CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Containers, algorithms, ranges

Péter Kardos

Standard containers

- Examples: `std::vector`, `std::map`, or `std::list`
- C++ standard library: generic implementation of common data structures
- Philosophy:
  - The standard specifies the properties of containers
    - Example: search in O(log(n))
  - The standard implementation can choose any data structure for the container
  - As long as it satisfies said properties
    - Example: `std::map` may be a red-black tree or an AVL tree
- Practical use:
  - The standard library covers most cases => Don't implement data structures yourself
  - The standard library aims to be good enough most often => You may be faster in special cases
    - Example: Google's dense hash vs sparse hash vs std::unordered_map

Containers in the standard library

Sequences:

- `std::array`
- `std::vector`
- `std::deque`
- `std::forward_list`
- `std::list`

Associative:

- `std::set`
- `std::map`
- `std::multiset`
- `std::multimap`

Unordered associative:

- `std::unordered_set`
- `std::unordered_map`
- `std::unordered_multiset`
- `std::unordered_multimap`

Container adaptors:

- `std::stack`
- `std::queue`
- `std::priority_queue`

Views:

- `std::span`
- `std::mdspan`

Lesson #1: always use containers (1)

Problem:

- Create an array of increasing numbers.

Bad solution:

```cpp
int* const numbers = new int[500];
for (int i = 0; i < 100; ++i) {
    numbers[i] = i;
}
```

This code has bugs. Can you tell me what?

Lesson #1: always use containers (2)

Another solution:

```
const int count = 500;
const std::unique_ptr<int[]> numbers(new int[count]);
for (int i = 0; i < count; ++i) {
    numbers[i] = i;
}
```

Breakdown:

- No more leaks
- No more underflow/overflow
- Complicated syntax
- Cannot resize array

Lesson #1: always use containers (3)

Proper solution:

```
std::vector<int> numbers(500);
for (int i = 0; i < numbers.size(); ++i) {
    numbers[i] = i;
}
```

Why use containers:

- They don't leak memory
- Easy access to data (e.g. `operator[]`)
- Easy modification of data (e.g. `resize()`)
- Improved exception safety (strong exception safety)
- Meaningful and consistent syntax

Iterators: basic usage

Let's have a `std::vector` of numbers:

```
std::vector<int> numbers(500);
```

Iterating over the elements by **indexing**:

```
for (int i = 0; i < numbers.size(); ++i) {
    numbers[i] *= 2;
}
```

Iterating over the elements by **iterators**:

```
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    *it *= 2;
}
```

Iterators: definition

- A **range** is a generalized memory block
  - It has a beginning
  - It has an end
  - It contains a sequence of objects (<-> sequence of bytes)
- An **iterator** is a generalized pointer
  - Points to an object within a range (<-> pointer)
  - You can dereference it to get the object (`operator*`)
  - You can get the iterator to the next object in the range (`operator++`)
- Uniform interface across containers & algorithms:
  - Each container is a range
  - Delimited by the `begin()` and `end()` iterators
  - Same syntax for iterating over a container

Iterators vs. pointers

Performance:

- Iterators often compile to the same machine code as pointers
- Zero overhead in most cases

Safety:

- Checked iterators
  - Enabled in debug builds on some compilers
  - Assertion error on out of range accesses
  - Assertion error on misuse (e.g. mixing iterators of different instances)

Abstraction:

- Not restricted to linear memory (e.g. can iterate over trees, lists)

Memory allocation

Several options:

- `malloc` & `free`
- `operator new` & `operator delete`
- Custom allocators: Hoard, tcmalloc, jemalloc
- Implementing your own

Question:

- Which one does `std::vector::resize()` use to allocate memory?

Standard allocators

- A template parameter control memory allocation of containers
- Example: the declaration of the `std::vector` class:

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

- Note the template parameter `Allocator`
- By default, `std::allocator<T>` is used
- That uses `operator new/operator delete` under the hood
- You can also write your own allocator
  - You can make it use Hoard or tcmalloc

Polymorphic memory resources (PMR) (1)

Problem with allocators:

```
void my_fun(const std::vector<int, my_allocator<int>>& values);
...
std::vector<int> values;
my_fun(values); // Does not work
```

- I've forced all users of `my_fun` to use `my_allocator` too
  - Even if they hate it

Polymorphic memory resources (PMR) (2)

Solution:

```
void my_fun(const std::vector<int, std::pmr::polymorphic_allocator<int>>& values);
...
std::vector<int, std::pmr::polymorphic_allocator<int>> values(
    std::pmr::polymorphic_allocator{their_resource}
);
my_fun(values); // Work fine, uses caller's memory allocator
```

- Use `polymorphic_allocator` everywhere
- The `polymorphic_allocator` delegates actual allocation to a `memory_resource`
- `memory_resource` is a base class -> you can implement it differently
- Callers of `my_fun` can now make `my_fun` use their allocators

CSCS

ETH zürich

Polymorphic memory resources (PMR) (3)

- Memory resource:
  - Realized by the base class `std::pmr::memory_resource`
  - Implemented by derived classes:
    - `synchronized_pool_resource` (thread-safe pool)
    - `unsynchronized_pool_resource` (single-threaded pool)
    - `monotonic_buffer_resource` (stack allocator)
- Polymorphic allocator:
  - Realized by the class `std::pmr::polymorphic_allocator`
  - You can supply a `std::memory_resource` at construction
- Containers using polymorphic allocator by default:
  - `std::pmr::vector<T>`, `std::pmr::map<K, V>`, etc.
  - You can supply a `polymorphic_allocator` at construction

Allocators vs polymorphic memory resources

When to use PMR?

- Prefer PMR over traditional allocator templates -> cleaner code

Overhead of PMR?

- PMR uses virtual function calls to allocate memory
- Usual allocators may be fully inlined
- This overhead is typically negligible

Special cases: HPC, embedded:

- Don't stop using containers
- You can use PMR to customize their memory behavior to suit your needs
  - At least in most cases
- Use non-owning containers like `std::span` and `std::string_view`

Standard algorithms

- Example: `std::sort`, `std::find`, `std::binary_search`
- The standard library has generic implementations of many common algorithms
- The algorithms work on *ranges* specified by *iterators*:
  - You can use your container's `begin()` and `end()` to supply the range
  - Algorithms thus run on any suitable container
- Practical use:
  - Many real-world problems can be reduced to a common algorithm
    - Example: compare two containers with `std::inner_product`
  - This helps you avoid reinventing the wheel
  - You can expect the standard algorithms to have very few bugs
    - Unlike your 2-minute attempts at these algorithms

Algorithms in the standard library

- There are two many to fit on a slide...
- They are organized into two headers
- `<algorithm>` : general algorithms, like sorting, enumerating, etc.
    - Full list: https://en.cppreference.com/w/cpp/algorithm
- `<numeric>` : numerical algorithms, like reduce, inner product, etc.
    - Full list: https://en.cppreference.com/w/cpp/numeric
- Look them up on the internet

Lesson #2: reduce your problems to STL algorithms (1)

Problem: insert an element into a sorted vector. Vector must stay sorted.

```
void InsertSorted(std::vector<int>& range, int item) {
    ...
}
```

Lesson #2: reduce your problems to STL algorithms (2)

Solution: use binary search.

```cpp
void InsertSorted(std::vector<int>& range, int item) {
    size_t a = 0;
    size_t b = range.size();
    size_t c = (a + b) / 2;
    do {
        if (item < range[c]) {
            b = c;
        }
        else {
            a = c;
        }
        c = (a + b) / 2;
    } while (a != c);
    range.insert(range.begin() + c, item);
}
```

- I haven't actually tested this code
- I'm quite confident it does **not** work
- Looks complicated

Lesson #2: reduce your problems to STL algorithms (3)

Attempt one: use `std::sort`

```
void InsertSorted(std::vector<int>& range, int item) {
    range.push_back(item);
    std::sort(range.begin(), range.end());
}
```

- This solution very likely actually works
- It's very inefficient though
  - Sorting the whole vector is a waste

Lesson #2: reduce your problems to STL algorithms (4)

Attempt one: use `std::upper_bound`

```
void InsertSorted(std::vector<int>& range, int item) {
    const auto location = std::upper_bound(range.begin(), range.end(), item);
    range.insert(location, item);
}
```

- This solution fairly likely works
  - Though `std::lower_bound` and `std::upper_bound` are not trivial
- It's as efficient as the hand-coded
- But much simpler

CSCS

ETH zürich

Lesson #3: don't abuse your problems into STL algorithms

A simple `for` loop over a range:

```
for (auto& item : range) {
    item *= 2;
}
```

The same code using `std::for_each`:

```
std::for_each(range.begin(), range.end(), [](auto& item){
    item *= 2;
})
```

- The STL algorithm is actually less readable here
- But you could also use parallel `for_each`
- Do what's best for your codebase

Range: definition

Ranges are now formalized as a C++20 concept:

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

In plain text, an object is a range if you can:

- call `ranges::begin` on it
- call `ranges::end` on it.
- The containers mentioned before are ranges
- Some container adaptors are not

CSCS ETH zürich

Algorithms on ranges

The syntax for algorithms is quite verbose:

```
std::sort(numbers.begin(), numbers.end());
```

Ranges simplify the syntax:

```
std::ranges::sort(numbers);
```

- A lot of algorithms have a `ranges` version
- They operate on ranges, not pairs of iterators
- Since containers are ranges, they work directly on entire containers

Views (range adaptors)

Let's take a list of numbers:

```
std::vector<int> numbers = {...};
```

Let's try to double each number **using algorithms**:

```
const auto doubleFunc = [](auto v) { return 2 * v; };

std::vector<int> doubledData = {...};
std::ranges::transform(numbers, std::back_inserter(doubledData), doubleFunc);
```

Let's do the same **using views**:

```
const auto doubledView = std::views::transform(numbers, doubleFunc);
```

- **Algorithms are eager:** they process every single element in the range immediately
- **Views are lazy:** they process an element only when you dereference the iterator to it

Example: lazy evaluation (1)

Let's use the same function object to double numbers:

```
const auto transformFunc = [](auto v) {
    std::cout << "(processing " << v << ")";
    return 2 * v;
};
```

This time it also prints when it's being called.

Example: lazy evaluation (2)

Using the traditional transform algorithm:

```
std::vector<int> doubled;
std::ranges::transform(
    numbers,
    std::back_inserter(doubled),
    transformFunc);

for (auto v : doubled) {
    std::cout << v << " ";
}
```

Output:

```
(processing 1) (processing 2) (processing 3) 2 4 6
```

Analysis: first, all items are processed when calling `transform`, then all are printed.

Using the new transform view:

```
const auto doubled = std::views::transform(
    numbers,
    transformFunc);

for (auto v : doubled) {
    std::cout << v << " ";
}
```

Output:

```
(processing 1) 2 (processing 2) 4 (processing 3) 6
```

Analysis: none of the items are processed when creating the view. They are evaluated lazily as we dereference the iterator.

Example: composition of views

- `iota` gives us an infinite range of numbers 1, 2, 3, ...
- `transform` can be used to modify each element in the sequence
  - First they are squared
  - Then we take the reciprocal
- `take` takes the first one million elements from this infinite range

```
const auto series =
    std::views::iota(1LL)
    | std::views::transform([](auto v) { return v * v; })
    | std::views::transform([](auto v) { return 1.0 / v; })
    | std::views::take(1'000'000);
```

Questions:

- What concept does `series` satisfy?
- What are the first 3 elements of `series`?
- Bonus: what do you get if you sum the elements of `series`?

CSCS

ETH zürich

Remarks

- The C++ standard library is focused on data structures and algorithms
- Both data structures and algorithms are generic and composable
- Many real-world problems can be modeled in terms of these data structures and algorithms
- Use the standard library as much as you can
    - Saves you development time
    - Improves code quality (e.g. fewer bugs, fewer lines)
    - May improve performance: `std::stable_sort` likely outperforms your `half_assed_sort` ™
- Expect to outperform the standard library for specific cases
    - But only after a lot of investment from you

References

- https://en.cppreference.com/w/cpp/container
- https://en.cppreference.com/w/cpp/algorithm
- https://en.cppreference.com/w/cpp/numeric

Resources

Get the slides and full source code on GitHub:

https://github.com/eth-cscs/cpp-course-2023