mdds Documentation

Release 2.1.1

Kohei Yoshida

Apr 29, 2023

CONTENTS

Multi-dimensional data structure, or mdds for short, is a collection of data structures and indexing algorithms that are useful for storing and indexing multi-dimensional data for C++ projects.

Contents:

CHAPTER

ONE

GLOBALS

1.1 Macros

MDDS_ASCII(literal)

Expands a *literal* string into two arguments: the first one is the literal string itself, and the second one is the length of that string.

Note that this macro only works with literal strings defined inline; it does not work with pointer values that point to strings defined elsewhere.

MDDS_N_ELEMENTS(name)

Calculates the length of *name* array provided that the array definition is given in the same compilation unit.

Deprecated:

Please use `std::size` instead.

1.2 Exceptions

class **general_error** : public std::exception

Subclassed by *mdds::integrity_error*, *mdds::invalid_arg_error*, *mdds::mtv::element_block_error*, *mdds::size_error*, *mdds::type_error*

Public Functions

inline **general_error**(const ::std::string &msg)

inline virtual **~general_error**() noexcept

inline virtual const char ***what**() const noexcept

class **invalid_arg_error** : public mdds::*general_error*

Public Functions

inline **invalid_arg_error**(const ::std::string &msg)

class **size_error** : public mdds::*general_error*

Public Functions

inline **size_error**(const std::string &msg)

class **type_error** : public mdds::*general_error*

Public Functions

inline **type_error**(const std::string &msg)

class **integrity_error** : public mdds::*general_error*

Public Functions

inline **integrity_error**(const std::string &msg)

CHAPTER

TWO

FLAT SEGMENT TREE

2.1 Overview

Flat segment tree is a derivative of segment tree, and is designed to store non-overlapping 1-dimensional range values such that *the values of the neighboring ranges are guaranteed to be different.* An insertion of a range value into this structure will always overwrite one or more existing ranges that overlap with the new range. If an insertion of a new range would cause any adjacent ranges to have the equal value, those ranges will be merged into one range.

An instance of this structure is initialized with fixed lower and upper bounaries, which will not change throughout the life time of the instance.

The flat segment tree structure consists of two parts: the leaf-node part which also forms a doubly-linked list, and the non-leaf-node part which forms a balanced-binary tree and is used only when performing tree-based queries. The range values are stored in the leaf-nodes, while the non-leaf nodes are used only for queries.

2.2 Quick start

This section demonstrates a simple use case of storing non-overlapping ranged values and performing queries using *flat_segment_tree*.

First, we need to instantiate a concrete type from the template:

```
using fst_type = mdds::flat_segment_tree<long, int>;
```

then create an instance of this type:

```
// Define the begin and end points of the whole segment, and the default
// value.
fst_type db(0, 500, 0);
```

Here, the first and second arguments specify the lower and upper boundaries of the whole segment. The third argument specifies the value for the empty segments. What this line does is to create a new instance and initializes it with one initial segment ranging from 0 to 500 with a value of 0:

Fig. 1: The new instance is initialized with an initial segment.

Internally, this initial range is represented by two leaf nodes, with the first one storing the start key and the value for the segment both of which happen to be 0 in this example, and the second one storing the end key of 500. Note that the end key of a segment is not inclusive.

The following lines insert two new segments into this structure:

```
db.insert_front(10, 20, 10);
db.insert_back(50, 70, 15);
```

The first line inserts a segment ranging from 10 to 20 with a value of 10, and the second line from 50 to 70 with a value of 15:

Fig. 2: Two new segments have been inserted.

You can insert a new segment either via *insert_front()* or *insert_back()*. The end result will be the same regardless of which method you use; the difference is that *insert_front()* begins its search for the insertion point from the first node associated with the minimum key value, whereas *insert_back()* starts its search from the last node associated with the maximum key value.

After the insertions, the tree now contains a total of six leaf nodes to represent all stored segments. Note that one leaf node typically represents both the end of a segment and the start of the adjacent segment that comes after it, unless it's either the first or the last node.

The next line inserts another segment ranging from 60 to 65 having a value of 5:

```
db.insert_back(60, 65, 5);
```

As this new segment overlaps with the existing segment of 50 to 70, it will cut into a middle part of that segment to make room for the new segment. At this point, the tree contains a total of eight leaf nodes representing seven segments:

Fig. 3: A new segment has been inserted that overlaps an existing non-empty segment.

Next, we are going to query the value associated with a key of 15 via *search()*:

```
// Perform linear search.  This doesn't require the tree to be built ahead
// of time.
if (auto it = db.search(15); it != db.end())
{
    auto segment = it.to_segment();
    std::cout << "The value at 15 is " << segment->value
        << ", and this segment spans from " << segment->start << " to "
        << segment->end << std::endl;
}
```

Executing this code will yield the following output:

```
The value at 15 is 10, and this segment spans from 10 to 20
```

One thing to note is that the *search()* method performs a linear search which involves traversing only through the leaf nodes in this data structure in order to find the target segment. As such, the worst-case lookup performance is directly proportional to the number of leaf nodes.

There is another way to perform the query with better worse-case performance, that is through *search_tree()* as seen in the following code:

```
// Don't forget to build tree before calling search_tree().
db.build_tree();

// Perform tree search.  Tree search is generally a lot faster than linear
```

(continues on next page)

(continued from previous page)

```
// search, but requires the tree to be built ahead of time.
if (auto it = db.search_tree(62); it != db.end())
{
    auto segment = it.to_segment();
    std::cout << "The value at 62 is " << segment->value
        << ", and this segment spans from " << segment->start << " to "
        << segment->end << std::endl;
}
```

The signature of the *search_tree()* method is identical to that of the *search()* method except for the name. This code generates the following output:

```
The value at 62 is 5, and this segment spans from 60 to 65
```

Query via *search_tree()* generally performs better since it traverses through the search tree to find the target segment. But it does require the search tree to be built ahead of time by calling *build_tree()*. Please be aware that if the segments have been modified after the tree was last built, you will have to rebuild the tree by calling *build_tree()*.

Warning: You need to build the tree by calling *build_tree()* before performing a tree-based search via *search_tree()*. If the segments have been modified after the tree was last built, you will have to rebuild the tree by calling *build_tree()* again.

2.3 Iterate through stored segments

flat_segment_tree supports two types of iterators to allow you to iterate through the segments stored in your tree. The first way is to iterate through the individual leaf nodes one at a time by using *begin()* and *end()*:

```
for (auto it = db.begin(); it != db.end(); ++it)
{
    std::cout << "key: " << it->first << "; value: " << it->second << std::endl;
}
```

Each iterator value contains a pair of two values named `first` and `second`, with the first one being the key of the segment that the node initiates, and the second one being the value associated with that segment. When executing this code with the tree from the example code above, you'll get the following output:

```
key: 0; value: 0
key: 10; value: 10
key: 20; value: 0
key: 50; value: 15
key: 60; value: 5
key: 65; value: 15
key: 70; value: 0
key: 500; value: 0
```

Each node stores the start key and the value of the segment it initiates, and the key stored in each node is also the end key of the segment that the previous node initiates except for the first node. Note that the value stored in the last node is currently not used. It is set to be the zero value of the value type, but this may change in the future.

One thing to keep in mind is that *flat_segment_tree* does not support mutable iterators that let you modify the stored keys or values.

Note: *flat_segment_tree* does not support mutable iterators; you can only traverse the values in a read-only fashion.

You can also use range-based for loop to iterate through the leaf nodes in a similar fashion:

```
for (const auto& node : db)
{
    std::cout << "key: " << node.first << "; value: " << node.second << std::endl;
}
```

The output from this code is identical to that from the previous one.

Now, one major inconvenience of navigating through the individual leaf nodes is that you need to manually keep track of the start and end points of each segment if you need to operate on the segments rather than the nodes that comprise the segments. The good news is that *flat_segment_tree* does provide a way to iterate through the segments directly as the following code demonstrates:

```
for (auto it = db.begin_segment(); it != db.end_segment(); ++it)
{
    std::cout << "start: " << it->start << "; end: " << it->end << "; value: " << it->
↪value << std::endl;
}
```

This code uses *begin_segment()* and *end_segment()* to iterate through one segment at a time with the value of each iterator containing `start`, `end` and `value` members that correspond with the start key, end key and the value of the segment, respectively. Running this code produces the following output:

```
start: 0; end: 10; value: 0
start: 10; end: 20; value: 10
start: 20; end: 50; value: 0
start: 50; end: 60; value: 15
start: 60; end: 65; value: 5
start: 65; end: 70; value: 15
start: 70; end: 500; value: 0
```

It's also possible to iterate through the segments in a range-based for loop, by calling *segment_range()*:

```
for (const auto& segment : db.segment_range())
{
    std::cout << "start: " << segment.start << "; end: " << segment.end << "; value: " <
↪< segment.value << std::endl;
}
```

This code should generate output identical to that of the previous code.

2.4 API Reference

template<typename **Key**, typename **Value**>

class **flat_segment_tree**

Public Types

typedef *Key* **key_type**

typedef *Value* **value_type**

typedef size_t **size_type**

typedef __st::node<*flat_segment_tree*> **node**

typedef *node*::node_ptr **node_ptr**

typedef __st::nonleaf_node<*flat_segment_tree*> **nonleaf_node**

using **const_segment_iterator** = mdds::fst::detail::const_segment_iterator<*flat_segment_tree*>

Public Functions

inline *const_iterator* **begin**() const

Return an iterator that points to the first leaf node that correspondes with the start position of the first segment.

Returns

immutable iterator that points to the first leaf node that corresponds with the start position of the first segment.

inline *const_iterator* **end**() const

Return an iterator that points to the position past the last leaf node that corresponds with the end position of the last segment.

Returns

immutable iterator that points to the position past last leaf node that corresponds with the end position of the last segment.

inline *const_reverse_iterator* **rbegin**() const

Return an iterator that points to the last leaf node that correspondes with the end position of the last segment. This iterator moves in the reverse direction of a normal iterator.

Returns

immutable reverse iterator that points to the last leaf node that corresponds with the end position of the last segment.

inline *const_reverse_iterator* **rend**() const

Return an iterator that points to the position past the first leaf node that corresponds with the start position of the first segment. This iterator moves in the reverse direction of a normal iterator.

Returns
immutable reverse iterator that points to the position past first leaf node that corresponds with the start position of the first segment.

const_segment_iterator **begin_segment**() const

Return an immutable iterator that points to the first segment stored in the tree. It iterates through the segments one segment at a time. Each iterator value consists of `start`, `end`, and `value` members that correspond with the start and end positions of a segment and the value of that segment, respectively.

Returns
immutable iterator that points to the first segment stored in the tree.

const_segment_iterator **end_segment**() const

Return an immutable iterator that points to the position past the last segment stored in the tree. It iterates through the segments one segment at a time. Each iterator value consists of `start`, `end`, and `value` members that correspond with the start and end positions of a segment and the value of that segment, respectively.

Returns
immutable iterator that points to the position past the last segment stored in the tree.

const_segment_range_type **segment_range**() const

Return a range object that provides a begin iterator and an end sentinel.

flat_segment_tree() = delete

flat_segment_tree(*key_type* min_val, *key_type* max_val, *value_type* init_val)

Constructor that takes minimum and maximum keys and the value to be used for the initial segment.

Parameters

- **min_val** – minimum allowed key value for the entire series of segments.
- **max_val** – maximum allowed key value for the entires series of segments.
- **init_val** – value to be used for the initial segment. This value will also be used for empty segments.

flat_segment_tree(const *flat_segment_tree* &r)

Copy constructor only copies the leaf nodes.

flat_segment_tree(*flat_segment_tree* &&other)

Move constructor.

Warning: The source instance will not be usable after the move construction.

~flat_segment_tree()

flat_segment_tree<*Key*, *Value*> &**operator=**(const *flat_segment_tree* &other)

Copy assignment operator.

Note: It only copies the leaf nodes.

Parameters
other – Source instance to copy from.

flat_segment_tree<*Key*, *Value*> &**operator=**(*flat_segment_tree* &&other)

Move assignment operator.

Parameters
other – Source instance to move from.

void **swap**(*flat_segment_tree* &other)

Swap the content of the tree with another instance.

Parameters
other – instance of *flat_segment_tree* to swap content with.

void **clear**()

Remove all stored segments except for the initial segment. The minimum and maximum keys and the default value will be retained after the call returns. This call will also remove the tree.

inline std::pair<*const_iterator*, bool> **insert_front**(*key_type* start_key, *key_type* end_key, *value_type* val)

Insert a new segment into the tree. It searches for the point of insertion from the first leaf node.

Parameters

- **start_key** – start value of the segment being inserted. The value is inclusive.
- **end_key** – end value of the segment being inserted. The value is not inclusive.
- **val** – value associated with this segment.

Returns
pair of *const_iterator* corresponding to the start position of the inserted segment, and a boolean value indicating whether or not the insertion has modified the tree.

inline std::pair<*const_iterator*, bool> **insert_back**(*key_type* start_key, *key_type* end_key, *value_type* val)

Insert a new segment into the tree. Unlike the *insert_front()* counterpart, this method searches for the point of insertion from the last leaf node toward the first.

Parameters

- **start_key** – start value of the segment being inserted. The value is inclusive.
- **end_key** – end value of the segment being inserted. The value is not inclusive.
- **val** – value associated with this segment.

Returns
pair of *const_iterator* corresponding to the start position of the inserted segment, and a boolean value indicating whether or not the insertion has modified the tree.

std::pair<*const_iterator*, bool> **insert**(const *const_iterator* &pos, *key_type* start_key, *key_type* end_key, *value_type* val)

Insert a new segment into the tree at or after specified point of insertion.

Parameters

- **pos** – specified insertion point
- **start_key** – start value of the segment being inserted. The value is inclusive.
- **end_key** – end value of the segment being inserted. The value is not inclusive.
- **val** – value associated with this segment.

Returns

pair of *const_iterator* corresponding to the start position of the inserted segment, and a boolean value indicating whether or not the insertion has modified the tree.

void **shift_left**(*key_type* start_key, *key_type* end_key)

Remove a segment specified by the start and end key values, and shift the remaining segments (i.e. those segments that come after the removed segment) to left. Note that the start and end positions of the segment being removed **must** be within the base segment span.

Parameters

- **start_key** – start position of the segment being removed.
- **end_key** – end position of the segment being removed.

void **shift_right**(*key_type* pos, *key_type* size, bool skip_start_node)

Shift all segments that occur at or after the specified start position to right by the size specified.

Parameters

- **pos** – position where the right-shift occurs.
- **size** – amount of shift (must be greater than 0)
- **skip_start_node** – if true, and the specified position is at an existing node position, that node will *not* be shifted. This argument has no effect if the position specified does not coincide with any of the existing nodes.

std::pair<*const_iterator*, bool> **search**(*key_type* key, *value_type* &value, *key_type* *start_key = nullptr, *key_type* *end_key = nullptr) const

Perform leaf-node search for a value associated with a key.

Parameters

- **key** – key value
- **value** – value associated with key specified gets stored upon successful search.
- **start_key** – pointer to a variable where the start key value of the segment that contains the key gets stored upon successful search.
- **end_key** – pointer to a varaible where the end key value of the segment that contains the key gets stored upon successful search.

Returns

a pair of *const_iterator* corresponding to the start position of the segment containing the key, and a boolean value indicating whether or not the search has been successful.

std::pair<*const_iterator*, bool> **search**(const *const_iterator* &pos, *key_type* key, *value_type* &value, *key_type* *start_key = nullptr, *key_type* *end_key = nullptr) const

Perform leaf-node search for a value associated with a key.

Parameters

- **pos** – position from which the search should start. When the position is invalid, it falls back to the normal search.
- **key** – key value.
- **value** – value associated with key specified gets stored upon successful search.
- **start_key** – pointer to a variable where the start key value of the segment that contains the key gets stored upon successful search.

- **end_key** – pointer to a varaible where the end key value of the segment that contains the key gets stored upon successful search.

Returns
a pair of *const_iterator* corresponding to the start position of the segment containing the key, and a boolean value indicating whether or not the search has been successful.

const_iterator **search**(*key_type* key) const

Perform leaf-node search for a value associated with a key.

Parameters
key – Key value to perform search for.

Returns
Iterator position associated with the start position of the segment containing the key, or end iterator position upon search failure.

const_iterator **search**(const *const_iterator* &pos, *key_type* key) const

Perform leaf-node search for a value associated with a key.

Parameters

- **pos** – Position from which the search should start if valid. In case of an invalid position, it falls back to a search starting with the first position.
- **key** – Key value to perform search for.

Returns
Iterator position associated with the start position of the segment containing the key, or end iterator position if the search has failed.

std::pair<*const_iterator*, bool> **search_tree**(*key_type* key, *value_type* &value, *key_type* *start_key = nullptr, *key_type* *end_key = nullptr) const

Perform tree search for a value associated with a key. This method assumes that the tree is valid. Call *is_tree_valid()* to find out whether the tree is valid, and *build_tree()* to build a new tree in case it's not.

Parameters

- **key** – key value
- **value** – value associated with key specified gets stored upon successful search.
- **start_key** – pointer to a variable where the start key value of the segment that contains the key gets stored upon successful search.
- **end_key** – pointer to a varaible where the end key value of the segment that contains the key gets stored upon successful search.

Returns
a pair of *const_iterator* corresponding to the start position of the segment containing the key, and a boolean value indicating whether or not the search has been successful.

const_iterator **search_tree**(*key_type* key) const

Perform tree search for a value associated with a key. The tree must be valid before performing the search, else the search will fail.

Parameters
key – Key value to perform search for.

Returns
Iterator position associated with the start position of the segment containing the key, or end iterator position if the search has failed.

void **build_tree**()

Build a tree of non-leaf nodes based on the values stored in the leaf nodes. The tree must be valid before you can call the *search_tree()* method.

inline bool **is_tree_valid**() const

Returns
true if the tree is valid, otherwise false. The tree must be valid before you can call the *search_tree()* method.

bool **operator==**(const *flat_segment_tree* &other) const

Equality between two *flat_segment_tree* instances is evaluated by comparing the keys and the values of the leaf nodes only. Neither the non-leaf nodes nor the validity of the tree is evaluated.

inline bool **operator!=**(const *flat_segment_tree* &other) const

inline *key_type* **min_key**() const

inline *key_type* **max_key**() const

inline *value_type* **default_value**() const

size_type **leaf_size**() const

Return the number of leaf nodes.

Returns
number of leaf nodes.

Friends

friend struct ::mdds::fst::detail::forward_itr_handler< flat_segment_tree >

friend struct ::mdds::fst::detail::reverse_itr_handler< flat_segment_tree >

class **const_iterator** : public mdds::fst::detail::const_iterator_base<*flat_segment_tree*, ::mdds::fst::detail::forward_itr_handler<*flat_segment_tree*>>

Public Functions

inline **const_iterator**()

const_segment_iterator **to_segment**() const

Create a segment iterator that references the same segment the source iterator references the start key of.

class **const_reverse_iterator** : public mdds::fst::detail::const_iterator_base<*flat_segment_tree*, ::mdds::fst::detail::reverse_itr_handler<*flat_segment_tree*>>

Public Functions

inline **const_reverse_iterator**()

class **const_segment_range_type**

Public Functions

const_segment_range_type(*node_ptr* left_leaf, *node_ptr* right_leaf)

const_segment_iterator **begin**() const

const_segment_iterator **end**() const

struct **dispose_handler**

Public Functions

inline void **operator()**(*node*&)

inline void **operator()**(__st::nonleaf_node<*flat_segment_tree*>&)

struct **fill_nonleaf_value_handler**

Public Functions

inline void **operator()**(__st::nonleaf_node<*flat_segment_tree*> &_self, const __st::node_base *left_node, const __st::node_base *right_node)

struct **init_handler**

Public Functions

inline void **operator()**(*node*&)

inline void **operator()**(__st::nonleaf_node<*flat_segment_tree*>&)

struct **leaf_value_type**

Public Functions

inline bool **operator==**(const *leaf_value_type* &r) const

inline **leaf_value_type**()

Public Members

key_type **key**

value_type **value**

struct **nonleaf_value_type**

Public Functions

inline bool **operator==**(const *nonleaf_value_type* &r) const

high range value (non-inclusive)

inline **nonleaf_value_type**()

Public Members

key_type **low**

key_type **high**

low range value (inclusive)

CHAPTER

THREE

SEGMENT TREE

3.1 API Reference

template<typename **_Key**, typename **_Value**>

class **segment_tree**

Public Types

typedef *_Key* **key_type**

typedef *_Value* **value_type**

typedef size_t **size_type**

typedef std::vector<*value_type*> **search_results_type**

typedef ::std::vector<*value_type*> **data_chain_type**

typedef std::unordered_map<*value_type*, ::std::pair<*key_type*, *key_type*>> **segment_map_type**

typedef ::std::map<*value_type*, ::std::pair<*key_type*, *key_type*>> **sorted_segment_map_type**

typedef __st::node<*segment_tree*> **node**

typedef *node*::node_ptr **node_ptr**

typedef __st::nonleaf_node<*segment_tree*> **nonleaf_node**

Public Functions

segment_tree()

segment_tree(const *segment_tree* &r)

~segment_tree()

bool **operator==**(const *segment_tree* &r) const

Equality between two *segment_tree* instances is evaluated by comparing the segments that they store. The trees are not compared.

inline bool **operator!=**(const *segment_tree* &r) const

inline bool **is_tree_valid**() const

Check whether or not the internal tree is in a valid state. The tree must be valid in order to perform searches.

Returns
true if the tree is valid, false otherwise.

void **build_tree**()

Build or re-build tree based on the current set of segments.

bool **insert**(*key_type* begin_key, *key_type* end_key, *value_type* pdata)

Insert a new segment.

Parameters

- **begin_key** – begin point of the segment. The value is inclusive.
- **end_key** – end point of the segment. The value is non-inclusive.
- **pdata** – pointer to the data instance associated with this segment. Note that *the caller must manage the life cycle of the data instance*.

bool **search**(*key_type* point, *search_results_type* &result) const

Search the tree and collect all segments that include a specified point.

Parameters

- **point** – specified point value
- **result** – doubly-linked list of data instances associated with the segments that include the specified point. *Note that the search result gets appended to the list; the list will not get emptied on each search.* It is caller's responsibility to empty the list before passing it to this method in case the caller so desires.

Returns
true if the search is performed successfully, false if the search has ended prematurely due to error conditions.

search_results **search**(*key_type* point) const

Search the tree and collect all segments that include a specified point.

Parameters
point – specified point value

Returns
object containing the result of the search, which can be accessed via iterator.

void **remove**(*value_type* value)

Remove a segment that matches by the value. This will *not* invalidate the tree; however, if you have removed lots of segments, you might want to re-build the tree to shrink its size.

Parameters
value – value to remove a segment by.

void **clear**()

Remove all segments data.

size_t **size**() const

Return the number of segments currently stored in this container.

bool **empty**() const

Return whether or not the container stores any segments or none at all.

size_t **leaf_size**() const

Return the number of leaf nodes.

Returns
number of leaf nodes.

struct **dispose_handler**

Public Functions

inline void **operator()**(*node* &_self)

inline void **operator()**(__st::nonleaf_node<*segment_tree*> &_self)

struct **fill_nonleaf_value_handler**

Public Functions

inline void **operator()**(__st::nonleaf_node<*segment_tree*> &_self, const __st::node_base *left_node, const __st::node_base *right_node)

struct **init_handler**

Public Functions

inline void **operator()**(*node* &_self)

inline void **operator()**(__st::nonleaf_node<*segment_tree*> &_self)

struct **leaf_value_type**

Public Functions

inline bool **operator==**(const *leaf_value_type* &r) const

Public Members

key_type **key**

data_chain_type ***data_chain**

struct **nonleaf_value_type**

Public Functions

inline bool **operator==**(const *nonleaf_value_type* &r) const

Public Members

key_type **low**

key_type **high**

low range value (inclusive)

data_chain_type ***data_chain**

high range value (non-inclusive)

class **search_result_inserter**

Public Functions

inline **search_result_inserter**(search_results_base &result)

inline void **operator()**(*data_chain_type* *node_data)

class **search_result_vector_inserter**

Public Functions

inline **search_result_vector_inserter**(*search_results_type* &result)

inline void **operator()**(*data_chain_type* *node_data)

class **search_results** : public mdds::*segment_tree*<*_Key*, *_Value*>::search_results_base

Public Functions

inline *search_results*::*iterator* **begin**()

inline *search_results*::*iterator* **end**()

class **iterator** : public mdds::*segment_tree*<*_Key*, *_Value*>::iterator_base

Public Functions

inline **iterator**()

Friends

friend class segment_tree< _Key, _Value >::search_results

CHAPTER

FOUR

POINT QUAD TREE

4.1 API Reference

template<typename **_Key**, typename **_Value**>

class **point_quad_tree**

Public Types

typedef *_Key* **key_type**

typedef *_Value* **value_type**

typedef size_t **size_type**

typedef ::std::vector<*value_type*> **data_array_type**

Public Functions

point_quad_tree()

point_quad_tree(const *point_quad_tree* &r)

~point_quad_tree()

void **insert**(*key_type* x, *key_type* y, *value_type* data)

Insert a new data at specified coordinates. It overwrites existing data in case one exists at the specified coordinates.

Parameters

- **x** – x coordinate of new data position
- **y** – y coordinate of new data position
- **data** – data being inserted at the specified coordinates.

void **search_region**(*key_type* x1, *key_type* y1, *key_type* x2, *key_type* y2, *data_array_type* &result) const

Perform region search (aka window search), that is, find all points that fall within specified rectangular region. The boundaries are inclusive.

Parameters

- **x1** – left coordinate of the search region
- **y1** – top coordinate of the search region
- **x2** – right coordinate of the search region
- **y2** – bottom coordinate of the search region
- **result** – this array will contain all data found without specified region.

search_results **search_region**(*key_type* x1, *key_type* y1, *key_type* x2, *key_type* y2) const

Perform region search (aka window search), that is, find all points that fall within specified rectangular region. The boundaries are inclusive.

Parameters

- **x1** – left coordinate of the search region
- **y1** – top coordinate of the search region
- **x2** – right coordinate of the search region
- **y2** – bottom coordinate of the search region

Returns

search result object containing all data found within the specified region.

value_type **find**(*key_type* x, *key_type* y) const

Find data at specified coordinates. If no data exists at the specified coordinates, this method throws a *point_quad_tree::data_not_found* exception.

Parameters

- **x** – x coordinate
- **y** – y coordinate

Returns

data found at the specified coordinates.

void **remove**(*key_type* x, *key_type* y)

Remove data from specified coordinates. This method does nothing if no data exists at the specified coordinates.

Parameters

- **x** – x coordinate
- **y** – y coordinate

void **swap**(*point_quad_tree* &r)

Swap the internal state with another instance.

Parameters

r – another instance to swap internals with.

void **clear**()

Remove all stored data.

bool **empty**() const

Check whether or not the container is empty.

Returns
bool true if empty, false otherwise.

size_t **size**() const

Get the number of stored data.

Returns
the number of data currently stored in the container.

node_access **get_node_access**() const

Get read-only access to the internal quad node tree.

Returns
root node

point_quad_tree &**operator=**(const *point_quad_tree* &r)

bool **operator==**(const *point_quad_tree* &r) const

inline bool **operator!=**(const *point_quad_tree* &r) const

class **data_not_found** : public std::exception

class **node_access**

Node wrapper to allow read-only access to the internal quad node structure.

Public Functions

inline *node_access* **northeast**() const

inline *node_access* **northwest**() const

inline *node_access* **southeast**() const

inline *node_access* **southwest**() const

inline *value_type* **data**() const

inline *key_type* **x**() const

inline *key_type* **y**() const

inline **operator bool**() const

inline bool **operator==**(const *node_access* &r) const

inline **node_access**()

inline **~node_access**()

struct **point**

Public Functions

inline **point**(*key_type* _x, *key_type* _y)

inline **point**()

Public Members

key_type **x**

key_type **y**

class **search_results**

Public Functions

inline **search_results**()

inline **search_results**(const *search_results* &r)

inline *search_results*::*const_iterator* **begin**()

inline *search_results*::*const_iterator* **end**()

class **const_iterator**

Public Types

typedef std::pair<*point*, parent_value_type> **value_type**

typedef *value_type* ***pointer**

typedef *value_type* &**reference**

typedef ptrdiff_t **difference_type**

typedef ::std::bidirectional_iterator_tag **iterator_category**

Public Functions

inline **const_iterator**(res_nodes_ptr &ptr)

inline **const_iterator**(const *const_iterator* &r)

inline *const_iterator* &**operator=**(const *const_iterator* &r)

inline bool **operator==**(const *const_iterator* &r) const

inline bool **operator!=**(const *const_iterator* &r) const

inline const *value_type* &**operator***() const

inline const *value_type* ***operator->**() const

inline const *value_type* ***operator++**()

inline const *value_type* ***operator--**()

Friends

friend class point_quad_tree< _Key, _Value >::search_results

CHAPTER

FIVE

MULTI TYPE VECTOR

5.1 Examples

5.1.1 Quick start

The following code demonstrates a simple use case of storing values of double and `std::string` types in a single container using *multi_type_vector*.

```
#include <mdds/multi_type_vector.hpp>
#include <iostream>
#include <vector>
#include <string>

using std::cout;
using std::endl;

using mtv_type = mdds::multi_type_vector<mdds::mtv::standard_element_blocks_traits>;

template<typename BlockT>
void print_block(const mtv_type::value_type& v)
{
    for (const auto& elem : BlockT::range(*v.data))
    {
        cout << " * " << elem << endl;
    }
}

int main() try
{
    mtv_type con(20); // Initialized with 20 empty elements.

    // Set values individually.
    con.set(0, 1.1);
    con.set(1, 1.2);
    con.set(2, 1.3);

    // Set a sequence of values in one step.
    std::vector<double> vals = { 10.1, 10.2, 10.3, 10.4, 10.5 };
    con.set(3, vals.begin(), vals.end());
```

(continues on next page)

(continued from previous page)

```
    // Set string values.
    con.set<std::string>(10, "Andy");
    con.set<std::string>(11, "Bruce");
    con.set<std::string>(12, "Charlie");

    // Iterate through all blocks and print all elements.
    for (const auto& v : con)
    {
        switch (v.type)
        {
            case mdds::mtv::element_type_double:
            {
                cout << "numeric block of size " << v.size << endl;
                print_block<mdds::mtv::double_element_block>(v);
                break;
            }
            case mdds::mtv::element_type_string:
            {
                cout << "string block of size " << v.size << endl;
                print_block<mdds::mtv::string_element_block>(v);
                break;
            }
            case mdds::mtv::element_type_empty:
                cout << "empty block of size " << v.size << endl;
                cout << " - no data - " << endl;
            default:
                ;
        }
    }

    return EXIT_SUCCESS;
}
catch (...)
{
    return EXIT_FAILURE;
}
```

You'll see the following console output when you compile and execute this code:

```
numeric block of size 8
 * 1.1
 * 1.2
 * 1.3
 * 10.1
 * 10.2
 * 10.3
 * 10.4
 * 10.5
empty block of size 2
 - no data -
string block of size 3
 * Andy
```

(continues on next page)

(continued from previous page)

```
 * Bruce
 * Charlie
empty block of size 7
 - no data -
```

Fig. 1: Logical structure between the primary array, blocks, and element blocks.

