Utilities

Pairs and Tuples

- pairs:
  - used by maps to manage the elements
  - minmax returns a pair
  - first_type and second_type
  - expected ctors
  - .first
  - .second
  - get<0>(p), get<1>(p)
  - swap
  - make_pair (can pass arguments with std::move if argument is no longer used)
  - tuple_size<>::value
  - tuple_element<0, >::type
  - tuple_element<1, >::type
  - no single copy constructor for non-constant type
  - pair(piecewise_construct_t, Args1..., Args2...) -> piecewise construction
  - can use std::tie to extract values
- tuples:
  - make_tuple()
  - get(t)
  - tie(ref1,...) -> creates a tuple of references which can be extracted (use std::ref) also allows for saving the values of a tuple back to variables
  - tuple_size::value
  - tuple_element<idx, tupletype>::type
  - tuple_get
  - tuple_cat
- i/o for tuples:
  - boost had support for stream I/O, not in std library
  - see this section for more info on the how to print a tuple
- conversions between tuples and pairs:
  - a pair can be initialized with a two-element tuple
  - a tuple can be initialized with a pair

Smart Pointers

- uses of pointers:
    - reference semantics outside of the boundaries of scope
- may use pointers in multiple containers but want it to stay alive on deletion of one container
- shared_ptr:
    - cleanup when the object was needed for use in multiple places but not needed anymore
    - defaults to delete
    - for arrays need to specify the cleanup
    - access with * and ->
    - cannot use assignment with "=" because this is an implicit conversion
        - std::shared_ptr name(new Type())
        - std::shared_ptr name{new Type()} or Type{}
    - prefer make shared -> only uses one instead of two allocations so faster and safer
    - cannot reassign with "=", need to use reset
    - .use_count()
    - can initialize with custom deleter

```
std::shared_ptr<int> name{new int[10], [](int* p) { delete[] p; }
```

```
- shared_ptr and unique_ptr deal with deleters in different ways
    - std::unique_ptr<int[]> name{new int[10]) // OK
    - std::shared_ptr<int[]> name{new int[10]) // ERROR
    - need a second template argument for unique_ptr deleters
```

```
std::unique_ptr<int, void(*)(int*)> name{new int[10], [](int* p) { delete[] p; }
```

```
- may need other destruction policies for resources, i.e. file deleter, etc
```

- weak_ptr:
    - uses:
        - cyclic references
        - sharing without ownership
    - cannot use * and ->, must create a shared_ptr
        - checks for existence of object and throws otherwise
        - the object won't be released while accessing it through a shared_ptr
    - provides just enough operations to create, copy and assign
    - perfect for tree structures with parent references

        - reset
        - swap
        - use_count
        - expired -> checks for existence of object
        - lock -> creates a shared_ptr
        - owner_before -> owner based ordering of weak_ptrs
        - options to check:
            - call expired
            - convert to shared_ptr and throw otherwise
            - use_count -> generally use for debugging due to performance
- misusing shared pointers:
    - ensure only one group of share_ptr own an object
        - do not create multiple shared_ptrs from the same object
        - duplicate shared_ptrs by creating new ones from the original
        - the problem might also occur indirectly by returning shared_ptr from a method that creates a shared_ptr from this
        - can use enable_shared_from_this<> to enable returning by deriving from it
- shared_ptr:
    - ctors:
        - (ptr)
        - (ptr, deleter)
        - (ptr, deleter, allocator)
        - (sp2) -> for sharing ownership
        - (move(sp2)) -> releases ownership to create the new shared_ptr
        - (move(up))
        - (move(ap))
    - = sp2 -> shares ownership
    - = move(sp2) -> releases ownership
    - = move(up)
    - = move(ap)
    - .swap
    - swap
    - reset() -> releases and reinitializes to empty
    - reset(ptr)
    - reset(ptr, deleter)
    - reset(ptr, deleter, allocator)
    - make_shared(args...)
    - allocate_shared(allocator, args...)
    - .get

    - *sp
    - sp->
    - use_count
    - unique
    - if (sp)
    - get_deleter
    - strm << sp
    - sp.owner_before(sp2)
    - sp.owner_before(wp)
- more sophisticated shared_ptr operations:
    - aliasing constructor -> ctor(sp, &sp->member) -> object owns member so the shared_ptr to the member needs to keep the object alive
    - also examples for containers
    - need to keep container alive
    - shared_ptr ctors from new Object causes two allocations, one for object and one for control block
    - make_shared and allocate_shared only cause one allocation
- thread-safe shared pointer interface:
    - shared_ptr is not thread safe
    - use std::atomic_store to write a local shared_ptr to thread shared shared_ptrs safely
    - use_count can be accessed concurrently but may not be correct
    - additional:
        - atomic_is_lock_free
        - atomic_load -> returns the sp
        - atomic_store -> assigns to sp
        - atomic_exchange -> swaps
- unique_ptr:
    - uses:
        - basic RAII for exclusive ownership of a heap allocated object
    - cannot initialize with up = new Object
    - must use up(new Object) or up{new Object)
    - can be empty, use up = nullptr or up.reset()
    - release gives up ownership and returns the pointer to the object
    - cannot copy by initialization from up1(up2)
    - must use up1(std::move(up2)) to release ownership
    - cannot assign with up1 = up2
    - must use up1 = std::move(up2)
    - cannot assign with up = new Object
    - can assign with up = std::unique_ptr(new Object)

- source and sink:
  - unique_ptrs are good for transferring ownership between functions
  - if function takes unique_ptr as an arg (as rvalue ref with std::move) and does not return it on exit, the function is a sink
  - returning a new unique_ptr creates a source of data, no need for std::move, the standard dictates that the compiler will try this automatically
- unique_ptrs as members:
  - use of unique_ptr allows for avoidance of a need to create a dtor
  - copy ctor needs to manually copy construct each of the associated values into a unique_ptr i.e.

```
Class(const Class& x) : ptr{new Member{*x.ptr} {}
```

```
- need dtor otherwise
```

- dealing with arrays:
  - unique_ptr needs the <Object[]> specialization to create an array of objects
  - - and -> are not usable, instead operator[] is provided
- class default delete:
- deleters for other associated resources:
- auto_ptr:

Numeric Limits

- prefer defaulting to platform independent limits:

```
char            1
short int       2
int             2
long int        4
long long int   8
float           4
double          8
long double     8
```

- numeric_limits
  - is_specialized = bool
  - is_signed
  - is_integer
  - is_exact
  - is_bounded
  - is_iec559 - conforms to IEC 759 and IEEE 754
  - min()

  - max()
  - lowest()
  - digits = int
  - digits10 - decimal digits for floats
  - max_digits10
  - radix - int: base of repr (usually 2), float: base of exponent repr
  - min_exponent
  - max_exponent
  - min_exponent10
  - max_exponent10
  - epsilon - diff between 1 and min value > 1
  - round_style
  - round_error()
  - has_infinity
  - infinity()
  - has_quiet_NaN
  - quiet_NaN()
  - has_signaling_NaN
  - signaling_NaN()
  - has_denorm
  - has_denorm_loss
  - denorm_min()
  - traps
  - tinyness_before
  - round style:
    - round_toward_zero
    - round_to_nearest
    - round_toward_infinity
    - round_toward_neg_infinity
    - round_indeterminate
  - denorm style:
    - denorm_absent
    - denorm_present
    - denorm_indeterminate

Type Traits and Type Utilities

- type selection

- flexible overloading for integral types
- processing the common type (std::common_type for common base class)
- unary type predicates:
    - is_void
    - is_integral
    - is_floating_point
    - is_arithmetic
    - is_signed
    - is_unsigned
    - is_const
    - is_volatile
    - is_array (ordinary array)
    - is_enum
    - is_union
    - is_class
    - is_function
    - is_reference
    - is_lvalue_reference
    - is_rvalue_reference
    - is_pointer
    - is_member_pointer
    - is_member_object_pointer
    - is_member_function_pointer
    - is_fundamental (void, integral (including bool and char), floating point, nullptr_t)
    - is_scalar
    - is_object (anything but void, function, or reference)
    - is_compound (array, enumeration, union, class, function, reference, or pointer)
    - is_trivial
    - is_trivially_copyable
    - is_standard_layout
    - is_pod
    - is_literal_type
    - is_empty
    - is_polymorphic
    - is_abstract
    - has_virtual_destructor
    - is_default_constructible
    - is_copy_constructible

    - is_move_constructible
    - is_copy_assignable
    - is_move_assignable
    - is_destructible
    - is_trivially_default_constructible
    - is_trivially_copy_constructible
    - is_trivially_move_constructible
    - is_trivially_copy_constructible
    - is_trivially_move_constructible
    - is_nothrow_default_constructible
    - is_nothrow_copy_constructible
    - is_nothrow_move_constructible
    - is_nothrow_copy_assignable
    - is_nothrow_move_assignable
    - is_nothrow_destructible
- traits for type relations:
    - is_same
    - is_base_of
    - is_convertible
    - is_constructible
    - is_trivially_constructible
    - is_nothrow_constructible
    - is_assignable
    - is_trivially_assignable
    - is_nothrow_assignable
    - uses_allocator
- type modifiers:
    - add_const
    - add_lvalue_reference
    - add_rvalue_reference
    - add_pointer
    - make_signed
    - make_unsigned
    - remove_const
    - remove_reference
    - remove_pointer
    - remove_const
    - remove_volatile

    - remove_cv
- other:
    - rank (dimensions of array type)
    - extent (extent of dimension)
    - remove_extent
    - remove_all_extents
    - underlying_type (of enum type)
    - decay (convert to corresponding by-value type)
    - enable_if
    - conditional
    - common_type
    - result_of (resulting type of calling f with args)
    - alignment_of
    - aligned_storage
    - aligned_union

Reference Wrappers

- std::reference_wrapper<> can allow containers to take references (usually can't) and can be used to pass references to functions that take value types

Function Type Wrappers

Auxiliary Functions

- processing the minimum and maximum
    - min(a, b)
    - min(a, b, cmp)
    - min(initlist)
    - min(initlist, cmp)
    - max(a, b)
    - max(a, b, cmp)
    - max(initlist)
    - max(initlist, cmp)
    - minmax(a, b)
    - minmax(a, b, cmp)
    - minmax(initlist)
    - minmax(initlist, cmp)
- swapping two values

- supplementary comparison operators
    - provides !=, >, <= and >= if == and < are defined ()

Compile-Time Fractional Arithmetic with Class ratio<>

Clocks and Timers

- duration
- clock
    - high_resolution_clock
    - steady_clock
- timepoint
    - epoch
    - current time
    - minimum timepoint
    - maximum timepoint
- duration arithmetic
- casting of time related types
- ctime:
    - clock_t
    - time_t
    - struct tm
    - clock()
    - time()
    - difftime()
    - localtime()
    - gmtime()
    - asctime()
    - strftime()
    - ctime()
    - mktime()

Header Files , , and

- cstddef

    - NULL
    - nullptr_t
    - size_t

    - ptrdiff_t
    - max_align_t
    - offsetof(type, member)

- cstdlib

    - EXIT_SUCCESS
    - EXIT_FAILURE
    - exit(int status)
    - quick_exit(int status)
    - _Exit(int status)
    - abort()
    - atexit(void (*func)())
    - at_quick_exit(void (*func)())

- cstring

    - memchr -> finds char in len bytes of ptr
    - memcmp -> compares bytes of memory
    - memcpy -> copies memory
    - memmove -> moves memory
    - memset -> assigns characters to len

The Standard Template Library

STL Components

Containers

- sequence containers:
    - array
    - vector
    - deque
    - list and forward_list
- associative containers:
    - set
    - multiset
    - map
    - multimap

- unordered containers:
    - unordered_set
    - unordered_multiset
    - unordered_map
    - unordered_multimap
- associative arrays:
    - all map types
- other containers:
    - strings
    - c-style arrays
    - user defined containers
- container adapters:
    - stack
    - queue
    - priority_queue

Iterators

- operator*
- operator++
- operator==
- operator=
- retrieve with begin() and end() (plus more)
- prefer preincrement over postincrement
- range base for loop generates the following code:

```
for (type elem : coll) {
    ...
}

// becomes
for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {
    type elem = *pos;
    ...
}
```

- categories:
    - forward
    - bidirectional
    - random-access
    - input
    - output

Algorithms

- process ranges
- need to check ranges on your own
- processes half-open ranges
    - includes beginning
    - excludes end
- some algorithms handle multiple ranges
    - make sure second range has at least as many as the first

Iterator Adapters

- abstraction with interface of iterator but used for a different purpose
- insert iterators: algorithms operate in insert mode rather than overwrite mode
    - allows destination to grow accordingly
    - assignment inserts at the current iterator location
    - a call to step forward is a no-op
- back inserters: append to back of a range by calling push_back
- front inserters: insert the element at the front by calling push front
- inserters: insert directly in front of the position specified as the second argument
- stream iterators: read from or write to a stream
- reverse iterators: operate backwards by switching increment operator to move in reverse
- move iterators: convert access to element to move operations

User-Defined Generic Functions

Manipulating Algorithms

- removing elements:
    - remove will just move the following elements back into the erased element but elements at the end will remain unchanged
    - the iterator returned by remove points to the new end (short of the actual end)
    - to erase, call the member function .erase(remove(), .end())
    - from effective STL, prefer for_each with a custom remove function for containers of pointers rather than remove_if
- manipulating associative and unordered containers:

- algorithms versus member functions:
    - sometimes member functions provide better performance

Functions as Algorithm Arguments

Using Lambdas

Function Objects

Container Elements

Errors and Exceptions inside the STL

Extending the STL

STL Containers

Common Container Abilities and Operations

- abilities:
    - value over reference semantics -> copy and/or move elements on insert, can use reference_wrapper to achieve reference semantics (or shared pointers)
    - for non-copyable objects, can rely on move or use pointers
    - containers (including unordered) have a specific order and this order is only invalidated on insert or delete
- operations:
    - see list (remember ContType c; -> default ctor, also ContType c(o.begin(), o.end()))
    - can initialize from the elements of a c-style array
    - default initialization results in empty container (except array, default initializes elements of array)
    - can initialize from std input ContType c{std::istream_iterator(std::cin), std::istream_iterator()}
    - size:
        - empty
        - size
        - max_size

- comparisons:
    - must have same type
    - equal if elements are same and have same order
    - < results in lexicographical compare
- access:
    - range based for loop (don't forget that modification can be done with `auto& elem : coll`)
    - iterator based loops
    - removal during iteration may invalidate the end iterator for comparison
- types:
    - size_type
    - difference_type
    - value_type
    - reference
    - const_reference
    - iterator
    - const_iterator
    - pointer
    - const_pointer

Arrays

Vectors

- operations:
    - push_back
    - pop_back
    - insert(pos, elem) + more
    - emplace(pos, args...)
    - erase(pos)
    - erase(beg, end)
    - resize(num)
    - resize(num, elem)
    - clear
- use as c-style arrays:
    - can access memory with &v[index]

Deques

Lists

Forward Lists

Sets and Multisets

Maps and Multimaps

Unordered Containers

Other STL Containers

Implementing Reference Semantics

When to Use Which Container

STL Container Members in Detail

Type Definitions

- value_type
  - type of elements
  - const for sets and multisets
  - pair<key_type, mapped_type> for maps and multimaps
  - a v d l fl s ms m mm us ums um umm str
- reference
  - type of element references
  - usually value_type&
  - aux class for vector
  - a v d l fl s ms m mm us ums um umm str
- const_reference
  - read only ref

    - const value_type&
    - vector -> bool
    - a v d l fl s ms m mm us ums um umm str
- iterator
    - a v d l fl s ms m mm us ums um umm str
- const_iterator
    - a v d l fl s ms m mm us ums um umm str
- reverse_iterator
    - a v d l s ms m mm str
- const_reverse_iterator
    - a v d l s ms m mm str
- pointer
    - type of pointer to element
    - a v d l fl s ms m mm us ums um umm str
- const_pointer
    - a v d l fl s ms m mm us ums um umm str
- size_type
    - unsigned integral type for size values
    - a v d l fl s ms m mm us ums um umm str
- difference_type
    - signed integral type for difference values
    - a v d l fl s ms m mm us ums um umm str
- key_type
    - type of the key of the elements for associative and unordered containers
    - for set types it is equivalent to value_type
    - s ms m mm us ums um umm
- mapped_type
    - type of value part for associative and unordered containers
    - m mm um umm
- key_compare
    - type for value comparator for keys for associative containers
    - s ms m mm
- value_compare
    - type for value comparator for elements for associative containers
    - for sets and multisets equivalent to key_compare
    - maps and multimaps -> aux class that compares only key part
    - s ms m mm
- hasher
    - hashing function type for unordered containers
    - us ums um umm

- key_equal
    - key equality predicate type for unordered containers
    - us ums um umm
- local_iterator
    - bucket iterators for unordered containers
    - since C++11
    - us ums um umm
- const_local_iterator
    - const bucket iterator
    - us ums um umm

Create, Copy, and Destroy Operations

- ()
    - default, empty container
    - for arrays, creates a container of size with elements of undefined values
    - a v d l fl s ms m mm us ums um umm str
- (const CompFunc& cmpPred)
    - cmpPred used for sorting
    - must use strict weak ordering
    - s ms m mm
- (size_type bnum)
- (size_type bnum, const Hasher& hasher)
- (size_type bnum, const Hasher& hasher, const KeyEqual& eqPred)
    - number of buckets, hashing function, and equality predicate
    - us ums um umm
- (initializer-list)
    - since C++11
    - implicitly defined for arrays
    - a v d l fl s ms m mm us ums um umm str
- (initializer-list, const CompFunc& cmpPred)
- (initializer-list, size_type bnum)
- (initializer-list, size_type bnum, const Hasher& hasher)
- (initializer-list, size_type bnum, const Hasher& hasher, const KeyEqual& eqPred)
    - combines init list and options above
    - us ums um umm
- (const container& c)
    - copy ctor
    - calls copy ctor for every element in the container

    - a v d l fl s ms m mm us ums um umm str
- (container&& c)
    - move ctor
    - moves elements of c into new container
    - c is valid but unspecified
    - a v d l fl s ms m mm us ums um umm str
- (size_type num)
    - creates container of size num
    - elements are default constructed
    - v d l fl
- (size_type num, const T& value)
    - same as above but copies value into elements
    - v d l fl
- (InputIterator beg, InputIterator end)
    - creates container defined by range
    - a v d l fl s ms m mm us ums um umm str
- (InputIterator beg, InputIterator end, const CompFunc& cmpPred)
    - see above s ms m mm
- (InputIterator beg, InputIterator end, size_type bnum)
- (InputIterator beg, InputIterator end, size_type bnum, const Hasher& hasher)
- (InputIterator beg, InputIterator end, size_type bnum, const Hasher& hasher, const KeyEqual& eqPred)
    - see above
    - s ums um umm
- dtor
    - removes elements and frees memory
    - calls dtor for each element
    - a v d l fl s ms m mm us ums um umm str

Non-modifying Operations

Size Operations

- empty() const
    - contains no elements
    - equivalent to begin() == end() but may be faster
    - constant
    - a v d l fl s ms m mm us ums um umm str
- size() const
    - number of elements

    - constant
    - a v d l fl s ms m mm us ums um umm str

Comparison Operations

- ==, !=
    - same number of elements and same elements, same orderings (except for unordered containers)
    - linear, quadratic at worst for unordered containers
    - a v d l fl s ms m mm us ums um umm str
- <, <=, >, >=
    - uses lexicographical_compare
    - linear
    - a v d l fl s ms m mm str

Non-modifying Operations for Associative and Unordered Containers

- count(const T& value) const
    - num elements equal to value
    - logarithmic
    - s ms m mm us ums um umm
- find(const T& value)
    - return iterator to position of first matching element, end otherwise
    - logarithmic, constant for unordered containers with a good hash function
    - s ms m mm us ums um umm
- lower_bound(const T& value)
    - first position for insertion of a copy of value according to the sorting criterion, end otherwise
    - logarithmic
    - s ms m mm
- upper_bound
    - see above
- equal_range(const T& value)
    - pair with 1st and last positions of insertion of a copy of value according to sorting criterion
    - equivalent to make_pair(lower_bound(value), upper_bound(value))
    - logarithmic
    - s ms m mm us ums um umm

Assignments

- = (const container& c)
- = (container&& c)
- = (initializer-list)
- .assign(initializer-list)
- array::fill(const T& value)
- assign(size_type num, const T& value)
- swap

Direct Element Access

- at(idx)
  - returns reference that may get invalidated later
  - throws out_of_range
  - a v d str
- operator at(const key_type& key)
  - retrieves value associated with key
  - m um
- operator
  - returns reference that may get invalidated later
  - throws out_of_range
  - may be faster than other options
  - a v d str
- operator
  - returns value associated with key
  - if key does not exist, creates key with default initialized value
  - m um
- front()
  - first element (idx 0)
  - a v d l fl str
- back()
- data()
  - returns c-style array
  - for arrays and vectors since C++11
  - a v str

Operations to Generate Iterators

```
array                random access
vector               random access
deque                random access
list                 bidirectional
forward list         forward
set                  bidirectional, constant element
multiset             bidirectional, constant element
map                  bidirectional, constant key
multimap             bidirectional, constant key
unordered_set        forward, constant element
unordered_multiset   forward, constant element
unorderd_map         forward, constant key
unordered_multimap   forward, constant key
string               random access
```

- begin
- cbegin
- end
- cend
- rbegin
- rend

Inserting and Removing Elements

Inserting Single Elements

- void container::insert(init-list)
  - inserts copies
  - references remain valid
  - associative containers iterators remain valid
  - unordered containers - iterators remain valid if no rehashing is forced
  - s ms m mm us ums um umm
- iterator container::insert(const T& value) Inserts copies of the elements of initializer-list at the position of iterator pos.
  - returns the position of the first inserted element or pos if initializer-list is empty.
  - for vectors, this operation invalidates iterators and references to other elements if reallocation happens (the new number of elements exceeds the previous capacity).
  - for deques, this operation invalidates iterators and references to other elements.
  - for lists, the function either succeeds or has no effect.
  - v d l s

- iterator container::insert(T&& value)
- pair<iterator,bool> container::insert(const T& value)
- pair<iterator,bool> container::insert (T&& value)
  - inserts into assoc. or unordered container
  - lvalue refs are copied
  - rvalue refs are moved from
  - TODO: restart here
  - a v d l fl s ms m mm us ums um umm
- emplace(args)
  - a v d l fl s ms m mm us ums um umm
- insert(iterator pos, value)
  - a v d l fl s ms m mm us ums um umm str
- emplace(iterator pos, args)
  - v d l
- emplace_hint(iterator pos, args)
  - no effect for containers that allow duplicate elements
  - s ms m mm us ums um umm
- push_front(value)
  - d l fl
- emplace_front(args)
  - d l fl
- push_back(value)
  - v d l str
- emplace_back(args)
  - v d l

Inserting Multiple Elements

- void container::insert (initializer-list)

  - inserts copies of the elements of initializer-list into an associative container
  - for all containers, references to existing elements remain valid
  - for associative containers, all iterators to existing elements remain valid
  - for unordered containers, iterators to existing elements remain valid if no rehashing is forced (if the number of resulting elements is equal to or greater than the bucket count times the maximum load factor)
  - s ms m mm us ums um umm

- iterator container::insert (const_iterator pos, initializer-list)

  - inserts copies of the elements of initializer-list at the position of iterator pos
  - returns the position of the first inserted element or pos if initializer-list is empty

    - for vectors, this operation invalidates iterators and references to other elements if reallocation happens (the new number of elements exceeds the previous capacity)
    - for deques, this operation invalidates iterators and references to other elements
    - for lists, the function either succeeds or has no effect.
    - v d l str

- iterator container::insert (const_iterator pos, size_type num, const T& value)

    - inserts num copies of value at the position of iterator pos
    - returns the position of the first inserted element or pos if num==0 (before c++11. nothing was returned)
    - for vectors, this operation invalidates iterators and references to other elements if reallocation happens (the new number of elements exceeds the previous capacity)
    - for deques, this operation invalidates iterators and references to other elements
    - t is the type of the container elements. thus, for maps and multimaps, it is a key/value pair
    - for strings, value is passed by value
    - for vectors and deques, if the copy/move operations (constructor and assignment operator) of the elements don't throw, the function either succeeds or has no effect. for lists, the function either succeeds or has no effect
    - before c++11, type iterator was used instead of const_iterator and the return type was void
    - v d l str

- void container::insert (InputIterator beg, InputIterator end)

    - inserts copies of all elements of the range [beg,end) into the associative container
    - this function is a member template
    - the elements of the source range may have any type convertible into the element type of the container
    - for all containers, references to existing elements remain valid
    - for associative containers, all iterators to existing elements remain valid
    - for unordered containers, iterators to existing elements remain valid if no rehashing is forced (if the number of resulting elements is equal to or greater than the bucket count times the maximum load factor)
    - the function either succeeds or has no effect, provided that for unordered containers the hash function does not throw
    - s ms m mm us ums um umm

- iterator container::insert (const_iterator pos, InputIterator beg, InputIterator end)

    - inserts copies of all elements of the range [beg,end) at the position of iterator pos

    - returns the position of the first inserted element or pos if beg==end (before c++11, nothing was returned)
    - this function is a member template
    - the elements of the source range may have any type convertible into the element type of the container
    - for vectors, this operation invalidates iterators and references to other
    - elements if reallocation happens (the new number of elements exceeds the previous capacity)
    - for vectors and deques, this operation might invalidate iterators and references to other elements
    - for lists, the function either succeeds or has no effect
    - before c++11, type iterator was used instead of const_iterator and the return type was void
    - v d l str

Removing Elements

size_type container::erase(const T& value) - removes all elements equivalent to value from an associative or unordered container - returns the number of removed elements - calls the destructors of the removed elements - T is the type of the sorted value: - for (unordered) sets and multisets, it is the type of the elements - for (unordered) maps and multimaps, it is the type of the keys - the function does not invalidate iterators and references to other elements - the function may throw if the comparison test or hash function object throws - for (forward) lists, remove() provides the same functionality - for other containers, the remove() algorithm can be used - provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap. - s ms m mm us ums um umm

- iterator container::erase (const_iterator pos)

    - removes the element at the position of iterator pos
    - returns the position of the following element (or end())
    - calls the destructor of the removed element
    - the caller must ensure that the iterator pos is valid. for example: coll.erase(coll.end ()); // error -> undefined behavior
    - for vectors and deques, this operation might invalidate iterators and references to other elements
    - for all other containers, iterators and references to other elements remain valid
    - for vectors, deques, and lists, the function does not throw
    - for associative and unordered containers, the function may throw if the comparison test or hash function object throws
    - before C++11, the return type was void for associative containers, and type iterator was used instead of const_iterator

    - for sets that use iterators as elements, calling erase() might be ambiguous since C ++11
    - C++11 currently gets fixed to provide overloads for both erase(iterator) and erase (const_iterator)
    - v d l s ms m mm us ums um umm str

- iterator container::erase (const_iterator beg, const_iterator end)

    - removes the elements of the range [beg,end)
    - returns the position of the element that was behind the last removed element on entry (or end())
    - as always for ranges, all elements, including beg but excluding end, are removed
    - calls the destructors of the removed elements
    - the caller must ensure that beg and end define a valid range that is part of the container
    - for vectors and deques, this operation might invalidate iterators and references to other elements
    - for all other containers, iterators and references to other elements remain valid
    - for vectors, deques, and lists the function does not throw
    - for associative and unordered containers, the function may throw if the comparison test or hash function object throws
    - before C++11, the return type was void for associative containers and type iterator was used instead of const_iterator
    - v d l s ms m mm us ums um umm str

- void container::pop_front()

    - removes the first element of the container
    - is equivalent to
        - container.erase(container.begin())
        - container.erase_after(container.before_begin()) for forward lists
    - if empty, the behavior is undefined
    - thus, the caller must ensure that the container contains at least one element (! empty())
    - the function does not throw
    - iterators and references to other elements remain valid
    - d l fl

- void container::pop_back()

    - removes the last element of the container
    - is equivalent to
        - container.erase(prev(container.end()))

    - if empty, the behavior is undefined
    - the caller must ensure that the container contains at least one element (!empty())
    - the function does not throw
    - iterators and references to other elements remain valid
    - for strings, it is provided since c++11
    - v d l str

- void container::clear()

    - removes all elements (empties the container)
    - calls the destructors of the removed elements
    - invalidates all iterators and references to elements of the container
    - for vectors, deques, and strings, it even invalidates any past-the-end-iterator, which was returned by end() or cend()
    - the function does not throw (before c++11, for vectors and deques, the function could throw if the copy constructor or assignment operator throws)
    - v d l fl s ms m mm us ums um umm str

Internationalization

Numerics

Concurrency

Allocators