the next of the new node

Returns

the new head of the stack after adding the new node

5.6.3.24 reach() [1/2]

```
template<typename T , typename N = std::size_t>
value_type& stack_pool< T, N >::reach (
            stack_type x,
            stack_type m )  [inline]
```

Function allowing to reach the value of the **mth** node in the stack.

The function allows to access the value of the mth node in the stack, where the first node: `m=1` and the last node: `m=ssize(x)`
The type of m is `stack_type` to be coherent with the order of magnitude of the stack size.
Be careful, if an intermediate index is passed instead of a head the function considers m=1 the node at the current index.

- Throws through `next()` or `value()` if x is equal to `end()` or larger than `psize()`
- Throws if the passed m is larger than `ssize()` since this would lead to the previous exception If m is equal to end() hence 0 in stack_type type, the function returns the value of the first node

Parameters

x	stack index
m	the hierarchical number of a node, going from the first (1) to the last (`ssize(x)`)

Returns

reference to the value of the reached node

5.6.3.25 reach() [2/2]

```
template<typename T , typename N = std::size_t>
```

```
const value_type& stack_pool< T, N >::reach (
            stack_type x,
            stack_type m ) const  [inline]
```

Constant function allowing to reach the value of the **mth** node in the stack.

The function allows to access the value of the mth node in the stack, where the first node: `m=1` and the last node: `m=ssize(x)`
The type of m is `stack_type` to be coherent with the order of magnitude of the stack size.
Be careful, if an intermediate index is passed instead of a head the function considers m=1 the node at the current index.

- Throws through `next()` or `value()` if x is equal to `end()` or larger than `psize()`
- Throws if the passed m is larger than `ssize()` since this would lead to the previous exception If m is equal to end() hence 0 in stack_type type, the function returns the value of the first node

Parameters

x	stack index
m	the hierarchical number of a node, going from the first (1) to the last (`ssize(x)`)

Returns

const reference to the value of the reached node

5.6.3.26 reserve()

```
template<typename T , typename N = std::size_t>
void stack_pool< T, N >::reserve (
            size_type n )  [inline]
```

Function allowing to reserve n nodes in an already present pool.

It behaves as `std::vector<T>::reserve(...)`, requesting that the vectory capacity should be at least equal to n.
`std::vector<T>::reserve(...)` throws in case insufficient memory is available

Parameters

n	number of required nodes

5.6.3.27 ssize()

```
template<typename T , typename N = std::size_t>
size_type stack_pool< T, N >::ssize (
            stack_type x ) const  [inline]
```

Function allowing to assess the size of the given stack.

Be careful, if an intermediate index is passed instead of a head the function only provides a partial size, not the entire size of the stack.
The function throws through `next()`: if `x` is larger than `psize()` the operator++ of the iterator calls `next()` which fails if there is no next element.
If `x` is equal to `end()` the function returns size zero

Parameters

x	stack index

Returns

the size of the stack

5.6.3.28 value() [1/2]

```
template<typename T , typename N = std::size_t>
T& stack_pool< T, N >::value (
            stack_type x )  [inline]
```

Function providing access to the node value at the given index.

The function throws if the given index is equal to `end()` or larger than `psize()`, since at these indexes there is no value at all

Parameters

x	index of a node

Returns

reference to the value of the node identified by `x`

5.6.3.29 value() [2/2]

```
template<typename T , typename N = std::size_t>
const T& stack_pool< T, N >::value (
            stack_type x ) const  [inline]
```

Constant function providing access to the node value at the given index.

The function throws if the given index is equal to `end()` or larger than `psize()`, since at these indexes there is no value at all

Parameters

x	index of a node

Returns

constant reference to the value of the node identified by `x`

5.6.4 Member Data Documentation

5.6.4.1 free_nodes

```
template<typename T , typename N = std::size_t>
stack_type stack_pool< T, N >::free_nodes  [private]
```

Stack of free nodes.

At the beginning it's empty. The nodes previously belonging to a stack will be added to this stack, which needs to be emptied before new nodes are added increasing the size of the vector

5.6.4.2 pool

```
template<typename T , typename N = std::size_t>
std::vector<node_t> stack_pool< T, N >::pool  [private]
```

std::vector of nodes, the support of the pool.

Initialized by the default constructor of vector

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

- stack_pool.hpp

Chapter 6

File Documentation

6.1 stack_iterator.hpp File Reference

Header file: implementation of class _stack_iterator, the iterator for the class stack_pool.

```
#include <iostream>
#include <utility>
#include <iterator>
#include "ap_error.hpp"
```

Classes

- class _stack_iterator< T, N, S_P >

  Class `_stack_iterator`: iterator allowing to navigate stacks in the stack_pool data structure.

6.1.1 Detailed Description

Header file: implementation of class _stack_iterator, the iterator for the class stack_pool.

6.2 stack_pool.hpp File Reference

Header file: implementation of class stack_pool, our pool of blazingly fast stacks.

```
#include <iostream>
#include <utility>
#include <iterator>
#include <vector>
#include "stack_iterator.hpp"
#include "ap_error.hpp"
```

Classes

- class stack_pool< T, N >

  Class `stack_pool`: pool of stacks, data structures compliant with the LastInFirstOut rule.
- struct stack_pool< T, N >::node_t

  Class `node_t`, implementing the concept of node of a stack.

6.2.1 Detailed Description

Header file: implementation of class stack_pool, our pool of blazingly fast stacks.

Index