# The C++ Standard Library, Iterators and Ranges

Lecture 6

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

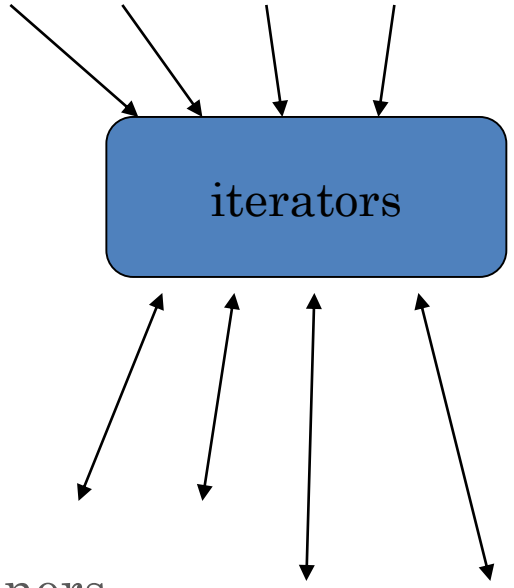
# Abstract

- Many containers share similar operations, like `insert()` or `erase()`. Those have the same interface for all of them (even for strings).
- All containers expose a companion iterator type allowing to navigate through the elements stored in the container. Again, all of them expose a similar interface
- We will see how the library exploits these similarities by exposing generic algorithms: by consuming uniform interfaces independent of the container they are applied to.

# Basic Model of STL

- Algorithms

  `sort`, `find`, `search`, `copy`, …




- Containers

  `vector`, `list`, `map`,
  `unordered_map`, …

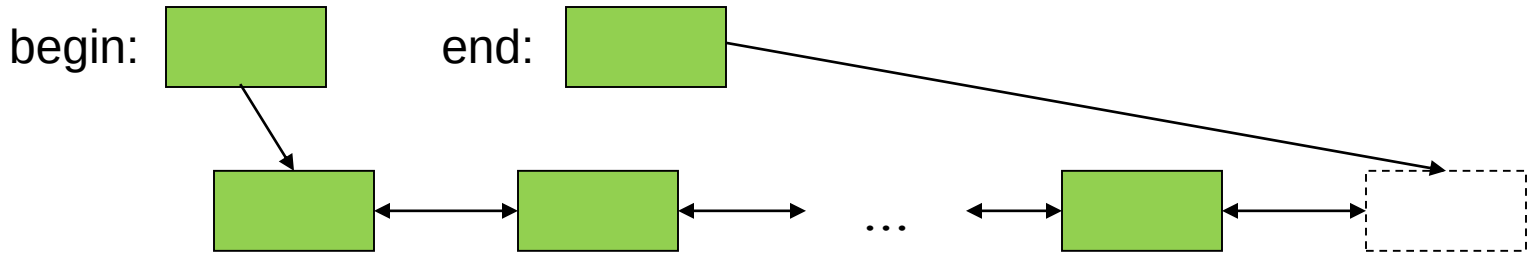
- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types
    - Iterators know about internal container structure and data
    - Iterators expose uniform interface for algorithms

# Iterators

Iterators are objects that refer to a single element inside a container

# Basic Model: Pair of Iterators (Range)

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)




- An iterator is a type that supports the "iterator operations" of
  - ++ Point to the next element in the sequence
  - * Get the value of the element the iterator refers to
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g., --, +, and [])

# Generic Algorithms

- `std::copy` is a generic algorithm
  - Not part of any container
  - Its operation is determined by its arguments
    - Most of the time the standard algorithms expect iterators
- `std::copy` takes 3 iterators (begin, end, out) and copies the range `[begin, end)` to a sequence starting at `out` (extending as necessary)

# Standard Algorithm: copy

- Writing

```
std::copy(begin, end, out);
```

- Is equivalent to (except for iterators not being copied):

```
while (begin != end)
    *out++ = *begin++;
```

- What does '`*out++ = *begin++;`' mean?

```
{ *out = *begin; ++out; ++begin; }
```

