High Level Parallel Programming

Programming with the STL

Alessandro Savino
Dipartimento di Automatica e Informatica
Politecnico di Torino

The Standard Library

- Provides a collection of useful C++ classes and functions
  - Is itself implemented in C++
  - Part of the ISO C++ standard
    - Defines interface, semantics and contracts the implementation has to abide by (e.g. runtime complexity)
    - Implementation is not part of the standard
    - Multiple vendors provide their own implementations
    - Best known
      - **libstdc++** (used by gcc)
      - **libc++** (used by llvm)

The Standard Library

- All features are declared within the std namespace
- Functionality is divided into sub-libraries each consisting of multiple headers
- Includes parts of the C standard library
  - For backward compatibility
  - Headers begin with “c” (e.g., cstring)
  - Never use them unless you absolutely know what you are doing!

Features Overview

- Most important library features
  - Utilities
    - Memory management (new, delete, unique_ptr, shared_ptr)
    - Error handling (exceptions, assert)
    - Time (clocks, durations, timestamps, etc.)
    - Optionals, Variants, Tuples, etc.
  - Strings
    - String class
    - String views
    - C-style string handling
  - Containers
    - Array, vector, lists, maps, sets

Features Overview

- Algorithms
  - (Stable) sort, search, max, min, ...
- Iterators
- Numerics
  - Common mathematic functions (sqrt, pow, mod, log, etc.)
  - Complex numbers
  - Random number generation
- I/O
  - Input-/Output streams
  - File streams
  - String streams

Features Overview

- Threads
  - Thread class
  - (shared) mutexes
  - Futures
- Much more
  - Localization
  - Regex
  - Atomics
  - Filesystem support
  - …

Optionals

- std::optional is a class encapsulating a value that might or might not exist
  - Defined in the header <optional>
  - Some functions might fail or return without a valid result (e.g. looking up a non-existing file)
  - It's unfavorable to encode such failures with a value of the function domain (e.g., an empty string when file could not be read)
  - It helps to express such results
    - At any point in time, an optional either has a value, or it doesn't
    - If the computation succeeded, it returns an optional containing a value

Optionals

  - If it failed, it returns an optional without a value
- The template parameter T denotes, of which type the optional may contain a value
  - For example, optional<int> might contain an int
- Guarantees to not dynamically allocate any memory when being assigned a value
- Is an object, despite supporting the dereference operators * and ->
- Internally implemented as an object with a member of type T and a boolean

Optionals: Creations

```
std::optional<std::string> might_fail(int arg) {
  if (arg == 0) {
    return std::optional<std::string>("zero");
  } else if (arg == 1) {
    return "one"; // equivalent to the case above
  } else if (arg < 7) {
    // std::make_optional takes constructor arguments of type T
    return std::make_optional<std::string>("less than 7");
  } else {
    return std::nullopt; // alternatively: return {}
  }
}
```

- Optionals are created through its constructor or with std::make_optional

Optionals: Creations

```
might_fail(3).value(); // "less than 7"

might_fail(8).value();
// throws std::bad_optional_access

*might_fail(3); // "less than 7"

might_fail(6)->size(); // 11

might_fail(7)->empty(); // undefined behavior
```

- The value of an optional can be read with value() (throws exception when empty) or dereferenced with * or -> (undefined behavior when empty)

Optionals: Checking and Accessing

```
might_fail(3).has_value(); // true
might_fail(8).has_value(); // false

// Or even simpler:
std::optional<std::string> opt5 = might_fail(5)
if (opt5) { //contextual conversion to bool
  opt5->size(); // 11
}
```

- There are multiple ways to check whether an optional has a value

Pairs

- std::pair<T, U> is a template class that stores exactly one object of type T and one of type U
  - Defined in the header <utility>
  - Constructor takes object of T and U
  - Pairs can also be constructed with std::make_pair()
  - Objects can be accessed with first and second
  - Can be compared for equality and inequality
  - Can be compared lexicographically with <, <=, >, and >=

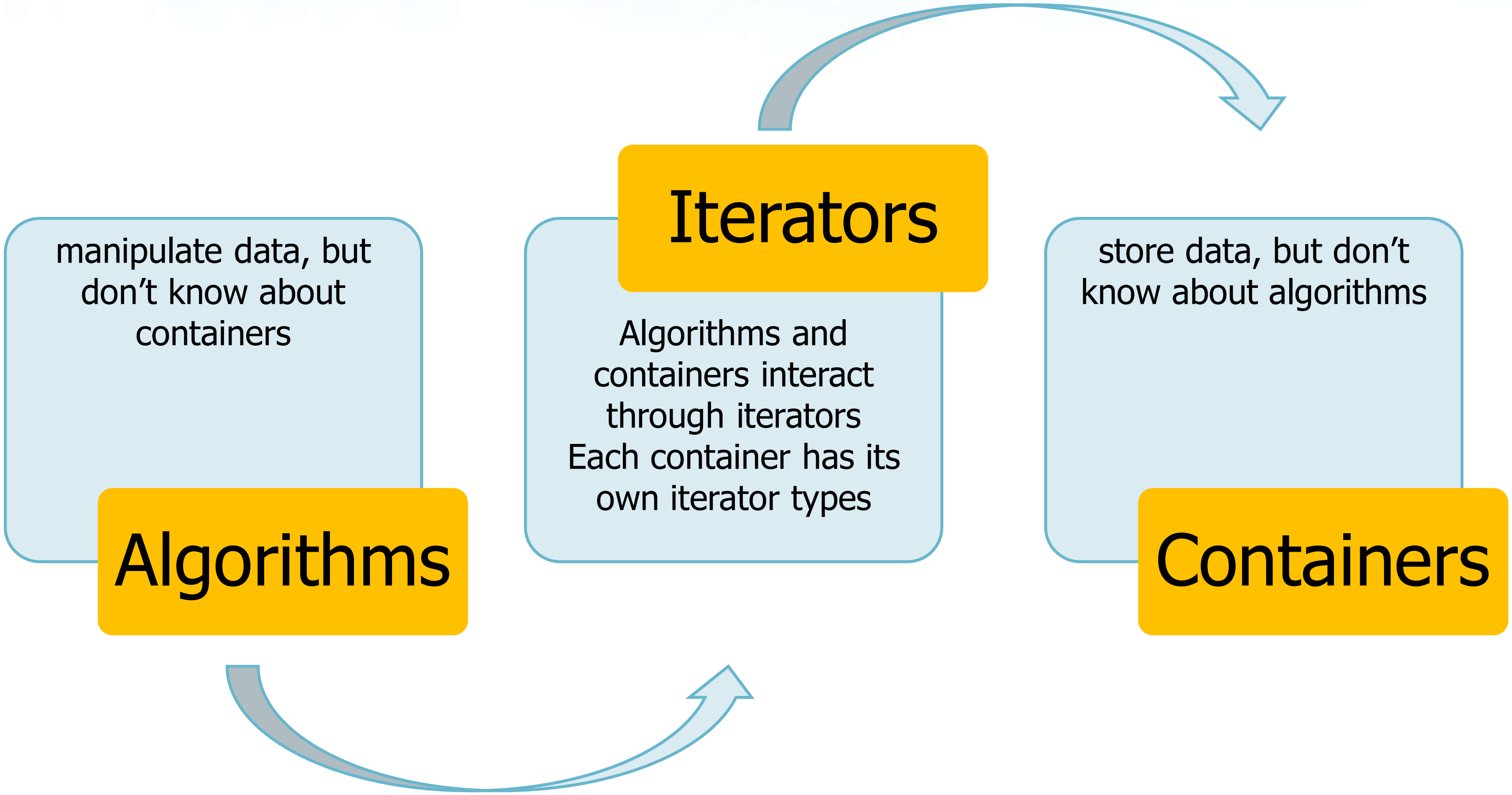
```
std::pair<int, double> p1(123, 4.56);
p1.first; // == 123
p1.second; // == 4.56

auto p2 = std::make_pair(456, 1.23);
// p2 has type std::pair<double, int>
p1 < p2; // true
```

Tuple

- std::tuple is a template class with n type template parameters that stores exactly one object of each of the n types.
  - Defined in the header <tuple>
  - Constructor takes all objects
  - Tuples can also be constructed with std::make_tuple()
  - The ith object can be accessed with std::get<i>()
  - Just like pairs, tuples define all relational comparison operators

```
std::pair<int, double, char> t1(123, 4.56, 'x');
std::get<1>(t1); // == 4.56

auto t2 = std::make_tuple(456, 1.23, 'y');
// t2 has type std::tuple<int, double, char>
t1 < t2; // true
```

The STL Basic Model



Containers

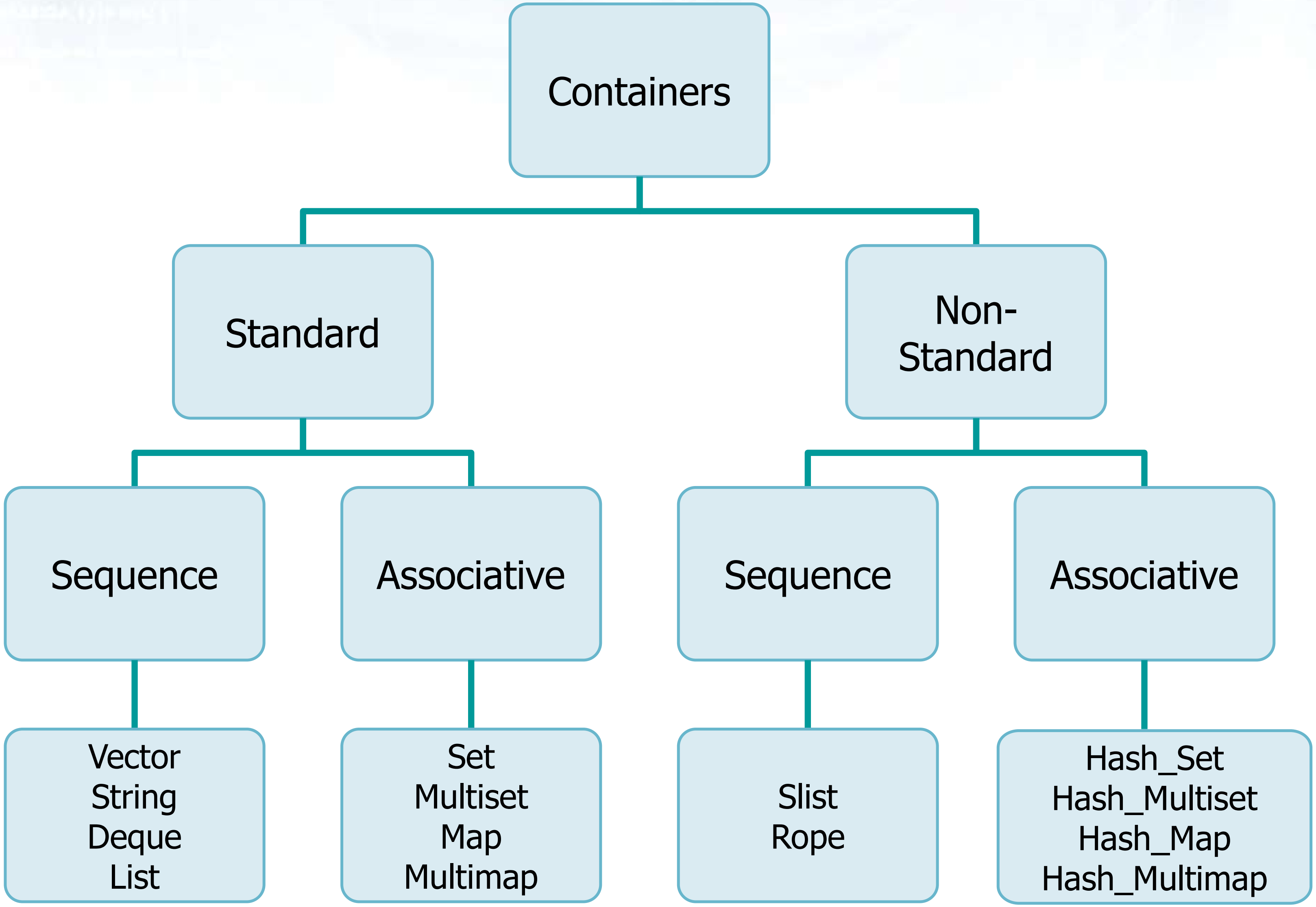
- A container is an object that stores a collection of other objects
  - Manage the storage space for their elements
  - Generic
    - The type(s) of elements stored are template parameter(s)
  - Provide member functions for accessing elements directly, or through iterators
  - (Most) member functions shared between containers

Containers

- Make guarantees about the complexity of their operations
  - Sequence containers (e.g. std::array, std::vector, std::list): Optimized for sequential access
  - Associative containers (e.g. std::set, std::map): Sorted, optimized for search (O(logn))
  - Unordered associative containers (e.g. std::unordered_set, std::unordered_map): Hashed, optimized for search
    - Amortized: O(1)
    - Worst case: O(n))

- Use containers whenever possible
  - When in doubt, use std::vector!

Types of Containers



Containers: std::vector

- Vectors are arrays that can dynamically grow
  - Defined in the header <vector>
  - Elements are still stored contiguously
  - Elements can be inserted and removed at any position
  - Pre-allocates memory for a certain amount of elements
  - Allocates new, larger chunk of memory and moves elements when memory is exhausted
  - Memory for a given amount of elements can be reserved with reserve

Containers: std::vector

- Time complexity
  - Random access
    - O(1)
  - Insertion and removal at the end
    - Typically: O(1)
    - Worst case: O(n) due to possible reallocation
  - Insertion and removal at any other position
    - O(n)
- Access to the underlying C-style data array with **data** member function

Containers: std::vector

- Vectors are constructed just like arrays
- Access elements via C-style array notation, via at(), or through a raw pointer
- Update elements via C-style array notation, via at(), or through a raw pointer
  - Note
    - It is not possible to insert new elements this way
    - You can only update existing ones

```
std::vector<int> fib = {1,1,2,3};

fib.at(0) // == 1;
fib[1] // == 1;
int* fib_ptr = fib.data();
fib_ptr[2] // == 3;

fib[3] = 43;
fib.at(2) = 42;
fib.data()[1] = 41; // fib is now 1, 41, 42, 43
```

Containers: std::unordered_map

- Maps are associative containers consisting of key-value pairs
  - Defined in the header <unordered_map>
  - Keys are required to be unique
  - At least two template parameters
    - Key and T (type of the values)
  - Is internally a hash table
  - Amortized **O(1)** complexity for random access, search, insertion, and removal

Containers: std::unordered_map

- No way to access keys or values in order (use std::map for that!)
- Accepts custom hash- and comparison functions through third and fourth template parameter

- Use std::unordered_map if you need a hash table, but don't need ordering

Containers: std::unordered_map

- Maps can be constructed pairwise
- Lookup the value to a key with C-style array notation, or with at()
- A pair can also be searched for with find
  - To check if a key exists, use count

```
std::unordered_map<std::string,double> name_to_grade
{{"maier", 1.3}, {"huber", 2.7}, {"schmidt", 5.0}};

name_to_grade["huber"]; // == 2.7
name_to_grade.at("schmidt"); // == 5.0

auto search = name_to_grade.find("schmidt");
if (search != name_to_grade.end()) {
    // Returns an iterator pointing to a pair!
    search->first; // == "schmidt"
    search->second; // == 5.0
}
name_to_grade.count("schmidt"); // == 1
name_to_grade.count("blafasel"); // == 0
```

Containers: std::map

- In contrast to unordered maps, the keys of std::map are sorted
  - Defined in the header <map>
  - Interface largely the same to std::unordered_map
  - Optionally accepts a custom comparison function as template parameter
  - Is internally a tree (usually AVL- or R/B-Tree)
  - O(log n) complexity for random access, search, insertion, and removal

Containers: std::map

- std:map also allows to search for ranges
  - upper_bound(key) returns an iterator pointing to the first **greater** element than key
  - lower_bound(key) returns an iterator pointing to the first element **not lower** than key

```
std::map<int, int> x_to_y = {{1, 1}, {3, 9}, {7, 49}};

// gt3 points to {7, 49}
auto gt3 = x_to_y.upper_bound(3);

// geq3 points to {3, 9}
auto geq3 = x_to_y.lower_bound(3);
```

Containers: Thread Safety

- Containers give some thread safety guarantees:
  - Two different containers: All member functions can be called concurrently by different threads (i.e. different containers don't share state)
  - The same container: All const member functions can be called concurrently. at(), [] (expect in associative containers), data(), front()/back(), begin()/end(), find() also count as const
  - Iterator operations that only read (e.g. incrementing or dereferencing an iterator) can be run concurrently with reads of other iterators and const member functions
  - Different elements of the same container can be modified concurrently

Containers: Thread Safety

- Be careful
  - As long as the standard does not explicitly require a member function to be sequential, the standard library implementation is allowed to parallelize it internally (see e.g. std::transform vs. std::for_each)
- **Rule of thumb**
  - Simultaneous reads on the same container are always okay
  - Simultaneous read/writes on different containers are also okay
  - Everything else requires synchronization

Iterators

- Iterators are objects that can be thought of as pointer abstractions
  - Problem
    - Different element access methods for each container
  - Therefore
    - Container types not easily exchangeable in code
  - Solution
    - Iterators abstract over element access and provide pointer-like interface

Iterators

- Allow for easy exchange of underlying container type
- The standard library defines multiple iterator types as containers have varying capabilities (random access, traversable in both directions, etc.)

- Be careful
  - When writing to a container, all existing iterators are invalidated and can no longer be used (some exceptions apply)

Iterators: An Example

```
std::vector<std::string> vec = {"one", "two", "three", "four"};

std::vector<std::string>::iterator it = vec.begin();
auto end = vec.end();

std::cout << *it; // prints "one"
std::cout << it->size(); // prints 3

std::cout << *end; // undefined behavior

++it; // Prefer to use pre-increment
std::cout << *it; // prints "two"

// prints "three,four,"
while (it != end) {
   std::cout << *it << ",";
   it++;
}

// available also in the range expression (C++11 and +)
for (auto elem : vec) {
     std::cout << elem << ","; // prints "one,two,three,four,"
}
```

Iterators: An Example

```
std::vector<std::string> vec = {"one", "two", "three", "four"};

std::vector<std::string>::iterator it = vec.begin();
auto end = vec.end();

std::cout << 
std::cout <<

std::cout <<

++it; // Pre
std::cout <<

// prints "t
while (it != end) {
   std::cout << *it <
   it++;
}

// available also in the range expression (C++11 and +)
for (auto elem : vec) {
      std::cout << elem << ","; // prints "one,two,three,four,"
}
```

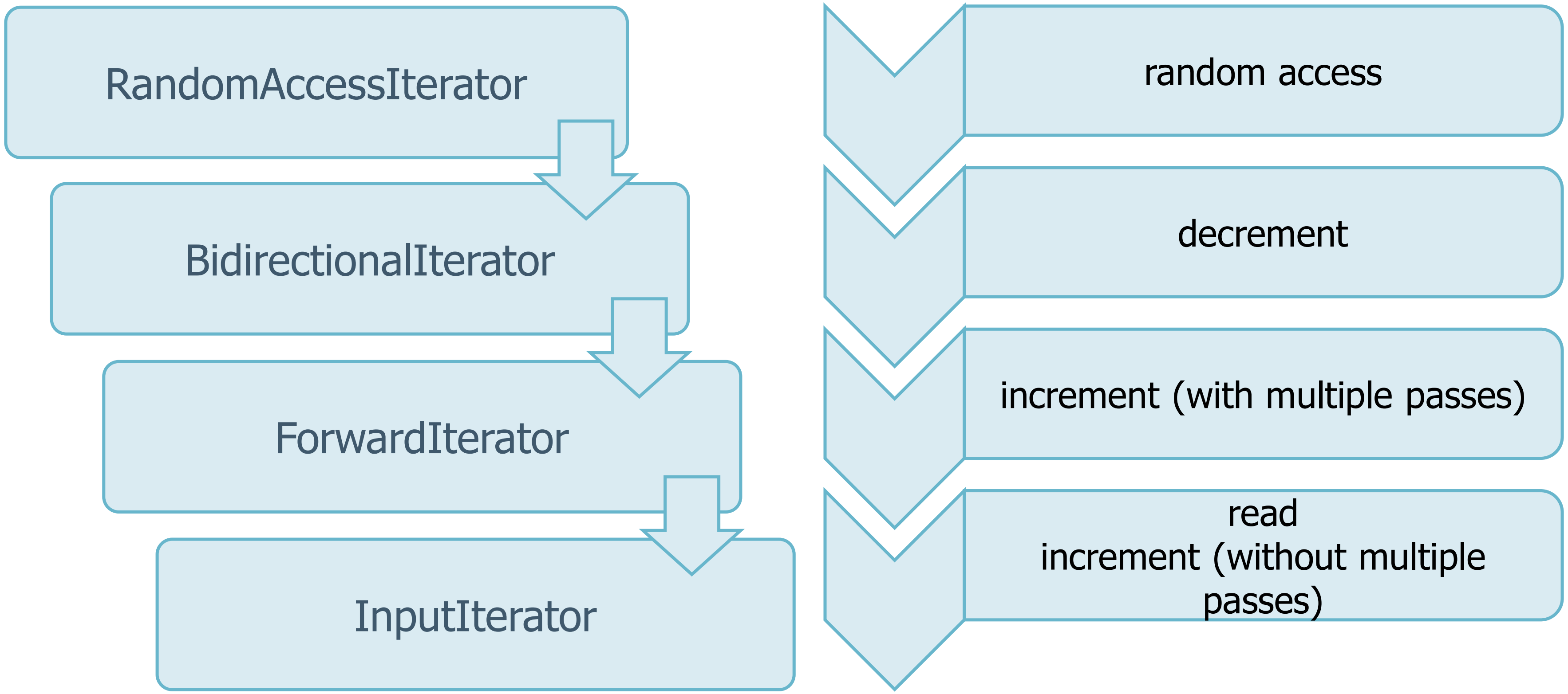
- Such a loop requires the range expression (here: vec) to have a begin() and end() member.
- vec.begin() is assigned to an internal iterator which is dereferenced, assigned to the range declaration (here: auto elem), and then incremented until it equals vec.end().

Iterators: An Advanced Example

```
std::vector<std::string> vec = {"one", "two", "three", "four"};

for (it = vec.begin(); it != vec.end(); ++it) {
   // (*it).size
   if (it->size == 3) {
        it = vec.insert(it,"foo");
        // it now points to the newly inserted element
        ++it;
   }
}

//vec == {"foo", "one", "foo", "two", "three", "four"}

for (it = vec.begin(); it != vec.end(); ++it) {
   if (it->size == 3) {
        it = vec.erase(it);
        // erase returns a new, valid iterator
        // pointing at the next element
   }
}
//vec == {"three", "four"}
```

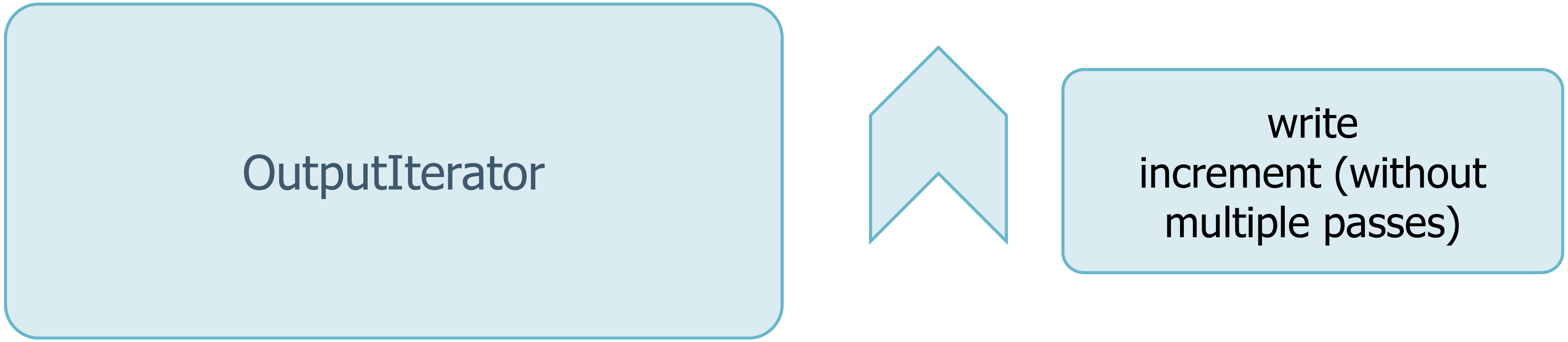
Iterators Hierarchy

- Iterators are instantiation of classes
  - It exist a hierarchy of Iterators
    - They differ by the kind of operations allowed



Iterators Hierarchy

- Iterators are instantiation of classes
  - It exist a hierarchy of Iterators
    - They differ by the kind of operations allowed



Iterators: Input and Output

- Input- and OutputIterator are the most basic iterators
- They have the following features
  - Equality comparison
    - Checks if two iterators point to the same position
  - Dereferencable with the * and -> operators
  - Incrementable, to point at the next element in sequence
  - A dereferenced **InputIterator** can only by read
  - A dereferenced **OutputIterator** can only be written to

Iterators: Input and Output

- As the most restrictive iterators, they have a few limitations
  - Single-pass only
    - They cannot be decremented
  - Only allow equality comparison, <, >=, etc. not supported
  - Can only be incremented by one (i.e., it + 2 does not work)
- Used in single-pass algorithms such as find() (InputIterator) or copy() (Copying from an InputIterator to an OutputIterator)

Iterators: Forward and Bidirectional

- ForwardIterator combines InputIterator and OutputIterator
  - All the features and restrictions shared between input- and output iterator apply
  - Dereferenced iterator can be read and written to
- BidirectionalIterator generalizes ForwardIterator
  - Additionally allows decrementing (walking backwards)
  - Therefore supports multi-pass algorithms traversing the container multiple times
  - All other restrictions of ForwardIterator still apply

Iterators: Random

- RandomAccessIterator generalizes BidirectionalIterator
  - Additionally allows random access with operator[]
  - Supports relational operators, such as < or >=
  - Can be incremented or decremented by any amount (i.e. it + 2 does work)

Function Objects

- Regular functions are not objects in C++
  - Cannot be passed as parameters (unless a function pointer is allowed
  - Cannot have state
- C++ additionally defines the FunctionObject named requirement
- For a type T to be a FunctionObject
  - T has to be an object
  - operator()(args) has to be defined for T for a suitable argument list args which can be empty
  - Often referred to as functors

Function Objects

- There are several valid function objects defined in C++
  - Pointers to functions
  - Lambda expressions
  - Stateful function objects in form of classes
- Functions and function references are not function objects
  - Can still be used in the same way due to implicit function-to-pointer conversion

Function Pointers

- While functions are not objects, they do have an address
  - Location in memory where the actual assembly code resides
  - Allows declaration of function pointers
- Function pointers to non-member functions
  - Declaration
    - return-type (*identifier)(args)
  - Allows passing functions as parameters
    - E.g., passing a custom compare function to **std::sort**
    - E.g., passing a callback to a method
- Can be invoked in the same way as a function

Function Pointers

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
  return (*func)(arg1, arg2);
}

//-----------------------------------------------------------
double callFunc(double (*func)(double), double argument) {
  return func(argument);  // Automatically dereferenced
}

//-----------------------------------------------------------
int add(int arg1, int arg2) { return arg1 + arg2; }
double add4(double argument) { return argument + 4; }

//-----------------------------------------------------------
int main() {
   auto i = callFunc(add, 2, 4); // i = 6
   auto j = callFunc(&add4, 4); // j = 8, "&" can be omitted
}
```

Lambda Expressions

- Function pointers can be unwieldy
  - Function pointers cannot easily capture environment
  - Must pass all variables that affect function by parameter
  - Cannot have "local" functions within other functions
- C++ defines lambda expressions as a more flexible alternative
  - Lambda expressions construct a closure
  - Closures store a function together with an environment
  - Lambda expressions can capture variables from the scope where they are defined

Lambda Expressions

- Lambda expression syntax

```
[ captures ] ( params ) -> ret { body }
```

  - **captures** specifies the parts of the environment that should be stored
  - **params** is a comma-separated list of function parameters
  - **ret** specifies the return type and can be omitted, in which case the return type is deduced from return statements inside the **body**

Lambda Expressions

- The list of captures can be empty
  - Results in stateless lambda expression
  - Stateless lambda expressions are implicitly convertible to function pointers
- Lambda expressions have unique unnamed class type
  - Must use auto when assigning lambda expressions to variables
  - Declaration of a lambda variable (e.g. as member) is not possible

Lambda Expression

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
     return func(arg1, arg2);
}

//----------------------------------------------------------
int main() {

   auto lambda = [](int arg1, int arg2) {
           return arg1 + arg2;
   };

   int i = callFunc(lambda, 2, 4); // i = 6

   int j = lambda(5, 6); // j = 11
}
```

Lambda Expression: Uniqueness constraint

```
// ERROR: Compilation will fail due to ambiguous return type
auto getFunction(bool first) {
  if (first) {
    return []() {
      return 42;
    };
  } else {
    return []() {
      return 42;
    };
  }
}
```

All lambda expressions have unique types

Lambda Expressions: Captures

- Lambda captures specify what constitutes the state of a lambda expression
  - Can refer to *automatic variables* in the surrounding scopes (up to the enclosing function)
  - Can refer to the "this" pointer in the surrounding scope (if present)
- Captures can either capture *by-copy* or *by-reference*
  - Capture by-copy creates a copy of the captured variable in the lambda state
  - Capture by-reference creates a reference to the captured variable in the lambda state
  - Captures can be used in the lambda expression body like regular variables or references

Lambda Expressions: Captures

- Lambda captures are provided as a comma-separated list of captures
  - By-copy
    - *identifier* or *identifier initializer*
  - By-reference
    - &*identifier* or &*identifier initializer*
      - *identifier* must refer to automatic variables in the surrounding scopes
      - *identifier* can be used as an identifier in the lambda body
  - Each variable may be captured only once

Lambda Expressions: Captures

- First capture can optionally be a capture-default
  - By-copy
    - =
  - By-reference
    - &
      - Allows any variable in the surrounding scopes to be used in the lambda body
      - Specifies the capture type for all variables without explicit captures
      - If present, only diverging capture types can be specified afterwards

Lambda Expression: Captures

```
int main() {
   int i = 0;
   int j = 42;

   auto lambda1 = [i](){};        // i by-copy
   auto lambda2 = [&i](){};       // i by-reference

   auto lambda3 = [&j, i](){};    // j by-reference, i by-copy
   auto lambda4 = [=, &i](){};    // j by-copy, i by-reference

   // ERROR: non-diverging capture types
   auto lambda5 = [&, &i](){};
   // ERROR: non-diverging capture types
   auto lambda6 = [=, i](){};
}
```

Lambda Expression: Captures

```
int main() {
   int i = 42;
   auto lambda1 = [i]() { return i + 42; };
   auto lambda2 = [&i]() { return i + 42; };

   i = 0;
   int a = lambda1(); // a = 84
   int b = lambda2(); // b = 42
}
```

Be careful, the capture is done at definition, thus if you capture by-copy the value is persistent, otherwise (by-reference) the reference is persistent

Lambda Expression: Captures

```
#include <memory>

int main() {
   auto ptr = std::make_unique<int>(4);
   auto f2 = [inner = ptr.get()]() { return *inner;};

   int a = f2(); //4
   ptr.reset();
   int b = f2(); // undefined behavior
}
```

Lifetime issues due to capturing…

Stateful Function Objects

- Situation so far
  - Functions are generally stateless
  - State must be kept in surrounding object, e.g., class instances
  - Lambda expressions allow limited state-keeping
- Function objects can be implemented in a regular class
  - Allows the function object to keep arbitrary state
  - Difference to lambda expressions
    - State can be changed during lifetime

Stateful Function Objects: std::function

- std::function is a general-purpose wrapper for all callable targets
  - Defined in the <functional> header
  - Able to store, copy and invoke the wrapped target
  - Potentially incurs dynamic memory allocations
  - Often adds unnecessary overhead
  - Should be avoided where possible

Stateful Function Objects: std::function

```
#include <functional>

//-------------------------------------------------------------
std::function<int()> getFunction(bool first){
  int a = 14;
  if (first)
    return [=]() { return a; };
  else
    return [=]() { return 2 * a; };
}

//-------------------------------------------------------------
int main() {
  return getFunction(false)() + getFunction(true)(); // 42
}
```

The Algorithms Library

- The algorithms library is part of the C++ standard library
  - Defines operations on ranges of elements [first, last)
  - Bundles functions for sorting, searching, manipulating, etc.
  - Ranges can be specified using pointers or any appropriate iterator type
  - Spread in 4 headers
    - <algorithm>
    - <numeric>
    - <memory>
    - <cstdlib>
- We will focus on <algorithm> as it bundles the most relevant parts

std::sort

- Sorts all elements in a range [first, last) in ascending order
  - void sort(RandomIt first, RandomIt last);
    - Iterators must be RandomAccessIterators
    - Elements must be swappable (std::swap or user-defined swap)
    - Elements must be move-assignable and move-constructible
  - Does not guarantee order of equal elements
  - Needs $O(n * \log(N))$ comparisons

std::sort

```
// using default comparison
// operator
#include <algorithm>
#include <vector>

//--------------------------------
int main() {
   std::vector<unsigned> v =
                       {3, 4, 1, 2};
   std::sort(v.begin(), v.end());
   // 1, 2, 3, 4
}
```

```
// using custom comparison
#include <algorithm>
#include <vector>

//--------------------------------
int main() {
   std::vector<unsigned> v =
                    {3, 4, 1, 2};
   std::sort(v.begin(),
           v.end(),
           [](unsigned lhs,
               unsigned rhs) {
               return lhs > rhs;
           }); // 4, 3, 2, 1
}
```

std::sort

```
// using default comparison
// operator
#include <algorithm>
#include <vector>

//-
int

}
```

- Sorting algorithms can be modified through custom comparison functions
  - Supplied as function objects (Compare named requirement)
- Must establish a strict weak ordering on the elements
  - Syntax:
    bool cmp(const Type1 &a, const Type2 &b);
- Return true if and only if a and be must be swaped

```
// using custom comparison
#include <algorithm>
#include <vector>

//---------------------------------
int main() {
   std::vector<unsigned> v =
                        {3, 4, 1, 2};
   std::sort(v.begin(),
           v.end(),
           [](const unsigned &lhs,
           const unsigned &rhs) {
               return lhs > rhs;
           }); // 4, 3, 2, 1
}
```

Further Sorting

- Sometimes std::sort may not be the optimal choice
  - Does not necessarily keep order of equal-ranked elements
  - Sorts the entire range (unnecessary e.g. for top-k queries)
- Keep the order of equal-ranked elements
  - std::stable_sort
- Partially sort a range
  - std::partial_sort
- Check if a range is sorted
  - std::is_sorted
  - std::is_sorted_until

Searching

- The algorithms library offers a variety of searching operations
  - Different set of operations for sorted and unsorted ranges
  - Searching on sorted ranges is faster in general
  - Sorting will pay off for repeated lookups
- Arguments against sorting
  - Externally prescribed order that may not be modified
  - Frequent updates or insertions
- General semantics
  - Search operations return iterators pointing to the result
  - Unsuccessful operations are usually indicated by returning the end iterator of a range [first, last)

Searching

- Find the first element satisfying some criteria
  - std::find
  - std::find_if
  - std::find_if_not
- Search for a range of elements in another range of elements
  - std::search
- Count matching elements
  - std::count
  - std::count_if
- Many more useful operations (see reference documentation)

std::find, std::find_if

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

//---------------------------------------------------------

int main() {
   std::vector<int> v = {2, 6, 1, 7, 3, 7};
   auto res1 = std::find(vec.begin(), vec.end(), 7);
   int a = std::distance(vec.begin(), res1);
   // a = 3 is the index distance between Iterator begin() and res1

   auto res2 = std::find(vec.begin(), vec.end(), 9);
   if(res2 == vec.end()
        std::cout << "Not found!";

   auto res1 = std::find_if(vec.begin(),
                            vec.end(),
                            [](int val) { return (val % 2) == 1; }
                            );
   int a = std::distance(vec.begin(), res1); // 2

}
```

Searching: sorted ranges

- On sorted ranges, binary search operations are offered
  - Complexity O(log(N)) when range is given as RandomAccessIterator
  - Can employ custom comparison function (see above)
    - When called with ForwardIterators complexity is linear in number of iterator increments
- Search for one occurrence of a certain element
  - std::binary_search
- Search for range boundaries
  - std::lower_bound
  - std::upper_bound
- Search for all occurrences of a certain element
  - std::equal_range
- Certain Containers already implement (some of) them!

Permutations

- The algorithms library offers operations to permute a given range
  - Can iterate over permutations in lexicographical order
  - Requires at least BidirectionalIterators
  - Values must be swappable
  - Order is determined using operator< by default
  - A custom comparison function can be supplied (see above)
- Initialize a dense range of elements
  - std::iota
- Iterate over permutations in lexicographical order
  - std::next_permutation
  - std::prev_permutation

Additional Functionalities

- The algorithms library offers many more operations
  - std::min & std::max over a range instead of two elements
  - std::merge & std::in_place_merge for merging of sorted ranges
  - Multiple set operations (intersection, union, difference, ...)
  - Heap functionality
  - Sampling of elements using std::sample
  - Swapping elements using std::swap
  - Range modifications
    - std::copy To copy elements to new location
    - std::rotate To rotate range
    - std::shuffle To randomly reorder elements
- For even more operations see the reference documentation