



Useful Boost Containers

C++ User Group Aachen, 2023-11-29
Daniel Evers

Boost Container Libs

CONTAINERS

- **Array**: STL compliant container wrapper for arrays of constant size.
- **Bimap**: Bidirectional maps library for C++. With Boost.Bimap you can create associative containers in which both types can be used as key.
- **Circular Buffer**: A STL compliant container also known as ring or cyclic buffer.
- **Container**: Standard library containers and extensions.
- **Dynamic Bitset**: The `dynamic_bitset` class represents a set of bits. It provides accesses to the value of individual bits via an operator[] and provides integer-like operations, such as operator& and operator<<. The number of bits in the set is specified at runtime via a parameter to the constructor of the dynamic_bitset.
- **GIL**: (C++14) Generic Image Library
- **Graph**: The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).
- **GraphParallel**: The PBGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).
- **ICL**: Interval Container Library, interval sets and maps and aggregation of associated values
- **Intrusive**: Intrusive containers and algorithms.
- **JSON**: JSON parsing, serialization, and DOM in C++11
- **Multi-Array**: Boost.MultiArray provides a generic N-dimensional array concept definition and common implementations of that interface.
- **Multi-Index**: The Boost Multi-index Containers Library provides a class template named `multi_index_container` which enables the construction of containers with sorting and access semantics.
- **Pointer Container**: Containers for storing heap-allocated polymorphic objects to ease OO-programming.
- **PolyCollection**: Fast containers of polymorphic objects.
- **Property Map**: Concepts defining interfaces which map key objects to value objects.
- **Property Map (Parallel)**: Parallel extensions to Property Map for use with Parallel Graph.
- **Property Tree**: A tree data structure especially suited to storing configuration data.
- **String Ref**: String view templates.
- **Unordered**: Unordered associative containers.
- **URL**: URL parsing in C++11
- **Variant**: Safe, generic, stack-based discriminated union container.
- **Variant2**: A never-valueless, strong guarantee implementation of `std::variant`.

Boost Container Libs

CONTAINERS

- ~~Array: STL compliant container wrapper for arrays of constant size.~~
- **Bimap**: Bidirectional maps library for C++. With Boost.Bimap you can create associative containers in which both types can be used as key.
- **Circular Buffer**: A STL compliant container also known as ring or cyclic buffer.
- **Container**: Standard library containers and extensions.
- ~~Dynamic Bitset: The dynamic_bitset class represents a set of bits. It provides accesses to the value of individual bits via an operator[] and provides integer, such as operator& and operator<<. The number of bits in the set is specified at runtime via a parameter to the constructor of the dynamic_bitset.~~
- **GIL**: (C++14) Generic Image Library
- **Graph**: The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).
- **GraphParallel**: The PBGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).
- **ICL**: Interval Container Library, interval sets and maps and aggregation of associated values
- **Intrusive**: Intrusive containers and algorithms.
- **JSON**: JSON parsing, serialization, and DOM in C++11
- **Multi-Array**: Boost.MultiArray provides a generic N-dimensional array concept definition and common implementations of that interface.
- **Multi-Index**: The Boost Multi-index Containers Library provides a class template named multi_index_container which enables the construction of containers with sorting and access semantics.
- **Pointer Container**: Containers for storing heap-allocated polymorphic objects to ease OO-programming.
- **PolyCollection**: Fast containers of polymorphic objects.
- **Property Map**: Concepts defining interfaces which map key objects to value objects.
- **Property Map (Parallel)**: Parallel extensions to Property Map for use with Parallel Graph.
- **Property Tree**: A tree data structure especially suited to storing configuration data.
- ~~String Ref: String view templates.~~
- **Unordered**: Unordered associative containers.
- **URL**: URL parsing in C++11
- ~~Variant: Safe, generic, stack-based discriminated union container.~~
- ~~Variant2: A never-valueless, strong guarantee implementation of std::variant.~~

Boost Container Libs

CONTAINERS

- ~~Array: STL compliant container wrapper for arrays of constant size.~~
- **Bimap**: Bidirectional maps library for C++. With Boost.Bimap you can create associative containers in which both types can be used as key.
- **Circular Buffer**: A STL compliant container also known as ring or cyclic buffer.
- **Container**: Standard library containers and extensions.
- ~~Dynamic Bitset: The dynamic_bitset class represents a set of bits. It provides accesses to the value of individual bits via an operator[] and provides integer, such as operator& and operator<<. The number of bits in the set is specified at runtime via a parameter to the constructor of the dynamic_bitset class.~~
- ~~GIL: (C++14) Generic Image Library~~
- ~~Graph: The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).~~
- ~~GraphParallel: The PBGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).~~
- **ICL**: Interval Container Library, interval sets and maps and aggregation of associated values
- **Intrusive**: Intrusive containers and algorithms.
- **JSON**: JSON parsing, serialization, and DOM in C++11
- **Multi-Array**: Boost.MultiArray provides a generic N-dimensional array concept definition and common implementations of that interface.
- **Multi-Index**: The Boost Multi-index Containers Library provides a class template named multi_index_container which enables the construction of containers with sorting and access semantics.
- **Pointer Container**: Containers for storing heap-allocated polymorphic objects to ease OO-programming.
- **PolyCollection**: Fast containers of polymorphic objects.
- **Property Map**: Concepts defining interfaces which map key objects to value objects.
- **Property Map (Parallel)**: Parallel extensions to Property Map for use with Parallel Graph.
- **Property Tree**: A tree data structure especially suited to storing configuration data.
- ~~String Ref: String view templates.~~
- **Unordered**: Unordered associative containers.
- **URL**: URL parsing in C++11
- ~~Variant: Safe, generic, stack-based discriminated union container.~~
- ~~Variant2: A never-valueless, strong guarantee implementation of std::variant.~~

Boost Container Libs

CONTAINERS

- ~~Array: STL compliant container wrapper for arrays of constant size.~~
- **Bimap**: Bidirectional maps library for C++. With Boost.Bimap you can create associative containers in which both types can be used as key.
- **Circular Buffer**: A STL compliant container also known as ring or cyclic buffer.
- **Container**: Standard library containers and extensions.
- ~~Dynamic Bitset: The dynamic_bitset class represents a set of bits. It provides accesses to the value of individual bits via an operator[] and provides integer, such as operator& and operator<<. The number of bits in the set is specified at runtime via a parameter to the constructor of the dynamic_bitset class.~~
- ~~GIL: (C++14) Generic Image Library~~
- ~~Graph: The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).~~
- ~~GraphParallel: The PBGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).~~
- **ICL**: Interval Container Library, interval sets and maps and aggregation of associated values
- **Intrusive**: Intrusive containers and algorithms.
- ~~JSON: JSON parsing, serialization, and DOM in C++11~~
- **Multi-Array**: Boost.MultiArray provides a generic N-dimensional array concept definition and common implementations of that interface.
- **Multi-Index**: The Boost Multi-index Containers Library provides a class template named multi_index_container which enables the construction of containers with sorting and access semantics.
- **Pointer Container**: Containers for storing heap-allocated polymorphic objects to ease OO-programming.
- **PolyCollection**: Fast containers of polymorphic objects.
- **Property Map**: Concepts defining interfaces which map key objects to value objects.
- **Property Map (Parallel)**: Parallel extensions to Property Map for use with Parallel Graph.
- **Property Tree**: A tree data structure especially suited to storing configuration data.
- ~~String Ref: String view templates.~~
- **Unordered**: Unordered associative containers.
- ~~URL: URL parsing in C++11~~
- ~~Variant: Safe, generic, stack-based discriminated union container.~~
- ~~Variant2: A never-valueless, strong guarantee implementation of std::variant.~~

Boost Container Libs

CONTAINERS

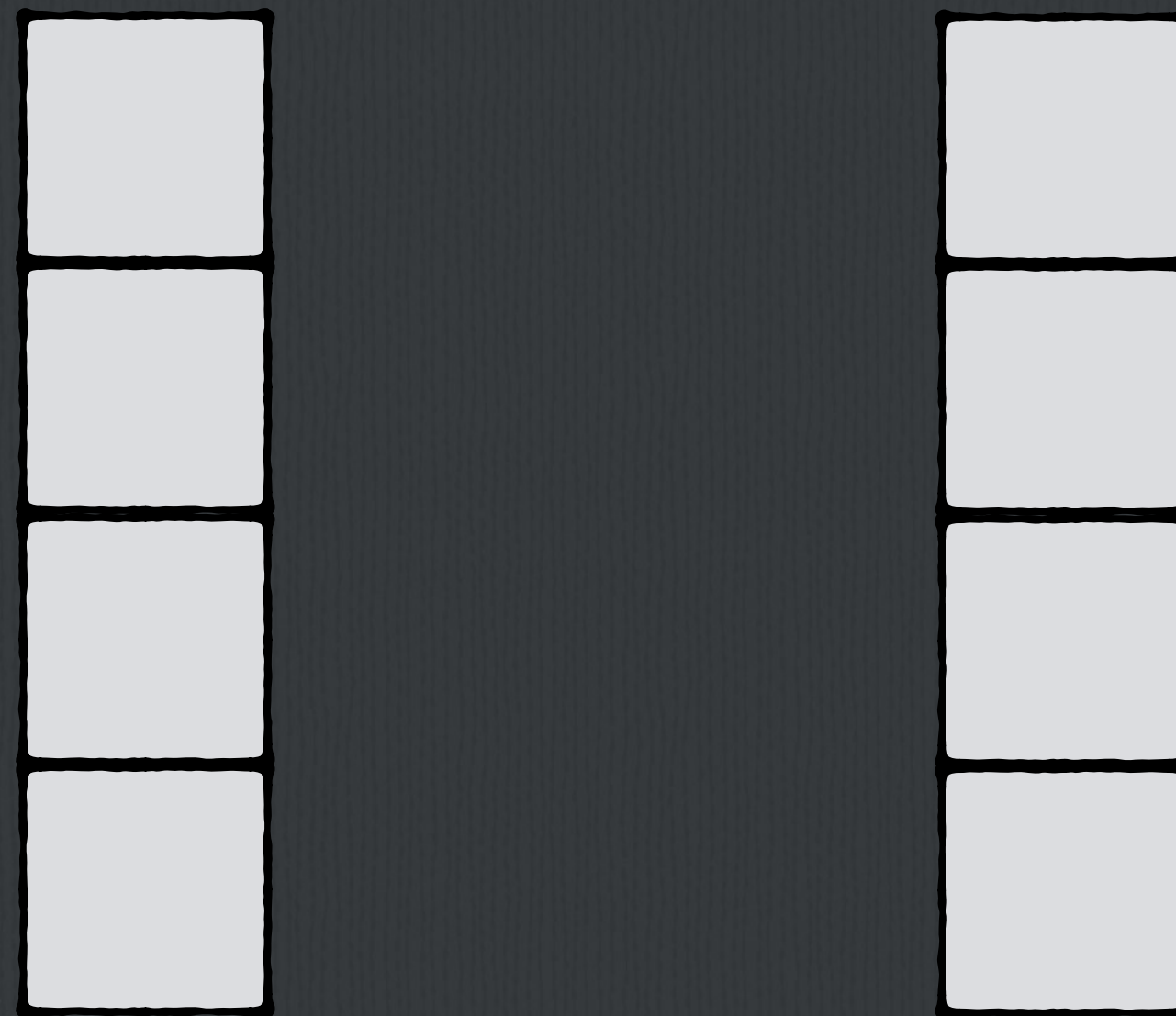
- ~~Array: STL compliant container wrapper for arrays of constant size.~~
- **Bimap**: Bidirectional maps library for C++. With Boost.Bimap you can create associative containers in which both types can be used as key.
- **Circular Buffer**: A STL compliant container also known as ring or cyclic buffer.
- **Container**: Standard library containers and extensions.
- ~~Dynamic Bitset: The dynamic_bitset class represents a set of bits. It provides accesses to the value of individual bits via an operator[] and provides integer, such as operator& and operator<<. The number of bits in the set is specified at runtime via a parameter to the constructor of the dynamic_bitset class.~~
- ~~GIL: (C++14) Generic Image Library~~
- ~~Graph: The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).~~
- ~~GraphParallel: The PBGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).~~
- **ICL**: Interval Container Library, interval sets and maps and aggregation of associated values
- **Intrusive**: Intrusive containers and algorithms.
- ~~JSON: JSON parsing, serialization, and DOM in C++11~~
- **Multi-Array**: Boost.MultiArray provides a generic N-dimensional array concept definition and common implementations of that interface.
- **Multi-Index**: The Boost Multi-index Containers Library provides a class template named multi_index_container which enables the construction of containers with sorting and access semantics.
- **Pointer Container**: Containers for storing heap-allocated polymorphic objects to ease OO-programming.
- **PolyCollection**: Fast containers of polymorphic objects.
- ~~Property Map: Concepts defining interfaces which map key objects to value objects.~~
- ~~Property Map (Parallel): Parallel extensions to Property Map for use with Parallel Graph.~~
- ~~Property Tree: A tree data structure especially suited to storing configuration data.~~
- ~~String Ref: String view templates.~~
- **Unordered**: Unordered associative containers.
- ~~URL: URL parsing in C++11~~
- ~~Variant: Safe, generic, stack-based discriminated union container.~~
- ~~Variant2: A never-valueless, strong guarantee implementation of std::variant.~~

Boost.Bimap

- associative containers in which both types can be used as key
- i.e. $\text{bimap}\langle X, Y \rangle \sim \text{map}\langle X, Y \rangle + \text{map}\langle Y, X \rangle$

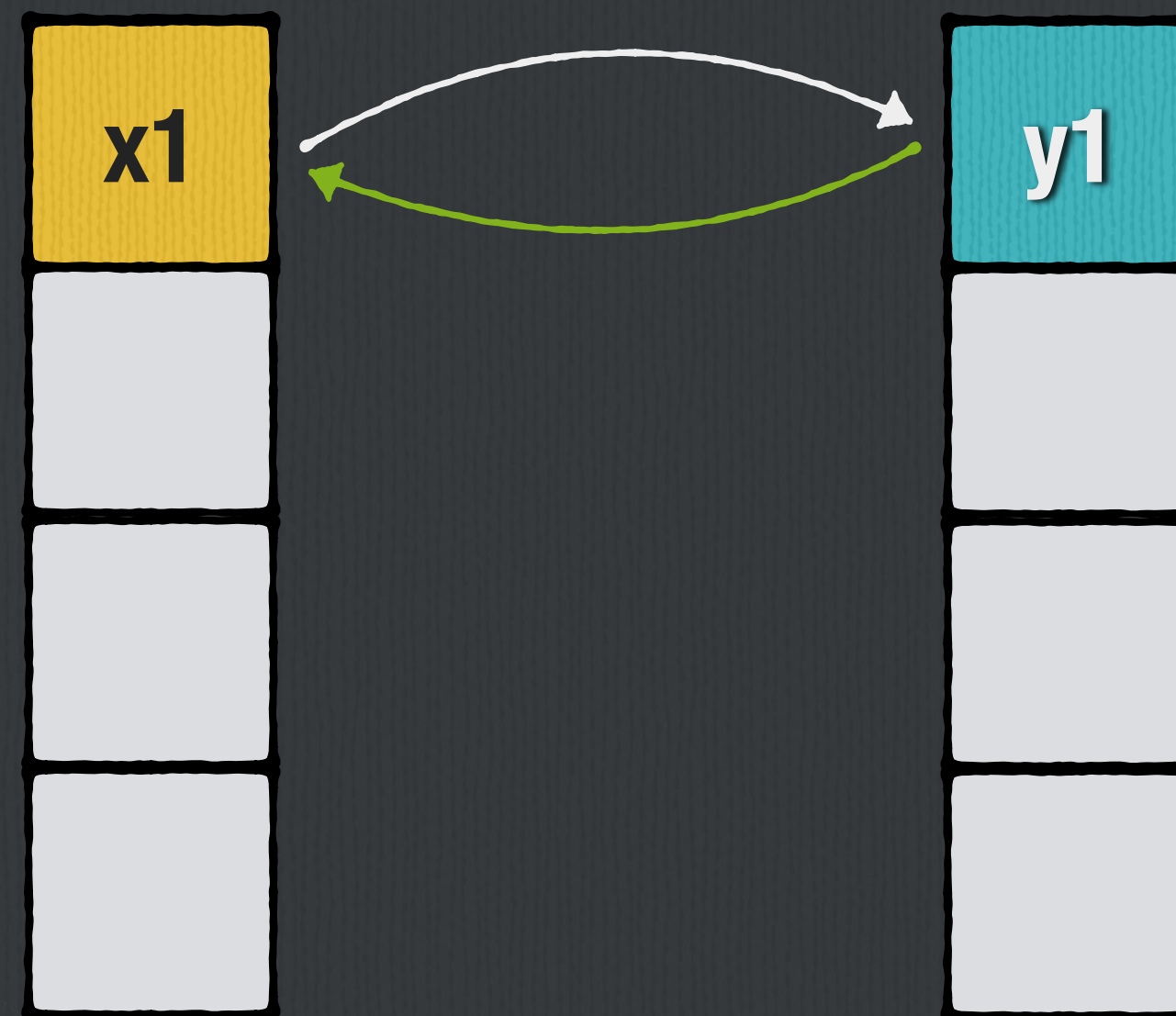
Boost.Bimap

- associative containers in which both types can be used as key
- i.e. $\text{bimap}\langle X, Y \rangle \sim \text{map}\langle X, Y \rangle + \text{map}\langle Y, X \rangle$



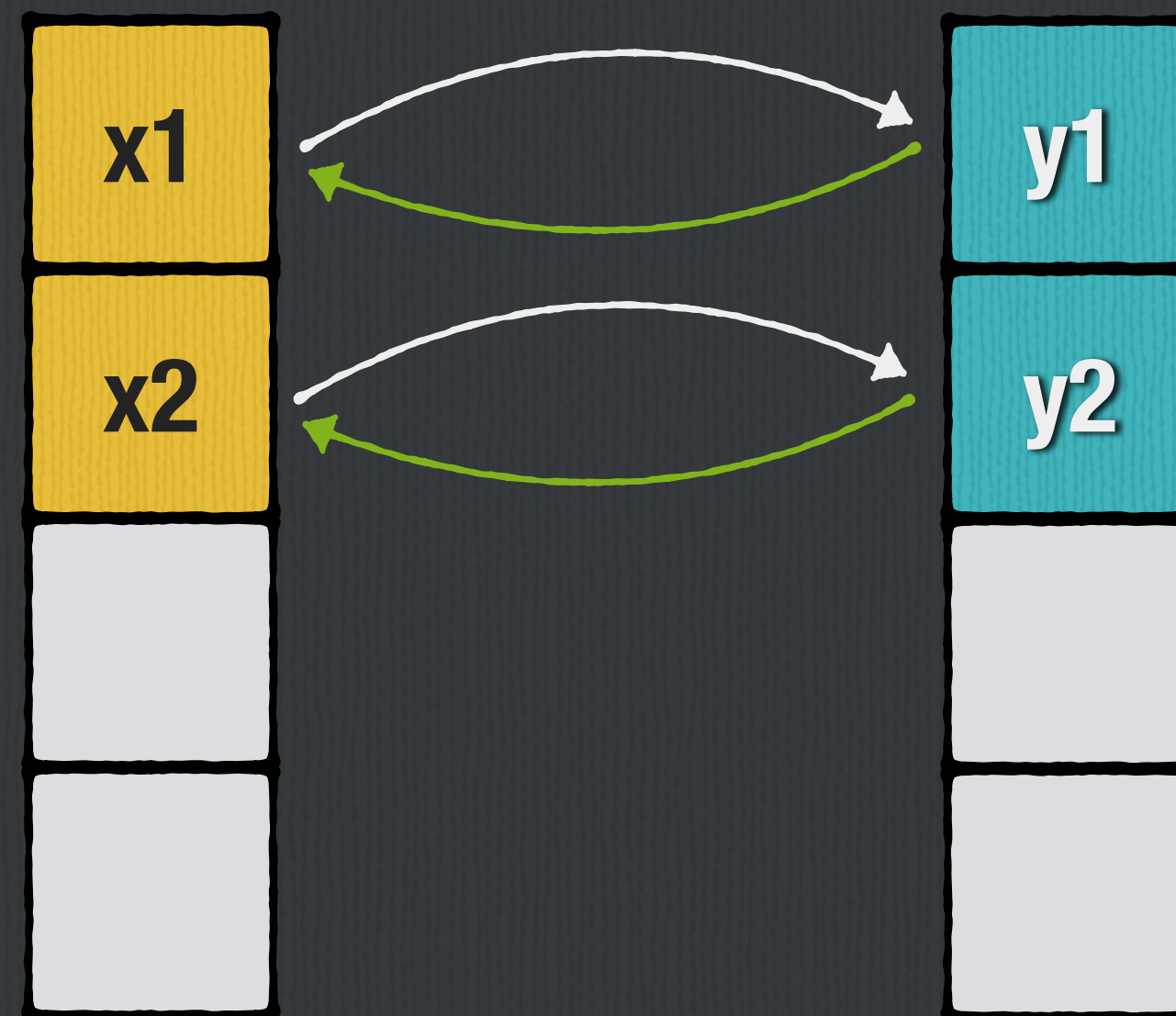
Boost.Bimap

- associative containers in which both types can be used as key
- i.e. $\text{bimap}\langle X, Y \rangle \sim \text{map}\langle X, Y \rangle + \text{map}\langle Y, X \rangle$



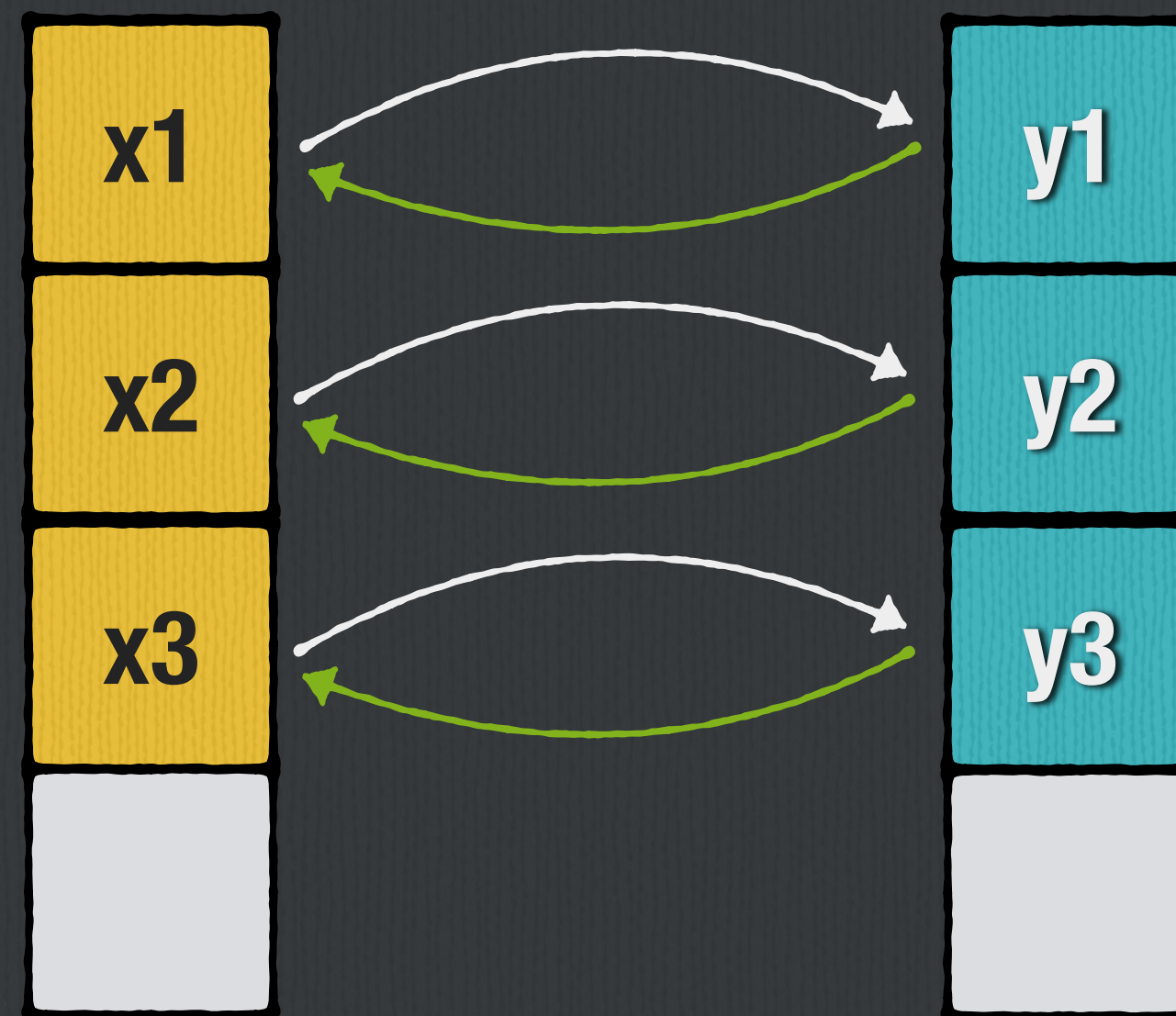
Boost.Bimap

- associative containers in which both types can be used as key
- i.e. $\text{bimap}\langle X, Y \rangle \sim \text{map}\langle X, Y \rangle + \text{map}\langle Y, X \rangle$



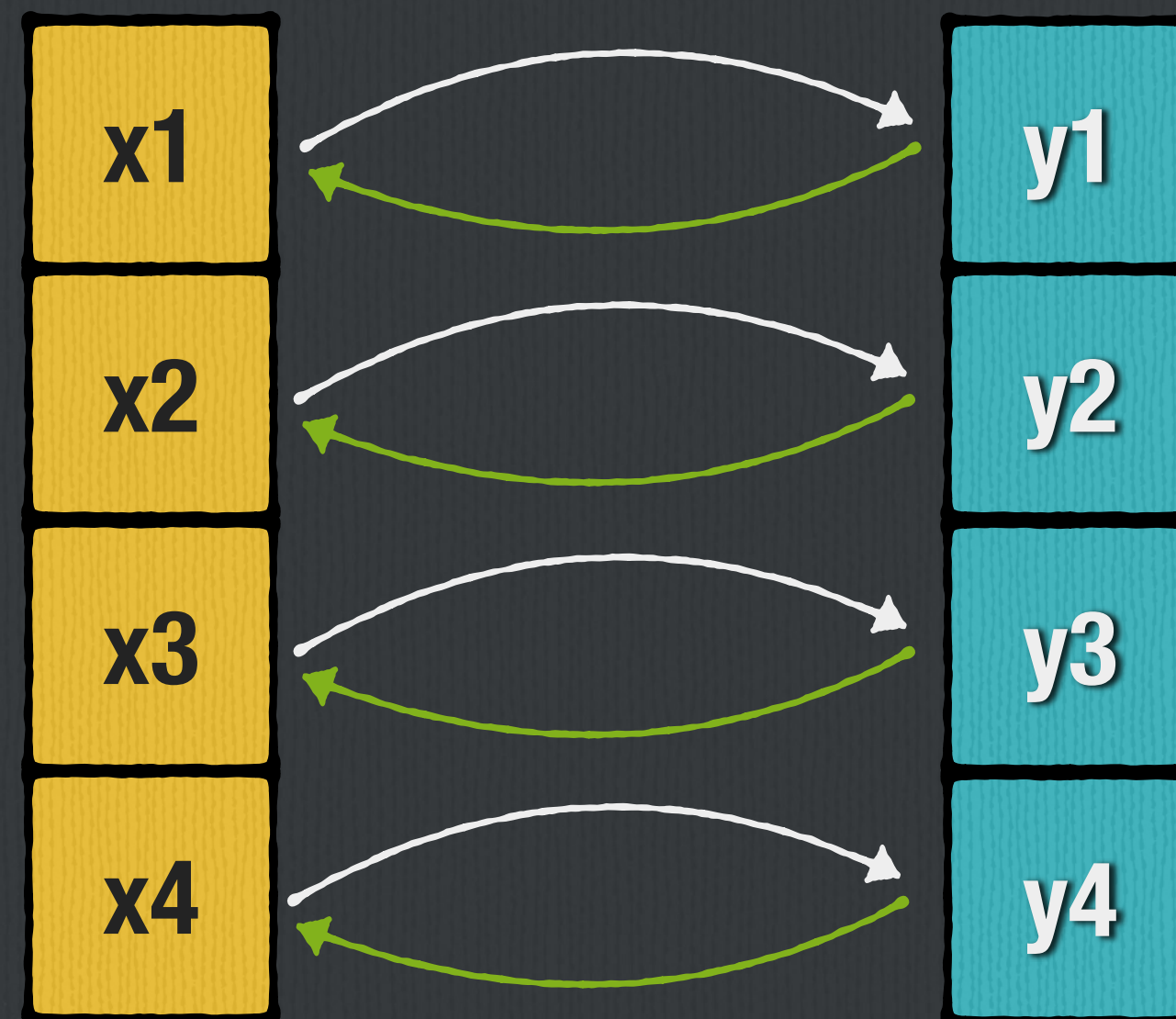
Boost.Bimap

- associative containers in which both types can be used as key
- i.e. $\text{bimap}\langle X, Y \rangle \sim \text{map}\langle X, Y \rangle + \text{map}\langle Y, X \rangle$



Boost.Bimap

- associative containers in which both types can be used as key
- i.e. $\text{bimap}\langle X, Y \rangle \sim \text{map}\langle X, Y \rangle + \text{map}\langle Y, X \rangle$

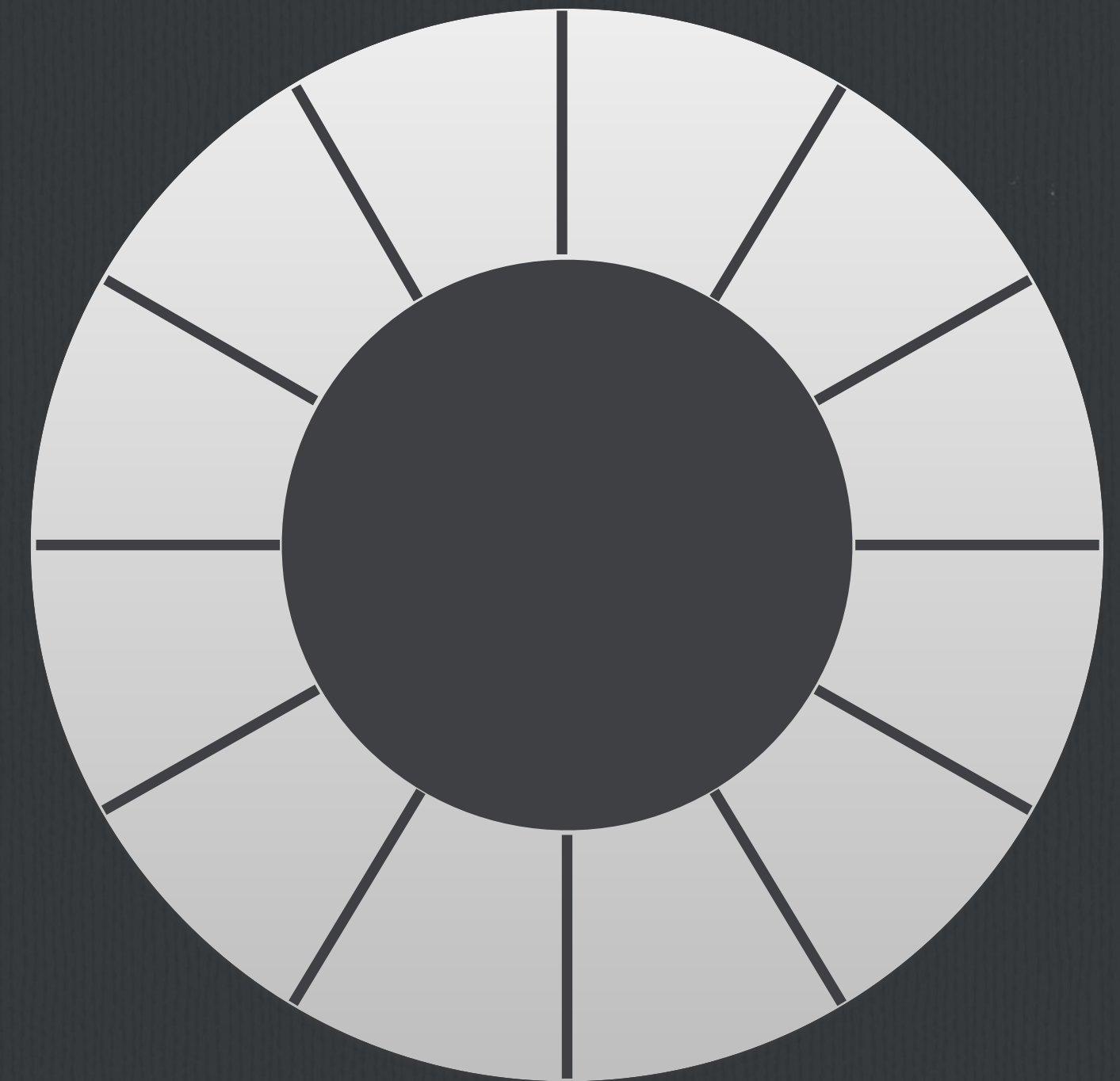


Boost.CircularBuffer

- ☐ **Ring buffer implementation, fixed capacity**
- ☐ **STL-compatible API**
- ☐ **Possible use cases:**
 - ☐ **Cache, LIFO, FIFO**

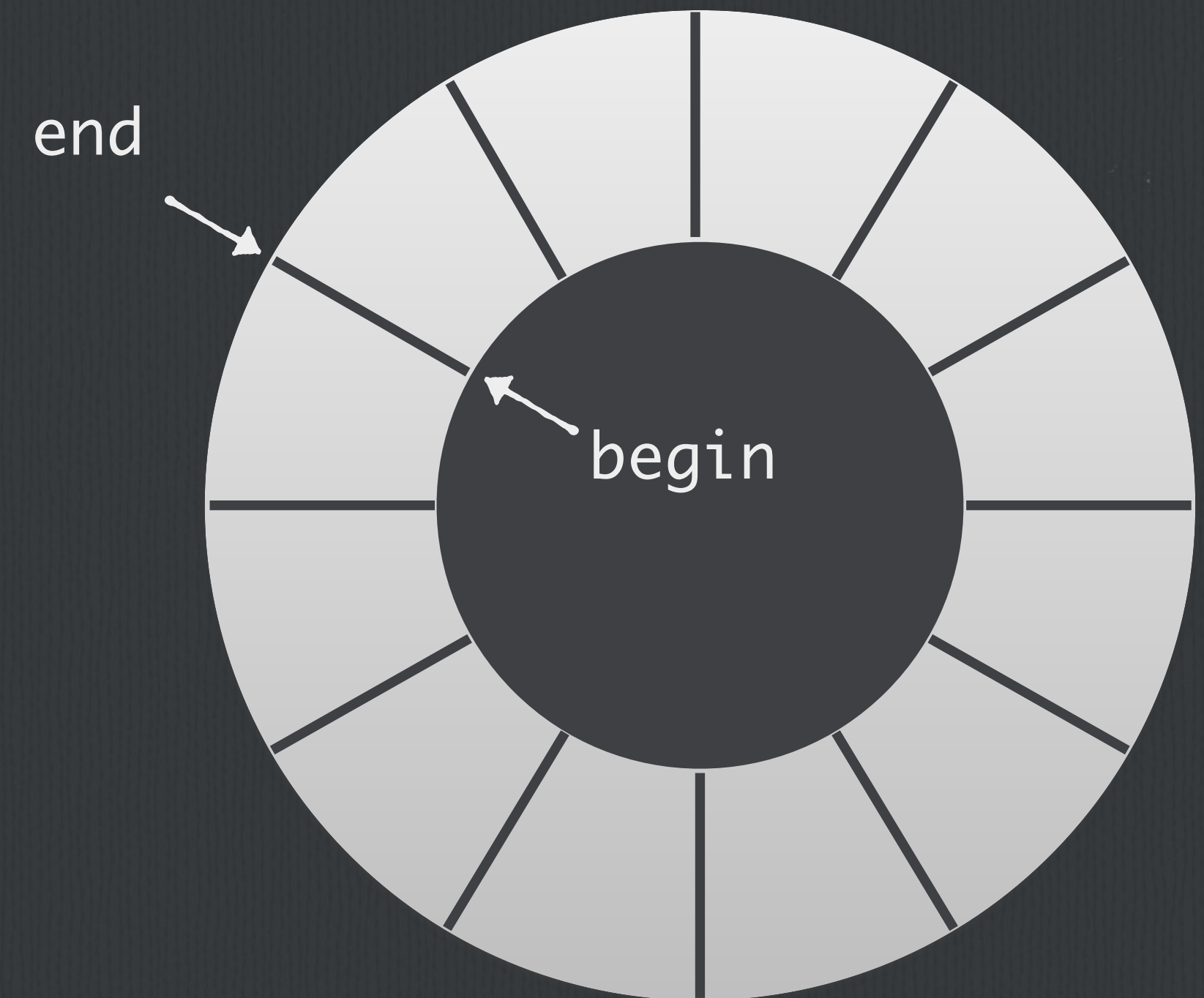
Boost.CircularBuffer

- ☐ Ring buffer implementation, fixed capacity
- ☐ STL-compatible API
- ☐ Possible use cases:
 - ☐ Cache, LIFO, FIFO



Boost.CircularBuffer

- ☐ Ring buffer implementation, fixed capacity
- ☐ STL-compatible API
- ☐ Possible use cases:
 - ☐ Cache, LIFO, FIFO



Boost.CircularBuffer

- ☐ Ring buffer implementation, fixed capacity
- ☐ STL-compatible API
- ☐ Possible use cases:
 - ☐ Cache, LIFO, FIFO



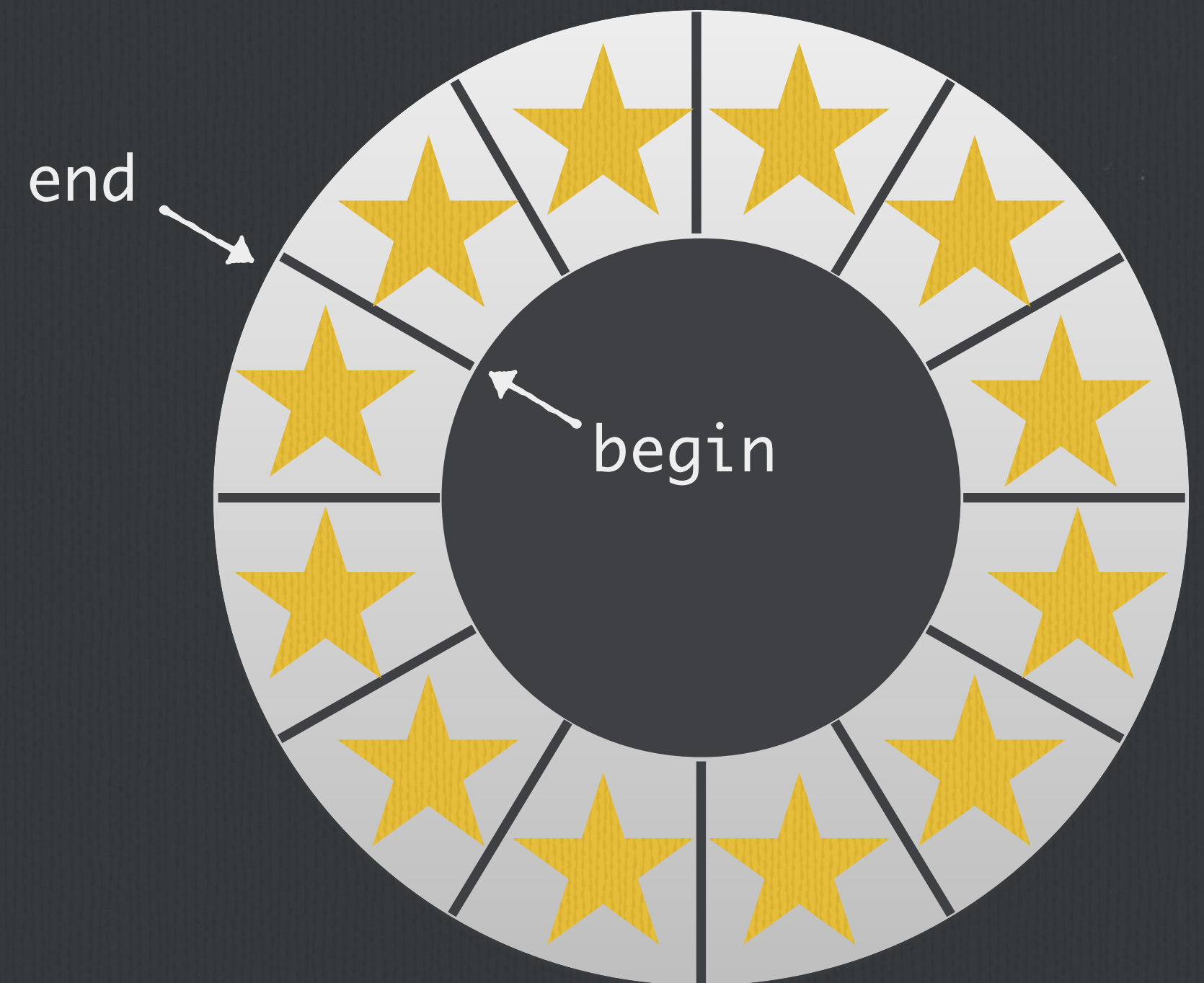
Boost.CircularBuffer

- ☐ Ring buffer implementation, fixed capacity
- ☐ STL-compatible API
- ☐ Possible use cases:
 - ☐ Cache, LIFO, FIFO



Boost.CircularBuffer

- ☐ Ring buffer implementation, fixed capacity
- ☐ STL-compatible API
- ☐ Possible use cases:
 - ☐ Cache, LIFO, FIFO



Boost.CircularBuffer

- ☐ Ring buffer implementation, fixed capacity
- ☐ STL-compatible API
- ☐ Possible use cases:
 - ☐ Cache, LIFO, FIFO



```
#include <boost/circular_buffer.hpp>

// Create a circular buffer with a capacity for 3 integers.
boost::circular_buffer<int> cb(3);

// Insert three elements into the buffer.
cb.push_back(1);
cb.push_back(2);
cb.push_back(3);

int a = cb[0]; // a == 1
int b = cb[1]; // b == 2
int c = cb[2]; // c == 3

// The buffer is full now, so pushing subsequent
// elements will overwrite the front-most elements.
cb.push_back(4); // Overwrite 1 with 4.
cb.push_back(5); // Overwrite 2 with 5.

// The buffer now contains 3, 4 and 5.
a = cb[0]; // a == 3
b = cb[1]; // b == 4
c = cb[2]; // c == 5

// Elements can be popped from either the front or the back.
cb.pop_back(); // 5 is removed.
cb.pop_front(); // 3 is removed.
```

Boost.Container

Boost.Container

- Several well-known containers, incl. STL containers

Boost.Container

- ☐ Several well-known containers, incl. STL containers
- ☐ Additional, advanced features

Boost.Container

- ☐ Several well-known containers, incl. STL containers
- ☐ Additional, advanced features
 - ☐ E.g. shared memory, support for incomplete types, recursive containers

Boost.Container

- ☐ Several well-known containers, incl. STL containers
- ☐ Additional, advanced features
 - ☐ E.g. shared memory, support for incomplete types, recursive containers
- ☐ Extended memory allocators & Polymorphic Memory Resources

Boost.Container: non-standard containers

- flat_(multi)set/map

Boost.Container: non-standard containers

- flat_(multi)set/map
 - **Ordered, vector-like containers instead of tree structures**

Boost.Container: non-standard containers

- ☐ `flat_(multi)set/map`
 - ☐ **Ordered, vector-like containers instead of tree structures**
 - ☐ **Provide faster lookup and iteration but with map/set-style access**

Boost.Container: non-standard containers

- ☐ `flat_(multi)set/map`
 - ☐ **Ordered, vector-like containers instead of tree structures**
 - ☐ **Provide faster lookup and iteration but with map/set-style access**
 - ☐ **Slower insertion and erasure**

Boost.Container: non-standard containers

Boost.Container: non-standard containers

- `small_vector`

Boost.Container: non-standard containers

- `small_vector`
 - **Vector-like container, optimized for few elements (stored on the stack)**

Boost.Container: non-standard containers

- ☐ **small_vector**
 - ☐ **Vector-like container, optimized for few elements (stored on the stack)**
 - ☐ **Can still grow beyond the initial capacity**

Boost.Container: non-standard containers

- ☐ `small_vector`
 - ☐ Vector-like container, optimized for few elements (stored on the stack)
 - ☐ Can still grow beyond the initial capacity
- ☐ `static_vector`

Boost.Container: non-standard containers

- ☐ `small_vector`
 - ☐ Vector-like container, optimized for few elements (stored on the stack)
 - ☐ Can still grow beyond the initial capacity
- ☐ `static_vector`
 - ☐ Like an array, but with a dynamic (limited) size

Boost.Container: non-standard containers

- ❑ **devector: Hybrid of vector and deque that can grow in both directions**

Boost.Container: non-standard containers

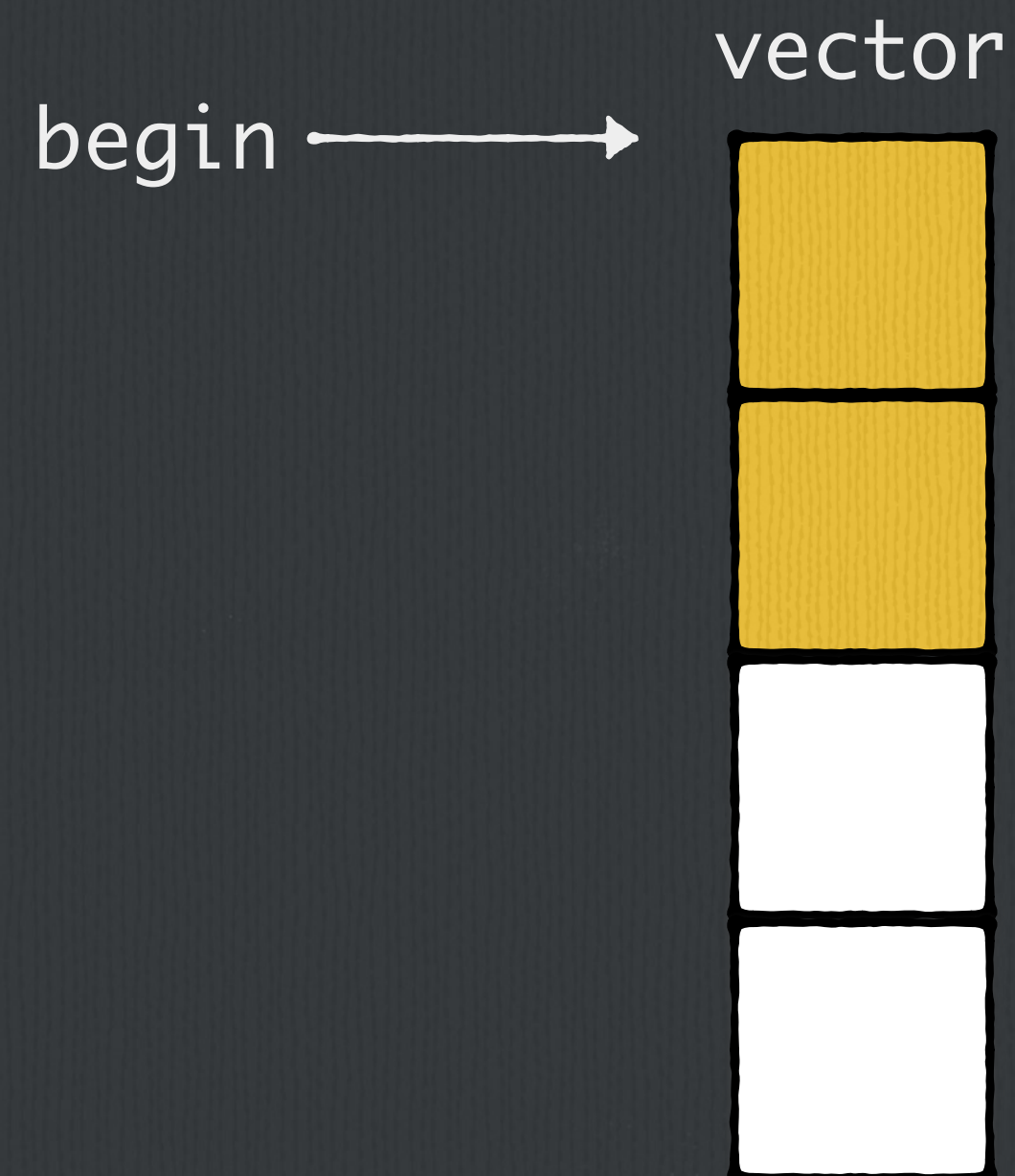
- ❑ devector: Hybrid of vector and deque that can grow in both directions

vector



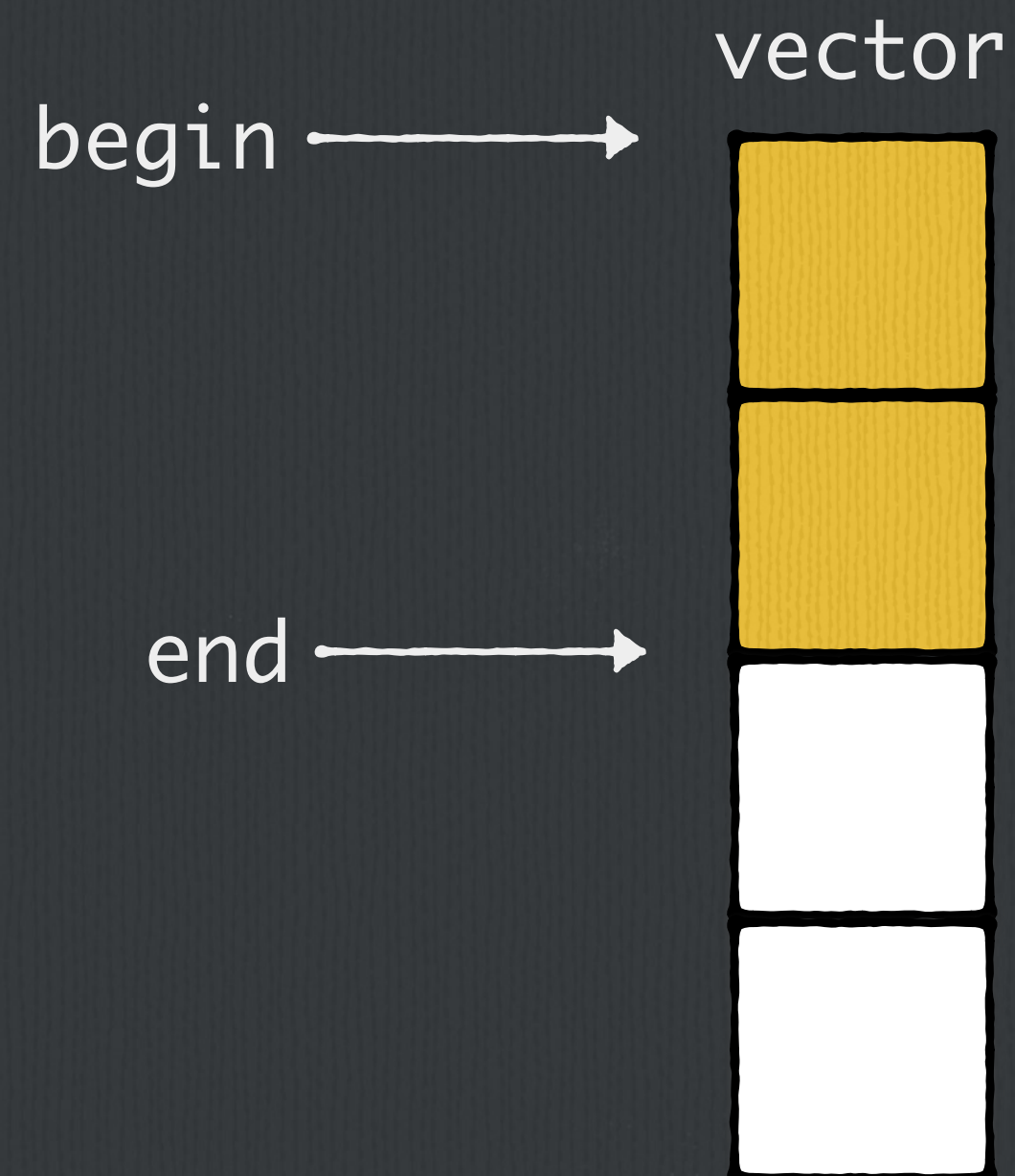
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



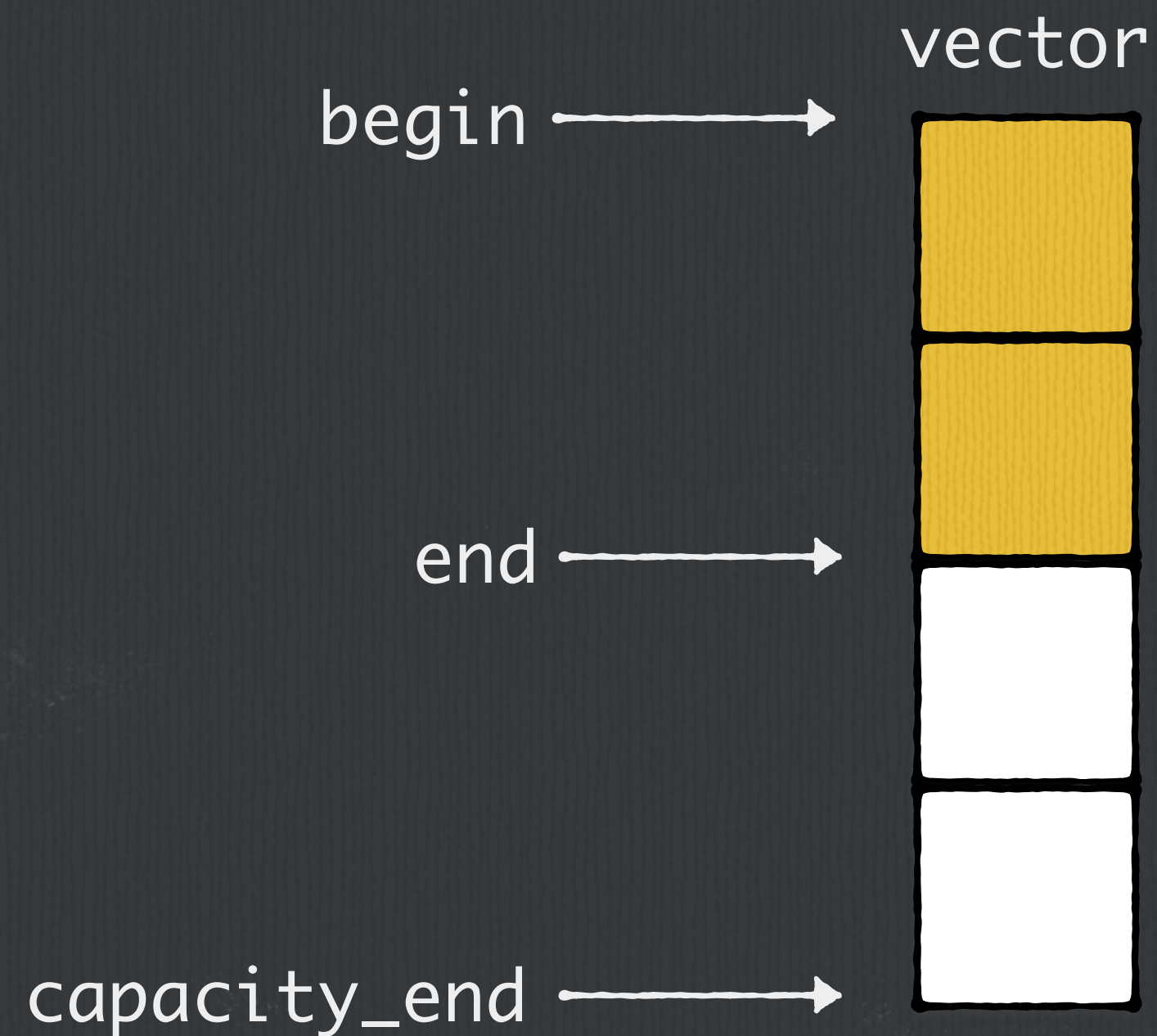
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



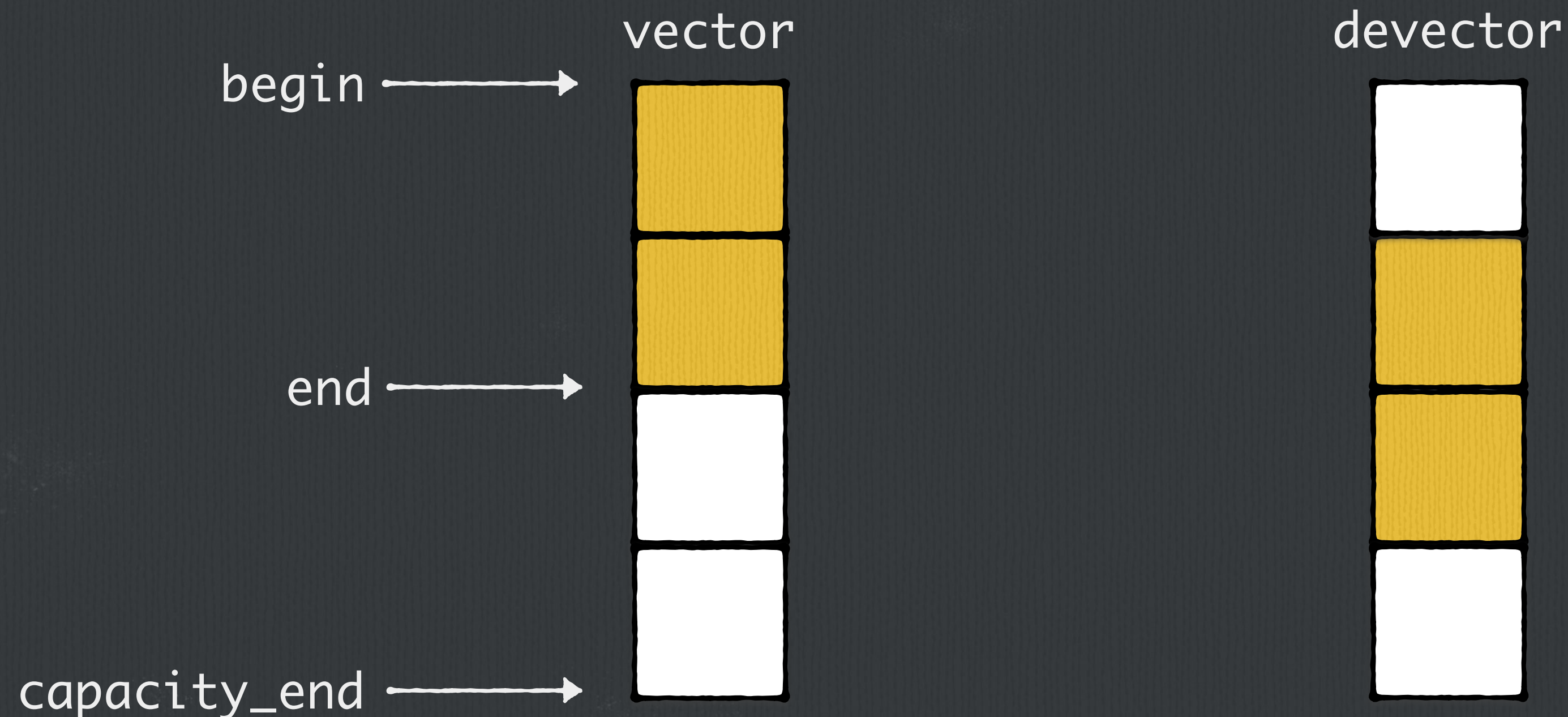
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



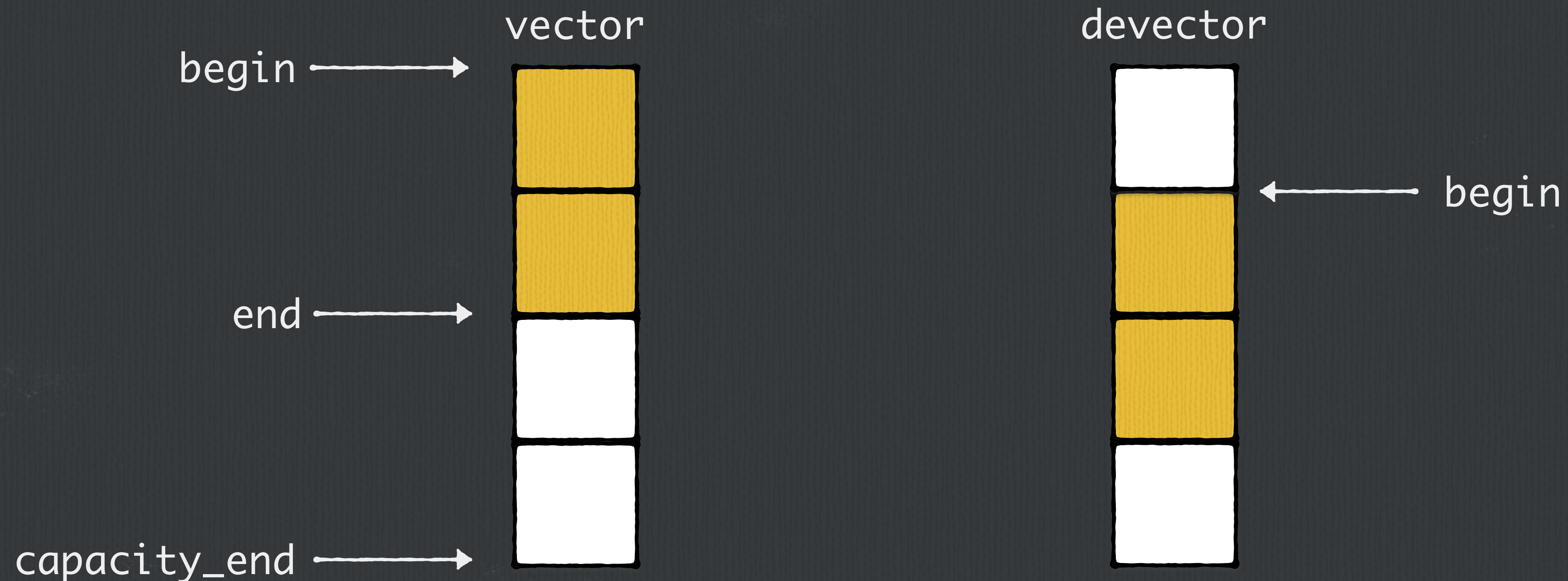
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



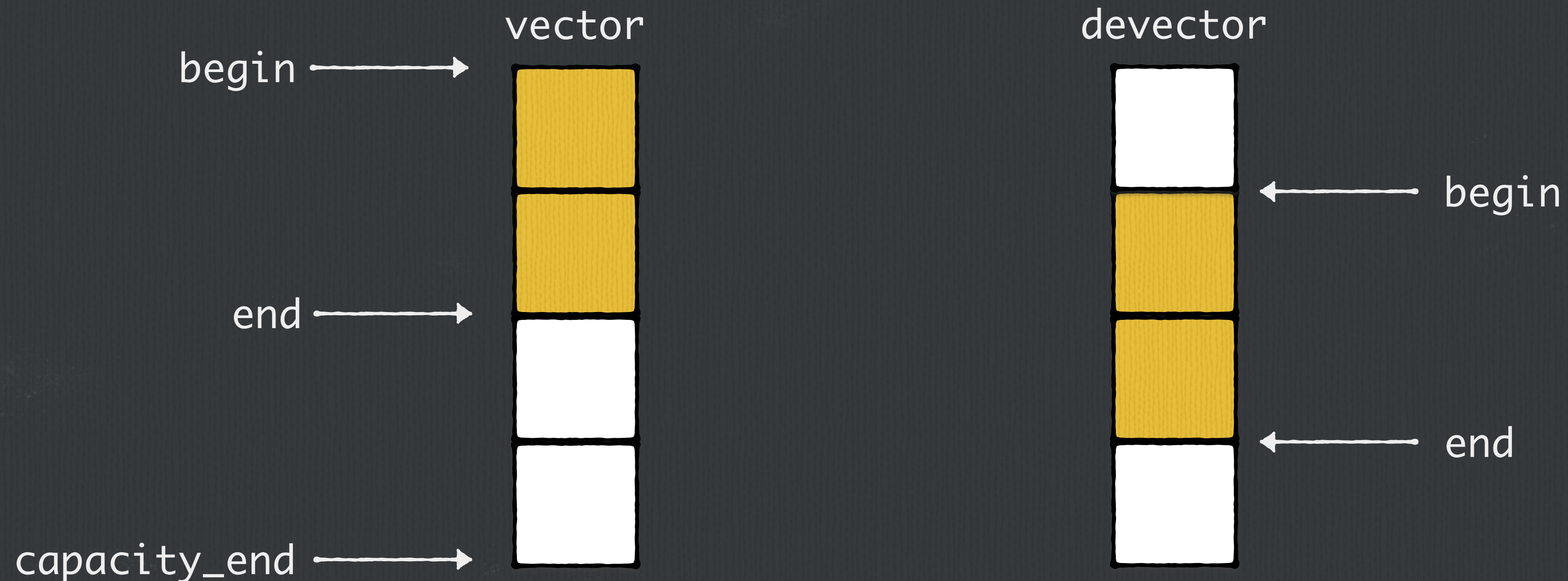
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



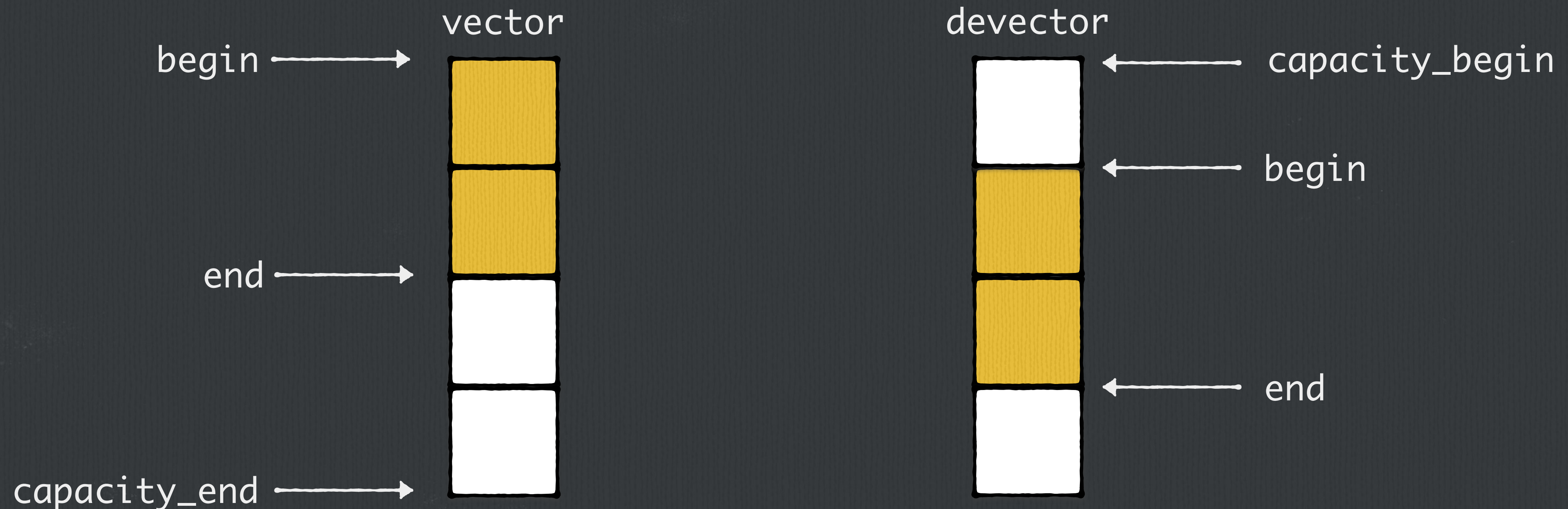
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



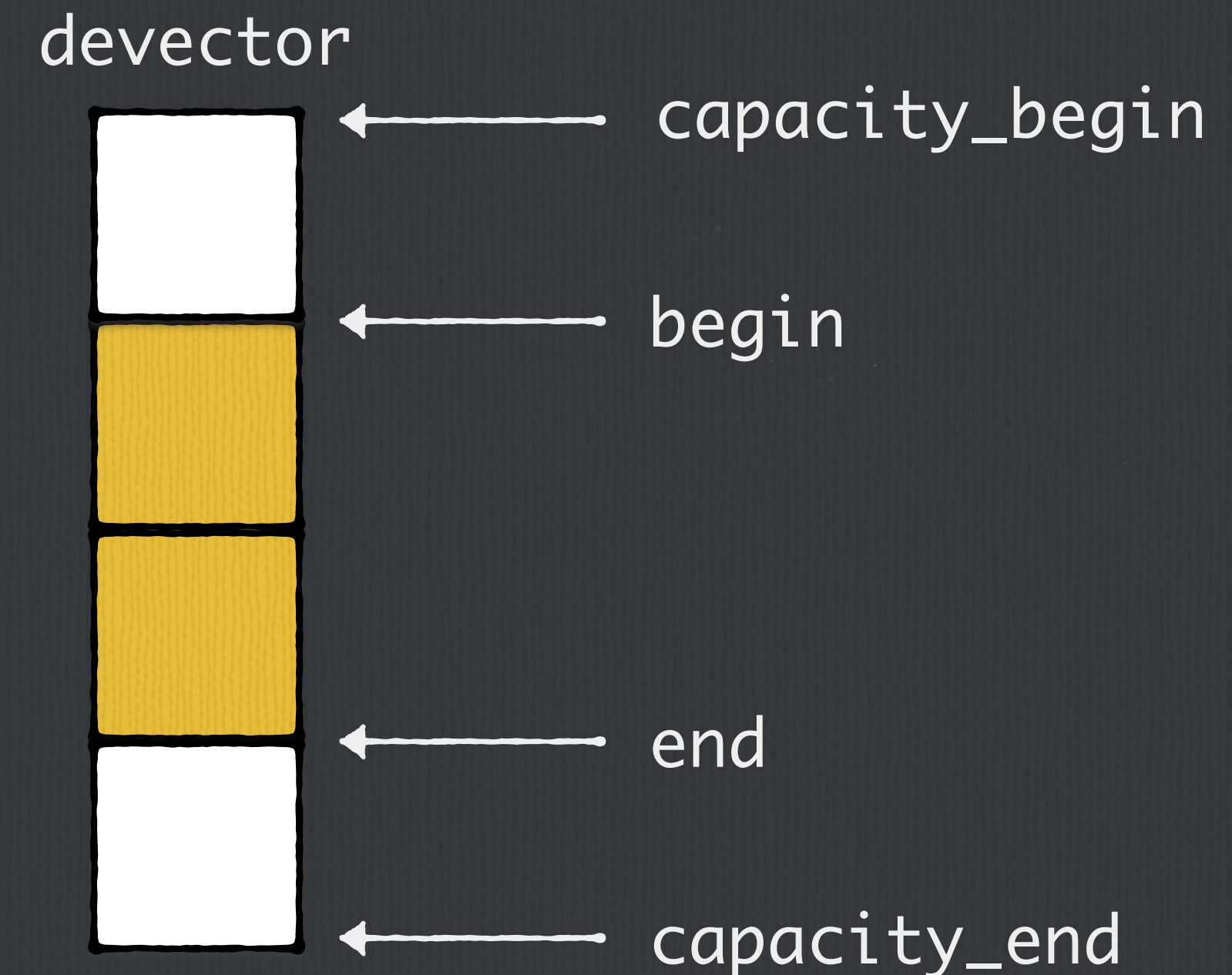
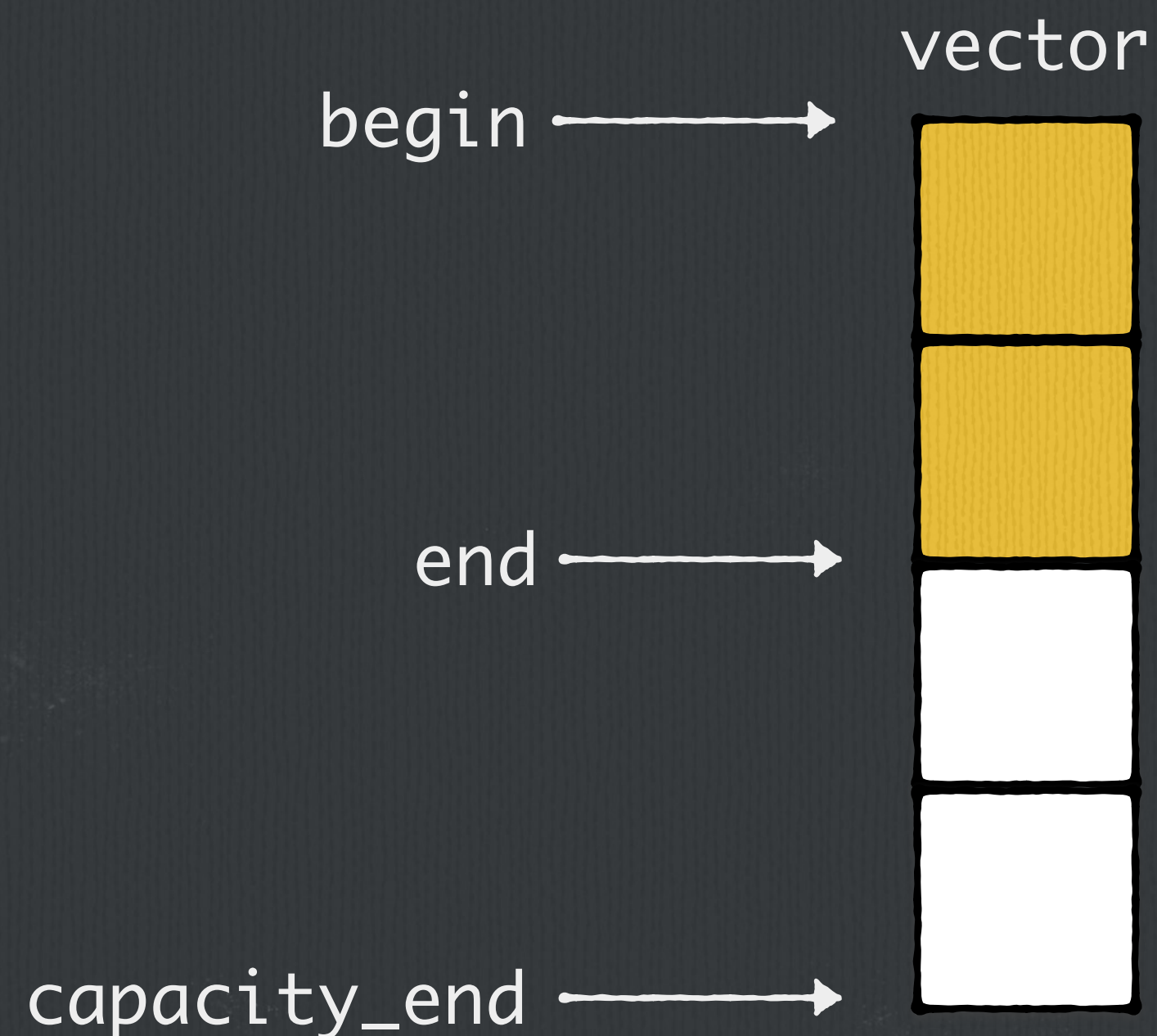
Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



Boost.Container: non-standard containers

- devector: Hybrid of vector and deque that can grow in both directions



Boost.Container: advanced options

Boost.Container: advanced options

- ☐ **Change the `size_type` of a vector (e.g. use 8 bit to store the size)**

Boost.Container: advanced options

- ☐ **Change the `size_type` of a vector (e.g. use 8 bit to store the size)**
- ☐ **Change the growth policy of a vector / `small_vector`**

Boost.Container: advanced options

- ☐ **Change the `size_type` of a vector (e.g. use 8 bit to store the size)**
- ☐ **Change the growth policy of a vector / `small_vector`**
- ☐ **Custom in-place alignment for `static_vector` / `small_vector`**

Boost.Container: advanced options

- ☐ **Change the `size_type` of a vector (e.g. use 8 bit to store the size)**
- ☐ **Change the growth policy of a vector / `small_vector`**
- ☐ **Custom in-place alignment for `static_vector` / `small_vector`**
- ☐ **Customize the block size of a deque**

Boost.Container: advanced options

- ☐ **Change the `size_type` of a vector (e.g. use 8 bit to store the size)**
- ☐ **Change the growth policy of a vector / `small_vector`**
- ☐ **Custom in-place alignment for `static_vector` / `small_vector`**
- ☐ **Customize the block size of a deque**
- ☐ **Different memory allocation strategies**

Boost.ICL

Boost.ICL



Boost.ICL

☐ “Interval Container Library”



Boost.ICL

- ☐ “Interval Container Library”

- ☐ `interval_set`

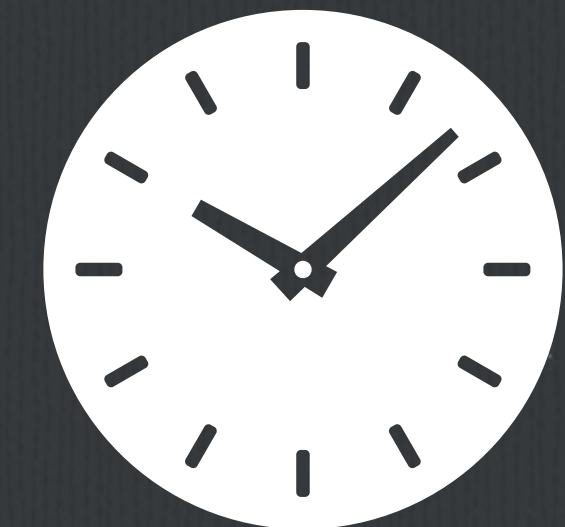


Boost.ICL

- ☐ “Interval Container Library”
- ☐ `interval_set`
- ☐ `interval_map`



Boost.ICL



- ☐ “Interval Container Library”
- ☐ `interval_set`
- ☐ `interval_map`
- ☐ Supports various interval properties (open, closed, overlap, accumulation, ...)

Boost.Intrusive

Boost.Intrusive

- Building block for other libraries, e.g. Boost.MultiIndex

Boost.Intrusive

- ❑ Building block for other libraries, e.g. Boost.MultiIndex
- ❑ Main differences to “normal” containers:

Boost.Intrusive

- ❑ Building block for other libraries, e.g. Boost.MultiIndex
- ❑ Main differences to “normal” containers:
 - ❑ Objects hold meta-information required by containers (e.g. next*, prev*)

Boost.Intrusive

- ❑ Building block for other libraries, e.g. Boost.MultiIndex
- ❑ Main differences to “normal” containers:
 - ❑ Objects hold meta-information required by containers (e.g. next*, prev*)
 - ❑ Containers don't store/manage objects, just their order

Boost.Intrusive

- ☐ Building block for other libraries, e.g. Boost.MultiIndex
- ☐ Main differences to “normal” containers:
 - ☐ Objects hold meta-information required by containers (e.g. next*, prev*)
 - ☐ Containers don't store/manage objects, just their order
 - ☐ Can be faster for some use cases, but easy to blow up

Boost.MultiArray

Boost.MultiArray

- Class template for multi-dimensional arrays (+ views)

Boost.MultiArray

- ❑ Class template for multi-dimensional arrays (+ views)
- ❑ STL-compatible alternative to native arrays

Boost.MultiArray

- ☐ **Class template for multi-dimensional arrays (+ views)**
- ☐ **STL-compatible alternative to native arrays**
- ☐ **Dynamic size support**

Boost.MultiArray

- ☐ Class template for multi-dimensional arrays (+ views)
- ☐ STL-compatible alternative to native arrays
- ☐ Dynamic size support
- ☐ Configurable storage order (e.g. for FORTRAN interoperability)

Boost.MultiArray

- ☐ Class template for multi-dimensional arrays (+ views)
- ☐ STL-compatible alternative to native arrays
- ☐ Dynamic size support
- ☐ Configurable storage order (e.g. for FORTRAN interoperability)
- ☐ Supports non-zero-based indices

```
// Create a 3D array that is 3 x 4 x 2
using array_type = boost::multi_array<double, 3>;
array_type A(boost::extents[3][4][2]);

// Assign values to the elements
using index = array_type::index;
int values = 0;
for (index i = 0; i != 3; ++i)
    for (index j = 0; j != 4; ++j)
        for (index k = 0; k != 2; ++k)
            A[i][j][k] = values++;

// Alternative access using a collection of indices
boost::array<array_type::index, 3> idx = {{0,0,0}};
A(idx) = 42;
```

Boost.MultiIndex

Boost.MultiIndex

- Allows construction of containers with one or more “indices”

Boost.MultiIndex

- ☐ Allows construction of containers with one or more “indices”
- ☐ Indices can have different sorting and access semantics (e.g. sort orders, hash table lookup etc.)

Boost.MultiIndex

- ☐ Allows construction of containers with one or more “indices”
- ☐ Indices can have different sorting and access semantics (e.g. sort orders, hash table lookup etc.)
- ☐ Sample use cases:

Boost.MultiIndex

- ☐ Allows construction of containers with one or more “indices”
- ☐ Indices can have different sorting and access semantics (e.g. sort orders, hash table lookup etc.)
- ☐ Sample use cases:
 - ☐ Containers with different iteration orders (e.g. by ID + by name)

Boost.MultiIndex

- ☐ Allows construction of containers with one or more “indices”
- ☐ Indices can have different sorting and access semantics (e.g. sort orders, hash table lookup etc.)
- ☐ Sample use cases:
 - ☐ Containers with different iteration orders (e.g. by ID + by name)
 - ☐ Lists with fast look up and/or without duplicates

Boost.MultiIndex

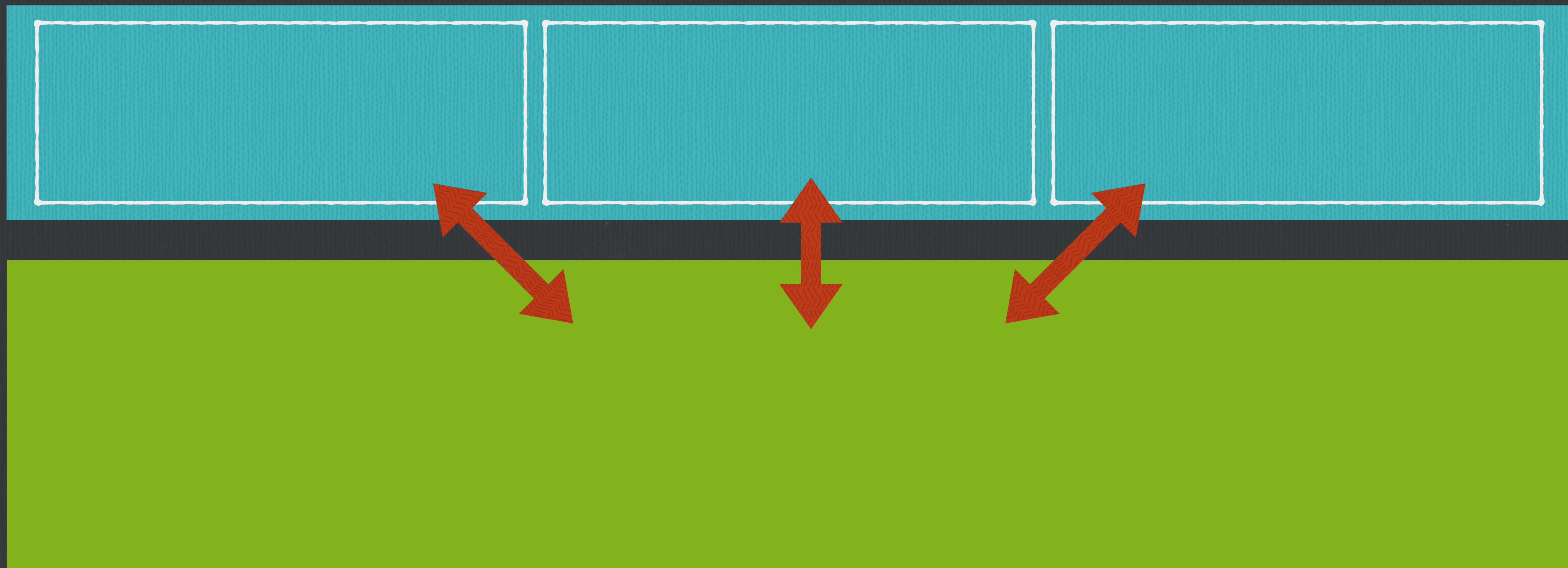
- ☐ Allows construction of containers with one or more “indices”
- ☐ Indices can have different sorting and access semantics (e.g. sort orders, hash table lookup etc.)
- ☐ Sample use cases:
 - ☐ Containers with different iteration orders (e.g. by ID + by name)
 - ☐ Lists with fast look up and/or without duplicates
 - ☐ Bidirectional maps (search by key or value)

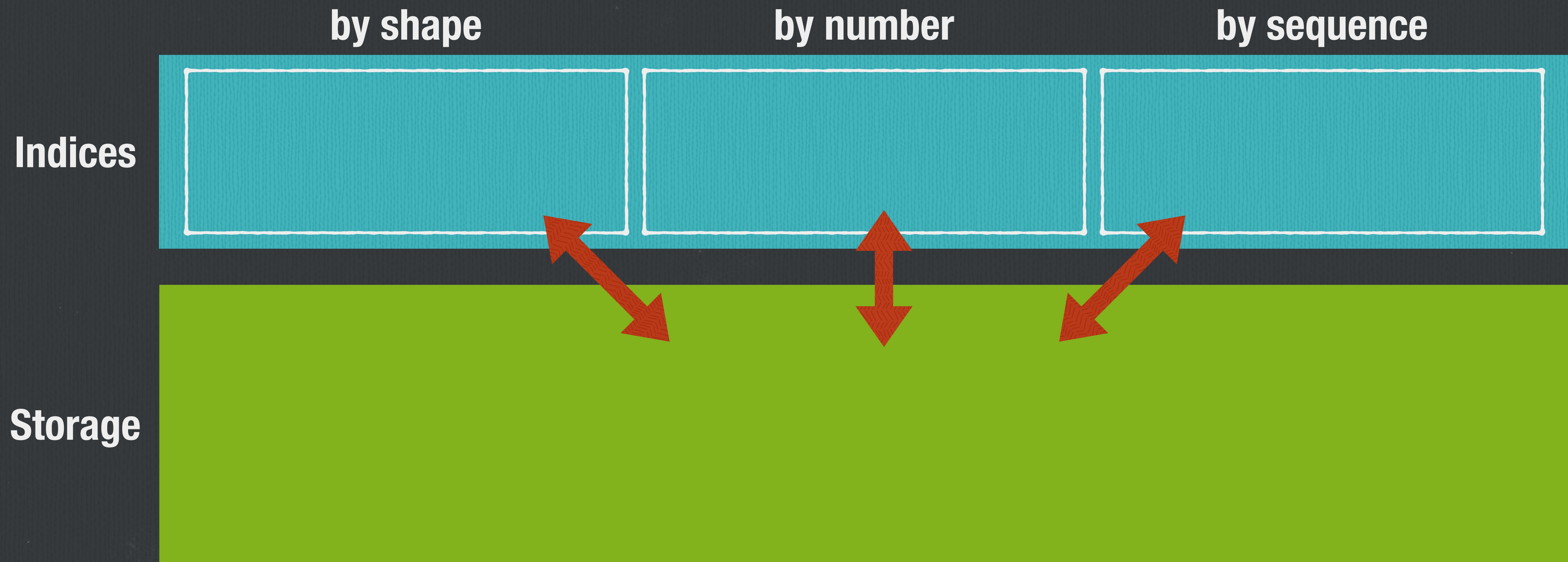
Boost.MultiIndex

- ☐ Allows construction of containers with one or more “indices”
- ☐ Indices can have different sorting and access semantics (e.g. sort orders, hash table lookup etc.)
- ☐ Sample use cases:
 - ☐ Containers with different iteration orders (e.g. by ID + by name)
 - ☐ Lists with fast look up and/or without duplicates
 - ☐ Bidirectional maps (search by key or value)
 - ☐ MRU/LRU lists

Indices

Storage





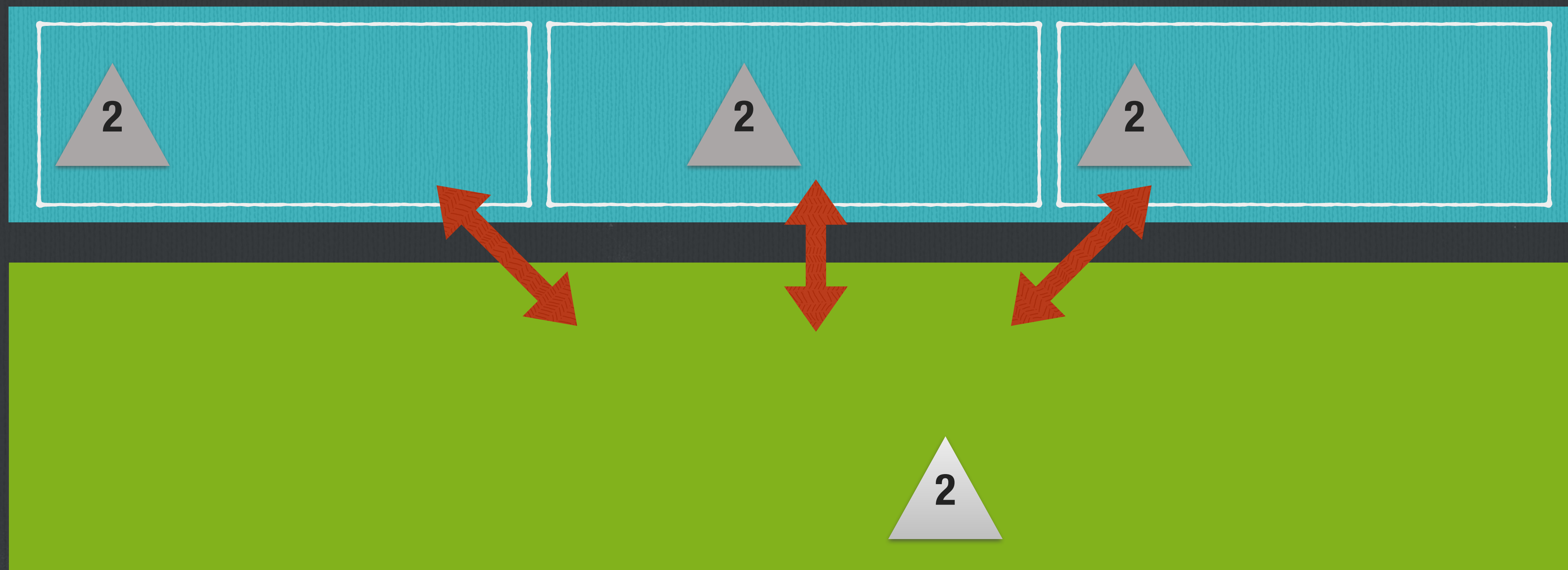
Indices

by shape

by number

by sequence

Storage



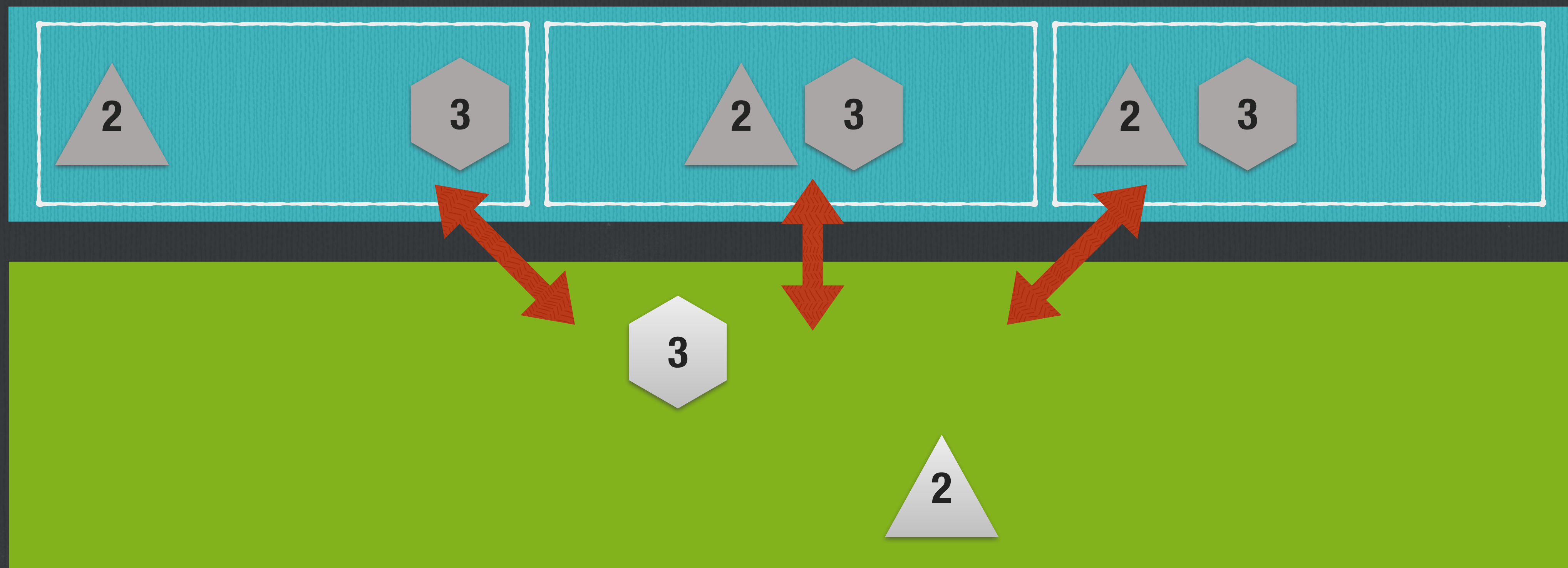
Indices

by shape

by number

by sequence

Storage



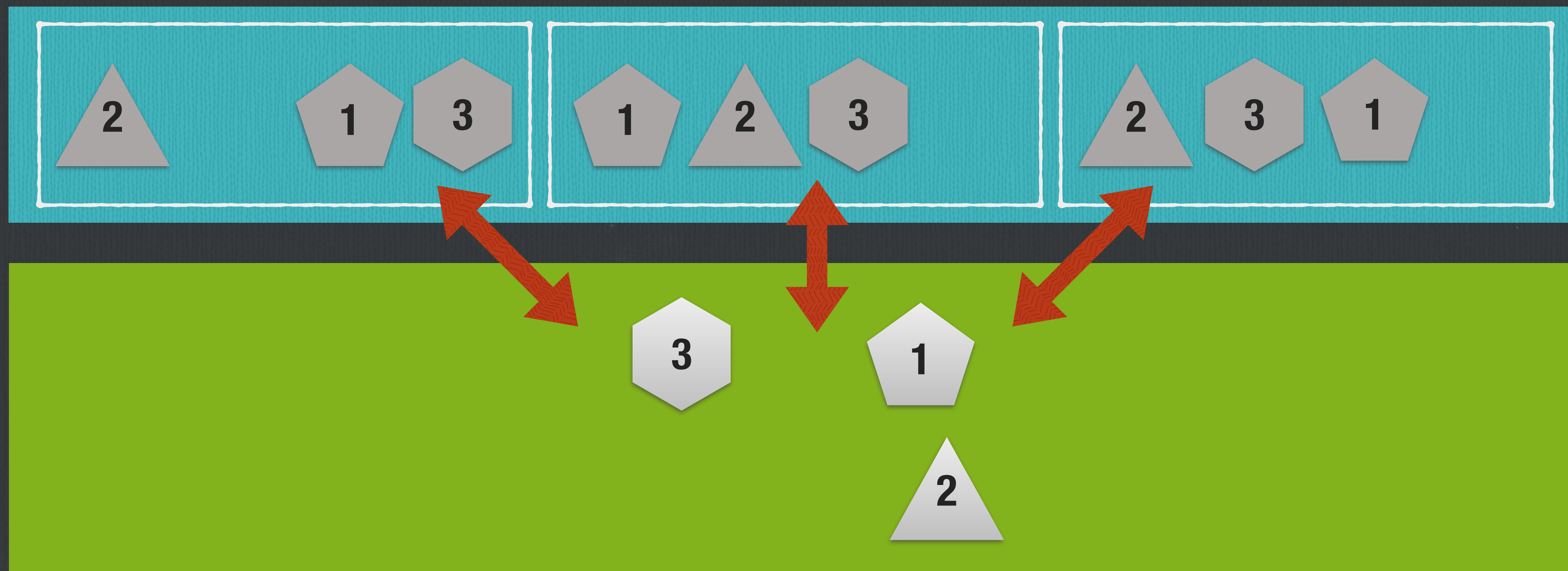
Indices

by shape

by number

by sequence

Storage



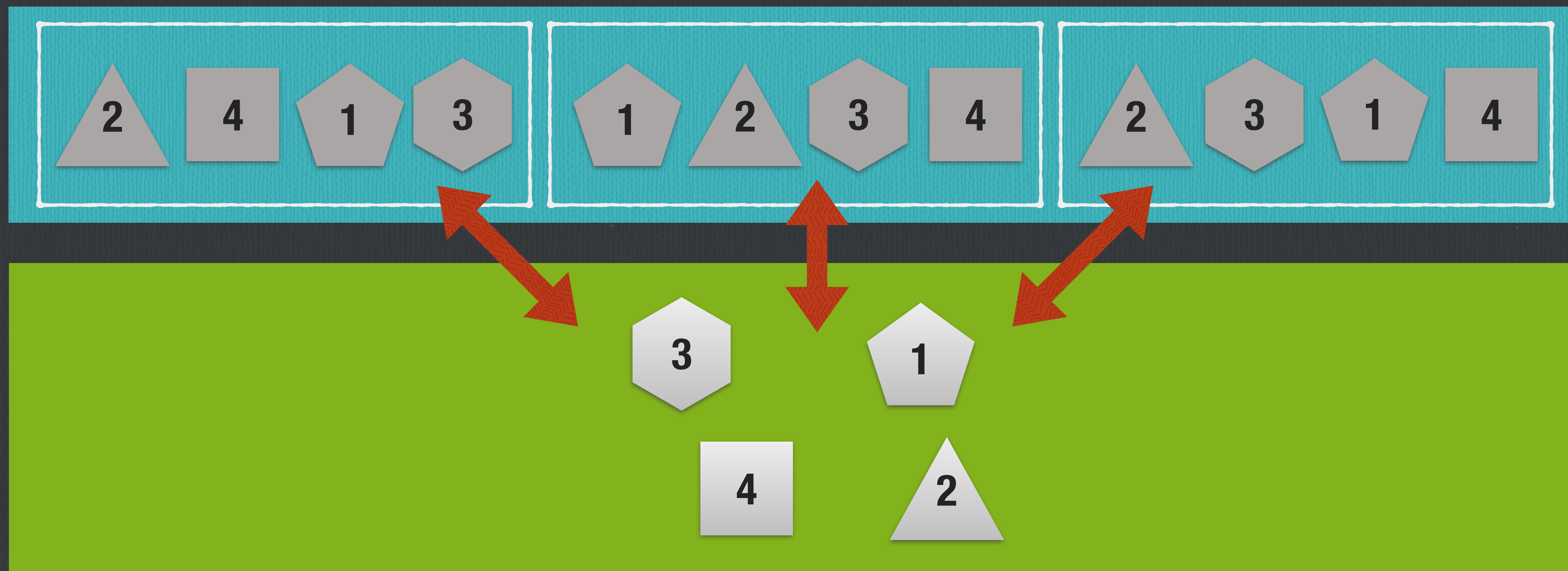
Indices

by shape

by number

by sequence

Storage



```

#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/member.hpp>
using namespace boost::multi_index;

struct Employee {
    int id;
    std::string name;
    bool operator<(const Employee& e) const { return id < e.id; }
};
// define a multiply indexed set with indices by id and name
using EmployeeSet = multi_index_container<
    Employee,
    indexed_by<
        // sort by employee::operator< (no duplicate entries)
        ordered_unique<identity<Employee>>,
        // sort by less<string> on name
        ordered_non_unique<member<Employee, std::string, &Employee::name>>
    >
>;

void iterate_by_name(const EmployeeSet& es) {
    // get a view to index #1 (name) – also supports "tagging" instead of numeric indexes
    const auto& name_index = es.get<1>();
    for (const auto& employee: name_index) { /* ... */ }
}

```

Boost.PointerContainer

Boost.PointerContainer

- Containers storing pointers to heap-allocated objects

Boost.PointerContainer

- Containers storing pointers to heap-allocated objects
 - ptr_vector, ptr_set, ptr_map, ...

Boost.PointerContainer

- ❑ Containers storing pointers to heap-allocated objects
 - ❑ ptr_vector, ptr_set, ptr_map, ...
- ❑ Can be faster than storing values or smart pointers

Boost.PointerContainer

- ❑ Containers storing pointers to heap-allocated objects
 - ❑ ptr_vector, ptr_set, ptr_map, ...
- ❑ Can be faster than storing values or smart pointers
- ❑ Supports deep-copy, propagates const

Boost.PointerContainer

- ❑ Containers storing pointers to heap-allocated objects
 - ❑ `ptr_vector`, `ptr_set`, `ptr_map`, ...
- ❑ Can be faster than storing values or smart pointers
- ❑ Supports deep-copy, propagates `const`
- ❑ API designed for pointers (e.g. returns references instead of pointers)

Boost.PolyCollection

Boost.PolyCollection

□ base_collection:

Boost.PolyCollection

- `base_collection`:
 - Optimizes for containers of “base classes”

Boost.PolyCollection

- **base_collection:**
 - **Optimizes for containers of “base classes”**
 - **Avoids storing (smart) pointers by storing derived classes by value**

Boost.PolyCollection

- **base_collection:**
 - **Optimizes for containers of “base classes”**
 - **Avoids storing (smart) pointers by storing derived classes by value**
 - **i.e. less heap allocations, better memory locality**

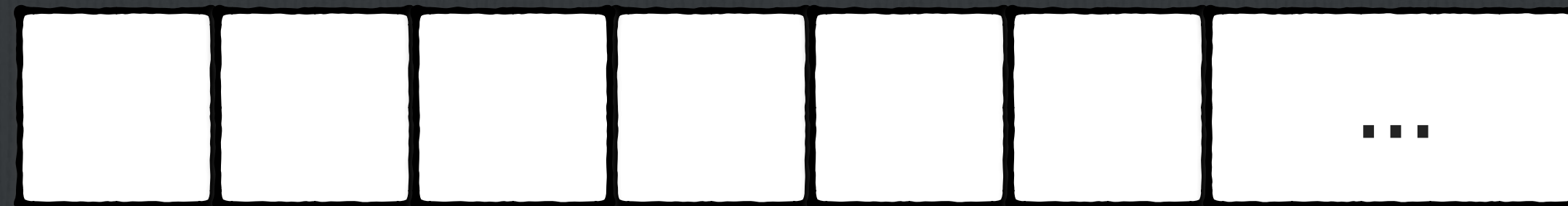
Boost.PolyCollection

- **base_collection:**
 - **Optimizes for containers of “base classes”**
 - **Avoids storing (smart) pointers by storing derived classes by value**
 - **i.e. less heap allocations, better memory locality**
 - **Order of traversal not defined ...**

Boost.PolyCollection

- `base_collection`:
 - Optimizes for containers of “base classes”
 - Avoids storing (smart) pointers by storing derived classes by value
 - i.e. less heap allocations, better memory locality
 - Order of traversal not defined ...
- Similar: `function_collection`, `any_collection`

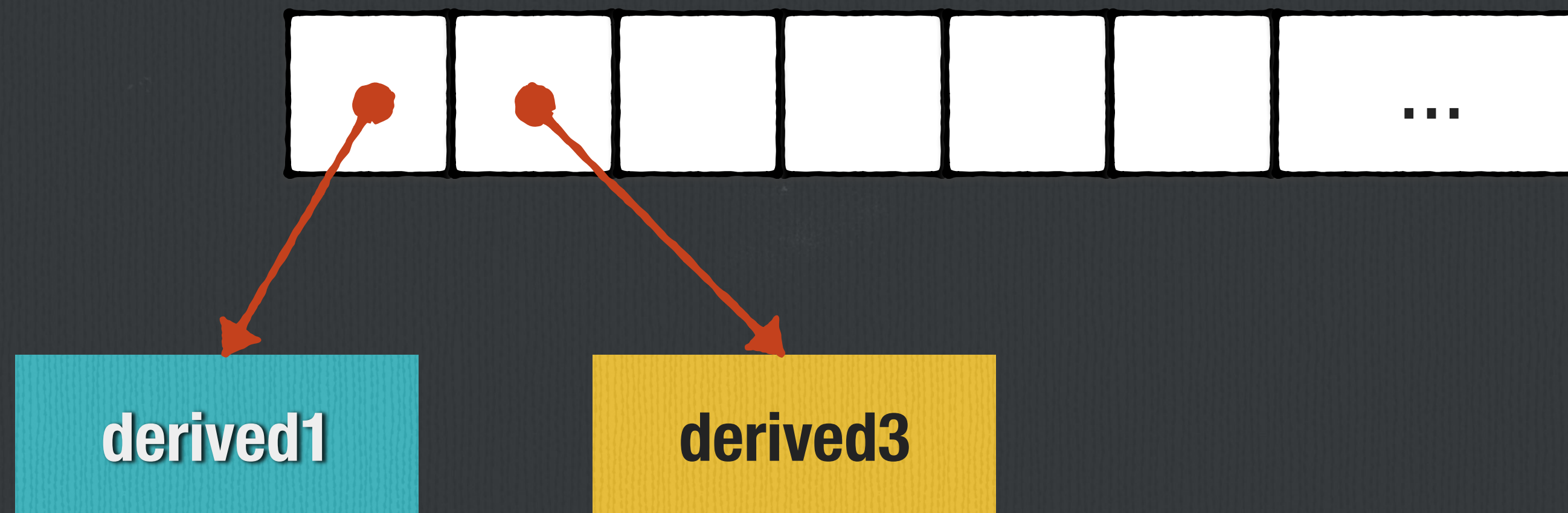
Vector of base class pointers



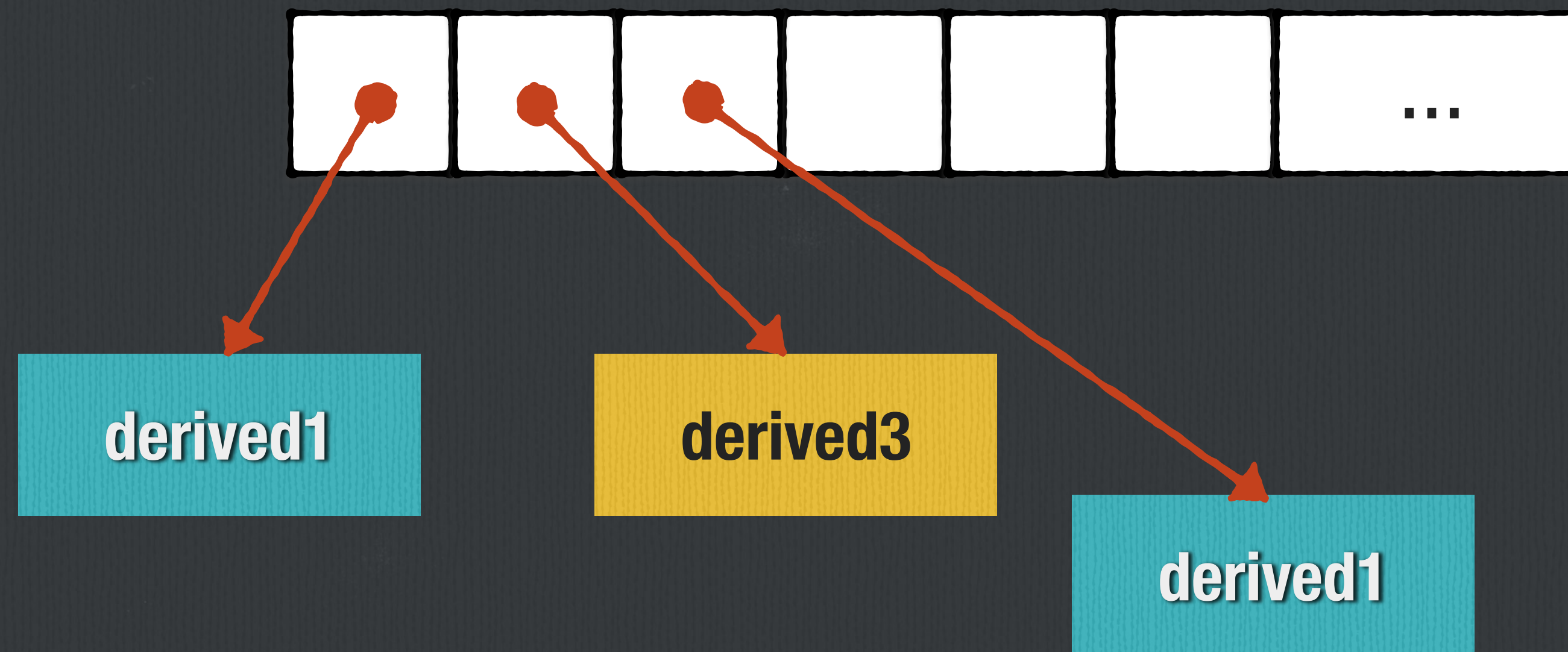
Vector of base class pointers



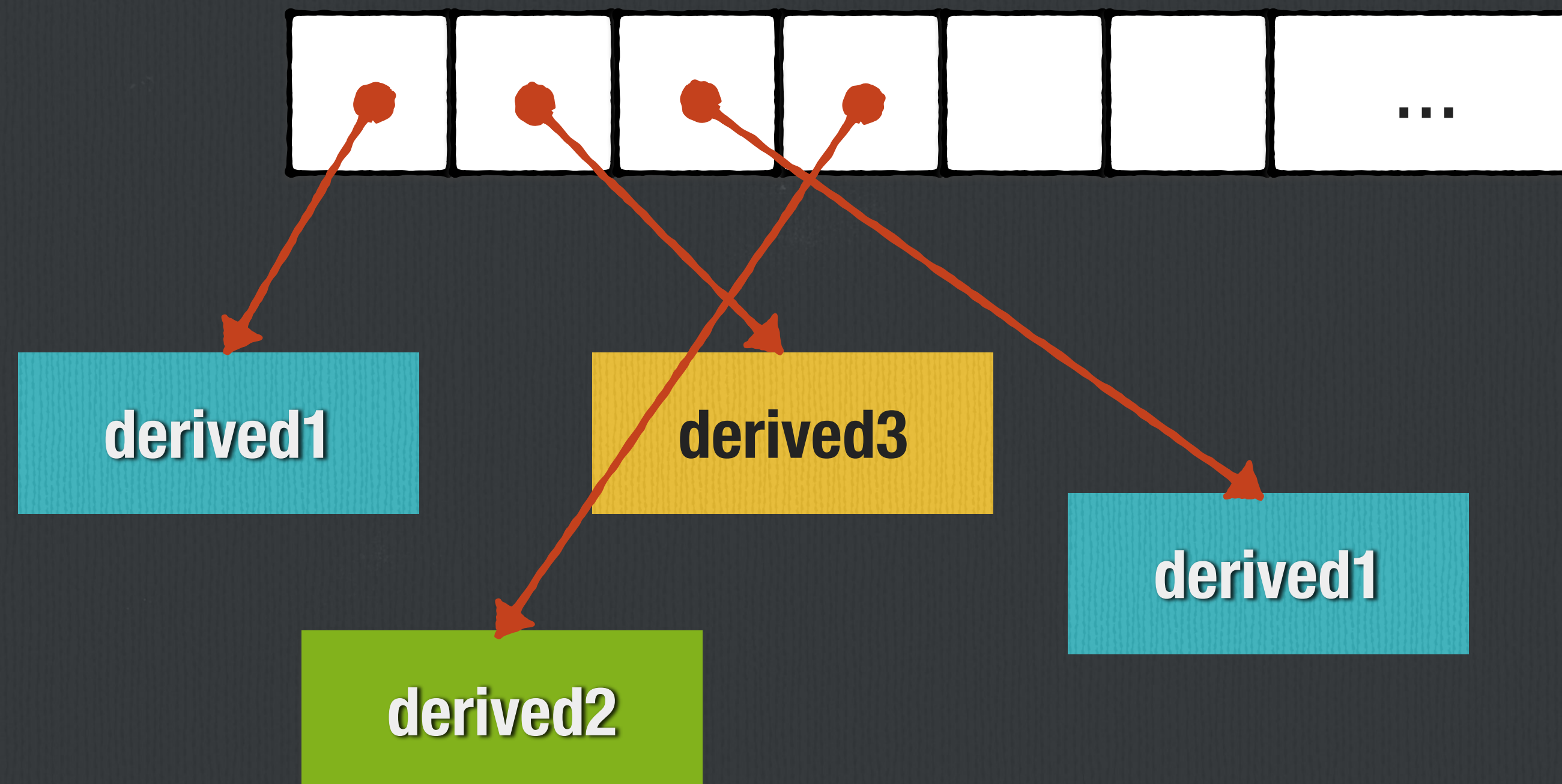
Vector of base class pointers



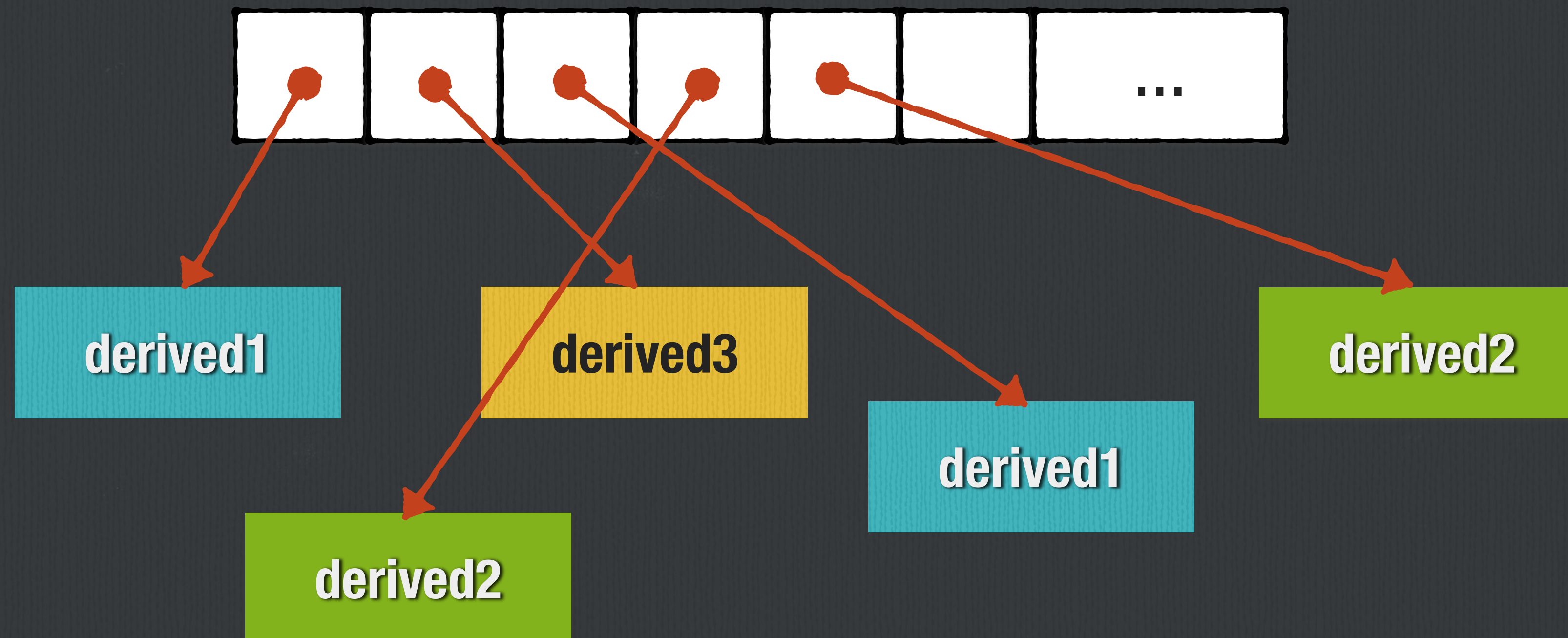
Vector of base class pointers



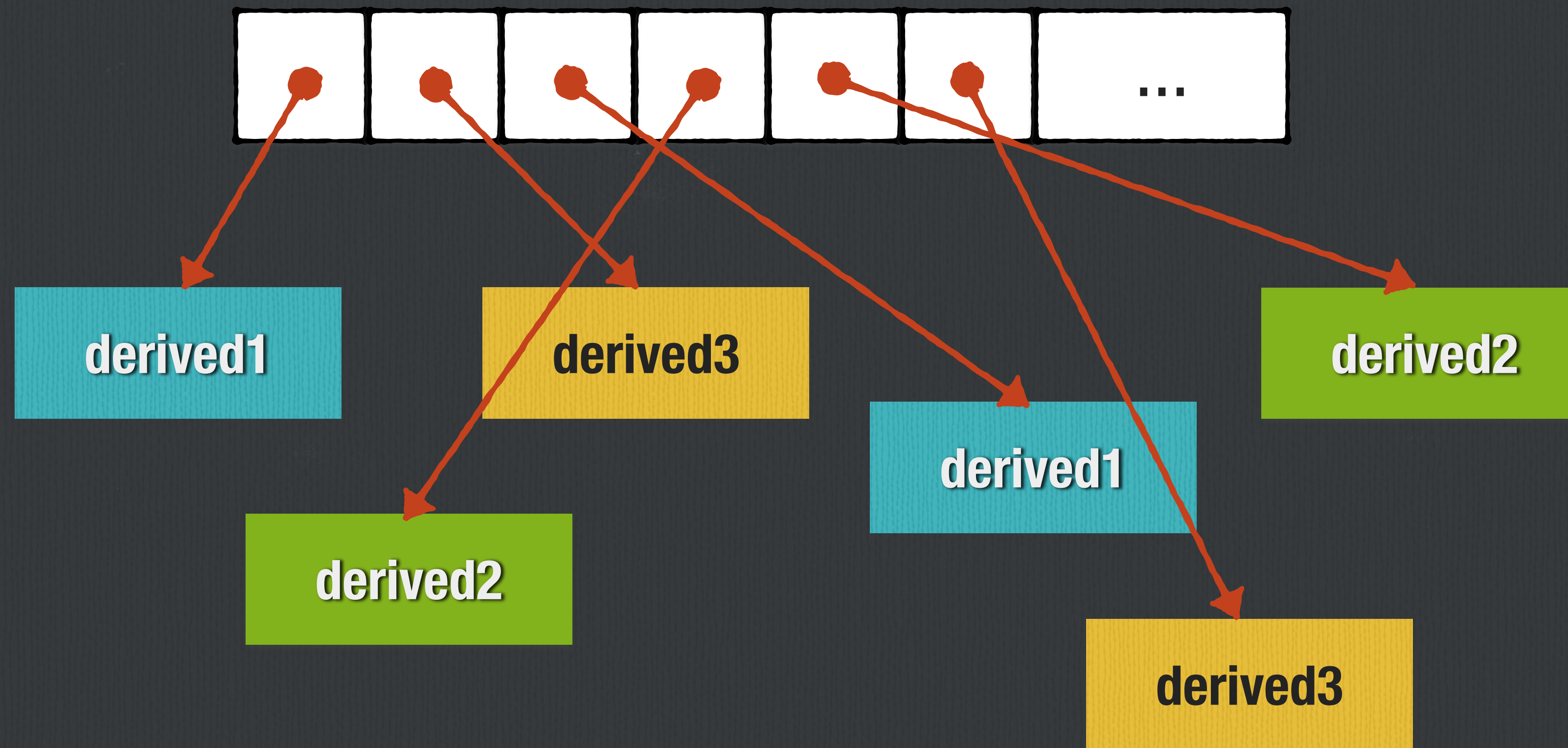
Vector of base class pointers



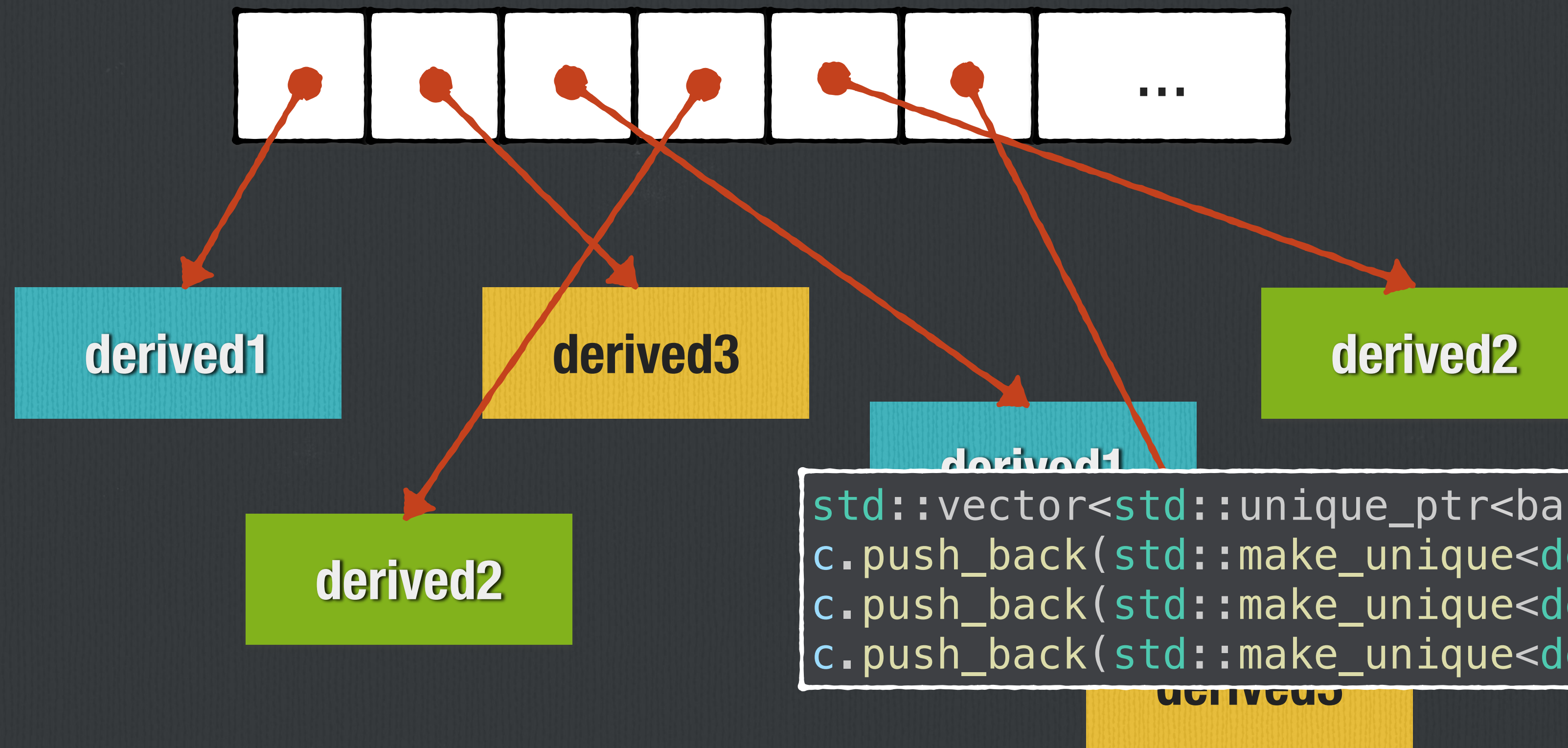
Vector of base class pointers



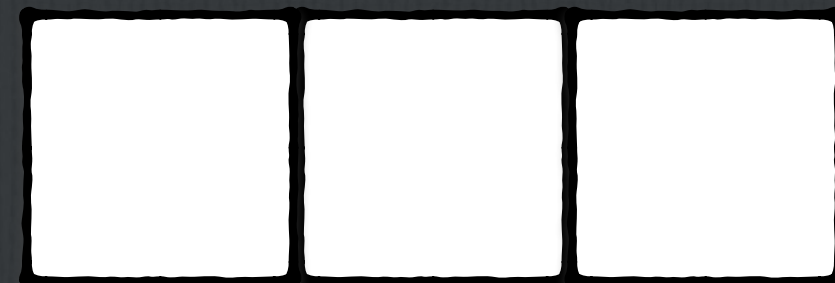
Vector of base class pointers



Vector of base class pointers



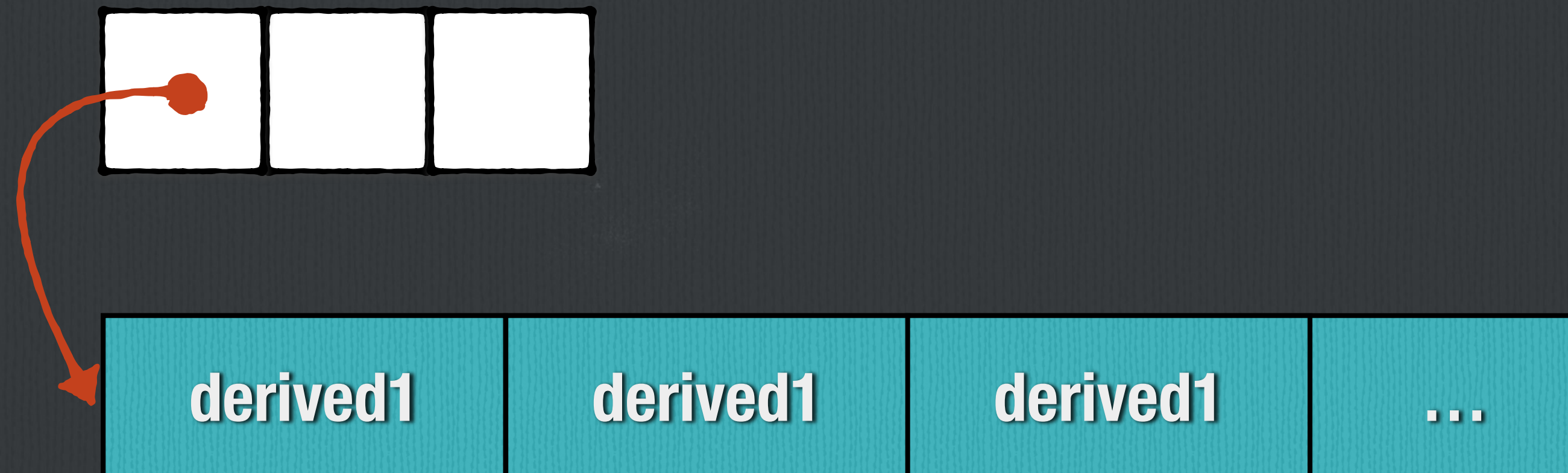
boost::base_collection



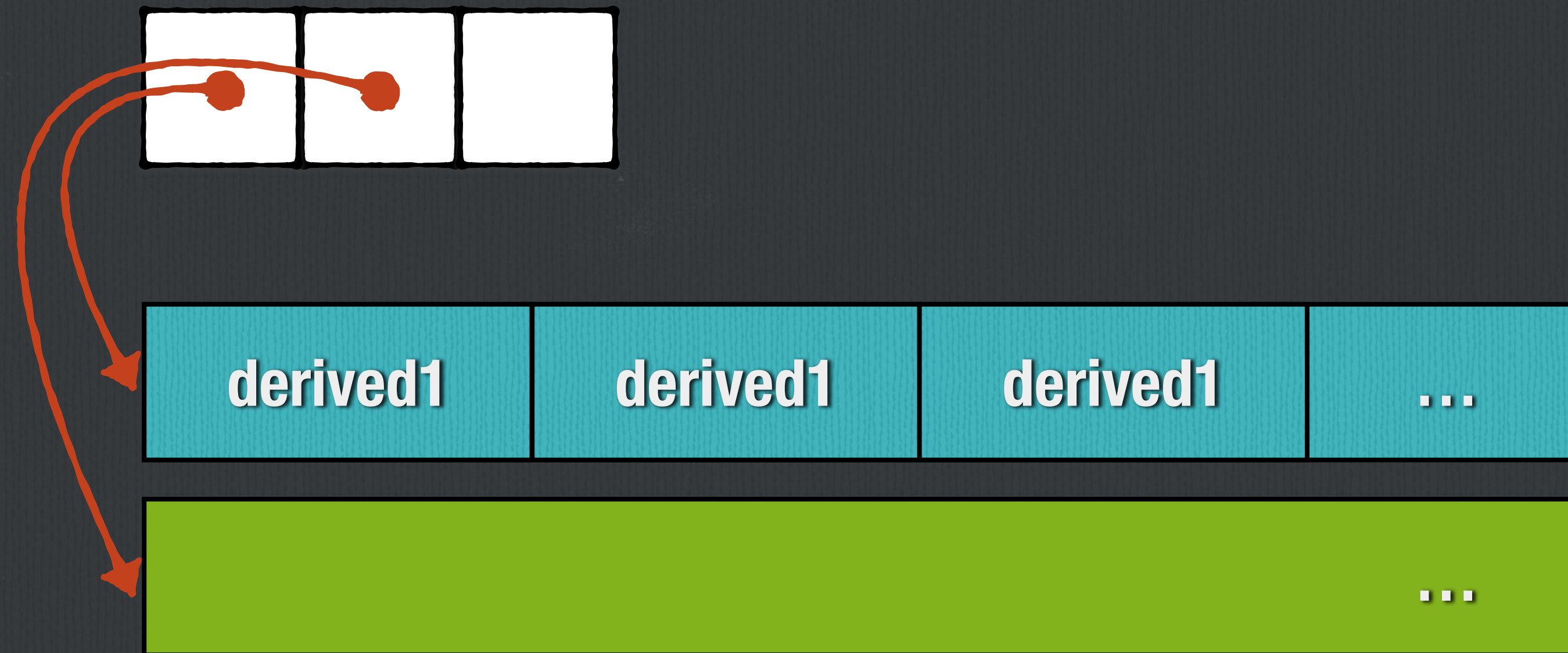
boost::base_collection



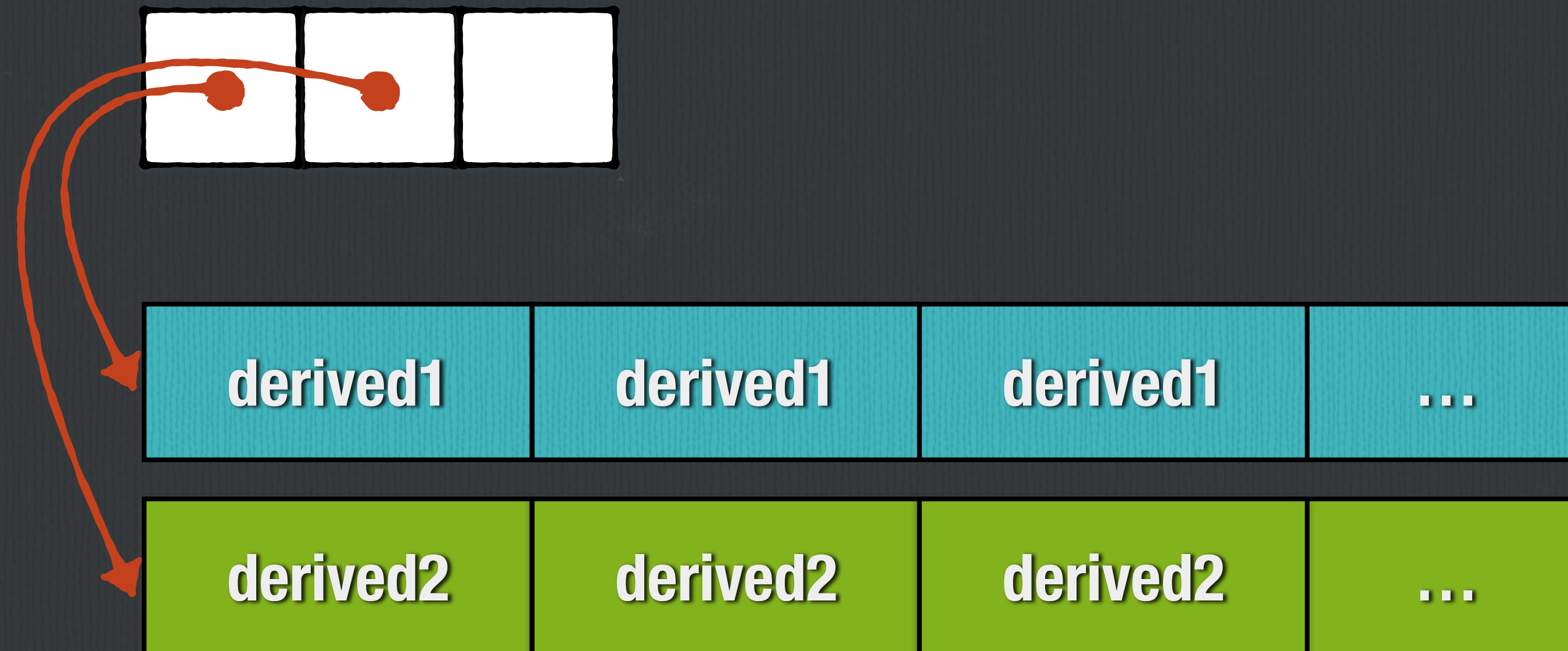
boost::base_collection



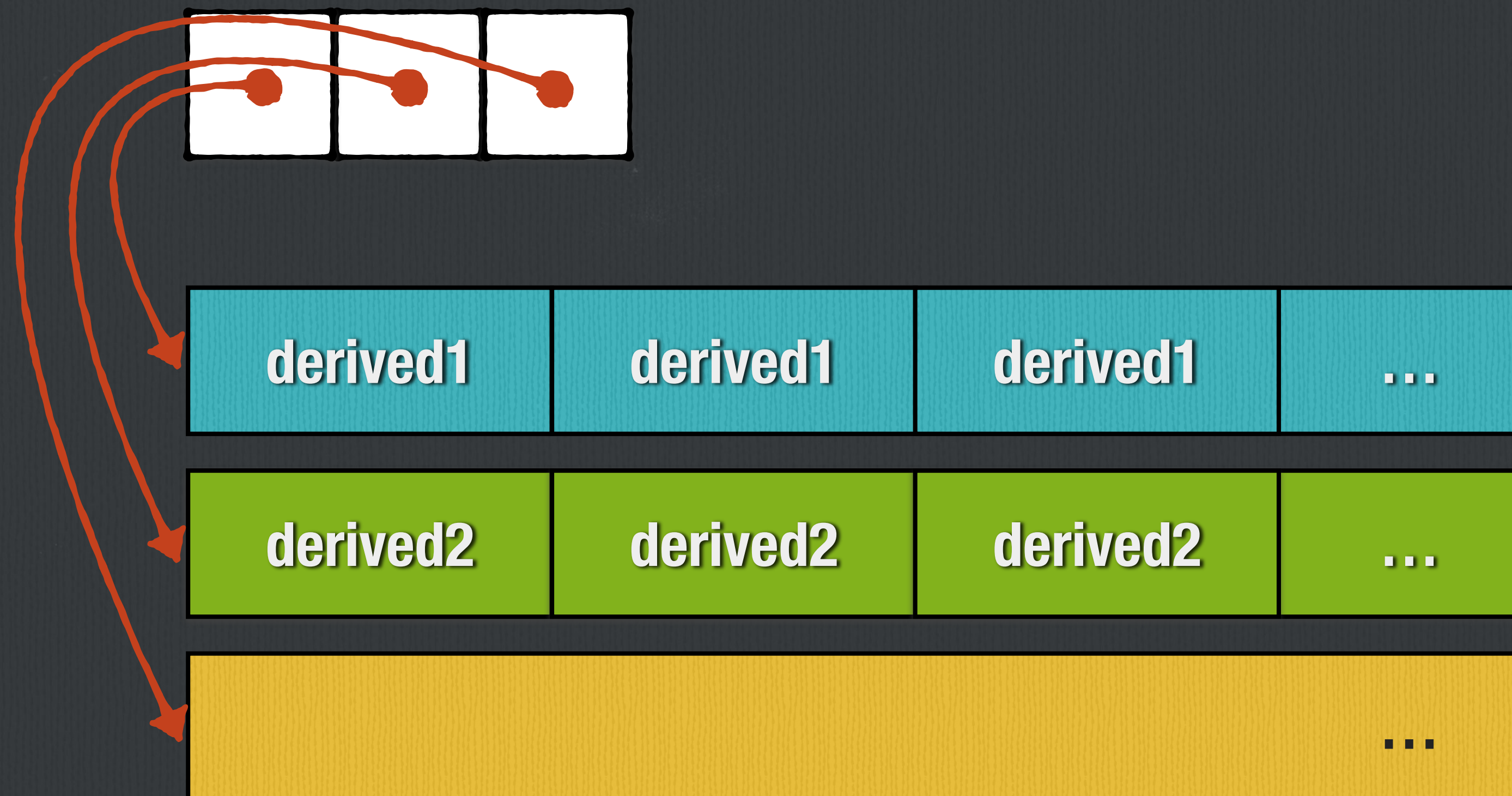
boost::base_collection



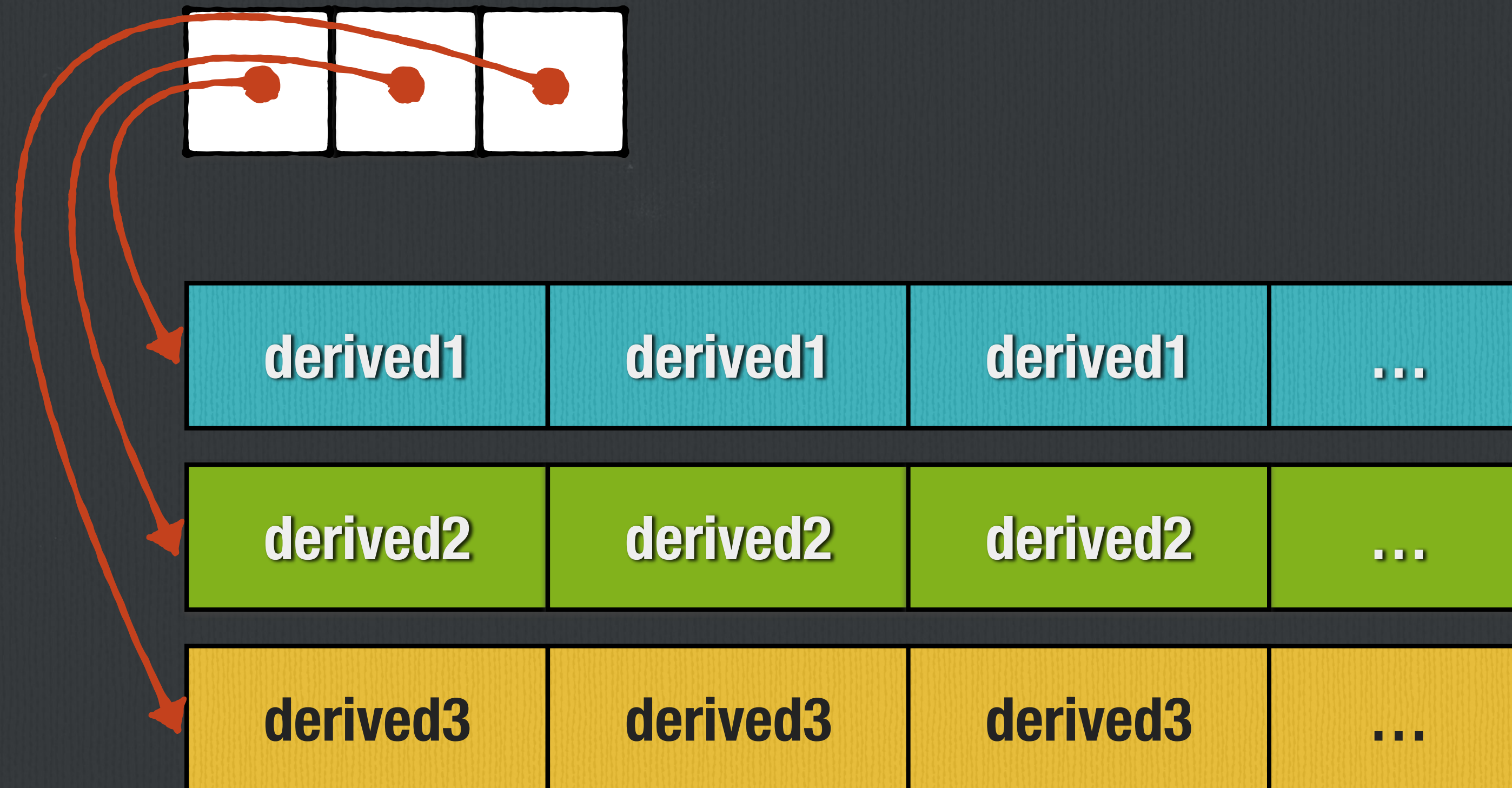
boost::base_collection



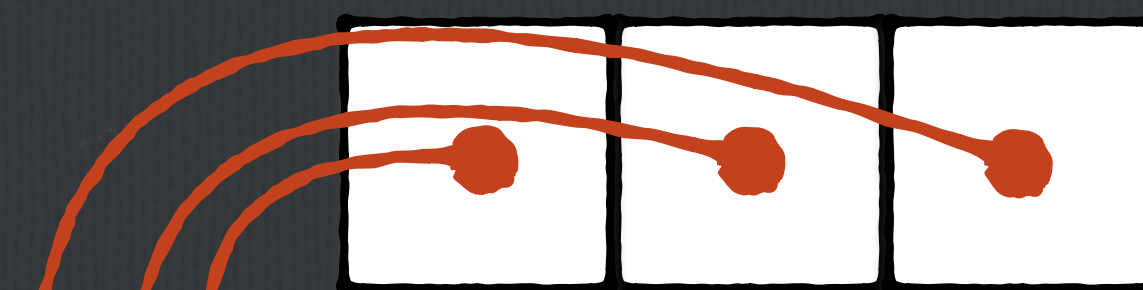
boost::base_collection



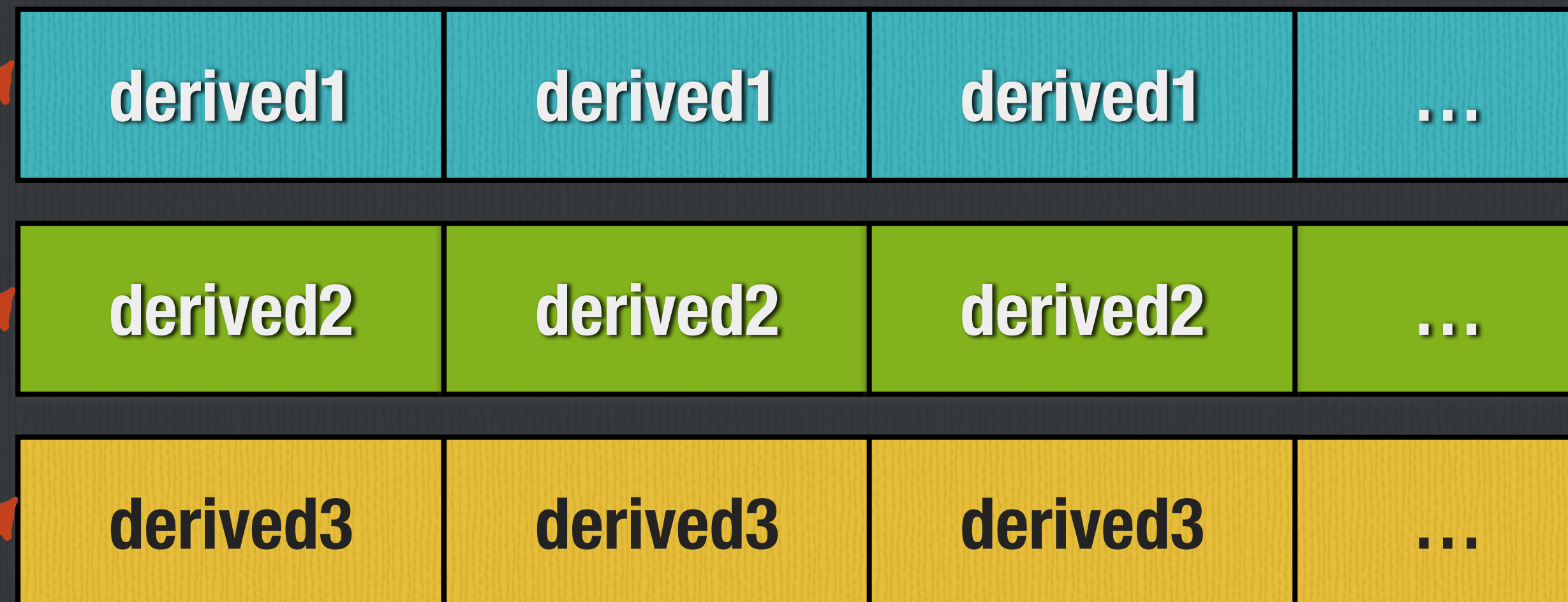
boost::base_collection



boost::base_collection



```
boost::base_collection<base> c;  
c.insert(derived1{});  
c.insert(derived2{});  
c.insert(derived3{});
```



Boost.Unordered

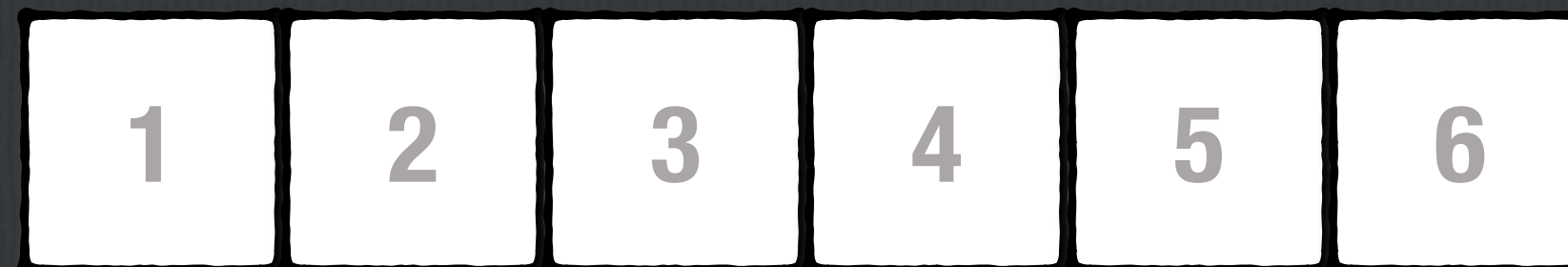
(Hash-tables)	Node-based	Flat
Closed addressing	<code>boost::unordered_set</code> <code>boost::unordered_map</code> <code>boost::unordered_multiset</code> <code>boost::unordered_multimap</code>	
Open addressing	<code>boost::unordered_node_set</code> <code>boost::unordered_node_map</code>	<code>boost::unordered_flat_set</code> <code>boost::unordered_flat_map</code>
Concurrent		<code>boost::concurrent_flat_map</code>

Closed-addressing containers

- ☐ fully compliant with the C++ specification for unordered associative containers
- ☐ “one of the fastest implementations in the market”

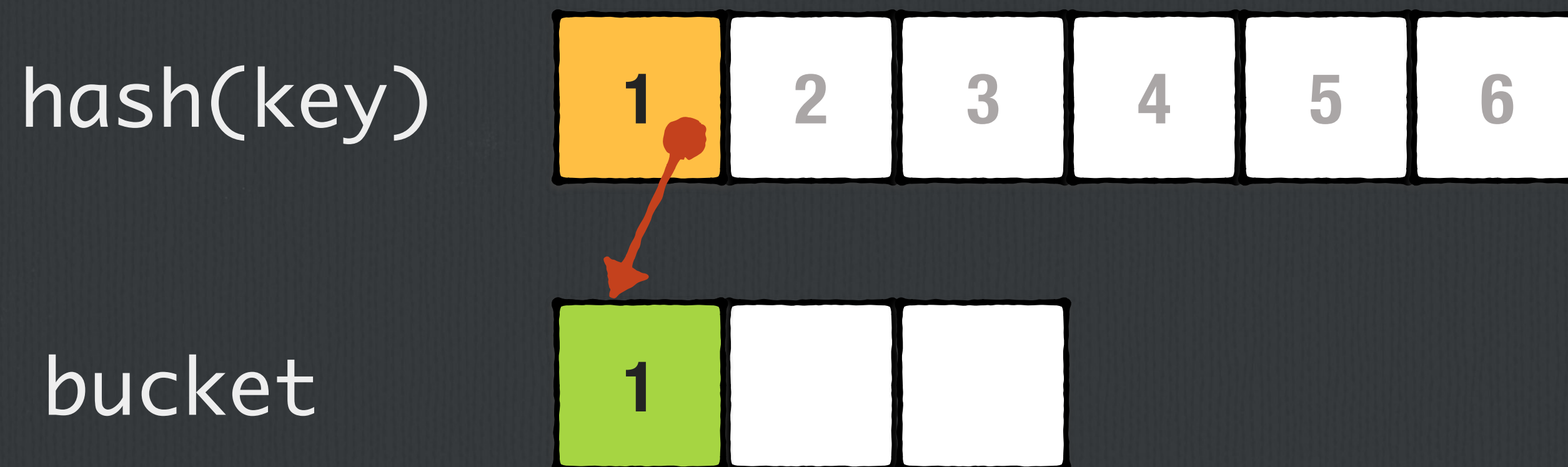
Closed-addressing containers

- ❑ fully compliant with the C++ specification for unordered associative containers
- ❑ “one of the fastest implementations in the market”



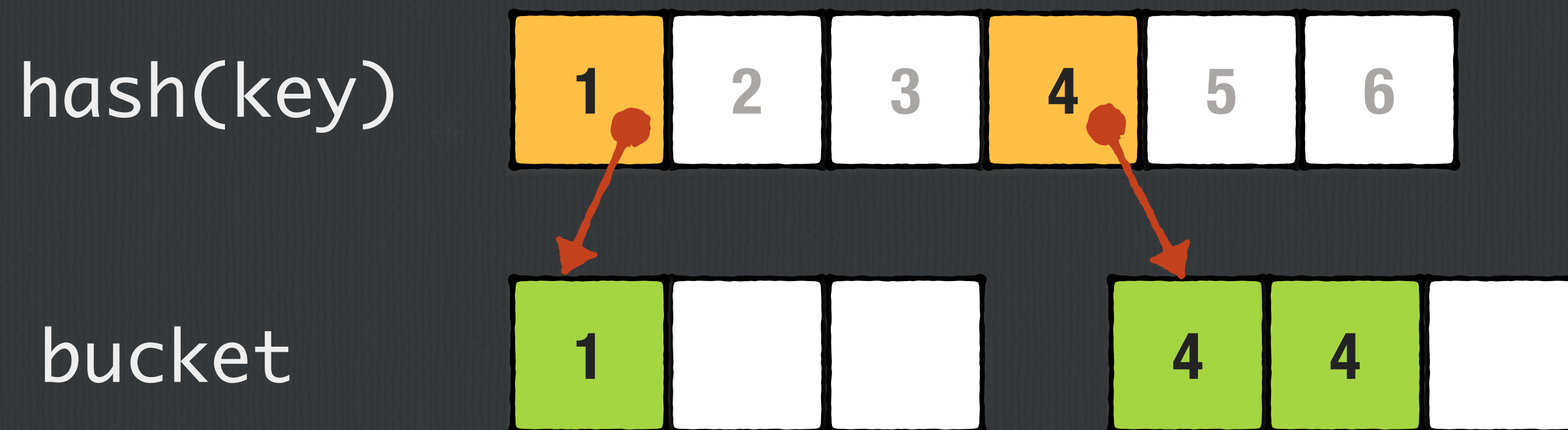
Closed-addressing containers

- ❑ fully compliant with the C++ specification for unordered associative containers
- ❑ “one of the fastest implementations in the market”



Closed-addressing containers

- ❑ fully compliant with the C++ specification for unordered associative containers
- ❑ “one of the fastest implementations in the market”

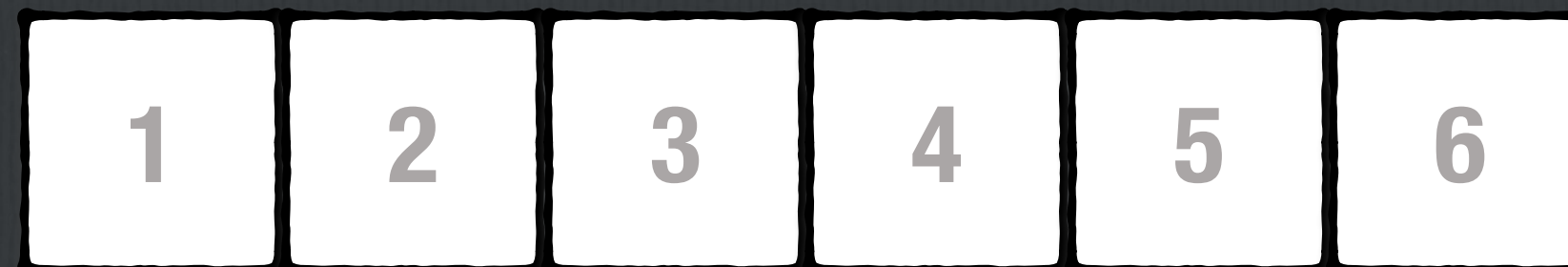


Open-addressing containers

- ☐ Can be (much) faster than closed-addressing containers
- ☐ Diverge from the “standard interface”
- ☐ Variants: flat (faster) vs. node-based (pointer-stable)

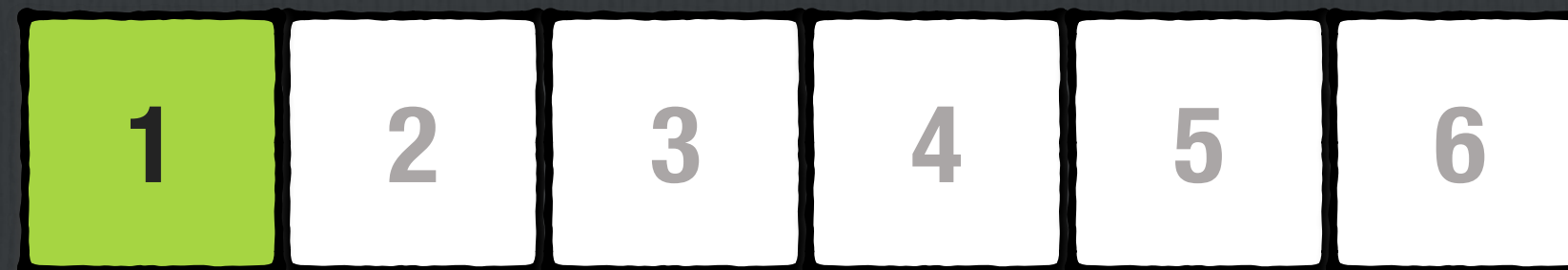
Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



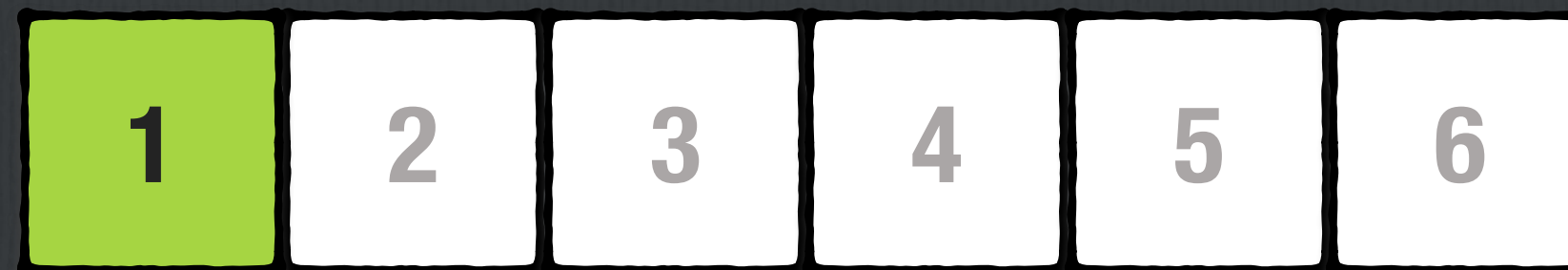
Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Open-addressing containers

- ❑ Can be (much) faster than closed-addressing containers
- ❑ Diverge from the “standard interface”
- ❑ Variants: flat (faster) vs. node-based (pointer-stable)



Concurrent container

Concurrent container

- `boost::concurrent_flat_map` is based on open-addressing

Concurrent container

- ❑ `boost::concurrent_flat_map` is based on open-addressing
- ❑ For high-performance, multi-threaded scenarios

Concurrent container

- ❑ `boost::concurrent_flat_map` is based on open-addressing
- ❑ For high-performance, multi-threaded scenarios
- ❑ Interface diverges from the standard

Concurrent container

- ☐ `boost::concurrent_flat_map` is based on open-addressing
- ☐ For high-performance, multi-threaded scenarios
- ☐ Interface diverges from the standard
- ☐ Based on a read-write mutex (container level) and read-write spin lock (group level)

Thank you!

(Benchmark your code - your mileage may vary)