Each multi_type_vector instance maintains a logical storage structure of one primary array containing one or more blocks each of which consists of `type`, `position`, `size` and `data` members:

- `type` - numeric value representing the block type.
- `position` - numeridc value representing the logical position of the first element of the block.
- `size` - number of elements present in the block a.k.a its logical size.
- `data` - pointer to the secondary storage (element block) storing the element values.

In this example code, the `type` member is referenced to determine its block type and its logical size is determined from the `size` member. For the numeric and string blocks, their `data` members, which should point to the memory addresses of their respective element blocks, are dereferenced in order to print out their element values to stdout inside the `print_block` function.

Standard element block types

It is worth noting that the two block types used in the previous example, namely *double_element_block* and *string_element_block* didn't come out of nowhere. By default, including the header that defines multi_type_vector implicitly also defines the following block types:

- *mdds::mtv::boolean_element_block*
- *mdds::mtv::int8_element_block*
- *mdds::mtv::uint8_element_block*
- *mdds::mtv::int16_element_block*
- *mdds::mtv::uint16_element_block*
- *mdds::mtv::int32_element_block*
- *mdds::mtv::uint32_element_block*
- *mdds::mtv::int64_element_block*
- *mdds::mtv::uint64_element_block*
- *mdds::mtv::float_element_block*
- *mdds::mtv::double_element_block*
- *mdds::mtv::string_element_block*

which respectively store elements of the following value types:

- `bool`
- `int8_t`
- `uint8_t`
- `int16_t`

- uint16_t
- int32_t
- uint32_t
- int64_t
- uint64_t
- float
- double
- std::string

The header also defines the *mdds::mtv::standard_element_blocks_traits* struct which you can pass to the *multi_type_vector* template definition in order to have all of the above mentioned block types and their respective value types available for use.

5.1.2 Specifying custom types in element blocks

There are times when you need to store a set of user-defined types in *multi_type_vector*. That is what we are going to talk about in this section.

First, let's include the header:

```
#include <mdds/multi_type_vector.hpp>
```

then proceed to define some constant values to use as element types. We are going to define three custom value types, so we need three element types defined:

```
constexpr mdds::mtv::element_t custom_value1_type = mdds::mtv::element_type_user_start;
constexpr mdds::mtv::element_t custom_value2_type = mdds::mtv::element_type_user_start +␣
↪1;
constexpr mdds::mtv::element_t custom_value3_type = mdds::mtv::element_type_user_start +␣
↪2;
```

Here, you need to ensure that the values used will not collide with the values that may be used for the standard value types. The best way to ensure that is to assign the values that are greater than or equal to *element_type_user_start* as the code above does. Values less than *element_type_user_start* are reserved for use either for the standard value types or any other internal uses in the future.

Now, let's define the first two custom value types, and their respective block types:

```
struct custom_value1 {};
struct custom_value2 {};

using custom_value1_block = mdds::mtv::default_element_block<custom_value1_type, custom_
↪value1>;
using custom_value2_block = mdds::mtv::default_element_block<custom_value2_type, custom_
↪value2>;
```

Here, we are using the *default_element_block* as the basis to define their block types. At minimum, you need to specify the element type constant and the value type as its template arguments. There is a third optional template argument you can specify which will become the underlying storage type. By default, *delayed_delete_vector* is used when the third argument is not given. But you can specify other types such as `std::vector` or `std::deque` instead, or any other types that have similar interfaces to `std::vector`.

Once the block types are defined, it's time to define callback functions for them. This should be as simple as using the *MDDS_MTV_DEFINE_ELEMENT_CALLBACKS* with all necessary parameters provided:

```
MDDS_MTV_DEFINE_ELEMENT_CALLBACKS(custom_value1, custom_value1_type, custom_value1{},␣
↪custom_value1_block)
MDDS_MTV_DEFINE_ELEMENT_CALLBACKS(custom_value2, custom_value2_type, custom_value2{},␣
↪custom_value2_block)
```

Our third type is defined in a namespace `ns`, and its associated block type is also defined in the same namespace. One thing to keep in mind is that, when the custom type is defined in a namespace, its callback functions must also be defined in the same namespace in order for them to be discovered per argument dependent lookup during overload resolution. This means that you must place the macro that defines the callback functions in the same namespace as the namespace that encompasses the original value type:

```
namespace ns {

struct custom_value3 {};

using custom_value3_block = mdds::mtv::default_element_block<custom_value3_type, custom_
↪value3>;

// This macro MUST be in the same namespace as that of the value type, in order for
// argument-dependent lookup to work properly during overload resolution.
MDDS_MTV_DEFINE_ELEMENT_CALLBACKS(custom_value3, custom_value2_type, custom_value3{},␣
↪custom_value3_block)

} // namespace ns
```

Warning: If the original value type is defined inside a namespace, its associated callback functions must also be defined in the same namespace, due to the way argument dependent lookup works during overload resolution.

The next step is to define a trait type that specifies these block types. The easiest way is to have your trait inherit from *mdds::mtv::default_traits* and overwrite the *block_funcs* static member type with an instance of *mdds::mtv::element_block_funcs* with one or more block types specified as its template arguments:

```
struct my_custom_traits : public mdds::mtv::default_traits
{
    using block_funcs = mdds::mtv::element_block_funcs<
        custom_value1_block, custom_value2_block, ns::custom_value3_block>;
};
```

Now we are ready to define the final `multi_type_vector` type with the trait we just defined:

```
using mtv_type = mdds::multi_type_vector<my_custom_traits>;
```

And that's it! With this in place, you can write a code like the following:

```
mtv_type con{}; // initialize it as empty container.

// Push three values of different types to the end.
con.push_back(custom_value1{});
con.push_back(custom_value2{});
```

(continues on next page)

(continued from previous page)

```
con.push_back(ns::custom_value3{});

auto v1 = con.get<custom_value1>(0);
auto v2 = con.get<custom_value2>(1);
auto v3 = con.get<ns::custom_value3>(2);

std::cout << "is this custom_value1? " << std::is_same_v<decltype(v1), custom_value1> <<␣
↪std::endl;
std::cout << "is this custom_value2? " << std::is_same_v<decltype(v2), custom_value2> <<␣
↪std::endl;
std::cout << "is this ns::custom_value3? " << std::is_same_v<decltype(v3), ns::custom_
↪value3> << std::endl;
```

to put some values of the custom types into your container and accessing them. This code should generate the following output:

```
is this custom_value1? 1
is this custom_value2? 1
is this ns::custom_value3? 1
```

5.1.3 Specifying different storage type

By default, *mdds::mtv::default_element_block* uses *mdds::mtv::delayed_delete_vector* as its underlying storage type to store its elements starting with version 2.1. Prior to 2.1, `std::vector` was used as the only storage type of choice. If you use 2.1 or newer versions of the library, you can specify your own storage type as the third template argument to *mdds::mtv::default_element_block*.

Let's tweak the previous example to specify `std::vector` and `std::deque` as the storage types for `custom_value1_block` and `custom_value2_block`, respectively:

```
using custom_value1_block = mdds::mtv::default_element_block<
    custom_value1_type, custom_value1, std::vector>;

using custom_value2_block = mdds::mtv::default_element_block<
    custom_value2_type, custom_value2, std::deque>;
```

For `custom_value3_block`, we will leave it as the default storage type, namely, *mdds::mtv::delayed_delete_vector*:

```
// This implicitly uses mdds::mtv::delayed_delete_vector.
using custom_value3_block = mdds::mtv::default_element_block<
    custom_value3_type, custom_value3>;
```

You can specify different storage types for different block types as you can see above. But unless you have a good reason to do so, you may want to stick with the same storage type for all of your blocks in order to have consistent performance characteristics.

With this now in place, let's run the following code:

```
std::cout << "custom_value1 stored in std::vector? "
    << std::is_same_v<custom_value1_block::store_type, std::vector<custom_value1>>
    << std::endl;
```

(continues on next page)

(continued from previous page)

```
std::cout << "custom_value2 stored in std::deque? "
    << std::is_same_v<custom_value2_block::store_type, std::deque<custom_value2>>
    << std::endl;

std::cout << "ns::custom_value3 stored in delayed_delete_vector? "
    << std::is_same_v<ns::custom_value3_block::store_type, mdds::mtv::delayed_delete_
↪vector<ns::custom_value3>>
    << std::endl;
```

which should generate the following output:

```
custom_value1 stored in std::vector? 1
custom_value2 stored in std::deque? 1
ns::custom_value3 stored in delayed_delete_vector? 1
```

One thing to note is that, in order for a class to be usable as the storage type for *default_element_block*, it must be a template class with two parameters: the first one being the value type while the second one is the allocator type just like how `std::vector` or `std::deque` are defined.

5.1.4 Different storage types in standard element blocks

Now, what if you need to specify different storage types in the blocks already defined for the standard value types, given that, as explained in the *Standard element block types* section, those standard element blocks are automagically defined?

The answer is that it is possible to do such a thing, but it will require that you follow a certain set of steps, as outlined below:

First, manually define the element type constants, block types, and their respecitve callback functions for the standard value types you need to use as if they were user-defined types. When doing so, specify the non-default storage types you need to use for these blocks.

Include the header for the multi_type_vector definition with the special macro value named `MDDS_MTV_USE_STANDARD_ELEMENT_BLOCKS` defined and its value is set to 0. This bypasses the automatic inclusion of the block types for the standard value types when this header is included.

Lastly, define a custom trait type and overwrite the `block_funcs` member type to specify the block types defined in the first step. This is essentially the same step you would take when you define custom block types for user-defined value types.

Let's do this step-by-step. First, include the necessary headers:

```
#include <mdds/multi_type_vector/types.hpp>
#include <mdds/multi_type_vector/macro.hpp>

#include <deque>
```

The `types.hpp` header is required for the *element_t* and *default_element_block*, and the `macro.hpp` header is required for the *MDDS_MTV_DEFINE_ELEMENT_CALLBACKS* macro. The `<deque>` header is so that we can use `std::deque` as storage types in our block types.

Next, let's define the element and block types as well as their callback functions:

```
// Define element ID's for the standard element types.
constexpr mdds::mtv::element_t my_doube_type_id = mdds::mtv::element_type_user_start;
constexpr mdds::mtv::element_t my_int32_type_id = mdds::mtv::element_type_user_start + 1;

// Define the block types.
using my_double_block = mdds::mtv::default_element_block<my_doube_type_id, double,␣
↪std::deque>;
using my_int32_block = mdds::mtv::default_element_block<my_int32_type_id, std::int32_t,␣
↪std::deque>;

MDDS_MTV_DEFINE_ELEMENT_CALLBACKS(double, my_doube_type_id, 0.0, my_double_block)
MDDS_MTV_DEFINE_ELEMENT_CALLBACKS(std::int32_t, my_int32_type_id, 0, my_int32_block)
```

This is very similar to how it is done in the *Specifying different storage type* section. The only difference is that, this part needs to happen *before* the header for the multi_type_vector type gets included, in order for the multi_type_vector implementation code to reference the callback functions now that the callback functions for the standard value types will no longer be included.

Let's proceed to include the multi_type_vector header:

```
#define MDDS_MTV_USE_STANDARD_ELEMENT_BLOCKS 0
#include <mdds/multi_type_vector/soa/main.hpp>
```

Here, we define the `MDDS_MTV_USE_STANDARD_ELEMENT_BLOCKS` macro and set its value to 0, to skip the inclusion of the standard element blocks. It is also worth noting that we are including the `mdds/multi_type_vector/soa/main.hpp` header directly instead of `mdds/multi_type_vector.hpp`, which indirectly includes the first header.

Lastly, let's define the trait type to specify the block types to use, and instantiate the final multi_type_vector type:

```
struct my_custom_traits : public mdds::mtv::default_traits
{
    using block_funcs = mdds::mtv::element_block_funcs<my_double_block, my_int32_block>;
};

using mtv_type = mdds::mtv::soa::multi_type_vector<my_custom_traits>;
```

Now that the concrete multi_type_vector is defined, we can use it to store some values of the specified types:

```
mtv_type con(20); // Initialized with 20 empty elements.

con.set<std::int32_t>(0, 234);
con.set<double>(1, 425.1);
```

If you inspect the storage types of the element blocks like the following:

```
std::cout << "my_double_block: is std::deque its store type? "
    << std::is_same_v<my_double_block::store_type, std::deque<double>>
    << std::endl;
std::cout << "my_int32_block: is std::deque its store type? "
    << std::is_same_v<my_int32_block::store_type, std::deque<std::int32_t>>
    << std::endl;
```

you should see the following output:

```
my_double_block: is std::deque its store type? 1
my_int32_block: is std::deque its store type? 1
```

which indicates that they are indeed `std::deque`.

5.1.5 Use custom event handlers

It is also possible to define custom event handlers that get called when certain events take place. To define custom event handlers, you need to define either a class or a struct that has the following methods:

- **void element_block_acquired(mdds::mtv::base_element_block* block)**
- **void element_block_released(mdds::mtv::base_element_block* block)**

as its public methods, specify it as type named `event_func` in a trait struct, and pass it as the second template argument when instantiating your *multi_type_vector* type. Refer to *mdds::mtv::empty_event_func* for the detail on when each event handler method gets triggered.

The following code example demonstrates how this all works:

```
#include <mdds/multi_type_vector.hpp>
#include <iostream>

using std::cout;
using std::endl;

class event_hdl
{
public:
    void element_block_acquired(mdds::mtv::base_element_block* block)
    {
        (void)block;
        cout << "  * element block acquired" << endl;
    }

    void element_block_released(mdds::mtv::base_element_block* block)
    {
        (void)block;
        cout << "  * element block released" << endl;
    }
};

struct trait : mdds::mtv::standard_element_blocks_traits
{
    using event_func = event_hdl;
};

using mtv_type = mdds::multi_type_vector<trait>;

int main() try
{
    mtv_type db;  // starts with an empty container.

    cout << "inserting string 'foo'..." << endl;
```

(continues on next page)

(continued from previous page)

```
    db.push_back(std::string("foo"));  // creates a new string element block.

    cout << "inserting string 'bah'..." << endl;
    db.push_back(std::string("bah"));  // appends to an existing string block.

    cout << "inserting int 100..." << endl;
    db.push_back(int(100)); // creates a new int element block.

    cout << "emptying the container..." << endl;
    db.clear(); // releases both the string and int element blocks.

    cout << "exiting program..." << endl;

    return EXIT_SUCCESS;
}
catch (...)
{
    return EXIT_FAILURE;
}
```

You'll see the following console output when you compile and execute this code:

```
inserting string 'foo'...
  * element block acquired
inserting string 'bah'...
inserting int 100...
  * element block acquired
emptying the container...
  * element block released
  * element block released
exiting program...
```

In this example, the **element_block_acquired** handler gets triggered each time the container creates (thus acquires) a new element block to store a value. It does *not* get called when a new value is appended to a pre-existing element block. Similarly, the **element_block_releasd** handler gets triggered each time an existing element block storing non-empty values gets deleted. One thing to keep in mind is that since these two handlers respond to events related to element blocks which are owned by non-empty blocks in the primary array, and empty blocks don't store any element block instances, creations or deletions of empty blocks don't trigger these event handlers.

The trait also allows you to configure other behaviors of *multi_type_vector*. Refer to *mdds::mtv::default_traits* for all available parameters.

5.1.6 Get raw pointer to element block array

Sometimes you need to expose a pointer to an element block array especially when you need to pass such an array pointer to C API that requires one. You can do this by calling the `data` method of the element_block template class. This works since the element block internally just wraps `std::vector` or one that acts like it, such as `std::deque` or *delayed_delete_vector*, and its `data` method simply exposes the internal storage types's own `data` method which returns the memory location of its internal buffer.

The following code demonstrates this by exposing raw array pointers to the internal arrays of numeric and string element blocks, and printing their element values directly from these array pointers.

```cpp
#include <mdds/multi_type_vector.hpp>
#include <iostream>

using std::cout;
using std::endl;
using mdds::mtv::double_element_block;
using mdds::mtv::string_element_block;

using mtv_type = mdds::multi_type_vector<mdds::mtv::standard_element_blocks_traits>;

int main() try
{
    mtv_type db;  // starts with an empty container.

    db.push_back(1.1);
    db.push_back(1.2);
    db.push_back(1.3);
    db.push_back(1.4);
    db.push_back(1.5);

    db.push_back(std::string("A"));
    db.push_back(std::string("B"));
    db.push_back(std::string("C"));
    db.push_back(std::string("D"));
    db.push_back(std::string("E"));

    // At this point, you have 2 blocks in the container.
    cout << "block size: " << db.block_size() << endl;
    cout << "--" << endl;

    // Get an iterator that points to the first block in the primary array.
    mtv_type::const_iterator it = db.begin();

    // Get a pointer to the raw array of the numeric element block using the
    // 'data' method.
    const double* p = double_element_block::data(*it->data);

    // Print the elements from this raw array pointer.
    for (const double* p_end = p + it->size; p != p_end; ++p)
        cout << *p << endl;

    cout << "--" << endl;

    ++it; // move to the next block, which is a string block.

    // Get a pointer to the raw array of the string element block.
    const std::string* pz = string_element_block::data(*it->data);

    // Print out the string elements.
    for (const std::string* pz_end = pz + it->size; pz != pz_end; ++pz)
        cout << *pz << endl;

    return EXIT_SUCCESS;
```

(continues on next page)

(continued from previous page)

```
}
catch (...)
{
    return EXIT_FAILURE;
}
```

Compiling and execute this code produces the following output:

```
block size: 2
--
1.1
1.2
1.3
1.4
1.5
--
A
B
C
D
E
```

5.1.7 Traverse multiple multi_type_vector instances "sideways"

In this section we will demonstrate a way to traverse multiple instances of *multi_type_vector* "sideways" using the *mdds::mtv::collection* class. What this class does is to wrap multiple instances of *multi_type_vector* and generate iterators that let you iterate the individual element values collectively in the direction orthogonal to the direction of the individual vector instances.

The best way to explain this feature is to use a spreadsheet analogy. Let's say we are implementing a data store to store a 2-dimensional tabular data where each cell in the data set is associated with row and column indices. Each cell may store a value of string type, integer type, numeric type, etc. And let's say that the data looks like the following spreadsheet data:

It consists of five columns, with each column storing 21 rows of data. The first row is a header row, followed by 20 rows of values. In this example, We will be using one *multi_type_vector* instance for each column thus creating five instances in total, and store them in a `std::vector` container.

The declaration of the data store will look like this:

```
using mtv_type = mdds::multi_type_vector<mdds::mtv::standard_element_blocks_traits>;
using collection_type = mdds::mtv::collection<mtv_type>;

std::vector<mtv_type> columns(5);
```

The first two lines specify the concrete *multi_type_vector* type used for each individual column and the collection type that wraps the columns. The third line instantiates the `std::vector` instance to store the columns, and we are setting its size to five to accommodate for five columns. We will make use of the collection_type later in this example after the columns have been populated.

Now, we need to populate the columns with values. First, we are setting the header row:

	A	B	C	D	E
1	ID	Make	Model	Year	Color
2	1	Nissan	Frontier	1998	Turquoise
3	2	Mercedes-Benz	W201	1986	Fuscia
4	3	Nissan	Frontier	2009	Teal
5	4	Suzuki	Equator	unknown	Fuscia
6	5	Saab	9-5	unknown	Green
7	6	Subaru	Tribeca	2008	Khaki
8	7	GMC	Yukon XL 2500	2009	Pink
9	8	Mercedes-Benz	E-Class	2008	Goldenrod
10	9	Toyota	Camry Hybrid	2010	Turquoise
11	10	Nissan	Frontier	2001	Yellow
12	11	Mazda	MX-5	2008	Orange
13	12	Dodge	Ram Van 1500	2000	Goldenrod
14	13	Ford	Edge	unknown	Fuscia
15	14	Bentley	Azure	2009	Goldenrod
16	15	GMC	Sonoma Club Coupe	1998	Mauv
17	16	Audi	S4	2013	Crimson
18	17	GMC	3500 Club Coupe	1994	Turquoise
19	18	Mercury	Villager	2000	Teal
20	19	Pontiac	Sunbird	1990	Indigo
21	20	BMW	3 Series	1993	Khaki

```
// Populate the header row.
const char* headers[] = { "ID", "Make", "Model", "Year", "Color" };
size_t i = 0;
for (const char* v : headers)
    columns[i++].push_back<std::string>(v);
```

We are then filling each column individually from column 1 through column 5. First up is column 1:

```
// Fill column 1.
int c1_values[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,␣
↪20 };

for (int v : c1_values)
    columns[0].push_back(v);
```

Hopefully this code is straight-forward. It initializes an array of values and push them to the column one at a time via *push_back()*. Next up is column 2:

```
// Fill column 2.
const char* c2_values[] =
{
    "Nissan", "Mercedes-Benz", "Nissan", "Suzuki", "Saab", "Subaru", "GMC", "Mercedes-
↪Benz", "Toyota", "Nissan",
    "Mazda", "Dodge", "Ford", "Bentley", "GMC", "Audi", "GMC", "Mercury", "Pontiac", "BMW
↪",
};
```

(continues on next page)

(continued from previous page)

```
for (const char* v : c2_values)
    columns[1].push_back<std::string>(v);
```

This is similar to the code for column 1, except that because we are using an array of string literals which implicitly becomes an initializer list of type `const char*`, we need to explicitly specify the type for the *push_back()* call to be `std::string`.

The code for column 3 is very similar to this:

```
// Fill column 3.
const char* c3_values[] =
{
    "Frontier", "W201", "Frontier", "Equator", "9-5", "Tribeca", "Yukon XL 2500", "E-
↪Class", "Camry Hybrid", "Frontier",
    "MX-5", "Ram Van 1500", "Edge", "Azure", "Sonoma Club Coupe", "S4", "3500 Club Coupe
↪", "Villager", "Sunbird", "3 Series",
};

for (const char* v : c3_values)
    columns[2].push_back<std::string>(v);
```

Populating column 4 needs slight pre-processing. We are inserting a string value of "unknown" in lieu of an integer value of -1. Therefore the following code will do:

```
// Fill column 4.  Replace -1 with "unknown".
int32_t c4_values[] =
{
    1998, 1986, 2009, -1, -1, 2008, 2009, 2008, 2010, 2001,
    2008, 2000, -1, 2009, 1998, 2013, 1994, 2000, 1990, 1993,
};

for (int32_t v : c4_values)
{
    if (v < 0)
        // Insert a string value "unknown".
        columns[3].push_back<std::string>("unknown");
    else
        columns[3].push_back(v);
}
```

Finally, the last column to fill, which uses the same logic as for columns 2 and 3:

```
// Fill column 5
const char* c5_values[] =
{
    "Turquoise", "Fuscia", "Teal", "Fuscia", "Green", "Khaki", "Pink", "Goldenrod",
↪"Turquoise", "Yellow",
    "Orange", "Goldenrod", "Fuscia", "Goldenrod", "Mauv", "Crimson", "Turquoise", "Teal",
↪ "Indigo", "LKhaki",
};

for (const char* v : c5_values)
    columns[4].push_back<std::string>(v);
```

At this point, the content we've put into the `columns` variable roughly reflects the tabular data shown at the beginning of this section. Now we can use the collection type we've declared earlier to wrap the columns:

```
// Wrap the columns with the 'collection'...
collection_type rows(columns.begin(), columns.end());
```

We are naming this variable `rows` since what we are doing with this wrapper is to traverse the content of the tabular data in row-wise direction. For this reason, calling it `rows` is quite fitting.

The *collection* class offers some flexibility as to how the instances that you are trying to traverse orthogonally are stored. That being said, you must meet the following prerequisites when passing the collection of vector instances to the constructor of the *collection* class:

1. All *multi_type_vector* instances that comprise the collection must be of the same logical length i.e. their *size()* methods must all return the same value.
2. The instances in the collection must be stored in the source container either as
   - concrete instances (as in this example),
   - as pointers, or
   - as heap instances wrapped within smart pointer class such as `std::shared_ptr` or `std::unique_ptr`.

Although we are storing the vector instances in a `std::vector` container in this example, you have the flexibility to pick a different type of container to store the individual vector instances as long as it provides STL-compatible standard iterator functionality.

Additionally, when using the *collection* class, you must ensure that the content of the vector instances that it references will not change for the duration of its use.

Finally, here is the code that does the traversing:

```
// Traverse the tabular data in row-wise direction.
for (const auto& cell : rows)
{
    if (cell.index > 0)
        // Insert a column separator before each cell except for the ones in the first␣
↪column.
        std::cout << " | ";

    switch (cell.type)
    {
        // In this example, we use two element types only.
        case mdds::mtv::element_type_int32:
            std::cout << cell.get<mdds::mtv::int32_element_block>();
            break;
        case mdds::mtv::element_type_string:
            std::cout << cell.get<mdds::mtv::string_element_block>();
            break;
        default:
            std::cout << "???"; // The default case should not hit in this example.
    }

    if (cell.index == 4)
        // We are in the last column. Insert a line break.
        std::cout << std::endl;
}
```

It's a simple for-loop, and in each iteration you get a single cell node that contains metadata about that cell including its value. The node contains the following members:

- `type` - an integer value representing the type of the value.
- `index` - a 0-based index of the *multi_type_vector* instance within the collection. You can think of this as column index in this example.
- `position` - a 0-based logical element position within each *multi_type_vector* instance. You can think of this as row index in this example.

In the current example we are only making use of the `type` and `index` members, but the `position` member will be there if you need it.

The node also provides a convenient `get()` method to fetch the value of the cell. This method is a template method, and you need to explicitly specify the element block type in order to access the value.

When executing this code, you will see the following outout:

```
ID | Make | Model | Year | Color
1 | Nissan | Frontier | 1998 | Turquoise
2 | Mercedes-Benz | W201 | 1986 | Fuscia
3 | Nissan | Frontier | 2009 | Teal
4 | Suzuki | Equator | unknown | Fuscia
5 | Saab | 9-5 | unknown | Green
6 | Subaru | Tribeca | 2008 | Khaki
7 | GMC | Yukon XL 2500 | 2009 | Pink
8 | Mercedes-Benz | E-Class | 2008 | Goldenrod
9 | Toyota | Camry Hybrid | 2010 | Turquoise
10 | Nissan | Frontier | 2001 | Yellow
11 | Mazda | MX-5 | 2008 | Orange
12 | Dodge | Ram Van 1500 | 2000 | Goldenrod
13 | Ford | Edge | unknown | Fuscia
14 | Bentley | Azure | 2009 | Goldenrod
15 | GMC | Sonoma Club Coupe | 1998 | Mauv
16 | Audi | S4 | 2013 | Crimson
17 | GMC | 3500 Club Coupe | 1994 | Turquoise
18 | Mercury | Villager | 2000 | Teal
19 | Pontiac | Sunbird | 1990 | Indigo
20 | BMW | 3 Series | 1993 | LKhaki
```

which clearly shows that the code has traversed the content of the tabular data horizontally across columns as intended.

Now, one feature that may come in handy is the ability to limit the iteration range within the collection. You can do that by calling either *set_collection_range()* to limit the column range or *set_element_range()* to limit the row range, or perhaps both.

Let's see how this works in the current example. Here, we are going to limit the iteration range to only columns 2 and 3, and rows 2 through 11. The following code will set this limit:

```
rows.set_collection_range(1, 2); // only columns 2 and 3.
rows.set_element_range(1, 10);   // only rows 2 through 11.
```

Then iterate through the collection once again:

```
for (const auto& cell : rows)
{
    if (cell.index > 1)
```

(continues on next page)

(continued from previous page)

```
        // Insert a column separator before each cell except for the ones in the first␣
↪column.
        std::cout << " | ";

    switch (cell.type)
    {
        // In this example, we use two element types only.
        case mdds::mtv::element_type_int32:
            std::cout << cell.get<mdds::mtv::int32_element_block>();
            break;
        case mdds::mtv::element_type_string:
            std::cout << cell.get<mdds::mtv::string_element_block>();
            break;
        default:
            std::cout << "???"; // The default case should not hit in this example.
    }

    if (cell.index == 2)
        // We are in the last column. Insert a line break.
        std::cout << std::endl;
}
```

This code is nearly identical to the previous one except for the index values used to control when to insert column separators and line breaks at the top and bottom of each iteration. When executing this code, you'll see the following output:

```
Nissan | Frontier
Mercedes-Benz | W201
Nissan | Frontier
Suzuki | Equator
Saab | 9-5
Subaru | Tribeca
GMC | Yukon XL 2500
Mercedes-Benz | E-Class
Toyota | Camry Hybrid
Nissan | Frontier
```

which clearly shows that your iteration range did indeed shrink as expected.