# Iterator Adaptor

- `std::back_inserter()` is an iterator adaptor
  - Function returning a special iterator created based on the function's arguments
  - It takes a container and returns an iterator, which when used as a destination, appends elements to that container (by extending it)
- This will append all of bottom to the container `ret`:

```
std::vector<...> ret;
std::copy(bottom.begin(), bottom.end(), std::back_inserter(ret));
```

# Caveats: copy

- This will not work (will not compile) - why?

```
std::copy(bottom.begin(), bottom.end(), ret);
// ret is not an iterator, but a container
```

- This will compile, but not work - why?

```
std::copy(bottom.begin(), bottom.end(), ret.end());
// while ret.end() is an iterator, it does not refer to
// any element (remember, it 'points' past last element)
```

- Many problems, why is it designed that way?
  - Separation of copying and appending (expanding a container) allows for more flexibility
  - None of the algorithms change the container itself, may reorder or change elements, though
  - `std::back_inserter` useful in other contexts as well

# Another Copy Example

```
void f(std::vector<double> const& vd, std::list<int>& li)
{
    if (vd.size() > li.size())
        throw std::runtime_error("target container too small");

    std::copy(vd.begin(), vd.end(), li.begin());   // note: different container types
                                                   // and different element types
                                                   // (li better have enough elements
                                                   // to hold copies of vd's elements)
    // ...
}
```

```
while (begin != end)
     *out++ = *begin++;
```

# Aside Stream I/O

# Aside: Console I/O

- Read a list of doubles and compute their median:

```
// compute median value of all doubles taken from input
int main()
{
    std::vector<double> values;

    double x;
    while (std::cin >> x)          // read next value and append it to vector
        values.push_back(x);

    // sort values stored in the vector
    std::sort(values.begin(), values.end());
    std::cout << "Median value: " << values[values.size()/2] << std::endl;
}
```

# Aside: Revisiting I/O

- `std::fstream` is similar to `std::cin` and `std::cout`, just refers to a file
- Files are also represented by streams:

```
{
    // Open for writing or create if file doesn't exist.
    std::ofstream out("data.txt");
    out << "Hello World!\n";
}    // out closes

{
    // Open for reading.
    std::ifstream in("data.txt");
    std::string line;
    std::getline(in, line);
    std::cout << "line == " << line << "\n";    // line == "Hello World!"
}
```

# Aside: Revisiting I/O

- It is also possible to use an output stream that fills a `std::string`

```
std::string line;
{
    std::stringstream strm;
    strm << "Hello world!";
    line = strm.str();
}
std::cout << "line == " << line << "\n";    // line == "Hello World!"
```

# Aside: I/O for your own Types

```
struct X
{
    int64_t value;
};

std::ostream& operator<<(std::ostream& out, X const& el)
{
    return out << el.value;
}

std::istream& operator>>(std::istream& in, X& el)
{
    return in >> el.value;
}
```

# Aside: I/O for your own Types

```
{
    std::ofstream out("data.txt");
    X a{42};
    X b{7};
    out << a << " " << b;
}    // out closes

{
    std::ifstream in("data.txt");
    X a{0};
    X b{0};
    in >> a >> b;
    std::cout << "a.value == " << a << ", b.value == " << b
              << "\n";    // a.value == 42, b.value == 7
}
```

# Back to Iterators

# Input and Output Stream Iterators

```
// we can create iterators for output streams:
std::ostream_iterator<std::string> oo(std::cout); // assigning to *oo is the same
                                                  // as writing a string to cout

*oo = "Hello, ";      // meaning: std::cout << "Hello, "
++oo;                 // "get ready for next output operation"
*oo = "world!\n";     // meaning: std::cout << "world!\n"

// we can create iterators for input streams:
std::istream_iterator<std::string> ii(std::cin);  // reading *ii is the same as
                                                  // reading a string from cin

std::string s1 = *ii;     // meaning: std::cin >> s1
++ii;                     // "get ready for the next input operation"
std::string s2 = *ii;     // meaning: std::cin >> s2
```

# Make a Quick Dictionary (using a std::vector)

```
std::ifstream is("from.txt");                 // open input stream
std::ofstream os("to.txt");                   // open output stream

std::istream_iterator<std::string> ii(is); // make input iterator for stream
std::istream_iterator<std::string> eos;    // input sentinel (defaults to EOF)

// make output iterator for stream, append "\n" each time
std::ostream_iterator<std::string> oo(os, "\n");

std::vector<std::string> words(ii, eos);   // words is a vector initialized
                                           // from input
std::sort(b.begin(), b.end());             // sort the buffer
std::unique_copy(b.begin(), b.end(), oo); // copy buffer to output,
                                           // discard replicated values
```

# An Input File

This lecture and the next previous the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with “policies”.

# Part of the Output

(data)
(processing)
(the
C++
First,
Function
I
It
STL
The
This
a
algorithms
algorithms.
an
and
are
concepts,
containers
data
dealing
examples
extensible
finally
Framework
fundamental
general
ideal,
in
is
iterator
key
lecture
library).
next
notions
objects
of
parameterize
part
present
presented.
presents
program.
sequence
…

# Anatomy of an Iterator

# Iterators

- Iterators are special types
  - Identify a container and an element in the container
  - Let us examine the value stored in that element
  - Provide operations for moving between elements in the container
  - Restrict the available operations in ways that correspond to what the container can handle efficiently
- Iterators can be more general purpose
  - As we have seen, and as we will see later

# Iterator Types

- Every standard container, such as `std::vector`, defines two associated iterator types:

```
container_type::iterator
container_type::const_iterator
```

  - Where `container_type` is the container (`std::vector<double>`)
  - Use `iterator` to modify the element, `const_iterator` otherwise (read only access)

- Note, that we don‘t actually see the actual type, we just know what we can do with it.
  - Abstraction is selective ignorance!

# Iterator Types

- Every `container_type::iterator` is convertible to the corresponding `container_type::const_iterator`
  - I.e., `words.begin()` returns an iterator, but we can assign it to a `const_iterator`
- Opposite is not true! Why?

# Iterator Operations

- Containers do not only expose their (specific) iterator types, but also actual the iterators themselves:

```
words.begin(), words.end()
```

  - `begin()`: 'points' to the first element
  - `end()`: 'points' to the element after the last one

- Iterators can be compared:

```
iter != words.end()
```

  - Tests, whether both iterators refer to the same element

- Iterators can be incremented:

```
++iter
```

  - Make the iterator 'point' (refer) to the next element in the container

# Iterator Operations

- Iterators can be dereferenced:

```
*iter
```

  - Evaluates to the value of the element the iterator refers to

- In order to access a member of the element the iterator refers to, we write:

```
(*iter).name
```

  - (why not: `*iter.name` ?)

- Syntactic sugar, 100% equivalent:

```
iter->name
```

```
struct X { std::string name; };
std::vector<X> many_xs;
auto iter = many_xs.begin();
```

