# The C++ Standard Library, Containers and Algorithms

Lecture 5

Hartmut Kaiser

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

# Abstract

- We will look at the C++ Standard Template Library
  - A vast collection of extremely useful containers, algorithms, and supporting data structures
- This will be a whirlwind overview over certain aspects and facilities
  - Containers, Algorithms & Iterators

# Containers & Algorithms

# Containers, Algorithms

- The Standard Template Library is an extensible framework dealing with data in a C++ program.
- First, I will present the general idea, then the fundamental concepts, and finally examples of containers and algorithms.
- The key notions of sequence and iterator used to tie data together with algorithms (for general processing) are also presented.

- We can (already) write programs that are very similar independent of the data type used
  - Using an `int` isn’t that different from using a `double`
  - Using a `std::vector<int>` isn’t that different from using a `std::vector<string>`

# Common Tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **"Chocolate"**)
  - By properties (e.g., get the first elements where "**age < 64**")
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

# Ideals

- We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data
    - Finding a value in a `std::vector` isn't all that different from finding a value in a `std::list` or an array
    - Looking for a `std::string` ignoring case isn't all that different from looking at a `std::string` not ignoring case
    - Graphing experimental data with exact values isn't all that different from graphing data with rounded values
    - Copying a file isn't all that different from copying a vector

# Ideals (continued)

- Code that's
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- …

# Ideals (continued)

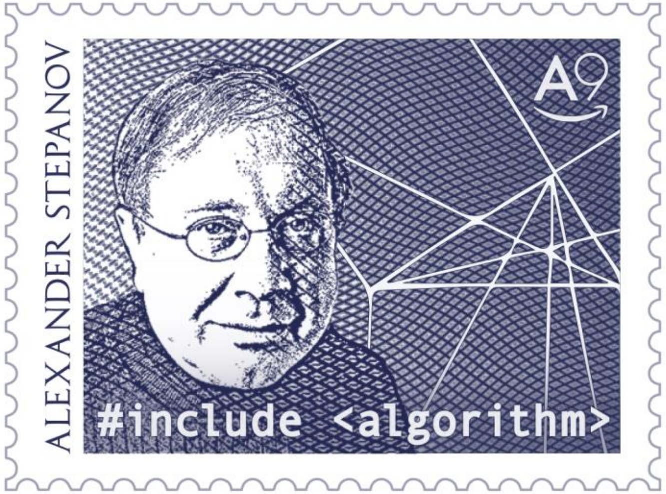
- …
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, …

# Examples

- Sort a vector of strings
- Find an number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than "Petersen"?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pair wise product of the elements of two sequences
- What's the highest temperatures for each day in a month?
- What's the top 10 best-sellers?
- What's the first entry for "C++" (say, in Google)?
- What's the sum of the elements?

# Generic Programming

- Generalize algorithms
  - Sometimes called "lifting an algorithm"
- The aim (for the end user) is
  - Increased correctness
    - Through better specification
  - Greater range of uses
    - Possibilities for re-use
  - Better performance
    - Through wider use of tuned libraries
    - Unnecessarily slow code will eventually be thrown away
    - Enable parallelizing existing code easily
- Go from the concrete to the more abstract
  - The other way most often leads to bloat



# Lifting example (concrete algorithms)

```
// one concrete algorithm (double's in array)
double sum(double array[], int n) {
    double s = 0;
    for (int i = 0; i < n; ++i)
        s = s + array[i];
    return s;
}

// another concrete algorithm (int's in list)
struct Node {
    Node* next; int data;
};

int sum(Node* first) {
    int s = 0;
    while (first != 0) {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

# Lifting Example (abstract the Data Structure)

```
// somehow parameterize with the data structure
int sum(data)
{
    int s = 0;                  // initialize
    while (not-at-end)
    {                           // loop through all elements
        s = s + get-value;      // compute sum
        get-to-next-data-element;
    }
    return s;    // return result
}
```

- We need three operations (on the data structure):
  - not at end
  - get value
  - get to next data element

# Lifting Example (STL version)

```
// Concrete STL-style code  for a more general version of both algorithms
template <typename Iter, typename T>
T sum(Iter first, Iter last, T s)
{
    while (first != last) {
        s = s + *first;
        ++first;
    }
    return s;
}
// 'Iter' should be an Input_iterator (supports ==, ++, *)
// 'T' should be something we can + and =, is the accumulator type
```

- Let the user initialize the accumulator:

```
float a[] = {1, 2, 3, 4, 5, 6, 7, 8};
double d = sum(a, a + std::size(a), 0.0f);
```

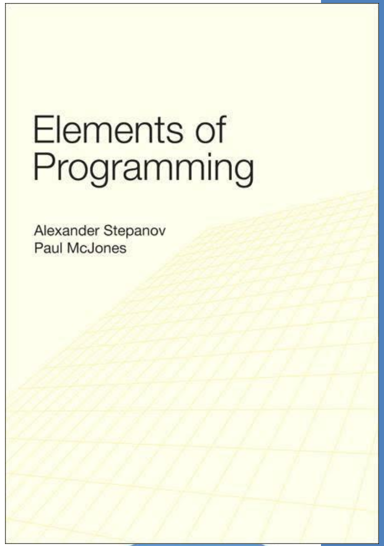
# Lifting Example

- Almost the standard library `reduce`
  - Simplified a bit for terseness
- Works for
  - C arrays
  - `std::vector`'s
  - `std::list`'s
  - `std::istream`'s
  - …
- Runs as fast as "hand-crafted" code
  - Given decent inlining
- The code's requirements on its data has become explicit
  - We understand the code better

# The C++ Standard Template Library

# The STL (Standard Template Library)

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
    - Represent separate concepts separately in code
    - Combine concepts freely wherever meaningful
- General aim to make programming “like math”
    - or even “Good programming is math”
    - works for integers, for floating-point numbers, for polynomials, for …

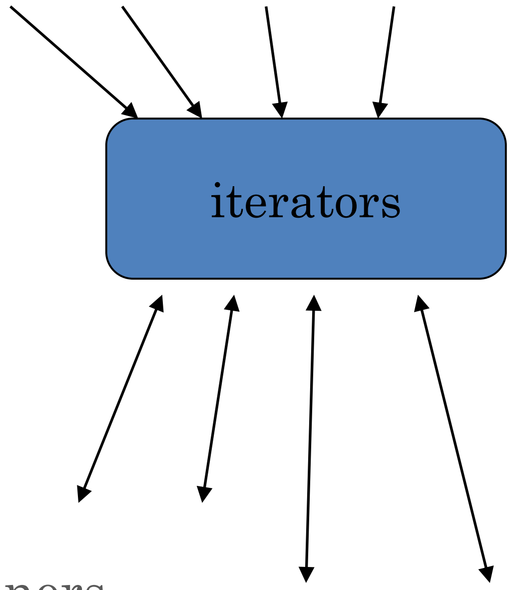



Alexander Stepanov and Paul McJones. 2009. *Elements of Programming* (1st ed.). Addison-Wesley Professional.

# 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

# The STL

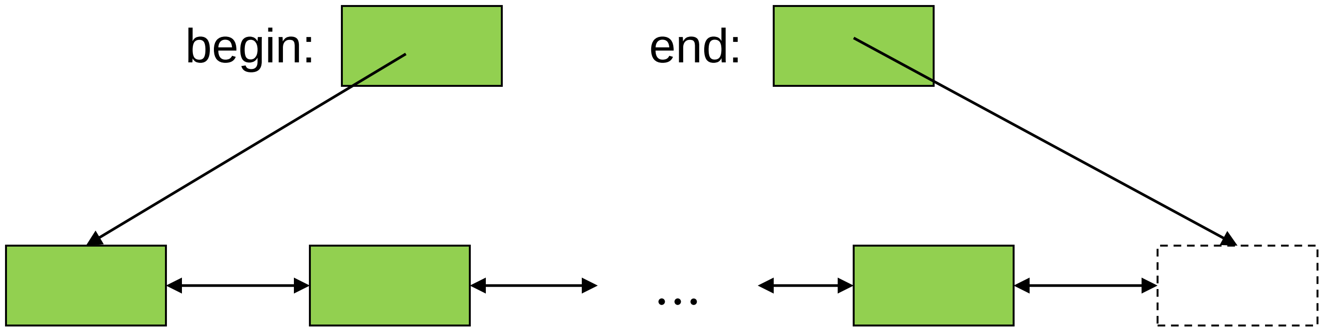
- An ISO C++ standard framework of about a dozen containers and over 100 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, SGI, …
- Probably the currently best known and most widely used example of generic programming

# The STL

- If you know the basic concepts and a few examples you can use the rest
- Documentation
  - Cpp-Reference: https://en.cppreference.com
  - Draft C++ Standard: https://eel.is/c++draft/

# Basic Model

- A pair of iterators defines a sequence (a range)
  - 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"
  - ++ Go to next element
  - * Get value of element
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. --, +, and [ ])

# Containers

# Containers

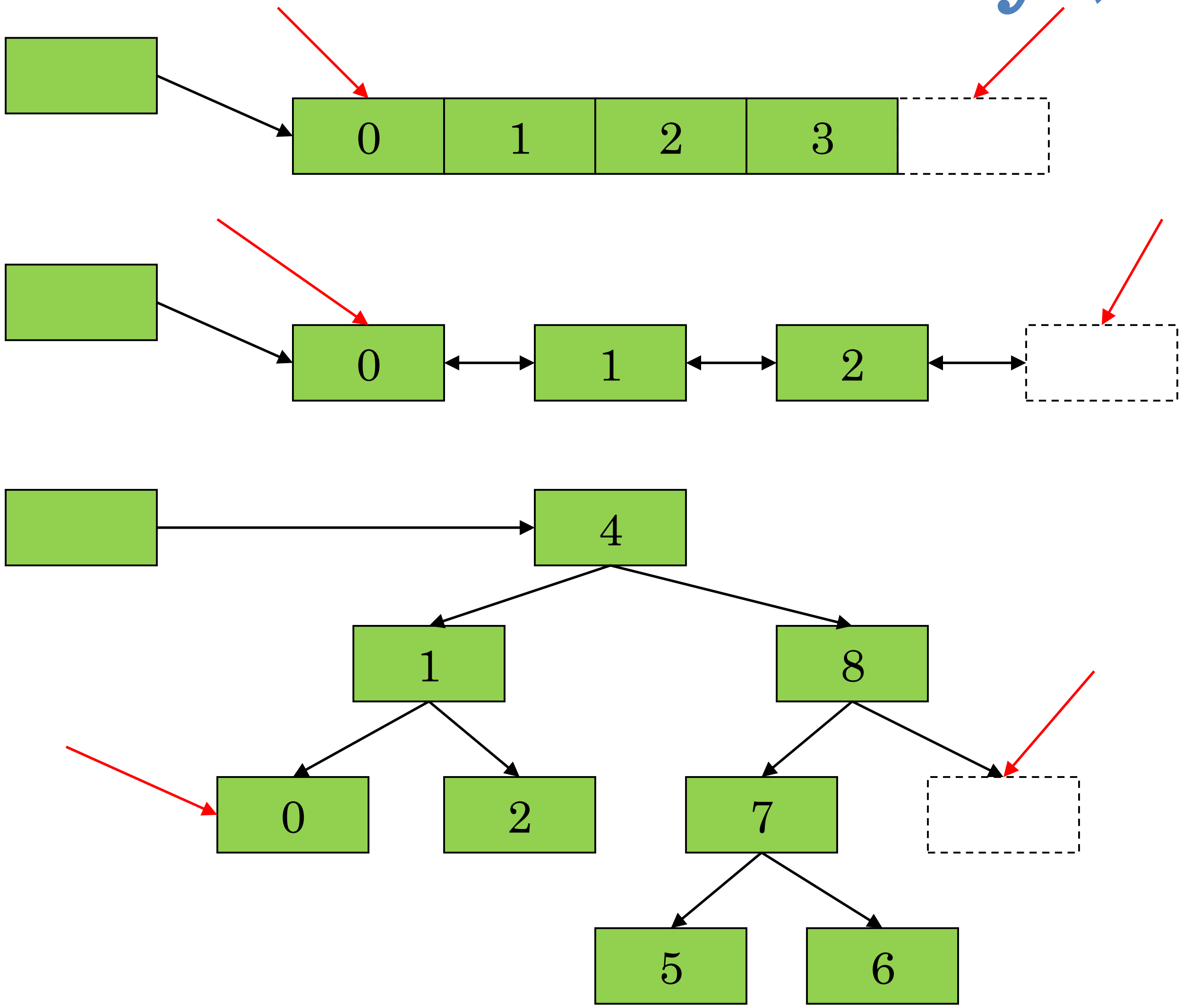
- Containers are special data structures that help storing one or more items of the same type
  - Collections of items
- Manages the storage space for its elements and provides functions to access them
- Containers replicate structures very commonly used in programming:
  - Arrays: dynamic (`std::vector`), static (`std::array`)
  - Linked lists (`std::list`, `std::forward_list`)
  - Trees (`std::set`, `std::multi_set`)
  - Hash sets (`std::unordered_set`)
  - Associative arrays (`std::map`, `std::multi_map`, `std::unordered_map`)
  - Various adaptors:
    - Queues (`std::queue`, `std::priority_queue`), stacks (`std::stack`)
    - Associative interfaces for arrays (`std::flat_map`, `std::flat_set`)

# Containers
## (hold sequences in difference ways)

std::vector

std::list

std::set



# Dynamic arrays (std::vector)

- All containers are generic types (templates) so they can be used to store items of arbitrary types
  - Are generated by type functions: `T` → `vector<T>`
- Dynamic arrays are the first choice container type
  - Rule: use `std::vector` by default
  - If you can't use `std::vector`, think again and change your approach such that you can use `std::vector`
- Rely on contiguous memory
  - Makes them `O(1)` in terms of access time
  - Insertion and deletion is `O(N)`, however (except at the end)

# Dynamic arrays (std::vector)

```cpp
// An array of integers, initialized with five elements:
std::vector<std::int64_t> data = {1, 2, 3, 4, 5};

// Elements can be accessed using the operator []
std::println("data[3] == {}", data[3]);    // data[3] == 4

// An array of arrays of integers follows the same pattern
std::vector<std::vector<std::int64_t>> data2d = {{1, 2, 3}, {2, 3}, {4, 5, 6}, {7}};

// Same logic for accessing elements
std::println("data2d[1][0] == {}", data2d[1][0]);    // data2d[1][0] == 2
std::println("data2d[2][2] == {}", data2d[2][2]);    // data2d[2][2] == 6

// We can use element access to mutate the array
data2d[3] = {1, 2, 3, 4, 5};
std::println("data2d[3][2] == {}", data2d[3][2]);    // data2d[3][2] == 3
data2d[1][0] = 42;
std::println("data2d[1][0] == {}", data2d[1][0]);    // data2d[1][0] == 42
```

# Dynamic arrays (std::vector)

- Note that accessing nonexistent elements makes the program malformed
  - Always ensure the index is valid (is in range) when accessing elements by index (i.e. `0 <= index && index < size`)
  - Various debugging tools help with identifying index-out-of range errors
    - Windows (msvc): standard library in debug builds
    - Linux, Mac (gcc, clang): address sanitizer, undefined behavior sanitizer
  - If you want to be sure, use `at()` instead `operator[]()`
    - `at()` performs index checking at runtime and throws an error if index is out of bounds

# Generic Functions

# Generic Functions

- For most of the functions we've seen so far we knew the types of its parameters
- Seems natural, however we have already used functions which didn't have that property
- For instance `std::unique()`
  - Takes two iterators
  - Usable for any appropriate type for any container
  - Implies we do not know types until we use the function
- This is called a Generic Function
  - Key feature of C++

# Generic Functions

- What exactly does it mean to have arguments of "any appropriate type"?
  - How can we know whether it will work for a given set of argument types?
- Two parts to that answer
  - Inside C++: the ways a function uses the arguments of unknown type constrains that arguments type
    - Function does x + y, this implies there is a defined operator + applicable to the types of 'x' and 'y'
    - Implementation checks whether x + y is defined, and if yes, types of 'x' and 'y' are 'appropriate'

# Generic Functions

- Two parts to that answer
  - Outside C++: the way the Standards library constrains the argument types for its functions
    - Iterators: support a collection of operations with well defined semantics
    - Function expecting iterators as arguments will use those in a way relying on the iterator semantics
    - Writing your own containers implies to write iterators exposing ‘appropriate’ operators and semantics
  - Modern C++ library implementations use concepts to constrain the types a function is usable with

# Lifting Example (STL version)

```
template <typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};

// Concrete STL-style code
template <std::input_iterator Iter, Addable T>
T sum(Iter first, Iter last, T s)
{
    while (first != last) {
        s = s + *first;
        ++first;
    }
    return s;
}
```

# Median of Unknown Type

- Generic functions are implemented using template functions
  - Single definition for a family of functions (or types) that behave similarly
  - Types are parameters, relies on knowledge that different types still have common properties and behavior
- Template functions are written in terms of this common behavior
  - When function is used a concrete type is known, which allows to compile and link program
  - Concrete types are being 'deduced' by the compiler based on the parameter types used to invoke it

# Median of vector of double's

- Here is how you could define median:

```
double median(std::vector<double> v)
{
    size_t size = v.size();
    if (size == 0)
        throw std::domain_error("median of an empty vector");
    std::sort(v.begin(), v.end());
    size_t mid = size / 2;
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```

# Median of Unknown Type

- That's what how it looks like:

```
template <typename T>
T median(std::vector<T> v)
{
    size_t size = v.size();
    if (size == 0)
        throw domain_error("median of an empty vector");
    std::sort(v.begin(), v.end());
    size_t mid = size / 2;
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```

# Median of Unknown Type

- For template functions, the type of T is deduced by the compiler when function is used:

```
std::vector<int> vi;    // = { 1, 2, 3, 4 };
median(vi);             // instantiates median with T == int

std::vector<double> vd; // = { 1.0, 2.0, 3.0, 4.0 };
median(vd);             // instantiates median with T == double
```

- The deduced template parameter type pervades the whole function:

```
return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
```

  - As `v` is a `vector<T>`, `v[mid]` is of type `T`

# Template instantiation

- When a template function is called, the compiler instantiates a version of the function based on concrete types supplied
  - Concrete types are deduced from arguments
  - Return type cannot be deduced
  - Once deduced all occurrences of those types are 'replaced' by concrete ones
- Requires the compiler to see all of the code
  - Compiler needs access to all of the sources
  - Templates are often fully defined in header files

# Generic Functions and Types

- What's an 'appropriate type' when instantiating a template
  - Median: types stored in the passed vector need to support addition and division with normal arithmetic meaning

- But subtle problems may occur

```
std::vector<double> vd; // = { 1.0, 2.0, 3.0, 4.0 };
```

  - This is ok:

```
std::find(vd.begin(), vd.end(), 0);
```

  - This is not (why?):

```
std::reduce(vd.begin(), vd.end(), 0);
```

# Generic Functions and Types

- std::find:

```
template <std::input_iterator Iterator, typename T>
Iterator find(Iterator first, Iterator last, T val)
{
    for (/**/; first != last; ++first)
        if (*first == val)
            break;
    return first;
}
```

- std::reduce:

```
template <std::input_iterator Iterator, typename T>
T reduce(Iterator first, Iterator last, T val)
{
    for (/**/; first != last; ++first)
        val = val + *first;
    return val;
}
```

# Generic Functions and Types

- `std::max` is supposed to get arguments of the same type:

```
int maxlen = 0;
maxlen = std::max(maxlen, name.size());
```

- Implementation:

```
template <typename T>
T const& max(T const& left, T const& right)
{
    return left < right ? right : left;
}
```

# Generic Functions and Types

- Why do we have that restriction (before C++11)?
- Unfortunately, this does not work:

```
template <typename T1, typename T2>
??? const& max(T1 const& left, T2 const& right)
{
    return left < right ? right : left;
}
```

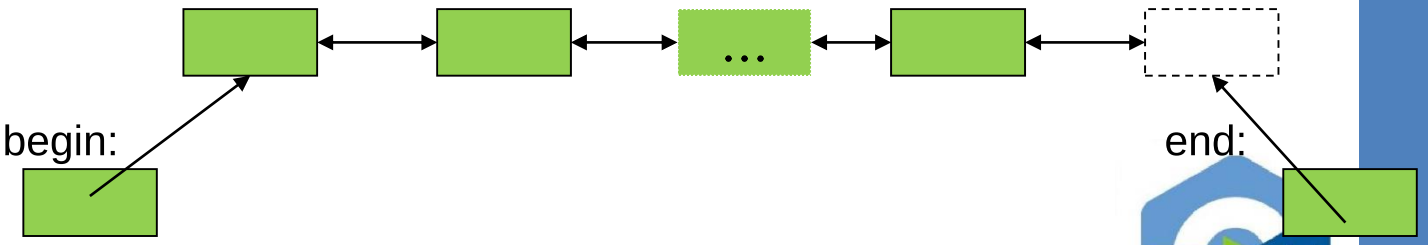
- But this does (C++11), however, it's not defined that way in toady's standard:

```
template <typename T1, typename T2>
auto max(T1 const& left, T2 const& right) ->
    decltype(left < right ? right : left)
{
    return left < right ? right : left;
}
```

# Algorithms and Iterators

# The simplest Algorithm: std::find

```
// Find the first element that is equal to a value
template <typename In, typename T> requires (std::input_iterator<In> && ...)
In find(In first, In last, T const& val)
{
    while (first != last && *first != val)
        ++first;
    return first;
}