5.2 Performance Considerations

5.2.1 Select SoA or AoS storage types

If you instantiate a multi_type_vector instance via *mdds::multi_type_vector*, which is an alias type for *mdds::mtv::soa::multi_type_vector*, you will be using the structure-of-arrays (SoA) variant of its implementation which is new in 2.0. Prior to 2.0, multi_type_vector used the array-of-structures (AoS) layout which is still available post 2.0 via *mdds::mtv::aos::multi_type_vector* in case you need it.

Note, however, that the SoA variant generally yields better overall performance since it can make more efficient use of CPU caches. It is therefore highly recommended that you stick with the SoA variant unless you have a specific reason not to.

Also note that both variants are API compatibile with each other.

5.2.2 Use of position hints to avoid the cost of block position lookup

Consider the following example code:

```
using mtv_type = mdds::multi_type_vector<mdds::mtv::standard_element_blocks_traits>;

size_t size = 50000;

// Initialize the container with one empty block of size 50000.
mtv_type db(size);

// Set non-empty value at every other logical position from top down.
for (size_t i = 0; i < size; ++i)
{
    if (i % 2)
        db.set<double>(i, 1.0);
}
```

which, when executed, may take quite sometime to complete especially when you are using an older version of mdds. This particular example exposes one weakness that multi_type_vector has; because it needs to first look up the position of the block to operate with, and that lookup *always* starts from the first block, the time it takes to find the correct block increases as the number of blocks goes up. This example demonstrates the worst case scenario of such lookup complexity since it always inserts the next value at the last block position.

Fortunately, there is a simple solution to this which the following code demonstrates:

```
using mtv_type = mdds::multi_type_vector<mdds::mtv::standard_element_blocks_traits>;

size_t size = 50000;

// Initialize the container with one empty block of size 50000.
mtv_type db(size);
mtv_type::iterator pos = db.begin();

// Set non-empty value at every other logical position from top down.
for (size_t i = 0; i < size; ++i)
{
    if (i % 2)
        // Pass the position hint as the first argument, and receive a new
        // one returned from the method for the next call.
        pos = db.set<double>(pos, i, 1.0);
}
```

Compiling and executing this code should take only a fraction of a second.

The only difference between the second example and the first one is that the second one uses an interator as a position hint to keep track of the position of the last modified block. Each `set()` method call returns an iterator which can then be passed to the next `set()` call as the position hint. Because an iterator object internally stores the location of the block the value was inserted to, this lets the method to start the block position lookup process from the last modified block, which in this example is always one block behind the one the new value needs to go. Using the big-O notation, the use of the position hint essentially turns the complexity of O(n^2) in the first example into O(1) in the second one if you are using an older version of mdds where the block position lookup had a linear complexity.

This strategy should work with any methods in *multi_type_vector* that take a position hint as the first argument.

Note that, if you are using a more recent version of mdds (1.6.0 or newer), the cost of block position lookup is significantly lessoned thanks to the switch to binary search in performing the lookup.

Note: If you are using mdds 1.6.0 or newer, the cost of block position lookup is much less significant even without the use of position hints. But the benefit of using position hints may still be there. It's always a good idea to profile your specific use case and decide whether the use of position hints is worth it.

One important thing to note is that, as a user, you must ensure that the position hint you pass stays valid between the calls. A position hint becomes invalid when the content of the container changes. A good strategy to maintain a valid position hint is to always receive the iterator returned from the mutator method you called to which you passed the previous position hint, which is what the code above does. Passing an invalid position hint to a method that takes one may result in invalid memory access or otherwise in some sort of undefined behavior.

Warning: You must ensure that the position hint you pass stays valid. Passing an invalid position hint to a method that takes one may result in invalid memory access or otherwise in some sort of undefined behavior.

5.2.3 Block shifting performance and loop-unrolling factor

The introduction of binary search in the block position lookup implementation in version 1.6 has significantly improved its lookup performance, but has also resulted in slight performance hit when shifting blocks during value insertion. This is because when shifting the logical positions of the blocks below the insertion point, their head positions need to be re-calculated to account for their new positions.

The good news is that the switch to the structure-of-arrays (SoA) storage layout in 2.0 alone may bring subtle but measurable improvement in the block position adjustment performance due to the logical block positions now being stored in a separate array thereby improving its cache efficiency. In reality, however, this was somewhat dependent on the CPU types since some CPU's didn't show any noticeable improvements or even showed worse performance, while other CPU types showed consistent improvements with SoA over AoS.

Another factor that may play a role is loop unrolling factor which can be configured via the *loop_unrolling* variable in your custom trait type if you use version 2.0 or newer. This variable is an enum class of type *mdds::mtv::lu_factor_t* which enumerates several pre-defined loop-unrolling factors as well as some SIMD features.

The hardest part is to figure out which loop unrolling factor is the best option in your runtime environment, since it is highly dependent on the environment. Luckily mdds comes with a tool called runtime-env which, when run, will perform some benchmarks and give you the best loop-unrolling factor in your runtime environment. Be sure to build this tool with the same compiler and compiler flags as your target program in order for this tool to give you a representative answer.

5.3 Debugging

5.3.1 Tracing of public methods

When using *multi_type_vector* to handle a series of data reads and writes in an non-trivial code base, sometimes you may find yourself needing to track which methods are getting called when following a certain code path during a debugging session. In such a situation, you can enable an optional trace method which gets called whenever a public method of *multi_type_vector* is called.

First, you need to define a preprocessor macro named `MDDS_MULTI_TYPE_VECTOR_DEBUG` before including the header for *multi_type_vector*:

```
#define MDDS_MULTI_TYPE_VECTOR_DEBUG 1
#include <mdds/multi_type_vector/soa/main.hpp>

#include <iostream>
```

to enable additional debug code. In this example the value of the macro is set to 1, but it doesn't matter what the value of the macro is, as long as it is defined. You can also define one as a compiler option as well.

Once defined, the next step is to add a `trace` method as a static function to the trait type you pass as a template argument of multi_type_vector:

```
namespace mtv = mdds::mtv;

struct mtv_traits : public mtv::standard_element_blocks_traits
{
    static void trace(const mtv::trace_method_properties_t& props)
    {
        std::cout << "function:" << std::endl
                  << "  name: " << props.function_name << std::endl
                  << "  args: " << props.function_args << std::endl;
    }
};

using mtv_type = mtv::soa::multi_type_vector<mtv_traits>;
```

Here, we are simply inheriting our trait type from the *default_traits* type and simply adding a static `trace` function to it, and passing this trait type to the mtv_type definition below. This trace function must take one argument of type *mdds::mtv::trace_method_properties_t* which includes various properties of the traced call. In this example, we are simply printing the properties named *function_name* and *function_args* each time a traced method is called. Both of these properties are printable string types.

Note that this `trace` function is entirely optional; the code will compile fine even when it's not defined. Also, it must be declared as static for it to be called.

Let's instantiate an object of `mtv_type`, call some of its methods and see what happens. When executing the following code:

```
mtv_type db(10);
db.set<int32_t>(0, 12);
db.set<int8_t>(2, 34);
db.set<int16_t>(4, 56);
```

You will see the following output:

```
function:
  name: multi_type_vector
  args: init_size=10
function:
  name: set
  args: pos=0; value=? (type=5)
function:
  name: set
```

(continues on next page)

(continued from previous page)

```
  args: pos=2; value=? (type=1)
function:
  name: set
  args: pos=4; value=? (type=3)
function:
  name: ~multi_type_vector
  args:
```

The *function_name* property is hopefully self-explanatory. The *function_args* property is a single string value containing the information about the function's arguments and optionally their values if their values are known to be printable. If the value of an argument cannot be printed, ? is placed instead. For some argument types, an additional information is displayed e.g. `(type=5)` in the above output which indicates that the type of the value being passed to the function is *element_type_int32*.

If you want to limit your tracing to a specific function type or types, you can make use of the *type* property which specifies the type of the traced method. Likewise, if you want to only trace methods of a certain instance, use *instance* to filter the incoming trace calls based on the memory addresses of the instances whose methods are being traced.

Note that this feature is available for version 2.0.2 and newer, and currently only available for the SoA variant of *multi_type_vector*.

Note: This feature is only available for version 2.0.2 and newer, and only for the SoA variant.

5.4 API Reference

5.4.1 Core

mdds::multi_type_vector

template<typename **Traits** = mtv::*default_traits*>

using `mdds::`**multi_type_vector** = mtv::soa::*multi_type_vector*<*Traits*>

Type alias for the concrete implementation to maintain backward API compatibility.

mdds::mtv::soa::multi_type_vector

template<typename **Traits** = mdds::mtv::*default_traits*>

class **multi_type_vector**

Multi-type vector consists of a series of one or more blocks, and each block may either be empty, or stores a series of non-empty elements of identical type. These blocks collectively represent a single logical one-dimensional array that may store elements of different types. It is guaranteed that the block types of neighboring blocks are always different.

Structurally, the primary array stores block instances whose types are of `value_type`, which in turn consists of the following data members:

- `type` which indicates the block type,

- `position` which stores the logical position of the first element of the block,
- `size` which stores the logical size of the block, and
- `data` which stores the pointer to a secondary array (a.k.a. element block) which stores the actual element values, or `nullptr` in case the block represents an empty segment.

This variant implements a structure-of-arrays (SoA) storage.

See also:

mdds::mtv::soa::multi_type_vector::value_type

Public Types

using **size_type** = std::size_t

using **element_block_type** = mdds::mtv::*base_element_block*

using **element_category_type** = mdds::mtv::*element_t*

using **block_funcs** = typename *Traits*::block_funcs

using **event_func** = typename *Traits*::event_func

Optional event handler function structure, whose functions get called at specific events. The following events are currently supported:

- `element_block_acquired` - this gets called whenever the container acquires a new element block either as a result of a new element block creation or a tranfer of an existing element block from another container.
- `element_block_released` - this gets called whenever the container releases an existing element block either because the block gets deleted or gets transferred to another container.

See also:

mdds::mtv::empty_event_func for the precise function signatures of the event handler functions.

using **iterator** = detail::iterator_base<iterator_trait>

using **const_iterator** = detail::const_iterator_base<const_iterator_trait, *iterator*>

using **reverse_iterator** = detail::iterator_base<reverse_iterator_trait>

using **const_reverse_iterator** = detail::const_iterator_base<const_reverse_iterator_trait, *reverse_iterator*>

using **position_type** = std::pair<*iterator*, *size_type*>

using **const_position_type** = std::pair<*const_iterator*, *size_type*>

using **value_type** = mdds::detail::mtv::iterator_value_node<*multi_type_vector*, *size_type*>

value_type is the type of a block stored in the primary array. It consists of the following data members:

- `type` which indicates the block type,
- `position` which stores the logical position of the first element of the block,
- `size` which stores the logical size of the block, and
- `data` which stores the pointer to a secondary array (a.k.a. element block) which stores the actual element values, or `nullptr` in case the block represents an empty segment.

Public Functions

event_func &**event_handler**()

const *event_func* &**event_handler**() const

multi_type_vector()

Default constructor. It initializes the container with empty size.

multi_type_vector(const *event_func* &hdl)

Constructor that takes an lvalue reference to an event handler object. The event handler instance will be copy-constructed.

Parameters
hdl – lvalue reference to an event handler object.

multi_type_vector(*event_func* &&hdl)

Constructor that takes an rvalue reference to an event handler object. The event handler instance will be move-constructed.

Parameters
hdl – rvalue reference to an event handler object.

multi_type_vector(*size_type* init_size)

Constructor that takes initial size of the container. When the size specified is greater than 0, it initializes the container with empty elements.

Parameters
init_size – initial container size.

template<typename **T**>
multi_type_vector(*size_type* init_size, const *T* &value)

Constructor that takes initial size of the container and an element value to initialize the elements to. When the size specified is greater than 0, it initializes the container with elements that are copies of the value specified.

Parameters

- **init_size** – initial container size.
- **value** – initial element value.

template<typename **T**>

multi_type_vector(*size_type* init_size, const *T* &it_begin, const *T* &it_end)

Constructor that takes initial size of the container and begin and end iterator positions that specify a series of elements to initialize the container to. The container will contain copies of the elements specified after this call returns.

Parameters

- **init_size** – initial container size.
- **it_begin** – iterator that points to the begin position of the values the container is being initialized to.
- **it_end** – iterator that points to the end position of the values the container is being initialized to. The end position is *not* inclusive.

multi_type_vector(const *multi_type_vector* &other)

Copy constructor.

Parameters

other – the other instance to copy values from.

multi_type_vector(*multi_type_vector* &&other)

Move constructor.

Parameters

other – the other instance to move values from.

~multi_type_vector()

Destructor. It deletes all allocated element blocks.

position_type **position**(*size_type* pos)

Given the logical position of an element, get the iterator of the block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range, except when the specified position is the position immediately after the last valid position, it will return a valid position object representing the end position.

Parameters

pos – logical position of the element.

Returns

position object that stores an iterator referencing the element block where the element resides, and its offset within that block.

position_type **position**(const *iterator* &pos_hint, *size_type* pos)

Given the logical position of an element, get the iterator of the block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range, except when the specified position is the position immediately after the last valid position, it will return a valid position object representing the end position.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the element position.
- **pos** – logical position of the element.

Returns
position object that stores an iterator referencing the element block where the element resides, and its offset within that block.

const_position_type **position**(*size_type* pos) const

Given the logical position of an element, get an iterator referencing the element block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element.

Returns
position object that stores an iterator referencing the element block where the element resides, and its offset within that block.

const_position_type **position**(const *const_iterator* &pos_hint, *size_type* pos) const

Given the logical position of an element, get an iterator referencing the element block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the element position.
- **pos** – logical position of the element.

Returns
position object that stores an iterator referencing the element block where the element resides, and its offset within the block.

iterator **transfer**(*size_type* start_pos, *size_type* end_pos, *multi_type_vector* &dest, *size_type* dest_pos)

Move elements from one container to another. After the move, the segment where the elements were in the source container becomes empty. When transferring managed elements, this call transfers ownership of the moved elements to the destination container. The moved elements will overwrite any existing elements in the destination range of the receiving container. Transfer of elements within the same container is not allowed.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is greater than or equal to the source container size, or the destination container is not large enough to accommodate the transferred elements.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.
- **dest** – destination container to which the elements are to be moved.
- **dest_pos** – position in the destination container to which the elements are to be moved.

Returns
iterator referencing the block where the moved elements were prior to the transfer.

iterator **transfer**(const *iterator* &pos_hint, *size_type* start_pos, *size_type* end_pos, *multi_type_vector* &dest, *size_type* dest_pos)

Move elements from one container to another. After the move, the segment where the elements were in the source container becomes empty. When transferring managed elements, this call transfers ownership of the moved elements to the new container. The moved elements will overwrite any existing elements in the destination range of the receiving container. Transfer of elements within the same container is not allowed.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is greater than or equal to the source container size, or the destination container is not large enough to accommodate the transferred elements.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the blocks where the elements to be transferred reside.
- **start_pos** – starting position
- **end_pos** – ending position, inclusive.
- **dest** – destination container to which the elements are to be moved.
- **dest_pos** – position in the destination container to which the elements are to be moved.

Returns

iterator referencing the block where the moved elements were prior to the transfer.

template<typename **T**>
iterator **set**(*size_type* pos, const *T* &value)

Set a value of an arbitrary type to a specified position. The type of the value is inferred from the value passed to this method. The new value will overwrite an existing value at the specified position position if any.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos** – position to insert the value to.
- **value** – value to insert.

Returns

iterator position pointing to the block where the value is inserted.

template<typename **T**>
iterator **set**(const *iterator* &pos_hint, *size_type* pos, const *T* &value)

Set a value of an arbitrary type to a specified position. The type of the value is inferred from the value passed to this method. The new value will overwrite an existing value at the specified position position if any.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the right block to insert the value into. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the insertion position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block to insert the value into.
- **pos** – position to insert the value to.
- **value** – value to insert.

Returns

iterator position pointing to the block where the value is inserted.

template<typename **T**>
iterator **set**(*size_type* pos, const *T* &it_begin, const *T* &it_end)

Set multiple values of identical type to a range of elements starting at specified position. Any existing values will be overwritten by the new values.

The method will throw an `std::out_of_range` exception if the range of new values would fall outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos** – position of the first value of the series of new values being inserted.
- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

Returns

iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
iterator **set**(const *iterator* &pos_hint, *size_type* pos, const *T* &it_begin, const *T* &it_end)

Set multiple values of identical type to a range of elements starting at specified position. Any existing values will be overwritten by the new values.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first insertion block. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the insertion position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if the range of new values would fall outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block to insert the value into.
- **pos** – position of the first value of the series of new values being inserted.

- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

Returns
iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
iterator **push_back**(const *T* &value)

Append a new value to the end of the container.

Parameters
value – new value to be appended to the end of the container.

Returns
iterator position pointing to the block where the value is appended, which in this case is always the last block of the container.

iterator **push_back_empty**()

Append a new empty element to the end of the container.

Returns
iterator position pointing to the block where the new empty element is appended, which in this case is always the last block of the container.

template<typename **T**>
iterator **insert**(*size_type* pos, const *T* &it_begin, const *T* &it_end)

Insert multiple values of identical type to a specified position. Existing values that occur at or below the specified position will get shifted after the insertion. No existing values will be overwritten by the inserted values.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the new values inserted.

Parameters

- **pos** – position at which the new values are to be inserted.
- **it_begin** – iterator that points to the begin position of the values being inserted.
- **it_end** – iterator that points to the end position of the values being inserted.

Returns
iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
iterator **insert**(const *iterator* &pos_hint, *size_type* pos, const *T* &it_begin, const *T* &it_end)

Insert multiple values of identical type to a specified position. Existing values that occur at or below the specified position will get shifted after the insertion. No existing values will be overwritten by the inserted values.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first insertion block. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the insertion position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the new values inserted.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block to insert the value into.
- **pos** – position at which the new values are to be inserted.
- **it_begin** – iterator that points to the begin position of the values being inserted.
- **it_end** – iterator that points to the end position of the values being inserted.

Returns

iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

mtv::*element_t* **get_type**(*size_type* pos) const

Get the type of an element at specified position.

Parameters

pos – position of the element.

Returns

element type.

bool **is_empty**(*size_type* pos) const

Check if element at specified position is empty of not.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

pos – position of the element to check.

Returns

true if the element is empty, false otherwise.

iterator **set_empty**(*size_type* start_pos, *size_type* end_pos)

Set specified range of elements to be empty. Any existing values will be overwritten.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns

iterator position pointing to the block where the elements are emptied.

iterator **set_empty**(const *iterator* &pos_hint, *size_type* start_pos, *size_type* end_pos)

Set specified range of elements to be empty. Any existing values will be overwritten.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first block to empty. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the start position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right blocks to empty.
- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are emptied.

void **erase**(*size_type* start_pos, *size_type* end_pos)

Erase elements located between specified start and end positions. The end positions are both inclusive. Those elements originally located after the specified end position will get shifted up after the erasure.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container range.

Calling this method will decrease the size of the container by the length of the erased range.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

iterator **insert_empty**(*size_type* pos, *size_type* length)

Insert a range of empty elements at specified position. Those elements originally located after the insertion position will get shifted down after the insertion.

The method will throw an `std::out_of_range` exception if either the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the inserted empty elements.

Parameters

- **pos** – position at which to insert a range of empty elements.
- **length** – number of empty elements to insert.

Returns
iterator position pointing to the block where the empty range is inserted. When no insertion occurs because the length is zero, the end iterator position is returned.

iterator **insert_empty**(const *iterator* &pos_hint, *size_type* pos, *size_type* length)

Insert a range of empty elements at specified position. Those elements originally located after the insertion position will get shifted down after the insertion.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the block in which to insert the new empty segment. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the start position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if either the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the inserted empty elements.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block in which to insert the empty segment.
- **pos** – position at which to insert a range of empty elements.
- **length** – number of empty elements to insert.

Returns

iterator position pointing to the block where the empty range is inserted. When no insertion occurs because the length is zero, the end iterator position is returned.

void **clear**()

Clear the content of the container. The size of the container will become zero after calling this method.

size_type **size**() const

Return the current container size.

Returns

current container size.

size_type **block_size**() const

Return the current number of blocks in the primary array. Each non-empty block stores a secondary block that stores elements in a contiguous memory region (element block) and the number of elements it stores. An empty block only stores its logical size and does not store an actual element block.

For instance, if the container stores values of double-precision type at rows 0 to 2, values of std::string type at 3 to 7, and empty values at 8 to 10, it would consist of three blocks: one that stores double values, one that stores std::string values, and one that represents the empty value range in this exact order. In this specific scenario, *block_size()* returns 3, and *size()* returns 11.

Returns

current number of blocks in the primary array.

bool **empty**() const

Return whether or not the container is empty.

Returns

true if the container is empty, false otherwise.

template<typename **T**>
void **get**(*size_type* pos, *T* &value) const

Get the value of an element at specified position. The caller must pass a variable of the correct type to store the value.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos** – position of the element value to retrieve.
- **value** – (out) variable to store the retrieved value.

template<typename **T**>

T **get**(*size_type* pos) const

Get the value of an element at specified position. The caller must specify the type of the element as the template parameter e.g. get<double>(1).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element value to retrieve.

Returns
element value.

template<typename **T**>
T **release**(*size_type* pos)

Return the value of an element at specified position and set that position empty. If the element resides in a managed element block, this call will release that element from that block. If the element is on a non-managed element block, this call is equivalent to set_empty(pos, pos).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element to release.

Returns
element value.

template<typename **T**>
iterator **release**(*size_type* pos, *T* &value)

Retrieve the value of an element at specified position and set that position empty. If the element resides in a managed element block, this call will release that element from that block. If the element is on a non-managed element block, this call is equivalent to set_empty(pos, pos).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos** – position of the element to release.
- **value** – element value.

Returns
iterator referencing the block where the position of the released element is.

template<typename **T**>
iterator **release**(const *iterator* &pos_hint, *size_type* pos, *T* &value)

Retrieve the value of an element at specified position and set that position empty. If the element resides in a managed element block, this call will release that element from that block. If the element is on a non-managed element block, this call is equivalent to set_empty(pos, pos).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the block where the element resides.
- **pos** – position of the element to release.
- **value** – element value.

Returns
iterator referencing the block where the position of the released element is.

void **release**()

Release all its elements, and empties its content. Calling this method relinquishes the ownership of all elements stored in managed element blocks if any.

This call is equivalent of *clear()* if the container consists of no managed element blocks.

iterator **release_range**(*size_type* start_pos, *size_type* end_pos)

Make all elements within specified range empty, and relinquish the ownership of the elements in that range. All elements in the managed element blocks within the range will be released and the container will no longer manage their life cycles after the call.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are released.

iterator **release_range**(const *iterator* &pos_hint, *size_type* start_pos, *size_type* end_pos)

Make all elements within specified range empty, and relinquish the ownership of the elements in that range. All elements in the managed element blocks within the range will be released and the container will no longer manage their life cycles after the call.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first block to empty. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right blocks in which elements are to be released.
- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are released.

iterator **begin**()

iterator **end**()

const_iterator **begin**() const

const_iterator **end**() const

const_iterator **cbegin**() const

const_iterator **cend**() const

reverse_iterator **rbegin**()

reverse_iterator **rend**()

const_reverse_iterator **rbegin**() const

const_reverse_iterator **rend**() const

const_reverse_iterator **crbegin**() const

const_reverse_iterator **crend**() const

void **resize**(*size_type* new_size)

Extend or shrink the container. When extending the container, it appends a series of empty elements to the end. When shrinking, the elements at the end of the container get stripped off.

Parameters
new_size – size of the container after the resize.

void **swap**(*multi_type_vector* &other)

Swap the content with another container.

Parameters
other – another container to swap content with.

void **swap**(*size_type* start_pos, *size_type* end_pos, *multi_type_vector* &other, *size_type* other_pos)

Swap a part of the content with another instance.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.
- **other** – another instance to swap the content with.
- **other_pos** – insertion position in the other instance.

void **shrink_to_fit**()

Trim excess capacity from all non-empty blocks.

bool **operator==**(const *multi_type_vector* &other) const

bool **operator!=**(const *multi_type_vector* &other) const

multi_type_vector &**operator=**(const *multi_type_vector* &other)

multi_type_vector &**operator=**(*multi_type_vector* &&other)

Public Static Functions

static *position_type* **next_position**(const *position_type* &pos)

Move the position object to the next logical position. Caller must ensure the the position object is valid.

Parameters
pos – position object.

Returns
position object that points to the next logical position.

static *position_type* **advance_position**(const *position_type* &pos, int steps)

Increment or decrement the position object by specified steps. Caller must ensure the the position object is valid.

Parameters

- **pos** – position object.
- **steps** – steps to advance the position object.

Returns
position object that points to the new logical position.

static *const_position_type* **next_position**(const *const_position_type* &pos)

Move the position object to the next logical position. Caller must ensure the the position object is valid.

Parameters
pos – position object.

Returns
position object that points to the next logical position.

static *const_position_type* **advance_position**(const *const_position_type* &pos, int steps)

Increment or decrement the position object by specified steps. Caller must ensure the the position object is valid.

Parameters

- **pos** – position object.
- **steps** – steps to advance the position object.

Returns
position object that points to the new logical position.

static *size_type* **logical_position**(const *const_position_type* &pos)

Extract the logical position from a position object.

Parameters
pos – position object.

Returns
logical position of the element that the position object references.

template<typename **_Blk**>
static *_Blk*::value_type **get**(const *const_position_type* &pos)

Get element value from a position object. The caller must specify the type of block in which the element is expected to be stored.

Parameters
pos – position object.

Returns
element value.

template<typename **T**>
static mtv::*element_t* **get_element_type**(const *T* &elem)

Return the numerical identifier that represents passed element.

Parameters
elem – element value.

Returns
numerical identifier representing the element.

mdds::mtv::aos::multi_type_vector

template<typename **Traits** = mdds::mtv::*default_traits*>

class **multi_type_vector**

Multi-type vector consists of a series of one or more blocks, and each block may either be empty, or stores a series of non-empty elements of identical type. These blocks collectively represent a single logical one-dimensional array that may store elements of different types. It is guaranteed that the block types of neighboring blocks are always different.

Structurally, the primary array stores block instances whose types are of `value_type`, which in turn consists of the following data members:

- `type` which indicates the block type,
- `position` which stores the logical position of the first element of the block,
- `size` which stores the logical size of the block, and
- `data` which stores the pointer to a secondary array (a.k.a. element block) which stores the actual element values, or `nullptr` in case the block represents an empty segment.

This variant implements an array-of-structures (AoS) storage.

See also:

mdds::mtv::aos::multi_type_vector::value_type

Public Types

typedef size_t **size_type**

typedef mdds::mtv::*base_element_block* **element_block_type**

typedef mdds::mtv::*element_t* **element_category_type**

using **block_funcs** = typename *Traits*::block_funcs

using **event_func** = typename *Traits*::event_func

Optional event handler function structure, whose functions get called at specific events. The following events are currently supported:

- `element_block_acquired` - this gets called whenever the container acquires a new element block either as a result of a new element block creation or a tranfer of an existing element block from another container.

- `element_block_released` - this gets called whenever the container releases an existing element block either because the block gets deleted or gets transferred to another container.

See also:

mdds::mtv::empty_event_func for the precise function signatures of the event handler functions.

typedef detail::iterator_base<iterator_trait, itr_forward_update> **iterator**

typedef detail::iterator_base<reverse_iterator_trait, itr_no_update> **reverse_iterator**

typedef detail::const_iterator_base<const_iterator_trait, itr_forward_update, *iterator*> **const_iterator**

typedef detail::const_iterator_base<const_reverse_iterator_trait, itr_no_update, *reverse_iterator*> **const_reverse_iterator**

typedef itr_node **value_type**

value_type is the type of a block stored in the primary array. It consists of the following data members:

- `type` which indicates the block type,
- `position` which stores the logical position of the first element of the block,
- `size` which stores the logical size of the block, and
- `data` which stores the pointer to a secondary array (a.k.a. element block) which stores the actual element values, or `nullptr` in case the block represents an empty segment.

typedef std::pair<*iterator*, *size_type*> **position_type**

typedef std::pair<*const_iterator*, *size_type*> **const_position_type**

Public Functions

iterator **begin**()

iterator **end**()

const_iterator **begin**() const

const_iterator **end**() const

const_iterator **cbegin**() const

const_iterator **cend**() const

reverse_iterator **rbegin**()

reverse_iterator **rend**()

const_reverse_iterator **rbegin**() const

const_reverse_iterator **rend**() const

const_reverse_iterator **crbegin**() const

const_reverse_iterator **crend**() const

event_func &**event_handler**()

const *event_func* &**event_handler**() const

multi_type_vector()

Default constructor. It initializes the container with empty size.

multi_type_vector(const *event_func* &hdl)

Constructor that takes an lvalue reference to an event handler object. The event handler instance will be copy-constructed.

Parameters
hdl – lvalue reference to an event handler object.

multi_type_vector(*event_func* &&hdl)

Constructor that takes an rvalue reference to an event handler object. The event handler instance will be move-constructed.

Parameters
hdl – rvalue reference to an event handler object.

multi_type_vector(*size_type* init_size)

Constructor that takes initial size of the container. When the size specified is greater than 0, it initializes the container with empty elements.

Parameters
init_size – initial container size.

template<typename **T**>
multi_type_vector(*size_type* init_size, const *T* &value)

Constructor that takes initial size of the container and an element value to initialize the elements to. When the size specified is greater than 0, it initializes the container with elements that are copies of the value specified.

Parameters

- **init_size** – initial container size.
- **value** – initial element value.

template<typename **T**>
multi_type_vector(*size_type* init_size, const *T* &it_begin, const *T* &it_end)

Constructor that takes initial size of the container and begin and end iterator positions that specify a series of elements to initialize the container to. The container will contain copies of the elements specified after this call returns.

Parameters

- **init_size** – initial container size.
- **it_begin** – iterator that points to the begin position of the values the container is being initialized to.
- **it_end** – iterator that points to the end position of the values the container is being initialized to. The end position is *not* inclusive.

multi_type_vector(const *multi_type_vector* &other)

Copy constructor.

Parameters
other – the other instance to copy values from.

multi_type_vector(*multi_type_vector* &&other)

Move constructor.

Parameters
other – the other instance to move values from.

~multi_type_vector()

Destructor. It deletes all allocated data blocks.

template<typename **T**>
iterator **set**(*size_type* pos, const *T* &value)

Set a value of an arbitrary type to a specified position. The type of the value is inferred from the value passed to this method. The new value will overwrite an existing value at the specified position position if any.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos** – position to insert the value to.
- **value** – value to insert.

Returns
iterator position pointing to the block where the value is inserted.

template<typename **T**>
iterator **set**(const *iterator* &pos_hint, *size_type* pos, const *T* &value)

Set a value of an arbitrary type to a specified position. The type of the value is inferred from the value passed to this method. The new value will overwrite an existing value at the specified position position if any.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the right block to insert the value into. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the insertion position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block to insert the value into.
- **pos** – position to insert the value to.
- **value** – value to insert.

Returns
iterator position pointing to the block where the value is inserted.

template<typename **T**>
iterator **set**(*size_type* pos, const *T* &it_begin, const *T* &it_end)

Set multiple values of identical type to a range of elements starting at specified position. Any existing values will be overwritten by the new values.

The method will throw an `std::out_of_range` exception if the range of new values would fall outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos** – position of the first value of the series of new values being inserted.
- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

Returns
iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
iterator **set**(const *iterator* &pos_hint, *size_type* pos, const *T* &it_begin, const *T* &it_end)

Set multiple values of identical type to a range of elements starting at specified position. Any existing values will be overwritten by the new values.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first insertion block. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the insertion position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if the range of new values would fall outside the current container range.

Calling this method will not change the size of the container.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block to insert the value into.
- **pos** – position of the first value of the series of new values being inserted.
- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

Returns
iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
iterator **push_back**(const *T* &value)

Append a new value to the end of the container.

Parameters
value – new value to be appended to the end of the container.

Returns
iterator position pointing to the block where the value is appended, which in this case is always the last block of the container.

iterator **push_back_empty**()

Append a new empty element to the end of the container.

Returns
iterator position pointing to the block where the new empty element is appended, which in this case is always the last block of the container.

template<typename **T**>
iterator **insert**(*size_type* pos, const *T* &it_begin, const *T* &it_end)

Insert multiple values of identical type to a specified position. Existing values that occur at or below the specified position will get shifted after the insertion. No existing values will be overwritten by the inserted values.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the new values inserted.

Parameters

- **pos** – position at which the new values are to be inserted.
- **it_begin** – iterator that points to the begin position of the values being inserted.
- **it_end** – iterator that points to the end position of the values being inserted.

Returns
iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
iterator **insert**(const *iterator* &pos_hint, *size_type* pos, const *T* &it_begin, const *T* &it_end)

Insert multiple values of identical type to a specified position. Existing values that occur at or below the specified position will get shifted after the insertion. No existing values will be overwritten by the inserted values.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first insertion block. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the insertion position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the new values inserted.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block to insert the value into.
- **pos** – position at which the new values are to be inserted.

- **it_begin** – iterator that points to the begin position of the values being inserted.
- **it_end** – iterator that points to the end position of the values being inserted.

Returns
iterator position pointing to the block where the value is inserted. When no value insertion occurs because the value set is empty, the end iterator position is returned.

template<typename **T**>
void **get**(*size_type* pos, *T* &value) const

Get the value of an element at specified position. The caller must pass a variable of the correct type to store the value.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos** – position of the element value to retrieve.
- **value** – (out) variable to store the retrieved value.

template<typename **T**>
T **get**(*size_type* pos) const

Get the value of an element at specified position. The caller must specify the type of the element as the template parameter e.g. get<double>(1).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element value to retrieve.

Returns
element value.

template<typename **T**>
T **release**(*size_type* pos)

Return the value of an element at specified position and set that position empty. If the element resides in a managed element block, this call will release that element from that block. If the element is on a non-managed element block, this call is equivalent to set_empty(pos, pos).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element to release.

Returns
element value.

template<typename **T**>
iterator **release**(*size_type* pos, *T* &value)

Retrieve the value of an element at specified position and set that position empty. If the element resides in a managed element block, this call will release that element from that block. If the element is on a non-managed element block, this call is equivalent to set_empty(pos, pos).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos** – position of the element to release.

- **value** – element value.

Returns
iterator referencing the block where the position of the released element is.

template<typename **T**>
iterator **release**(const *iterator* &pos_hint, *size_type* pos, *T* &value)

Retrieve the value of an element at specified position and set that position empty. If the element resides in a managed element block, this call will release that element from that block. If the element is on a non-managed element block, this call is equivalent to set_empty(pos, pos).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the block where the element resides.
- **pos** – position of the element to release.
- **value** – element value.

Returns
iterator referencing the block where the position of the released element is.

void **release**()

Release all its elements, and empties its content. Calling this method relinquishes the ownership of all elements stored in managed element blocks if any.

This call is equivalent of *clear()* if the container consists of no managed element blocks.

iterator **release_range**(*size_type* start_pos, *size_type* end_pos)

Make all elements within specified range empty, and relinquish the ownership of the elements in that range. All elements in the managed element blocks within the range will be released and the container will no longer manage their life cycles after the call.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are released.

iterator **release_range**(const *iterator* &pos_hint, *size_type* start_pos, *size_type* end_pos)

Make all elements within specified range empty, and relinquish the ownership of the elements in that range. All elements in the managed element blocks within the range will be released and the container will no longer manage their life cycles after the call.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first block to empty. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right blocks in which elements are to be released.
- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are released.

position_type **position**(*size_type* pos)

Given the logical position of an element, get the iterator of the block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range, except when the specified position is the position immediately after the last valid position, it will return a valid position object representing the end position.

Parameters
pos – logical position of the element.

Returns
position object that stores an iterator referencing the element block where the element resides, and its offset within that block.

position_type **position**(const *iterator* &pos_hint, *size_type* pos)

Given the logical position of an element, get the iterator of the block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range, except when the specified position is the position immediately after the last valid position, it will return a valid position object representing the end position.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the element position.
- **pos** – logical position of the element.

Returns
position object that stores an iterator referencing the element block where the element resides, and its offset within that block.

const_position_type **position**(*size_type* pos) const

Given the logical position of an element, get an iterator referencing the element block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element.

Returns
position object that stores an iterator referencing the element block where the element resides, and its offset within that block.

const_position_type **position**(const *const_iterator* &pos_hint, *size_type* pos) const

Given the logical position of an element, get an iterator referencing the element block where the element is located, and its offset from the first element of that block.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the element position.
- **pos** – logical position of the element.

Returns

position object that stores an iterator referencing the element block where the element resides, and its offset within the block.

iterator **transfer**(*size_type* start_pos, *size_type* end_pos, *multi_type_vector* &dest, *size_type* dest_pos)

Move elements from one container to another. After the move, the segment where the elements were in the source container becomes empty. When transferring managed elements, this call transfers ownership of the moved elements to the destination container. The moved elements will overwrite any existing elements in the destination range of the receiving container. Transfer of elements within the same container is not allowed.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is greater than or equal to the source container size, or the destination container is not large enough to accommodate the transferred elements.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.
- **dest** – destination container to which the elements are to be moved.
- **dest_pos** – position in the destination container to which the elements are to be moved.

Returns

iterator referencing the block where the moved elements were prior to the transfer.

iterator **transfer**(const *iterator* &pos_hint, *size_type* start_pos, *size_type* end_pos, *multi_type_vector* &dest, *size_type* dest_pos)

Move elements from one container to another. After the move, the segment where the elements were in the source container becomes empty. When transferring managed elements, this call transfers ownership of the moved elements to the new container. The moved elements will overwrite any existing elements in the destination range of the receiving container. Transfer of elements within the same container is not allowed.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is greater than or equal to the source container size, or the destination container is not large enough to accommodate the transferred elements.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the blocks where the elements to be transferred reside.
- **start_pos** – starting position
- **end_pos** – ending position, inclusive.
- **dest** – destination container to which the elements are to be moved.
- **dest_pos** – position in the destination container to which the elements are to be moved.

Returns

iterator referencing the block where the moved elements were prior to the transfer.

mtv::*element_t* **get_type**(*size_type* pos) const

Get the type of an element at specified position.

Parameters
pos – position of the element.

Returns
element type.

bool **is_empty**(*size_type* pos) const

Check if element at specified position is empty of not.

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters
pos – position of the element to check.

Returns
true if the element is empty, false otherwise.

iterator **set_empty**(*size_type* start_pos, *size_type* end_pos)

Set specified range of elements to be empty. Any existing values will be overwritten.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are emptied.

iterator **set_empty**(const *iterator* &pos_hint, *size_type* start_pos, *size_type* end_pos)

Set specified range of elements to be empty. Any existing values will be overwritten.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the first block to empty. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the start position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container size.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right blocks to empty.
- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

Returns
iterator position pointing to the block where the elements are emptied.

void **erase**(*size_type* start_pos, *size_type* end_pos)

Erase elements located between specified start and end positions. The end positions are both inclusive. Those elements originally located after the specified end position will get shifted up after the erasure.

The method will throw an `std::out_of_range` exception if either the starting or the ending position is outside the current container range.

Calling this method will decrease the size of the container by the length of the erased range.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.

iterator **insert_empty**(*size_type* pos, *size_type* length)

Insert a range of empty elements at specified position. Those elements originally located after the insertion position will get shifted down after the insertion.

The method will throw an `std::out_of_range` exception if either the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the inserted empty elements.

Parameters

- **pos** – position at which to insert a range of empty elements.
- **length** – number of empty elements to insert.

Returns

iterator position pointing to the block where the empty range is inserted. When no insertion occurs because the length is zero, the end iterator position is returned.

iterator **insert_empty**(const *iterator* &pos_hint, *size_type* pos, *size_type* length)

Insert a range of empty elements at specified position. Those elements originally located after the insertion position will get shifted down after the insertion.

This variant takes an iterator as an additional parameter, which is used as a block position hint to speed up the lookup of the block in which to insert the new empty segment. The other variant that doesn't take an iterator always starts the block lookup from the first block, which does not scale well as the block size grows.

This position hint iterator must **precede** the start position to yield any performance benefit.

The caller is responsible for ensuring that the passed iterator is valid. The behavior of this method when passing an invalid iterator is undefined.

The method will throw an `std::out_of_range` exception if either the specified position is outside the current container range.

Calling this method will increase the size of the container by the length of the inserted empty elements.

Parameters

- **pos_hint** – iterator used as a block position hint, to specify which block to start when searching for the right block in which to insert the empty segment.
- **pos** – position at which to insert a range of empty elements.
- **length** – number of empty elements to insert.

Returns

iterator position pointing to the block where the empty range is inserted. When no insertion occurs because the length is zero, the end iterator position is returned.

void **clear**()

Clear the content of the container. The size of the container will become zero after calling this method.

size_type **size**() const

Return the current container size.

Returns
current container size.

size_type **block_size**() const

Return the current number of blocks in the primary array. Each non-empty block stores a secondary block that stores elements in a contiguous region in memory (element block) and the number of elements it stores. An empty block only stores its logical size and does not store an actual element block.

For instance, if the container stores values of double-precision type at rows 0 to 2, values of std::string type at 3 to 7, and empty values at 8 to 10, it would consist of three blocks: one that stores double values, one that stores std::string values, and one that represents the empty value range in this exact order. In this specific scenario, *block_size()* returns 3, and *size()* returns 11.

Returns
current number of blocks in the primary array.

bool **empty**() const

Return whether or not the container is empty.

Returns
true if the container is empty, false otherwise.

void **resize**(*size_type* new_size)

Extend or shrink the container. When extending the container, it appends a series of empty elements to the end. When shrinking, the elements at the end of the container get stripped off.

Parameters
new_size – size of the container after the resize.

void **swap**(*multi_type_vector* &other)

Swap the content with another container.

Parameters
other – another container to swap content with.

void **swap**(*size_type* start_pos, *size_type* end_pos, *multi_type_vector* &other, *size_type* other_pos)

Swap a part of the content with another instance.

Parameters

- **start_pos** – starting position
- **end_pos** – ending position, inclusive.
- **other** – another instance to swap the content with.
- **other_pos** – insertion position in the other instance.

void **shrink_to_fit**()

Trim excess capacity from all non-empty blocks.

bool **operator==**(const *multi_type_vector* &other) const

bool **operator!=**(const *multi_type_vector* &other) const

multi_type_vector &**operator=**(const *multi_type_vector* &other)

multi_type_vector &**operator=**(*multi_type_vector* &&other)

Public Static Functions

static *position_type* **next_position**(const *position_type* &pos)

Move the position object to the next logical position. Caller must ensure the the position object is valid.

Parameters
pos – position object.

Returns
position object that points to the next logical position.

static *position_type* **advance_position**(const *position_type* &pos, int steps)

Increment or decrement the position object by specified steps. Caller must ensure the the position object is valid.

Parameters

- **pos** – position object.
- **steps** – steps to advance the position object.

Returns
position object that points to the new logical position.

static *const_position_type* **next_position**(const *const_position_type* &pos)

Move the position object to the next logical position. Caller must ensure the the position object is valid.

Parameters
pos – position object.

Returns
position object that points to the next logical position.

static *const_position_type* **advance_position**(const *const_position_type* &pos, int steps)

Increment or decrement the position object by specified steps. Caller must ensure the the position object is valid.

Parameters

- **pos** – position object.
- **steps** – steps to advance the position object.

Returns
position object that points to the new logical position.

static *size_type* **logical_position**(const *const_position_type* &pos)

Extract the logical position from a position object.

Parameters
pos – position object.

Returns
logical position of the element that the position object references.

template<typename **Blk**>

static *Blk*::value_type **get**(const *const_position_type* &pos)

Get element value from a position object. The caller must specify the type of block in which the element is expected to be stored.

Parameters
pos – position object.

Returns
element value.

template<typename **T**>
static mtv::*element_t* **get_element_type**(const *T* &elem)

Return the numerical identifier that represents passed element.

Parameters
elem – element value.

Returns
numerical identifier representing the element.

mdds::mtv::empty_event_func

struct **empty_event_func**

Empty event function handler structure, used when no custom function handler is specified.

Public Functions

inline void **element_block_acquired**(const *base_element_block* *block)

Callback function for element block acquisition events. This gets called whenever the container acquires a new element block either as a result of a new element block creation or a transfer of an existing element block from another container.

Parameters
block – pointer to the acquired element block instance.

inline void **element_block_released**(const *base_element_block* *block)

Callback function for element block release events. This gets called whenever the container releases an existing element block either because the block is about to be deleted or to be transferred to another container.

Parameters
block – pointer to the element block instance being released.

mdds::mtv::default_traits

struct **default_traits**

Default trait to be used when no custom trait is specified.

Subclassed by *mdds::mtv::standard_element_blocks_traits*

Public Types

using **event_func** = *empty_event_func*

Class or struct type that contains callback functions for element block events as its member functions.

using **block_funcs** = *element_block_funcs*<>

Type that contains block functions used throughout the multi_type_vector implementation. The user must overwrite this type to specify one or more block types as template arguments to *element_block_funcs*. Alternatively, you may be interested in using *standard_element_blocks_traits* which already supports the pre-defined block types for the optional standard data types.

Public Static Attributes

static constexpr *lu_factor_t* **loop_unrolling** = *lu_factor_t*::*lu16*

Static value specifying the loop-unrolling factor to use for the block position adjustment function. This must be a const expression.

5.4.2 Element Stores

template<typename **T**, typename **Allocator** = std::allocator<*T*>>

class **delayed_delete_vector**

Vector that delays deleting from the front of the vector, which avoids O(n^2) memory move operations when code needs to delete items from one element block and add to another element block.

Public Types

typedef store_type::value_type **value_type**

typedef store_type::size_type **size_type**

typedef store_type::difference_type **difference_type**

typedef store_type::reference **reference**

typedef store_type::const_reference **const_reference**

typedef store_type::pointer **pointer**

typedef store_type::const_pointer **const_pointer**

typedef store_type::iterator **iterator**

typedef store_type::reverse_iterator **reverse_iterator**

typedef store_type::const_iterator **const_iterator**

typedef store_type::const_reverse_iterator **const_reverse_iterator**

Public Functions

inline **delayed_delete_vector**()

inline **delayed_delete_vector**(size_t n, const *T* &val)

inline **delayed_delete_vector**(size_t n)

template<typename **InputIt**>
inline **delayed_delete_vector**(*InputIt* first, *InputIt* last)

inline *iterator* **begin**() noexcept

inline *iterator* **end**() noexcept

inline *const_iterator* **begin**() const noexcept

inline *const_iterator* **end**() const noexcept

inline *reverse_iterator* **rbegin**() noexcept

inline *const_reverse_iterator* **rbegin**() const noexcept

inline *reverse_iterator* **rend**() noexcept

inline *const_reverse_iterator* **rend**() const noexcept

inline *reference* **operator[]**(*size_type* pos)

inline *const_reference* **operator[]**(*size_type* pos) const

inline *reference* **at**(*size_type* pos)

inline *const_reference* **at**(*size_type* pos) const

inline void **push_back**(const *T* &value)

inline *iterator* **insert**(*iterator* pos, const *T* &value)

inline *iterator* **insert**(*const_iterator* pos, *T* &&value)

template<typename **InputIt**>
inline void **insert**(*iterator* pos, *InputIt* first, *InputIt* last)

inline void **resize**(*size_type* count)

inline *iterator* **erase**(*iterator* pos)

inline *iterator* **erase**(*iterator* first, *iterator* last)

inline *size_type* **capacity**() const noexcept

inline void **shrink_to_fit**()

inline void **reserve**(*size_type* new_cap)

inline *size_type* **size**() const

template<typename **InputIt**>
inline void **assign**(*InputIt* first, *InputIt* last)

inline *T* ***data**()

inline const *T* ***data**() const

5.4.3 Element Blocks

class **base_element_block**

Non-template common base type necessary for blocks of all types to be stored in a single container.

Subclassed by *mdds::mtv::element_block< default_element_block< TypeId, ValueT, delayed_delete_vector >, TypeId, ValueT, delayed_delete_vector >*, *mdds::mtv::element_block< managed_element_block< TypeId, ValueT, delayed_delete_vector >, TypeId, ValueT *, delayed_delete_vector >*, *mdds::mtv::element_block< noncopyable_managed_element_block< TypeId, ValueT, delayed_delete_vector >, TypeId, ValueT *, delayed_delete_vector >*, *mdds::mtv::element_block< Self, TypeId, ValueT, StoreT >*

template<typename **Self**, *element_t* **TypeId**, typename **ValueT**, template<typename, typename> class **StoreT**>

class **element_block** : public mdds::mtv::*base_element_block*

Subclassed by *mdds::mtv::copyable_element_block< default_element_block< TypeId, ValueT, delayed_delete_vector >, TypeId, ValueT, delayed_delete_vector >*, *mdds::mtv::copyable_element_block< managed_element_block< TypeId, ValueT, delayed_delete_vector >, TypeId, ValueT *, delayed_delete_vector >*, *mdds::mtv::noncopyable_element_block< noncopyable_managed_element_block< TypeId, ValueT, delayed_delete_vector >, TypeId, ValueT *, delayed_delete_vector >*, *mdds::mtv::copyable_element_block< Self, TypeId, ValueT, StoreT >*, *mdds::mtv::noncopyable_element_block< Self, TypeId, ValueT, StoreT >*

Public Types

using **store_type** = *StoreT*<*ValueT*, std::allocator<*ValueT*>>

typedef *store_type*::iterator **iterator**

typedef *store_type*::reverse_iterator **reverse_iterator**

typedef *store_type*::const_iterator **const_iterator**

typedef *store_type*::const_reverse_iterator **const_reverse_iterator**

typedef *ValueT* **value_type**

using **range_type** = base_range_type<true>

using **const_range_type** = base_range_type<false>

Public Functions

inline bool **operator==**(const *Self* &r) const

inline bool **operator!=**(const *Self* &r) const

Public Static Functions

static inline const *value_type* &**at**(const *base_element_block* &block, typename *store_type*::size_type pos)

static inline *value_type* &**at**(*base_element_block* &block, typename *store_type*::size_type pos)

static inline *value_type* ***data**(*base_element_block* &block)

static inline *store_type*::size_type **size**(const *base_element_block* &block)

static inline *iterator* **begin**(*base_element_block* &block)

static inline *iterator* **end**(*base_element_block* &block)

static inline *const_iterator* **begin**(const *base_element_block* &block)

static inline *const_iterator* **end**(const *base_element_block* &block)

static inline *const_iterator* **cbegin**(const *base_element_block* &block)

static inline *const_iterator* **cend**(const *base_element_block* &block)

static inline *reverse_iterator* **rbegin**(*base_element_block* &block)

static inline *reverse_iterator* **rend**(*base_element_block* &block)

static inline *const_reverse_iterator* **rbegin**(const *base_element_block* &block)

static inline *const_reverse_iterator* **rend**(const *base_element_block* &block)

static inline *const_reverse_iterator* **crbegin**(const *base_element_block* &block)

static inline *const_reverse_iterator* **crend**(const *base_element_block* &block)

static inline *const_range_type* **range**(const *base_element_block* &block)

static inline *range_type* **range**(*base_element_block* &block)

static inline *Self* &**get**(*base_element_block* &block)

static inline const *Self* &**get**(const *base_element_block* &block)

static inline void **set_value**(*base_element_block* &blk, size_t pos, const *ValueT* &val)

static inline void **get_value**(const *base_element_block* &blk, size_t pos, *ValueT* &val)

static inline *value_type* **get_value**(const *base_element_block* &blk, size_t pos)

static inline void **append_value**(*base_element_block* &blk, const *ValueT* &val)

static inline void **prepend_value**(*base_element_block* &blk, const *ValueT* &val)

static inline *Self* ***create_block**(size_t init_size)

static inline void **delete_block**(const *base_element_block* *p)

static inline void **resize_block**(*base_element_block* &blk, size_t new_size)

static inline void **print_block**(const *base_element_block*&)

static inline void **erase_value**(*base_element_block* &blk, size_t pos)

static inline void **erase_values**(*base_element_block* &blk, size_t pos, size_t size)

static inline void **append_block**(*base_element_block* &dest, const *base_element_block* &src)

static inline void **append_values_from_block**(*base_element_block* &dest, const *base_element_block* &src, size_t begin_pos, size_t len)

static inline void **assign_values_from_block**(*base_element_block* &dest, const *base_element_block* &src, size_t begin_pos, size_t len)

static inline void **prepend_values_from_block**(*base_element_block* &dest, const *base_element_block* &src, size_t begin_pos, size_t len)

static inline void **swap_values**(*base_element_block* &blk1, *base_element_block* &blk2, size_t pos1, size_t pos2, size_t len)

static inline bool **equal_block**(const *base_element_block* &left, const *base_element_block* &right)

template<typename **Iter**>
static inline void **set_values**(*base_element_block* &block, size_t pos, const *Iter* &it_begin, const *Iter* &it_end)

template<typename **Iter**>
static inline void **append_values**(*base_element_block* &block, const *Iter* &it_begin, const *Iter* &it_end)

template<typename **Iter**>
static inline void **prepend_values**(*base_element_block* &block, const *Iter* &it_begin, const *Iter* &it_end)

template<typename **Iter**>
static inline void **assign_values**(*base_element_block* &dest, const *Iter* &it_begin, const *Iter* &it_end)

template<typename **Iter**>
static inline void **insert_values**(*base_element_block* &block, size_t pos, const *Iter* &it_begin, const *Iter* &it_end)

static inline size_t **capacity**(const *base_element_block* &block)

static inline void **reserve**(*base_element_block* &block, std::size_t size)

static inline void **shrink_to_fit**(*base_element_block* &block)

Public Static Attributes

static constexpr *element_t* **block_type** = *TypeId*

template<*element_t* **TypeId**, typename **ValueT**, template<typename, typename> class **StoreT** = *delayed_delete_vector*>
struct **default_element_block** : public mdds::mtv::*copyable_element_block*<*default_element_block*<*TypeId*, *ValueT*, *delayed_delete_vector*>, *TypeId*, *ValueT*, *delayed_delete_vector*>

Template for default, unmanaged element block for use in multi_type_vector.

Public Types

using **self_type** = *default_element_block*<*TypeId*, *ValueT*, *StoreT*>

using **base_type** = *copyable_element_block*<*self_type*, *TypeId*, *ValueT*, *StoreT*>

Public Functions

inline **default_element_block**()

inline **default_element_block**(size_t n)

inline **default_element_block**(size_t n, const *ValueT* &val)

template<typename **Iter**>
inline **default_element_block**(const *Iter* &it_begin, const *Iter* &it_end)

Public Static Functions

static inline *self_type* *__create_block_with_value__(size_t init_size, const *ValueT* &val)

template<typename **Iter**>
static inline *self_type* *__create_block_with_values__(const *Iter* &it_begin, const *Iter* &it_end)

static inline void **overwrite_values**(*base_element_block*&, size_t, size_t)

template<typename **Self**, *element_t* **TypeId**, typename **ValueT**, template<typename, typename> class **StoreT**>

class **copyable_element_block** : public mdds::mtv::*element_block*<*Self*, *TypeId*, *ValueT*, *StoreT*>

Subclassed by *mdds::mtv::default_element_block< TypeId, ValueT, StoreT >*, *mdds::mtv::managed_element_block< TypeId, ValueT, StoreT >*

Public Static Functions

static inline *Self* *__clone_block__(const *base_element_block* &blk)

static inline *Self* &__get__(*base_element_block* &block)

static inline const *Self* &__get__(const *base_element_block* &block)

template<typename __Self__, *element_t* __TypeId__, typename __ValueT__, template<typename, typename> class __StoreT__>

class __noncopyable_element_block__ : public mdds::mtv::*element_block*<*Self*, *TypeId*, *ValueT*, *StoreT*>

Subclassed by *mdds::mtv::noncopyable_managed_element_block< TypeId, ValueT, StoreT >*

Public Functions

__noncopyable_element_block__(const *noncopyable_element_block*&) = delete

noncopyable_element_block &__operator=__(const *noncopyable_element_block*&) = delete

Public Static Functions

static inline *Self* *__clone_block__(const *base_element_block*&)

template<*element_t* __TypeId__, typename __ValueT__, template<typename, typename> class __StoreT__ = *delayed_delete_vector*>
struct __managed_element_block__ : public mdds::mtv::*copyable_element_block*<*managed_element_block*<*TypeId*, *ValueT*, *delayed_delete_vector*>, *TypeId*, *ValueT**, *delayed_delete_vector*>

Template for element block that stores pointers to objects whose life cycles are managed by the block.

Public Types

using __self_type__ = *managed_element_block*<*TypeId*, *ValueT*, *StoreT*>

using __base_type__ = *copyable_element_block*<*self_type*, *TypeId*, *ValueT**, *StoreT*>

Public Functions

inline __managed_element_block__()

inline __managed_element_block__(size_t n)

inline __managed_element_block__(const *managed_element_block* &r)

template<typename __Iter__>
inline __managed_element_block__(const *Iter* &it_begin, const *Iter* &it_end)

inline ~__managed_element_block__()

Public Static Functions

static inline *self_type* *__create_block_with_value__(size_t init_size, *ValueT* *val)

template<typename __Iter__>
static inline *self_type* *__create_block_with_values__(const *Iter* &it_begin, const *Iter* &it_end)

static inline void __overwrite_values__(*base_element_block* &block, size_t pos, size_t len)

static inline Self &__get__(*base_element_block* &block)

static inline const Self &__get__(const *base_element_block* &block)

template<*element_t* __TypeId__, typename __ValueT__, template<typename, typename> class __StoreT__ = *delayed_delete_vector*>
struct __noncopyable_managed_element_block__ : public mdds::mtv::*noncopyable_element_block*<*noncopyable_managed_element_block*<*TypeId*, *ValueT*, *delayed_delete_vector*>, *TypeId*, *ValueT**, *delayed_delete_vector*>

Public Types

using __self_type__ = *noncopyable_managed_element_block*<*TypeId*, *ValueT*, *StoreT*>

using __base_type__ = *noncopyable_element_block*<*self_type*, *TypeId*, *ValueT**, *StoreT*>

Public Functions

inline __noncopyable_managed_element_block__()

inline __noncopyable_managed_element_block__(size_t n)

template<typename __Iter__>
inline __noncopyable_managed_element_block__(const *Iter* &it_begin, const *Iter* &it_end)

inline ~__noncopyable_managed_element_block__()

Public Static Functions

static inline *self_type* *__create_block_with_value__(size_t init_size, *ValueT* *val)

template<typename __Iter__>
static inline *self_type* *__create_block_with_values__(const *Iter* &it_begin, const *Iter* &it_end)

static inline void __overwrite_values__(*base_element_block* &block, size_t pos, size_t len)

template<typename ...__Ts__>

struct __element_block_funcs__

Public Static Functions

static inline *base_element_block* ***create_new_block**(*element_t* type, std::size_t init_size)

static inline *base_element_block* ***clone_block**(const *base_element_block* &block)

static inline void **delete_block**(const *base_element_block* *p)

static inline void **resize_block**(*base_element_block* &block, std::size_t new_size)

static inline void **print_block**(const *base_element_block* &block)

static inline void **erase**(*base_element_block* &block, std::size_t pos)

static inline void **erase**(*base_element_block* &block, std::size_t pos, std::size_t size)

static inline void **append_block**(*base_element_block* &dest, const *base_element_block* &src)

static inline void **append_values_from_block**(*base_element_block* &dest, const *base_element_block* &src, std::size_t begin_pos, std::size_t len)

static inline void **assign_values_from_block**(*base_element_block* &dest, const *base_element_block* &src, std::size_t begin_pos, std::size_t len)

static inline void **prepend_values_from_block**(*base_element_block* &dest, const *base_element_block* &src, std::size_t begin_pos, std::size_t len)

static inline void **swap_values**(*base_element_block* &blk1, *base_element_block* &blk2, std::size_t pos1, std::size_t pos2, std::size_t len)

static inline bool **equal_block**(const *base_element_block* &left, const *base_element_block* &right)

static inline void **overwrite_values**(*base_element_block* &block, std::size_t pos, std::size_t len)

static inline void **shrink_to_fit**(*base_element_block* &block)

static inline std::size_t **size**(const *base_element_block* &block)

5.4.4 Types

mdds::mtv::element_t

using `mdds::mtv::`**element_t** = int

constexpr *element_t* `mdds::mtv::`**element_type_empty** = -1

constexpr *element_t* `mdds::mtv::`**element_type_reserved_start** = 0

constexpr *element_t* `mdds::mtv::`**element_type_reserved_end** = 49

constexpr *element_t* `mdds::mtv::`**element_type_user_start** = 50

mdds::mtv::lu_factor_t

enum class `mdds::mtv::lu_factor_t` : int

Loop-unrolling factor with optional SIMD feature.

In each enumerator value, the first byte contains the loop-unrolling factor (either 0, 4, 8, 16 or 32), while the second byte stores SIMD flags.

Values:

enumerator **none**

enumerator **lu4**

enumerator **lu8**

enumerator **lu16**

enumerator **lu32**

enumerator **sse2_x64**

enumerator **sse2_x64_lu4**

enumerator **sse2_x64_lu8**

enumerator **sse2_x64_lu16**

enumerator **avx2_x64**

enumerator **avx2_x64_lu4**

enumerator **avx2_x64_lu8**

mdds::mtv::trace_method_t

enum class `mdds::mtv::trace_method_t` : int

Type of traced method.

An `accessor` in this context is a method whose call alone does not mutate the state of the container. All const methods are accessors. Note that some non-const methods that return non-const references to internal data are still considered accessors.

A `mutator` is a method that, when called, may change the state of the stored data immediately.

The `accessor_with_pos_hint` label signifies an accessor that takes a position hint as its first argument. Likewise, `mutator_with_pos_hint` signifies a mutator that takes a position hint as its first argument.

The `constructor` and `destructor` labels are hopefully self-explanatory.

Values:

enumerator **unspecified**

enumerator **accessor**

enumerator **accessor_with_pos_hint**

enumerator **mutator**

enumerator **mutator_with_pos_hint**

enumerator **constructor**

enumerator **destructor**

mdds::mtv::trace_method_properties_t

struct **trace_method_properties_t**

Struct containing the information about each traced method.

Public Members

trace_method_t **type** = *trace_method_t*::*unspecified*

const void ***instance** = nullptr

Memory address of the container instance the traced method belongs to. This is essentially the `this` pointer inside the traced method.

const char ***function_name** = nullptr

Name of the method.

std::string **function_args**

String containing the argument names as well as their values if available.

const char ***filepath** = nullptr

Path of the file where the method body is.

int **line_number** = -1

Line number of the first line of the traced method body.

5.4.5 Standard Element Blocks

The following types are automatically defined by default when including one of the following headers:

- `mdds/multi_type_vector.hpp`
- `mdds/multi_type_vector/aos/main.hpp`
- `mdds/multi_type_vector/soa/main.hpp`

To disable automatic definitions of these standard element block types, you must define the `MDDS_MTV_USE_STANDARD_ELEMENT_BLOCKS` macro and set its value to 0. Refer to the *Specifying different storage type* section for more details on when you may want to disable these block types.

Constants

constexpr *element_t* `mdds::mtv::`**`element_type_boolean`** = *element_type_reserved_start*

constexpr *element_t* `mdds::mtv::`**`element_type_int8`** = *element_type_reserved_start* + 1

constexpr *element_t* `mdds::mtv::`**`element_type_uint8`** = *element_type_reserved_start* + 2

constexpr *element_t* `mdds::mtv::`**`element_type_int16`** = *element_type_reserved_start* + 3

constexpr *element_t* `mdds::mtv::`**`element_type_uint16`** = *element_type_reserved_start* + 4

constexpr *element_t* `mdds::mtv::`**`element_type_int32`** = *element_type_reserved_start* + 5

constexpr *element_t* `mdds::mtv::`**`element_type_uint32`** = *element_type_reserved_start* + 6

constexpr *element_t* `mdds::mtv::`**`element_type_int64`** = *element_type_reserved_start* + 7

constexpr *element_t* `mdds::mtv::`**`element_type_uint64`** = *element_type_reserved_start* + 8

constexpr *element_t* `mdds::mtv::`**`element_type_float`** = *element_type_reserved_start* + 9

constexpr *element_t* `mdds::mtv::`**`element_type_double`** = *element_type_reserved_start* + 10

constexpr *element_t* `mdds::mtv::`**`element_type_string`** = *element_type_reserved_start* + 11

Block Types and Traits

using mdds::mtv::**boolean_element_block** = *default_element_block*<*element_type_boolean*, bool>

using mdds::mtv::**int8_element_block** = *default_element_block*<*element_type_int8*, int8_t>

using mdds::mtv::**uint8_element_block** = *default_element_block*<*element_type_uint8*, uint8_t>

using mdds::mtv::**int16_element_block** = *default_element_block*<*element_type_int16*, int16_t>

using mdds::mtv::**uint16_element_block** = *default_element_block*<*element_type_uint16*, uint16_t>

using mdds::mtv::**int32_element_block** = *default_element_block*<*element_type_int32*, int32_t>

using mdds::mtv::**uint32_element_block** = *default_element_block*<*element_type_uint32*, uint32_t>

using mdds::mtv::**int64_element_block** = *default_element_block*<*element_type_int64*, int64_t>

using mdds::mtv::**uint64_element_block** = *default_element_block*<*element_type_uint64*, uint64_t>

using mdds::mtv::**float_element_block** = *default_element_block*<*element_type_float*, float>

using mdds::mtv::**double_element_block** = *default_element_block*<*element_type_double*, double>

using mdds::mtv::**string_element_block** = *default_element_block*<*element_type_string*, std::string>

struct **standard_element_blocks_traits** : public mdds::mtv::*default_traits*

Public Types

using **block_funcs** = *element_block_funcs*<*boolean_element_block*, *int8_element_block*, *uint8_element_block*, *int16_element_block*, *uint16_element_block*, *int32_element_block*, *uint32_element_block*, *int64_element_block*, *uint64_element_block*, *float_element_block*, *double_element_block*, *string_element_block*>

5.4.6 Exceptions

class **element_block_error** : public mdds::*general_error*

Generic exception used for errors specific to element block operations.

Public Functions

inline **element_block_error**(const std::string &msg)

5.4.7 Macros

MDDS_MTV_DEFINE_ELEMENT_CALLBACKS(type, type_id, empty_value, block_type)

Defines required callback functions for multi_type_vector.

Parameters

- **type** – element value type.
- **type_id** – constant value used as an ID for the value type. It should be of type mdds::mtv::element_t.
- **empty_value** – value that should be used as the default "false" value for the value type.
- **block_type** – block type that stores the specified value type.

MDDS_MTV_DEFINE_ELEMENT_CALLBACKS_PTR(type, type_id, empty_value, block_type)

A variant of MDDS_MTV_DEFINE_ELEMENT_CALLBACKS that should be used for a pointer type.

5.4.8 Collection

template<typename **_MtvT**>

class **collection**

Special-purpose collection of multiple multi_type_vector instances to allow them to be traversed "sideways" i.e. orthogonal to the direction of the vector instances. All involved multi_type_vector instances must be of the same type and length.

Public Types

typedef *_MtvT* **mtv_type**

typedef *mtv_type*::size_type **size_type**

typedef detail::side_iterator<*mtv_type*> **const_iterator**

collection range.

Public Functions

collection()

template<typename **_T**>
collection(const *_T* &begin, const *_T* &end)

Constructor that takes the start and end iterators of the multi_type_vector instances to reference in the collection.

Parameters

- **begin** – iterator that references the first multi_type_vector instance to place in the collection.
- **end** – iterator that references the position past the last multi_type_vector instance to place in the collection.

const_iterator **begin**() const

Return an iterator that references the first element in the collection.

Returns
iterator that references the first element in the collection.

const_iterator **end**() const

Return an iterator that references the position past the last element in the collection.

Returns
iterator that references the position past the last element in the collection.

size_type **size**() const

Return the length of the vector instances stored in the collection. This will be equivalent of the length of each multi_type_vector instance, since all stored instances have the same length.

Returns
length of the stored multi_type_vector instances.

void **swap**(*collection* &other)

Swap the entire collection with another collection instance.

Parameters
other – another collection instance to swap contents with.

void **set_collection_range**(*size_type* start, *size_type* size)

Set the sub-range of the collection to iterate through.

For instance, if the collection consists of 100 multi_type_vector instances, and you want to iterate through only 50 of them starting from the second instance, you set the start index to 1 (as it's 0-based), and the size to 50.

Parameters

- **start** – 0-based index of the first multi_type_vector instance to iterate through.
- **size** – length of the collection range i.e. the number of vector instances to iterate through starting from the specified first vector instance.

void **set_element_range**(*size_type* start, *size_type* size)

Set the sub element range to iterate through. This limits the element range in each multi_type_vector instance to iterate through. The direction of the element range is orthogonal to the direction of the collection range.

For instance, if the collection consists of multiple multi_type_vector instances all of which have a length of 50, and you only wish to iterate from the 3rd element through the 10th element in each vector instance, then you set the start index to 2 and the size to 8.

Parameters

- **start** – 0-based index of the starting element position.
- **size** – length of the element range to iterate through starting from the specified start element position.

CHAPTER

SIX

MULTI TYPE MATRIX

6.1 API Reference

template<typename **Traits**>

class **multi_type_matrix**

Matrix that can store numeric, integer, boolean, empty and string types. The string and integer types can be specified in the matrix trait template parameter. To use std::string as the string type and int as the integer type, use *mdds::mtm::std_string_traits*.

Internally it uses mdds::multi_type_vector as its value store. The element values are linearly stored in column-major order.

Public Types

typedef traits_type::string_element_block **string_block_type**

typedef traits_type::integer_element_block **integer_block_type**

typedef *string_block_type*::value_type **string_type**

typedef *integer_block_type*::value_type **integer_type**

typedef size_t **size_type**

typedef store_type::position_type **position_type**

typedef store_type::const_position_type **const_position_type**

typedef store_type::element_block_type **element_block_type**

typedef mtv::*boolean_element_block* **boolean_block_type**

typedef mtv::*double_element_block* **numeric_block_type**

Public Functions

multi_type_matrix()

Default constructor.

multi_type_matrix(*size_type* rows, *size_type* cols)

Construct a matrix of specified size.

Parameters

- **rows** – size of rows.
- **cols** – size of columns.

template<typename **_T**>
multi_type_matrix(*size_type* rows, *size_type* cols, const *_T* &value)

Construct a matrix of specified size and initialize all its elements with specified value.

Parameters

- **rows** – size of rows.
- **cols** – size of columns.
- **value** – value to initialize all its elements with.

template<typename **_T**>
multi_type_matrix(*size_type* rows, *size_type* cols, const *_T* &it_begin, const *_T* &it_end)

Construct a matrix of specified size and initialize its elements with specified values. The values are assigned to 2-dimensional matrix layout in column-major order. The size of the value array must equal `rows` x `cols`.

Parameters

- **rows** – size of rows.
- **cols** – size of columns.
- **it_begin** – iterator that points to the value of the first element.
- **it_end** – iterator that points to the position after the last element value.

multi_type_matrix(const *multi_type_matrix* &r)

Copy constructor.

~multi_type_matrix()

Destructor.

bool **operator==**(const *multi_type_matrix* &other) const

bool **operator!=**(const *multi_type_matrix* &other) const

multi_type_matrix &**operator=**(const *multi_type_matrix* &r)

position_type **position**(*size_type* row, *size_type* col)

Get a mutable reference of an element (position object) at specified position. The position object can then be passed to an additional method to get the type or value of the element it references, or set a new value to it.

Parameters

- **row** – row position of the referenced element.
- **col** – column position of the referenced element.

Returns
reference object of element at specified position.

position_type **position**(const *position_type* &pos_hint, *size_type* row, *size_type* col)

Get a mutable reference of an element (position object) at specified position. The position object can then be passed to an additional method to get the type or value of the element it references, or set a new value to it.

Parameters

- **pos_hint** – position object to be used as a position hint for faster lookup.
- **row** – row position of the referenced element.
- **col** – column position of the referenced element.

Returns
reference object of element at specified position.

const_position_type **position**(*size_type* row, *size_type* col) const

Get an immutable reference of an element (position object) at specified position. The position object can then be passed to an additional method to get the type or value of the element it references.

Parameters

- **row** – row position of the referenced element.
- **col** – column position of the referenced element.

Returns
reference object of element at specified position.

const_position_type **position**(const *const_position_type* &pos_hint, *size_type* row, *size_type* col) const

Get an immutable reference of an element (position object) at specified position. The position object can then be passed to an additional method to get the type or value of the element it references.

Parameters

- **pos_hint** – position object to be used as a position hint for faster lookup.
- **row** – row position of the referenced element.
- **col** – column position of the referenced element.

Returns
reference object of element at specified position.

size_pair_type **matrix_position**(const *const_position_type* &pos) const

Get the row and column positions of the current element from a position object.

Parameters
pos – position object.

Returns
0-based row and column positions.

position_type **end_position**()

Return a position type that represents an end position. This can be used to compare with another position object to see if it is past the last element position.

Returns
end position object.

const_position_type **end_position**() const

Return a position type that represents an end position. This can be used to compare with another position object to see if it is past the last element position.

Returns
end position object.

mtm::*element_t* **get_type**(const *const_position_type* &pos) const

Get the type of element from a position object. The type can be one of empty, string, numeric, or boolean.

Parameters
pos – position object of an element

Returns
element type.

mtm::*element_t* **get_type**(*size_type* row, *size_type* col) const

Get the type of element specified by its position. The type can be one of empty, string, numeric, or boolean.

Returns
element type.

double **get_numeric**(*size_type* row, *size_type* col) const

Get a numeric representation of the element. If the element is of numeric type, its value is returned. If it's of boolean type, either 1 or 0 is returned depending on whether it's true or false. If it's of empty or string type, 0 is returned.

Parameters

- **row** – row position of the element.
- **col** – column position of the element.

Returns
numeric representation of the element.

double **get_numeric**(const *const_position_type* &pos) const

Get a numeric representation of the element from a position object. If the element is of numeric type, its value is returned. If it's of boolean type, either 1 or 0 is returned depending on whether it's true or false. If it's of empty or string type, 0 is returned.

Parameters
pos – position object of an element

Returns
numeric representation of the element.

integer_type **get_integer**(*size_type* row, *size_type* col) const

Get an integer representation of the element. If the element is of integer type, its value is returned. If it's of boolean type, either 1 or 0 is returned depending on whether it's true or false. If it's of empty or string type, 0 is returned.

Parameters

- **row** – row position of the element.
- **col** – column position of the element.

Returns
integer representation of the element.

integer_type **get_integer**(const *const_position_type* &pos) const

Get an integer representation of the element. If the element is of integer type, its value is returned. If it's of boolean type, either 1 or 0 is returned depending on whether it's true or false. If it's of empty or string type, 0 is returned.

Parameters
pos – position object of an element

Returns
integer representation of the element.

bool **get_boolean**(*size_type* row, *size_type* col) const

Get a boolean representation of the element. If the element is of numeric type, true is returned if it's non-zero, otherwise false is returned. If it's of boolean type, its value is returned. If it's of empty or string type, false is returned.

Parameters

- **row** – row position of the element.
- **col** – column position of the element.

Returns
boolean representation of the element.

bool **get_boolean**(const *const_position_type* &pos) const

Get a boolean representation of the element from a position object. If the element is of numeric type, true is returned if it's non-zero, otherwise false is returned. If it's of boolean type, its value is returned. If it's of empty or string type, false is returned.

Parameters
pos – position object of an element

Returns
boolean representation of the element.

const *string_type* &**get_string**(*size_type* row, *size_type* col) const

Get the value of a string element. If the element is not of string type, it throws an exception.

Parameters

- **row** – row position of the element.
- **col** – column position of the element.

Returns
value of the element.

const *string_type* &**get_string**(const *const_position_type* &pos) const

Get the value of a string element from a position object. If the element is not of string type, it throws an exception.

Parameters
pos – position object of an element

Returns
value of the element.

template<typename **_T**>
_T **get**(*size_type* row, *size_type* col) const

Get the value of element at specified position. The caller must explicitly specify the return type. If the element is not of the specified type, it throws an exception.

Parameters

- **row** – row position of the element.
- **col** – column position of the element.

Returns
value of the element.

void **set_empty**(*size_type* row, *size_type* col)

Set specified element position empty.

Parameters

- **row** – row position of the element.
- **col** – column position of the element.

void **set_empty**(*size_type* row, *size_type* col, *size_type* length)

Set a range of elements empty. The range starts from the position specified by the `row` and `col`, and extends downward first then to the right.

Parameters

- **row** – row position of the first element.
- **col** – column position of the first element.
- **length** – length of the range to set empty. When the length is greater than 1, the range extends downward first then to the right.

position_type **set_empty**(const *position_type* &pos)

Set element referenced by the position object empty.

Parameters
pos – position object that references element.

Returns
position of the element that has just been made empty.

void **set_column_empty**(*size_type* col)

Set the entire column empty.

Parameters
col – index of the column to empty.

void **set_row_empty**(*size_type* row)

Set the entire row empty.

Parameters
row – index of the row to empty.

void **set**(*size_type* row, *size_type* col, double val)

Set a numeric value to an element at specified position.

Parameters

- **row** – row index of the element.
- **col** – column index of the element.
- **val** – new value to set.

position_type **set**(const *position_type* &pos, double val)

Set a numeric value to an element at specified position.

Parameters

- **pos** – position of the element to update.
- **val** – new value to set.

Returns

position of the element block where the new value has been set.

void **set**(*size_type* row, *size_type* col, bool val)

Set a boolean value to an element at specified position.

Parameters

- **row** – row index of the element.
- **col** – column index of the element.
- **val** – new value to set.

position_type **set**(const *position_type* &pos, bool val)

Set a boolean value to an element at specified position.

Parameters

- **pos** – position of the element to update.
- **val** – new value to set.

Returns

position of the element where the new value has been set.

void **set**(*size_type* row, *size_type* col, const *string_type* &str)

Set a string value to an element at specified position.

Parameters

- **row** – row index of the element.
- **col** – column index of the element.
- **str** – new value to set.

position_type **set**(const *position_type* &pos, const *string_type* &str)

Set a string value to an element at specified position.

Parameters

- **pos** – position of the element to update.
- **str** – new value to set.

Returns

position of the element block where the new value has been set.

void **set**(*size_type* row, *size_type* col, *integer_type* val)

Set an integer value to an element at specified position.

Parameters

- **row** – row index of the element.
- **col** – column index of the element.

- **val** – new value to set.

position_type **set**(const *position_type* &pos, *integer_type* val)

Set an integer value to an element at specified position.

Parameters

- **pos** – position of the element to update.
- **val** – new value to set.

Returns
position of the element block where the new value has been set.

template<typename **_T**>
void **set**(*size_type* row, *size_type* col, const *_T* &it_begin, const *_T* &it_end)

Set values of multiple elements at once, starting at specified element position following the direction of columns. When the new value series does not fit in the first column, it gets wrapped into the next column(s).

The method will throw an `std::out_of_range` exception if the specified position is outside the current container range.

Parameters

- **row** – row position of the start element.
- **col** – column position of the start element.
- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

template<typename **_T**>
position_type **set**(const *position_type* &pos, const *_T* &it_begin, const *_T* &it_end)

Set values of multiple elements at once, starting at specified element position following the direction of columns. When the new value series does not fit in the first column, it gets wrapped into the next column(s).

Parameters

- **pos** – position of the first element.
- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

Returns
position of the first element that has been modified.

template<typename **_T**>
void **set_column**(*size_type* col, const *_T* &it_begin, const *_T* &it_end)

Set values of multiple elements at once in a single column. When the length of passed elements exceeds that of the column, it gets truncated to the column size.

Parameters

- **col** – column position
- **it_begin** – iterator that points to the begin position of the values being set.
- **it_end** – iterator that points to the end position of the values being set.

size_pair_type **size**() const

Return the size of matrix as a pair. The first value is the row size, while the second value is the column size.

Returns
matrix size as a value pair.

multi_type_matrix &**transpose**()

Transpose the stored matrix data.

Returns
reference to this matrix instance.

void **copy**(const *multi_type_matrix* &src)

Copy values from the passed matrix instance. If the size of the passed matrix is smaller, then the element values are copied by their positions, while the rest of the elements that fall outside the size of the passed matrix instance will remain unmodified. If the size of the passed matrix instance is larger, then only the elements within the size of this matrix instance will get copied.

Parameters
src – passed matrix object to copy element values from.

template<typename **_T**>
void **copy**(*size_type* rows, *size_type* cols, const *_T* &it_begin, const *_T* &it_end)

Copy values from an array or array-like container, to a specified sub-matrix range. The length of the array must match the number of elements in the destination range, else it will throw a *mdds::size_error*.

Parameters

- **rows** – row size of the destination range.
- **cols** – column size of the destination range.
- **it_begin** – iterator pointing to the beginning of the input array.
- **it_end** – iterator pointing to the end position of the input array.

void **resize**(*size_type* rows, *size_type* cols)

Resize the matrix to specified size. This method supports resizing to zero-sized matrix; however, either specifying the row or column size to zero will resize the matrix to 0 x 0. When resizing the matrix larger, empty elements will be inserted in the region where no elements existed prior to the call.

Parameters

- **rows** – new row size
- **cols** – new column size

template<typename **_T**>
void **resize**(*size_type* rows, *size_type* cols, const *_T* &value)

Resize the matrix to specified size and initial value. The initial value will be applied to new elements created when the specified size is larger than the current one.

Parameters

- **rows** – new row size
- **cols** – new column size
- **value** – initial value for new elements

void **clear**()

Empty the matrix.

bool **numeric**() const

Check whether or not this matrix is numeric. A numeric matrix contains only numeric or boolean elements. An empty matrix is not numeric.

Returns
true if the matrix contains only numeric or boolean elements, or false otherwise.

bool **empty**() const

Check whether or not this matrix is empty.

Returns
true if this matrix is empty, or false otherwise.

void **swap**(*multi_type_matrix* &r)

Swap the content of the matrix with another instance.

template<typename **FuncT**>
FuncT **walk**(*FuncT* func) const

Walk all element blocks that consist of the matrix.

Parameters
func – function object whose operator() gets called on each element block.

Returns
function object passed to this method.

template<typename **FuncT**>
FuncT **walk**(*FuncT* func, const *size_pair_type* &start, const *size_pair_type* &end) const

Walk through the element blocks in a sub-matrix range defined by start and end positions passed to this method.

Parameters

- **func** – function object whose operator() gets called on the element block.
- **start** – the column/row position of the upper-left corner of the sub-matrix.
- **end** – the column/row position of the lower-right corner of the sub-matrix. Both column and row must be greater or equal to those of the start position.

Returns
function object passed to this method.

template<typename **FuncT**>
FuncT **walk**(*FuncT* func, const *multi_type_matrix* &right) const

Walk through all element blocks in parallel with another matrix instance. It stops at the block boundaries of both matrix instances during the walk.

Parameters

- **func** – function object whose operator() gets called on each element block.
- **right** – another matrix instance to parallel-walk with.

template<typename **FuncT**>
FuncT **walk**(*FuncT* func, const *multi_type_matrix* &right, const *size_pair_type* &start, const *size_pair_type* &end) const

Walk through the element blocks in a sub-matrix range in parallel with another matrix instance. It stops at the block boundaries of both matrix instances during the walk. The sub-matrix range is defined by start and end positions passed to this method.

Parameters

- **func** – function object whose operator() gets called on each element block.
- **right** – another matrix instance to parallel-walk with.

- **start** – the column/row position of the upper-left corner of the sub-matrix.
- **end** – the column/row position of the lower-right corner of the sub-matrix. Both column and row must be greater or equal to those of the start position.

Public Static Functions

static inline mtm::*element_t* **to_mtm_type**(mdds::mtv::*element_t* mtv_type)

static *position_type* **next_position**(const *position_type* &pos)

Move to the next logical position. The movement is in the top-to-bottom then left-to-right direction.

Parameters
pos – position object.

Returns
position object that references the element at the next logical position.

static *const_position_type* **next_position**(const *const_position_type* &pos)

Move to the next logical position. The movement is in the top-to-bottom then left-to-right direction.

Parameters
pos – position object.

Returns
non-mutable position object that references the element at the next logical position.

struct **element_block_node_type**

Public Functions

element_block_node_type()

element_block_node_type(const *element_block_node_type* &other)

template<typename **_Blk**>
_Blk::const_iterator **begin**() const

template<typename **_Blk**>
_Blk::const_iterator **end**() const

Public Members

mtm::*element_t* **type**

size_type **offset**

size_type **size**

const *element_block_type* ***data**

Friends

friend class multi_type_matrix

struct **size_pair_type**

Public Functions

inline **size_pair_type**()

inline **size_pair_type**(*size_type* _row, *size_type* _column)

inline **size_pair_type**(std::initializer_list<*size_type*> vs)

inline bool **operator==**(const *size_pair_type* &r) const

inline bool **operator!=**(const *size_pair_type* &r) const

Public Members

size_type **row**

size_type **column**

struct **std_string_traits**

Default matrix trait that uses std::string as its string type.

Public Types

typedef mdds::mtv::*int32_element_block* **integer_element_block**

typedef mdds::mtv::*string_element_block* **string_element_block**

enum **mdds::mtm::element_t**

Element type for *multi_type_matrix*.

Values:

enumerator **element_empty**

enumerator **element_boolean**

enumerator **element_string**

enumerator **element_numeric**

enumerator **element_integer**

CHAPTER

SEVEN

SORTED STRING MAP

7.1 API Reference

template<typename **ValueT**, template<typename, typename> class **EntryT** = chars_map_entry>

class **sorted_string_map**

sorted_string_map provides an efficient way to map string keys to arbitrary values, provided that the keys are known at compile time and are sorted in ascending order.

Public Types

using **value_type** = *ValueT*

using **size_type** = std::size_t

using **entry** = *EntryT*<*ValueT*, *size_type*>

Public Functions

sorted_string_map(const *entry* *entries, *size_type* entry_size, *value_type* null_value)

Constructor.

Parameters

- **entries** – pointer to the array of key-value entries.
- **entry_size** – size of the key-value entry array.
- **null_value** – null value to return when the find method fails to find a matching entry.

value_type **find**(const char *input, *size_type* len) const

Find a value associated with a specified string key.

Parameters

- **input** – pointer to a C-style string whose value represents the key to match.
- **len** – length of the matching string value.

Returns

value associated with the key, or the null value in case the key is not found.

value_type **find**(std::string_view input) const

Find a value associated with a specified string key.

Parameters
input – string key to match.

Returns
value associated with the key, or the null value in case the key is not found.

size_type **size**() const

Return the number of entries in the map. Since the number of entries is statically defined at compile time, this method always returns the same value.

Returns
the number of entries in the map.

CHAPTER

EIGHT

TRIE MAPS

8.1 Examples

8.1.1 Populating Trie Map

This section illustrates how to use *trie_map* to build a database of city populations and perform prefix searches. In this example, we will use the 2013 populations of cities in North Carolina, and use the city names as keys.

Let's define the type first:

```
using trie_map_type = mdds::trie_map<mdds::trie::std_string_traits, int>;
```

The first template argument specifies the trait of the key. In this example, we are using a pre-defined trait for std::string, which is defined in *std_string_traits*. The second template argument specifies the value type, which in this example is simply an `int`.

Once the type is defined, the next step is instantiation:

```
trie_map_type nc_cities;
```

It's pretty simple as you don't need to pass any arguments to the constructor. Now, let's populate this data structure with some population data:

```
// Insert key-value pairs.
nc_cities.insert("Charlotte",     792862);
nc_cities.insert("Raleigh",       431746);
nc_cities.insert("Greensboro",    279639);
nc_cities.insert("Durham",        245475);
nc_cities.insert("Winston-Salem", 236441);
nc_cities.insert("Fayetteville",  204408);
nc_cities.insert("Cary",          151088);
nc_cities.insert("Wilmington",    112067);
nc_cities.insert("High Point",    107741);
nc_cities.insert("Greenville",    89130);
nc_cities.insert("Asheville",     87236);
nc_cities.insert("Concord",       83506);
nc_cities.insert("Gastonia",      73209);
nc_cities.insert("Jacksonville",  69079);
nc_cities.insert("Chapel Hill",   59635);
nc_cities.insert("Rocky Mount",   56954);
nc_cities.insert("Burlington",    51510);
nc_cities.insert("Huntersville",  50458);
```

(continues on next page)

(continued from previous page)

```
nc_cities.insert("Wilson",         49628);
nc_cities.insert("Kannapolis",     44359);
nc_cities.insert("Apex",           42214);
nc_cities.insert("Hickory",        40361);
nc_cities.insert("Goldsboro",      36306);
```

It's pretty straight-forward. Each *insert()* call expects a pair of string key and an integer value. You can insert your data in any order regardless of key's sort order.

Now that the data is in, let's perform prefix search to query all cities whose name begins with "Cha":

```
cout << "Cities that start with 'Cha' and their populations:" << endl;
auto results = nc_cities.prefix_search("Cha");
for (const auto& kv : results)
{
    cout << "  " << kv.first << ": " << kv.second << endl;
}
```

You can perform prefix search via *prefix_search()* method, which returns a results object that can be iterated over using a range-based for loop. Running this code will produce the following output:

```
Cities that start with 'Cha' and their populations:
  Chapel Hill: 59635
  Charlotte: 792862
```

Let's perform another prefix search, this time with a prefix of "W":

```
cout << "Cities that start with 'W' and their populations:" << endl;
results = nc_cities.prefix_search("W");
for (const auto& kv : results)
{
    cout << "  " << kv.first << ": " << kv.second << endl;
}
```

You'll see the following output when running this code:

```
Cities that start with 'W' and their populations:
  Wilmington: 112067
  Wilson: 49628
  Winston-Salem: 236441
```

Note that the results are sorted in key's ascending order.

Note: Results from the prefix search are sorted in key's ascending order.

8.1.2 Creating Packed Trie Map from Trie Map

There is also another variant of trie called *packed_trie_map* which is designed to store all its data in contiguous memory region. Unlike *trie_map* which is mutable, *packed_trie_map* is immutable; once populated, you can only perform queries and it is no longer possible to add new entries into the container.

One way to create an instance of *packed_trie_map* is from *trie_map* by calling its *pack()* method:

```
auto packed = nc_cities.pack();
```

The query methods of *packed_trie_map* are identical to those of *trie_map*. For instance, performing prefix search to find all entries whose key begins with "C" can be done as follows:

```
cout << "Cities that start with 'C' and their populations:" << endl;
auto packed_results = packed.prefix_search("C");
for (const auto& kv : packed_results)
{
    cout << "  " << kv.first << ": " << kv.second << endl;
}
```

Running this code will generate the following output:

```
Cities that start with 'C' and their populations:
  Cary: 151088
  Chapel Hill: 59635
  Charlotte: 792862
  Concord: 83506
```

You can also perform an exact-match query via *find()* method which returns an iterator associated with the key-value pair entry:

```
// Individual search.
auto it = packed.find("Wilmington");
cout << "Population of Wilmington: " << it->second << endl;
```

You'll see the following output with this code:

```
Population of Wilmington: 112067
```

What if you performed an exact-match query with a key that doesn't exist in the container? You will basically get the end iterator position as its return value. Thus, running this code:

```
// You get an end position iterator when the container doesn't have the
// specified key.
it = packed.find("Asheboro");

cout << "Population of Asheboro: ";

if (it == packed.end())
    cout << "not found";
else
    cout << it->second;

cout << endl;
```

will generate the following output:

```
Population of Asheboro: not found
```

The complete source code for the examples in these two sections is available here.

8.1.3 Using Packed Trie Map directly

In the previous example, we showed a way to create an instance of *packed_trie_map* from a populated instance of *trie_map*. There is also a way to instantiate and populate an instance of *packed_trie_map* directly, and that is what we will cover in this section.

First, declare the type:

```
using trie_map_type = mdds::packed_trie_map<mdds::trie::std_string_traits, int>;
```

Once again, we are using the pre-defined trait for std::string as its key, and int as its value type. The next step is to prepare its entries ahead of time:

```
trie_map_type::entry entries[] =
{
    { MDDS_ASCII("Apex"),           42214 },
    { MDDS_ASCII("Asheville"),      87236 },
    { MDDS_ASCII("Burlington"),     51510 },
    { MDDS_ASCII("Cary"),          151088 },
    { MDDS_ASCII("Chapel Hill"),    59635 },
    { MDDS_ASCII("Charlotte"),     792862 },
    { MDDS_ASCII("Concord"),        83506 },
    { MDDS_ASCII("Durham"),        245475 },
    { MDDS_ASCII("Fayetteville"),  204408 },
    { MDDS_ASCII("Gastonia"),       73209 },
    { MDDS_ASCII("Goldsboro"),      36306 },
    { MDDS_ASCII("Greensboro"),    279639 },
    { MDDS_ASCII("Greenville"),     89130 },
    { MDDS_ASCII("Hickory"),        40361 },
    { MDDS_ASCII("High Point"),    107741 },
    { MDDS_ASCII("Huntersville"),   50458 },
    { MDDS_ASCII("Jacksonville"),   69079 },
    { MDDS_ASCII("Kannapolis"),     44359 },
    { MDDS_ASCII("Raleigh"),       431746 },
    { MDDS_ASCII("Rocky Mount"),    56954 },
    { MDDS_ASCII("Wilmington"),    112067 },
    { MDDS_ASCII("Wilson"),         49628 },
    { MDDS_ASCII("Winston-Salem"), 236441 },
};
```

We need to do this since *packed_trie_map* is immutable, and the only time we can populate its content is at instantiation time. Here, we are using the *MDDS_ASCII* macro to expand a string literal to its pointer value and size. Note that you need to ensure that the entries are sorted by the key in ascending order.

Warning: When instantiating *packed_trie_map* directly with a static set of entries, the entries must be sorted by the key in ascending order.

You can then pass this list of entries to construct the instance:

```
trie_map_type nc_cities(entries, MDDS_N_ELEMENTS(entries));
```

The *MDDS_N_ELEMENTS* macro will infer the size of a fixed-size array from its static definition. Once it's instantiated, the rest of the example for performing searches will be the same as in the previous section, which we will not repeat here.

The complete source code for the example in this section is available here.

8.1.4 Saving and loading Packed Trie Map instances

There are times when you need to save the state of a *packed_trie_map* instance to a file, or an in-memory buffer, and load it back later. Doing that is now possible by using the *save_state()* and *load_state()* member methods of the *packed_trie_map* class.

First, let's define the type of use:

```
using map_type = mdds::packed_trie_map<mdds::trie::std_string_traits, int>;
```

As with the previous examples, we will use `std::string` as the key type and `int` as the value type. In this example, we are going to use the world's largest cities and their 2018 populations as the data to store in the container.

The following code defines the entries:

```
std::vector<map_type::entry> entries =
{
    { MDDS_ASCII("Ahmedabad"),         7681000  },
    { MDDS_ASCII("Alexandria"),        5086000  },
    { MDDS_ASCII("Atlanta"),           5572000  },
    { MDDS_ASCII("Baghdad"),           6812000  },
    { MDDS_ASCII("Bangalore"),         11440000 },
    { MDDS_ASCII("Bangkok"),           10156000 },
    { MDDS_ASCII("Barcelona"),         5494000  },
    { MDDS_ASCII("Beijing"),           19618000 },
    { MDDS_ASCII("Belo Horizonte"),    5972000  },
    { MDDS_ASCII("Bogota"),            10574000 },
    { MDDS_ASCII("Buenos Aires"),      14967000 },
    { MDDS_ASCII("Cairo"),             20076000 },
    { MDDS_ASCII("Chengdu"),           8813000  },
    { MDDS_ASCII("Chennai"),           10456000 },
    { MDDS_ASCII("Chicago"),           8864000  },
    { MDDS_ASCII("Chongqing"),         14838000 },
    { MDDS_ASCII("Dalian"),            5300000  },
    { MDDS_ASCII("Dallas"),            6099000  },
    { MDDS_ASCII("Dar es Salaam"),     6048000  },
    { MDDS_ASCII("Delhi"),             28514000 },
    { MDDS_ASCII("Dhaka"),             19578000 },
    { MDDS_ASCII("Dongguan"),          7360000  },
    { MDDS_ASCII("Foshan"),            7236000  },
    { MDDS_ASCII("Fukuoka"),           5551000  },
    { MDDS_ASCII("Guadalajara"),       5023000  },
    { MDDS_ASCII("Guangzhou"),         12638000 },
    { MDDS_ASCII("Hangzhou"),          7236000  },
    { MDDS_ASCII("Harbin"),            6115000  },
    { MDDS_ASCII("Ho Chi Minh City"), 8145000  },
```

(continues on next page)

(continued from previous page)

```
    { MDDS_ASCII("Hong Kong"),         7429000  },
    { MDDS_ASCII("Houston"),          6115000  },
    { MDDS_ASCII("Hyderabad"),        9482000  },
    { MDDS_ASCII("Istanbul"),         14751000 },
    { MDDS_ASCII("Jakarta"),          10517000 },
    { MDDS_ASCII("Jinan"),            5052000  },
    { MDDS_ASCII("Johannesburg"),     5486000  },
    { MDDS_ASCII("Karachi"),          15400000 },
    { MDDS_ASCII("Khartoum"),         5534000  },
    { MDDS_ASCII("Kinshasa"),         13171000 },
    { MDDS_ASCII("Kolkata"),          14681000 },
    { MDDS_ASCII("Kuala Lumpur"),     7564000  },
    { MDDS_ASCII("Lagos"),            13463000 },
    { MDDS_ASCII("Lahore"),           11738000 },
    { MDDS_ASCII("Lima"),             10391000 },
    { MDDS_ASCII("London"),           9046000  },
    { MDDS_ASCII("Los Angeles"),      12458000 },
    { MDDS_ASCII("Luanda"),           7774000  },
    { MDDS_ASCII("Madrid"),           6497000  },
    { MDDS_ASCII("Manila"),           13482000 },
    { MDDS_ASCII("Mexico City"),      21581000 },
    { MDDS_ASCII("Miami"),            6036000  },
    { MDDS_ASCII("Moscow"),           12410000 },
    { MDDS_ASCII("Mumbai"),           19980000 },
    { MDDS_ASCII("Nagoya"),           9507000  },
    { MDDS_ASCII("Nanjing"),          8245000  },
    { MDDS_ASCII("New York City"),    18819000 },
    { MDDS_ASCII("Osaka"),            19281000 },
    { MDDS_ASCII("Paris"),            10901000 },
    { MDDS_ASCII("Philadelphia"),     5695000  },
    { MDDS_ASCII("Pune"),             6276000  },
    { MDDS_ASCII("Qingdao"),          5381000  },
    { MDDS_ASCII("Rio de Janeiro"),   13293000 },
    { MDDS_ASCII("Riyadh"),           6907000  },
    { MDDS_ASCII("Saint Petersburg"), 5383000  },
    { MDDS_ASCII("Santiago"),         6680000  },
    { MDDS_ASCII("Sao Paulo"),        21650000 },
    { MDDS_ASCII("Seoul"),            9963000  },
    { MDDS_ASCII("Shanghai"),         25582000 },
    { MDDS_ASCII("Shenyang"),         6921000  },
    { MDDS_ASCII("Shenzhen"),         11908000 },
    { MDDS_ASCII("Singapore"),        5792000  },
    { MDDS_ASCII("Surat"),            6564000  },
    { MDDS_ASCII("Suzhou"),           6339000  },
    { MDDS_ASCII("Tehran"),           8896000  },
    { MDDS_ASCII("Tianjin"),          13215000 },
    { MDDS_ASCII("Tokyo"),            37400068 },
    { MDDS_ASCII("Toronto"),          6082000  },
    { MDDS_ASCII("Washington, D.C."), 5207000  },
    { MDDS_ASCII("Wuhan"),            8176000  },
    { MDDS_ASCII("Xi'an"),            7444000  },
    { MDDS_ASCII("Yangon"),           5157000  },
```

(continues on next page)

(continued from previous page)

```
};
```

It's a bit long as it contains entries for 81 cities. We are then going to create an instance of the *packed_trie_map* class directly:

```
map_type cities(entries.data(), entries.size());
```

Let's print the size of the container to make sure the container has been successfully populated:

```
cout << "Number of cities: " << cities.size() << endl;
```

You will see the following output:

```
Number of cities: 81
```

if the container has been successfully populated. Now, let's run a prefix search on names beginning with an 'S':

```
cout << "Cities that begin with 'S':" << endl;
auto results = cities.prefix_search("S");
for (const auto& city : results)
    cout << "  * " << city.first << ": " << city.second << endl;
```

to make sure you get the following ten cities and their populations as the output:

```
Cities that begin with 'S':
  * Saint Petersburg: 5383000
  * Santiago: 6680000
  * Sao Paulo: 21650000
  * Seoul: 9963000
  * Shanghai: 25582000
  * Shenyang: 6921000
  * Shenzhen: 11908000
  * Singapore: 5792000
  * Surat: 6564000
  * Suzhou: 6339000
```

So far so good. Next, we will use the *save_state()* method to dump the internal state of this container to a file named **cities.bin**:

```
std::ofstream outfile("cities.bin", std::ios::binary);
cities.save_state(outfile);
```

This will create a file named **cities.bin** which contains a binary blob representing the content of this container in the current working directory. Run the `ls -l cities.bin` command to make sure the file has been created:

```
-rw-r--r-- 1 kohei kohei 17713 Jun 20 12:49 cities.bin
```

Now that the state of the container has been fully serialized to a file, let's work on restoring its content in another, brand-new instance of *packed_trie_map*.

```
map_type cities_loaded;

std::ifstream infile("cities.bin", std::ios::binary);
cities_loaded.load_state(infile);
```

Here, we used the *load_state()* method to restore the state from the file we have previously created. Let's make sure that this new instance has content equivalent to that of the original:

```
cout << "Equal to the original? " << std::boolalpha << (cities == cities_loaded) << endl;
```

If you see the following output:

```
Equal to the original? true
```

then this new instance has equivalent contant as the original one. Let's also make sure that it contains the same number of entries as the original:

```
cout << "Number of cities: " << cities_loaded.size() << endl;
```

Hopefully you will see the following output:

```
Number of cities: 81
```

Lastly, let's run on this new instance the same prefix search we did on the original instance, to make sure we still get the same results:

```
cout << "Cities that begin with 'S':" << endl;
auto results = cities_loaded.prefix_search("S");
for (const auto& city : results)
    cout << "  * " << city.first << ": " << city.second << endl;
```

You should see the following output:

```
Cities that begin with 'S':
  * Saint Petersburg: 5383000
  * Santiago: 6680000
  * Sao Paulo: 21650000
  * Seoul: 9963000
  * Shanghai: 25582000
  * Shenyang: 6921000
  * Shenzhen: 11908000
  * Singapore: 5792000
  * Surat: 6564000
  * Suzhou: 6339000
```

which is the same output we saw in the first prefix search.

The complete source code for this example is found here.

8.1.5 Saving Packed Trie Map with custom value type

In the previos example, you didn't have to explicitly specify the serializer type to the *save_state()* and *load_state()* methods, even though these two methods require the serializer type as their template arguments. That's because the library provides default serializer types for

- numeric value types i.e. integers, float and double,
- `std::string`, and
- the standard sequence types, such as `std::vector`, whose elements are of numeric value types,

and the previous example used `int` as the value type.

In this section, we are going to illustrate how you can write your own custom serializer to allow serialization of your own custom value type. In this example, we are going to use the list of presidents of the United States, with the names of the presidents as the keys, and their years of inauguration and political affiliations as the values.

We will use the following structure to store the values:

```
enum affiliated_party_t : uint8_t
{
    unaffiliated = 0,
    federalist,
    democratic_republican,
    democratic,
    whig,
    republican,
    national_union,
    republican_national_union,
};

struct us_president
{
    uint16_t year;
    affiliated_party_t party;
};
```

Each entry stores the year as a 16-bit integer and the affiliated party as an enum value of 8-bit width.

Next, let's define the container type:

```
using map_type = mdds::packed_trie_map<mdds::trie::std_string_traits, us_president>;
```

As with the previous example, the first step is to define the entries that are sorted by the keys, which in this case are the president's names:

```
std::vector<map_type::entry> entries =
{
    { MDDS_ASCII("Abraham Lincoln"),        { 1861, republican_national_union } },
    { MDDS_ASCII("Andrew Jackson"),         { 1829, democratic                } },
    { MDDS_ASCII("Andrew Johnson"),         { 1865, national_union            } },
    { MDDS_ASCII("Barack Obama"),           { 2009, democratic                } },
    { MDDS_ASCII("Benjamin Harrison"),      { 1889, republican                } },
    { MDDS_ASCII("Bill Clinton"),           { 1993, democratic                } },
    { MDDS_ASCII("Calvin Coolidge"),        { 1923, republican                } },
    { MDDS_ASCII("Chester A. Arthur"),      { 1881, republican                } },
    { MDDS_ASCII("Donald Trump"),           { 2017, republican                } },
    { MDDS_ASCII("Dwight D. Eisenhower"),   { 1953, republican                } },
    { MDDS_ASCII("Franklin D. Roosevelt"),  { 1933, democratic                } },
    { MDDS_ASCII("Franklin Pierce"),        { 1853, democratic                } },
    { MDDS_ASCII("George H. W. Bush"),      { 1989, republican                } },
    { MDDS_ASCII("George W. Bush"),         { 2001, republican                } },
    { MDDS_ASCII("George Washington"),      { 1789, unaffiliated              } },
    { MDDS_ASCII("Gerald Ford"),            { 1974, republican                } },
    { MDDS_ASCII("Grover Cleveland 1"),     { 1885, democratic                } },
    { MDDS_ASCII("Grover Cleveland 2"),     { 1893, democratic                } },
```

(continues on next page)

(continued from previous page)

```
    { MDDS_ASCII("Harry S. Truman"),         { 1945, democratic                } },
    { MDDS_ASCII("Herbert Hoover"),          { 1929, republican                } },
    { MDDS_ASCII("James A. Garfield"),       { 1881, republican                } },
    { MDDS_ASCII("James Buchanan"),          { 1857, democratic                } },
    { MDDS_ASCII("James K. Polk"),           { 1845, democratic                } },
    { MDDS_ASCII("James Madison"),           { 1809, democratic_republican     } },
    { MDDS_ASCII("James Monroe"),            { 1817, democratic_republican     } },
    { MDDS_ASCII("Jimmy Carter"),            { 1977, democratic                } },
    { MDDS_ASCII("John Adams"),              { 1797, federalist                } },
    { MDDS_ASCII("John F. Kennedy"),         { 1961, democratic                } },
    { MDDS_ASCII("John Quincy Adams"),       { 1825, democratic_republican     } },
    { MDDS_ASCII("John Tyler"),              { 1841, whig                      } },
    { MDDS_ASCII("Lyndon B. Johnson"),       { 1963, democratic                } },
    { MDDS_ASCII("Martin Van Buren"),        { 1837, democratic                } },
    { MDDS_ASCII("Millard Fillmore"),        { 1850, whig                      } },
    { MDDS_ASCII("Richard Nixon"),           { 1969, republican                } },
    { MDDS_ASCII("Ronald Reagan"),           { 1981, republican                } },
    { MDDS_ASCII("Rutherford B. Hayes"),     { 1877, republican                } },
    { MDDS_ASCII("Theodore Roosevelt"),      { 1901, republican                } },
    { MDDS_ASCII("Thomas Jefferson"),        { 1801, democratic_republican     } },
    { MDDS_ASCII("Ulysses S. Grant"),        { 1869, republican                } },
    { MDDS_ASCII("Warren G. Harding"),       { 1921, republican                } },
    { MDDS_ASCII("William Henry Harrison"), { 1841, whig                       } },
    { MDDS_ASCII("William Howard Taft"),     { 1909, republican                } },
    { MDDS_ASCII("William McKinley"),        { 1897, republican                } },
    { MDDS_ASCII("Woodrow Wilson"),          { 1913, democratic                } },
    { MDDS_ASCII("Zachary Taylor"),          { 1849, whig                      } },
};
```

Note that we need to add numeric suffixes to the entries for Grover Cleveland, who became president twice in two separate periods, in order to make the keys for his entries unique.

Now, proceed to create an instance of *packed_trie_map*:

```
map_type us_presidents(entries.data(), entries.size());
```

and inspect its size to make sure it is instantiated properly:

```
cout << "Number of entries: " << us_presidents.size() << endl;
```

You should see the following output:

```
Number of entries: 45
```

Before we proceed to save the state of this instance, let's define the custom serializer type first:

```
struct us_president_serializer
{
    union bin_buffer
    {
        char buffer[2];
        uint16_t i16;
        affiliated_party_t party;
```

(continues on next page)

(continued from previous page)

```
    };

    static constexpr bool variable_size = false;
    static constexpr size_t value_size = 3;

    static void write(std::ostream& os, const us_president& v)
    {
        bin_buffer buf;

        // Write the year value first.
        buf.i16 = v.year;
        os.write(buf.buffer, 2);

        // Write the affiliated party value.
        buf.party = v.party;
        os.write(buf.buffer, 1);
    }

    static void read(std::istream& is, size_t n, us_president& v)
    {
        // For a fixed-size value type, this should equal the defined value size.
        assert(n == 3);

        bin_buffer buf;

        // Read the year value.
        is.read(buf.buffer, 2);
        v.year = buf.i16;

        // Read the affiliated party value.
        is.read(buf.buffer, 1);
        v.party = buf.party;
    }
};
```

A custom value type can be either variable-size or fixed-size. For a variable-size value type, each value segment is preceded by the byte length of that segment. For a fixed-size value type, the byte length of all of the value segments is written only once up-front, followed by one or more value segments of equal byte length.

Since the value type in this example is fixed-size, we set the value of the `variable_size` static constant to false, and define the size of the value to 3 (bytes) as the `value_size` static constant. Keep in mind that you need to define the `value_size` constant *only* for fixed-size value types; if your value type is variable-size, you can leave it out.

Additionally, you need to define two static methods - one for writing to the output stream, and one for reading from the input stream. The write method must have the following signature:

```
static void write(std::ostream& os, const T& v);
```

where the `T` is the value type. In the body of this method you write to the output stream the bytes that represent the value. The length of the bytes you write must match the size specified by the `value_size` constant.

The read method must have the following signature:

```
static void read(std::istream& is, size_t n, T& v);
```

where the `T` is the value type, and the `n` specifies the length of the bytes you need to read for the value. For a fixed-size value type, the value of `n` should equal the `value_size` constant. Your job is to read the bytes off of the input stream for the length specified by the `n`, and populate the value instance passed to the method as the third argument.

Now that we have defined the custom serializer type, let's proceed to save the state to a file:

```
std::ofstream outfile("us-presidents.bin", std::ios::binary);
us_presidents.save_state<us_president_serializer>(outfile);
```

This time around, we are specifying the serializer type explicitly as the template argument to the *save_state()* method. Otherwise it is no different than what we did in the previous example.

Let's create another instance of *packed_trie_map* and restore the state back from the file we just created:

```
map_type us_presidents_loaded;

std::ifstream infile("us-presidents.bin", std::ios::binary);
us_presidents_loaded.load_state<us_president_serializer>(infile);
```

Once again, aside from explicitly specifying the serializer type as the template argument to the *load_state()* method, it is identical to the way we did in the previous example.

Let's compare the new instance with the old one to see if the two are equal:

```
cout << "Equal to the original? " << std::boolalpha << (us_presidents == us_presidents_
↪loaded) << endl;
```

The output says:

```
Equal to the original? true
```

They are. While we are at it, let's run a simple prefix search to find out all the US presidents whose first name is 'John':

```
cout << "Presidents whose first name is 'John':" << endl;
auto results = us_presidents_loaded.prefix_search("John");
for (const auto& entry : results)
    cout << "  * " << entry.first << " (" << entry.second.year << "; " << entry.second.
↪party << ")" << endl;
```

Here is the output:

```
Presidents whose first name is 'John':
  * John Adams (1797; Federalist)
  * John F. Kennedy (1961; Democratic)
  * John Quincy Adams (1825; Democratic Republican)
  * John Tyler (1841; Whig)
```

This looks like the correct results!

You can find the complete source code for this example here.

8.2 API Reference

8.2.1 Trie Map

template<typename **KeyTraits**, typename **ValueT**>

class **trie_map**

Trie map provides storage for multiple key-value pairs where keys are either strings, or otherwise consist of arrays of characters. The keys are stored in an ordered tree structure known as trie, or sometimes referred to as prefix tree.

Public Types

typedef *packed_trie_map*<*KeyTraits*, *ValueT*> **packed_type**

typedef *KeyTraits* **key_traits_type**

typedef *key_traits_type*::key_type **key_type**

typedef *key_traits_type*::key_buffer_type **key_buffer_type**

typedef *key_traits_type*::key_unit_type **key_unit_type**

typedef *ValueT* **value_type**

typedef size_t **size_type**

typedef std::pair<*key_type*, *value_type*> **key_value_type**

using **const_iterator** = trie::detail::const_iterator<*trie_map*>

using **iterator** = trie::detail::iterator<*trie_map*>

typedef trie::detail::search_results<*trie_map*> **search_results**

Public Functions

trie_map()

Default constructor.

trie_map(const *trie_map* &other)

trie_map(*trie_map* &&other)

const_iterator **begin**() const

iterator **begin**()

const_iterator **end**() const

iterator **end**()

trie_map &**operator=**(*trie_map* other)

void **swap**(*trie_map* &other)

void **insert**(const *key_type* &key, const *value_type* &value)

Insert a new key-value pair.

Parameters

- **key** – key value.
- **value** – value to associate with the key.

void **insert**(const *key_unit_type* *key, *size_type* len, const *value_type* &value)

Insert a new key-value pair.

Parameters

- **key** – pointer to the first character of a character array that stores key value.
- **len** – length of the character array storing the key.
- **value** – value to associate with the key.

bool **erase**(const *key_unit_type* *key, *size_type* len)

Erase a key and the value associated with it.

Parameters

- **key** – pointer to the first character of a character array that stores key value.
- **len** – length of the character array storing the key.

Returns

true if a key is erased, false otherwise.

const_iterator **find**(const *key_type* &key) const

Find a value associated with a specified key.

Parameters

key – key to match.

Returns

immutable iterator that references a value associated with the key, or the end position in case the key is not found.

const_iterator **find**(const *key_unit_type* *input, *size_type* len) const

Find a value associated with a specified key.

Parameters

- **input** – pointer to an array whose value represents the key to match.
- **len** – length of the matching key value.

Returns

immutable iterator that references a value associated with the key, or the end position in case the key is not found.

iterator **find**(const *key_type* &key)

Find a value associated with a specified key.

Parameters
key – key to match.

Returns
mutable iterator that references a value associated with the key, or the end position in case the key is not found.

iterator **find**(const *key_unit_type* *input, *size_type* len)

Find a value associated with a specified key.

Parameters

- **input** – pointer to an array whose value represents the key to match.
- **len** – length of the matching key value.

Returns
mutable iterator that references a value associated with the key, or the end position in case the key is not found.

search_results **prefix_search**(const *key_type* &prefix) const

Retrieve all key-value pairs whose keys start with specified prefix. You can also retrieve all key-value pairs by passing a null prefix and a length of zero.

Parameters
prefix – prefix to match.

Returns
results object that contains all matching key-value pairs. The results are sorted by the key in ascending order.

search_results **prefix_search**(const *key_unit_type* *prefix, *size_type* len) const

Retrieve all key-value pairs whose keys start with specified prefix. You can also retrieve all key-value pairs by passing a null prefix and a length of zero.

Parameters

- **prefix** – pointer to an array whose value represents the prefix to match.
- **len** – length of the prefix value to match.

Returns
results object that contains all matching key-value pairs. The results are sorted by the key in ascending order.

size_type **size**() const

Return the number of entries in the map.

Returns
the number of entries in the map.

bool **empty**() const noexcept

void **clear**()

Empty the container.

packed_type **pack**() const

Create a compressed and immutable version of the container which provides better search performance and requires much less memory footprint.

Returns
an instance of *mdds::packed_trie_map* with the same content.

Friends

friend class trie::detail::iterator_base< trie_map, true >

friend class trie::detail::iterator_base< trie_map, false >

friend class trie::detail::const_iterator< trie_map >

friend class trie::detail::iterator< trie_map >

friend class trie::detail::search_results< trie_map >

8.2.2 Packed Trie Map

template<typename **KeyTraits**, typename **ValueT**>

class **packed_trie_map**

An immutable trie container that packs its content into a contiguous array to achieve both space efficiency and lookup performance. The user of this data structure must provide a pre-constructed list of key-value entries that are sorted by the key in ascending order, or construct from an instance of *mdds::trie_map*.

Note that, since this container is immutable, the content of the container cannot be modified once constructed.

Public Types

typedef *KeyTraits* **key_traits_type**

typedef *key_traits_type*::key_type **key_type**

typedef *key_traits_type*::key_buffer_type **key_buffer_type**

typedef *key_traits_type*::key_unit_type **key_unit_type**

typedef *ValueT* **value_type**

typedef size_t **size_type**

typedef std::pair<*key_type*, *value_type*> **key_value_type**

typedef trie::detail::packed_iterator_base<*packed_trie_map*> **const_iterator**

typedef trie::detail::packed_search_results<*packed_trie_map*> **search_results**

Public Functions

packed_trie_map()

packed_trie_map(const *entry* *entries, *size_type* entry_size)

Constructor that initializes the content from a static list of key-value entries. The caller *must* ensure that the key-value entries are sorted in ascending order, else the behavior is undefined.

Parameters

- **entries** – pointer to the array of key-value entries.
- **entry_size** – size of the key-value entry array.

packed_trie_map(const *trie_map*<*key_traits_type*, *value_type*> &other)

Constructor to allow construction of an instance from the content of a *mdds::trie_map* instance.

Parameters

other – *mdds::trie_map* instance to build content from.

packed_trie_map(const *packed_trie_map* &other)

packed_trie_map(*packed_trie_map* &&other)

packed_trie_map &**operator=**(*packed_trie_map* other)

bool **operator==**(const *packed_trie_map* &other) const

bool **operator!=**(const *packed_trie_map* &other) const

const_iterator **begin**() const

const_iterator **end**() const

const_iterator **cbegin**() const

const_iterator **cend**() const

const_iterator **find**(const *key_type* &key) const

Find a value associated with a specified key.

Parameters

key – key to match.

Returns

iterator that references a value associated with the key, or the end position in case the key is not found.

const_iterator **find**(const *key_unit_type* *input, *size_type* len) const

Find a value associated with a specified key.

Parameters

- **input** – pointer to an array whose value represents the key to match.
- **len** – length of the matching key value.

Returns

iterator that references a value associated with the key, or the end position in case the key is not found.

search_results **prefix_search**(const *key_type* &prefix) const

Retrieve all key-value pairs whose keys start with specified prefix. You can also retrieve all key-value pairs by passing a null prefix and a length of zero.

Parameters
prefix – prefix to match.

Returns
results object containing all matching key-value pairs.

search_results **prefix_search**(const *key_unit_type* *prefix, *size_type* len) const

Retrieve all key-value pairs whose keys start with specified prefix. You can also retrieve all key-value pairs by passing a null prefix and a length of zero.

Parameters

- **prefix** – pointer to an array whose value represents the prefix to match.
- **len** – length of the prefix value to match.

Returns
results object that contains all matching key-value pairs. The results are sorted by the key in ascending order.

size_type **size**() const noexcept

Return the number of entries in the map.

Returns
the number of entries in the map.

bool **empty**() const noexcept

void **swap**(*packed_trie_map* &other)

template<typename **FuncT** = trie::*value_serializer*<*value_type*>>
void **save_state**(std::ostream &os) const

Save the state of the instance of this class to an external buffer.

Parameters
os – output stream to write the state to.

template<typename **FuncT** = trie::*value_serializer*<*value_type*>>
void **load_state**(std::istream &is)

Restore the state of the instance of this class from an external buffer.

Parameters
is – input stream to load the state from.

void **dump_structure**() const

Dump the structure of the trie content for debugging. You must define MDDS_TRIE_MAP_DEBUG in order for this method to generate output.

Friends

friend class trie::detail::packed_iterator_base< packed_trie_map >

friend class trie::detail::packed_search_results< packed_trie_map >

struct **entry**

Single key-value entry. Caller must provide at compile time a static array of these entries.

Public Functions

inline **entry**(const *key_unit_type* *_key, *size_type* _keylen, *value_type* _value)

Public Members

const *key_unit_type* ***key**

size_type **keylen**

value_type **value**

8.2.3 Traits

template<typename **ContainerT**>

struct **std_container_traits**

Template for a key type implemented using a typical STL container type.

Public Types

using **key_type** = *ContainerT*

type used to store a key value.

using **key_buffer_type** = *key_type*

type used to build an intermediate key value, from which a final key value is to be created. It is expected to be an array structure whose content is contiguous in memory. Its elements must be of key_unit_type.

using **key_unit_type** = typename *key_type*::value_type

type that represents a single character inside a key or a key buffer object. A key object is expected to store a series of elements of this type.

Public Static Functions

static inline *key_buffer_type* **to_key_buffer**(const *key_unit_type* *str, size_t length)

Function called to create and initialize a buffer object from a given initial key value.

Parameters

- **str** – pointer to the first character of the key value.
- **length** – length of the key value.

Returns
buffer object containing the specified key value.

static inline *key_buffer_type* **to_key_buffer**(const *key_type* &key)

Function called to create and initialize a buffer object from a given initial key value.

Parameters
key – key value

Returns
buffer object containing the specified key value.

static inline const *key_unit_type* ***buffer_data**(const *key_buffer_type* &buf)

static inline size_t **buffer_size**(const *key_buffer_type* &buf)

static inline void **push_back**(*key_buffer_type* &buffer, *key_unit_type* c)

Function called to append a single character to the end of a key buffer.

Parameters

- **buffer** – buffer object to append character to.
- **c** – character to append to the buffer.

static inline void **pop_back**(*key_buffer_type* &buffer)

Function called to remove a single character from the tail of an existing key buffer.

Parameters
buffer – buffer object to remove character from.

static inline *key_type* **to_key**(const *key_buffer_type* &buf)

Function called to create a final string object from an existing buffer.

Parameters
buf – buffer object to create a final string object from.

Returns
string object whose content is created from the buffer object.

using mdds::trie::**std_string_traits** = *std_container_traits*<std::string>

8.2.4 Value Serializers

template<typename **T**, typename **U** = void>

struct **value_serializer** : public mdds::trie::*numeric_value_serializer*<*T*>

Default value serializer for *mdds::packed_trie_map*'s load_state and save_state methods. The primary template is used for numeric value types, and template specializations exist for std::string, as well as sequence containers, such as std::vector, whose elements are of numeric types.

template<typename **T**>

struct **numeric_value_serializer**

Serializer for numeric data types.

Subclassed by *mdds::trie::value_serializer< T, U >*

Public Static Functions

static void **write**(std::ostream &os, const *T* &v)

static void **read**(std::istream &is, size_t n, *T* &v)

Public Static Attributes

static constexpr bool **variable_size** = false

static constexpr size_t **value_size** = sizeof(*T*)

template<typename **T**>

struct **variable_value_serializer**

Serializer for variable-size data types.

Public Static Functions

static void **write**(std::ostream &os, const *T* &v)

static void **read**(std::istream &is, size_t n, *T* &v)

Public Static Attributes

static constexpr bool **variable_size** = true

template<typename **T**>

struct **numeric_sequence_value_serializer**

Serializer for standard sequence container whose value type is of numeric value type.

Subclassed by mdds::trie::value_serializer< T, typename std::enable_if< mdds::detail::has_value_type< T >::value >::type >

Public Types

using **element_serializer** = *numeric_value_serializer*<typename *T*::value_type>

Public Static Functions

static void **write**(std::ostream &os, const *T* &v)

static void **read**(std::istream &is, size_t n, *T* &v)

Public Static Attributes

static constexpr bool **variable_size** = true

CHAPTER

NINE

R-TREE

9.1 Overview

R-tree is a tree-based data structure designed for optimal query performance on multi-dimensional spatial objects with rectangular bounding shapes. The R-tree implementation included in this library is a variant of R-tree known as R*-tree which differs from the original R-tree in that it may re-insert an object if the insertion of that object would cause the original target directory to overflow. Such re-insertions lead to more balanced tree which in turn lead to better query performance, at the expense of slightly more overhead at insertion time.

Our implementation of R-tree theoretically supports any number of dimensions although certain functionalities, especially those related to visualization, are only supported for 2-dimensional instances.

R-tree consists of three types of nodes. Value nodes store the values inserted externally and always sit at the bottom of the tree. Leaf directory nodes sit directly above the value nodes, and store only value nodes as their child nodes. The rest are all non-leaf directory nodes which can either store leaf or non-leaf directory nodes.

9.2 Quick start

Let's go through a very simple example to demonstrate how to use *rtree*. First, you need to specify a concrete type by specifying the key type and value type to use:

```
#include <mdds/rtree.hpp>

#include <string>
#include <iostream>

// key values are of type double, and we are storing std::string as a
// value for each spatial object.  By default, tree becomes 2-dimensional
// object store unless otherwise specified.
using rt_type = mdds::rtree<double, std::string>;
```

You'll only need to specify the types of key and value here unless you want to customize other properties of *rtree* including the number of dimensions. By default, *rtree* sets the number of dimensions to 2.

```
rt_type tree;
```

Instantiating an rtree instance should be no brainer as it requires no input parameters. Now, let's insert some data:

```
tree.insert({{0.0, 0.0}, {15.0, 20.0}}, "first rectangle data");
```

This inserts a string value associated with a bounding rectangle of (0, 0) - (15, 20). Note that in the above code we are passing the bounding rectangle parameter to rtree's *insert()* method as a nested initializer list, which implicitly gets converted to *extent_type*. You can also use the underlying type directly as follows:

```
rt_type::extent_type bounds({-2.0, -1.0}, {1.0, 2.0});
std::cout << "inserting value for " << bounds.to_string() << std::endl;
tree.insert(bounds, "second rectangle data");
```

which inserts a string value associated with a bounding rectangle of (-2, -1) to (1, 2). You may have noticed that this code also uses extent_type's *to_string()* method which returns a string representation of the bounding rectangle. This may come in handy when debugging your code. This method should work as long as the key type used in your rtree class overloads `std::ostream`'s << operator function.

Running this code will generate the following output:

```
inserting value for (-2, -1) - (1, 2)
```

As *extent_type* consists of two members called `start` and `end` both of which are of type *point_type*, which in turn contains an array of keys called `d` whose size equals the number of dimensions, you can modify the extent directly:

```
bounds.start.d[0] = -1.0; // Change the first dimension value of the start rectangle␣
↪point.
bounds.end.d[1] += 1.0; // Increment the second dimension value of the end rectangle␣
↪point.
std::cout << "inserting value for " << bounds.to_string() << std::endl;
tree.insert(bounds, "third rectangle data");
```

This code will insert a string value associated with a rectangle of (-1, -1) to (1, 3), and will generate the following output:

```
inserting value for (-1, -1) - (1, 3)
```

So far we have only inserted data associated with rectangle shapes, but *rtree* also allows data associated with points to co-exist in the same tree. The following code inserts a string value associated with a point (5, 6):

```
tree.insert({5.0, 6.0}, "first point data");
```

Like the verfy first rectangle data we've inserted, we are passing the point data as an initializer list of two elements (for 2-dimensional data storage), which will implicitly get converted to *point_type* before it enters into the call.

Now that some data have been inserted, it's time to run some queries. Let's query all objects that overlap with a certain rectangular region either partially or fully. The following code will do just that:

```
// Search for all objects that overlap with a (4, 4) - (7, 7) rectangle.
auto results = tree.search({{4.0, 4.0}, {7.0, 7.0}}, rt_type::search_type::overlap);

for (const std::string& v : results)
    std::cout << "value: " << v << std::endl;
```

In this query, we are specifying the search region to be (4, 4) to (7, 7) which should overlap with the first rectangle data and the first point data. Indeed, when you execute this code, you will see the following output:

```
value: first rectangle data
value: first point data
```

indicating that the query region does overlap with two of the stored values

Note that the *search()* method takes exactly two arguments; the first one specifies the search region while the second two specifies the type of search to be performed. In the above call we passed *search_type*'s `overlap` enum value which picks up all values whose bounding rectangles overlap with the search region either partially or fully.

Sometimes, however, you may need to find a value whose bounding rectangle matches exactly the search region you specify in your query. You can achieve that by setting the search type to `match`.

Here is an example:

```
// Search for all objects whose bounding rectangles are exactly (4, 4) - (7, 7).
auto results = tree.search({{4.0, 4.0}, {7.0, 7.0}}, rt_type::search_type::match);
std::cout << "number of results: " << std::distance(results.begin(), results.end()) <<␣
↪std::endl;
```

The search region is identical to that of the previous example, but the search type is set to `match` instead. Then the next line will count the number of results and print it out. The output you will see is as follows:

```
number of results: 0
```

indicating that the results are empty. That is expected since none of the objects stored in the tree have an exact bounding rectangle of (4, 4) - (7, 7). When you change the search region to (0, 0) - (15, 20), however, you'll get one object back. Here is the actual code:

```
// Search for all objects whose bounding rectangles are exactly (0, 0) - (15, 20).
auto results = tree.search({{0.0, 0.0}, {15.0, 20.0}}, rt_type::search_type::match);
std::cout << "number of results: " << std::distance(results.begin(), results.end()) <<␣
↪std::endl;
```

which is identical to the previous one except for the search resion. This is its output:

```
number of results: 1
```

indicating that it has found exactly one object whose bounding rectangle exactly matches the search region.

It's worth mentioning that *rtree* supports storage of multiple objects with identical bounding rectangle. As such, searching with the search type of `match` can return more than one result.

As you may have noticed in these example codes, the *search_results* object does provide *begin()* and *end()* methods that return standard iterators which you can plug into various iterator algorithms from the STL. Dereferencing the iterator will return a reference to the stored value i.e. this line:

```
std::cout << "value: " << *results.begin() << std::endl;
```

which immediately comes after the previous search will output:

```
value: first rectangle data
```

In addition to accessing the value that the iterator references, you can also query from the same iterator object the bounding rectangle associated with the value as well as its depth in the tree by calling its *extent()* and *depth()* methods, respectively, as in the following code:

```
auto it = results.begin();
std::cout << "value: " << *it << std::endl;
std::cout << "extent: " << it.extent().to_string() << std::endl;
std::cout << "depth: " << it.depth() << std::endl;
```

Running this code will produce the following output:

```
value: first rectangle data
extent: (0, 0) - (15, 20)
depth: 1
```

A depth value represents the distance of the node where the value is stored from the root node of the tree, and is technically 0-based. However, you will never see a depth of 0 in the search results since the root node of a R-tree is always a directory node, and a directory node only stores other child nodes and never a value (hence never appears in the search results).

9.3 Removing a value from tree

Removing an existing value from the tree first requires you to perform the search to obtian search results, then from the search results get the iterator and advance it to the position of the value you wish to remove. Once you have your iterator set to the right position, pass it to the *erase()* method to remove that value.

Note that you can only remove one value at a time, and the iterator becomes invalid each time you call the *erase()* method to remove a value.

Here is a contrived example to demonstrate how erasing a value works:

```
#include <mdds/rtree.hpp>

#include <string>
#include <iostream>

int main()
{
    using rt_type = mdds::rtree<int, std::string>;

    rt_type tree;

    // Insert multiple values at the same point.
    tree.insert({1, 1}, "A");
    tree.insert({1, 1}, "B");
    tree.insert({1, 1}, "C");
    tree.insert({1, 1}, "D");
    tree.insert({1, 1}, "E");

    // This should return all five values.
    auto results = tree.search({1, 1}, rt_type::search_type::match);

    for (const std::string& v : results)
        std::cout << v << std::endl;

    // Erase "C".
    for (auto it = results.begin(); it != results.end(); ++it)
    {
        if (*it == "C")
        {
            tree.erase(it);
            break; // This invalidates the iterator.  Bail out.
        }
```

(continues on next page)

(continued from previous page)

```
    }

    std::cout << "'C' has been erased." << std::endl;

    // Now this should only return A, B, D and E.
    results = tree.search({1, 1}, rt_type::search_type::match);

    for (const std::string& v : results)
        std::cout << v << std::endl;

    return EXIT_SUCCESS;
}
```

In this code, we are intentionally putting 5 values to the same 2-dimensional point (1, 1), then removing one of them based on matching criteria (of being equal to "C").

Compiling and running this code will generate the following output:

```
A
B
C
D
E
'C' has been erased.
A
B
D
E
```

which clearly shows that the 'C' has been successfully erased.

9.4 Visualize R-tree structure

In this section we will illustrate a way to visualize an R-tree structure via *export_tree()* method, which can be useful when you need to visually inspect the tree structure to see how well balanced it is (or not).

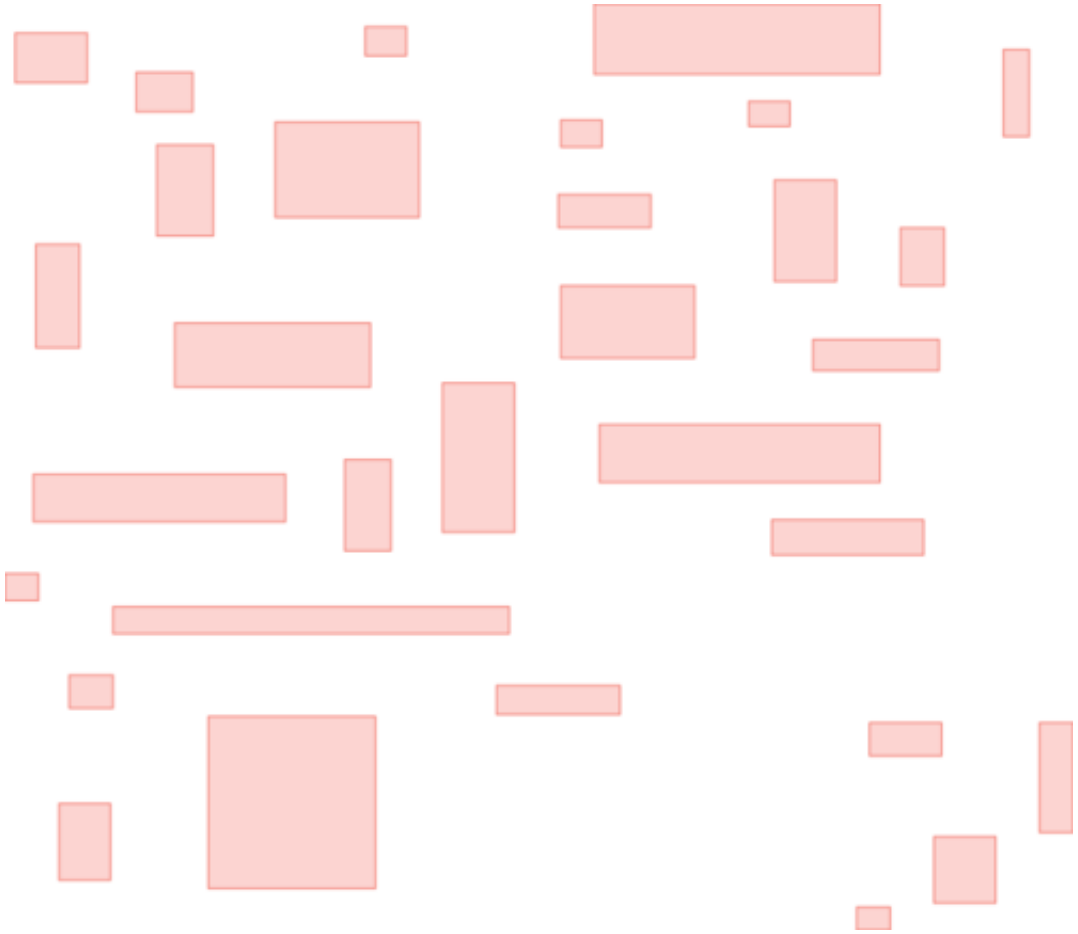
We will be using the following set of 2-dimensional rectangles as the bounding rectangles for input values.

For input values, we'll simply use linearly increasing series of integer values, but the values themselves are not the focus of this section, and we'll not talk much about that. We will also intentionally make the capacity of directory nodes smaller so that the tree will split more frequently during insertion even for smaller number of inputs.

Now, let's take a look at the code:

```
#include <mdds/rtree.hpp>

#include <iostream>
#include <fstream>

// Make the node capacity intentionally small.
struct tiny_trait_2d
{
    constexpr static size_t dimensions = 2;
```

(continues on next page)



(continued from previous page)

```
    constexpr static size_t min_node_size = 2;
    constexpr static size_t max_node_size = 5;
    constexpr static size_t max_tree_depth = 100;

    constexpr static bool enable_forced_reinsertion = true;
    constexpr static size_t reinsertion_size = 2;
};

using rt_type = mdds::rtree<int, int, tiny_trait_2d>;

int main()
{
    // 2D rectangle with the top-left position (x, y), width and height.
    struct rect
    {
        int x;
        int y;
        int w;
        int h;
    };

    std::vector<rect> rects =
    {
        {  3731,  2433, 1356,  937 },
        {  6003,  3172, 1066,  743 },
        {  4119,  6403,  825, 1949 },
        { 10305,  2315,  776,  548 },
        { 13930,  5468, 1742,  626 },
        {  8614,  4107, 2709, 1793 },
        { 14606,  1887, 5368, 1326 },
        { 17990,  5196, 1163, 1911 },
        {  6728,  7881, 3676, 1210 },
        { 14704,  9789, 5271, 1092 },
        {  4071, 10723, 4739,  898 },
        { 11755,  9010, 1357, 2806 },
        { 13978,  4068,  776,  509 },
        { 17507,  3717,  777,  471 },
        { 20358,  6092,  824, 1093 },
        {  6390,  4535, 1066, 1715 },
        { 13978,  7182, 2516, 1365 },
        { 17942, 11580, 2854,  665 },
        {  9919, 10450,  873, 1716 },
        {  5568, 13215, 7446,  509 },
        {  7357, 15277, 3145, 3234 },
        {  3539, 12592,  631,  509 },
        {  4747, 14498,  825,  626 },
        {  4554, 16913,  969, 1443 },
        { 12771, 14693, 2323,  548 },
        { 18714,  8193, 2372,  586 },
        { 22292,  2743,  487, 1638 },
        { 20987, 17535, 1163, 1249 },
        { 19536, 18859,  632,  431 },
```

(continues on next page)

(continued from previous page)

```
        { 19778, 15394, 1356,  626 },
        { 22969, 15394,  631, 2066 },
    };

    rt_type tree;

    // Insert the rectangle objects into the tree.
    int value = 0;
    for (const auto& rect : rects)
        tree.insert({{rect.x, rect.y}, {rect.x + rect.w, rect.y + rect.h}}, value++);

    // Export the tree structure as a SVG for visualization.
    std::string tree_svg = tree.export_tree(rt_type::export_tree_type::extent_as_svg);
    std::ofstream fout("bounds.svg");
    fout << tree_svg;

    return EXIT_SUCCESS;
}
```

First, we need to talk about how the concrete rtree type is instantiated:

```
// Make the node capacity intentionally small.
struct tiny_trait_2d
{
    constexpr static size_t dimensions = 2;
    constexpr static size_t min_node_size = 2;
    constexpr static size_t max_node_size = 5;
    constexpr static size_t max_tree_depth = 100;

    constexpr static bool enable_forced_reinsertion = true;
    constexpr static size_t reinsertion_size = 2;
};

using rt_type = mdds::rtree<int, int, tiny_trait_2d>;
```

The first and second template arguments specify the key and value types to be both `int`. This time around, however, we are passing a third template argument which is a struct containing several static constant values. These constant values define certain characteristics of your R-tree, and there are some restrictions you need to be aware of in case you need to use your own custom trait for your R-tree. Refer to *`default_rtree_traits`*, which is the default trait used when you don't specify your own, for the descriptions of the individual constants that your trait struct is expected to have as well as restrictions that you must be aware of.

Also be aware that these constants must all be constant expressions with `constexpr` specifiers, as some of them are used within `static_assert` declarations, and even those that are currently not used within `static_assert` may be used in `static_assert` in the future.

As far as our current example goes, the only part of the custom trait we need to highlight is that we are setting the directory node size to 2-to-5 instead of the default size of 40-to-100, to trigger more node splits and make the tree artificially deeper.

Let's move on to the next part of the code:

```
// 2D rectangle with the top-left position (x, y), width and height.
struct rect
```

(continues on next page)

(continued from previous page)

```
{
    int x;
    int y;
    int w;
    int h;
};

std::vector<rect> rects =
{
    {  3731,  2433, 1356,  937 },
    {  6003,  3172, 1066,  743 },
    {  4119,  6403,  825, 1949 },
    { 10305,  2315,  776,  548 },
    { 13930,  5468, 1742,  626 },
    {  8614,  4107, 2709, 1793 },
    { 14606,  1887, 5368, 1326 },
    { 17990,  5196, 1163, 1911 },
    {  6728,  7881, 3676, 1210 },
    { 14704,  9789, 5271, 1092 },
    {  4071, 10723, 4739,  898 },
    { 11755,  9010, 1357, 2806 },
    { 13978,  4068,  776,  509 },
    { 17507,  3717,  777,  471 },
    { 20358,  6092,  824, 1093 },
    {  6390,  4535, 1066, 1715 },
    { 13978,  7182, 2516, 1365 },
    { 17942, 11580, 2854,  665 },
    {  9919, 10450,  873, 1716 },
    {  5568, 13215, 7446,  509 },
    {  7357, 15277, 3145, 3234 },
    {  3539, 12592,  631,  509 },
    {  4747, 14498,  825,  626 },
    {  4554, 16913,  969, 1443 },
    { 12771, 14693, 2323,  548 },
    { 18714,  8193, 2372,  586 },
    { 22292,  2743,  487, 1638 },
    { 20987, 17535, 1163, 1249 },
    { 19536, 18859,  632,  431 },
    { 19778, 15394, 1356,  626 },
    { 22969, 15394,  631, 2066 },
};
```

This `rects` variable holds an array of 2-dimensional rectangle data that represent the positions and sizes of rectangles shown earlier in this section. This will be used as bounding rectangles for the input values in the next part of the code:

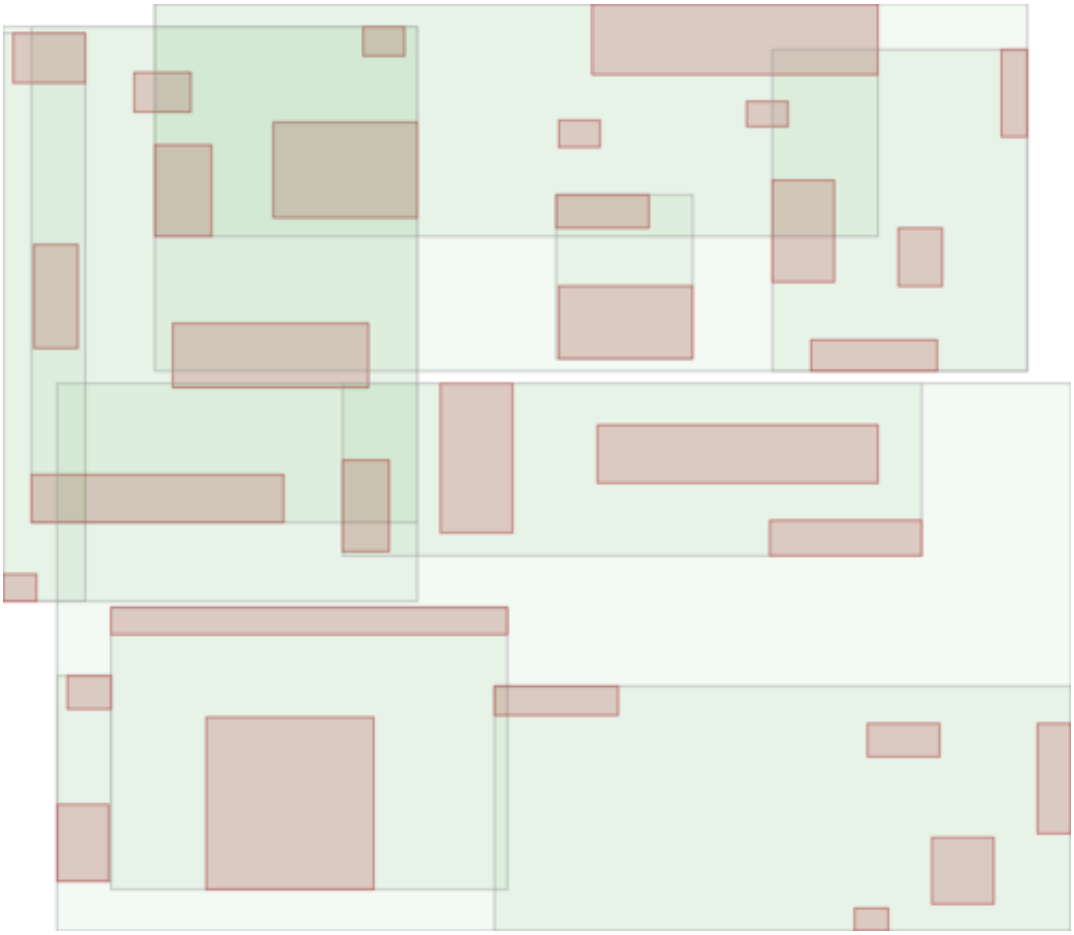
```
rt_type tree;

// Insert the rectangle objects into the tree.
int value = 0;
for (const auto& rect : rects)
    tree.insert({{rect.x, rect.y}, {rect.x + rect.w, rect.y + rect.h}}, value++);
```

Here, the tree is instantiated, and the rectangles are inserted with their associated values one at a time. Once the tree

is populated, the code that follows will export the structure of the tree as an SVG string, which will then be saved to a file on disk:

```
// Export the tree structure as a SVG for visualization.
std::string tree_svg = tree.export_tree(rt_type::export_tree_type::extent_as_svg);
std::ofstream fout("bounds.svg");
fout << tree_svg;
```

When you open the exported SVG file named **bounds.svg** in a SVG viewer, you'll see something similar to this:



which depicts not only the bounding rectangles of the inserted values (the red rectangles), but also the bounding rectangles of the directory nodes as well (the light green rectangles).

9.5 Bulk-loading data

In this section we will explore on how to bulk-load data into an *rtree* instance via rtree's own *bulk_loader* class. In this example, we'll be using the same custom trait we've used in the previous section in order to artificially promote the rate of node splits. The first part of the code:

```
#include <mdds/rtree.hpp>

#include <iostream>
```

(continues on next page)

(continued from previous page)

```
#include <fstream>

// Make the node capacity intentionally small.
struct tiny_trait_2d
{
    constexpr static size_t dimensions = 2;
    constexpr static size_t min_node_size = 2;
    constexpr static size_t max_node_size = 5;
    constexpr static size_t max_tree_depth = 100;

    constexpr static bool enable_forced_reinsertion = true;
    constexpr static size_t reinsertion_size = 2;
};

using rt_type = mdds::rtree<int, int, tiny_trait_2d>;
```

is pretty much identical to the example in the last section. The next part of the code defines what bounding rectangles to be inserted. Here, we are using a different set of rectangles than the previous example to illustrate the difference between a series of normal insertions and bulk-loading:

```
// 2D rectangle with the top-left position (x, y), width and height.
struct rect
{
    int x;
    int y;
    int w;
    int h;
};

std::vector<rect> rects =
{
    {  3538,  9126, 1908,  1908 },
    { 34272, 52053, 2416,  2543 },
    { 32113,  9761, 2416,   638 },
    { 16493, 16747, 7369,  2289 },
    { 29192, 23732, 3432,  2035 },
    { 35797, 17000, 1781,   892 },
    { 15857, 29319, 2162,  1654 },
    {  5825, 24239, 3559,  8512 },
    {  9127, 46846, 2543,  1019 },
    {  7094, 54338, 5210,   892 },
    { 18779, 39734, 3813, 10417 },
    { 32749, 35923, 2289,  2924 },
    { 26018, 31098,  257,  2797 },
    {  6713, 37066, 2924,  1146 },
    { 19541,  3157, 3305,  1146 },
    { 21953, 10904, 4448,   892 },
    { 15984, 24240, 5210,  1273 },
    {  8237, 15350, 2670,  2797 },
    { 17001, 13826, 4067,  1273 },
    { 30970, 13826, 3940,   765 },
    {  9634,  6587, 1654,  1781 },
```

(continues on next page)

(continued from previous page)

```
    { 38464, 47099,  511,  1400 },
    { 20556, 54085, 1400,  1527 },
    { 37575, 24113, 1019,   765 },
    { 20429, 21064, 1146,  1400 },
    { 31733,  4427, 2543,   638 },
    {  2142, 27161, 1273,  7369 },
    {  3920, 43289, 8131,  1146 },
    { 14714, 34272, 1400,  4956 },
    { 38464, 41258, 1273,  1273 },
    { 35542, 45703,  892,  1273 },
    { 25891, 50783, 1273,  5083 },
    { 35415, 28431, 2924,  1781 },
    { 15476,  7349, 1908,   765 },
    { 12555, 11159, 1654,  2035 },
    { 11158, 21445, 1908,  2416 },
    { 23350, 28049, 3432,   892 },
    { 28684, 15985, 2416,  4321 },
    { 24620, 21953, 1654,   638 },
    { 30208, 30716, 2670,  2162 },
    { 26907, 44179, 2797,  4067 },
    { 21191, 35416, 2162,  1019 },
    { 27668, 38717,  638,  3178 },
    {  3666, 50528, 2035,  1400 },
    { 15349, 48750, 2670,  1654 },
    { 28430,  7221, 2162,   892 },
    {  4808,  3158, 2416,  1273 },
    { 38464,  3666, 1527,  1781 },
    {  2777, 20937, 2289,  1146 },
    { 38209,  9254, 1908,  1781 },
    {  2269, 56497, 2289,   892 },
};
```

As with the previous example, each line contains the top-left position as well as the size of a rectangle. We are now going to insert these rectangles in two different ways.

First, we insert them via normal *insert()* method:

```
void load_tree()
{
    rt_type tree;

    // Insert the rectangle objects into the tree.
    int value = 0;
    for (const auto& rect : rects)
        tree.insert({{rect.x, rect.y}, {rect.x + rect.w, rect.y + rect.h}}, value++);

    // Export the tree structure as a SVG for visualization.
    std::string tree_svg = tree.export_tree(rt_type::export_tree_type::extent_as_svg);
    std::ofstream fout("bounds2.svg");
    fout << tree_svg;
}
```

This code should look familiar since it's nearly identical to the code in the previous section. After the insertion is done,

we export the tree as an SVG to visualize its structure.

Next, we insert the same set of rectangles via *bulk_loader*:

```
void bulkload_tree()
{
    rt_type::bulk_loader loader;

    // Insert the rectangle objects into the tree.
    int value = 0;
    for (const auto& rect : rects)
        loader.insert({{rect.x, rect.y}, {rect.x + rect.w, rect.y + rect.h}}, value++);

    // Start bulk-loading the tree.
    rt_type tree = loader.pack();

    // Export the tree structure as a SVG for visualization.
    std::string tree_svg = tree.export_tree(rt_type::export_tree_type::extent_as_svg);
    std::ofstream fout("bounds2-bulkload.svg");
    fout << tree_svg;
}
```

Inserting via *bulk_loader* shouldn't be too different than inserting via rtree's own insert methods. The only difference is that you instantiate a *bulk_loader* instance to insert all your data to it, then call its *pack()* method at the end to construct the final *rtree* instance.

When the insertion is done and the tree instance created, we are once again exporting its structure to an SVG file for visualization.

There are primarily two advantages to using *bulk_loader* to load data. First, unlike the normal insertion, bulk-loading does not trigger re-insertion nor node splits on the fly. Second, a tree created from bulk loader is typically well balanced than if you insert the same data through normal insertion. That is because the bulk loader sorts the data with respect to their bounding rectangles ahead of time and partition them evenly. The tree is then built from the bottom-up. You can visually see the effect of this when comparing the two trees built in our current example.

The first one is from the tree built via normal insertion:

The top part of the picture looks very "busy" indicated by a darker green area representative of more directory nodes overlaping with each other. In general, the rectangles look bigger and show higher degree of overlaps.

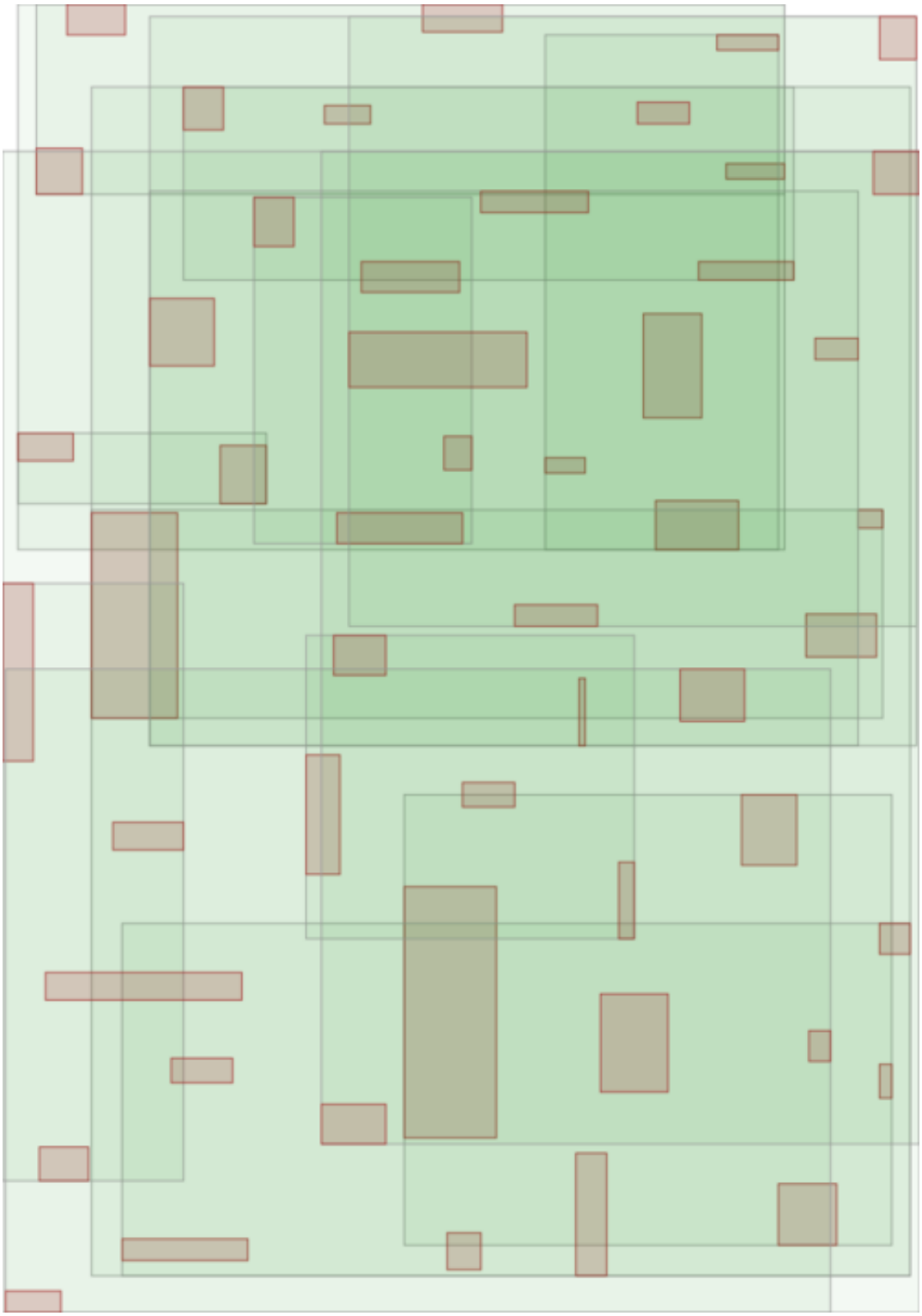
This one, on the other hand, is from the tree built with the same data set but through bulk-loading:

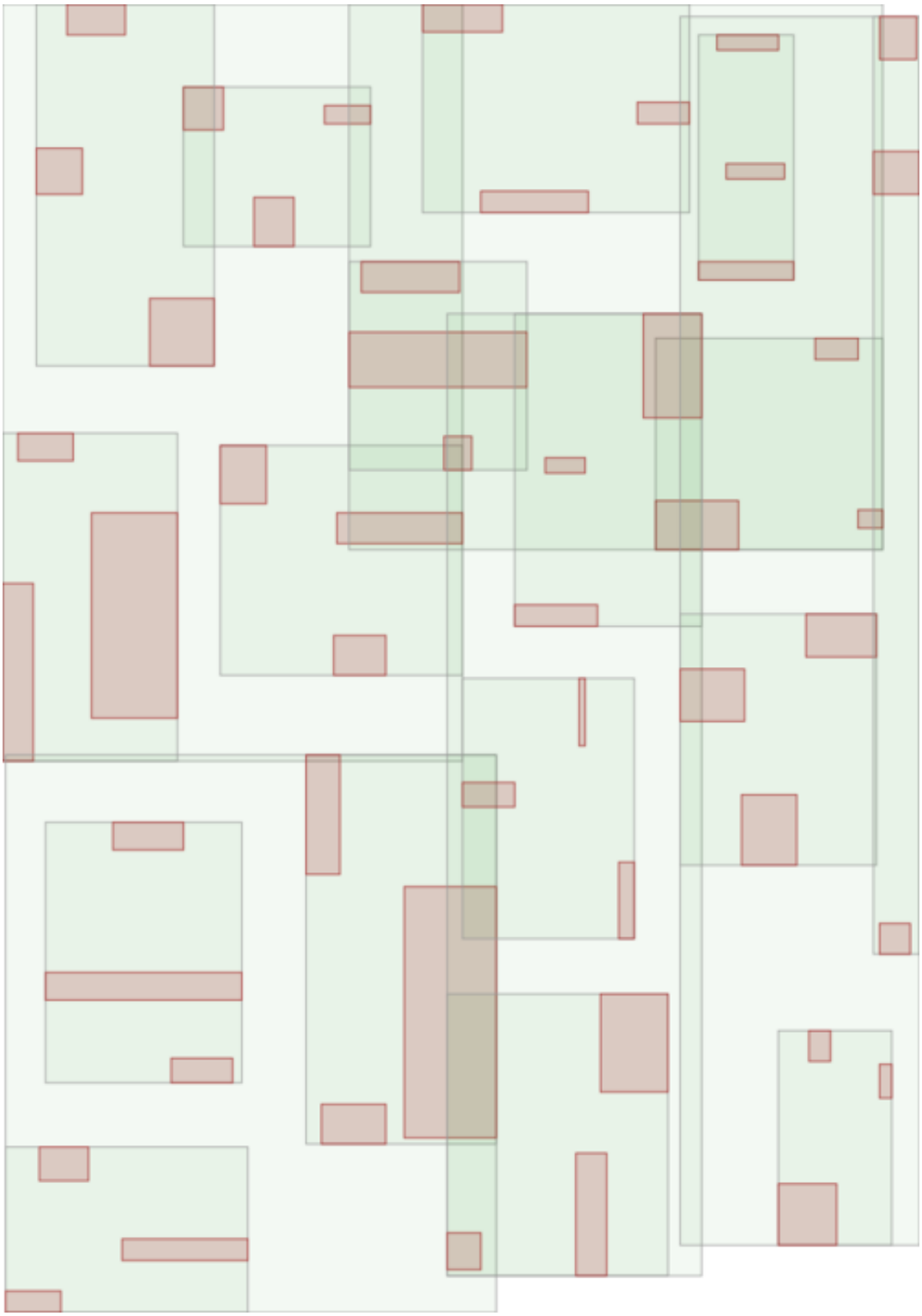
The rectangles generally look smaller and show much less overlaps than the previous picture, which is considered to be a more balanced R-tree structure.

9.6 API Reference

template<typename **KeyT**, typename **ValueT**, typename **Traits** = detail::rtree::*default_rtree_traits*>

class **rtree**





Public Types

using **key_type** = *KeyT*

using **value_type** = *ValueT*

using **node_type** = detail::rtree::node_type

using **export_tree_type** = detail::rtree::*export_tree_type*

using **search_type** = detail::rtree::*search_type*

using **integrity_check_properties** = detail::rtree::*integrity_check_properties*

Public Functions

rtree()

rtree(*rtree* &&other)

rtree(const *rtree* &other)

~rtree()

rtree &**operator=**(const *rtree* &other)

rtree &**operator=**(*rtree* &&other)

void **insert**(const *extent_type* &extent, *value_type* &&value)

Insert a new value associated with a bounding box. The new value object will be moved into the container.

Parameters

- **extent** – bounding box associated with the value.
- **value** – value being inserted.

void **insert**(const *extent_type* &extent, const *value_type* &value)

Insert a new value associated with a bounding box. A copy of the new value object will be placed into the container.

Parameters

- **extent** – bounding box associated with the value.
- **value** – value being inserted.

void **insert**(const *point_type* &position, *value_type* &&value)

Insert a new value associated with a point. The new value object will be moved into the container.

Parameters

- **position** – point associated with the value.
- **value** – value being inserted.

void **insert**(const *point_type* &position, const *value_type* &value)

Insert a new value associated with a point. A copy of the new value object will be placed into the container.

Parameters

- **position** – point associated with the value.
- **value** – value being inserted.

const_search_results **search**(const *point_type* &pt, *search_type* st) const

Search the tree and collect all value objects whose extents either contain the specified point, or exactly match the specified point.

Parameters

- **pt** – reference point to use for the search.
- **st** – search type that determines the satisfying condition of the search with respect to the reference point.

Returns

collection of all value objects that satisfy the specified search condition. This collection is immutable.

search_results **search**(const *point_type* &pt, *search_type* st)

Search the tree and collect all value objects whose extents either contain the specified point, or exactly match the specified point.

Parameters

- **pt** – reference point to use for the search.
- **st** – search type that determines the satisfying condition of the search with respect to the reference point.

Returns

collection of all value objects that satisfy the specified search condition. This collection is mutable.

const_search_results **search**(const *extent_type* &extent, *search_type* st) const

Search the tree and collect all value objects whose extents either overlaps with the specified extent, or exactly match the specified extent.

Parameters

- **extent** – reference extent to use for the search.
- **st** – search type that determines the satisfying condition of the search with respect to the reference extent.

Returns

collection of all value objects that satisfy the specified search condition. This collection is immutable.

search_results **search**(const *extent_type* &extent, *search_type* st)

Search the tree and collect all value objects whose extents either overlaps with the specified extent, or exactly match the specified extent.

Parameters

- **extent** – reference extent to use for the search.
- **st** – search type that determines the satisfying condition of the search with respect to the reference extent.

Returns
collection of all value objects that satisfy the specified search condition. This collection is mutable.

void **erase**(const *const_iterator* &pos)

Erase the value object referenced by the iterator passed to this method.

The iterator object will become invalid if the call results in an erasure of a value.

Parameters
pos – iterator that refernces the value object to erase.

void **erase**(const *iterator* &pos)

Erase the value object referenced by the iterator passed to this method.

The iterator object will become invalid if the call results in an erasure of a value.

Parameters
pos – iterator that refernces the value object to erase.

const *extent_type* &**extent**() const

Get the minimum bounding extent of the root node of the tree. The extent returned from this method is the minimum extent that contains the extents of all objects stored in the tree.

Returns
immutable reference to the extent of the root node of the tree.

bool **empty**() const

Check whether or not the tree stores any objects.

Returns
true if the tree is empty, otherwise false.

size_t **size**() const

Return the number of value nodes currently stored in the tree.

Returns
number of value nodes currently in the tree.

void **swap**(*rtree* &other)

Swap the content of the tree with another instance.

Parameters
other – another instance to swap the content with.

void **clear**()

Empty the entire container.

template<typename **FuncT**>
void **walk**(*FuncT* func) const

Walk down the entire tree depth first.

Parameters
func – function or function object that gets called at each node in the tree.

void **check_integrity**(const *integrity_check_properties* &props) const

Check the integrity of the entire tree structure.

Parameters
props – specify how the check is to be performed.

Throws
integrity_error – if the integrity check fails.

std::string **export_tree**(*export_tree_type* mode) const

Export the structure of a tree in textural format.

Parameters
mode – specify the format in which to represent the structure of a tree.

Returns
string representation of the tree structure.

class **bulk_loader**

Loader optimized for loading a large number of value objects. A resultant tree will have a higher chance of being well balanced than if the value objects were inserted individually into the tree.

Public Functions

bulk_loader()

void **insert**(const *extent_type* &extent, *value_type* &&value)

void **insert**(const *extent_type* &extent, const *value_type* &value)

void **insert**(const *point_type* &position, *value_type* &&value)

void **insert**(const *point_type* &position, const *value_type* &value)

rtree **pack**()

class **const_iterator** : public mdds::*rtree*<*KeyT*, *ValueT*, *Traits*>::iterator_base<*const_iterator*, *const_search_results*::store_type::const_iterator, const *rtree*::*value_type*>

Public Types

using **value_type** = _ValueT

Public Functions

inline *value_type* &**operator***() const

inline *value_type* ***operator->**() const

class **const_search_results** : public mdds::*rtree*<*KeyT*, *ValueT*, *Traits*>::search_results_base<const node_store>

Public Functions

const_iterator **cbegin**() const

const_iterator **cend**() const

const_iterator **begin**() const

const_iterator **end**() const

struct **extent_type**

Public Functions

extent_type()

extent_type(const *point_type* &_start, const *point_type* &_end)

std::string **to_string**() const

bool **is_point**() const

bool **operator==**(const *extent_type* &other) const

bool **operator!=**(const *extent_type* &other) const

bool **contains**(const *point_type* &pt) const

Determine whether or not the specified point lies within this extent.

Parameters

pt – point to query with.

Returns

true if the point lies within this extent, or false otherwise.

bool **contains**(const *extent_type* &bb) const

Determine whether or not the specified extent lies *entirely* within this extent.

Parameters

bb – extent to query with.

Returns

true if the specified extent lies entirely within this extent, or otherwise false.

bool **intersects**(const *extent_type* &bb) const

Determine whether or not the specified extent overlaps with this extent either partially or fully.

Parameters

bb – extent to query with.

Returns

true if the specified extent overlaps with this extent, or otherwise false.

bool **contains_at_boundary**(const *extent_type* &other) const

Determine whether or not another bounding box is within this bounding box and shares a part of its boundaries.

Public Members

point_type **start**

point_type **end**

class **iterator** : public mdds::*rtree*<*KeyT*, *ValueT*, *Traits*>::iterator_base<*iterator*, *search_results*::store_type::iterator, *rtree*::*value_type*>

Public Types

using **value_type** = _ValueT

Public Functions

inline *value_type* &**operator***()

inline *value_type* ***operator->**()

template<typename **_SelfIter**, typename **_StoreIter**, typename **_ValueT**>

class **iterator_base**

Public Types

using **store_iterator_type** = *_StoreIter*

using **self_iterator_type** = *_SelfIter*

using **value_type** = *_ValueT*

using **pointer** = *value_type**

using **reference** = *value_type*&

using **difference_type** = std::ptrdiff_t

using **iterator_category** = std::bidirectional_iterator_tag

Public Functions

bool **operator==**(const *self_iterator_type* &other) const

bool **operator!=**(const *self_iterator_type* &other) const

self_iterator_type &**operator++**()

self_iterator_type **operator++**(int)

self_iterator_type &**operator--**()

self_iterator_type **operator--**(int)

const *extent_type* &**extent**() const

size_t **depth**() const

struct **node_properties**

Public Members

node_type **type**

extent_type **extent**

struct **point_type**

Public Functions

point_type()

point_type(std::initializer_list<*key_type*> vs)

std::string **to_string**() const

bool **operator==**(const *point_type* &other) const

bool **operator!=**(const *point_type* &other) const

Public Members

key_type **d**[traits_type::dimensions]

class **search_results** : public mdds::*rtree*<*KeyT*, *ValueT*, *Traits*>::search_results_base<node_store>

Public Functions

iterator **begin**()

iterator **end**()

template<typename **NS**>

class **search_results_base**

struct **default_rtree_traits**

Public Static Attributes

static constexpr size_t **dimensions** = 2

Number of dimensions in bounding rectangles.

static constexpr size_t **min_node_size** = 40

Minimum number of child nodes that must be present in each directory node. Exception is the root node, which is allowed to have less than the minimum number of nodes, but only when it's a leaf directory node.

static constexpr size_t **max_node_size** = 100

Maximum number of child nodes that each directory node is allowed to have. There are no exceptions to this rule.

static constexpr size_t **max_tree_depth** = 100

Maximum depth a tree is allowed to have.

static constexpr bool **enable_forced_reinsertion** = true

A flag to determine whether or not to perform forced reinsertion when a directory node overflows, before attempting to split the node.

static constexpr size_t **reinsertion_size** = 30

Number of nodes to get re-inserted during forced reinsertion. This should be roughly 30% of max_node_size + 1, and should not be greater than max_node_size - min_node_size + 1.

struct **integrity_check_properties**

Public Members

bool **throw_on_first_error** = true

When true, the integrity check will throw an exception on the first validation failure. When false, it will run through the entire tree and report all encountered validation failures then throw an exception if there is at least one failure.

bool **error_on_min_node_size** = true

When true, a node containing children less than the minimum node size will be treated as an error.

enum class mdds::detail::rtree::**export_tree_type**

Values:

enumerator **formatted_node_properties**

Textural representation of a tree structure. Indent levels represent depths, and each line represents a single node.

enumerator **extent_as_obj**

The extents of all directory and value nodes are exported as Wavefront .obj format. Only 2 dimensional trees are supported for now.

For a 2-dimensional tree, each depth is represented by a 2D plane filled with rectangles representing the extents of either value or directory nodes at that depth level. The depth planes are then stacked vertically.

enumerator **extent_as_svg**

The extents of all directory and value nodes are exported as a scalable vector graphics (SVG) format. Only 2 dimensional trees are supported.

enum class mdds::detail::rtree::**search_type**

Values:

enumerator **overlap**

Pick up all objects whose extents overlap with the specified search extent.

enumerator **match**

Pick up all objects whose extents exactly match the specified search extent.

CHAPTER

TEN

API INCOMPATIBILITY NOTES

10.1 v2.1

- The following public template types have been put into `mdds::detail` namespace:
  - `has_value_type`
  - `const_or_not`
  - `const_t`
  - `get_iterator_type`
  - `invalid_static_int`

10.1.1 multi_type_vector

- In 2.0, multi_type_vector took two template parameters: one for the element block functions and one for other traits. In 2.1, multi_type_vector only takes one template parameter for the traits, and the traits now must include the element block functions.
- The following block function helpers have been removed:
  - `mdds::mtv::custom_block_func1`
  - `mdds::mtv::custom_block_func2`
  - `mdds::mtv::custom_block_func3`

  They have been replaced with `mdds::mtv::element_block_funcs` which uses template parameter pack to allow unspecified number of standard and custom blocks.

  As a result of this change, the following headers have been removed:
  - `include/mdds/multi_type_vector/custom_func1.hpp`
  - `include/mdds/multi_type_vector/custom_func2.hpp`
  - `include/mdds/multi_type_vector/custom_func3.hpp`
  - `include/mdds/multi_type_vector/trait.hpp`
  - `include/mdds/multi_type_vector_custom_func2.hpp`
  - `include/mdds/multi_type_vector_custom_func3.hpp`
  - `include/mdds/multi_type_vector_trait.hpp`
- `mdds::mtv::default_trait` has been renamed to `mdds::mtv::default_traits`.

- The following element block types have an additional template parameter to specify the underlying storage type. This can be used to specify, for instance, whether the element block uses `std::vector`, `std::deque` or another custom container type with API compatible with the aforementioned two.
  - `mdds::mtv::default_element_block`
  - `mdds::mtv::managed_element_block`
  - `mdds::mtv::noncopyable_managed_element_block`

  With this change, the compiler macro named `MDDS_MULTI_TYPE_VECTOR_USE_DEQUE`, which was previously used to switch from `std::vector` to `std::deque` as the underlying storage type for all element blocks, has been removed.

10.1.2 multi_type_matrix

- `mdds::mtm::std_string_trait` has been renamed to `mdds::mtm::std_string_traits`.

10.1.3 trie_map / packed_trie_map

- The following public types have been renamed:
  - `mdds::trie::std_container_trait` -> `mdds::trie::std_container_traits`
  - `mdds::trie::std_string_trait` -> `mdds::trie::std_string_traits`

10.2 v2.0

- baseline C++ version has been set to C++17.
- deprecated `rectangle_set` data structure has been removed.

10.2.1 multi_type_vector

- The second template parameter is now a trait type that specifies custom event function type and loop-unrolling factor. Prior to 2.0 the second template parameter was custom event function type.
- Due to the addition of the structure-of-arrays variant, the following header files have been relocated:

Table 1: Relocated Headers

Old header location	New header location
mdds/multi_type_vector_types.hpp	mdds/multi_type_vector/types.hpp
mdds/multi_type_vector_macro.hpp	mdds/multi_type_vector/macro.hpp
mdds/multi_type_vector_trait.hpp	mdds/multi_type_vector/trait.hpp
mdds/multi_type_vector_itr.hpp	mdds/multi_type_vector/aos/iterator.hpp
mdds/multi_type_vector_custom_func1.hpp	mdds/multi_type_vector/custom_func1.hpp
mdds/multi_type_vector_custom_func2.hpp	mdds/multi_type_vector/custom_func2.hpp
mdds/multi_type_vector_custom_func3.hpp	mdds/multi_type_vector/custom_func3.hpp

The old headers will continue to work for the time being, but consider them deprecated.

- Since now we have array-of-structures (AoS) and structure-of-arrays (SoA) variants of multi_type_vector, there are two instances of multi_type_vector class in two different headers and namespace locations. To use the AoS variant, include the header

```
#include <mdds/multi_type_vector/aos/main.hpp>
```

and instantiate the template class as `mdds::mtv::aos::multi_type_vector`. Likewise, to use the SoA variant, include the header

```
#include <mdds/multi_type_vector/soa/main.hpp>
```

and instantiate the template class as `mdds::mtv::soa::multi_type_vector`.

If you include the original header

```
#include <mdds/multi_type_vector.hpp>
```

it will include a template alias `mdds::multi_type_vector` that simply references `mdds::mtv::soa::multi_type_vector`.

10.2.2 segment_tree

- The following public types have been renamed:
  - `search_result` -> `search_results`
  - `search_result_type` -> `search_results_type`

10.3 v1.5

10.3.1 multi_type_vector

- The standard integer blocks previously used non-standard integer types, namely:
  - short
  - unsigned short
  - int
  - unsigned int
  - long
  - unsigned long
  - char
  - unsigned char

  Starting with this version, the integer blocks now use:
  - (u)int8_t
  - (u)int16_t
  - (u)int32_t
  - (u)int64_t

- The numeric_element_block type has been renamed to *double_element_block*, to make room for a new element block for float type named *float_element_block*.

10.4 v1.4

10.4.1 multi_type_matrix

- The walk() methods previously took the function object by reference, but the newer versions now take the function object by value. With this change, it is now possible to pass inline lambda function. However, if you were dependent on the old behavior, *this change may adversely affect the outcome of your code especially when your function object stores data members that are expected to be altered by the walk() methods*.

10.5 v1.2

10.5.1 trie_map / packed_trie_map

- The find() method now returns a const_iterator instance rather than a value type. It returns an end position iterator when the method fails to find a match.
- The prefix_search() method now returns a search_results instance that has begin() and end() methods to allow iterating through the result set.
- The constructor no longer takes a null value parameter.
- Some nested type names have been renamed:
  - string_type -> key_type
  - char_type -> key_unit_type
  - string_buffer_type -> key_buffer_type
- Some functions expected from the key trait class have been renamed:
  - init_buffer() -> to_key_buffer()
  - to_string() -> to_key()
- The kay trait class now expects the following additional static methods:
  - key_buffer_type to_key_buffer(const key_type& key)
  - key_unit_type* buffer_data(const key_buffer_type& buf)
  - size_t buffer_size(const key_buffer_type& buf)

10.5.2 quad_point_tree

- The search_result nested class has been renamed to search_results, to keep the name consistent with that of the same name in trie_map and packed_trie_map.

10.5.3 multi_type_matrix

- The matrix trait structure (formerly known as the string trait structure) now needs to specify the type of block that stores integer values as its **integer_element_block** member.

10.6 v1.0

- Starting with version 1.0, mdds now requires support for C++11. Stick with 0.12 or earlier versions if you use a compiler that doesn't support C++11.
- data_type has been renamed to value_type for segment_tree, rectangle_set, and point_quad_tree.

10.7 v0.9

10.7.1 multi_type_vector

- The number of template parameters in custom_block_func1, custom_block_func2 and custom_block_func3 have been reduced by half, by deducing the numerical block type ID from the block type definition directly. If you use the older variant, simply remove the template arguments that are numerical block IDs.

10.8 v0.8

10.8.1 flat_segment_tree

- The search_tree() method in 0.8.0 returns std::pair<const_iterator, bool> instead of just returning bool as of 0.7.1. If you use this method and relies on the return value of the old version, use the second parameter of the new return value which is equivalent of the previous return value.

10.9 v0.5

10.9.1 flat_segment_tree

- The search() method now returns ::std::pair<const_iterator, bool>. This method previously returned only bool. Use the second parameter of the new return value which is equivalent of the previous return value.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- search

INDEX

M