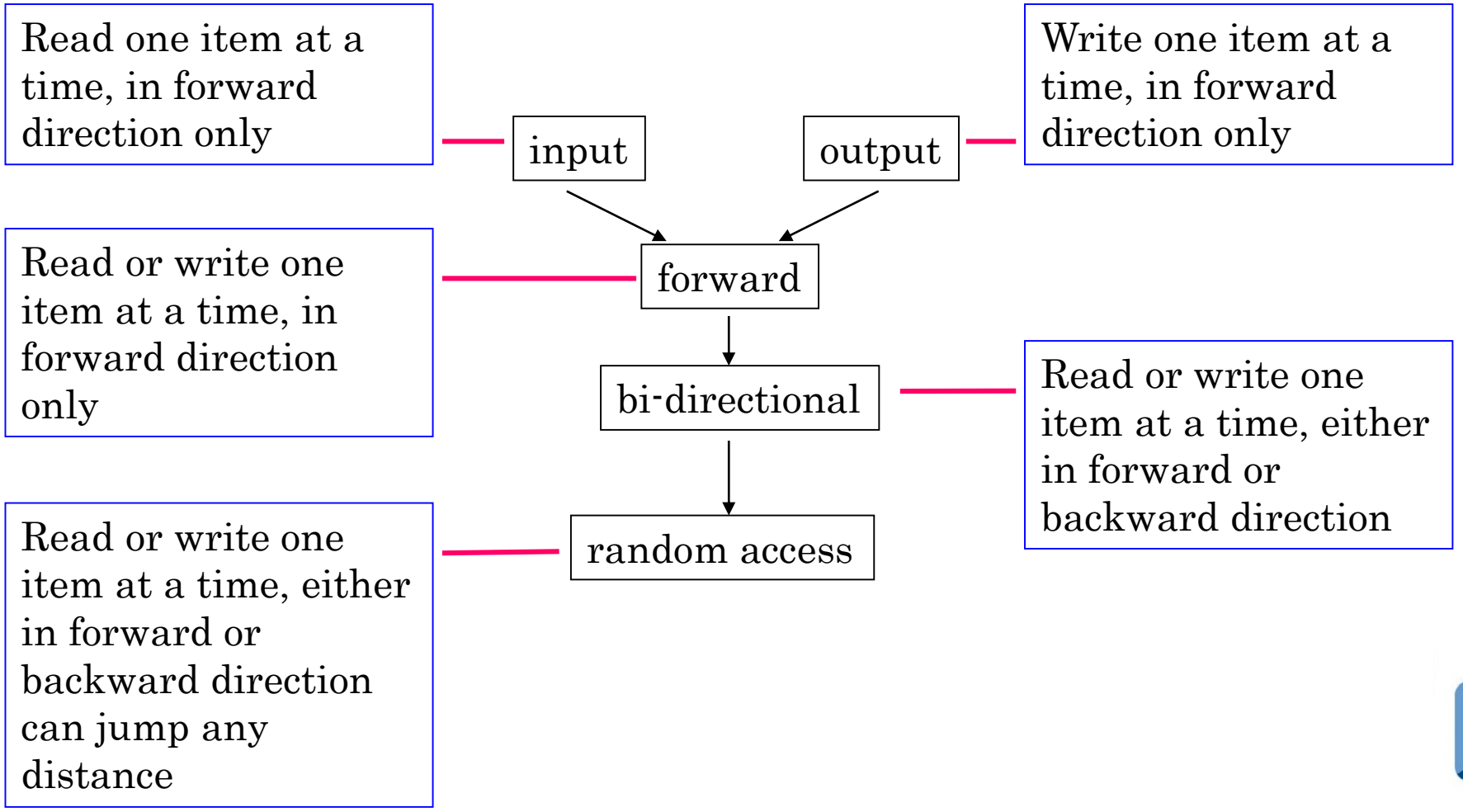
# Iterator Operations

- Some iterators can get a number added

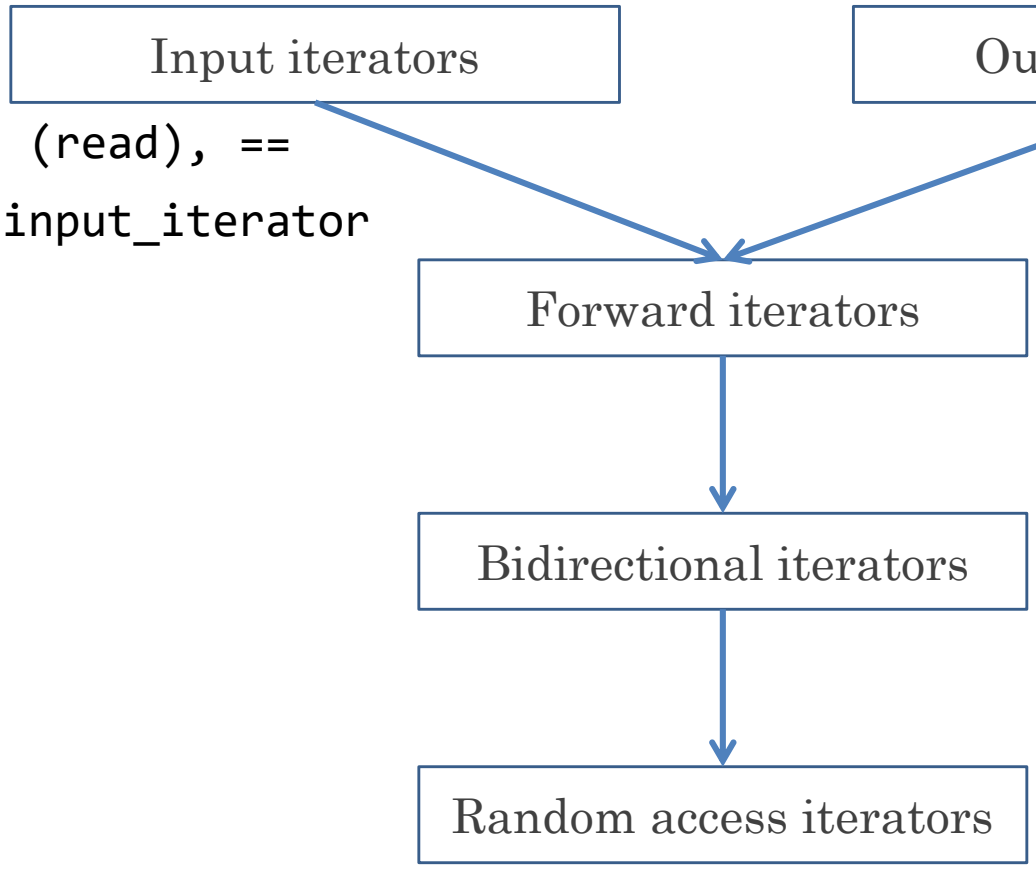
```
words.erase(words.begin() + i);
```

- Overloaded `operator+`, makes the iterator refer to the ‘i’ –s element after begin
- Equivalent to invoking ++ ‘i’ times
- Defined only for iterators from random access containers
  - `std::vector`, `std::string` are random access (indexing is possible)
  - Will result in compilation error for sequential (non-random access) containers

# Iterator Categories (concepts)



# Iterator Types




```
++, * (read), ==
std::input_iterator
```

```
++, * (write), ==
std::output_iterator
```

```
++, *, ==
     std::forward_list
std::forward_iterator
```

```
++, --, *, ==
     std::list, std::map
std::bidirectional_iterator
```

```
++, --, +/-N, [], *, ==
     std::vector, std::array
std::random_access_iterator
```

# Containers and Iterators

| Container | Iterator | Container | Iterator |
|---|---|---|---|
| `vector` | random access | `map` | bidirectional |
| `deque` | random access | `multimap` | bidirectional |
| `list` | bidirectional | `stack` | none |
| `set` | bidirectional | `queue` | none |
| `multiset` | bidirectional | `priority_queue` | none |
| `forward_list` | forward | | |

- Every container
  - Has embedded typedefs for this (no need to remember above):
    - iterator, const_iterator, reverse_iterator, const_reverse_iterator
  - Exposes functions returning iterators:
    - begin(), end(), rbegin(), rend() (non-const and const variants)

# More Generic Algorithms

# Splitting Strings: Take 1

```
std::vector<std::string> split(std::string const& s) {
    std::vector<std::string> words;
    size_t i = 0;

    // invariant: we have processed characters in range [original value of i, i)
    while (i != s.size()) {
        // ignore leading blanks, find begin of word
        while (i != s.size() && std::isspace(s[i]))   // short-circuiting
            ++i;

        // find end of next word
        size_t j = i;
        while (j != s.size() && !std::isspace(s[j]))  // short-circuiting
            ++j;

        // if we found some non-whitespace characters, store the word
        if (i != j) {
            // copy from s starting at i and taking j - i chars
            words.push_back(s.substr(i, j - i));
            i = j;
        }
    }
    return words;
}
```