// find an int in a given vector
void f(std::vector<int> const& v, int x) {
    auto p = std::find(v.begin(), v.end(), x);
    if (p != v.end()) {
        // we found x
        std::println("{}", *p);
    }
}
```

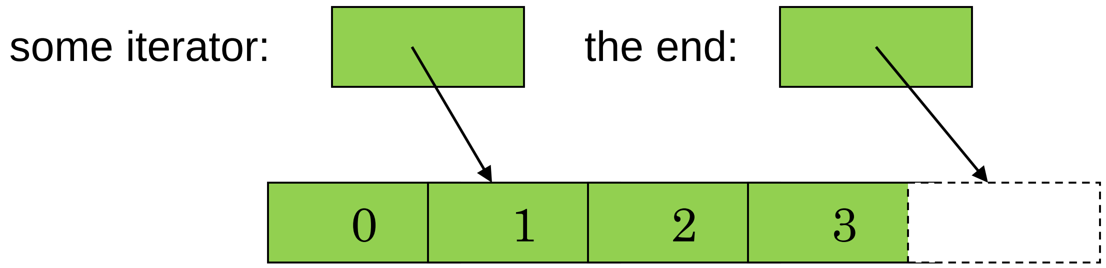


# `std::find`: Generic for both, element type and container type

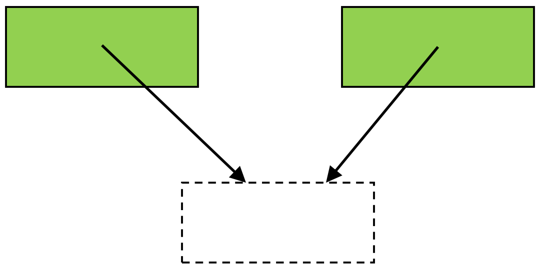
```
// works for list of strings
void f(std::list<std::string> const& l, std::string x) {
    auto p = std::find(l.begin(), l.end(), x);        // <-- here
    if (p != l.end())    // did we find x?
    {
        std::println("Found x: {}", *p);
    }
}

// works of set of doubles
void f(std::set<double> const& s, double x) {
    auto p = std::find(s.begin(), s.end(), x);        // <-- here
    if (p != s.end())    // did we find x?
    {
        std::println("Found x: {}", *p);
    }
}
```

# Algorithms and Iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is "one past the last element"
  - **not** "the last element"
  - That's necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn't an element
    - You can compare an iterator pointing to it
    - You can't dereference it (read its value)
- Returning the end of the sequence is the standard idiom for "not found" or "unsuccessful"




An empty sequence:



# Simple algorithm: `std::find_if`

- Find the first element that matches a criteria (predicate)
  - Here, a predicate takes one argument and returns a bool

```
template <typename In, typename Pred>
In find_if(In first, In last, Pred pred) {
    while (first != last && !pred(*first))
        ++first;
    return first;
}

void f(std::vector<int> const& v) {
    auto p = std::find_if(v.begin(), v.end(), odd);
    if (p != v.end())
        // we found an odd number
}
```

# Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a `bool`
- A function:

```
bool odd(int i) { return i % 2; }   // % is the remainder
odd(7);                             // call odd: is 7 odd?
```

- A function object:

```
struct Odd {
    bool operator()(int i) const { return i % 2; }
};
Odd odd;     // make an object odd of type Odd
odd(7);      // call odd: is 7 odd?
```

- A lambda function:

```
auto odd = [](int i) { return i % 2; }
odd(7);      // call odd: is 7 odd?
```

# Function Objects/Lambdas

- A concrete example using state:

```
template <typename T>
struct less_than {
    T val;    // value to compare with

    bool operator()(T const& x) const {
        return x < val;
    }
};

// find x < 43 in std::vector<int>:
p = find_if(v.begin(), v.end(), less_than{43});

// find x < “perfection” in std::list<std::string>:
q = find_if(ls.begin(), ls.end(), less_than{"perfection“});
```

# Function Objects/Lambdas

- A very efficient technique
  - inlining very easy
    - and effective with current compilers
  - Faster than equivalent function
    - And sometimes you can’t write an equivalent function
- The main method of policy parameterization in the STL
- Key to emulating functional programming techniques in C++
  - ‘Using code as data’

# Lambda

- A concrete example:

```
// find x < 43 in std::vector<int>:
p = std::find_if(v.begin(), v.end(), [](auto x) { return x < 43; });

// find x < “perfection” in std::list<std::string>:
q = std::find_if(ls.begin(), ls.end(), [](auto x) { return x < "perfection"; });
```

# Policy Parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".
  - For example, we need to parameterize sort by the comparison criteria

```
struct record {
    std::string name;    // standard string for ease of use
    char addr[24];       // old C-style string to match database layout
    // ...
};

std::vector<record> vr = {
    {"John Doe", "42 Main Street"},
    {"Donald Duck", "123 3rd Avenue"},
};

std::sort(vr.begin(), vr.end(), cmp_by_name());    // sort by name
std::sort(vr.begin(), vr.end(), cmp_by_addr());    // sort by addr
```

# Comparisons

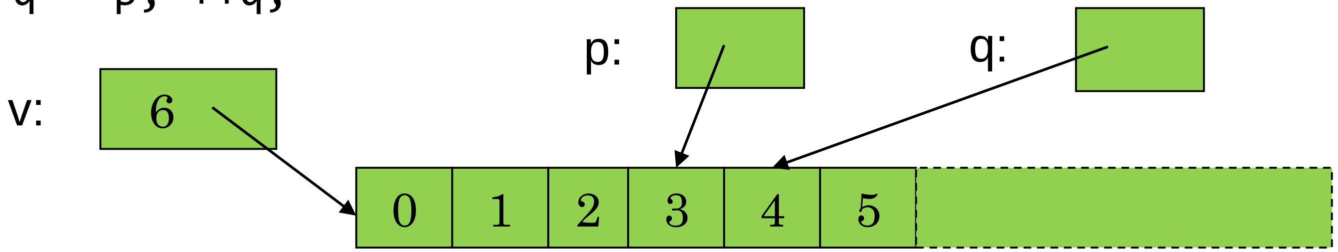
```
// Different comparisons for record objects:
struct cmp_by_name {
    bool operator()(record const& a, record const& b) const {
        return a.name < b.name; // look at the name field of Rec
    }
};

struct cmp_by_addr {
    bool operator()(record const& a, record const& b) const {
        return 0 < std::strncmp(a.addr, b.addr, 24);    // correct?
    }
};
// note how the comparison function objects are used to hide ugly
// and error-prone code
```

# std::vector

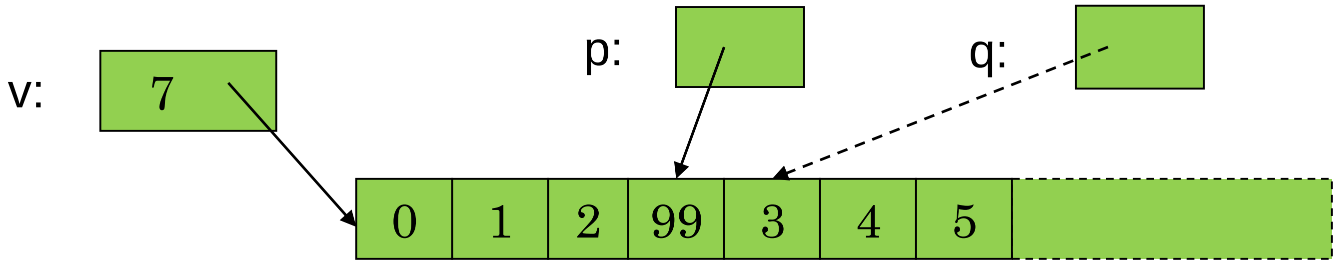
```
template <typename T>
class vector {
    T* elements;
public:
    // ...
    using iterator = ...;    // the type of an iterator is implementation defined
                             // and it (usefully) varies (e.g. range checked iterators)
    // a vector iterator could be a pointer to an element
    using const_iterator = ...;

    iterator begin();        // points to first element
    const_iterator begin() const;
    iterator end();          // points one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);    // remove element pointed to by p
    iterator insert(iterator p, T const& v); // insert a new element v before p
};
```

# insert() into `std::vector`

```
std::vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
std::vector<int>::iterator q = p; ++q;
```

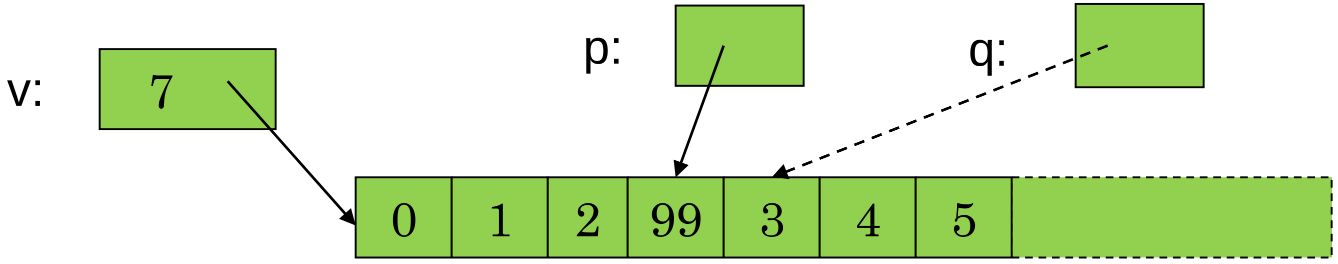



```
p = v.insert(p, 99);  // leaves p pointing at the inserted element
```

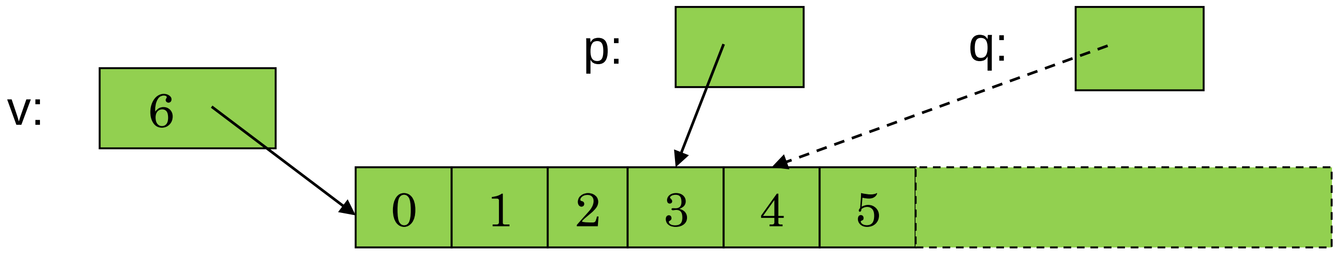



- Note: q is invalid after the **insert()**
- Note: Some elements moved; all elements could have moved

# erase() from `std::vector`




```
// leaves p pointing at the element after the erased one
p = v.erase(p);
```




- vector elements move when you insert() or erase()
- Iterators into a vector are invalidated by insert() and erase()

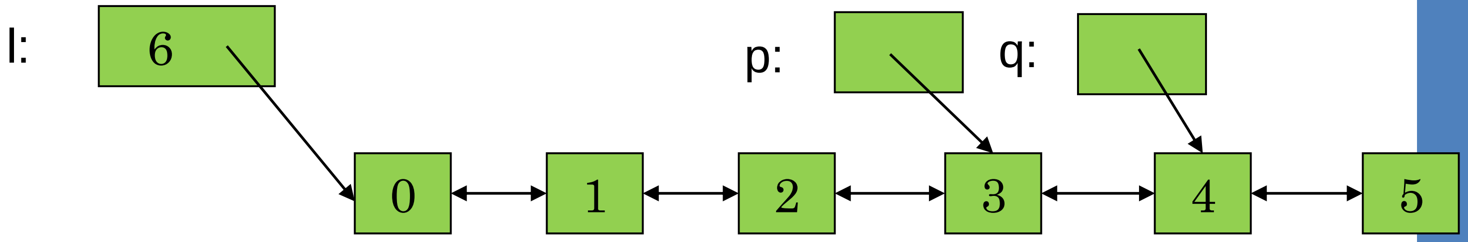
# std::list

Link: T value | Link* pre Link* post

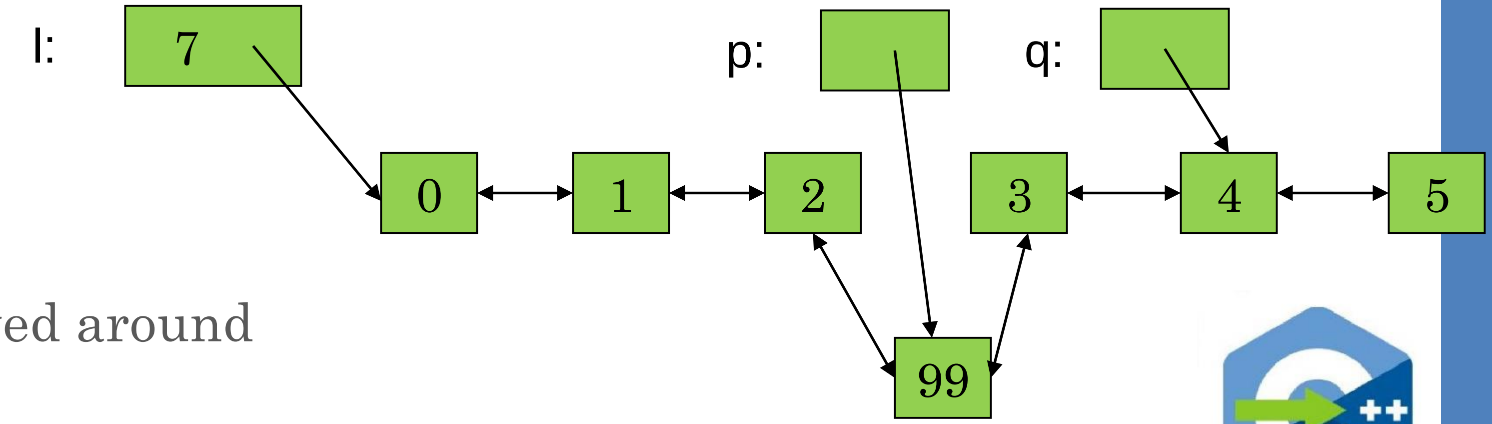
```
template <typename T>
class list {
    Link* elements;
public:
    // ...
    using iterator = ...;     // the type of an iterator is implementation defined
                              // and it (usefully) varies (e.g. range checked iterators)
        // a list iterator could be a pointer to a link node
    using const_iterator = ...;

    iterator begin();    // points to first element
    const_iterator begin() const;
    iterator end();      // points one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);     // remove element pointed to by p
    iterator insert(iterator p, const T& v);    // insert a new element v before p
};
```

# insert() into `std::list`

```
std::list<int>::iterator p = l.begin(); ++p; ++p; ++p;
std::list<int>::iterator q = p; ++q;
```

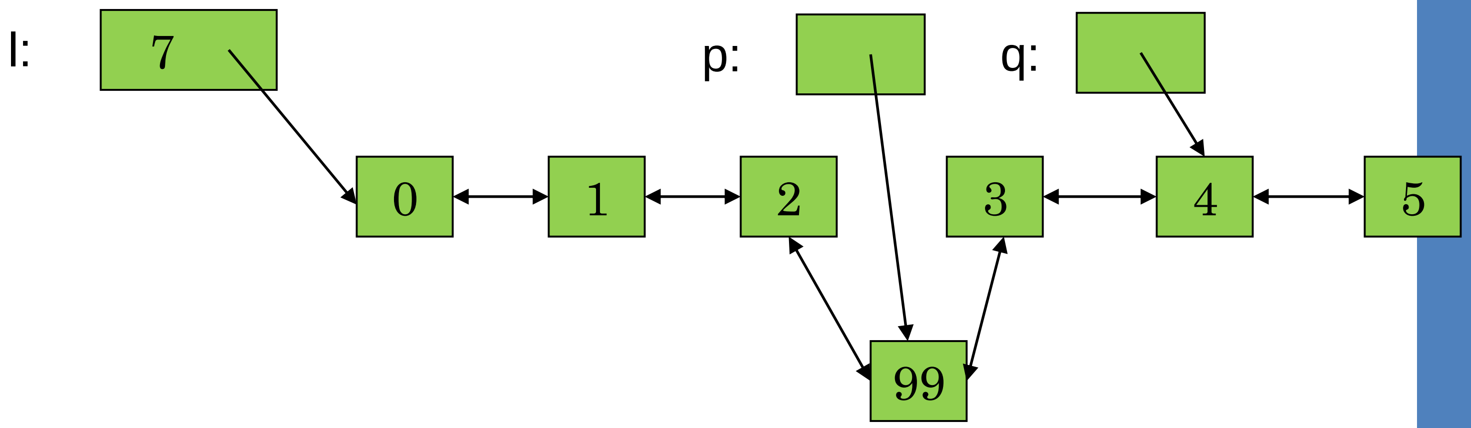



```
p = l.insert(p, 99);     // leaves p pointing at the inserted element
```

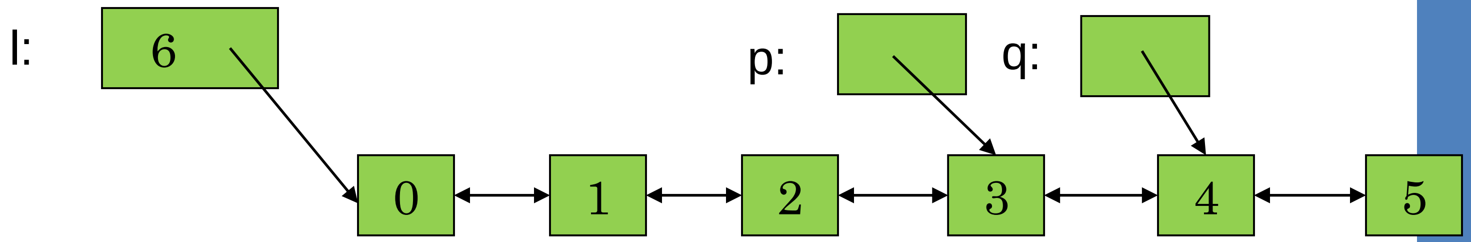



- Note: q is unaffected
- Note: No elements moved around

# erase() from `std::list`




```
// leaves p pointing at the element after the erased one
p = l.erase(p);
```




- Note: list elements do not move when you insert() or erase()

# std::array

```
template <typename T, size_t N>
class array {
    T elements[N];
public:
    // ...
    using iterator = ...;     // the type of an iterator is implementation defined
                              // and it (usefully) varies (e.g. range checked iterators)
    // a vector iterator could be a pointer to an element
    using const_iterator = ...;

    iterator begin();         // points to first element
    const_iterator begin() const;
    iterator end();           // points one beyond the last element
    const_iterator end() const;

    // no erase
    // no insert
};
```

# Parallel Algorithms

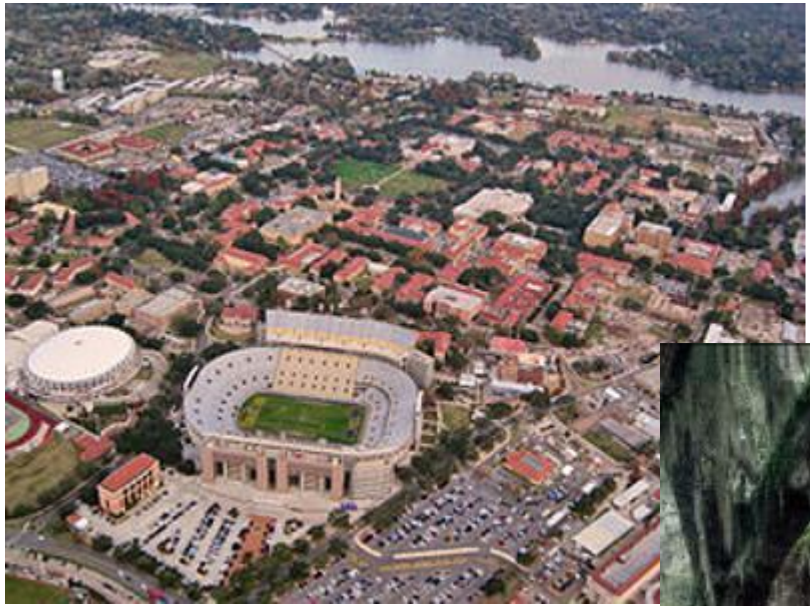
# Parallel Algorithms

- Same semantics and API as sequential algorithms
- Standard introduces: `std::seq`, `std::par`, `std::unseq` (C++20), `std::par_unseq`
  - Passed as additional first argument to algorithm
- Convey guarantees/requirements imposed by loop body
  - `seq`: execute in-order (sequenced) on current thread
  - `unseq`: allow out-of-order execution (unsequenced) on current thread - vectorization
  - `par`: allow parallel execution on different threads
  - `par_unseq`: allow parallel out-of-order (vectorized) execution on different threads

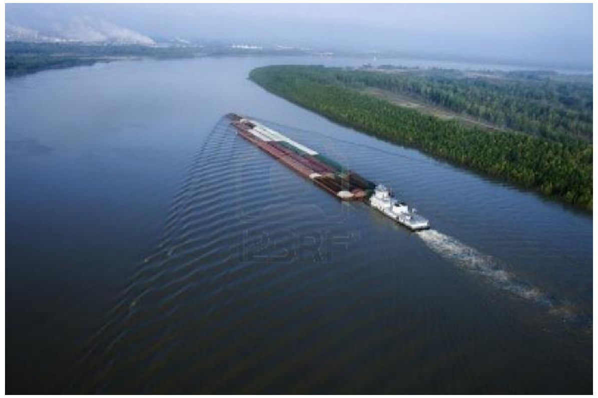
```
// Simplest case: parallel execution policy
std::vector<double> d(1000);
std::fill(std::execution::par, begin(d), end(d), 0.0);
```

# Parallel Algorithms

| | | | |
|---|---|---|---|
| adjacent_difference | adjacent_find | all_of | any_of |
| copy | copy_if | copy_n | count |
| count_if | equal | exclusive_scan | fill |
| fill_n | find | find_end | find_first_of |
| find_if | find_if_not | for_each | for_each_n |
| generate | generate_n | includes | inclusive_scan |
| inner_product | inplace_merge | is_heap | is_heap_until |
| is_partitioned | is_sorted | is_sorted_until | lexicographical_compare |
| max_element | merge | min_element | minmax_element |
| mismatch | move | none_of | nth_element |
| partial_sort | partial_sort_copy | partition | partition_copy |
| reduce | remove | remove_copy | remove_copy_if |
| remove_if | replace | replace_copy | replace_copy_if |
| replace_if | reverse | reverse_copy | rotate |
| rotate_copy | search | search_n | set_difference |
| set_intersection | set_symmetric_difference | set_union | sort |
| stable_partition | stable_sort | swap_ranges | transform |
| uninitialized_copy | uninitialized_copy_n | uninitialized_fill | uninitialized_fill_n |
| unique | unique_copy | | |



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP