Standard Containers Library (1)

C++ containers are objects that store other objects – data structures.

- controls allocation and deallocation of the stored objects through constructors, destructors, insert and erase operations
- physical implementation is not defined by the standard, instead
  - complexity requirements for each operation is stated, e.g. compile time, constant, linear, …
  - elements are ordered in some containers
- the type of objects stored in standard containers must meet some general requirements, e.g.
  - *constructible* – have copy constructor and destructor
  - *moveable* – have move constructor
  - *assignable* – have copy assignment operator
- especially efficient operations are typically applied directly at some specific, implicit position of the container, e.g.
  - `vector` has `push_back(x)` and `pop_back()`, `emplace_back(x)`
  - `deque` and `list` have also `push_front(x)` and `pop_front(), emplace_front(x)` and `emplace_back(x)`
- other operations typically take iterator arguments, e.g.
  - `insert(`*iterator*`,` *x*`)` – the insertion of `x` can typically be relatively costly
- emplace operations
  - copy/move-free, in-place construction
  - given arguments can be passed directly to a constructor for the element type
  - vector has `emplace_back`(*args*) and `emplace`(*iterator*, *args*) – *args* to be passed to a constructor of the element type

Standard Containers Library (2)

- basic *sequence containers*
  - `array` – a fixed-sized container with performance of built-in arrays (std::arrays are *aggregates*)
  - `vector` – the sequence container to “use by default”
  - `deque` – when most insertions and deletions take place at the beginning or end
  - `list` – when frequent insertions and deletions not at the beginning and/or end
  - `forward_list` – a container with no storage or time overhead compared to a hand-written singly linked list
- *sequence adaptors* represent three frequently used specialized data structures
  - `stack`
  - `queue`
  - `priority_queue`
  - have a basic sequence container as member
  - have an adapted interface – the set of operations is reduced and operations renamed according to convention
  - provide no iterators
- *associative containers* (*ordered*)
  - `set` and `multiset` store *keys only* – the key is the value
  - `map` and `multimap` store *key-value pairs*
- *unordered associative containers* – hash-table-like data structures, bucket-organized
  - `unordered_set` and `unordered_multiset`
  - `unordered_map` and `unordered_multimap`

Regarding exam

- important to have a good overview and understanding of the containers library
  - good knowledge of different container types and their common and specific features
  - sufficient practice in using containers, in combination with other components
  - unordered associative containers will not be required for solving programming problems
- *cplusplus.com Reference* will be available at exam (web browser)
- see the course examination page for more information

Container member types

- all containers defines *member types*, e.g. `vector` defines the following

```
typedef T                                                  value_type;
typedef Allocator                                          allocator_type;

typedef value_type&                                        reference;
typedef const value_type&                                  const_reference;

typedef implementation-defined                             iterator;
typedef implementation-defined                             const_iterator;

typedef reverse_iterator<iterator>                         reverse_iterator;
typedef reverse_iterator<const_iterator>                   const_reverse_iterator

typedef implementation-defined                             size_type;
typedef implementation-defined                             difference_type;

typedef typename allocator_traits<Allocator>::pointer       pointer;
typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
```

  - used in declarations of the member functions for declaring parameter types and return types
  - used by other standard library components
  - to be used otherwise whenever possible to reduce implementation dependencies and relax couplings
  - *implementation-defined* types are typically defined by the associated memory allocator
  - reverse iterators are easily defined by the iterator adapter `reverse_iterator` template

Container iterators

- all containers *except* the sequence adaptors provide container specific iterators
- container iterator types

```
iterator
const_iterator

reverse_iterator
const_reverse_iterator
```

  – `const` refers to the container and its elements
- iterators are divided into different *iterator categories* representing different levels of functionality
  – `vector`, `deque` and `array` provide *random access iterators*
  – `list` and all *associative containers* provide *bidirectional iterators*
  – `forward_list` and all *unordered associative containers* provide *forward iterators*
- there is a special iterator value – *past-the-end iterator*
  – represents the iterator position following after the last element in a container
  – only for comparing with other iterators

```
for (vector<int>::iterator it = begin(v); it != end(v); ++it)
{
   cout << *it << '\n';
}
```

  – there might not exist any valid *before-the-first* position for some container implementations

Container iterator interface

c`.begin()`	Returns an iterator to the first element, a *past-the-end* iterator if c is empty
c`.end()`	Returns a *past-the-end* iterator
c`.rbegin()`	Returns a *past-the-end* reverse iterator
c`.rend()`	Returns a reverse iterator to first element in c, a *past-the-end* reverse iterator if c is empty

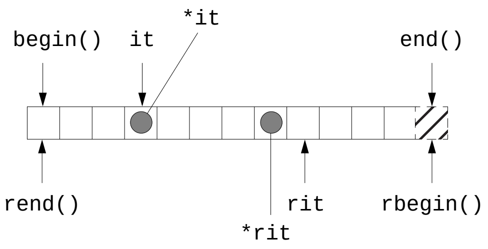
- each member function above is overloaded in a **const** and a non-const version
  – if the container object is non-const the returned iterator is an `iterator` or `reverse_iterator`
  – if the container object is **const** the returned iterator is a `const_iterator` or `const_reverse_iterator`

c`.cbegin()`	Returns a const iterator to first element, *past-the-end* const iterator if c is empty
c`.cend()`	Returns a *past-the-end* const iterator
c`.crbegin()`	Returns a *past-the-end* const reverse iterator
c`.crend()`	Returns a const reverse iterator to first element in c, a *past-the-end* const iterator if c is empty

- these are only declared as **const** member functions
  – can be invoked for both **const** and non-const containers
- `forward_list` and *unordered associative containers* does not provide reverse iterators
  – no `rbegin()`, `rend()`, `crbegin()`, or `crend()`

Container iterator semantics

- `reverse_iterator` is moved *backwards* with `++` – makes iterators and reverse iterators exchangeable
- `rend()` points to the same element as `begin()` – *the first element* (if any)
- `rbegin()` points to the same element as `end()` – *"past-the-end-iterator"*
- When a `reverse_iterator` is dereferenced, it is the *preceding* element that is obtained – don't dereference `rend()`

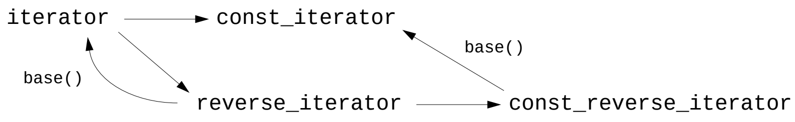


For reverse iterators, **`operator`**`*` and **`operator`**`->()` are implemented as:

```
reference operator*() const
{
   iterator tmp = *this;
   return *--tmp;
}
```

```
pointer operator->() const
{
   return &(operator*());
}
```

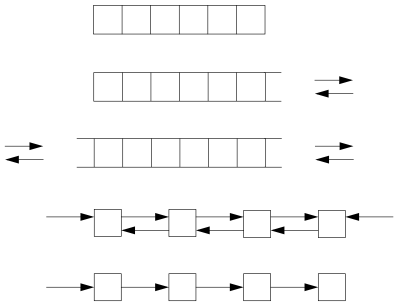

Container iterators – type conversions



- implicit conversions
  – from `iterator` to `const_iterator`
  – from `iterator` to `reverse_iterator`
  – from `const_iterator` to `const_reverse_iterator`
- explicit conversions – member function `base()`
  – from `reverse_iterator` to `iterator`
  – from `const_reverse_iterator` to `const_iterator`

Sequence containers

- organizes objects of the same kind into a strictly linear arrangement
- how they can be pictured
  - `array`
  - `vector`
  - `deque`
  - `list`
  - `forward_list`



- for some there is a distinction between *size* and *capacity*
  - *size* refers to the number of stored elements – the *logical size*
  - *capacity* refers to the size of the available storage – the *physical size*
- `deque` actually require a more complicated implementation than illustrated above to fulfill complexity requirements

Sequence containers – construction, copying, assignment, and destruction (1)

All sequence containers have

- *default constructor*, *copy constructor*, *copy assignment operator* and *destructor*

```
vector<int> v1;

vector<int> v2{ v1 };

v2 = v1;

array<int, 100> a;
```

- *move constructor* and *move assignment operator*

```
vector<int> v3{ std::move(v1) };

v3 = std::move(v2);
```

- constructor taking an *initializer list*

```
vector<int> v4{ 1, 2, 3 };
```


Sequence containers – construction, copying, assignment, and destruction (2)

All sequence containers *except* `array` have

- constructor for initializing with *n* value-initialized elements

```
vector<int> v5{ 10 };
```

- constructor for initializing with *n* elements with a *specific value – cannot use braces here!*

```
vector<int> v6(10, -1);
```

- constructor for initialize with values from an *iterator range*

```
vector<int> v7{ begin(v6), end(v6) };
```

- assignment operator taking an *initializer list*

```
v7 = { 1, 2, 3 };
```

- assign member functions

```
v5.assign({ 1, 2, 3 });              // values from an initializer list
v6.assign(10, -1);                   // 10 elements with value -1
v7.assign(begin(v6), end(v6));       // values from the range [begin(v6), end(v6))
```

Note: There is a versions of all constructors also taking an *allocator*.

Sequence containers – operations (1)

Size, capacity	vector	deque	list	forward_list	array	
n = *c*.size()	•	•	•		•	size() == max_size() for array
m = *c*.max_size()	•	•	•	•	•	
c.resize(*sz*)	•	•	•			increase or decrease
c.resize(*sz*, *x*)	•	•				
n = *c*.capacity()	•					
b = *c*.empty()	•	•	•	•	•	
c.reserve(*n*)	•					increase capacity to at least *n*
c.shrink_to_fit()	•	•				reduces capacity, maybe to size()

Element access	vector	deque	list	forward_list	array	
c.front()	•	•	•	•	•	
c.back()	•	•	•		•	
c[*i*]	•	•			•	
c.at(*i*)	•	•			•	throws out_of_range, if invalid *i*
data()	•				•	pointer to first element

Note: access functions, except data(), returns a *reference* which allows for modification, if c is non-const

Sequence containers – operations (2)

Modifiers	vector	deque	list	forward_list	array
`c.push_back(x)`	•	•	•		
`c.pop_back()`	•	•	•		
`c.push_front(x)`		•	•	•	
`c.pop_front()`		•	•	•	
`it = c.insert(begin(c), x)`	•	•	•		
`it = c.insert(begin(c), n, x)`	•	•	•		
`it = c.insert(begin(c), { x, y, z })`	•	•	•		
`c.insert(begin(c), it1, it2)`	•	•	•		
`it = c.erase(begin(c))`	•	•	•		
`it = c.erase(begin(c), end(c))`	•	•	•		
`c1.swap(c2)`	•	•	•	•	•
`c.clear()`	•	•	•	•	

Sequence containers – operations (3)

Modifiers, cont.	vector	deque	list	forward_list	array
`c.emplace_front(args)`		•	•	•	
`c.emplace_back(args)`	•	•	•		
`it = c.emplace(pos, args)`	•	•	•		
`it = c.emplace_after(args)`				•	

- *emplace* – “to put into place” or “put into position”
  - emplace functions are variadic template functions
  - *args* is a template parameter pack – constructor arguments matching a constructor for the element type is expected
  - stored objects are created inplace – no copy/move required (constructor arguments will be copied or moved)

Comparisons	vector	deque	list	forward_list	array
`== !=`	•	•	•	•	•
`< <= > >=`	•	•	•	•	•

Specialized algorithm

`swap(c1, c2)`	•	•	•	•	•

Sequence containers – special list and forward_list operations

Since list and forward_list does not provide random access iterators, a number of general algorithms cannot be applied.

The following member functions corresponds to such general algorithms	
`c.remove(x)`	remove *x*
`c.remove_if(pred)`	remove elements for which *pred*(*element*) is true
`c.unique()`	remove consecutive equivalent elements; == used for comparison
`c.unique(pred)`	remove consecutive equivalent elements; *pred* used for comparison
`c1.merge(c2)`	merge *c1* and *c2*; both must be ordered with <; *c2* becomes empty
`c1.merge(c2, comp)`	merge *c1* and *c2*; both must be ordered with comp; *c2* becomes empty
`c1.sort()`	order elements with <
`c1.sort(comp)`	order elements using *comp*
`c1.reverse()`	place elements in reverse order

There is also a number of special member functions, overloaded in several versions, related to list *inserting*, *splicing* and *erasing*

- `insert()`, `splice()`, and `erase()` for `list`
- `insert_after()`, `splice_after()`, and `erase_after()` for `forward_list`
- *inserting* will not affect source
- *splicing* will remove the element(s) in question from source

Sequence adaptors

- each adaptor class adapts the interface to model one of three classic data structures
  - `stack`
  - `queue`
  - `priority_queue`
- have a sequence container as private data member to store the elements
  - for each adaptor type a default sequence container type is used internally

```
stack           – deque
queue           – deque
priority_queue  – vector
```

  - this container can be replaced by any other container fulfilling the requirements (operations and complexity)
  - standard library *heap operations* are used to implement `priority_queue` operations
- these adaptions significantly simplifies the interface
  - substantial reduction of the number of operations, compared to the sequence container used internally
  - operations renamed according to “tradition” – e.g. *push*, *top* and *pop* for stack
  - no iterators provided – not relevant for these data structures

Sequence adaptors – construction, copying, assignment, and destruction

All sequence adaptors have

- *default constructor*, *copy constructor*, *copy assignment operator* and *destructor*

```
priority_queue<int> pq1;
```

- *move constructor* and *move assignment operator*
- constructor for initializing with elements from a *container*

```
vector<int> v;
...
queue<int> q1{ v };
```

- versions of the constructors and assignment operators above also taking an *allocator*

`priority_queue` also have

- constructor taking a *comparer* – default is `std::less` (corresponds to **operator**<)

```
priority_queue<int, std::greater<int>> pq2;
```

- constructor for initializing with elements from an *iterator range*

```
priority_queue<int> pq2{ begin(v), end(v) };
```

- versions of these constructors taking an *allocator*

Sequence adaptors – operations

Size, capacity	stack	queue	priority_queue
n = *a*.size()	•	•	•
b = *a*.empty()	•	•	•

Element access	stack	queue	priority_queue
x = *a*.top()	•		•
x = *pq*.front()		•	
x = *pq*.back()		•	

Modifiers	stack	queue	priority_queue
a.push(*x*)	•	•	•
a.pop()	•	•	•
a.emplace(*args*)	•	•	•

Comparisons	stack	queue	priority_queue
== !=	•	•	
< <= > >=	•	•	

std::initializer_list (1)

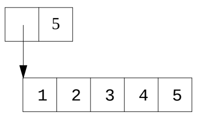
This is *not* a container type, it's a language support component for implementing initializer lists – `<initializer_list>`

- have some similarities with sequence containers
- elements are stored in a hidden array of **const** E
- typically used as function parameter type for passing initializer lists as argument

```
vector& operator=(initializer_list<T>);

v = { 1, 2, 3, 4, 5 };
```

A pair of pointers, or a pointer and a length is obvious representations (GCC uses the latter)



- a private constructor, to be used by the compiler initialize this

Special member functions not declared (all except default constructor) are compiler generated.

- copying an initializer list does not copy the underlying elements
  - shallow copy

std::initializer_list (2)

```
template<class E> class initializer_list
{
public:
   typedef E           value_type;
   typedef const E&    reference;             // elements are const !
   typedef const E&    const_reference;
   typedef std::size_t size_type;
   typedef const E*    iterator;              // elements are const !
   typedef const E*    const_iterator;

   initializer_list() noexcept;

   size_type size() const noexcept;           // number of elements

   const_iterator begin() const noexcept;     // first element
   const_iterator end() const noexcept;       // one past the last element
};

// initializer list range access
template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;

template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
```


Utility class pair

- utility class for storing pair of values – used by map containers and functions returning two values
- defines types used by other components of the standard library (`T1` and `T2` are the template parameters)

```
typedef T1 first_type;
typedef T2 second_type;
```

- have default constructor, copy/move constructor, copy/move assignment operator, destructor and type converting constructors

```
pair<int, char>      p2{ 1, 'A' };      initialize with a pair of values
pair<double, int>    p3{ p2 };          automatic type conversion
```

- element access – all members are **public**

```
p2.first
p2.second
```

- comparisons

```
==   !=   <   >   <=   >=
```

- utility template function `make_pair` to ease creation of a pair – template type parameters are deduced from the arguments

```
p = make_pair(x, y)
```


Associative containers

Provide fast retrieval of data based on *keys*

- the following associative containers are *ordered*

```
map          multimap          set          multiset
```

- corresponding *unordered associative containers*

```
unordered_map    unordered_multimap     unordered_set    unordered_multiset
```

- *set containers* store only *keys* (values)
- *map containers* store *key-value-pairs* – uses `pair` to store a key and its associated value
- non-multi variants only allows *unique keys*
- multi variants allows *equivalent keys*
  - several elements may have the same key and also the associated value may be the same
- in (*ordered) associative containers* elements are *ordered by key*
  - parametrized on *key type* and *ordering relation* (and *memory allocator*)
  - maps also associate an *arbitrary type* with the key, the *value type*
  - typically implemented as a *binary search tree* (e.g. a red-black tree, a height-balanced binary search tree)
- in *unordered associative containers* elements are stored in a *hash table*
  - parametrized on *key type*, *hash function* (unary function object), and *equality relation* (a binary predicate) (and *memory allocator*)
  - maps also associate an *arbitrary type* with the key, the *value type*
  - elements are organized into *buckets* – elements with keys having the same hash code appears in the same bucket

Associative containers – construction, copying, assignment, and destruction

All associative containers have

- *default constructor*, *copy constructor*, *copy assignment operator* and *destructor*

```
map<int, string>   m;

multi_set<string>  ms;

unordered_map<int, X, hasher, equal>  um;
```

- *move constructor* and *move assignment operator*
- *constructor* and *assignment operator* to initialize/assign with values from an *initializer list*
  - unordered associative container constructor version can also take *initial number of buckets*, *hash function*, and *equality relation*
- *constructor* for initializing with values from an *iterator range*
  - for ordered associative containers also with the possibility to give a *comparer* and an *allocator*
  - for unordered associative containers also the *initial number of buckets*, *hash function*, *equality relation*, and an *allocator* can be given
- *constructors* equivalent to the *default*, *copy and move constructors*, also taking an *allocator*

Associative containers (ordered)

```
#include <map>

#include <set>
```

Additional or redeclared member types

- map and multimap

```
typedef Key                    key_type;        // Key  is a template type parameter
typedef T                      mapped_type;     // T  is a template type parameter
typedef pair<const Key, T>     value_type;
typedef Compare                key_compare;     // Compare  is a template type parameter
```

`value_compare` *is declared as a nested function object class*

- set and multiset

```
typedef Key                    key_type;        // Key  is a template type parameter
typedef Key                    value_type;
typedef Compare                key_compare;     // Compare  is a template type parameter
typedef Compare                value_compare;
```

Associative containers (ordered) – operations (1)

Size, capacity	map	multimap	set	multiset
n = *a*.size()	•	•	•	•
n = *a*.max_size()	•	•	•	•
b = *a*.empty()	•	•	•	•

Comparisons	map	multimap	set	multiset
== !=	•	•	•	•
< <= > >=	•	•	•	•

Element access	map	multimap	set	multiset	
x = *a*[*k*]	•				
x = *a*.at(*k*)	•				throws out_of_range if no such element

Note: if an element with key *k* does not exist in the map, it is created by **operator**[], with the associated value default initialized

```
return value_type(k, T());
```

a *reference* is returned which can be used to assign a value to `second`

```
m[k] = x;
```


Associative containers (ordered) – operations (2)

Modifiers	map	multimap	set	multiset	
pair<*iterator*, **bool**> p = *a*.insert(*x*)	•		•		*
it = *a*.insert(*x*)		•		•	*
it = *a*.insert(*pos*, *x*)	•	•	•	•	*
a.insert(*first*, *last*)	•	•	•	•	
a.insert({ *x*, *y*, *z* })	•	•	•	•	
pair<*iterator*, **bool**> p = *a*.emplace(*args*);	•		•		
it = *a*.emplace(*args*)		•		•	
it = *a*.emplace_hint(*pos*, *args*)	•	•	•	•	
a.erase(*it*)	•	•	•	•	
n = *a*.erase(*k*)	•	•	•	•	
a.erase(*first*, *last*)	•	•	•	•	
a1.swap(*a2*)	•	•	•	•	
a.clear()	•	•	•	•	

*) in case `map` and `multimap`, *x* has type `pair<key_type, value_type>`

Associative containers (ordered) – operations (3)

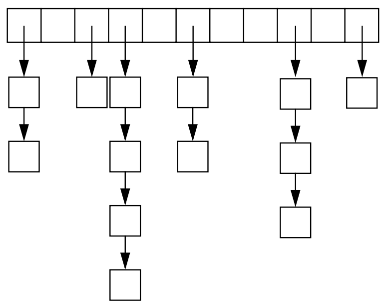
Map and set operations	map	multimap	set	multiset
it = *a*.find(*k*)	•	•	•	•
n = *a*.count(*k*)	•	•	•	•
it = *a*.lower_bound(*k*)	•	•	•	•
it = *a*.upper_bound(*k*)	•	•	•	•
pair<*iter*, *iter*> *p* = *a*.equal_range(*k*);	•	•	•	•

Specialized operation	map	multimap	set	multiset
swap(*a1*, *a2*)	•	•	•	•

Observers	map	multimap	set	multiset
comp = *a*.key_comp()	•	•	•	•
comp = *a*.value_comp()	•	•	•	•

Unordered associative containers

- declared in `<unordered_map>` and `<unordered_set>`
- based on hash tables
  - bucket count can change dynamically
  - initial number of buckets is implementation defined
  - each bucket has its own chain of elements
- default hash functions declared in <functional>, <string>, …
  - **bool**
  - character types
  - integer types
  - floating point types
  - string types
  - pointer types
  - smart pointer types



Conceptually, implementations may differ!

Unordered associative containers

Additional or redeclared member types

- `unordered_map` and `unordered_multimap`

```
typedef Key                     key_type;        // Key is a template type parameter
typedef T                       mapped_type;     // T is a template type parameter
typedef pair<const Key, T>      value_type;
```

- `unordered_set` and `unordered_multiset`

```
typedef Key                     key_type;        // Key is a template type parameter
typedef Key                     value_type;
```

- hash function and hash key related

```
typedef Hash                    hasher;          // Hash is a template type parameter
typedef Pred                    key_equal;       // Pred is a template type parameter
```

- bucket iterators

```
typedef implementation-defined  local_iterator;
typedef implementation-defined  const_local_iterator;
```


Unordered associative containers operations (1)

Size, capacity	unordered_map	unordered_multimap	unordered_set	unordered_multiset
n = *u*.size()	•	•	•	•
n = *u*.max_size()	•	•	•	•
b = *u*.empty()	•	•	•	•

Comparisons	unordered_map	unordered_multimap	unordered_set	unordered_multiset
== !=	•	•	•	•
< <= > >=				

Element access	unordered_map	unordered_multimap	unordered_set	unordered_multiset
x = *u*[*k*]	•			
x = *u*.at(*k*)	•			

Note 1: if an element with key k does not exist in the map, it is created by **operator**[], with the associated value default initialized. A reference to the associated value is returned and can be used to assign a value.

Note 2: if an element with key k does not exist in the map, member function `at(k)` throws `out_of_range`.

Unordered associative containers operations (2)

Modifiers	u_map	u_multimap	u_set	u_multiset	
pair<*iterator*, **bool**> *p* = *u*.insert(*x*);	•		•		*
it = *u*.insert(*x*)		•		•	*
it = *u*.insert(*it_hint*, *x*)	•	•	•	•	*
u.insert(*first*, *last*)	•	•	•	•	
u.insert({ *x*, *y*, *z* })	•	•	•	•	
pair<*iterator*, **bool**> *p* = *u*.emplace(*args*);	•		•		
it = *u*.emplace(*args*)		•		•	
it = *u*.emplace_hint(*pos*, *args*)	•	•	•	•	
u.erase(*it*)	•	•	•	•	
n = *u*.erase(*k*)	•	•	•	•	
u.erase(*first*, *last*)	•	•	•	•	
u1.swap(*u2*)	•	•	•	•	
u.clear()	•	•	•	•	

*) in case unordered_map or unordered_multimap, *x* has type `pair<key_type, value_type>`

Unordered associative containers operations (3)

Map and set operations	u_map	u_multimap	u_set	u_multiset
it = *u*.find(*k*)	•	•	•	•
n = *u*.count(*k*)	•	•	•	•
pair<*iter*, *iter*> *p* = *u*.equal_range(*k*);	•	•	•	•

Specialized operation	u_map	u_multimap	u_set	u_multiset
swap(*u1*, *u2*)	•	•	•	•

Unordered associative containers operations (4)

Bucket interface

The hash table used for storing elements is bucket organized – each hash entry can store several elements.

n = *u*.bucket_count()	current number of buckets for container *u*
m = *u*.max_bucket_count()	maximum number of buckets possible for container *u*
s = *u*.bucket_size(*b*)	number of elements in bucket *b*
b = *u*.bucket(*k*)	bucket number where keys equivalent to *k* would be found
it = *u*.begin(*b*)	iterator to the first element in bucket *b*
it = *u*.end(*b*)	**const** *past-the-end* iterator for bucket *b*
it = *u*.cbegin(*b*)	iterator to the first element in bucket *b*
it = *u*.cend(*b*)	**const** *past-the-end* iterator for bucket *b*
pair<*it*, *it*> p = *u*.equal_range(*k*);	range containing elements with keys equivalent to *k*

Unordered associative containers operations (5)

Hash policy interface

f = *u*.load_factor()	average number of elements per bucket.
f = *u*.max_load_factor()	a positive number that the container attempts to keep the load factor less than or equal to
u.max_load_factor(*f*)	may change the container's maximum load factor, using *f* as a hint
u.rehash(*n*)	alters the number of buckets to be at least *n* buckets – rebuilds the hash table as needed
u.reserve(*n*)	reserves space for at least the specified number of elements and regenerates the hash table

Observers

n = *u*.hash_function()	*u*'s hash function
eq = *u*.key_eq()	key equality predicate

Iterating over buckets and bucket contents

```
unordered_map<string> table;

// table is populated…

auto n_buckets = table.bucket_count();

for (decltype(n_buckets) b = 0; b < n_buckets; ++b)
{
   cout << "Bucket " << b " has " << table.bucket_size(b) << " elements: ";

   copy(table.cbegin(b), table.cend(b), ostream_iterator<string>(cout, ", "));
   cout << '\n';
}
```

The type of n_buckets and b is unordered_map<string, Item>::size_type

Some comments on containers library design

- covers a wide variety of common data structures
- uniform interfaces
- all containers, except the sequence adaptors, provide iterators
  - the elements in different type of containers can be operated upon in a uniform manner
- policy technique is used for different adaptions, e.g.
  - memory allocation
  - comparison and equivalence functions
  - hash functions
- supports static polymorphism and generic programming
  - templates parametrized on different aspects
- supports move semantics, perfect forwarding, and emplacement construction
- type traits and concept checking is frequent in the implementation
  - checking requirements for instantiation types
  - selecting the most efficient implementation among several candidates
  - solving ambiguities that may arise due to instantiation types
- given array, vector and string there is little reason to use C-style arrays