# Splitting Strings: Take 2

```
std::vector<std::string> split(std::string const& str)
{
    std::vector<std::string> words;
    auto i = str.begin();
    while (i != str.end()) {
        i = std::find_if(i, str.end(), not_space);   // ignore leading blanks
        auto j = std::find_if(i, str.end(), space);  // find end of next word

        // copy the characters in [i, j), append to word list
        if (i != str.end())
            words.push_back(std::string(i, j));
        i = j;
    }
    return words;
}
```

# Splitting Strings: Take 2

- Here are the predicates:

```
// true if the argument is a whitespace character, false otherwise
bool space(char c)
{
    return std::isspace(c);
}

// false if the argument is a whitespace character, true otherwise
bool not_space(char c)
{
    return !std::isspace(c);
}
```

# Standard Algorithm: find_if

- Find an entry in a sequence

```
std::find_if(begin, end, pred);
```

  - Goes over the sequence `[begin, end)` and calls the predicate '`pred`' for each element
  - Returns current position (iterator) as soon as the predicate returns `true` for the first time
  - Essentially this finds the first element in the sequence matching the predicate

# Splitting Strings: Take 2

```
std::vector<std::string> split(std::string const& str)
{
    std::vector<std::string> words;
    auto i = str.begin();
    while (i != str.end()) {
        i = std::find_if(i, str.end(), [](char c) { return !std::isspace(c); });
        auto j = std::find_if(i, str.end(), [](char c) { return std::isspace(c); });

        // copy the characters in [i, j), append to word list
        if (i != str.end())
            words.push_back(std::string(i, j));
        i = j;
    }
    return words;
}
```

# Palindromes

- Palindromes are words that are spelled the same way front to back as back to front: civic, eye, level, madam, etc.
- Simplest solution using a library algorithm:

```
bool is_palindrome(std::string const& s)
{
    return std::equal(s.begin(), s.end(), s.rbegin());
}
```

- New constructs: `equal()`, `rbegin()`

# Reverse Iterators

- Like `begin()`, `rbegin()` returns an iterator
  - It is an iterator that starts with the last element in the container
  - When incremented, it marches backward through the container
  - The iterator returned is called reverse iterator
- Correspondingly, like `end()`, `rend()` returns an iterator that marks the element before the first one

# Standard Algorithm: `equal`

- The standard algorithm `std::equal()` compares two sequences
  - Returns whether these sequences hold the same elements

```
std::equal(begin1, end1, begin2)
```

  - Compares `[begin1, end1)` with elements in sequence starting at `begin2`
  - Assumes second sequence is long enough, if not will return false
  - There is an additional version allowing to specify the end of the second sequence as well:

```
std::equal(begin1, end1, begin2, end2)
```

# Palindromes, Take Two

- Solution using other library algorithms:
  - Find the iterator pointing to the middle element and use that as the end of the first sequence:

```
bool is_palindrome2(std::string const& s)
{
    auto it = s.begin();
    std::advance(it, s.size() % 2 ? s.size()/2 + 1 : s.size()/2);
    return std::equal(s.begin(), it, s.rbegin());
}

bool is_palindrome3(std::string const& s)
{
    return std::equal(s.begin(),
        std::next(s.begin(), s.size() % 2 ? s.size()/2 + 1 : s.size()/2),
        s.rbegin());
}
```

# Standard Algorithm: advance (next)

- Advance a given iterator N times:

```
void std::advance(it, n);
```

- Changes `it` in-place
- Uses most efficient implementation depending on iterator type
  - Random access iterators: uses `operator+=()`
  - Sequential iterators: uses `operator++()` - N times
- The algorithm `next` is similar, except that it returns the new iterator

# Standard Algorithm: copy

- Non-constrained algorithm (will compile only if operations are supported):

```
template <typename InIt, typename OutIt>
void copy(InIt begin, InIt end, OutIt out)
{
    while (begin != end)
        *out++ = *begin++;
}
```

- Properly constrained algorithm (will report errors on the interface):

```
template <std::input_iterator InIt, std::output_iterator OutIt>
void copy(InIt begin, InIt end, OutIt out)
{
    while (begin != end)
        *out++ = *begin++;
}
```

# What's that all about?

- Orthogonality:

```
std::vector<int> v = { 3, 1, 4, 2 };
std::for_each(v.begin(), v.end(), [](auto) { ... });

std::list<int> l = { 3, 1, 4, 2 };
std::for_each(v.begin(), v.end(), [](auto) { ... });
```

- Any algorithm is usable with any container
  - Still optimal code, because STL contains optimal implementation for each iterator type
  - Optimal code with your data structures as well

# Creating `counting_iterator`

- How can we use indices with standard algorithms?

```
// Will this print: '0 1 2 3 4 5 6 7 8 9 '? -- No
std::for_each(0, 10, [](int val) { std::print("{} ", val); });
```

- This will not compile as the algorithm expects a type that conforms to the concept of `std::input_iterator`
  - i.e. the type needs to support ++, == (and !=), and *
  - The type `int` supports only two of those operations

- Wouldn't it be nice if we were able to convert 'normal' index based loops to standard algorithms?
- Let's create an iterator type that represents an integer counter
  - Returns current value of the counter when dereferenced

# Creating `counting_iterator`

- We need a type that:
  - Exposes iterator interface (++, ==/!=, and *)
  - Represents an integer
  - Returns that integer value when dereferenced
  - Increments that integer value when incremented
  - Compares the integer values of two iterators when compared

# Creating counting_iterator

- With that knowledge, let's create a counting iterator
  - An iterator that returns ever increasing integer values

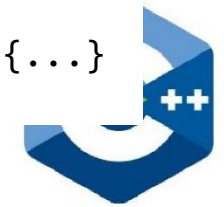
```
template <std::integral I>
class counting_iterator {
    I pos;

public:
    explicit counting_iterator(I start_at = 0) : pos(start_at) {}

    I& operator*() { return pos; }                                    // dereference

    counting_iterator& operator++() { ++pos; return *this; }          // prefix++
    counting_iterator operator++(int) { return counting_iterator(pos++); }  // postfix++

    friend bool operator==(counting_iterator const& lhs, counting_iterator const& rhs)
    {
        return lhs.pos == rhs.pos;
    }
    friend bool operator!=(counting_iterator const& lhs, counting_iterator const& rhs) {...}
};
```



# Creating counting_iterator

- Now we can write:

```
// will print: '0 1 2 3 4 5 6 7 8 9 '
std::for_each(counting_iterator(0), counting_iterator(10),
    [](int val) { std::print("{} ", val); });
```

- Very useful when converting ‘normal’ index based loops to standard algorithms

# Ranges

# Ranges are Sequences of Elements

- Three different types of ranges
  - Pairs of iterators: `[begin, end)`
  - Counted sequences: `[0, size)`
  - Conditionally-terminated sequences: `[begin, predicate)`
  - Unterminated sequences: `[0, ...)`
- Standard defines alternative range-based versions of all algorithms:

```
std::vector<int> src = {1, 2, 3, 4, 5};
std::vector<int> dest;
std::ranges::copy(src, std::back_inserter(dest));
```

- Where first argument (in this case) is a range

# Ranges are Sequences of Elements

- So, what is a range?
  - Any object that supports begin() and end():

```
template <typename T>
concept range = requires(T& t) {
    std::ranges::begin(t);
    std::ranges::end(t);
};
```

- Similarly to iterator categories, ranges are classified as:

```
std::ranges::input_range
std::ranges::output_range
std::ranges::forward_range
std::ranges::bidirectional_range
std::ranges::random_access_range
```

- Every range depends on at least one iterator, so the range category is determined by that

# Using Ranges

```
void double_even_numbers()
{
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    std::vector<int> even_numbers;
    std::copy_if(begin(numbers), end(numbers), std::back_inserter(even_numbers),
        [](int n) { return n % 2 == 0; });

    std::vector<int> results;
    std::transform(begin(even_numbers), end(even_numbers),
        std::back_inserter(results), [](int n) { return n * 2; });

    std::print("doubled even numbers (iterators): ");
    for (int n : results)
        std::print("{} ", n);
    std::println();
}
```

# Using Range Algorithms

- Algorithms like `copy_if` take pair of iterators
  - Wouldn't it be easier to just pass `numbers` to the algorithm?

# Using Range Algorithms

```
void double_even_numbers()
{
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    std::vector<int> even_numbers;
    std::ranges::copy_if(numbers, std::back_inserter(even_numbers),
        [](int n) { return n % 2 == 0; });

    std::vector<int> results;
    std::ranges::transform(even_numbers,
        std::back_inserter(results), [](int n) { return n * 2; });

    std::print("doubled even numbers (iterators): ");
    for (int n : results)
        std::print("{} ", n);
    std::println();
}
```

# Using Ranges

- Even numbers have to be stored separately: `even_numbers`
  - Why, those are intermediate results not needed for anything else?
  - Final results have to be explicitly 'materialized' as well
- Iterator-based algorithms are very flexible, but they do not compose well
- Standard introduces range based views
  - A view is a data structure that refers to a range without 'owning' the data
  - Indirectly represent iterable sequences (i.e. ranges)
  - Views are adaptors that allow to iterate over a range (external to the view), but apply additional operations while doing so
  - Views are 'lazily' evaluated, every element is calculated only when needed

# Using Range Views

```
void double_even_numbers_ranges()
{
    auto results = std::views::iota(1)
      | std::views::take(5)
      | std::views::filter([](int n) { return n % 2 == 0; })
      | std::views::transform([](int n) { return n * 2; });

    std::print("doubled even numbers (ranges): ");
    for (int n : results)
        std::print("{} ", n);
    std::println();
}
```

# Using Ranges

- Here we create an object `results` by composing various operations
  - `results` is a range as well, i.e. it exposes begin/end
  - Thus it can be iterated over (see `for(int n : results)` …)
- This does not create intermediate vectors of temporary values
- It is a chain of 'views', i.e. objects that encapsulate an operation that is executed whenever the view is iterated over
  - `iota`: each iteration returns a new integer
  - `take`: signals to be at the end after the given number of iterations
  - `filter`: calls given lambda for each value it is invoked with and passes it on only if the predicate returns true
  - `transform`: calls given lambda for each value it is invoked with and passes on the returned value

# Using Ranges

- Now very composable
- Composed range operations are iterable
  - Still based on iterators under the hood, but those are not being exposed
- No unnecessary intermediate vectors
  - More efficient than original code

# Creating counting Range

- Creating 'counting range':

```
// will print: '0 1 2 3 4 5 6 7 8 9 '
std::ranges::for_each(std::views::iota(0) | std::views::take(10),
    [](int val) { std::print("{} ", val); });
```

- All iterator-based algorithms have corresponding range based version
  - Simplifies calling those for containers (no need for begin/end)

# More Range Views

```
std::views::all          // takes all elements
std::views::filter       // takes the elements which satisfies the predicate
std::views::transform    // transforms each element
std::views::take         // takes the first N elements of another view
std::views::take_while   // takes the elements of a view as long as the predicate is true
std::views::drop         // skips the first N elements of another view
std::views::drop_while   // skips the initial elements until the predicate returns false
std::views::join         // joins a view of ranges
std::views::split        // splits a view by using a delimiter
std::views::common       // converts a view into a std::common_range
std::views::reverse      // iterates in reverse order
std::views::elements     // creates a view on the N-th element of tuples
std::views::keys         // creates a view on the first element of a pair-like values
std::views::values       // creates a view on the second elements of a pair-like values
```

# What about parallel Range Algorithms

- The C++ standard library does not implement parallel range based algorithms similar to the parallel iterator based algorithms
- However HPX does!
- For instance:
  - `hpx::ranges::for_each(hpx::execution::par, some_range, ...)`
- All parallel algorithms in HPX have range based counterparts

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP