```
--------------------------------------------------------------------------------
================================================================================
HACKING C++
https://hackingcpp.com/cpp/beginners_guide.html
================================================================================

--------------------------------------------------------------------------------
Standard Library - Part 1
--------------------------------------------------------------------------------


    ----------------------------------------------------------------------------
    Iterators
    ----------------------------------------------------------------------------

        - Objects that "point to a location"
            - May point to a readable memory address/object
        - Used to:
            - Iterate over container elements in a data-layout-agnostic way
            - Specify positions and ranges in containers
                - For insertion, deletion, ...

        - Note:
            - The @name notation is used to denot an iterator object/parameter
            - @ i neither an allowed operator nor does it have any other
              meaning in C++


    - Default Iterators

        - Obtainable from standard containers, either:
            (1) with member functions

                    container.begin()   -> @first_element
                    container.end()     -> @one_behind_las_element

            (2) with free-standing functions (C++11)

                    std::begin(container)   -> @first_emelent
                    std::end(container)     -> @one_behind_las_element

        - An iterator refers to a position in a container

                    vector<int> v {1,2,3,4,5,6,7};
                    auto i = begin(v);
                    auto e = end(v);

                            |1|2|3|4|5|6|7| |
                             ^             ^
                             i             e

        - *i accesses the lement at i's position

                    cout << *i;             // prints 1
                    cout << *(i+2);         // prints 3
                    cout << *e;             // UNDEFINED BEHAVIOR

        - The 'end' iterator
            - Is only intended to be used as position indicator
                - Must not be used to access an element!

        - Forward/backward

                ++i ..goes 1 toward end
                --i ..goes 1 towards begin

                i += 2  ..goes forward by 2
                i -= 3  ..goes backwards by 3
```

```
         - Example

                 i += 5;
                 if (i == end(v))
                     cout << "at end";


    - Reverse Iterators

       - Obtainable from (many, but not all) standard containers, either:
           (1) with member functions:

                 container.rbegin()      -> @last_element
                 container.rend()        -> @one_behind_las_element

           (2) with free-standing funtions (C++11)

                 std::rbegin(container)  -> @last_element
                 std:rend()              -> @one_behind_las_element

       - A reverse iterator refers to a position in a container

                 vector<int> v {1,2,3,4,5,6,7};
                 auto i = rbegin(v);
                 auto e = rend(v);

                         | |1|2|3|4|5|6|7|
                          ^             ^
                          e             i

       - *i accesses the lement at i's position

                 cout << *i;             // prints 7
                 cout << *(i+2);         // prints 5
                 cout << *e;             // UNDEFINED BEHAVIOR

       - The 'rend' iterator
           - Is only intended to be used as position indicator
               - Must not be used to access an element!

       - Forward/backward

               ++i ..goes 1 toward begin
               --i ..goes 1 towards end

               i += 2  ..goes backwards by 2
               i -= 3  ..goes forward by 3

         - Example

                 i += 5;
                 if (i == rend(v))
                     cout << "at end";

       - Returning a corresponding normal (non-reverse) iterator

                 ri.base()

         - Reverse position = normal  position -1
         - Normal position  = reverse position +1

                 vector<int> v {1,2,3};
                 auto re = rbegin(v);
                 auto fw = re.base();


                 re     v
```

```
                    |1|2|3| |
                  fw       ^
```

- Iterator-Based Loops

  - Forward Direction
    + works for all standard sequence containers
    - out-of-bounds access bugs possible
    - verbose

```
std::vector<int> v {1,2,3,4,5,6};
for (auto i = begin(v); i != end(v); ++i) {
    cout << *i;
}
```

  - Reverse Direction
    + works for all bidirectional containers
    - out-of-bounds access bugs possible
    - verbose

```
std::vector<int> v {1,2,3,4,5,6};
for (auto i = rbegin(v); i != rend(v); ++i) {
    cout << *i;
}
```

- Example: Swap Adjecent Pairs

```
void swap_adjacent_pairs (std::vecot<int>& v) {
    if (v.size() < 2) return;
    for (auto i=begin(v), j=i+1, e=end(v); j < e; i+=2, j+=2) {
        std::swap(*i,*j);
    }
}

vector<int> v {1,2,3,4,5,6};         // 1 2 3 4 5 6
swap_adjacent_pairs(v);              // 2 1 4 3 6 5
```

- Iterator Ranges

  = pair p,q of iterators

```
      begin(v)+3 begin(v)+6
             |p   q|
             v     v
    v |1|2|3|4|5|6|7|8|9| |
       0 1 2|3 4 5|6 7 8
       ^    |_____|      ^
    begin(v)            end(v)
```

    - IMPORTANT:
      - The end-of-range iterator q points ONE BEHIND the last
        element in the range!

```
std::vector<int> v {0,1,2,3,4,5};
                                  ____________
begin(v),   end(v)               |            |
                    6 elements   |0|1|2|3|4|5| |
                                  ^           ^
                                      ______
begin(v)+2, being(v)+5               |      |
                    3 elements   |0|1|2|3|4|5| |
                                      ^     ^
                                      _
begin(v)+2, begin(v)+3               | |
                    1 element    |0|1|2|3|4|5| |
```

```
                                                  ^ ^

                 begin(v)+2, begin(v)+2
                               empty range |0|1| 2 |3|4|5| |
                                                  ^ ^
```

- Used for specifying ranges of elements to be:

    - erased from a container

```
        std::vector<int> v {1,2,3,4,5,6,7,8,9};
        v.erase(begin(v)+3, begin(v)+6)

            |1|2|3|4|5|6|7|8|9| -> |1|2|3|7|8|9|
```

    - inserted into a container
    - assigned to a container
    - processed by a standard algorithm
    - ...

- Element Count of Iterator Range

```
    distance(@begin,     @end)  -> number of elements
                 v         v
            |0|1|2|3|4|5|6|7|8| ->  5
```

- Returns the size of an iterator range

```
distance(@range_begin, @element_in_range) -> index_of_element_in_range
```

- Example

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator> // std::distance

std::vector<int> {0,1,2,3,4,5,6,7,8};

// size of subrange
auto n = distance(begin(v)+2, begin(v)+7);  // int n = 5

// size of entire container
auto m = distance(begin(v), end(v));        // int m = 9


std::vector<int> w {4,5,1,9,8};
// get index of smallest element in w:
auto argmin = distance(begin(w), min_element(begin(w),end(w)) );
// int argmin = 2
```

- TIP:
    - Avoid using 'distance' with iterators into non-random access containers
        - e.g., std::list
        - The runtime will be proportional to the size of the input range!

Standard Sequence Containers

```
    array<T,size>       |a|1|2|3|4|5|6|

                        fixed-size contiguous array;
```

```
vector<T>            v -> |1|2|3|4|5|6|

                     dynamic contiguous array;
                     amortized O(1) growth strategy;
                     C++'s "defautl" container;

deque<T>             d -> |1|2|<->|3|4|5|<->|6|

                     double-ended queue;
                     fast isntert/erase at both ends;

                           v-------------------------------+
list<T>              l -> |1|<->|2|<->|3|<->|4|<->|5|<->|end|
                           +-------------------------------^

                     doubly-linked list;
                     O(1) instert, erase & splicing;
                     in practice often slower than vector;

forward_list<T>      fl -> |1|->|2|->...->|6|->|end|

                     singly-linked list;
                     O(1) isnert, erase & splicing;
                     needs less memory than 'list';
                     in practice often slower than vector;
```

Common Features

- Regularity: Copy, Assign, Compare

    - All standard sequence containers are regular types:
        - deeply copyable
                - copying creates a new container object and copies all contained values
        - deeply assignable
                - all contained objects are copied from source to assignment target
        - deeply comparable
                - comparing two containers compares the contained objects
        - deeply ownership
                - destroying the container destroys all contained objects

    - Example

```
std::vector<int> a {4,7,1,9};
std::vector<int> b {1,2,3};

bool equal1 = (a==b);       // false

b = a;                      // copy assignment -> b: 4 7 1 9
bool equal2 = (a==b);       // true

a[0] = 3;                   // a: 3 7 1 9; b: 4 8 1 9
bool equal3 = (a==b);       // false

// different ways of making exact copies,
// i.e., copy-constructing new containers:
std::vector<int> a2 {a};
std::vector<int> a3 (a);
std::vector<int> a4 = a;
auto a5 {a};
auto a6 (a);
auto a7 = a;
```

- Type Argument Deduction (C++17)

```
      - As of C++17 the element type can be deduced from constructor calls

              std::vector v {1, 2 ,3 ,4};         // std::vector<int>
              std::vector v {1.f, 2.3f, 4.5f};    // std::vector<float>
              std::vector v {1., 2.3, 4.6};       // std::vector<double>

              struct p2 { int x; int y};
              std::vector v {p2{1,2}};            //std::vector<p2>


  - Common Interface Parts

      - Iterators for Forward Traversal
          - can be obtained from all standard sequence containers either with:

              (1) member functions:

                      container.begin() -> @first_element
                      container.end()   -> @one_behind_last_element

              (2) free-standing functions (C++11):

                      std::begin(container) -> @first_element
                      std::end(container)   -> @one_behind_last_element

      - Const Iterators for Forward Traversal
          - can be obtained from all standard sequence containers either with:

              (1) member functions:

                      container.cbegin() -> @first_element
                      container.cend()   -> @one_behind_last_element

              (2) free-standing functions (C++11):

                      std::cbegin(container) -> @first_element
                      std::cend(container)   -> @one_behind_last_element

      - Emptiness Query
          - either with:

              (1) member function

                      container.empty() -> true, if container has no elements

              (2) free-standing frunction (C++11)

                      std:empt(container) -> true, if container has no element

      - Type Interface

                  container::value_type
                  container::size_type
                  container::iterator
                  container::const_iterator
                  ...

          - Example

                  using con_t = std::vector<double>;
                  con_t::size_type i = 0;             // std:size_t
                  auto x = con_t::value_type{0};      // double


  array<T, size>

          + overhead-free random access
```

```
        + fast traversal; good for linear searches
        * 'size' has to be constant expression (= known at compile time)
        * does not support size-changing operations (resize, insert, ...)
        - potentially slow if element type has high copy/assignment cost
          (reordering elements requires copying/moving them)

    - Example

            #include <array>

            std::array<int,6> a {4,8,15,16,23,42};
            cout << a.size();   // 6
            cout << a[0];       // 4
            cout << a[3];       // 16
            cout << a.front();  // 4
            cout << a.back();   // 42

            std::array<int,3> b {7,8,9};
            a = b;              // COMPILER ERROR: types don't match


vector<T>

    - C++'s "default" container

        + overhead-free random access
        + fast traversal; good for linear searches
        + insertion at the end in amortized constant time
        - potentially slow if insert/erase operations at front and/or
          random positions dominate
        - potentially slow if element type has high copy/assignment cost
          (reordering elements require coyping/moving them)
        - all operations that can change capacity (insert, push_back, ...)
          may invalidate references/pointers to any vector element
        - potentially long allocation times for very large amount of values
            -> can be mitigated:

        https://hackingcpp.com/cpp/recipe/uninitialized_numeric_array.html


- Quick Recap

            #include <vector>

            std::vector<int> v {2,4,5};     // 2 3 5
            v.push_back(6);                 // 2 4 5 6
            v.pop_back();                   // 2 4 5
            v[1] = 3;                       // 2 3 5

            cout << v[2];                       // prints 5
            for (int x : v) cout << x << ' ';   // prints 2 3 5

            v.reserve(8);                   // 2 3 5 _ _ _ _ _
            v.resize(5, 0);                 // 2 3 5 0 0 (0) _ _
            cout << v.capacity();           // prints 8
            cout << v.size();               // prints 5

             .capacity() -> 8
            +----------------+
            |.size() -> 5    |
            +---------+      |
            |_________|_____|__ contiguous buffer
            |2|3|5|0|0|?|?|?|X| on the HEAP
             ^         ^     ^  -----------------
             |.begin() |     |
             |         |.end |
            +|---------|-----|----+
            |*         *     * w  | STACK
```

```
               +---------------------+


    - Iterator Ranges
    - Insert Elements

        - Insert At The End (Fastest)

               push_back(element)

            - Example

               vector<int> v {1,2,3};      // 1 2 3
               v.push_back(4);             // 1 2 3 4

        - Insert Anywhere (Potentially Slow)
            - insert positions are specified with iterators:

               .insert(@insert_pos, element)
               .insert(@insert_pos, {elem1,elem2,...})

            - Example

               vector<int> v {1,2,3};          // 1 2 3

               v.insert(begin(v)+1, 5);        // 1 5 2 3
               v.insert(begin(v), {7,8});      // 7 8 1 5 2 3
               v.insert(end(v), 9);            // 7 8 1 5 2 3 9

        - Insert From Iterator Range

               .insert(@insert_pos, @first, @last)

            - Example

               vector<int> y {4,5,6,7,8};
               vector<int> x {1,2};
               x.insert(begin(x),
                        begin(y), begin(y)+3);     // 4 5 6 1 2


    - Insert & Construct Elements In-Place (C++11)

               .emplace_back(arg1, arg2, ...)
               .emplace(@insert_pos, arg1, arg2, ...)

        - makes a new element with forwarded constructor arguments
        - Example

               struct p2s {                    // custom
                   p2d(int x_, int y_):        // <- constructor
                       x{x_}, y{y_} {}         // necessary for
                   int x, y;                   // emplace(_back)
               };
               vector<p2d> v { p2d{2,3} };     // 2 3

               // insert copy
               v.push_back( p2d{6,4} );        // 2 3 | 6 4

               // construct in place with
               // constructor arguments: 9,7
               v.emplace_back(9,7);            // 2 3 | 6 4 | 9 7

               // iterator to first pos (begin...)
               v.emplace(begin(v), 5,8);       // 5 8 | 2 3 | 6 4 | 9 7


    - Erase Elements
```

```
      - Erase At The End (Fast)

              vector<int> v {1,2,3,4,5,6};    // 1 2 3 4 5 6
              v.pop_back();                   // 1 2 3 4 5 _

      - Erase Everything (Fast)

              vector<int> v {1,2,3,4,5,6};    // 1 2 3 4 5 6
              v.clear();                      // _ _ _ _ _ _

      - Erase Anywhere (Potentially Slow)

              .erase(@postition)
              .erase(@range_begin, @range_end)

          - Example

              vector<int> v {1,2,3,4,5,6};        // 1 2 3 4 5 6
              v.erase( begin(v)+2 );              // 1 2 4 5 6 _

              vector<int> v {1,2,3,4,5,6};        // 1 2 3 4 5 6
              v.erase( begin(v)+1, begin(v)+4 )   // 1 5 6 _ _ _

      - NOTE:
          - Erasing does not affect the capacity
              - i.e., none of the vector's memory is freed


  - Shrink The Capacity / Free Memory

      - May work:

              .shrink_to_fit()

          - ISO standard does not demand that it actually shrinks
          - standard library implementation might decide not to shrink

              vector<int> v;
              // add elements
              // erase elements
              v.shrink_to_fit();      // C++11

      - Guaranteed to work:

          - Procedure

              (1) make temporary copy -> copy does exactly fit the elements
              (2) exchange memory buffers by swapping/moving
              (3) temporary gets automatically destroyed

          - Example

              vector<int> v;
              // add elements
              // erase elements

              // shrink: make new copy and replace v's content with it
              v = vector<int>(v);     // C++11-20

              // or:
              v.swap( vector<int>(v) );   // C++98-20


  - Attention After Insert/Erase!

      - All iterators into a vector are invalidated if either:
          (1) its capacity is changed or
```

(2) elements are moved by:

- insert	- =
- push_back	- assign
- emplace	- resize
- emplace_back	- reserve
- erase	

- Note:
  - Swapping two vector’s contents does not invalidate iterators
    - Except for the ’end’ iterator

- Example

```
vector<int> v {1,2,3,4,5,6}
auto i = begin(v) + 3;
```

  - Dangerous
    - use of iterator after insert/erase:

```
v.insert(i,8);
cout << *i;     // dangerous

v.erase(i);
cout << *i;     // dangerous
```

  - Correct
    - use new valid iterator returned by insert/erase
    - the returned iterator refers to the original position

```
i = v.insert(i,8);
cout << *i;

i = v.erase(i);
cout << *i;
```

- Overview: Iterator Invalidating Operations

Operations	Invalidated Iterators
’const’ (ro) operations	none
swap, std::swap	only end()
reserve, shrink_to_fit	if capacity changed: all else: none
push_back, emplace_back	if capacity changed: all else: only end()
insert, emplace	if capacity changed: all else: only at or after insertion point (incl. end())
resize	if capactiy changed: all else: only end() and iterators to erased elements
pop_back	iterators to erased element and end()
erase	iterators to erased elements and all after them (incl. end())
clear, operator=, assign	all

```
    deque<T>

    - Double Ended Queue

                d -> |1|2| <-> |3|4|5| <-> |6|

            + constant-time random access (extremely small overhead)
            + fast traversal; good for linear searches
            + good insertion and deletein performance at both ends
            + insertion does not invalidate references/pointers to elements
            - potentially slow if insert/erase operations at random positions
              dominate
            - potentially slow if element type has high copy/assignment cost
              (reordering elements requires copying/moving them)
            - potentially long allocation times for very large amount of values;
              can be mitigated:
              https://hackingcpp.com/cpp/recipe/uninitialized_numeric_array.html

        - Example

                #include <deque>

                std::deque<int> d {0,0,0};          // 0 0 0
                d.push_back(1);                     // 0 0 0 1
                d.push_front(2);                    // 2 0 0 0 1

                vector<int> v {3,4,5,6};            // v: 3 4 5
                d.insert(begin(d),
                         begin(v), end(v);          // d: 3 4 5 6 2 0 0 0 1

                d.pop_front();                      // 4 5 6 2 0 0 0 1
                d.erase(begin(d)+2, begin(d)+5);    // 4 5 0 0 1


    list<T>

    - Doubly-linked List

                  v------------------------------+
                l -> |1|<->|2|<->|3|<->|4|<->|5|<->|end|
                  +------------------------------^

            + restructuring operations don't require elements to be moved/copied
              (good for stroing large objects with high copy/assignment cost)
            + constant-time splicing (of complete lists)
            - random access only in linear time
            - slow traversal due to bad memory locality

        - Example

                #include <list>

                std::list<int> l {3};       // 3

                l.push_back(2);             // 3 <-> 2
                l.push_front(4);            // 4 <-> 3 <-> 2
                l.splice(begin(l)+1,
                         list<int>{8,4,7}); // 4 <-> 8 <-> 4 <-> 7 <-> 3 <-> 2
                l.reverse();                // 2 <-> 3 <-> 8 <-> 4 <-> 7 <-> 4
                l.sort();                   // 2 <-> 3 <-> 4 <-> 4 <-> 7 <-> 8
                l.unique();                 // 2 <-> 3 <-> 4 <-> 7 <-> 8


    forward_list<T>

    - Singly-linked List

                fl -> |1|->|2|-> ... ->|6|->|end|
```

        + uses less memory than 'list'
        + restructuring operations don't require elements to be moved/copied
          (good for storing large objects with high/assignment cost)
        + constant-time plicing (of complete lists)
        - random access only in linear time
        - slow traversal due to bad memory locality
        - only forward traversal possible
        - somewhat cumbersome interface due to forward-only links:
            -> no: size(), back(), push_back(), pop_back(), insert()
            -> instead: insert_after(), splice_after(), before_begin()

    - Example

```
#inlcude <forward_list>

std::forward_list<int> l {23,42,4};     // 23 -> 42 -> 4

l.insert_after(begin(l), 5);            // 23 -> 5 -> 42 -> 4
l.insert_after(before_begin(l), 88); // 88 -> 23 -> 5 -> 42 -> 4
l.erase_after(begin(l));             // 88 -> 5 -> 42 -> 4
```

Guidelines

- Which Sequence Container Should I Use?

    - Default choice:

                std::vector

    - number of elements to be stored is small and known at compile time:
        - yes
            -> array
        - no
            -> try std::vector

    - std::vector
        - too slow and/or memory usage too high?
            -> measure/profile: identify bottleneck operation(s)
                - insert/erase at begin and end dominate
                    -> try std::deque
                - copying elements and/or insert/erase at random position
                  dominate
                    -> try std::list
                        - memory usage a little too high and reverse
                          traversal not needed
                            -> forward_list
                        - still bad: look for non-standard solution with
                          better characteristics
                - too many memory allocations and/or consumption too high
                    -> .reserve(approx_expected_size) before incrementally
                       filling vector
                        - still bad: look for non-standard solution with
                          better characteristics
                - too slow; non-linear memory accesses dominate and can't be
                  avoided
                        - look for non-standard solution
                          with better characteristics

Sequence Views

- Views Don't Own Resources

    - An object is said to be an owner of a resource (memory, file handle,

```
        connection, ...) if it is responsible for its lifetime


    std::string_view    (C++17)

            #include <string_view>

        - Properties
            - lightweight
                - cheap on copy
                - can be passed by value
            - non-owning
                - not responsible for allocating or deleting memory
            - read-only view
                - does not allow modification of target string
            - of character range or string(-like) object
                - std::string/"literal"/...

        - Primary Use Case:
            - read-only function parameters
                - avoids temporary copies


    - Avoids UNnecessary Memory Allocations

        - Motivation: Read-only String Parameters
            - We don't want/expext additional copies or memory allocations for
              read-only parameter!

        - Traditional choice:

                std::string const&

            - Problematic
                - A std::string can be constructed from string literals or an
                  iterator range to a char sequence
                - If we pass an object as function argument that is not a string
                  itself, but something that can be used to construct a string,
                  e.g., a string literal or an iterator range, a new temporary
                  string object will be allocated and bound to the const
                  reference

        - string_view avoids temporary copies:

            #include <vector>
            #include <string>
            #include <string_view>

            void f_cref (std::string const& s) {...}
            void f_view (std::string_view s)   {...}

            int main () {
                std::string stdStr = "Standard String";
                auto const cStr = "C-string";
                std::vector<char> v {'c','h','a','r','s','\0'};

                f_cref(stdStr);              // no copy
                f_cref(cStr);                // temp copy
                f_cref("Literal");           // temp copy
                f_cref({begin(v),end(v)});   // temp copy

                f_view(stdStr);              // no copy
                f_view(cStr);                // no copy
                f_view("Literal");           // no copy
                f_view({begin(v),end(v)});   // no copy
            }
```

- String-Like Function Parameters

If You ...	Use Parameter Type
..always need a copy of the input string inside the function	std::string "pass by value"
..want read-only access - don't (always) need a copy - are using C++17/20	#inlcude <string_view> std::string_view
..want read-only access - don't (always) need a copy - stuck with C++98/11/14	std::string const& "pass by const reference"
..want the function to modify the input string in-place (try to avoid such "output params"	std::string & "pass by (non-const) ref"

- Making string_viewS

  - With Constructor Calls

```
std::string s = "Some Text";

// view whole string
std::string_view sv1 { s };

// view subrange
std::string_view sv2 {begin(s)+2, begin(s)+5};
std::string_view sv3 {begin(s)+2, end(s)};
```

  - With Special Literal "..."sv

```
using namespace std::string_view_literals;
auto literal_view = "C-String Literal"sv;
cout << literal_view;
```

  - CAREFUL: View might outlive string!

```
std::string_view sv1 {std::string{"Text"}};
cout << sv1;        // string object already destroyed!

using namespace std::string_literals;
std::string_view sv2 {"std::string Literal"s};
cout << sv2;        // string object already destroyed!
```

  - TIP
    - Use string_view mainly as function parameter!

- string_view Interface

```
void foo (std::string_view sv) {...}
     foo ("I'm sorry, Dave.");
```

sv.size()	16 (number of chars)
sv[2]	'm' (char at index 2)
sv.front()	'I' (first char)
sv.back()	'.' (last char)

sv.find("r")	6 (1st match from start)
sv.rfind("r")	7 (1st match from end)
sv.find("X")	string::npos (not found - invalid index)
sv.substr(4,5)	string_view of "sorry"
sv.contains("sorry")	true (C++23)
sv.starts_with('I')	true (C++20)
sv.ends_with("Dave.")	true (C++20)
sv.find_first_of("ems")	2 (1st occurence) (C++17)
sv.find_last_of('r')	7 (last occurence)(C++17)
sv.compare("I'm sorry, Dave.")	0 (identical)
sv.compare("I'm sorry, Anna.")	> 0 (same length, D > A)
sv.compare("I'm sorry, Saul.")	< 0 (same length, D < S)
sv.remove_suffix(7);	-> "I'm sorry"
sv.remove_prefix(4);	-> "sorry"

std::span (C++20)

```
#inlcude <span>
```

- Properties
  - Lightweight
    - cheap to copy
    - can be passed by value
  - Non-owning view
    - not responsible for allocating or deleting memory
  - Of a contiguous memory block
    - e.g., std::vector, std::array, ...

- Primary Use Case:
  - as a function parameter
    - container-independent access to values

- Sequences

span<int>	sequence of integers whose values can be changed
span<int const>	sequence of integers whose values CAN'T be changed
span<int,5>	sequence of exactly 5 integers (numbers of values fixed at compile time)

- Example

```
                w.begin()
                    v
vector<int> w { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
                          ^
                     w.begin()+2
```

```
            span<int> s {w.begin()+2, 5};           // 2 3 4 5 6
```

- As Parameter (Primary Use Case)

```
            void print_ints  (std::span<int const> s);
            void print_chars (std::span<char const> s);
            void modify_ints (std::span<int> s);
```

   - Call With Container/Range

```
            std::vector<int> v {1,2,3,4};
            print_ints( v );

            std::array<int,3> a {1,2,3};
            print_its( a );

            std::string s = "SOme Text",
            print_chars( s );

            std::string_view sv = s;
            print_chars( sv );
```

- NOTE:
   - A 'span' decouples the storage strategy for sequential data from code that only needs to access the elements in the sequence, but not alter its structure

- Explicitly Making Spans

   - As View of Whole Container/Range

```
            std::vector<int>  w {0,1,2,3,4,5,6};
            std::array<int,4> a {0,1,2,3};

            // auto-deduce type/length:
            std::span sw1 { w };        // span<int>
            std::span sa1 { a };        // span<int,4>

            // explicit read-only view:
            std::span sw2 { std::as_const(w) };

            // with explicit type parameter:
            std::span<int>       sw3 { w };
            std::span<int>       sa2 { a };
            std::span<int const> sw4 { w };

            // with explicit type parameter and length:
            std::span<int,4> sa3 { a };
```

   - As View of Container Subsequence

```
            vector<int> w {0,1,2,3,4,5,6};

            std::span s1 {begin(w)+2, 4};
            std::span s2 {begin(w)+2, end(w)};
```

- Size and Data Access

```
            std::span<int> s = ...;

            if (s.emty()) return;
            if (s.size() < 1024) { ... };
```

```
// spans in range-based for loops
for (auto x : s) { ... }

// indexed access
s[0] = 8;
if (s[2] > 0) { ... }

// iterator access
auto m1 = std::min_element(s.begin(), s.end());
auto m2 = std::min_element(begin(s), end(s));
```

- Comparing Spans

```
#include <algorithm>    // std::ranges::equal

std::vector<int> v {1,2,3,4};
std::vector<int> w {1,2,3,4};

std:span sv {v};
std:span sw {w};

bool memory_same = sv.data() == sw.data();      // false
bool values_same = std::ranges::equal(sv,sw);   // true
```

- Making Spans From Spans

```
std::vector<int> v {0,1,2,3,4,5,6,7,8};
std::span s = v;

auto first3elements = s.first(3);
auto last3elements  = s.last(3);

size_t offset = 2;
size_t count = 4;

auto subs = s.subspan(offset, count);
```

Usage Guidelines

- Views As Function Parameters

    + decouple function implementations from the data representation/ container type
    + clearly communicate the intent of only reading/altering elements in a sequence, but not modifying the underlying memory/data structure
    + make it easy to apply functions to sequence subranges
    + can almost never be dangling, because parameters outlive all function-local variables

```
int foo (std::span<int const> in) {...}

std::vector<int> v {...};
// v will always outlive parameter 'in'!
foo(v);
foo({begin(v), 5});
```

    + a view's target cannot invalidate the memory that the view refers to during the function's execution (unless it is done in another thread)
    + views can speed up accesses by avoiding a level of indirection:

```
vector<int> const&  --> vector<int>   --> |1|1|3|5|7|4|1|0|
reference to vector     vector object      ^   dynamic mem block
                                           |
span<int const> ---------------------------+
span object
```

- CAREFUL When Returning Views

    - not always clear what object/memory the view refers to
    - returned views can be (inadvertently) invalidated

        - Example

```
// which parameter is the span's target?
std::span<int const>
foo (std::vector<int> const& x, std::vector<int> const& y);
```

        - Example

```
// we can assume that the returned span
// refers to elements of the vector
std::span<int const>
random_subrange (std::vector<int> const& v);

// however, this is still problematic:
auto s = random_subrange(std::vector<int>{1,2,3,4});
// 's' is dangling - vector object already destroyed!
```

        - Example

```
class Payments { ...
public:
    std::span<Money const> of (Customer const&) const;
    ...
};

Customer const& john = ...;
Payments pms = read_payments(file1);
auto m = pms.of(john);
pms = read_payments(file2);
// depending on the implementation of Payments
// m's target memory might no longer be valid
// after the assignment
```

- AVOID Local View Variables

    - easy to produce dangling views, because we have to manually track
      lifetimes to ensure that no view outlives its target
    - even if the memory owner is still alive, it might invalidate the
      memory that a view is referring to

        - Example

```
std::string str1 = "Text";
std::string_view sv {str1};
if (...) {
    std::string str2 = "Text";
    sv = str2;
}
cout << sv;     // str2 already destroyed
```

        - Example

```
std::string_view sv1 {"C-String Literal"};
cout << sv1;    // ok

std::string_view sv2 {std::string{"Text"}};
cout << sv2;    // string object already destroyed!

using namespace std::string_literals;
std::string_view sv3 {"std::string Literal"s};
cout << sv3;    // string object already destroyed!
```

```
    - Memory Invalidation By Owner
        - Containers like 'vector' might allocate new memory thus
          invalidating all views of it:

            std::vector<int> w {1,2,3,4,5};
            std::span s {w};
            w.push_back({6,7,8,9});
            cout << s[0];   // w might hold new memory


----------------------------------------------------------------------------
Standard Associative Containers
----------------------------------------------------------------------------


Quick Overview

- Sets

        (1) Ordered Sets

                #include <set>

        (2) Hash Sets

                #include <unordered_set>

    set<Key> / unordered_set<Key>

        - unique orderable / hashable keys

            std::set<int> s {9,2,8};            // {2,8,9}

            s.insert(7);                        // {2,7,8,9}
            s.earse(8);                         // {2,7,9}

            if (s.find(7) != end(s)) {---}      // true
            if (s.contains(7)) {...}            // true (C++20)

            // find returns an iterator:        // {2,7,9}
            auto i s.find(7);                   //    ^i
            if (i != end(s)) i = s.erase(i);    // {2,9}
                                                //    ^i (after)

    multiset<Key> / unordered_multiset<Key>

        - mutliple equivalent keys possible

            std::multiset<int> s;               // {}
            s.insert(8);                        // {8}
            s.insert(7);                        // {7,8}
            s.insert(2);                        // {2,7,8}
            s.insert(7);                        // {2,7,7,8}
            s.erase(7);                         // {2,8}


- Key->Value Maps

    - Ordered Key->Value Maps

            #inlcude <map>

    - Hashed Key->Value Maps

            #inlcude <unordered_map>

    - Maps store
```

```
            std::pair<Key const, Value>

        - the standard library associative containers are based on nodes
          that are linked by pointers
            - each node stores a pair of a key and a value

    - std::pair<First,Second>

        - contains two values of different or same type
        - Example

            #include <utility>

            std::pair<int,double> p {4, 8.15};
            cout << p.first  << '\n'            // 4
                 << p.second << '\n';           // 8.15

            // C++17 features:
            std::pair p2 {1, 2.3};              // std::pair<int,double>
            auto [fst,snd] = p2;                // "structured binding"
            cout << fst << " " << snd << '\n';  // 1 2.3

    - map<Key,Value> / unordered_map<Key,Value>

        - unique orderable / hashable keys
        - Example

            std::map<int,std::string> m;        // {}

            m.insert({2, "B"});                 // {2:B}
            m.emplace(1, "A");                  // {1:A,2:B}

            m[2] = "Y";                         // modify: {1:A,2:Y}
            m[3] = "C";                         // insert: {1:A,2:Y,3:C}

            auto i = m.find(2);                 // -> iterator
            if (i != end(m))                    // if found
                cout << i->first                // 2 (key)
                     << i->second;              // Y (value)

            if (m.contains(2)) {...}            // true (2 found) (C++20)

            m.erase(2);                         // {1:A,3:C}
            auto j = m.find(3);                 //      ^j
            if (j != end(m)) j = m.eraase(j);   // {1:A  }
                                                //      ^j (after)
            // C++17 features:                  // {1:A}
            m.insert_or_assign(4,"D");          // {1:A,4:D}
            m.insert_or_assign(1,"X");          // {1:X,4:D}
            m.try_emplace(4,"Z");               // {1:X,4:D}
            m.try_emplace(5,"E");               // {1:X,4:D,5:E}

    - multimap<Key,Value> / unordered_multimap<Key,Value>

        - multiple equivalent keys possible
        - Example

            std::multimap<int,std::string> m;   // {}

            m.emplace(1, "A");                  // {1:A}
            m.insert({2, "B"});                 // {1:A,2:B}
            m.emplace(1, "C");                  // {1:A,1:C,2:B}
            m.erase(1);                         // { 2:B}


  - Standard Sets And Maps Are Node-Based
```

```
    - Keys or key-value pairs are stored in nodes that are "linked" by
      pointers

    - Ordered Sets/Maps
        - usually implemented as balanced binary tree

    - Unordered Sets/Maps
        - implemented as hash table


Interface: How To

- Make New Sets/Maps

    - Make An Empty Set/Map

        SetOrMapType variable;
        SetOrMapType variable {};

            std::set<int> s1;
            std::set<int> s2 {};
            std::map<int,float> m1;
            std::map<int,float> m2 {};

        - CARE!

            std::set<int> s3 ();    // THIS IS A FUNCTION DECLARATION!

    - Make Sets From Key Lists

        set<KeyType>{key1,key2,...}
        set{key1,key2,...}              // C++17 - key type deduced

            std::set<int> s1 {12};
            std::set<int> s2 {3,2,1,4,5};

            std::set s3 {1,2,3,4};          // set<int>     C++17
            std::set s4 {1.f,2.3f,4.5f};    // set<float>   C++17
            std::set s5 {1.,2.3,4.6};       // set<double>  C++17

    - Make Sets From Key Ranges

        set<Key>(@keys_begin,@keys_end)
        set(@keys_begin,@keys_end)      // C++17 - key type deduced

            std:vector<int> v {2,3,1,4};
            std::set<int> s (begin(v), begin(v)+3);


- Inset Keys Into Sets

    - Insert Single Keys

        .insert(key) -> pair<@pos,insert_success>

            std::set<int> s;                // { }
            s.insert(3);                    // {3}
            auto r1 = s.insert(7);          // {3,7}
            cout << r1.second;              // true (inserted)
            cout << *r1.first;              // 7

            auto r2 = s.insert(7);
            cout << r2.second;              // false (NOT inserted)
                                            // why? because of uniqueness

    - Emplace: Insert And Construct Keys In-Place C++11

        .emplace(keyArg1,keyArg2,...) -> <@pos,insert_success>
```

```
        // Construct key(s) directly inside the set using key constructor
        // arguments (can prevent copying of large key objects)

            using VS = std::vector<std::string>;
            std::set<VS> s;                  // { }
            s.insert( VS{"a","c"} );         // {{"a","c"}}
            s.emplace("v","w","x");          // {{"a","c"},{"v","w","x"}}

    - Insert Ranges of Keys

        .insert({key1,key2,...})
        .insert(@keys_begin,@keys_end)

    - Insert/Emplace With Position Hint

        .insert(@hint,key) -> @insert_pos
        .emplace_hint(@hint,keyArg1,keyArg2,...) -> @insert_pos

        // Potentially faster than regular insert: amortized constant cost,
        // if @hint is before/after the key's final position in the set


- Insert Keys+Values Into Maps

    - Insert Single Key-Value Pairs

        .insert({key,value}) -> pair<@pos,insert_success>

        // Copies/moves key-value pairs into the map
        // Use emplace if your key and/or value types are expensive to copy

            std::map<int,std::string> m;        // { }

            auto r1 = m.insert({1,"a"});        // {1:"a"}
            cout << r1.second;                  // true     (inserted)
            cout << *r1.first;                  // "a"      (key @ position)

            auto r2 = m.insert({1,"b"});        // {1:"a"}
            cout << r2.second;                  // false    (NOT inserted)
            cout << *r2.first;                  // "a"      (key @ position)

    - Emplace: Insert & Construct Key-Value Pairs In-Place

        .emplace(key,value) -> pair<@pos,inser_success> C++11
        .try_emplace(key,valArg1,valArg2,...)
                -> pair<@pos,insert_success> C++17

        // Constructs key-value pairs directly inside the map using
        // constructor arguments (prevents copying of large value or
        // key objects)

            struct P2 { // custom value type
                P2 (int x_, int y_): x{x_}, y{y_} {}
                int x, y;
            };

            std::map<char,P2> m;             // { }
            m.insert( {'a', P2{1,2}} );      // {'a':{1,2}}
            m.emplace('c', P2{3,4});         // {'a':{1,2}, 'c':{3,4}}
            m.try_emplace('b', 5,6);     // {'a':{1,2}, 'b':{5,6}, 'c':{3,4}}
            m.try_emplace('a', 8,9);     // {'a':{1,2}, 'b':{5,6}, 'c':{3,4}}

    - Insert / Access / Assign

        [key] = value (inserts key if not found)
        .at(key) = value (throws std::out_of_range if key not found)
        .insert_or_assign(key,value) -> pair<@pos,insert_success> C++17
```

```
        - Insert Ranges of Key-Value Pairs

            .insert (@kv_pairs_begin,@kv_pairs_end)

        - Insert/Emplace With Position Hint

            .insert(@hint,{key,value}) -> @pos
            .emplace_hint(@hint,key,value) -> pair<@pos,insert_success> C++11

            // Potentially faster than regular insert: amortized constant cost,
            // if @hint is before/after the key's final position in the map


    - Find/Access/Count Keys

        - Check If Set/Map Contains Key

            .find(key) != @end -> true, if container has 'key'
            .contains(key) -> true, if container has 'key' C++20

                std::set<int> s {1,3,5,6};

                if (s.find(6) != end(s)) {...}      // true
                if (s.find(7) != end(s)) {...}      // false

                if (s.contains(6)) {...}            // true
                if (s.contains(7)) {...}            // false

        - Find/Get Key Position

            .find(key) -> @position, if 'key' found or @end otherwise

                std::set<int> s {1,3,5,6};          // {1, 3, 5, 6 }
                auto i = s.find(3);                 //     ^i
                if (i != end(s)) {                  // true
                    cout << *i;                     // 3
                    i = s.erase(i);                 // {1, 5, 6 }
                }                                   //     ^i (after)

                auto j = s.find(4);
                if (j != end(s)) {...}              // false

        - Access / Assign (maps only)

            [key] = value (inserts key if not found)
            .at(key) = value (throws std::out_of_range if key not found)

                std::map<int,std::string> m;        // { }
                m[1] = "a";                         // {1:"a"}
                m[2] = "b";                         // {1:"a", 2:"b"}
                m[1] = "x";                         // {1:"x", 2:"b"}

                try {
                    m.at(2) = "y";                  // {1:"x",2:"y"}
                    m.at(3) = "z";                  // <- throws exception
                } catch(std::out_of_range&) {...}

        - COunt Key Occurrences

            .count(key) -> number of occurrences of 'key'


    - Iterate Over Elements

        - It is not possible to modify keys through iterators or in range based
          loops
            - This could break the container invariant: key ordering or position
```

```
              in the hash table

        - Ranged-Based 'for' Loops

            - Sets:

                    for (auto lightKey : mySet) {...}
                    for (auto const& heavyKey : mySet) {...}

            - Maps:

                    for (auto const& keyValuePair : myMap) {...}
                    for (auto const& [key,value] : myMap) {...}

            - Examples

                    // keys that are cheap to copy
                    std::set<int> s1 {...};
                    for (auto key : s1) {...};

                    // keys that are expensive to copy
                    std::set<std::string> s2 {...};
                    for (auto const& key : s2) {...}

                    // C++17: lightweight key-value pairs
                    std::map<int,std::string> m2 {...};
                    for (auto [key,value] : m2) {
                        cout << key <<":"<< value << " ";
                    }

                    // C++17: expensive-to-copy key-value pairs
                    std::map<int,std::string> m2 {...};
                    for (auto const& [key,value] : m2) {
                        cout << key <<":"<< value << " ";
                    }

                    // C++11: lightweight key-value pairs
                    std::map<int,std::string> m1 {...};
                    for (auto kv : m1) {
                        cout << kv.first <<":"<< kv.second << " ";
                    }

                    // C++11: expensive-to-copy key-value pairs
                    std::map<int,std::string> m1 {...};
                    for (auto const& kv : m1) {
                        cout << kv.first <<":"<< kv.second << " ";
                    }

        - Obtain Iterators (Forward Direction)

            - Either with:

                - Member functions:

                    container.begin() -> @first_element
                    container.end()   -> @one_behind_last_element

                - Free-standing function:

                    std::begin(container) -> @first_element (C++11)
                    std::end(container)   -> @one_behind_last_element (C++11)

        - Obtain Iterators (Reverse Direction)

            - Only available for ordered containers
            - Either with:

                - Member functions:
```

```
                    container.rbegin() -> @last_element
                    container.rend()   -> @one_behind_first_element

                - Free-standing function:

                    std::rbegin(container) -> @last_element (C++11)
                    std::rend(container)   -> @one_behind_first_element (C++11)


    - Get Range of Equivalent Keys (available for all sets/maps)

            .equal_range(key) -> pair<@range_begin,@range_end>

        - Follows the usual iterator range convention:
            - @range_end points to one behind the last element in the range

                    std::multiset<int> s {2,4,4,4,6};

                    // 4 is a non-unique key:       // {2,4,4,4,6}
                    auto e4 = s.equal_range(4);     //    ^     ^
                    cout << *(e4.first)             // 4 (first in range)
                         << *(e4.second);           // 6 (1 behind last)
                    auto n = distance(e4.first,     // n: 3 (range size)
                                      e4.second);

                    // 1 is smaller than all:       //  v
                    auto e1 = s.equal_range(1);     // {2,4,4,4,6}
                    // -> empty range               //  ^

                    // 2 is the smallest key:       //  v
                    auto e2 = s.equal_range(2);     // {2,4,4,4,6}
                    // -> range with 1 element      //    ^

                    // 3 is in between keys:        //    v
                    auto e3 = s.equal_range(3);     // {2,4,4,4,6}
                    // -> empty range               //    ^

                    // 6 is the largest key:        //          v
                    auto e6 = s.equal_range(6);     // {2,4,4,4,6 }
                    // -> range with 1 element      //           ^

                    // 7 larger than all:           //           v
                    auto e7 = s.equal_range(7);     // {2,4,4,4,6 }
                    // -> empty range               //           ^


    - Get Upper/Lower Key Bound Positions (ordered sets/maps only)

            .lower_bound(key) -> @first_not_less_than_key
            .upper_bound(key) -> @first_greater_than_key

                    std::multiset<int> s {2,4,4,4,6};

                    // 1 is smaller than all:           // {2,4,4,4,6}
                    auto l1 = s.lower_bound(1);         //  ^
                    auto u1 = s.upper_bound(1);         //  ^
                    if (l1 != end(s)) cout << *l1;      // true -> 2
                    if (u1 != end(s)) cout << *u1;      // true -> 2

                    // 2 is the smallest key:           // {2,4,4,4,6}
                    auto l2 = s.lower_bound(2);         //  ^
                    auto u2 = s.upper_bound(2);         //    ^
                    if (l2 != end(s)) cout << *l2;      // true -> 2
                    if (u2 != end(s)) cout << *u2;      // true -> 4

                    // 3 is in between keys:            // {2,4,4,4,6}
                    auto l3 = s.lower_bound(3);         //    ^
```

```
                  auto u3 = s.upper_bound(3);          //    ^
                  if (l3 != end(s)) cout << *l3;       // true -> 4
                  if (u3 != end(s)) cout << *u3;       // true -> 4

                  // 4 is a non-unique key:            // {2,4,4,4,6}
                  auto l4 = s.lower_bound(4);          //  ^
                  auto u4 = s.upper_bound(4);          //          ^
                  if (l4 != end(s)) cout << *l4;       // true -> 4
                  if (u4 != end(s)) cout << *u4;       // true -> 6

                  // 6 is is the largest key:          // {2,4,4,4,6, }
                  auto l6 = s.lower_bound(6);          //          ^
                  auto u6 = s.upper_bound(6);          //            ^
                  if (l6 != end(s)) cout << *l6;       // true  -> 6
                  if (u6 != end(s)) cout << *u6;       // false (@end)

                  // 7 is larger than all:             // {2,4,4,4,7, }
                  auto l7 = s.lower_bound(7);          //            ^
                  auto u7 = s.upper_bound(7);          //            ^
                  if (l7 != end(s)) cout << *l7;       // false (@end)
                  if (u7 != end(s)) cout << *u7;       // false (@end)


    - Get Size / Query Emptiness

        - Either with

            - Member functions:

                container.empty() -> true, if container has no keys
                container.size()  -> total number of keys or key-value pairs

            - Free-standing functions:

                std::empty(container) -> true|false (C++17)
                std::size(container)  -> #(keys/key-value pairs) (C++17)

            - Example

                  std::map<int,std::string> m;         // { }
                  cout << m.empty();                   // true
                  cout << m.size();                    // 0

                  //m.insert(3,"x");                   // {3:"x"}
                  m.insert( {3,"x"} );
                  cout << m.empty();                   // false
                  cout << m.size();                    // 1

                  //m.insert(1,"y");                   // {1:"y",3:"x"}
                  m.insert( {1,"y"} );
                  cout << m.size();                    // 2


    - Erase Elements/Ranges

        - Erase A Single Key / Key-Value Pair

                .erase(key) -> number of deleted keys/key-value pairs

            - Example

                  std::multiset<int> s {1,3,3,7,9};    // {1,3,3,7,9}
                  auto n = s.erase(3)                  // {1,7,9}
                  cout << n;                           // 2

                  s.erase(7)                           // {1,9}

        - Erase With Iterator (Ranges)
```

```
            .erase(@position)                -> @behind_deleted
            .erase(@range_begin,@range_end) -> @behind_last_deleted

        - Example

                std::set<int> s {1,3,6,8,9};        // {1,3,6,8,9}
                auto i = s.find(3);                 //    ^i
                i = s.erase(i);                     // {1,6,8,9}
                                                    //    ^i (after)

                auto j = s.erase(begin(s),s.find(8));   // {1,6,8,9}
                                                        //  ^   ^
                                                        // {8,9}
                                                        //  ^j

      - Erase Everything

                std::set<int> s {1,2,5,8};          // {1,2,5,8}
                s.clear();                          // { }
                s.empty();                          // true


    - Extract And (Re-)Insert Nodes (C++17)

          .extract(key) -> node
          .extract(@position) -> node
          .insert(node) -> @insert_position

      container::node_type

          - Important Memeber Functions:

              .key()    -> key_type&
              .mapped() -> mapped_type&

      - Transfer Key(-Value Pair) Between Sets/Maps (without copying or moving
        key/value objects)

              set<string> s {"a","b","e"};    // s: {"a","b","e"}
              set<string> t {"x","z"};        // t: {"x","z"}

              t.insert(s.extract("a"));       // s: {"b","e"}
                                              // t: {"a","x","z"}

      - Change Keys Without Copying It

              set<string> s {"a","c","e"};    // {"a","c","e"}
              auto na = s.extract("a");       // get node
              na.key() = "z";                 // change key
              // move node back in
              s.insert(std::move(na));        // {"c","e","z"}

              map<int,string> m {{2."a"},{3,"x"}};    // {2:"a",3:"x"}
              auto n2 = m.extract(2);                 // get node
              //n2.key() = 5;                         // DOES NOT WORK
              n2.value() = "z";                       // change value
              // move node back in
              m.insert(std::move(n2));                // {3:"x",5:"a"}


    - Copy And Assign Entire Sets/Maps

      - Copying
          - Creates a new container object and copies all contained keys and
            values

      - Assigning
```

- All contained objects are copied from source to assignment target

- TIP
  - Copying / copy-assigning sets or maps might be expensive, if they contain a large number of elements

- Example

```
std::set<int> a {4,7,1,9};      // a: {4,7,1,9}
std::set<int> b {1,2,3};        // b: {1,2,3}
bool equal1 = (a == b);         // false

// copy assignment:
a = b;                          // a: {1,2,3}
bool equal2 = (a == b);         // true

b.clear();                      // b: {}  a: {1,2,3}
bool equal3 = (a == b);         // false

// ways of making copies:
std::set<int> a2 {a};           // a2: {1,2,3}
std::set<int> a3 (a);           // a3: {1,2,3}
std::set<int> a4 = a;           // a4: {1,2,3}
auto a5 {a};                    // a5: {1,2,3}
auto a6 (a);                    // a6: {1,2,3}
auto a7 = a;                    // a7: {1,2,3}
```

- Assigning Key Ranges

```
assign(@keys_begin, @keys_end)  // DOES NOT EXIST/WORK?
```

- Compare Entire Sets/Maps

  - Comparisons are value-based:
    - Comparing two associative containers compares keys/key-value pairs

  - Equality
    - Comparison of all set/map types with == and !=
    - Two associative containers are equal, if their key(-valu) content is equal

  - Lexicographical
    - Comparison of ordered set/maps with <, <=, >, >=

  - Examples

```
std::set<int> s1 {1,2,7,8,9};
std::set<int> s2 {1,4,5};

bool s1_equals_s2  = (s1 == s2);    // false
bool s1_unequal_s2 = (s1 != s2);    // true
bool s1_smaller_s2 = (s1 <  s2);    // true
bool s1_greater_s2 = (s1 >  s2);    // false

std::map<int,std::string> m1 {{1,"z"},{2,"f"},{7,"b"}};
std::map<int,std::string> m2 {{1,"a"},{3,"x"}};

bool m1_equals_m2  = (m1 == m2);    // false
bool m1_unequal_m2 = (m1 != m2);    // true
bool m1_smaller_m2 = (m1 <  m2);    // true
bool m1_greater_m2 = (m1 >  m2);    // false
```

- Merge Entire Sets/Maps (C++17)

  - Integrate nodes from source into target without copying keys or values

```
            target.merge(source)

      - Runtime complexity

            source.size() * log(target.size() + source.size())

      - Example

            set<string> s {2,7};
            set<string> t {1,5,8,9};
            t.merge(s)                  // s: {} t: {1,2,5,7,8,9}


   - Inspect/Control Hash Table (unordered sets and maps only)

      - NOTE
         - The exact number of hash buckets as a function of the number of
           inserted elements is not standardized and depends on the standard
           library implementation

      - Check And Control Hash Table Size

         .size()                      -> #elements (#keys/key-value pairs)
         .reserve(#elements)             make space for at least '#elements'
                                         (might trigger rehash)
         .bucket_count()              -> #hash_table_buckets
         .bucket_size(bucket_index)   -> #elements_in_bucket
         .load_factor()               -> #non_empty_buckets / #bucket_count
         .rehash(new_min_bucket_count)   (also takes max_load_factor into
                                          account)

      - Example

            unordered_map<int,string> m {{6,"x"},{4,"a"},{7,"n"},{2,"z"}};
            cout
                << m.size()          // 4
                << m.bucket_count()  // 5
                << m.bucket_size(1)  // 1
                << m.bucket_size(2)  // 0
                << m.bucket_size(3)  // 2
                << m.load_factor();  // 0.6 = 3/5


                         hash collision:      +---+---+---+
                           h(2) = h(4)   ---> | 6 | x |   |
                          +------+---+  /     +---+---+---+
             +--------+ 4 | h(6) | * ---
             | m | * --> +------+---+         +---+---+---+
             +--------+ 3 | h(4) | * -------> | 4 | a | * |
                          +------+---+        +---+---+-|-+
                        2 |      |   |                  |
                          +------+---+            +---+-v-+---+
         bucket index ---- 1 | h(7) | * ---          | 2 | z |   |
                          +------+---+  \         +---+---+---+
                        0 |      |   |   \
                          +------+---+    \   +---+---+---+
                         / hash table      -> | 7 | n |   |
                        /                     +---+---+---+
                   hash bucket

      - Example

            unordered_map<int,string> m {{6,"x"},{4,"a"},{7,"n"},{2,"z"}};
            // set to at least 7 buckets:
            m.rehash(7)
            cout
                << m.size()          // 4
                << m.bucket_count()  // 7
```

```
                << m.bucket_size(1) // 1
                << m.bucket_size(2) // 0
                << m.load_factor(); // 0.57 = 5/7


                            +------+---+    +---+---+---+
                            | h(4) | * ---> | 4 | a |   |
                            +------+---+    +---+---+---+
                            |      |   |
                            +------+---+    +---+---+---+
                +--------+ 4 | h(6) | * ---> | 6 | x |   |
                | m | * --> +------+---+    +---+---+---+
                +--------+   | h(2) | * --
                            +------+---+ \   +---+---+---+
                            |      |   |  -> | 2 | z |   |
                            +------+---+     +---+---+---+
                            | h(7) | * --
                            +------+---+ \   +---+---+---+
                            |      |   |  -> | 7 | n |   |
                            +------+---+     +---+---+---+

        - Check And Control Load Factor

                .load_factor()      -> #occupied_slots / #all_slots
                .max_load_factor()  -> max_allowed_load_factor
                .max_load_factor(new_maximum) (might trigger rehash)

        - Access Or Iterate Over Hash Buckets

                .bucket(key)                -> index_of_bucket_with_key
                .bucket_size(bucket_index)  -> number_of_nodes_in_bucket

            - Get (const_)local_iterator

                .begin(bucket_index)        -> @bucket_begin
                .end(bucket_index)          -> @bucket_end


    - Get Types: Key/Value/...

                using set_t = std::set<double>;
                set_t::size_type i = 0;         // std::size_t
                set_t::key_type{0} k = 0;       // double
                set_t::value_type{0} v = 0;     // double

                using map_t = std::map<int,std::string>;
                map_t::key_type{0} k = 0;           // int
                map_t::mapped_type{0} m = 0;        // string
                map_t::value_type{0,"a"} v {1,"a"}; //pair<const int,string>

            - Example

                                node type
                                    /
                            +---+---+---+---+
                | m | * ------> | 6 | d | * | * |   balanced tree
                            +---+---+-|-+-|-+
                                      /   \
                          +-----------     -------+
                          |                       |
                    +---+-|-+---+---+       +---+-|-+---+---+
                    | 4 | x |   | * |       | 9 | c |   |   |
                    +---+---+---+-|-+       +---+---+---+---+
                                  |
                            +---+-|-+---+---+
                            | 9 | c |   |   |
                            +---+---+---+---+
                             /     \
```

```
                    key_type    mapped_type
                   |_______________________|
                          value_type


  - Make Keys Orderable

      - Key Comparison is Based on Equivalence

          - a and b are equal

                  if a == b is true       (i.e. their values are the same)

          - a and b are quivalent

                  if !(a < b) && !(b < a) is true     (i.e., neither one is
                                                   ordered before the other)

      - C++ standard library
          - Uses equivalence based on "less than" for ordering objects
            (according to strict weak ordering)

      - Alternative 1: Supply A Custom Comparator

          - Ordered containers take an additional type parameter

                      set<Key,KeyComp>
                      map<Key,Mapped,KeyComp>

              - The type 'KeyComp' needs to provide a public member funciton

                      bool operator() (Key const&, Key const&) const

                which returns true, if the first argument should be ordered
                before the second
                  - The defautl is

                      std::less<Key>

      - Alternative 2: Make Your Type Comparable

          - You should only do that
              (1) if objects of your type can be ordered in a way that is
                  "natural" and unambiguous
              (2) if you can at least provide a strict weak ordering

          - More details:

              https://hackingcpp.com/cpp/lang/comparisons.html


  - Make Keys Hashable

      - Unordered containers take an additional type parameter

                  set<Key,Hasher>              ???
                  map<Key,Mapped,Hasher>       ???

              - Shouldnt it be:

                  unordered_set<Key,Hasher>
                  unordered_map<Key,Mapped,Hasher>

          - The type 'Hasher' needs to provide a public member function

                  std::size_t operator() (Key const&) const

            which returns the hash value (= an unsigned index) for a given key
```

```
          - The default is

              std::hash<Key>

    - Example

              struct TM_hash {
                  // 32bit integer hash by T.Mueller
                  constexpr std::size_t
                  operator () (std::uint32_t k) const noexcept {
                      k = (( k >> 16) ^ k ) * 0x45d9f3b;
                      k = (( k >> 16) ^ k ) * 0x45d9f3b;
                      k = (( k >> 16) ^ k );
                      return k;
                  }
              }

              // make set with custom hasher
              std::unordered_set<std::uint32_t,TM_hash> s;
              ...
              // get copy of hasher from set
              auto h = s.hasher();

              // custom key type
              class A { ... };
              // hasher function class
              struct A_hash {
                  std::size_t operator () (A const& k) const noexcept {
                      ... // suitable hash function
                  }
              };

              // make set with cutom hasher
              std::unordered_set<A,A_hash> s;
              ...


- Related

    - Examples of 7 Handy Functions For Associative Containers

      https://www.cppstories.com/2021/handy-map-functions/

    - A Tour Of C++: Containers and Algorithms

      https://isocpp.org/files/papers/4-Tour-Algo-draft.pdf



----------------------------------------------------------------------------
Standard Algorithms Introduction
----------------------------------------------------------------------------


- C++'s Standard Algorithms are

    - algorithmic building blocks
    - operating on (iterator) ranges of elements
    - implemented as free-standing function
    - generic: implemented in a (mostly) container/element-agnostic way
    - many are customizable with function(object)s / lambdas
    - well-tested and efficient


- First Example

            min_element(@begin,             @end) -> @minimum
                         |                    |         |
```

```
                +---+---+-|-+---+---+---+---+-|-+---+    |
                | 7 | 9 | 3 | 5 | 3 | 2 | 4 | 1 | 8 |    |
                +---+---+---+---+---+-|-+---+---+---+    |
                                      |__________________|

        - Returns an iterator to the smallest element and thereby bot its
          position and value

            #include <vector>
            #include <algorithm> //std::min_element

            std::vector<int> v {7,9,3,5,3,2,4,1,8,0};

            // smallest in subrange (as shown above)
            auto i = min_element(begin(v)+2, begin(v)+7);
            auto min = *i       // int min = 2

            // smallest in entire container
            auto j = min_element(begin(v), end(v));
            std::cout << *j << '\n';    //prints '0'
            v.rease(j);                 // erases smallest element


    - Organization

        #include <algorithm>

            - Non-Modifying Queries

                - finding elements / existence queries
                - minimum / maximum
                - comparing ranges of elements
                - binary search of sorted ranges

            - Modifying Operations

                - copying / moving elements
                - replacing / transforming elements
                - removing elements
                - union/intersection/etc. of sorted ranged

        #include <numeric>

            - Operations on ranges of numbers (sums, reductions, ...)

        #include <ranges>

            - Composable range views, range utilities

        #inlcude <iterator>

            - Iterator utilities (distance, next, ...)

        #include <memory>

            - Operations on uninitialized memory

      - C++20

        - improved and easier to use versions of most standard algorithms
        - range and view adapters
        - more stringent handling of algorithm input requirements (based
          on "Concepts")


    - Input Ranges

      - Iterators
```

```
        - Standard algorithms use iterators to traverse/access input elemnts
          - allows algorithms to be implemented independent from container
            types
          - eliminates the need for having one algorithm implementation
            per container type
          - new (third) party containers can be used with existing
            standard algorithm implementations

      - Iterator Ranges

          = pair p,q of iterators

                   begin(v)+3   begin(v)+6
                        │ p          │ q
          +---+---+---+-│-+---+---+-│-+---+---+
        v │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │ 7 │ 8 │ X │
          +-│-+---+---+---+---+---+---+---+-│-+
            │0  1   2   3  ...                │
            │                                 │
          begin(v)                          end(v)

        - NOTE:

          - End-of-range iterator q points one BEHIND the last element in
            the range

        - Example: empty range

            begin(v)+2, begin(v)+2

      - Range Objects As Inputs (C++20)

          ranges::min_element(...) -> @minimum

        - Example

          #include <vector>
          #include <algorithm> // std::ranges::min_element

          std::vector<int> v {3,5,3,2,4,1};
          auto j = std::ranges::min_element(v);
          std::cout << *j << '\n';    // prints '1'

        - Algorithms in C++20's namespace std::ranges

          - also accept single range objects like containers or views as
            inputs (before C++20: only iterator pairs)
          - must be called with the full namespace ("namespace-qualified"
            in C++ parlance) because they can't be found by argument
            dependent lookup (= look up a function in the namespace of its
            arguments)
          - A range is any object r for which std::ranges::begin(r) and
            std::ranges::end(r) return either valid iterators or end-of-
            range indicating sentinels


    - Customization with Callable Prameters

      - Many standarad algorithms can be customized by passing a "callable
        entity" like a function, lambda or custom function object as parameter

          min_element(@begin, @end) compare = o < o       -> @minimum
          min_element(@begin, @end, compare(o,o)│->bool)  -> @minimum

        - The second version of min_elements takes a callable entity as 3rd
          argument for comparing pairs of elements unlike the first version
          which uses operator <
```

- Example: min_element with Custom Type

```
#inlcude <vector>
#include <algorithm>

struct P {int q; char c; };
std::vector<P> v { {2,'c'}, {1,'b'}, {3,'a'} };
```

  - Compare using a function

```
// compares Ps by their 'q' member
bool less_q (P const& x, P const& y) {
    return x.q < y.q;
}

auto i = min_element(begin(v), end(v), less_q);
auto q1 = i->q;     // int  q1 = 1
auto c1 = i->c;     // char c1 = 'b'
```

  - Compare using a lambda

```
// use lambda to compare Ps by 'c'
auto j = min_element(begin(v), end(v),
    [](P const& x, P const& y){
        return x.c <y.c;
    });

auto q2 = i->q;     // int  q2 = 3
auto c2 = i->c;     // char c2 = 'a'
```

  - Lambdas

    - can be thought of as "anonymous function"
    - can be defined within functions (regular C++ functions can not be nested)
    - are function objects whose type is auto-generated by the compiler

- Parallel Execution (C++17)

  - Most standard algorithms can be executed in parallel
    - Provide an 'execution policy' object as first argument:

```
#inlcude <execution>
...
sort(std::execution::par, begin(v), end(v));
```

  - Execution Policies And Effects

Execution Policy	Effect
std::execution::seq	parallelization and vectorization are not allowed
std::execution::unseq	may vectorize, but parallelization is not allowed (C++20)
std::execution::par	may parallelize, but vectorization is not allowed
std::execution::par_unseq	may parallelize, vectorize, or migrate computation accross threads. allows to invoke input element access functions in an unordered fashion, und unsequenced with respect to each other within each thread

  - Compiler Support (min. required versions)

```
        - GNU g++9
            - Requires TBB Library (Intel Thred Building Blocks)
            - Installation

                    $ sudo apt install libtbb-dev

            - Link executable against TBB

                    $ g++ -std=c++17 ... -o exename -ltbb

        - Microsoft MSVC 19.14 (VS 2017 15.7)
        - NVIDIA NVC++


- Iterator / Range Categories

    - Category = set of supported iterator/range objct operations/guarantees

        - based on common algorithm requirements (input, output, efficiency,
          correctness, ...)
        - determined by the input range object or the host container
          providing the iterator


        +--------------+-------------------------------------+-------------+
        | Sentinel     | iterator-like position specifier;   | supports    |
        | (C++20)      | usually only used for denoting the  |  == !=      |
        |              | end of a range                      |             |
        +--------------+-------------------------------------+-------------+
        | Input        | read access to objects; advancable  | supports    |
        |              | to next position                    |  * ++ == != |
        |              | example: iterator that reads        |             |
        |              |   values from a file                |             |
        +--------------+-------------------------------------+-------------+
        | Output       | write access to objects;advancable  | supports    |
        |              | to next position                    |  * ++ == != |
        |              | example: iterator that writes       |             |
        |              |   values to a file                  |             |
        +--------------+-------------------------------------+-------------+
        | Forward      | read/write access; forward          | supports    |
        |              | traversal, no random access;        |  * ++ == != |
        |              | multi-pass guarantee: iterators to  |             |
        |              |   the same range can be used to     |             |
        |              |   access the same objects mulitple  |             |
        |              |   times                             |             |
        |              | example: std::forward_list iteratr  |             |
        +--------------+-------------------------------------+-------------+
        | BiDirectional| multi-pass guarantee, traversal in  | supports    |
        |              | both directions (but no random      |  * ++ -- == |
        |              | access                              |  !=         |
        |              | example: std::list iterator         |             |
        +--------------+-------------------------------------+-------------+
        | RandomAccess | random access, but not necessarily  | supports    |
        |              | to a contiguous memory block        |  * [] ++ -- |
        |              | example: std::deque iterators       |  += -= - +  |
        |              |                                     |  == != < <= |
        |              |                                     |  > >=       |
        +--------------+-------------------------------------+-------------+
        | Contiguous   | random access to contiguous memory  | supports    |
        |              | example: std::vector iterators      |  * [] ++ -- |
        |              |                                     |  += -= - +  |
        |              |                                     |  == != < <= |
        |              |                                     |  > >=       |
        +--------------+-------------------------------------+-------------+


- Error Messages
```

```
    of generic algorithms can be quite confusing:

            std::list x {3,2,8,1};
            std::sort(begin(x), end(x));

        - This does not compile because
            (1) 'sort' requires random access iterators
            (2) 'list' provides bi-directional iterators

    - Always look for the first message that contains the word 'error'

    - Algorithms in Namespace std::ranges (C++20)

        - requirements are checked at the call site using "Concepts"
        - requirements are overall more consistently specified
        - allow compiler error messages to be more helpful (still room
          for improvement)


- Related

        https://hackingcpp.com/cpp/std/algorithms/intro.html

            - Check Youtube Videos!



-----------------------------------------------------------------------------
Container Traversal
-----------------------------------------------------------------------------

- TIPS

    - Try to only write loops if there is no well-tested (standard) library
      function/algorithm for the job to do
    - Prefer non-random linear forward traversal for sequence containers
      like std::vector
        -> best performance due to cache and prefetching friendliness
    - Reverse traversal is only supported by some standard containers


Forward Traversal


- Range-Based Loop

        for (type variable : container)

            + works for all standard sequence and associative conatainers
            + container agnostic -> easy to change container type
            + no out-of-bounds access bugs possible
            + no signed/unsigned index type hassle
            + best performance when using sequence containers (due to linear
              access pattern); cache and prefethcing friendly

            * early exit possible possible with 'break;'

            - not suited for algorithms that require random access patterns

    - Example

            std::vector<Type> v {...};

            // read-only, type cheap to copy/or copy needed:
            for (Type x : v) { cout << x; }
            for (auto x : v) { cout << x; }
```

```
            // read-only, type expensive to copy:
            for (Type const& x : v) { cout << x; }
            for (auto const& x : v) { cout << x; }

            // modify values:
            for (Type& x : v) { cin >> x; }
            for (auto& x : v) { cin >> x; }


    - for_each / for_each_n

            + convenient if having a function(object) to be applied to each
              element
            + works for all standard sequence and associative containers
            + container agnostic -> easy to change container type
            + no signed/unsigned index type hassle
            + self-documenting name

            - out-of-bounds access bugs possible with iterator ranges


        ranges::for_each( range, f(o) )     (invokes f on each input elemnt)

            - C++20
            - {9,1,3,8,5}   -> f(9),f(1),f(3),f(8),f(5)

            + no out-of-bounds access possible


                #include <algorithm>  // std::ranges::for_each
                namespace ranges = std::ranges;  // alias

                Container<Type> v; ...

                // read-only, type cheap to copy or copy needed:
                ranges::for_each(v, [](Type x){ cout << x; };
                ranges::for_each(v, [](auto x){ cout << x;};

                // read-only, type expensive to copy:
                ranges::for_each(v, [](Type const& x){ cout << x;};
                ranges::for_each(v, [](auto const& x){ cout << x;};

                // modify values:
                ranges::for_each(v, [](Type& x){ cin >> x; };
                ranges::for_each(v, [](auto& x){ cin >> x; };

        for_each(@begin, @end, f(o))        (invokes f on each input elemnt)

            + can be used on subranges
            - out-of-bounds access bugs possible


                #inlcude <algorithm>  //std::for_each

                // read-only, type cheap to copy or copy needed:
                for_each(begin(v), end(v),     [](Type x){ cout << x; });
                for_each(begin(v)+2, end(v)+5, [](auto x){ cout << x; });

                // read-only, type expensive to copy:
                for_each(begin(v), end(v), [](Type const& x){ cout << x; });
                for_each(begin(v), end(v), [](auto const& x){ cout << x; });

                // modify values:
                for_each(begin(v), end(v), [](Type& x){ cout << x; });
                for_each(begin(v), end(v), [](auto& x){ cout << x; });

        for_each_n(@begin, n, f(o)) (C++17) (invokes f on each input elemnt)
```

```
            + can be used on subranges
            - out-of-bounds access bugs possible


    - Explicit Use of Iterators

            + container agnostic -> easy to change container type
            + works for all standard sequence containers
            + no signed/unsigned index type hassle
            + possible to skip multiple elements
            - out-of-bounds access bugs possible
            - verbose


              std::vector<int> v {1,2,3,4,5,6};

              for (auto i = begin(v); i != end(v); ++i) { cout << *i; }
              for (auto i = begin(v); i != end(v); ++i) { cin  >> *i; }

              // read-only - using const iterators
              for (auto i = cbegin(v); i != cend(v); ++i) { cout << *i }


    - Index-Based Loop

            + possible to skip multiple elements
            - prone to out-of-bounds access bugs
            - easy to write sublte bugs due to signed/unsigned index type
              conversions
            - does not work for all sequence containers -> not easy to
              change container type
            - making sure that loop doesn't modify elements requires more
              discipline
            - verbose


              std::vector<int> v {1,2,3,4,5,6};

              for (std::size_t i = 0; i < v.size(); ++i) { cout << v[i]; }

              // explicitly read-only
              for (std::size_t i = 0; i < cv.size(); ++i) {cout << cv[i];}


    Reverse Traversal


    - Reverse Range-Based Loop (C++20)

        for (type variable : container | std::views::reverse)

            + works for all bidirectional containers
            + no out-of-bounds access bugs possible
            + no signed/unsigned index type hassle
            * ealry exit possible with 'break;'


              #include <ranges>  // C++20

              std::vector<int> v {1,2,3,4,5,6};
              for (int x : v | std::views::reverse) {cout << x << '\n';}

              // read-only, if type cheap to copy or copy needed:
              for (auto x : v | std::views::reverse) {cout << x;}

              //read-only, if type expensive to copy:
              for (auto const& x : v | std::views::reverse) {...}
```

```
                // modify items:
                for (auto& x : v | std::views::reverse) {...}


    - Reverse for_each/for_each_n

              + convenient if having a function(object) to be applied to each
                element
              + works for all bidirectional containers
              + easy to change container type
              + no signed/unsigned index type hassle
              + self-documenting name
              - out-of-bounds access bugs possible with iterator ranges


        - ranges::for_each(range, f(o))

                #include <algorithm>    // std::ranges::for_each
                #include <ranges>       // range views

                namespace ranges = std::ranges;         // alias
                namespace views = std::ranges::views;   // alias

                Container<Type> v;

                // read-only, type cheap to copy or copy needed:
                ranges::for_each(views::reverse(v), [](Type x){cout << x;});
                ranges::for_each(views::reverse(v), [](auto x){cout << x;});

                // read-only, type expensive to copy:
                ranges::for_each(views::reverse(v), [](Type const& x){
                        cout << x; });
                ranges::for_each(views::reverse(v), [](auto const& x({
                        cout << x; });

                // modify values:
                ranges::for_each(views::reverse(v), [](Type& x){
                        cout << x; });
                ranges::for_each(views::reverse(v), [](auto& x){
                        cout << x; });

        - for_each(@begin, @end, f(o))

              + can be used on subranges
              - out-of-bounds access bugs possible

                for_each(rbegin(v), rend(v), [](Type x){...});
                ...

        - for_each_n(@begin, n, f(o))

              + can be used on subranges
              - out-of-bounds access bugs possible

                for_each_n(rbegin(v), 2, [](Type x){...});
                ...


    - Explicit Use of Reverse Iterators

              + works for all bidirectional containers
              + no signed/unsigned index type hassle
              + possible to skip multiple elements
              - out-of-bounds access bugs possible
              - verbose

                for (auto i = rbegin(v); i != rend(v); ++i) {...}
```

```
                // read-only - using const iterators
                for (auto i = crbegin(v); i != crend(v); ++i) {...}


    - Reverse Index-Based Loop

              - prone to out-of-bounds access bugs
              - easy to write subtle bugs due to unsigned size type:
                implicit conversions to signed int, overflow/wrap-around, ...
              - making sure that loop doesn't modify elements requires more
                discipline
              - verbose

                // std containers use UNsigned size types
                // -> be careful not to decrement unsigned '0'
                for (auto i = v.size(); i > 0; --i) { cout << v[i-1]; }

                // explicitly read-only
                const auto& cv = v;
                for (auto i = cv.size(); i > 0; --i) { cout << cv[i-1]; }


    Utilities


    - Get Next/Previous Iterator

        #include <iterator>

            - std::prev and std::next

              - Functions
              - Provide a universal way of incrementing/decrementing iterators
                - Even if the iterator type does not support random access
                  (e.g., 'it += 5')
              - WARNING
                - Be aware that advancing non-random access iterators (e.g.,
                  those from std::list) by N steps might be costly
                    - i.e., involve on the order of N memory operations

            next(@position)          -> @one_after
            next(@psoition, steps)   -> @steps_after


                std::vector<int> v {1,2,3,4,5,6};    // 1 2 3 4 5 6 _
                auto i = next(v.begin());            //   ^i
                auto j = next(i, 3);                 //   ^i    ^j

            prev(@position)          -> @one_before
            prev(@psoition, steps)   -> @steps_before

                std::vector<int> v {1,2,3,4,5,6};    // 1 2 3 4 5 6 _
                auto i = prev(v.end());              //           ^i
                     i = prev(i);                    //         ^i
                auto j = prev(i, 3);                 //   ^j    ^i
```

Standard Library min/max Algorithms

```
    - min

        min(a, b) -> a if (a < b) is true, b otherwise
        min(a, b, cmp(o,o)->bool) -> a if cmp(a,b) is true, b otherwise
```

```
        - Example

            int const a = 2;
            int const b = 9;
            int x 0 std::min(a,b);  // int x = 2

        - Example

            struct P { int q; char c; };
            P pmin = std::min(p{1,'y'}, P{2,'x'}, [](P p1, P p2){
                return p1.q < p2.q });      // P min {1,'y'}

    min({v1,v2,v3,...}) -> smallest_value (C++11)
    min({v1,v2,v3,...}, cmp(o,o)->bool) -> smallest_value

        - The second version uses cmp for comparing elements,
          while the first version uses 'operator <'
        - CARE
            - ALL elements in the input list {...} must have the same type!
        - Example

            int const a = 2;
            int const b = 9;
            int x = std::min({3,4,b,3,a,8});

        - Example

            std::set<int> s1 {3,5,2};
            std::set<int> s2 {9,3,1};
            std::set<int> s3 {4,2,6};

            int s_min = std::min({s1,s2,s3});

        - Example

            struct P { int q; char c; };

            P px {3,'x'};
            P py {2,'y'};
            P pz {1,'z'};

            P p_min = std::min({px,py,pz},
                [](P p1, P p2){ return p1.q < p2.q; });

    ranges::min(range) -> smallest_value (C++20)
    ranges::min(range, cmp(o,o)->bool) -> smallest_value

        - Returns (a const reference to) the smallest element in range
        - The second version uses cmp for comparing elements,
          while the first version uses 'operator <'
        - Example

            std::vector<int> v {7,9,3,5,3,1,5,8};
            auto x = std::ranges::min(v);           // int x = 1

        - Example

            struct P { int q; char c; };

            std::vector<P> const w {P{3,'a'},P{1,'c'},P{2,'b'}};
            auto pmin = std::ranges::min(w,
                [](P const& p1, P const& p2){ return p1.q < p.2; });

  - max

    max(a, b) -> a if (a < b) is false, b otherwise
    max(a, b, cmp(o,o)->bool) -> a if cmp(a,b) is false, b otherwise
```

```
      max({v1,v2,v3,...}) -> largest_value (C++11)
      max({v1,v2,v3,...}, cmp(o,o)->bool) -> largest_value

      ranges::max(range) -> largest_value (C++20)
      ranges::max(range, cmp(o,o)->bool) -> largest_value


  - minmax

      minmax(a, b) -> {smallest,largest} (C++11)
      minmax(a, b, cmp(o.o)-> bool) -> {smallest, largest}

          - Comparison function/object cmp(a,b) must return true if 'a' should
            be ordered before 'b'
          - Example

              int a = 2;
              int b = 9;

              auto p = std::minmax(a,b);  // std::pair<int,int> p {2,9}
              auto min = p.first;         // int min = 2
              auto max = p.second;        // int max = 9

              auto [lo,hi] = std::minmax(a,b); // int lo = 2, hi = 9 (C++17)

      minmax({v1,v2,v3}) -> {smallest,largest}
      minmax({v1,v2,v3}, cmp(o.o)->bool) -> {smallest,largest}

          - The second version uses cmp for comparing elements,
            while the first version uses 'operator <'
          - CARE
              - All elements in the input list { ... } must have the same type

              auto p = std::minxmax({3,0,b,3,a,8}); // std::pair<int,int> p
                                                    // {0,9}
              auto min = p.first;
              auto max = p.second;

              auto [lo,hi] = std::minmax({3,0,b,3,a,8});
                                                // int lo = 0, hi = 9 (C++17)

      ranges::minmax(range) -> {smallest,largest} (C++20)
      ranges::minmax(range, cmp(o.o)->bool) -> {smallest,largest}

          - Returns a pair of (const references to) the smallest and largest
            elements in range
          - The second version uses cmp for comparing elements,
            while the first version uses 'operator <'

              std::vector<int> v {7,9,3,6,3,1,4,8};
              auto p = std::ranges::minmax(v);  // std::pair<int,int> p {1,9}

              struct P { int q; char c; };
              std::vector<P> const w {P{3,'a'},P{2,'b'},P{1,'c'}};
              auto [lo,hi] = std::ranges::minmax(w,
                  [](P p1, P p2){ return p1.q < p2.q; });


  - clamp (C++17)

      clamp(value, lo, hi) -> clamped_value
      clamp(value, lo, hi cmp(o.o)->bool) -> clamped_value

          - clamps value in the interval given by lo and hi
          - The second version uses cmp to compare values
            instead of 'operator <'

              int a = std::clamp( 8, 1, 5);   // int a =  5
```

```
            int b = std::clamp(-4, 1, 5);   // int b =  1
            int c = std::clamp(-4,-2, 5);   // int c = -2


    - min_element

        min_element(@begin, @end)                       -> @minimum
        min_element(@begin, @end, compare(o.o)-> bool   -> @minimum

                  @begin      @end
                     |          |
                     v          v
                |7|9|3|5|3|2|4|1|8|0
                           ^
                           |
                        @minimum

            - The second version uses comp for comparing elements,
              while the first version uses 'operation <'


            std::vector<int> v {7,9,3,5,3,2,4,1,8,0};

            // smallest in subrange (as shown above)
            auto i = min_element(begin(v)+2, begin(v)+7);
            auto min = *i;      // int min = 2

            // smallest in entire vector
            auto j = min_element(begin(v), end(v));
            std::cout << *j;    // print '0'

            // index of smallest
            auto argmin = distance(begin(v), j);    // int argmin = 9

            // erase at i's position       7 9 3 5 3 2 4 1 8 0
            i = v.erase(i);             //           ^

            std::cout << *i;            // 7 9 3 5 3 4 1 8 0
            // prints '4'                            ^

        ranges::min_element(v)                  -> @minimum (C++20)
        ranges::min_element(v, comp(o.o)->bool) -> @minimum

            - The second version uses comp for comparing elements,
              while the first version uses 'operation <'


            std::vector<int> v {7,9,8,3,6,4,0,4};

            auto i = std::ranges::min_element(v);
            auto min = *i;  // int min = 0;


    - max_element

        max_element(@begin, @end)                   -> @maximum
        max_element(@begin, @end, comp(o.o)->bool)  -> @maximum

        ranges::max_element(v)                      -> @maximum (C++20)
        ranges::max_element(v, comp(o.o)->bool)     -> @maximum

    - minmax_element

        minmax_element(@begin, @end)                    -> {@min,@max}
        minmax_element(@begin, @end, comp(o.o)->bool)   -> {@min,@max}

        ranges::minmax_element(v) -> {@min,@max} (C++20)
```

```
            auto [min,max] = std::ranges::minmax_element(v);

            std::cout << "min: " << *min << '\n'
                      << "max: " << *max << '\n';

    ranges::minmax_element(v, comp(o.o)->bool) -> {@min,@max}



--------------------------------------------------------------------------------
Standard Library Existence Queries
--------------------------------------------------------------------------------

- any_of / all_of / none_of (C++11)

    all_of (@begin, @end, check(o)->bool)   -> true,
    any_of (@begin, @end, check(o)->bool)   -> if check yields true for all,
    none_of(@begin, @end, check(o)->bool)   -> any, or none of the elements
                                               in input range


            auto const check = [](int x) { return x >= 1; };

            cout << all_of (begin(v), end(v), check);

    ranges::all_of (range, check(o)->bool)  -> bool
    ranges::any_of (range, check(o)->bool)  -> bool
    ranges::none_of(range, check(o)->bool)  -> bool


- count

    count(@begin, @end, value)  -> number of occurrences

    ranges::count(range, value) -> number of occurrences (C++20)


- count_if

    count_if(@begin, @end, f(o)->bool) -> #elements(f=true)


            std::vector<int> v {5,4,9,1,3,2,5,6,8,9};

            auto const is_even = [](int x) { return !(x & 1); };

            auto n = count_if (begin(v)+1, begin(v)+8, is_even);    // 3
            auto m = count_if (begin(v), end(v), is_even);

    count_if(range, f(o)->bool) -> #elements(f=true)


--------------------------------------------------------------------------------
Standard Library Finding Algorithms
--------------------------------------------------------------------------------


Find / Locate One Element


- find

    find(@begin, @end, value)   -> @1st element equal to value
                                -> @end if no match
    ranges::find(range, value) (C++20)

- find_if
```

```
        find_if(@begin, @end, f(o)->bool)   -> @1st_element for which f is true
                                            -> @end if no such element found

                auto const f = [](int x){return x >= 6; };

                auto i = find_if(begin(v)+2, begin(v)+7, f);

        ranges::find_if(range, f(o)->bool) (C++20)

    - find_if_not

        find_if_not(@begin, @end, f(o)->bool)   -> @1st_element for which f is
        ranges::find_if_not (range, f(o)->bool) -> false - @end if no such elem

    - find_last / _if / _if_not

        ranges::find_last(range, value) -> last2end_view (C++23)

                     range
                +-------------+
                |             |
                |2|1|7|1|1|5|8|
                        |     |
                        +-----+
                      last2end_view (empty if nothing found)


                std::vector<int> v {2,1,7,1,1,5,8};

                auto const result = std::ranges::find_last(v, 1);

                if (not result.empty()){                     // if found
                    auto const value = result.front();      // int value = 1
                    auto const index = distance(begin(v), begin(result));   // 4
                }

                for (int x : result) { cout << x << ' '; }  // 1 5 8

        ranges::find_last_if    (range, value)  -> last2end_view (C++23)
        ranges::find_last_if_not(range, value)  -> last2end_view (C++23)

                auto const f = [](int x){ return x >= 2; };
                auto const result = ranges::find_last_if_not(v,f);


    - find_first_of

        find_first_of(@s_begin,@s_end,@w_begin,@w_end)  -> @1st match
                                                        -> @s_end if no match
          @s_begin|  search here   |@s_end
                  +---------------+|
                  |               ||
                |0|1|3|2|5|7|4|8|9|9|

                    |1|4|6|5|8|7|
              @w_begin|      ||@w_end
                      +------+
                find any of these values


                std::vector<int> s {0,1,3,2,5,7,4,8,9,9};
                std::vector<int> w {1,4,6,5,8,7};

                auto i = find_first_of(begin(s)+1,begin(s)+9,
                                       begin(w)+1,begin(w)+4);
                if (i != begin(s)+9) {                       // true, found one
                    auto const value = *i;                   // int value = 5
                    auto const index = distance(begin(s), i); // index = 4
```

```
            }

      ranges::find_first_of(range_s, range_w) -> @1st match
                                              -> @end_s if no match

            std::vector<int> s {3,2,5,7,4,8};
            std::vector<int> w {4,6,5};

            auto i = std::ranges::find_first_of(s,w);


   Find Subrange in Range


   - search

      search(@s_begin,@s_end,@w_begin,@w_end)     -> @1st occurrence of range
                                                     'w' inside range 's'
                                                  -> @s_end otherwise
                  seach here
              +---------------+
        @s_begin|               ||@w_end
             |0|4|6|5|1|4|6|5|8|9|

             |1|4|6|5|8|9|
        @w_begin|     ||@w_end
              +-----+
          find this range


            std::vector<int> s {0,4,6,5,1,4,6,5,8,9};
            std::vector<int> w {1,4,6,5,8,9};

            auto i search = search(begin(s)+1,begin(s)+9,
                                   begin(w)+1,begin(w)+4);


      ranges::search(range_s,range_w) -> subrange_view (C++20)


            std::vector<int> s {1,4,6,5,8,4,6,5};
            std::vector<int> w {4,6,5};

            auto r = std::ranges::search(s,w);
            if (not empty(r)) {
                for (int x : r) {cout << x << ' ';}     // 4 6 5
            }


   - find_end

      find_end(@s_begin,@s_end,@w_begin,@w_end)   -> @last occurrence of
                                                     range w inside range s
                                                  -> @s_end otherwise
                  seach here
              +---------------+
        @s_begin|               ||@w_end
             |0|4|6|5|1|4|6|5|8|9|
                     |     |
                     +-----+ found


             |1|4|6|5|8|9|
        @w_begin|     ||@w_end
              +-----+
          find this range
```

```
    ranges::find_end(range_s,range_w) -> subrange_view (C++20)

            |4|6|5|8|4|6|5|      |5|6|5|
                    |     |
                    +-----+


- starts_with

    ranges::starts_with(@s_begin,@s_end,@w_begin,@w_end) (C++23)
        -> true if s starts with the vales in w

            s |1|2|3|4|      w |1|2|
              |   |
              +---+

    ranges::stars_with(range_s,range_w) (C++23)
        -> true if s starts with the values w


- ends_with

    ranges::ends_with(@s_begin,@s_end,@w_begin,@w_end) (C++23)
        -> true if s ends with the vales in w

            s |1|2|3|4|      w |3|4|
                  |   |
                  +---+

    ranges::ends_with(range_s,range_w) (C++23)
        -> true if s ends with the vales in w


Find Run of Equal Elements


- adjacent_find

    adjacent_find(@begin,@end)  -> @1st occurrence of two consecutive elmnts
                                -> @end if no consecutive elements found

    ranges::adjacent_find(range) (C++20)
                                -> @1st occurrence of two consecutive elmnts
                                -> @end if no consecutive elements found


- search_n

    search_n(@begin,@end, n, value) -> @1st occurrence of n consecutive
                                       values
                                    -> @end if no subsequence found

    ranges:search_n(range,n,value)  -> subrange_view


----------------------------------------------------------------------------
Standard Library Range Comparisons
----------------------------------------------------------------------------


- equal

    equal(@begin1, @end1, @begin2)          -> true, if all elements in both
    equal(@begin1, @end1, @begin2, @end2)      ranges are equal

    ranges::equal(range1,range2) (C++20)    -> true, if all elements in both
                                               ranges are equal
```

```
    - mismatch

        mismatch(@begin1, @end1, @begin2)       -> {@mismatch in range1,
        mismatch(@begin1, @end1, @begin2, @end2)    @mismatch in range2}

        ranges::mismatch(range1,range2)         -> {@in1, @in2}


    - lexicographical_compare

        lexicographical_compare(@begin1, @end1, @begin2, @end2)
        -> true, if range 1 should be ordered before range 2

        ranges::lexicographical_compare(range1, range2)
        -> true, if range 1 should be ordered before range 2


                std::vector<char> range1 = {'a','l','g','o'};
                std::vector<char> range2 = {'b','c','e'};

                // true:
                cout << std::ranges::lexicographical_compare(range1, range2);
                // false:
                cout << std::ranges::lexicographical_compare(
                    range1, range2, std::greater<>{});


    - lexicographical_compare_three_way

        lexicographical_compare_three_way(@begin1, @end1, @begin2, @end2)(C++20)
        -> 3-way comparison result

                result < 0 -> range1 before    range2
                result = 0 -> range1 equiv. to range2
                result > 0 -> range1 after     range2


            - Comparing entire strings with C++20's 'spaceship' operator:

                std::string r1 = "xalgori";
                std::string r2 = "abced";

                auto const lcB = r1 <=> r2;
                cout
                    << std::boolalpha
                    << (lcB <  0)       // false
                    << (lcB == 0)       // false
                    << (lcB >  0);      // true
```