Chunked sequence

Deepsea project

3 September 2015

Contents

1 Introduction

This package provides a C++ template library that implements ordered, in-memory containers that are based on a B-tree-like data structure.

Like STL deque, our chunkedseq data structure supports fast constant-time update operations on the two ends of the sequence, and like balanced tree structures, such as STL rope, our chunkedseq structure supports efficient logarithmic-time split (at a specified position) and merge operations. However, unlike prior data structures, ours provides all of these operations simultaneously. Our research paper presents evidence to back these claims.

Key features of chunkedseq are:

- Fast constant-time push and pop operations on the two ends of the sequence.
- Logarithmic-time split at a specified position.
- Logarithmic-time concatenation.
- Familiar STL-style container interface.
- A *segment* abstraction to expose to clients of the chunked sequence the contiguous regions of memory that exist inside chunks.

1.1 Provided container types

- Double-ended queue
- Stack
- Bag
- Associative map

1.2 Advanced features

- Parallel processing
- Weighted container
- STL-style iterator
- Segments
- Derived data structures by cached measurement

1.3 Compatibility

This codebase makes extensive use of C++11 features, such as lambda expressions. Therefore, we recommend a recent version of GCC or Clang. We have tested the code on GCC v4.9.

1.4 Credits

The chunkedseq package is maintained by the members of the Deepsea Project. Primary authors include:

- Umut Acar
- Arthur Chargueraud
- Michael Rainey.

2 Double-ended queue

```
namespace pasl {
namespace data {
namespace chunkedseq {
namespace bootstrapped {

template <class Item>
class deque;

}}}}
```

The `deque` class implements a double-ended queue that, in addition to fast access to both ends, provides logarithmic-time operations for both weighted split and concatenation.

The deque interface implements much of the interface of the STL deque. All operations for accessing the front and back of the container (e.g., `front`, `push_front`, `pop_front`, etc.) are supported. Additionally, the deque supports splitting and concatenation in logarithmic time and provides a random-access iterator.

2.1 Template parameters

```
namespace pasl {
namespace data {
namespace chunkedseq {
namespace bootstrapped {

template <
  class Item,
  int Chunk_capacity = 512,
  class Cache = cachedmeasure::trivial<Item, size_t>,
  template <
    class Chunk_item,
    int Capacity,
    class Chunk_item_alloc=std::allocator<Item>
  >
  class Chunk_struct = fixedcapacity::heap_allocated::ringbuffer_ptrx,
  class Item_alloc = std::allocator<Item>
>
class deque;

}}}}
```

The signature above gives the complete list of the template parameters of the `deque` class and the table below the meanings of each one.

Table 1: Template parameters for the `deque` class (short version).

Template parameter	Description
`Item`	Type of the objects to be stored in the container
`Chunk_capacity`	Specifies capacity of chunks.
`Cache`	Specifies the policy by which to cache measurements on interior chunks.
`Chunk_struct`	Specifies the type of the chunks.
`Item_alloc`	Allocator to be used by the container to construct and destruct objects of type `Item`

2.1.1 Item type

```
class Item;
```

Type of the elements. Only if `Item` is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations. Aliased as member type `deque::value_type`.

2.1.2 Chunk capacity

```
int Chunk_capacity = 512;
```

The `Chunk_capacity` specifies the maximum number of items that can fit in each chunk.

Although each chunk can store *at most* `Chunk_capacity` items, the container can only guarantee that at most half of the cells of any given chunk are filled.

2.1.3 Cache type

```
class Cache = cachedmeasure::trivial<Item, size_t>;
```

The `Cache` type specifies the strategy to be used internally by the deque to maintain monoid-cached measurements of groups of items (see Cached measurement).

2.1.4 Chunk-struct type

```
template <
  class Chunk_item,
  int Capacity,
  class Chunk_item_alloc=std::allocator<Item>
>
class Chunk_struct = fixedcapacity::heap_allocated::ringbuffer_ptrx;
```

The `Chunk_struct` type specifies the fixed-capacity ring-buffer representation to be used for storing items (see Fixed-capacity buffers).

2.1.5 Allocator type

```
class Item_alloc = std::allocator<Item>;
```

Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent. Aliased as member type `deque::allocator_type`.

2.2 Member types

Table 2: Member types of the `deque` class.

Type	Description
`self_type`	Alias for the type of this container (e.g., `deque`, `stack`, `bag`)
`value_type`	Alias for template parameter `Item`
`reference`	Alias for `value_type&`
`const_reference`	Alias for `const value_type&`
`pointer`	Alias for `value_type*`
`const_pointer`	Alias for `const value_type*`
`size_type`	Alias for `size_t`
`segment_type`	Alias for `pasl::data::segment<pointer>`
`cache_type`	Alias for template parameter `Cache`
`measured_type`	Alias for `cache_type::measured_type`
`algebra_type`	Alias for `cache_type::algebra_type`
`measure_type`	Alias for `cache_type::measure_type`
`iterator`	Iterator
`const_iterator`	Const iterator

2.2.1 Iterator

The types `iterator` and `const_iterator` are instances of the random-access iterator concept. In addition to providing standard methods, our iterator provides the methods that are specified in the following table.

Table 3: Additional methods provided by the random-access iterator.

Method	Description
`size`	Returns the number of preceding items
`search_by`	Search to some position guided by a given predicate
`get_segment`	Returns the current segment

2.2.1.1 Iterator size

```
size_type size() const;
```

Returns the number of items preceding and including the item pointed to by the iterator.

Complexity. Constant time.

2.2.1.2 Search by predicate

```
template <class Predicate>
void search_by(const Predicate& p);
```

Moves the iterator to the first position `i` in the sequence for which the call `p(m_i)` returns `true`, where `m_i` denotes the accumulated cached measurement at position `i`.

Complexity. Logarithmic time.

2.2.1.3 Get enclosing segment

```
segment_type get_segment() const;
```

Returns the segment that encloses the iterator.

Complexity. Constant time.

2.3 Constructors and destructors

Table 4: Constructors of the `deque` class.

Constructor	Description
empty container constructor (default constructor)	constructs an empty container with no items
fill constructor	constructs a container with a specified number of copies of a given item
copy constructor	constructs a container with a copy of each of the items in the given container, in the same order
initializer list	constructs a container with the items specified in a given initializer list
move constructor	constructs a container that acquires the items of a given container
destructor	destructs a container

2.3.1 Empty container constructor

```
deque();
```

Complexity. Constant time.

Constructs an empty container with no items;

2.3.2 Fill container

```
deque(long n, const value_type& val);
```

Constructs a container with `n` copies of `val`.

Complexity. Time is linear in the size of the resulting container.

2.3.3 Copy constructor

```
deque(const deque& other);
```

Constructs a container with a copy of each of the items in `other`, in the same order.

Complexity. time is linear in the size of the resulting container.

2.3.4 Initializer-list constructor

```
deque(initializer_list<value_type> il);
```

Constructs a container with the items in `il`.

Complexity. Time is linear in the size of the resulting container.

2.3.5 Move constructor

```
deque(deque&& x);
```

Constructs a container that acquires the items of `other`.

Complexity. Constant time.

2.3.6 Destructor

```
~deque();
```

Destructs the container.

Complexity. Time is linear and logarithmic in the size of the container.

2.4 Item access

Table 5: Item accessors of the `deque` class.

Operation	Description

`front` `back`
 `operator[]`
	Access item on end.
	 Access member item

2.4.1 Front and back

```
value_type front() const;
value_type back() const;
```

Returns a reference to the last item in the container.

Calling this method on an empty container causes undefined behavior.

Complexity. Constant time.

2.4.2 Indexing operator

```
reference operator[](size_type i);
const_reference operator[](size_type i) const;
```

Returns a reference at the specified location `i`. No bounds check is performed.

Complexity. Logarithmic time.

2.5 Capacity

Table 6: Capacity methods of the `deque` class.

Operation	Description

`empty`
 `size`
	Checks whether the container is empty.
	 Returns the number of items.

2.5.1 Empty operator

```
bool empty() const;
```

Returns `true` if the container is empty, `false` otherwise.

Complexity. Constant time.

2.5.2 Size operator

```
size_type size() const;
```

Returns the size of the container.

Complexity. Constant time.

2.6 Iterators

Table 7: Iterators of the `deque` class.

Operation	Description
`front back`	Access item on end.
`operator[]`	Access member item
`begin cbegin`	Returns an iterator to the beginning
`end cend`	Returns an iterator to the end

2.6.1 Iterator begin

```
iterator begin() const;
const_iterator cbegin() const;
```

Returns an iterator to the first item of the container.

If the container is empty, the returned iterator will be equal to end().

Complexity. Constant time.

2.6.2 Iterator end

```
iterator end() const;
const_iterator cend() const;
```

Returns an iterator to the element following the last item of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.

Complexity. Constant time.

Table 8: Modifiers of the `deque` class.

Operation	Description
`push_front` `push_back`	Adds items to the end
`pop_front` `pop_back`	Removes items from the end
`split`	Splits off part of the container
`concat`	Merges contents of another container
`clear`	Erases contents
`resize`	Changes number of items stored
`swap`	Swaps contents

2.6.3 Push

```
void push_front(const value_type& value);
void push_back(const value_type& value);
```

Prepends the given element `value` to the beginning of the container.

Iterator validity. All iterators, including the past-the-end iterator, are invalidated. No references are invalidated.

Complexity. Constant time.

2.6.4 Pop

```
value_type pop_back();
value_type pop_front();
```

Removes the last element of the container and returns the element.

Calling `pop_back` or `pop_front` on an empty container is undefined.

Returns the removed element.

Complexity. Constant time.

2.6.5 Split

```
void split(iterator position, self_type& other);    // (1)
void split(size_type position, self_type& other);   // (2)
```

```
template <class Predicate>
void split(const Predicate& p, self_type& other);    // (3)
template <class Predicate>
void split(const Predicate& p,                       // (4)
           reference middle_item,
           self_type& other);
```

1. The container is erased after and including the item at the specified position.
2. The container is erased after and including the item at (zero-based) index `position`.
3. The container is erased after and including the item at the first position `i` for which `p(m_i)` returns `true`, where `m_i` denotes the accumulated cached measurement at position `i`.
4. The container is erased after the item at the first position `i` for which `p(m_i)` returns `true`, where `m_i` denotes the accumulated cached measurement at position `i`. The item at position `i` is also erased, but in this case, the item is copied into the reference `middle_item`.

The erased items are moved to the other container.

Precondition. The `other` container is empty.

Complexity. Time is logarithmic in the size of the container.

Iterator validity. Invalidates all iterators.

2.6.6 Concatenate

```
void concat(self_type other);
```

Removes all items from `other`, effectively reducing its size to zero.

Adds items removed from `other` to the back of this container, after its current last item.

Complexity. Time is logarithmic in the size of the container.

Iterator validity. Invalidates all iterators.

2.6.7 Clear

```
void clear();
```

Erases the contents of the container, which becomes an empty container.

Complexity. Time is linear in the size of the container.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

2.6.8 Resize

```
void resize(size_type n, const value_type& val); // (1)
void resize(size_type n) {                       // (2)
  value_type val;
  resize(n, val);
}
```

Resizes the container to contain `n` items.

If the current size is greater than `n`, the container is reduced to its first `n` elements.

If the current size is less than `n`,

1. additional copies of `val` are appended
2. additional default-inserted elements are appended

Complexity. Let m be the size of the container just before and n just after the resize operation. Then, the time is linear in $\max(m, n)$.

Iterator validity. Invalidates all iterators, if the size before the operation differs from the size after.

2.6.9 Exchange operation

```
void swap(deque& other);
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual items.

Complexity. Constant time.

2.7 Example: push and pop

```
#include <iostream>
#include <string>
#include <assert.h>
```

```
#include "chunkedseq.hpp"

int main(int argc, const char * argv[]) {

  using mydeque_type = pasl::data::chunkedseq::bootstrapped::deque<int>;

  const int nb = 5;

  mydeque_type mydeque;

  for (int i = 0; i < nb; i++)
    mydeque.push_back(i);
  for (int i = 0; i < nb; i++)
    mydeque.push_front(nb+i);

  assert(mydeque.size() == 2*nb);

  std::cout << "mydeque contains:";
  for (int i = 0; i < 2*nb; i++) {
    int v = (i % 2) ? mydeque.pop_front() : mydeque.pop_back();
    std::cout << " " << v;
  }
  std::cout << std::endl;

  assert(mydeque.empty());

  return 0;

}
```

source

Output

```
mydeque contains: 4 9 3 8 2 7 1 6 0 5
```

2.8 Example: split and concat

```
#include <iostream>
#include <string>
#include <assert.h>

#include "chunkedseq.hpp"

using mydeque_type = pasl::data::chunkedseq::bootstrapped::deque<int>;
```

```
static
void myprint(mydeque_type& mydeque) {
  auto it = mydeque.begin();
  while (it != mydeque.end())
    std::cout << " " << *it++;
  std::cout << std::endl;
}

int main(int argc, const char * argv[]) {

  mydeque_type mydeque = { 0, 1, 2, 3, 4, 5 };
  mydeque_type mydeque2;

  mydeque.split(size_t(3), mydeque2);

  mydeque.pop_back();
  mydeque.push_back(8888);

  mydeque2.pop_front();
  mydeque2.push_front(9999);

  std::cout << "Just after split:" << std::endl;
  std::cout << "contents of mydeque:";
  myprint(mydeque);
  std::cout << "contents of mydeque2:";
  myprint(mydeque2);

  mydeque.concat(mydeque2);

  std::cout << "Just after merge:" << std::endl;
  std::cout << "contents of mydeque:";
  myprint(mydeque);
  std::cout << "contents of mydeque2:";
  myprint(mydeque2);

  return 0;

}
```

source

Output

```
Just after split:
contents of mydeque: 0 1 8888
contents of mydeque2: 9999 4 5
Just after merge:
contents of mydeque: 0 1 8888 9999 4 5
contents of mydeque2:
```

3 Stack

```
namespace pasl {
namespace data {
namespace chunkedseq {
namespace bootstrapped {

template <class Item>
class stack;

}}}}
```

The stack is a container that supports the same set of operations as the deque, but has two key differences:

- Thanks to using a simpler stack structure to represent the chunks, the stack offers faster access to the back of the container and faster indexing operations than deque.
- Unlike deque, the stack cannot guarantee fast updates to the front of the container: each update operation performed on the front position can require at most `Chunk_capacity` items to be shifted toward to back.

3.1 Template interface

The complete template interface of the stack constructor is the same as that of the deque constructor, except that the chunk structure is not needed.

```
namespace pasl {
namespace data {
namespace chunkedseq {
namespace bootstrapped {

template <
  class Item,
  int Chunk_capacity = 512,
  class Cache = cachedmeasure::trivial<Item, size_t>,
  class Item_alloc = std::allocator<Item>
>
class stack;

}}}}
```

3.2 Example

```
#include <iostream>
#include <string>
#include <assert.h>

#include "chunkedseq.hpp"

int main(int argc, const char * argv[]) {

  using mystack_type = pasl::data::chunkedseq::bootstrapped::stack<int>;

  mystack_type mystack = { 0, 1, 2, 3, 4 };

  std::cout << "mystack contains:";
  while (! mystack.empty())
    std::cout << " " << mystack.pop_back();
  std::cout << std::endl;

  return 0;

}
```

source

Output

```
mystack contains: 4 3 2 1 0
```

4 Bag

```
namespace pasl {
namespace data {
namespace chunkedseq {
namespace bootstrapped {

template <class Item>
class bagopt;

}}}}
```

Our bag container is a generic container that trades the guarantee of order among its items for stronger guarantees on space usage and faster push and pop operations than the corresponding properties of the stack structure. In particular, the bag guarantees that there are no empty spaces in between consecutive items of the sequence, whereas stack and deque can guarantee only that no more than half of the cells of the chunks are empty.

Although our bag is unordered in general, in particular use cases, order among items is guaranteed. Order of insertion and removal of the items is guaranteed by the bag under any sequence of push or pop operations that affect the back of the container. The split and concatenation operations typically reorder items.

The container supports `front`, `push_front` and `pop_front` operations for the sole purpose of interface compatibility. These operations simply perform the corresponding actions on the back of the container.

4.1 Template interface

The complete template interface of the bag is similar to that of stack.

```
namespace pasl {
namespace data {
namespace chunkedseq {
namespace bootstrapped {

template <
  class Item,
  int Chunk_capacity = 512,
  class Cache = cachedmeasure::trivial<Item, size_t>,
  class Item_alloc = std::allocator<Item>
>
class bagopt;

}}}}
```

4.2 Example

```
#include <iostream>
#include <string>

#include "chunkedbag.hpp"

int main(int argc, const char * argv[]) {

  using mybag_type = pasl::data::chunkedseq::bootstrapped::bagopt<int>;
  mybag_type mybag = { 0, 1, 2, 3, 4 };

  std::cout << "mybag contains:";
  while (! mybag.empty())
    std::cout << " " << mybag.pop();
  std::cout << std::endl;

  return 0;

}
```

source

Output

```
mybag contains: 4 3 2 1 0
```

5 Associative map

```
namespace pasl {
namespace data {
namespace map {

template <class Key,
          class Item,
          class Compare = std::less<Key>,
          class Key_swap = std_swap<Key>,
          class Alloc = std::allocator<std::pair<const Key, Item> >,
          int chunk_capacity = 8
>
class map;

}}}
```

Using the cached-measurement feature of our chunked sequence structure, we have implemented asymptotically efficient associative maps in the style of STL map. Our implementation is, however, not designed to compete with highly optimized implementations, such as that of STL. Rather, the main purpose of our implementation is to provide an example of advanced use of cached measurement so that others can apply similar techniques to build their own custom data structures.

Our map interface implements only a subset of the STL interface. The operations that we do implement have the same time and space complexity as do the operations implemented by the STL container. However, the constant factors imposed by our container may be significantly larger than those of the STL container because our structure is not specifically optimized for this use case.

5.1 Example: `insert`

```
// accessing mapped values
#include <iostream>
#include <string>

#include "map.hpp"

int main () {
  pasl::data::map::map<char,std::string> mymap;

  mymap['a']="an element";
  mymap['b']="another element";
  mymap['c']=mymap['b'];

  std::cout << "mymap['a'] is " << mymap['a'] << '\n';
  std::cout << "mymap['b'] is " << mymap['b'] << '\n';
  std::cout << "mymap['c'] is " << mymap['c'] << '\n';
  std::cout << "mymap['d'] is " << mymap['d'] << '\n';

  std::cout << "mymap now contains " << mymap.size() << " elements.\n";

  return 0;
}
```

source

Output

```
mymap['a'] is an element
mymap['b'] is another element
mymap['c'] is another element
mymap['d'] is
mymap now contains 4 elements.
```

5.2 Example: `erase`

```
// accessing mapped values
#include <iostream>
#include <string>
```

```
#include "map.hpp"

int main ()
{
  pasl::data::map::map<char,int> mymap;
  pasl::data::map::map<char,int>::iterator it;

  // insert some values:
  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;
  mymap['d']=40;
  mymap['e']=50;
  mymap['f']=60;

  it=mymap.find('b');
  mymap.erase (it);                   // erasing by iterator

  mymap.erase ('c');                  // erasing by key

  // show content:
  for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << (*it).first << " => " << (*it).second << '\n';

  return 0;
}
```

source

Output

```
f => 60
e => 50
d => 40
a => 10
```

6 Parallel processing

The containers of the chunkedseq package are well suited to applications which use fork-join parallelism: thanks to the logarithmic-time split operations, chunkedseq containers can be divided efficiently, and thanks to the logarithmic-time concatenate operations, chunkedseq containers can be merged efficiently. Moreover, chunkedseq containers can be processed efficiently in a sequential fashion, thereby enabling a liberal programming style in which sequential and parallel processing styles are combined synergistically. The following example programs deomonstrate this style.

Remark:

> The data structures of the chunkedseq package are *not* concurrent data structures, or, put differently, chunkedseq data structures admit only single-threaded update operations.

Remark:

> The following examples are evidence that this single-threading restriction does *not* necessarily limit parallelism.

6.1 Example: `pkeep_if`

To see how our deque can be used for parallel processing, let us consider the following program, which constructs the subsequence of a given sequence, based on selections taken by a client-supplied predicate function. Assuming fork-join parallel constructs, such as Cilk's `spawn` and `sync`, the selection and build process of the `pkeep_if` function can achieve a large (in fact, unbounded) amount of parallelism thanks to the fact that the span of the computation is logarithmic in the size of the input sequence. Moreover, `pkeep_if` is *work efficient* thanks to the fact that the algorithm takes linear time in the size of the input sequence (assuming, of course, that the client-supplied predicate takes constant time).

```
#include <iostream>

#include "chunkedseq.hpp"

using cbdeque = pasl::data::chunkedseq::bootstrapped::deque<long>;

// moves items which satisfy a given predicate p from src to dst
// preserving original order of items in src
template <class Predicate_function>
void pkeep_if(cbdeque& dst, cbdeque& src, const Predicate_function& p) {

  const int cutoff = 8096;

  long sz = src.size();

  if (sz <= cutoff) {
    // compute result in a sequential fashion
    while (sz-- > 0) {
      long item = src.pop_back();
      if (p(item))
        dst.push_front(item);
    }

  } else {

    cbdeque src2;
    cbdeque dst2;

    // divide the input evenly in two halves
    size_t mid = sz / 2;
    src.split(mid, src2);

    // recurse on subproblems
    // calls can execute in parallel
    pkeep_if(dst,  src,  p);
    pkeep_if(dst2, src2, p);

    // combine results (after parallel calls complete)
    dst.concat(dst2);

  }
}

int main(int argc, const char * argv[]) {

  cbdeque src;
  cbdeque dst;

  const long n = 1000000;

  // fill the source container with [1, ..., 2n]
  for (long i = 1; i <= 2*n; i++)
    src.push_back(i);

  // leave src empty and dst = [1, 3, 5, ... n-1]
  pkeep_if(dst, src, [] (long x) { return x%2 == 1; });

  assert(src.empty());
  assert(dst.size() == n);

  // calculate the sum
  long sum = 0;
  while (! dst.empty())
    sum += dst.pop_front();

  // the sum of n consecutive odd integers starting from 1 equals n^2
  assert(sum == n*n);
  std::cout << "sum = " << sum << std::endl;

  return 0;

}
```

source

Output

```
sum = 1000000000000
```

6.2 Example: pcopy

This algorithm implements a parallel version of std::copy. Note, however, that the two versions differ slightly: in our version, the type of the destination parameter is a reference to the destination, whereas the corresponding type in std::copy is instead an iterator that points to the beginning of the destination container.

```
#include <iostream>
#include <string>
#include <assert.h>

#include "chunkedseq.hpp"

template <class Chunkedseq>
void pcopy(typename Chunkedseq::iterator first,
           typename Chunkedseq::iterator last,
           Chunkedseq& destination) {
  using iterator = typename Chunkedseq::iterator;
  using ptr = typename Chunkedseq::const_pointer;

  const long cutoff = 8192;

  long sz = last.size() - first.size();

  if (sz <= cutoff) {

    // compute result in a sequential fashion
    Chunkedseq::for_each_segment(first, last, [&] (ptr lo, ptr hi) {
      destination.pushn_back(lo, hi-lo);
    });

  } else {

    // select split position to be the median
    iterator mid = first + (sz/2);

    Chunkedseq destination2;

    // recurse on subproblems
    // calls can execute in parallel
    pcopy(first, mid,  destination);
    pcopy(mid,   last, destination2);

    // combine results
    destination.concat(destination2);

  }
}

int main(int argc, const char * argv[]) {

  const int chunk_size = 2;
  using mydeque_type = pasl::data::chunkedseq::bootstrapped::deque<int, chunk_size>;

  mydeque_type mydeque = { 0, 1, 2, 3, 4, 5 };
  mydeque_type mydeque2;

  pcopy(mydeque.begin(), mydeque.end(), mydeque2);

  std::cout << "mydeque2 contains:";
  auto p = mydeque2.begin();
  while (p != mydeque2.end())
    std::cout << " " << *p++;
  std::cout << std::endl;

  return 0;

}
```

source

Output

```
mydeque2 contains: 0 1 2 3 4 5
```

6.3 Example: `pcopy_if`

This algorithm implements a parallel version of std::copy_if. Just as before, our implementation uses a type for the third parameter that is different from the corresponding third parameter of the STL version.

```
#include <iostream>
#include <string>
#include <assert.h>

#include "chunkedseq.hpp"

template <class Chunkedseq, class UnaryPredicate>
void pcopy_if(typename Chunkedseq::iterator first,
              typename Chunkedseq::iterator last,
              Chunkedseq& destination,
              const UnaryPredicate& pred) {
  using iterator = typename Chunkedseq::iterator;
  using value_type = typename Chunkedseq::value_type;
  using ptr = typename Chunkedseq::const_pointer;

  const long cutoff = 8192;

  long sz = last.size() - first.size();

  if (sz <= cutoff) {

    // compute result in a sequential fashion
    Chunkedseq::for_each_segment(first, last, [&] (ptr lo, ptr hi) {
      for (ptr p = lo; p < hi; p++) {
        value_type v = *p;
        if (pred(v))
          destination.push_back(v);
      }
    });

  } else {

    // select split position to be the median
    iterator mid = first + (sz/2);

    Chunkedseq destination2;

    // recurse on subproblems
    // calls can execute in parallel
    pcopy_if(first, mid,  destination,  pred);
    pcopy_if(mid,   last, destination2, pred);

    destination.concat(destination2);
  }
}

int main(int argc, const char * argv[]) {

  const int chunk_size = 2;
  using mydeque_type = pasl::data::chunkedseq::bootstrapped::deque<int, chunk_size>;

  mydeque_type mydeque = { 0, 1, 2, 3, 4, 5 };
  mydeque_type mydeque2;

  pcopy_if(mydeque.begin(), mydeque.end(), mydeque2, [] (int i) { return i%2==0; });

  std::cout << "mydeque2 contains:";
  auto p = mydeque2.begin();
  while (p != mydeque2.end())
    std::cout << " " << *p++;
  std::cout << std::endl;

  return 0;

}
```

source

Output

```
mydeque2 contains: 0 2 4
```

7 Weighted container

The `chunkedseq` containers can easily generalize to *weighted containers*. A weighted container is a container that assigns to each item in the container an integral weight value. The weight value is typically expressed as a weight function that is defined by the client and passed to the container via template argument.

The purpose of the weight is to enable the client to use the weighted-split operation, which divides the container into two pieces by a specified weight. The split operation takes only logarithmic time.

7.1 Example: split sequence of strings by length

The following example program demonstrates how one can use weighted split to split a sequence of string values based on the number of even-length strings. In this case, our split divides the sequence into two pieces so that the first piece goes into `d` and the second to `f`. The split function specifies that `d` is to receive the first half of the original sequence of strings that together contain half of the total number of even-length strings in the original sequence; `f` is to receive the remaining strings. Because the lengths of the strings are cached internally by the weighted container, the split operation takes logarithmic time in the number of strings.

```
#include <iostream>
#include <string>

#include "chunkedseq.hpp"

namespace cachedmeasure = pasl::data::cachedmeasure;
namespace chunkedseq = pasl::data::chunkedseq::bootstrapped;

const int chunk_capacity = 512;

int main(int argc, const char * argv[]) {

  using value_type = std::string;
  using weight_type = int;

  class my_weight_fct {
  public:
    // returns 1 if the length of the string is an even number; 0 otherwise
    weight_type operator()(const value_type& str) const {
      return (str.size() % 2 == 0) ? 1 : 0;
    }
  };

  using my_cachedmeasure_type =
    cachedmeasure::weight<value_type, weight_type, size_t, my_weight_fct>;

  using my_weighted_deque_type =
    chunkedseq::deque<value_type, chunk_capacity, my_cachedmeasure_type>;

  my_weighted_deque_type d = { "Let's", "divide", "this", "sequence", "of",
                               "strings", "into", "two", "pieces" };

  weight_type nb_even_length_strings = d.get_cached();
  std::cout << "nb even-length strings: " << nb_even_length_strings << std::endl;
  my_weighted_deque_type f;

  d.split([=] (weight_type v) { return v >= nb_even_length_strings/2; }, f);

  std::cout << "d = " << std::endl;
  d.for_each([] (value_type& s) { std::cout << s << " "; });
  std::cout << std::endl;
  std::cout << std::endl;

  std::cout << "f = " << std::endl;
  f.for_each([] (value_type& s) { std::cout << s << " "; });
  std::cout << std::endl;

  return 0;

}
```

source

Output

```
nb even strings: 6
d =
Let's divide this

f =
sequence of strings into two pieces
```

8 STL-style iterator

Our deque, stack and bag containers implement the random-access iterators in the style of STL's random-access iterators.

8.1 Example

```
#include <iostream>
#include <string>
#include <assert.h>

#include "chunkedseq.hpp"

int main(int argc, const char * argv[]) {
  using mydeque_type = pasl::data::chunkedseq::bootstrapped::deque<int>;
  using iterator = typename mydeque_type::iterator;

  mydeque_type mydeque = { 0, 1, 2, 3, 4 };

  std::cout << "mydeque contains:";
  iterator it = mydeque.begin();
  while (it != mydeque.end())
    std::cout << " " << *it++;

  std::cout << std::endl;

  return 0;

}
```

source

Output

```
mydeque contains: 0 1 2 3 4
```

9 Segments

In this package, we use the term segment to refer to pointer values which reference a range in memory. We define two particular forms of segments:

- A *basic segment* is a value which consists of two pointers, namely `begin` and `end`, that define the right-open interval, `(begin,   end]`.
- An *enriched segment* is a value which consists of a basic segment, along with a pointer, namely `middle`, which points at some location in between `begin` and `end`, such that `begin <= middle < end`.

The following class defines a representation for enriched segments.

```
template <class Pointer>
class segment {
public:

  using pointer_type = Pointer;

  // points to the first cell of the interval
  pointer_type begin;
  // points to a cell contained in the interval
  pointer_type middle;
  // points to the cell that is one cell past the last cell of interval
  pointer_type end;

  segment()
  : begin(nullptr), middle(nullptr), end(nullptr) { }

  segment(pointer_type begin, pointer_type middle, pointer_type end)
  : begin(begin), middle(middle), end(end) { }

};
```

source

9.1 Example

```
#include <iostream>
#include <string>
#include <assert.h>

#include "chunkedseq.hpp"

int main(int argc, const char * argv[]) {

  const int chunk_size = 2;
  using mydeque_type = pasl::data::chunkedseq::bootstrapped::deque<int, chunk_size>;

  mydeque_type mydeque = { 0, 1, 2, 3, 4, 5 };

  std::cout << "mydeque contains:";
  // iterate over the segments in mydeque
  mydeque.for_each_segment([&] (int* begin, int* end) {
    // iterate over the items in the current segment
    int* p = begin;
    while (p != end)
      std::cout << " " << *p++;
  });
  std::cout << std::endl;

  using iterator = typename mydeque_type::iterator;
  using segment_type = typename mydeque_type::segment_type;

  // iterate over the items in the segment which contains the item at position 3
  iterator it = mydeque.begin() + 3;
  segment_type seg = it.get_segment();

  std::cout << "the segment which contains mydeque[3] contains:";
  int* p = seg.begin;
  while (p != seg.end)
    std::cout << " " << *p++;
  std::cout << std::endl;

  std::cout << "mydeque[3]=" << *seg.middle << std::endl;

  return 0;

}
```

source

Output

```
mydeque contains: 0 1 2 3 4 5
the segment which contains mydeque[3] contains: 2 3
mydeque[3]=3
```

10 Cached measurement

This documentation covers essential concepts that are needed to implement custom data structures out of various instantiations of the chunkedseq structure. Just like the Finger Tree of Hinze and Patterson, the chunkedseq can be instantiated in certain ways to yield asymptotically efficient data structures, such as associative maps, priority queues, weighted sequences, interval trees, etc. A summary of these ideas that is presented in greater detail can be find in the original publication on finger trees and in a blog post.

In this tutorial, we present the key mechanism for building derived data structures: *monoid-cached measurement*. We show how to use monoid-cached measurements to implement a powerful form of split operation that affects chunkedseq containers. Using this split operation, we then show how to apply our cached measurement scheme to build two new data structures:

- weighted containers with weighted splits
- asymptotically efficient associative map containers in the style of std::map

10.1 Taking measurements

Let S denote the type of the items contained by the chunkedseq container and T the type of the cached measurements. For example, suppose that we want to define a weighted chunkedseq container of `std::string`s for which the weights have type `weight_type`. Then we have: $S = \texttt{std} :: \texttt{string}$ and $T = \texttt{weight\_type}$. How exactly are cached measurements obtained? The following two methods are the ones that are used by our C++ package.

10.1.1 Measuring items individually

A *measure function* is a function m that is provided by the client; the function takes a single item and returns a single measure value: $m(s) : S \rightarrow T$.

10.1.2 Example: the "size" measure

Suppose we want to use our measurement to represent the number of items that are stored in the container. We call this measure the *size measure*. The measure of any individual item always equals one: $\texttt{size}(s) : S \rightarrow \texttt{long} = 1$.

10.1.3 Example: the "string-size" measure

The string-size measurement assigns to each item the weight equal to the number of characters in the given string: $\texttt{string\_size}(str) : \texttt{string} \rightarrow \texttt{long} = str.\texttt{size}()$.

10.1.4 Measuring items in contiguous regions of memory

Sometimes it is convenient to have the ability to compute, all at once, the combined measure of a group of items that is referenced by a given "basic" segment. For this reason, we require that, in addition to m, each measurement scheme provides a segment-wise measure operation, namely $\gtrdot$, which takes the pair of pointer arguments *begin* and *end* which correspond to a basic segment, and returns a single measured value: $\gtrdot(begin, end) : (S^*, S^*) \rightarrow T$.

The first and second arguments correspond to the range in memory defined by the segment $(begin, end]$. The value returned by $\gtrdot(begin, end)$ should equal the sum of the values $m(*p)$ for each pointer p in the range $(begin, end]$.

10.1.4.1 Example: segmented version of our size measurement

This operation is simply $\gtrdot(begin, end) = |end - begin|$, where our segment is defined by the sequence of items represented by the range of pointers $(begin, end]$.

10.1.5 The measure descriptor

The *measure descriptor* is the name that we give to the C++ class that describes a given measurement scheme. This interface exports deinitions of the following types:

Type	Description
`value_type`	type S of items stored in the container
`measured_type`	type T of item-measure values

And this interface exports definitions of the following methods:

Members	Description
`measured_type operator()(const value_type& v)`	returns $m($`v`$)$
`measured_type operator()(const value_type* begin, const value_type* end)`	returns $\gtrdot($`begin`$,$ `enc`

10.1.5.1 Example: trivial measurement

Our first kind of measurement is one that does nothing except make fresh values whose type is the same as the type of the second template argument of the class.

```
template <class Item, class Measured>
class trivial {
public:

  using value_type = Item;
  using measured_type = Measured;

  measured_type operator()(const value_type& v) const {
    return measured_type();
  }

  measured_type operator()(const value_type* lo, const value_type* hi) const {
    return measured_type();
  }

};
```

source

The trivial measurement is useful in situations where cached measurements are not needed by the client of the chunkedseq. Trivial measurements have the advantage of being (almost) zero overhead annotations.

10.1.6 Example: weight-one (uniformly sized) items

This kind of measurement is useful for maintaining fast access to the count of the number of items stored in the container.

```
template <class Item, class Measured, int Item_weight=1>
class uniform {
public:

  using value_type = Item;
  using measured_type = Measured;
  const int item_weight = Item_weight;

  measured_type operator()(const value_type& v) const {
    return measured_type(item_weight);
  }

  measured_type operator()(const value_type* lo, const value_type* hi) const {
    return measured_type(hi - lo);
  }
};
```

source

10.1.6.1 Example: dynamically weighted items

This technique allows the client to supply to the internals of the chunkedseq container an arbitrary weight function. This client-supplied weight function is passed to the following class by the third template argument.

```
template <class Item, class Weight, class Client_weight_fct>
class weight {
public:

  using value_type = Item;
  using measured_type = Weight;
  using client_weight_fct_type = Client_weight_fct;

private:

  client_weight_fct_type client_weight_fct;

  // for debugging purposes
  bool initialized;
public:

  weight() : initialized(false) { }

  weight(const client_weight_fct_type& env)
  : client_weight_fct(env), initialized(true) { }

  measured_type operator()(const value_type& v) const {
    return client_weight_fct(v);
  }

  measured_type operator()(const value_type* lo, const value_type* hi) const {
    measured_type m = 0;
    for (auto p = lo; p < hi; p++)
      m += operator()(*p);
    return m;
  }

  client_weight_fct_type get_env() const {
    assert(initialized);
    return client_weight_fct;
  }

  void set_env(client_weight_fct_type wf) {
    client_weight_fct = wf;
    initialized = true;
  }

};
```

source

10.1.6.2 Example: combining cached measurements

Often it is useful to combine meaurements in various configurations. For this purpose, we define the measured pair, which is just a structure that has space for two values of two given measured types, namely `Measured1` and `Measured2`.

```
template <class Measured1, class Measured2>
class measured_pair {
public:
  Measured1 value1;
  Measured2 value2;
  measured_pair() { }
  measured_pair(const Measured1& value1, const Measured2& value2)
  : value1(value1), value2(value2) { }
};

template <class Measured1, class Measured2>
measured_pair<Measured1,Measured2> make_measured_pair(Measured1 m1, Measured2 m2) {
  measured_pair<Measured1,Measured2> m(m1, m2);
  return m;
}
```

source

The combiner measurement just combines the measurement strategies of two given measures by pairing measured values.

```
template <class Item, class Measure1, class Measure2>
class combiner {
public:

  using measure1_type = Measure1;
  using measure2_type = Measure2;

  using value_type = Item;
  using measured_type = measured_pair<measure1_type, measure2_type>;

  measure1_type meas1;
  measure2_type meas2;

  combiner() { }

  combiner(const measure1_type meas1)
  : meas1(meas1) { }

  combiner(const measure2_type meas2)
  : meas2(meas2) { }

  combiner(const measure1_type meas1, const measure2_type meas2)
  : meas1(meas1), meas2(meas2) { }

  measured_type operator()(const value_type& v) const {
    return make_measured_pair(meas1(v), meas2(v));
  }

  measured_type operator()(const value_type* lo, const value_type* hi) const {
    return make_measured_pair(meas1(lo, hi), meas2(lo, hi));
  }
};
```

source

10.2 Using algebras to combine measurements

Recall that a *monoid* is an algebraic structure that consists of a set T, an associative binary operation $\oplus$ and an identity element $\mathbf{I}$. That is, $(T, \oplus, \mathbf{I})$ is a monoid if:

- $\oplus$ is associative: for every x, y and z in T, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.
- $\mathbf{I}$ is the identity for $\oplus$: for every x in T, $x \oplus \mathbf{I} = \mathbf{I} \oplus x$.

Examples of monoids include the following:

- T = the set of all integers; $\oplus$ = addition; $\mathbf{I} = 0$
- T = the set of 32-bit unsigned integers; $\oplus$ = addition modulo 2^{32}; $\mathbf{I} = 0$
- T = the set of all strings; $\oplus$ = concatenation; $\mathbf{I}$ = the empty string

A *group* is a closely related algebraic structure. Any monoid is also a group if the monoid has an inverse operation $\ominus$:

- $\ominus$ is inverse for $\oplus$: for every x in T, there is an item $y = \ominus x$ in T, such that $x \oplus y = \mathbf{I}$.

10.2.1 The algebra descriptor

We require that the descriptor export a binding to the type of the measured values that are related by the algebra.

Type	Description
`value_type`	type of measured values T to be related by the algebra

We require that the descriptor export the following members. If `has_inverse` is false, then it should be safe to assume that the `inverse(x)` operation is never called.

Static members	Description
const bool has_inverse	`true`, iff the algebra is a group

Static members	Description
value_type identity()	returns **I**
value_type combine(value_type x, value_type y)	returns `x` $\oplus$ `y`
value_type inverse(value_type x)	returns $\ominus$ `x`

10.2.1.1 Example: trivial algebra

The trivial algebra does nothing except construct new identity elements.

```
class trivial {
public:

  using value_type = struct { };

  static constexpr bool has_inverse = true;

  static value_type identity() {
    return value_type();
  }

  static value_type combine(value_type, value_type) {
    return identity();
  }

  static value_type inverse(value_type) {
    return identity();
  }

};
```

source

10.2.1.2 Example: algebra for integers

The algebra that we use for integers is a group in which the identity element is zero, the plus operator is integer addition, and the minus operator is integer negation.

```
template <class Int>
class int_group_under_addition_and_negation {
public:

  using value_type = Int;
  static constexpr bool has_inverse = true;

  static value_type identity() {
    return value_type(0);
  }

  static value_type combine(value_type x, value_type y) {
    return x + y;
  }

  static value_type inverse(value_type x) {
    return value_type(-1) * x;
  }

};
```

source

10.2.2 Example: combining algebras

Just like with the measurement descriptor, an algebra descriptor can be created by combining two given algebra descriptors pairwise.

```
template <class Algebra1, class Algebra2>
class combiner {
public:

  using algebra1_type = Algebra1;
  using algebra2_type = Algebra2;

  using value1_type = typename Algebra1::value_type;
  using value2_type = typename Algebra2::value_type;

  using value_type = measure::measured_pair<value1_type, value2_type>;

  static constexpr bool has_inverse =
                   algebra1_type::has_inverse
                && algebra2_type::has_inverse;

  static value_type identity() {
    return measure::make_measured_pair(algebra1_type::identity(),
                                       algebra2_type::identity());
  }

  static value_type combine(value_type x, value_type y) {
    return measure::make_measured_pair(algebra1_type::combine(x.value1, y.value1),
                                       algebra2_type::combine(x.value2, y.value2));
  }

  static value_type inverse(value_type x) {
    return measure::make_measured_pair(algebra1_type::inverse(x.value1),
                                       algebra2_type::inverse(x.value2));
  }

};
```

source

10.2.3 Scans

A *scan* is an iterated reduction that maps to each item v_i in a given sequences of items $S = [v_1, v_2, \ldots, v_n]$ a single measured value $c_i = \mathbf{I} \oplus m(v_1) \oplus m(v_2) \oplus \ldots \oplus m(v_i)$, where $m(v)$ is a given measure function. For example, consider the "size" (i.e., weight-one) scan, which is specified by the use of a particular measure function: $m(v) = 1$. Observe that the size scan gives the positions of the items in the sequence, thereby enabling us later on to index and to split the chunkedseq at a given position.

For convenience, we define scan formally as follows. The operator returns the combined measured values of the items in the range of positions $[i, j)$ in the given sequence s.

$M_{i,j} : \texttt{Sequence}(S) \to T$

$M_{i,i}(s) = \mathbf{I}$

$M_{i,j}(s) = m(s_i) \oplus m(s_{i+1}) \oplus \ldots \oplus m(s_j)$ if $i < j$

10.2.4 Why associativity is necessary

The cached value of an internal tree node k in the chunkedseq structure is computed by $M_{i,j}(s)$, where $s = [v_i, \ldots, v_j]$ represents a subsequence of values contained in the chunks of the subtree below node k. When this reduction is performed by the internal operations of the chunkedseq, this expression is broken up into a set of subexpressions, for example: $((m(v_i) \oplus m(v_{i+1})) \oplus (m(v_{i+2}) \oplus m(v_{i+3}) \oplus (m(v_{i+4}) \oplus m(v_{i+5})))... \oplus m(v_j))$. The partitioning into subexpressions and the order in which the subexpressions are combined depends on the particular shape of the underlying chunkedseq. Moreover, the particular shape is determined uniquely by the history of update operations that created the finger tree. As such, we could build two chunkedseqs by, for example, using different sequences of push and pop operations and end up with two different chunkedseq structures that represent the same sequence of items. Even though the two chunkedseqs represent the same sequence, the cached measurements of the two chunkedseqs are combined up to the root of the chunkedseq by two different partitionings of combining operations. However, if $\oplus$ is associative, it does not matter: regardless of how the expression are broken up, the cached measurement at the root of the chunkedseq is guaranteed to be the same for any two chunkedseqs that represent the same sequence of items. Commutativity is not necessary, however, because the ordering of the items of the sequence is respected by the combining operations performed by the chunkedseq.

10.2.5 Why the inverse operation can improve performance

Suppose we have a cached measurement $C = M_{i,j}(s)$, where $s = [v_i, \ldots, v_j]$ represents a subsequence of values contained in the same chunk somewhere inside our chunkedseq structure. Now, suppose that we wish to remove the first item from our sequence of measurements, namely v_i. On the one hand, without an inverse operation, and assuming that we have not cached partial sums of C, the only way to compute the new cached value is to recompute $(m(v_{i+1}) \oplus ... \oplus m(v_j))$. On the other hand, if the inverse operation is cheap, it may be much more efficient to instead compute $\ominus m(v_i) \oplus C$.

Therefore, it should be clear that using the inverse operation can greatly improve efficiency in situations where the combined cached measurement of a group of items needs to be recomputed on a regular basis. For example, the same situation is triggered by the pop operations of the chunks stored inside the chunkedseq structure. On the one hand, by using inverse, each pop operation requires only a few additional operations to reset the cached measured value of the chunk. On the other, if inverse is not available, each pop operation requires recomputing the combined measure of the chunk, which although constant time, takes time proportion with the chunk size, which can be a fairly large fixed constant, such as 512 items. As such, internally, the chunkedseq operations use inverse operations whenever permitted by the algebra (i.e., when the algebra is identified as a group) but otherwise fall back to the most general strategy when the algebra is just a monoid.

10.3 Defining custom cached-measurement policies

The cached-measurement policy binds both the measurement scheme and the algebra for a given instantiation of chunkedseq. For example, the following are cached-measurement policies:

- nullary cached measurement: $m(s) = \emptyset$; $\gtrdot(v) = \emptyset$; $A_T = (\mathcal{P}(\emptyset), \cup, \emptyset, \ominus)$, where $\ominus\emptyset = \emptyset$
- size cached measurement: $m(s) = 1$; $\gtrdot(v) = |v|$; $A_T = (\texttt{long}, +, 0, \ominus)$

- pairing policies (monoid): for any two cached-measurement policies m_1; $\gtrdot_{\nVdash}$; $A_{T_1} = (T_1, \oplus_1, \mathtt{I}_1)$ and m_2; $\gtrdot_{\nvDash}$; $A_{T_2} = (T_2, \oplus_2, \mathtt{I}_2)$, $m(s_1, s_2) = (m_1(s_1), m_2(s_2))$; $\gtrdot(v_1, v_2) = (\gtrdot_{\nVdash}(v_1), \gtrdot_{\nvDash}(v_2))$; $A = (T_1 \times T_2, \oplus, (\mathtt{I}_1, \mathtt{I}_2))$ is also a cached-measurement policy, where $(x_1, x_2) \oplus (y_1, y_2) = (x_1 \oplus y_1, x_2 \oplus y_2)$
- pairing policies (group): for any two cached-measurement policies m_1; $\gtrdot_{\nVdash}$; $A_{T_1} = (T_1, \oplus_1, \mathtt{I}_1, \ominus_1)$ and m_2; $\gtrdot_{\nvDash}$; $A_{T_2} = (T_2, \oplus_2, \mathtt{I}_2, \ominus_2)$, $m(s_1, s_2) = (m_1(s_1), m_2(s_2))$; $\gtrdot(v_1, v_2) = (\gtrdot_{\nVdash}(v_1), \gtrdot_{\nvDash}(v_2))$; $A = (T_1 \times T_2, \oplus, (\mathtt{I}_1, \mathtt{I}_2), \ominus)$ is also a cached-measurement policy, where $(x_1, x_2) \oplus (y_1, y_2) = (x_1 \oplus y_1, x_2 \oplus y_2)$ and $\ominus(x_1, x_2) = (\ominus_1 x_1, \ominus_2 x_2)$
- pairing policies (mixed): if only one of two given cached-measurement policies is a group, we demote the group to a monoid and apply the pairing policy for two monoids

Remark:

> To save space, the chunkedseq structure can be instantiated with the nullary cached measurement alone. No space is taken by the cached measurements in this configuration because the nullary measurement takes zero bytes. However, the only operations supported in this configuration are push, pop, and concatenate. The size cached measurement is required by the indexing and split operations. The various instantiations of chunkedseq, namely deque, stack and bag all use the size measure for exactly this reason.

10.3.1 The cached-measurement descriptor

The interface exports four key components: the type of the items in the container, the type of the measured values, the measure function to gather the measurements, and the algebra to combine measured values.

Type	Description
`measure_type`	type of the measure descriptor
`algebra_type`	type of the algebra descriptor
`value_type`	type S of items to be stored in the container
`measured_type`	type T of measured values
`size_type`	`size_t`

The only additional function that is required by the policy is a swap operation.

Static members	Description
`void swap(measured_type& x, measured_type& y)`	exchanges the values of `x` and `y`

10.3.2 Example: trivial cached measurement

This trivial cached measurement is, by itself, completely inert: no computation is required to maintain cached values and only a minimum of space is required to store cached measurements on internal tree nodes of the chunkedseq.

```
template <class Item, class Size>
class trivial {
public:

  using size_type = Size;
  using value_type = Item;
  using algebra_type = algebra::trivial;
  using measured_type = typename algebra_type::value_type;
  using measure_type = measure::trivial<value_type, measured_type>;

  static void swap(measured_type& x, measured_type& y) {
    // nothing to do
  }

};
```

source

10.3.3 Example: weight-one (uniformly sized) items

In our implementation, we use this cached measurement policy to maintain the size information of the container. The `size()` methods of the different chunkedseq containers obtain the size information by referencing values cached inside the tree by this policy.

```
template <class Item, class Size>
class size {
public:

  using size_type = Size;
  using value_type = Item;
  using algebra_type = algebra::int_group_under_addition_and_negation<size_type>;
  using measured_type = typename algebra_type::value_type;
  using measure_type = measure::uniform<value_type, measured_type>;

  static void swap(measured_type& x, measured_type& y) {
    std::swap(x, y);
  }
};
```

source

10.3.4 Example: weighted items

Arbitrary weights can be maintained using a slight generalization of the `size` measurement above.

```
template <class Item, class Weight, class Size, class Measure_environment>
class weight {
public:

  using size_type = Size;
  using value_type = Item;
  using algebra_type = algebra::int_group_under_addition_and_negation<Weight>;
  using measured_type = typename algebra_type::value_type; // = Weight
  using measure_env_type = Measure_environment;
  using measure_type = measure::weight<value_type, measured_type, measure_env_type>;

  static void swap(measured_type& x, measured_type& y) {
    std::swap(x, y);
  }

};
```

source

10.3.5 Example: combining cached measurements

Using the same combiner pattern we alredy presented for measures and algebras, we can use the following template class to build combinations of any two given cached-measurement policies.

```
template <class Cache1, class Cache2>
class combiner {
public:

  using algebra1_type = typename Cache1::algebra_type;
  using algebra2_type = typename Cache2::algebra_type;
  using measure1_type = typename Cache1::measure_type;
  using measure2_type = typename Cache2::measure_type;
  using size_type = typename Cache1::size_type;
  using value_type = typename Cache1::value_type;
  using algebra_type = algebra::combiner<algebra1_type, algebra2_type>;
  using measured_type = typename algebra_type::value_type;
  using measure_type = measure::combiner<value_type, measure1_type, measure2_type>;

  static void swap(measured_type& x, measured_type& y) {
    Cache1::swap(x.value1, y.value1);
    Cache2::swap(x.value2, y.value2);
  }

};
```

source

10.4 Splitting by predicate functions

Logically, the split operation on a chunkedseq container divides the underlying sequence into two pieces, leaving the first piece in the container targeted by the split and moving the other piece to another given container. The position at which the split occurs is determined by a search process that is guided by a *predicate function*. What carries out the search process? That job is the job of the internals of the chunkedseq class; the client is responsible only to provide the predicate function that is used by the search process. Formally, a predicate function is simply a function p which takes a measured value and returns either `true` or `false`: $p(m) : T \rightarrow \texttt{bool}$.

The search process guarantees that the position at which the split occurs is the position i in the target sequence, $s = [v_1, \ldots, v_i, \ldots v_n]$, at which the value returned by $p(M_{0,i}(s))$ first switches from false to true. The first part of the split equals $[v_1, \ldots, v_{i-1}]$ and the second $[v_i, \ldots, v_n]$.

10.4.1 The predicate function descriptor

In our C++ package, we represent predicate functions as classes which export the following public method.

Members	Description
`bool operator()(measured_type m)`	returns $p(\texttt{m})$

10.4.2 Example: weighted splits

Let us first consider the small example which is given already for the weighted container. The action performed by the example program is to divide a given sequence of strings so that the first piece of the split contains approximately half of the even-length strings and the second piece the second half. In our example code (see the page linked above), we assign to each item a certain weight as follows: if the length of the given string is an even number, return a 1; else, return a 0.

$m(str) : \texttt{string} \rightarrow \texttt{int} = 1$ if $str.\texttt{size}()$ is an even number and 0 otherwise

Let n denote the number of even-length strings in our source sequence. Then, the following predicate function delivers the exact split that we want: $p(m) : int \rightarrow \texttt{bool} = m \geq n/2$. Let s denote the sequence of strings (i.e., `["Let's", "divide", "this", "string", "into", "two", "pieces"]` that we want to split. The following table shows the logical states of the split process.

i	0	1	2	3	4	5	6
v_i	`Let's`	`divide`	`this`	`string`	`into`	`two`	`pieces`
$m(v_i)$	0	1	1	1	1	0	1
$M_{0,i}(s)$	0	1	2	3	4	4	5
$p(M_{0,i}(s))$	`false`	`false`	`false`	`true`	`true`	`true`	`true`

Remark:

> Even though the search process might look like a linear search, the process in fact takes just logarithmic time in the number of items in the sequence. The logarithmic time bound is possible thanks to the fact that internal nodes of the chunkedseq tree (which is itself a tree whose height is logarithmic in the number of items) are annotated by partial sums of weights.

10.5 Example: using cached measurement to implement associative maps

Our final example combines all of the elements of cached measurement to yield an asymptotically efficient implementation of associative maps. The idea behind the implementation is to represent the map internally by a chunkedseq container of key-value pairs. The key to efficiency is that the items in the chunkedseq are stored in descending order. When key-value pairs are logically added to the map, the key-value pair is physically added to a position in the underlying sequence so that the descending order is maintained. The insertion and removal of key-value pairs is achieved by splitting by a certain predicate function which we will see later. At a high level, what is happening is a kind of binary search that navigates the underlying chunkedseq structure, guided by carefully chosen cached key values that annotate the interior nodes of the chunkedseq.

Remark:

> We could have just as well maintain keys in ascending order.

10.5.1 Optional values

Our implementation uses *optional values*, which are values that logically either contain a value of a given type or contain nothing at all. The concept is similar to that of the null pointer, except that the optional value applies to any given type, not just pointers.

```
template <class Item, class Item_swap>
class option {
public:

  using self_type = option<Item, Item_swap>;

  Item item;
  bool no_item;

  option()
  : item(), no_item(true) { }

  option(const Item& item)
  : item(item), no_item(false) { }

  option(const option& other)
  : item(other.item), no_item(other.no_item) { }

  void swap(option& other) {
    Item_swap::swap(item, other.item);
    std::swap(no_item, other.no_item);
  }

  bool operator<(const self_type& y) const {
    if (no_item && y.no_item)
      return false;
    if (no_item)
      return true;
    if (y.no_item)
      return false;
    return item < y.item;
  }

  bool operator>=(const self_type& y) const {
    return ! (*this < y);
  }

};
```

source

Observe that our class implements the “less-than” operator: `<`. Our implementation of this operator lifts any implementation of the same operator at the type `Item` to the space of our `option<Item>`: that is, our operator treats the empty (i.e., nullary) optional value as the smallest optional value. Otherwise, our the comparison used by our operator is the implementation already defined for the given type, `Item`, if such an implementation is available.

10.5.2 The measure descriptor

The type of value returned by the measure function (i.e., `measured_type`) is the optional key value, that is, a value of type `option<key_type>`. The measure function simply extracts the smallest key value from the key-value pairs that it has at hand and packages them as an optional key value.

```
template <class Item, class Measured>
class get_key_of_last_item {
public:

  using value_type = Item;
  using key_type = typename value_type::first_type;
  using measured_type = Measured;

  measured_type operator()(const value_type& v) const {
    key_type key = v.first;
    return measured_type(key);
  }

  measured_type operator()(const value_type* lo, const value_type* hi) const {
    if (hi - lo == 0)
      return measured_type();
    const value_type* last = hi - 1;
    key_type key = last->first;
    return measured_type(key);
  }
};
```

source

10.5.3 The monoid descriptor

The monoid uses for its identity element the nullary optional key. The combining operator takes two optional key values and of the two returns either the smallest one or the nullary optional key value.

```
template <class Option>
class take_right_if_nonempty {
public:

  using value_type = Option;

  static constexpr bool has_inverse = false;

  static value_type identity() {
    return value_type();
  }

  static value_type combine(value_type left, value_type right) {
    if (right.no_item)
      return left;
    return right;
  }

  static value_type inverse(value_type x) {
    // cannot happen
    return identity();
  }

};
```

source

10.5.4 The descriptor of the cached measurement policy

The cache measurement policy combines the measurement and monoid descriptors in a straightforward fashion.

```
template <class Item, class Size, class Key_swap>
class map_cache {
public:

  using size_type = Size;
  using value_type = Item;
  using key_type = typename value_type::first_type;
  using option_type = option<key_type, Key_swap>;
  using algebra_type = take_right_if_nonempty<option_type>;
  using measured_type = typename algebra_type::value_type; // = option_type
  using measure_type = get_key_of_last_item<value_type, measured_type>;

  static void swap(measured_type& x, measured_type& y) {
    x.swap(y);
  }

};
```

source

10.5.5 The associative map

The associative map class maintains the underlying sorted sequence of key-value pairs in the field called `seq`. The method called `upper` is the method that is employed by the class to maintain the invariant on the descending order of the keys. This method returns either the position of the first key that is greater than the given key, or the position of one past the end of the sequence if the given key is the greatest key.

As is typical of STL style, the indexing operator is used by the structure to handle both insertions and lookups. The operator works by first searching in its underlying sequence for the key referenced by its parameter; if found, the operator updates the value component of the corresponding key-value pair. Otherwise, the operator creates a new position in the sequence to put the given key by calling the `insert` method of `seq` at the appropriate position.

```
template <class Item>
class std_swap {
public:

  static void swap(Item& x, Item& y) {
    std::swap(x, y);
  }

};
```

source

```
template <class Key,
          class Item,
          class Compare = std::less<Key>,
          class Key_swap = std_swap<Key>,
          class Alloc = std::allocator<std::pair<const Key, Item> >,
          int chunk_capacity = 8
          >
class map {
public:

  using key_type = Key;
  using mapped_type = Item;
  using value_type = std::pair<key_type, mapped_type>;
  using key_compare = Compare;
  using allocator_type = Alloc;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = value_type*;
  using const_pointer = const value_type*;
  using difference_type = ptrdiff_t;
  using size_type = size_t;
  using key_swap_type = Key_swap;

private:

  using cache_type = map_cache<value_type, size_type, key_swap_type>;
  using container_type = chunkedseq::bootstrapped::deque<value_type, chunk_capacity, cache_t
  using option_type = typename cache_type::measured_type;

public:

  using iterator = typename container_type::iterator;

private:

  // invariant: items in seq are sorted in ascending order by their key values
  mutable container_type seq;
  mutable iterator it;

  iterator upper(const key_type& k) const {
    option_type target(k);
    it.search_by([target] (const option_type& key) {
      return target >= key;
    });
    return it;
  }

public:

  map() {
    it = seq.begin();
  }

  map(const map& other)
  : seq(other.seq) {
    it = seq.begin();
  }

  size_type size() const {
    return seq.size();
  }

  bool empty() const {
    return size() == 0;
  }

  iterator find(const key_type& k) const {
    iterator it = upper(k);
    return ((*it).first == k) ? it : seq.end();
  }

  mapped_type& operator[](const key_type& k) const {
    it = upper(k);
    if (it == seq.end()) {
      // key k is larger than any key current in seq
      value_type val;
      val.first = k;
      seq.push_back(val);
      it = seq.end()-1;
    } else if ((*it).first != k) {
      // iterator it points to the first key that is less than k
      value_type val;
      val.first = k;
      it = seq.insert(it, val);
    }
    return (*it).second;
  }

  void erase(iterator it) {
    if (it == seq.end())
```

```
      return;
    if (it == seq.end()-1) {
      seq.pop_front();
      return;
    }
    seq.erase(it, it+1);
  }

  size_type erase(const key_type& k) {
    size_type nb = seq.size();
    erase(find(k));
    return nb - seq.size();
  }

  std::ostream& stream(std::ostream& out) const {
    out << "[";
    size_type sz = size();
    seq.for_each([&] (value_type v) {
      out << "(" << v.first << "," << v.second << ")";
      if (sz-- != 1)
        out << ",";
    });
    return out << "]";
  }

  iterator begin() const {
    return seq.begin();
  }

  iterator end() const {
    return seq.end();
  }

  void check() const {
    seq.check();
  }

};
```

source