

# C++ Cheatsheet

Daniel Sinkin

February 16, 2026

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Changes per version</b>	<b>5</b>
2.1	C++26 . . . . .	5
2.1.1	Contracts . . . . .	5
2.1.2	Anonymous values . . . . .	5
2.2	C++23 . . . . .	5
2.2.1	std::expected . . . . .	5
2.2.2	std::print, std::println . . . . .	6
2.2.3	std::generator . . . . .	6
2.3	C++20 . . . . .	6
2.3.1	std::format . . . . .	6
2.3.2	Concepts . . . . .	6
2.3.3	Ranges . . . . .	6
2.3.4	Coroutines . . . . .	6
2.3.5	Modules . . . . .	7
2.3.6	Calendar support for std::chrono . . . . .	7
2.4	C++17 . . . . .	7
2.4.1	CTAD (Class Template Argument Deduction) . . . . .	7
2.4.2	std::scoped_lock . . . . .	7
2.5	C++14 . . . . .	7
2.5.1	constexpr loops and conditionals . . . . .	7
2.6	C++11 . . . . .	8
2.6.1	Value Semantics . . . . .	8
2.6.2	constexpr . . . . .	8
2.6.3	Lambdas . . . . .	8
2.6.4	std::unordered_map . . . . .	8
2.6.5	std::lock_guard . . . . .	8
<b>3</b>	<b>C++ Concepts</b>	<b>9</b>
3.1	Value Semantics . . . . .	9
3.1.1	Perfect Forwarding . . . . .	9
3.2	Resource Ownership . . . . .	10
3.2.1	Rule of Five . . . . .	10
3.2.2	Rule of Zero . . . . .	10
3.2.3	RAII (Resource Acquisition is Initialisation) . . . . .	10
3.3	NRVO (Named Return Value Optimisation) . . . . .	11
3.4	EBO (Empty Base Optimisation) . . . . .	12
3.5	Idioms and Patterns . . . . .	13
3.5.1	Two-pointer merge . . . . .	13
3.5.2	Lazy Delete Idiom / Tombstoning . . . . .	13
3.5.3	Scope Guard / Defer Pattern . . . . .	13
3.5.4	Unsigned reverse iteration idiom ( $i-- > \emptyset$ ) . . . . .	14
3.6	ODR (One Definition Rule) . . . . .	14
3.7	SIOF (Static Initialisation Order Fiasco) . . . . .	14
3.8	Important STL algorithms . . . . .	14

3.9	Templates . . . . .	14
3.9.1	Variadic Templates . . . . .	14
3.9.2	Template Metaprogramming . . . . .	14
<b>4</b>	<b>STL (Standard Templating Library)</b>	<b>15</b>
4.1	Iterators . . . . .	15
4.1.1	Iterator Categories . . . . .	15
4.2	Ranges . . . . .	15
4.2.1	Core Concepts . . . . .	15
4.2.2	Views . . . . .	15
4.2.3	Range Algorithms . . . . .	15
4.2.4	Range Concepts . . . . .	15
4.3	Containers . . . . .	15
4.4	Algorithms . . . . .	15
<b>5</b>	<b>Data Structures</b>	<b>16</b>
5.1	std::vector (Trivially Copyable Types) . . . . .	16
5.1.1	Rule of 5 . . . . .	16
5.1.2	Public API . . . . .	17
5.1.3	Private API . . . . .	18
5.2	std::vector (General) . . . . .	18
5.3	std::unique_ptr . . . . .	18
5.3.1	Rule of 5 . . . . .	18
5.3.2	Public API . . . . .	19
5.3.3	Custom Deleter . . . . .	19
5.3.4	Make Unique . . . . .	20
5.4	std::optional . . . . .	20
5.4.1	Public API . . . . .	20
5.4.2	Rule of 5 . . . . .	21
5.4.3	Private API . . . . .	21
5.5	Tree . . . . .	22
5.6	Binary Tree . . . . .	22
5.7	Complete Binary Tree . . . . .	23
5.8	std::priority_queue (Max Heap) . . . . .	23
5.9	std::unordered_map . . . . .	23
5.10	Binary Search Tree (BST), Binary Search . . . . .	23
5.11	Red-Black Tree . . . . .	24
5.12	std::map . . . . .	24
5.13	std::array . . . . .	24
5.14	std::string . . . . .	24
5.15	std::deque . . . . .	24
5.16	std::pmr::memory_resource . . . . .	24
5.17	std::weak_ptr, std::shared_ptr, Control Block . . . . .	24
5.18	std::variant . . . . .	24
<b>6</b>	<b>Algorithms</b>	<b>25</b>
6.1	DFS (Depth First Search) . . . . .	25
6.2	BFS (Breadth First Search) . . . . .	25
<b>7</b>	<b>Interview Specifics</b>	<b>26</b>
7.1	Tricks . . . . .	26
7.1.1	2 Pointer Palindrome iteration . . . . .	26
7.1.2	Reverse Heap for bounded count . . . . .	26
7.1.3	Debug Macro . . . . .	27
7.1.4	Rolling Average . . . . .	27
7.1.5	Partition-based Binary Search (Two Sorted Arrays Median) . . . . .	28
7.1.6	Prefix Sums . . . . .	28
7.2	Problems . . . . .	29
7.2.1	Rotting Fruit . . . . .	29

7.3	Clone Graph . . . . .	30
7.3.1	Maximum Subarray . . . . .	30
7.3.2	Median of Two Circularly Sorted Logs . . . . .	31
7.3.3	Lazy Deletion Stack . . . . .	33
7.3.4	Subarrays with Given Sum and Bounded Maximum . . . . .	34
7.4	Maximize Profit with Task Deadlines and Multiple Servers . . . . .	35
<b>8</b>	<b>Software Engineering</b>	<b>37</b>
8.1	Testing, Unit Tests, TDD (Test Driven Development) . . . . .	37
8.2	Design Patterns . . . . .	37
8.2.1	Visitor Pattern . . . . .	37
8.2.2	Strategy Pattern . . . . .	37
8.2.3	CRTP (Curiously Recurring Template Pattern) Design Pattern . . . . .	37
8.2.4	Type Erasure Design Pattern . . . . .	37
<b>9</b>	<b>Databases</b>	<b>38</b>
9.1	MySQL . . . . .	38
9.2	MongoDB . . . . .	38
<b>10</b>	<b>Computer Architecture, HPC (High Performance Computing)</b>	<b>39</b>
10.1	Memory Hierarchy and Cache . . . . .	39
10.1.1	Cache Lines and Cache Locality . . . . .	39
10.1.2	Spatial vs Temporal Locality . . . . .	39
10.1.3	False Sharing . . . . .	39
10.2	Memory Layout and Alignment . . . . .	40
10.2.1	Alignment vs Size . . . . .	40
10.2.2	Tight Packing vs Padding . . . . .	40
10.2.3	AoS vs SoA vs AoSoA . . . . .	40
10.2.4	SIMD Alignment Requirements . . . . .	42
10.2.5	Practical Trade-offs (Bandwidth vs Compute) . . . . .	42
10.3	CPU Microarchitecture . . . . .	42
10.3.1	Pipelining . . . . .	42
10.3.2	Branch Prediction and Speculative Execution . . . . .	42
10.3.3	Out-of-Order Execution . . . . .	42
10.4	GPU Architecture . . . . .	42
10.4.1	Thread Blocks . . . . .	42
10.4.2	Warps . . . . .	42
10.4.3	Memory Coalescing . . . . .	42
10.5	CPU-GPU Interaction . . . . .	42
10.5.1	Asynchronous Execution . . . . .	42
10.5.2	Synchronization Primitives . . . . .	42
10.6	Benchmarking and Measurement . . . . .	42
10.6.1	Sampling Benchmarks . . . . .	42
10.6.2	Cache and Memory Profiling (perf, Valgrind) . . . . .	42
<b>11</b>	<b>Concurrency</b>	<b>43</b>
11.1	C++ Memory Model . . . . .	43
11.1.1	Atomics . . . . .	43
11.2	Multiple Threads . . . . .	43
11.3	Multiple Processes . . . . .	43
11.3.1	OpenMP . . . . .	43
11.4	Data Structures . . . . .	43
11.4.1	SPSC (Single Producer Single Consumer) . . . . .	43
11.4.2	SPMC (Single Producer Multiple Consumer) . . . . .	43
11.4.3	MPSC (Multiple Producer Single Consumer) . . . . .	43
11.4.4	MPMC (Multiple Producer Multiple Consumer) . . . . .	43

<b>12 Operating Systems</b>	<b>44</b>
12.1 Process, Thread . . . . .	44
12.2 Scheduling . . . . .	44
12.3 CPU Virtualisation . . . . .	44
12.4 Memory Virtualisation . . . . .	44
<b>13 Networking</b>	<b>45</b>
13.1 OSI Model . . . . .	45
13.2 UDP . . . . .	45
13.3 TCP/IP . . . . .	45
<b>14 Trivia</b>	<b>46</b>
14.1 Error #323 on GCC . . . . .	46
<b>15 References</b>	<b>47</b>

# 1 Introduction

This document contains my notes on C++ specifics I'm reviewing for my job application.

## 2 Changes per version

### 2.1 C++26

#### 2.1.1 Contracts

Will introduce Contracts which give an explicit syntax to implement post and pre conditions (formalising `gsl::Requires` and `gsl::Ensures`) as well as a new more stable assert syntax in form of `contract_assert`.

Listing 1: Pre-C++26: `gsl::Requires/gsl::Ensures + assert`

```
1 auto div_round_down_pos(std::int32_t a, std::int32_t b) -> std::int32_t
2 {
3     gsl::Requires(a > 0);
4     gsl::Requires(b > 0);
5
6     const std::int32_t r = a / b;
7     assert((r + 1) * b > a);
8
9     Ensures(r * b <= a);
10    return r;
11 }
```

Listing 2: C++26: Contract syntax

```
1 auto div_round_down_pos(std::int32_t a, std::int32_t b) -> std::int32_t
2     pre (a > 0)
3     pre (b > 0)
4     post (r * b <= a)
5
6     contract_assert((r + 1) * b > a);
7     const std::int32_t r = a / b;
8     return r;
9 }
```

#### 2.1.2 Anonymous values

Sometimes we don't care about the name of a variable but still have to assign it, in the past you'd need to make up a (unique) name for the variable, now you can just use `_`.

Listing 3: C++26: Anonymous values

```
1 auto& [_, value] = f(); // Structured binding
2
3 Mutex m1{}, m2{};
4 {
5     std::scoped_lock _{m1, m2}; // RAI utils
6 }
```

### 2.2 C++23

#### 2.2.1 `std::expected`

Allows for functional / Rust style errors as values.

Listing 4: C++26: Anonymous values

```
1 enum class MyError {
2     negative_number;
3     zero_division;
```

```

4 }
5
6 std::expected<float, PositiveDivisionError> my_div(float a, float b) {
7     if((a * b) <= 0.0f) {
8         return std::unexpected{MyError::negative_number};
9     }
10    if(b == 0.0f) {
11        return std::unexpected{MyError::zero_division};
12    }
13    return a / b;
14}
15
16 int main() {
17     auto res = my_div(5.0, 3.0);
18     if(!res) {
19         switch(res.err()) {
20             case MyError::negative_number: /*...*/
21             case MyError::zero_division: /*...*/
22         }
23     } else {
24         float value{*res};
25         /* ... */
26     }
27 }
```

## 2.2.2 std::print, std::println

Introduced **print** which allows for structured printing by leveraging the C++20 feature **std::format**.

Listing 5: std::println

```
1 std::println("Hello, {}. The value of x is {}", "World", 5);
```

## 2.2.3 std::generator

<https://www.youtube.com/watch?v=7ZazVQB-RKc>

Can access like normal iterators

## 2.3 C++20

### 2.3.1 std::format

Introduced **std::format** which allows for formatting of variables into strings.

Listing 6: std::format

```
1 float x = 12.52343232f;
2 std::string s{std::format("x = {:.2f}", x)}; // s == "x = 12.52"
```

### 2.3.2 Concepts

Compile time constraints on templates, reducing the need for SFINAE boilerplate

Listing 7: std::format

```
1 template <std::integral T>
2 T add(T a, T b) { return a + b; }
```

### 2.3.3 Ranges

### 2.3.4 Coroutines

```
1 co_await task;
2 co_yield value;
```

### 2.3.5 Modules

Adoption of those is horrible so far.

### 2.3.6 Calendar support for std::chrono

Listing 8: Pre-C++17: std::array without CTAD

```
1 using namespace std::chrono;
2 auto zt = zoned_time{"Europe/Berlin", system_clock::now()};
```

## 2.4 C++17

### 2.4.1 CTAD (Class Template Argument Deduction)

Made template type deduction possible for class templates.

Listing 9: Pre-C++17: std::array without CTAD

```
1 #include <array>
2
3 std::array<int, 3> values{{1, 2, 3}};
```

Listing 10: C++17: std::array with CTAD

```
1 #include <array>
2
3 std::array values{1, 2, 3}; // std::array<int, 3>
```

### 2.4.2 std::scoped\_lock

An improvement on `std::lock_guard` which allows to lock multiple mutexes in one call.

Listing 11: std::scoped\_lock

```
1 std::mutex m1, m2;
2 {
3     std::scoped_lock locks{m1, m2};
4 }
```

## 2.5 C++14

### 2.5.1 constexpr loops and conditionals

Added support for loops and if/else branches.

Listing 12: C++14: constexpr with loops and conditionals

```
1 constexpr int sum_up_to(int n)
2 {
3     int result = 0;
4
5     for (int i = 1; i <= n; ++i)    // constexpr loop (C++14)
6     {
7         if (i % 2 == 0)            // constexpr conditional (C++14)
8         {
9             result += i;
10        }
11    }
12
13    return result;
14 }
```

## 2.6 C++11

### 2.6.1 Value Semantics

Added move semantics and rvalue references.

Listing 13: Move constructors, Move assignment

```
1 std::vector<int> a = make_vector();
```

Listing 14: RAII

```
1 struct File {
2     File(const char* path);
3     ~File();
4     File(File&&);
5     File& operator=(File&&);
6 };
```

### 2.6.2 constexpr

Allows for compile time execution of code, for example offloading computations to compile time. Support was very sparse at this point and got continually extended.

Listing 15: constexpr (initial form)

```
1 constexpr float k_pi = 3.14f;
2
3 constexpr int square(int x) {
4     return x * x;
5 }
```

### 2.6.3 Lambdas

Listing 16: C++11 Lambda

```
1 auto f = [](int x) {
2     return x * x;
3 };
```

### 2.6.4 std::unordered\_map

### 2.6.5 std::lock\_guard

Listing 17: Lock Guard

```
1 Mutex x;
2 {
3     std::lock_guard<Mutex> guard{x};
4 }
```

# 3 C++ Concepts

## 3.1 Value Semantics

Every Expression belongs to one of the following value categories:

- L value
- PR Value (Pure R value)
- X Value (eXpiring value)

When it is an L value or a X value we call it a GL (General L) value, if it is a PR value or a X value we call it a R value. In particular X values are both GL and R values. The naming is a bit unfortunate, it should PL value and L value; or GR value and R value.

Names are inspired by the fact that L values sit on the "left side of assignments" and R values sit on the "right side of assignments" and are temporaries in that sense. More concretely a L value is something that has a set memory location (`&x` or more accurately `std::address_of(x)`) which can be accessed. An R value is a temporary that does not have an addressable memory location (due to temporary materialisation this is no longer true, but an analogy to think about). X values are GL values which represent expiring objects, so they currently have a fixed memory handle but that one has a "soon" ending lifetime.

Listing 18: Rule of 5

```
1 int x = 5;
2 x; // This id-expression is an L value
3 (x); // L value
4 &x; // prvalue of type int*
5
6 1 + 1; // prvalue (of type int)
7
8 struct Foo {/*...*/}
9 Foo foo{}; // declaration not an expression, so no value category
10
11 foo; // lvalue
12 f(foo); // passes an L value
13 f(std::move(foo)); // X value
14 f(Foo{}); // passes an PR value
```

### 3.1.1 Perfect Forwarding

Under the hood `std::forward` is implemented as a cast:

```
1 template <class T>
2 constexpr T&& forward(std::remove_reference_t<T>& t) noexcept
3 {
4     return static_cast<T&&>(t);
5 }
6
7 template <class T>
8 constexpr T&& forward(std::remove_reference_t<T>&& t) noexcept
9 {
10     static_assert(!std::is_lvalue_reference_v<T>,
11                  "bad forward: cannot forward an rvalue as an lvalue");
12     return static_cast<T&&>(t);
13 }
```

```
1 class Foo{ /*...*/ };
2
3 void kind(T) { println("1"); }
4 void kind(const T&) { println("2"); }
5 void kind(T&&) { println("3"); }
6
7 template <typename T>
8 void bad_forward(T &&arg)
```

```

9   kind(arg);
10 }
11
12 template <typename T>
13 void good_forward(T &&arg)
14 {
15     kind(std::forward<T>(arg));
16 }
17
18 int main()
19 {
20     Foo f{};
21     bad_forward(f);           // prints: kind(Foo&)
22     bad_forward(Foo{});      // prints: kind(Foo&)
23     good_forward(f);         // prints: kind(Foo&)
24     good_forward(Foo{});     // prints: kind(Foo&&)
25 }
26

```

## 3.2 Resource Ownership

### 3.2.1 Rule of Five

If you implement any of the following then you should implement all of them. The reasoning behind that is that you implementing a non-standard constructor or destructor implies that you have some non-trivial resource you don't trust the compiler to manage properly (e.g. heap allocations, database connection, file handle). This used to be the Rule of Three but now we have move and copy assignment constructors since C++11.

Listing 19: Rule of 5

```

1 class Foo {
2     Foo() {}
3     ~Foo() {...} // Destructor
4     Foo(const Foo&) {...} // Copy Constructor
5     Foo(Foo&&) {...} // Move Constructor
6     Foo& operator=(const Foo&) {...} // Copy assignment constructor
7     Foo& operator=(Foo&&) {...} // Move assignment constructor
8 }

```

### 3.2.2 Rule of Zero

Given that implementing the Rule of Five is annoying and creates a lot of boilerplate one should avoid that whenever possible, that is the Rule of Zero.

### 3.2.3 RAII (Resource Acquisition is Initialisation)

In my opinion this is a terrible name and should instead be CADR (Constructor Acquires Destructor Releases). It is a type of scope based automatic resource cleanup, the main idea is that when you invoke the constructor you obtain some resource which you hold for the entire lifetime of that class and once that lifetime has passed (on leaving scope) the resource gets automatically released. This is for example how one can use `std::vector` to avoid having to manually call `new` and `delete`.

To employ this we define a class with a constructor which allocates memory on the heap and a destructor which frees this memory.

```

1 class RAII {
2     RAII(std::string_view name) : name_(name), data_(static_cast<int*>(std::malloc(8 *
3         sizeof(int)))) {
4         std::println("Allocated Memory '{}'", name_);
5     }
6     ~RAII() {
7         std::free(data_);
8         std::println("Deallocated Memory '{}'", name_);
9     }
private:

```

```

10     std::string name_{};
11     int* data_{};
12 };

```

we then can't leak memory anymore, even if exceptions get thrown:

```

1 auto func() -> void {
2     std::println("Starting Function");
3     RAIIScope("Function Scope");
4     std::println("Starting inner scope");
5     {
6         RAIIScope("Inner Scope");
7     }
8     std::println("Finished inner scope");
9     std::println("Throwing exception");
10    throw std::runtime_error("");
11
12    std::println("Finished function");
13 }
14
15 int main() {
16     std::println("Starting Program");
17     try {
18         func();
19     } catch (...) {
20         std::println("Caught exception");
21     }
22     std::println("Finishing Program");
23 }
24 /*
25 Starting Function
26 Allocated Memory 'Function Scope'
27 Starting inner scope
28 Allocated Memory 'Inner Scope'
29 Deallocated Memory 'Inner Scope'
30 Finished inner scope
31 Throwing exception
32 Deallocated Memory 'Function Scope'
33 Caught exception
34 Finishing Program
35 */

```

`scoped_lock` and (the now outdated `lock_guard`) are examples of RAII wrappers for mutexes, `std::vector` is a RAII wrapper around dynamic memory.

### 3.3 NRVO (Named Return Value Optimisation)

To show that NRVO happens we construct a simple tracker class which prints out whenever a copy / move / normal construction happens.

```

1 using Data = std::array<int, 32>;
2 class TrackMemory {
3 public:
4     TrackMemory() {
5         std::println("Empty Constructor");
6     }
7     TrackMemory(const TrackMemory &other) : data_(other.data_) {
8         std::println("Copy constructor");
9     }
10    TrackMemory(TrackMemory &&other) noexcept : data_(std::move(other.data_)) {
11        std::println("Move constructor");
12    }
13 private:
14     Data data_{};
15 };

```

Now we want to show three different cases, one where NRVO can't happen (different named variables of the same type get returned)

```
1 auto no_nrvo(bool return_a) -> TrackMemory {
2     TrackMemory a{}, b{};
3     if (return_a) { return a; }
4     return b;
5 }
```

one where we, according to the standard, might have NRVO (returning a single named variable)

```
1 auto possible_nrvo() -> TrackMemory {
2     TrackMemory a{};
3     return a;
4 }
```

and one case where we are forced to have RVO (case where we return an unnamed temporary)

```
1 auto forced_nrvo() -> TrackMemory {
2     return TrackMemory{};
3 }
```

We then run those

```
1 int main() {
2     std::println("Impossible NRVO:");
3     auto tm = no_nrvo();
4     // Empty Constructor
5     // Empty Constructor
6     // Move constructor
7     std::println("\nPossible NRVO:");
8     auto tm2 = possible_nrvo();
9     // Empty Constructor
10    std::println("\nForced NRVO:");
11    auto tm3 = forced_nrvo();
12    // Empty Constructor
13 }
```

### 3.4 EBO (Empty Base Optimisation)

```
1 struct alignas(16) EmptyAligned {};
2
3 class NoEBO {
4 private:
5     double data_{};
6     EmptyAligned empty_internals_{};
7 };
8
9 class WithEBO {
10 private:
11     double data_{};
12     [[no_unique_address]] EmptyAligned empty_internals_{};
13 };
14
15 static_assert(std::is_empty_v<EmptyAligned>);
16 static_assert(sizeof(EmptyAligned) == 16);
17 static_assert(alignof(EmptyAligned) == 16);
18 static_assert(sizeof(NoEBO) == 32);
19 static_assert(sizeof(WithEBO) == 16);
```

## 3.5 Idioms and Patterns

### 3.5.1 Two-pointer merge

[https://en.wikipedia.org/wiki/Merge\\_algorithm](https://en.wikipedia.org/wiki/Merge_algorithm)

If you want to merge two streams or lists together with some inequality relation on the values you do this:

```
1 std::vector<int> a{ /*...*/ };
2 std::vector<int> b{ /*...*/ };
3 std::vector<int> c{ };
4 c.reserve(a.size() + b.size());
5
6 auto it_a = a.begin();
7 auto it_b = b.begin()
8 while(it_a != a.end() && it_b != b.end()) {
9     if(*it_a <= *it_b) {
10         c.push_back(*it_a);
11         ++it_a;
12     } else {
13         c.push_back(*it_b);
14         ++it_b;
15     }
16 }
17 while(it_a != a.end()) {
18     c.push_back(*it_a);
19     ++it_a;
20 }
21 while(it_b != b.end()) {
22     c.push_back(*it_b);
23     ++it_b;
24 }
```

### 3.5.2 Lazy Delete Idiom / Tombstoning

When you want to delete an object in the middle (or beginning) of a vector you first have to push that element to the end and shift everything to the left by 1 and then `pop_back`, this can be very expensive, especially if many objects have to be deleted.

A common trick to avoid this is to keep a list of tombstones which are boolean values denoting if that value is deleted, and when you access or iterate you simply skip those elements.

If you occassionally pop elements off the back it can be advantageous to just keep popping as long as there are tombstone objects on the top, or if you have downtime in your program you could do cleanup (batching the removal is much faster) or just keep the tombstoned values if the memory cost is not too high.

```
1 struct DeleteableInt{
2     int value{};
3     bool is_deleted{false};
4 };
5
6 std::vector<DeleteableInt> vec{};
7 vec.resize(100);
8 for(auto i = 0; i < vec.size(); ++i) {
9     if(i & 1 == 0) {
10         vec[i].is_deleted = true;
11     }
12 }
```

### 3.5.3 Scope Guard / Defer Pattern

RAII wrappers can also be useful when trying to do some post-scope cleanup or operation, as a type of soft ‘texttdefer’. For example I use

```
1 using Clock = std::chrono::steady_clock;
2 using TimePoint = Clock::time_point;
3 using Duration = std::chrono::duration<f64>;
```

```

4 ScopeTimer::ScopeTimer(std::string_view label) noexcept : label_(label), start_(Clock::now()) {}
5 ScopeTimer::~ScopeTimer() noexcept
6 {
7     const auto dt = Clock::now() - start_;
8     const auto seconds = std::chrono::duration<f64>(dt).count();
9     std::println("{}: {:.3f} ms", label_, seconds * 1000.0);
10 }
11

```

to quickly time scopes (for example actual scoped or function invocations).

### 3.5.4 Unsigned reverse iteration idiom ( $i-- > 0$ )

When iterating backwards with an unsigned index (e.g. `std::size_t`),  $i \geq 0$  is always true, and  $i--$  can underflow. The idiom

```
for (auto i = n; i-- > 0;)
```

means: compare the current value to 0, then decrement, but the loop body sees the decremented value, running for indices  $n-1, \dots, 0$ .

Listing 20: Reverse loop for unsigned indices (safe)

```

1 for (std::size_t i = v.size(); i-- > 0; )
2 {
3     use(v[i]);
4 }

```

The range (C++20) based alternative to this is to use

```
\texttt{for (auto& x : std::views::reverse(v)) {...}}
```

## 3.6 ODR (One Definition Rule)

Use the `extern` keyword if you want to declare a variable but not define it to avoid ODR. Since C++17 you can (and should) use inline for variables instead.

A very subtle ODR bug can occur when you compile some files with debug symbols set and some without.

## 3.7 SIOF (Static Initialisation Order Fiasco)

When you have two non-local objects (e.g. globals or function-local statics) which have dynamic initialisation (run code in their constructors) which live in different translation units (.cpp files) then the initialisation order between them is not specified (standard only guarantees it within one translation unit (top-to-bottom)).

Listing 21: a.cpp

```

1 extern std::string g_name;
2
3 std::string make_greeting() { return "Hello " + g_name; }
4
5 std::string g_greeting = make_greeting();

```

Listing 22: b.cpp

```
1 std::string g_name = "Daniel";
```

Because `g_greeting` depends on `g_name` it might not have been initialised when you define it, so you'd run into UB. Note that if `a.cpp` would start with `std::string g_name = "Steve"` it would be an ODR (one definition rule) violation not SIOF.

## 3.8 Important STL algorithms

### 3.9 Templates

#### 3.9.1 Variadic Templates

#### 3.9.2 Template Metaprogramming

## 4 STL (Standard Templating Library)

### 4.1 Iterators

#### 4.1.1 Iterator Categories

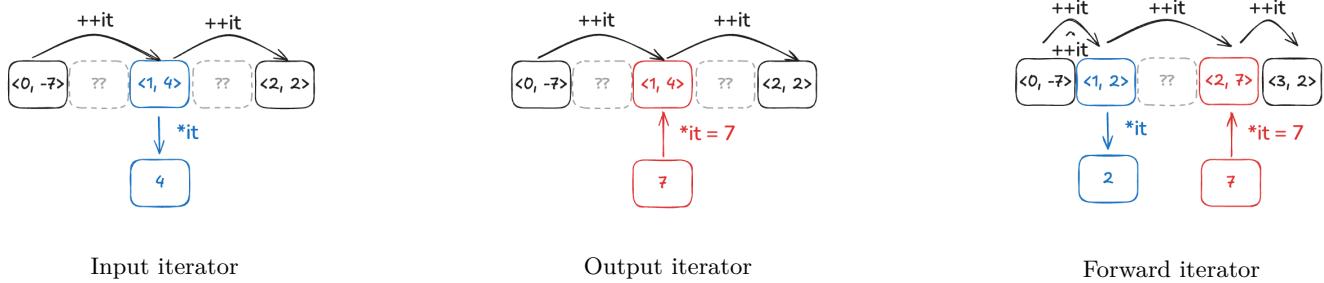


Figure 1: Single-pass and forward iterators

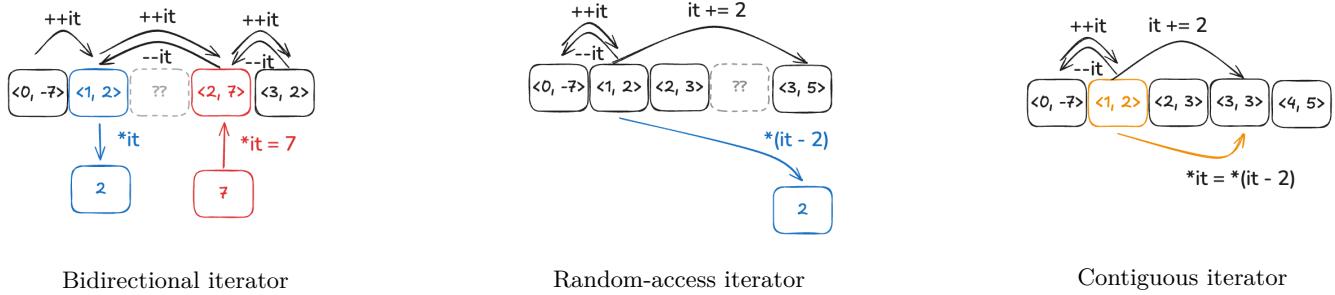


Figure 2: Bidirectional and stronger iterator categories

### 4.2 Ranges

First introduced in C++20, based on Eric Niebler's `Range-v3` library.

#### 4.2.1 Core Concepts

#### 4.2.2 Views

`std::ranges::filter`, `std::ranges::take`, `std::ranges::reverse`

For example the Unsigned reverse iteration idiom ( $i-- > 0$ ) can be implemented using ranges instead:

```
1 \texttt{for (auto& x : std::views::reverse(v)) \{ ... \}}
```

#### 4.2.3 Range Algorithms

`std::ranges::sort`, `std::ranges::find`, `std::ranges::copy`

#### 4.2.4 Range Concepts

### 4.3 Containers

### 4.4 Algorithms

# 5 Data Structures

## 5.1 std::vector (Trivially Copyable Types)

A `vector<T>` is a heap allocated sequence container with three pointers `start_`, `end_`, `capacity_`. The first denotes the start of the vector, the second the location AFTER(!) the last allocated elements and the third is the total memory we have allocated. They take up 24 bytes (`3 * sizeof(T*)`) of stack memory and `(capacity_ - start_) * sizeof(T)` bytes of heap memory.

The core operation on a vector is `push_back` which takes an object and either inserts it in the back if there is space (`end_ != capacity_`) or first resizes (doubling capacity on GCC and Clang, increasing it by 1.5 on MSVC) and then inserting in the end.

If you want to delete an element you swap that position with `end_` and reduce `end_` by one.

This is a famously badly named datastructure as it squats on the name of mathematical elements of vector spaces and physics vectors. A better name would for example be `DynamicArray`.

Listing 23: Vector class

```
1 template <typename T>
2 class VectorTrivial
3 {
4     static_assert(std::is_trivially_copyable_v<T>);
5
6 public:
7     using size_type = std::size_t;
8     /*Rule of 5*/
9     /*Public API*/
10 private:
11     T* start_{};
12     T* end_{};
13     T* cap_{};
14     /*Private API*/
15 };
```

### 5.1.1 Rule of 5

Listing 24: Rule of 5

```
1 VectorTrivial() = default;
2 explicit VectorTrivial(size_type n)
3 { // Allocate with fixed capacity
4     start_ = static_cast<T*>(std::malloc(n_capacity * sizeof(T)));
5     if(!start_) {/*...*/}
6     end_ = start_;
7     cap_ = start_ + n_capacity;
8 }
9
10 VectorTrivial(const Vector& other)
11 { // Copy Constructor
12     const auto n_elems = other.size();
13     const auto n_capacity = other.capacity();
14
15     start_ = static_cast<T*>(std::malloc(n_capacity * sizeof(T)));
16     if(!start_) {/*...*/}
17     std::memcpy(start_, other.start_, n_elems * sizeof(T));
18     end_ = start_ + n_elems;
19     cap_ = start_ + n_capacity;
20 }
21
22 VectorTrivial(Vector&& other) noexcept
23     : start_(other.start_), end_(other.end_), cap_(other.cap_)
24 { // Move Constructor
25     other.start_ = nullptr;
26     other.end_ = nullptr;
```

```

27     other.cap_ = nullptr;
28 }
29
30 VectorTrivial& operator=(const Vector& other)
31 { // Copy Assignment Constructor
32     if(this == &other)
33     {
34         return *this;
35     }
36
37     const auto n_elems = other.size();
38     const auto n_capacity = other.capacity();o
39
40     auto new_start = static_cast<T*>(std::malloc(n_capacity * sizeof(T)));
41     if(!new_start) { /* ... */ }
42
43     std::memcpy(new_start, other.start_, n_elems * sizeof(T));
44
45     std::free(start_);
46     start_ = new_start;
47     end_ = start_ + n_elems;
48     cap_ = start_ + n_capacity;
49     return *this;
50 }
51
52 VectorTrivial& operator=(Vector&& other) noexcept
53 { // Move Assignment Constructor
54     if(this == &other)
55     {
56         return *this;
57     }
58
59     std::free(start_);
60     start_ = other.start_;
61     end_ = other.end_;
62     cap_ = other.cap_
63
64     other.start_ = nullptr;
65     other.end_ = nullptr;
66     other.cap_ = nullptr;
67     return *this;
68 }
69 ~VectorTrivial()
70 { // Destructor
71     std::free(start_);
72 }
```

### 5.1.2 Public API

```

1 [[nodiscard]] auto size() const noexcept -> size_type
2 {
3     return static_cast<size_type>(end_ - start_);
4 }
5 [[nodiscard]] auto capacity() const noexcept -> size_type
6 {
7     return static_cast<size_type>(cap_ - start_);
8 }
9 [[nodiscard]] auto empty() const noexcept -> bool
10 { // More efficient than checking size() == 0
11     return end_ == start_;
12 }
13 void push_back(const T& v)
14 { // push copy of element to the back, growing if necessary }
```

```

15     if (end_ == cap_) { grow_(); }
16     *end_ = v;
17     ++end_;
18 }
19 void push_back(T&& v)
20 { // take ownership of element and push it to the back, growing if necessary
21     if (end_ == cap_) { grow_(); }
22     *end_ = std::move(v);
23     ++end_;
24 }
```

### 5.1.3 Private API

```

1 // 2.0 on MSVC, 1.5 on GCC and Clang
2 constexpr double k_resize_factor = 2.0;
3 constexpr double k_initial_capacity = 1;
4 {
5     const auto n_elems = size();
6     const auto old_cap = capacity();
7     const auto new_cap = old_cap * k_resize_factor;
8     const auto new_cap = (old_cap == 0) ? 8 : new_cap;
9
10    void* new_mem = std::realloc(start_, new_cap * sizeof(T));
11    if (!new_mem) { /* handle allocation failure */ }
12
13    start_ = static_cast<T*>(new_mem);
14    end_ = start_ + n_elems;
15    cap_ = start_ + new_cap;
16 }
```

## 5.2 std::vector (General)

### 5.3 std::unique\_ptr

A unique pointer is a type of smart pointer which semantically has unique ownership over a pointer. It allows for arbitrary types and supports custom deleters (and therefore different types of resources, not only heap memory).

There is a free function which can create a unique pointer with its content inplace called `make_unique` which uses perfect forwarding to push arguments into the constructor of your underlying type using a variadic template.

Because it (by definition) managed non-trivial resources it must implement the Rule of 5. The destructor should automatically invoke the deleter in its destructor.

It must expose a way to access the underlying data, and it must be able to release / relinquish its control over the data it holds.

```

1 template <class T, class Deleter = DefaultDeleter<T>>
2 class UniquePtr {
3 public:
4     using pointer_type = T*;
5     /* Rule of 5 */
6 private:
7     // Trick to make it zero cost abstraction
8     [[no_unique_address]] Deleter deleter_{};
9     pointer_type ptr_{};
10 }
```

#### 5.3.1 Rule of 5

A unique pointer is responsible for cleaning up the underlying resource (as it is a RAII wrapper), we should be able to move one unique pointer into another to move ownership, but copying a unique pointer is meaningless so copy constructor and copy assignment constructor both get deleted.

```

1 // Take ownership over a pointer
2 UniquePtr(pointer_type ptr) { ptr_ = ptr; }
3
4 // Copying is disallowed
5 UniquePtr(const UniquePtr&) = delete;
6 UniquePtr& operator=(const UniquePtr&) = delete;
7
8 // Moving is allowed
9 UniquePtr(UniquePtr&& other) noexcept(/*Deleter must be noexcept moveable and move constructible*/)
10 : ptr_(other.release()), deleter_(std::move(other.deleter_)) {}
11
12 UniquePtr& operator=(UniquePtr&& other) {
13     if(*this == other) {
14         return this;
15     }
16     if(ptr_) {
17         deleter_(ptr_);
18     }
19
20     ptr_ = other.release();
21     deleter_ = std::move(other.deleter_);
22     return *this;
23 }
24
25 ~UniquePtr() {
26     if(ptr_) {
27         deleter_(ptr_);
28     }
29 }
```

### 5.3.2 Public API

It should be possible to `get` the underlying data of the pointer and for the pointer to `release` its ownership to the memory.

```

1 auto release() -> pointer_type {
2     auto ptr = ptr_;
3     ptr_ = nullptr;
4     return ptr;
5 }
6
7 auto get() const -> pointer_type {
8     return ptr_;
9 }
```

### 5.3.3 Custom Deleter

For the unique pointer to be able to manage different types of resources it is important to offer an (optional) custom deleter, but also provide a default deleter (equivalent to `std::default_delete`). Of course that has to be templated over the underlying type. There must be two overloads, one for arrays and one for non-arrays.

```

1 template <class T>
2 struct DefaultDelete {
3     auto operator()(T* p) -> void {
4         delete p;
5     }
6 }
7
8 template <class T>
9 struct DefaultDelete<T[]> {
10     auto operator()(T* p) -> void {
11         delete[] p;
12     }
13 }
```

### 5.3.4 Make Unique

A free function which constructs the object and unique pointer together to avoid having to directly construct the object and then move it into a pointer. It is a variadic template that forwards arguments to the constructor of the underlying type. There are three different cases to consider, non-array pointers, array pointers with unknown bounds and array pointers with known bounds. We only want to support the first two.

```

1  template <class T, class...Args>
2    requires (!std::is_array(T))
3  auto make_unique(Args&&...args) -> UniquePtr<T> {
4    return UniquePtr<T>(new T(std::forward<Args>(args)...));
5  }
6
7  template <class T, class...Args>
8    requires (std::is_array(T) && std::extent_v<T> == 0)
9  auto make_unique(Args&&...args) -> UniquePtr<T> {
10   using U = std::remove_extent_t<T>;
11   return UniquePtr<U>(new U[n]());
12 }
13
14 template <class T, class...Args>
15   requires (std::is_array(T) /*&& bounds side > 0*/)
16 auto make_unique(Args&&...) -> UniquePtr<T> = delete;

```

## 5.4 std::optional

This is a container that is used to either store a value or denote that absence of a value. Main idea is that unlike the pointer types it actually own its memory on the stack and we creat objects using placement new inside of a inner aligned memory buffer.

```

1  template <class T>
2  class Optional{
3  public:
4    /*Rule of 5*/
5    /*Public API*/
6  private:
7    bool is_filled_{false};
8    alignas(T) std::byte storage[sizeof(T)];
9    /*Private API*/
10 };

```

### 5.4.1 Public API

```

1  auto release() -> void {
2    if(is_filled_) {
3      *ptr() ~T();
4      is_filled_ = false;
5    }
6  }
7  template <class...Args>
8  auto emplace(Args...args) -> void {
9    release();
10   ::new (static_cast<void*>storage())
11     T(std::forward<Args>(args));
12   is_filled_ = true;
13 }
14 explicit bool() const noexcept {
15   return is_filled_;
16 }
17 [[nodiscard]] auto has_value() const noexcept -> bool {

```

```

18     return is_filled_;
19 }
20 [[nodiscard]] auto value() & -> T& {
21     return **ptr();
22 }
23 [[nodiscard]] auto value() const & -> const T& {
24     return **ptr();
25 }
26 [[nodiscard]] auto value() & -> T&& {
27     return **ptr();
28 }
29 [[nodiscard]] auto value() const & -> const T&& {
30     return **ptr();
31 }

```

#### 5.4.2 Rule of 5

```

1 Optional() = default();
2 Optional(const T& t) {
3     emplace(t);
4 }
5 Optional(T&& t) {
6     emplace(t);
7 }
8 Optional(const Optional& other) {
9     if(other.is_filled_) {
10         emplace(*other.ptr());
11         is_filled_ = true;
12     }
13 }
14 Optional(Optional&& other) {
15     if(other.is_filled_) {
16         emplace(*other.ptr());
17         is_filled_ = true;
18     }
19 }
20 ~Optional() {
21     reset();
22 }
23 Optional& operator=(const Optional& other) {
24     if(other == *this) {
25         return *this;
26     }
27     if(other.is_filled_) {
28         emplace(*other.ptr());
29         is_filled_ = true;
30     }
31     return *this;
32 }
33 Optional& operator=(Optional&& other) {
34     if(other == *this) {
35         return *this;
36     }
37     if(other.is_filled_) {
38         emplace(*other.ptr());
39         is_filled_ = true;
40     }
41     return *this;
42 }

```

#### 5.4.3 Private API

```

1 auto ptr() -> T& {
2     return std::launder(reinterpret_cast<T>(storage_));
3 }
4 auto ptr() const -> const T& {
5     return std::launder(reinterpret_cast<const T>(storage_));
6 }
7 auto storage() -> T* {
8     return storage_;
9 }
10 auto storage() const -> const T* {
11     return storage_;
12 }

```

## 5.5 Tree

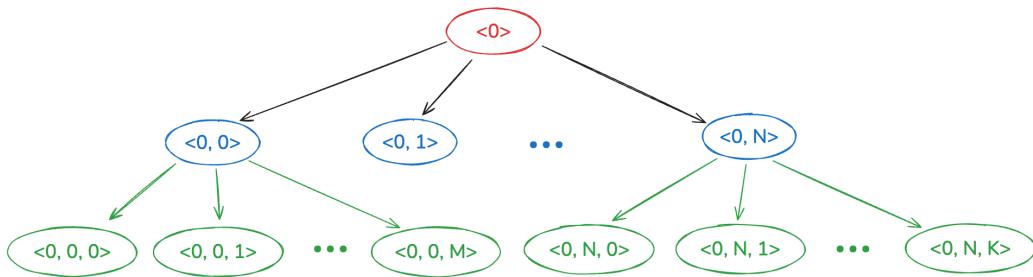


Figure 3: General tree (arbitrary number of children per node)

```

1 struct TreeNode {
2     int value{};
3     std::vector<std::unique_ptr<TreeNode>> children{};
4 };

```

## 5.6 Binary Tree

A special type of Tree where every node has at most 2 children.

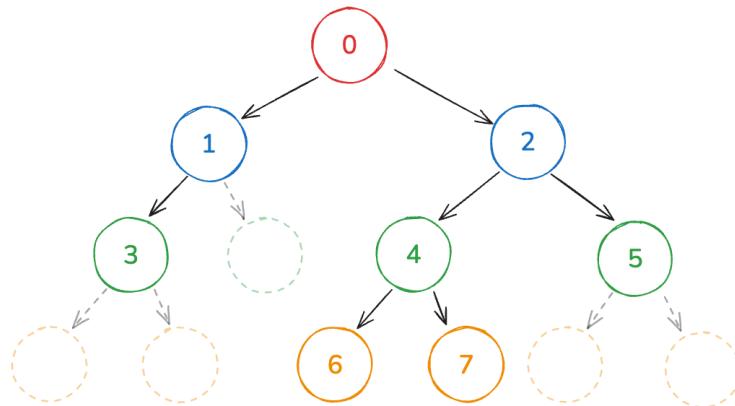


Figure 4: Binary tree (each node has at most two children)

```

1 struct BinaryTreeNode {
2     int value{};
3     std::unique_ptr<std::unique_ptr<TreeNode>> left{};
4     std::unique_ptr<std::unique_ptr<TreeNode>> right{};
5 };

```

## 5.7 Complete Binary Tree

A complete binary tree is one that is "filled up" from left to right, meaning that each level gets filled in first if possible.

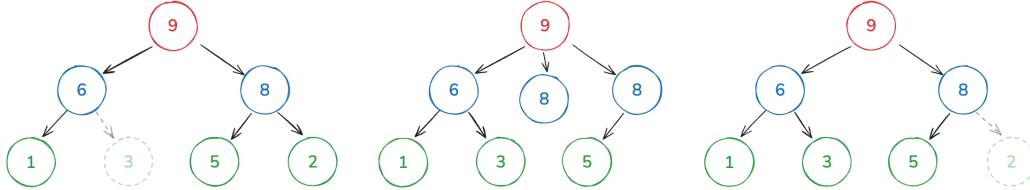


Figure 5: Left is a binary tree that is not complete, middle is not a binary tree, right is a complete binary tree

Complete binary trees can be addressed very efficiently by the rule that the element with offset  $i$  in level  $L$  is indexed by

$$I(L, i) = 2^L - 1 + i.$$

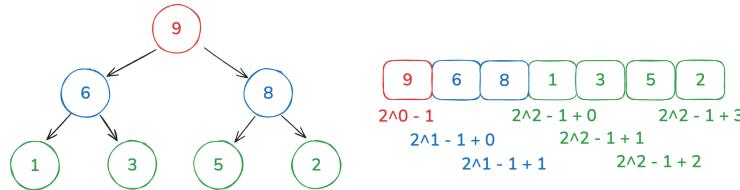


Figure 6: Complete binary tree (filled left-to-right)

## 5.8 std::priority\_queue (Max Heap)

Suppose you want to efficiently pop the largest / smallest entry of a list, this is what max / min heaps are great for. Idea is that you have a linear array representing a complete binary tree in the usual way (root is idx 0, its children are 1, 2, their children are 3, 4 and 5, 6 respectively and so on).

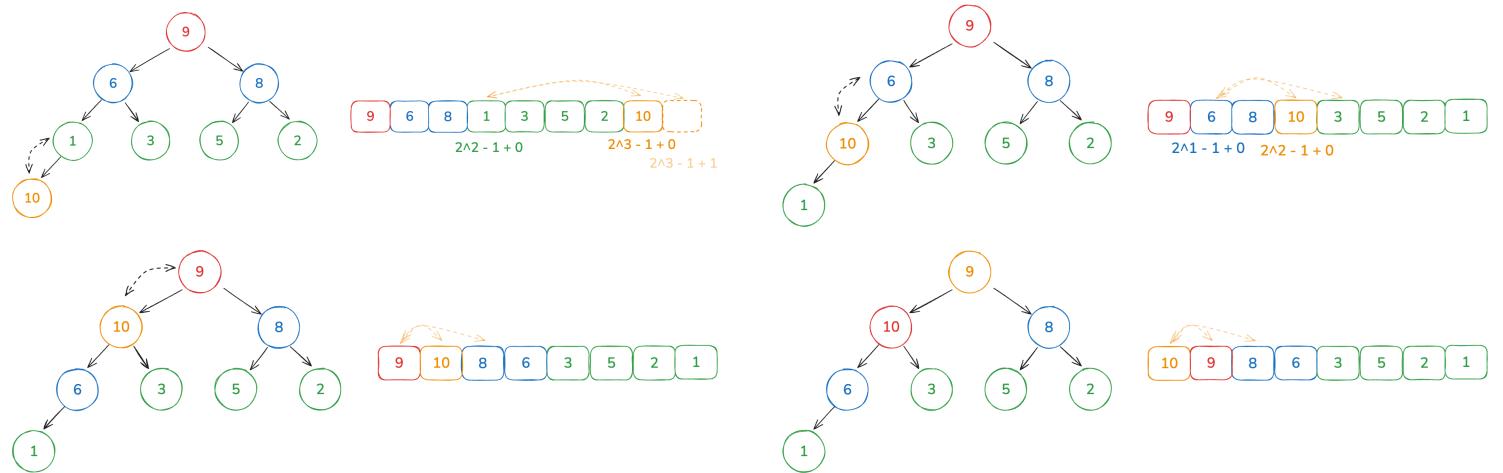


Figure 7: Max-heap insertion (heapify-up steps)

## 5.9 std::unordered\_map

Hashmap, you can use your own hashing function.

## 5.10 Binary Search Tree (BST), Binary Search

A BST is a binary tree which satisfies the property that each node is greater or equal to all nodes in its left subtree and less than all nodes in its right subtree. It allows efficient searching for elements.

## 5.11 Red-Black Tree

A self-balancing BST.

## 5.12 std::map

This stores the keys as a (balanced) binary search tree (more specifically as a red-black tree).

## 5.13 std::array

## 5.14 std::string

## 5.15 std::deque

## 5.16 std::pmr::memory\_resource

## 5.17 std::weak\_ptr, std::shared\_ptr, Control Block

Both `std::weak_ptr` and `std::shared_ptr` internally track objects via the control block which holds strong counter, weak counter, deleter and (optionally) an allocator.

```
1 struct ControlBlock {
2     std::atomic<usize> strong{};
3     std::atomic<usize> weak{};
4     void (*deleter)(void*);
5     void* ptr;
6 }
```

and `std::weak_ptr` holds a pointer to a `ControlBlock`, `std::shared_ptr` holds a pointer to the object and a `ControlBlock`.

## 5.18 std::variant

## 6 Algorithms

### 6.1 DFS (Depth First Search)

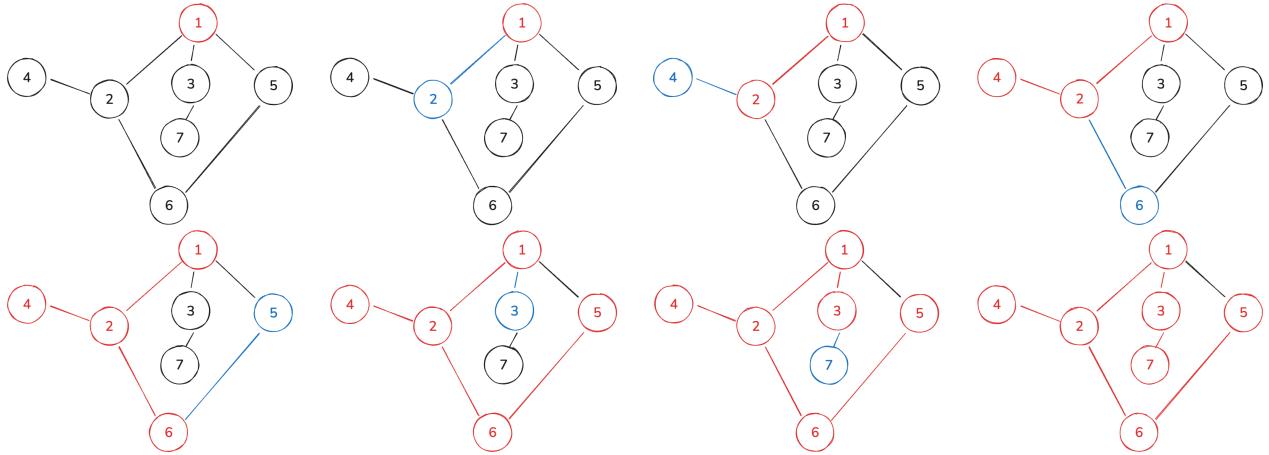


Figure 8: Depth First Search traversal steps (min-neighbor tie rule)

### 6.2 BFS (Breadth First Search)

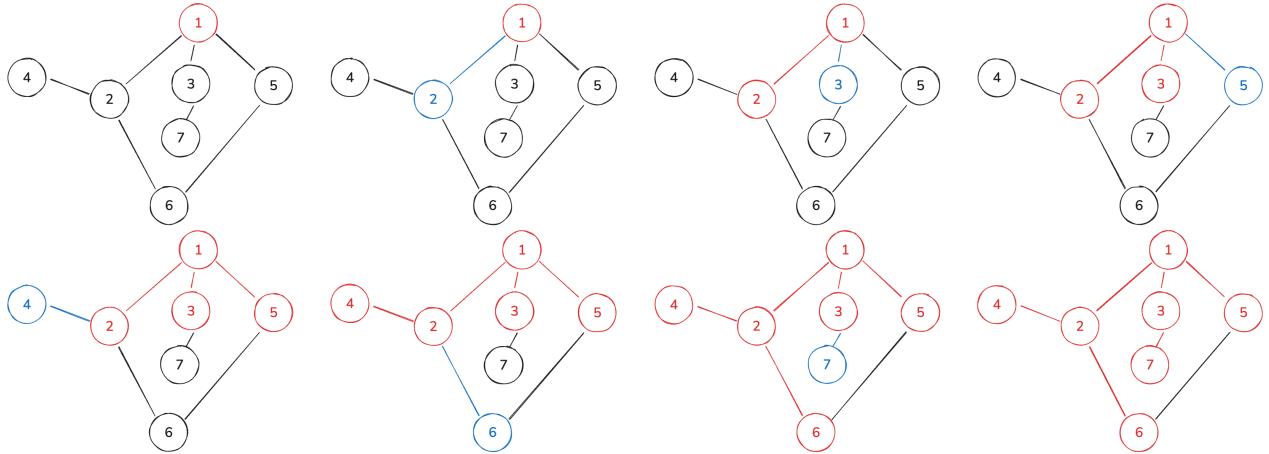


Figure 9: Breadth First Search traversal steps (min-neighbor tie rule)

# 7 Interview Specifics

## 7.1 Tricks

### 7.1.1 2 Pointer Palindrome iteration

To count up all the palindromes in a string we you can sweep center index  $i$  left to right and check for odd palindromes

$$xs[i - a] = xs[i + a]$$

and even palindrome

$$xs[i - a] = xs[i + 1 + a]$$

in the following way:

```
1 const auto n = static_cast<int>(s.size());
2 auto palindrome_count = 0;
3 const auto expand_and_update = [&](int left, int right) -> void {
4     while(left >= 0 && right < n && s[left] == s[right]) {
5         --left;
6         ++right;
7         ++palindrome_count;
8     }
9 };
10 for(auto i = 0; i < n; ++i) {
11     expand_and_update(i, i); // Odd Palindrome
12     expand_and_update(i, i + 1); // Even Palindrome
13 }
```

### 7.1.2 Reverse Heap for bounded count

Suppose that you have an array of  $n$  elements and you want to get the top  $k$  elements of it. One way of doing this is to create a **MaxHeap** and to push all elements on it and pop  $k$  times, but this means you have  $n$  elements in your heap but only need  $k$ .

```
1 auto top_k_naive(const std::vector<int>& xs, int k) -> std::vector<int> {
2     std::priority_queue<int> max_heap{};
3     for(const auto x : xs) {
4         heap.push(x);
5     }
6     std::vector<int> out;
7     out.reserve(static_cast<usize>(k));
8     for(auto i = 0; !heap.empty() && i < k; ++i) {
9         out.push_back(heap.top());
10        heap.pop();
11    }
12    return out;
13 }
```

The way to do this is to instead create a **MinHeap** and to iteratively pop elements as we insert them once we have reached our target:

```
1 auto top_k(const std::vector<int>& xs, int k) -> std::vector<int> {
2     std::priority_queue<int, std::vector<int>, std::greater<int>> min_heap{};
3     for(const auto x : xs) {
4         min_heap.push(x);
5         if(min_heap.size() > k) {
6             // Pops the smallest element
7             min_heap.pop();
8         }
9     }
10    std::vector<int> out;
11    out.reserve(static_cast<usize>(k));
12    for(auto i = 0; !min_heap.empty() && i < k; ++i) {
13        out.push_back(min_heap.top());
14        min_heap.pop();
15    }
16 }
```

```

15     }
16     return out;
17 }
```

**Claim.** Let  $x_1, \dots, x_n$  be the input values and let  $k \geq 1$ . Maintain a min-heap  $H$  with the rule: push each  $x_i$  into  $H$  and, if  $|H| > k$ , pop once. Then after processing all  $n$  elements:

1.  $|H| = \min(k, n)$ ,
2. every element in  $H$  is among the  $\min(k, n)$  largest elements of  $\{x_1, \dots, x_n\}$ ,
3. the element  $\min(H)$  (the heap top) equals the  $k$ -th largest element (when  $n \geq k$ ).

**Proof (invariant argument).** Process the stream left-to-right. For each prefix  $P_i := \{x_1, \dots, x_i\}$  define  $T_i$  to be the multiset of the  $\min(k, i)$  largest elements of  $P_i$ .

We show by induction on  $i$  that after processing  $x_i$ , the heap content equals  $T_i$  as multisets.

**Base case ( $i = 1$ ).** After pushing  $x_1$ , the heap contains  $\{x_1\}$  and no pop occurs. Thus  $H = \{x_1\} = T_1$ .

**Induction step.** Assume after processing  $x_{i-1}$  we have  $H = T_{i-1}$ . Now insert  $x_i$  into  $H$ .

*Case 1:  $i \leq k$ .* No pop occurs, so  $H = T_{i-1} \cup \{x_i\}$ . Since  $i \leq k$ , the set of the  $\min(k, i) = i$  largest elements of  $P_i$  is just all of  $P_i$ , hence  $T_i = P_i = T_{i-1} \cup \{x_i\}$ , so  $H = T_i$ .

*Case 2:  $i > k$ .* After the push we have  $k+1$  elements, then we pop the minimum element  $m$ . So the new heap is

$$H' = (T_{i-1} \cup \{x_i\}) \setminus \{m\}, \quad m = \min(T_{i-1} \cup \{x_i\}).$$

We claim  $H' = T_i$ .

Indeed,  $T_{i-1} \cup \{x_i\}$  consists of  $k+1$  candidates for the top- $k$  of  $P_i$ . Removing the smallest element among these  $k+1$  leaves exactly the  $k$  largest elements of the union. Equivalently, the only element that can be excluded from the top- $k$  of  $P_i$  is the smallest among these  $k+1$  numbers; all remaining elements are  $\geq m$  and therefore dominate  $m$ . Thus the  $k$ -largest multiset of  $P_i$  is precisely

$$T_i = (T_{i-1} \cup \{x_i\}) \setminus \{\min(T_{i-1} \cup \{x_i\})\} = H'.$$

In both cases the invariant holds, so by induction  $H = T_n$  after all insertions. Items (1)–(3) follow immediately: when  $n \geq k$ ,  $H$  contains exactly the  $k$  largest elements, and the minimum of these is the  $k$ -th largest overall.  $\square$

### 7.1.3 Debug Macro

You can use macros to quickly print out the names value of expressions with the following macro. Note that you should always output to `std::cerr` (I think `std::clog`) as `std::cout` is used to evaluate the result.

```

1 #define DS_DBG(x) std::cerr << #x << " = " << (x) << '\n';
```

### 7.1.4 Rolling Average

$$\text{avg}(k) := \frac{1}{k} \sum_{i=1}^k a_i.$$

**Claim.** For all  $k \geq 1$ ,

$$\text{avg}(k+1) = \frac{\text{avg}(k)k + a_{k+1}}{k+1}.$$

**Proof by induction.** **Base case.**

$$\text{avg}(1) = \frac{1}{1} \sum_{i=1}^1 a_i = a_1.$$

**Induction step.** Assume for some  $k \geq 1$  that

$$\text{avg}(k) = \frac{1}{k} \sum_{i=1}^k a_i.$$

Then

$$\text{avg}(k+1) = \frac{1}{k+1} \sum_{i=1}^{k+1} a_i \quad (1)$$

$$= \frac{1}{k+1} \left( \sum_{i=1}^k a_i + a_{k+1} \right) \quad (2)$$

$$= \frac{1}{k+1} \left( k \cdot \frac{1}{k} \sum_{i=1}^k a_i + a_{k+1} \right) \quad (3)$$

$$= \frac{k \text{avg}(k) + a_{k+1}}{k+1}. \quad (4)$$

Thus the identity holds for  $k+1$ , and therefore for all  $k \in \mathbb{N}$  by induction.

### Examples.

$$\text{avg}(1) = a_1,$$

$$\text{avg}(2) = \frac{a_1 + a_2}{2} = \frac{\text{avg}(1) \cdot 1 + a_2}{2},$$

$$\text{avg}(3) = \frac{a_1 + a_2 + a_3}{3} = \frac{\text{avg}(2) \cdot 2 + a_3}{3}.$$

### 7.1.5 Partition-based Binary Search (Two Sorted Arrays Median)

Given two sorted sequences  $A$  and  $B$  with lengths  $n$  and  $m$ , we often want the *lower median* of the multiset union.

Let  $N = n + m$  and define the left-partition size

$$L := \frac{N+1}{2}.$$

We choose counts  $i$  and  $j$  such that

$$i + j = L, \quad 0 \leq i \leq n, \quad 0 \leq j \leq m.$$

Define boundary values (with sentinels):

$$A_L = \begin{cases} -\infty, & i = 0, \\ A[i-1], & \text{else,} \end{cases} \quad A_R = \begin{cases} +\infty, & i = n, \\ A[i], & \text{else,} \end{cases}$$

and analogously  $B_L, B_R$  using  $j$ . The partition is valid iff

$$A_L \leq B_R \quad \text{and} \quad B_L \leq A_R.$$

When valid, the lower median is

$$\max(A_L, B_L).$$

We binary search  $i$  in the feasible range

$$i \in [\max(0, L-m), \min(n, L)],$$

moving left if  $A_L > B_R$  and right otherwise.

### 7.1.6 Prefix Sums

$$S(k) := \sum_{i=1}^k x_i, \quad S(0) := 0.$$

**Claim.** For all integers  $1 \leq i \leq k \leq n$ ,

$$\sum_{j=i}^k x_j = S(k) - S(i-1).$$

**Proof.** By definition,

$$S(k) = \sum_{j=1}^k x_j \quad \text{and} \quad S(i-1) = \sum_{j=1}^{i-1} x_j.$$

Subtracting yields

$$S(k) - S(i-1) = \sum_{j=1}^k x_j - \sum_{j=1}^{i-1} x_j \tag{5}$$

$$= \sum_{j=i}^k x_j. \tag{6}$$

**Examples.**

$$\begin{aligned} \sum_{j=3}^5 x_j &= x_3 + x_4 + x_5 \\ &= (x_1 + x_2 + x_3 + x_4 + x_5) - (x_1 + x_2) \\ &= S(5) - S(2). \end{aligned}$$

Thus, once the cumulative sum array  $S(k)$  is known, any subarray sum can be computed in constant time via

$$\sum_{j=i}^k x_j = S(k) - S(i-1).$$

## 7.2 Problems

### 7.2.1 Rotting Fruit

```

1 int orangesRotting(vector<vector<int>>& grid) {
2     const auto n = static_cast<int>(grid.size());
3     const auto m = static_cast<int>(grid[0].size());
4
5     auto n_fresh = 0;
6     std::queue<Pos> rotten_current_step{};
7     std::queue<Pos> rotten_next_step{};
8     for(auto y = 0; y < n; ++y) {
9         for(auto x = 0; x < m; ++x) {
10            const auto cell = grid[y][x];
11            if(cell == k_fresh) {
12                ++n_fresh;
13            } else if (cell == k_rotten) {
14                rotten_current_step.push({y, x});
15            }
16        }
17    }
18
19    const auto process_neighbor = [&](int y, int x) {
20        if(y < 0 || y >= n || x < 0 || x >= m) return;
21        if(grid[y][x] != k_fresh) return;
22        grid[y][x] = k_rotten;
23        --n_fresh;
24        rotten_next_step.push({y, x});
25    };
26    auto time = 0;
27    while(true) {
28        while(!rotten_current_step.empty()) {
29            const auto [y, x] = rotten_current_step.front();
30            rotten_current_step.pop();

```

```

31     process_neighbor(y, x + 1);
32     process_neighbor(y, x - 1);
33     process_neighbor(y + 1, x);
34     process_neighbor(y - 1, x);
35   }
36   if(rotten_next_step.empty()) {
37     break;
38   }
39   std::swap(rotten_current_step, rotten_next_step);
40   ++time;
41 }
42 if(n_fresh > 0) {
43   return k_fruit_remain_at_end;
44 }
45 return time;
46 }
47 }
```

## 7.3 Clone Graph

Suppose you are given a graph node `Node*`, adjacency is encoded by a per-element `std::vector<Node*>`. We are interested in making a deep copy of that graph.

Core idea is to do a `DFS` (or `BFS`) through the graph to touch all nodes, store the "new-to-old" `Node` relationship in a `std::unordered_map`.

```

1 using OldNode = Node;
2 using NewNode = Node;
3
4 auto graph_deep_copy(OldNode* node) -> Node* {
5   if(!node) return nullptr;
6
7   std::unordered_map<OldNode*, NewNode*> old_to_new{};
8   std::stack<OldNode*> to_visit{};
9
10  old_to_new[node] = new NewNode(node->val);
11  to_visit.push(node);
12  while(!to_visit.empty()) {
13    OldNode* curr = to_visit.top();
14    to_visit.pop();
15
16    NewNode* curr_cpy = old_to_new[curr];
17    for(OldNode* nb : curr->neighbors) {
18      if(!nb) continue;
19      if(const auto it = old_to_new.find(nb); it == old_to_new.end()) {
20        old_to_new[nb] = new NewNode(nb->val);
21        to_visit.push(nb);
22      }
23      curr_cpy->neighbors.push_back(old_to_new[nb]);
24    }
25  }
26  return old_to_new[node];
27 }
```

### 7.3.1 Maximum Subarray

Let `nums` be an array of integers, a subarray is a contiguous non-empty sequence of elements in that array. We are interested in finding the largest subarray.

The solution to this is known Kadane's algorithm, we greedily decide in each step if we should start a new subarray or extend the current one. There is never a reason to backtrack.

```

1 auto best_sum = std::numeric_limits<int>::lowest();
2 auto current_sum = 0;
```

```

3  for(const auto num : nums) {
4      current_sum = std::max(
5          current_sum + num, // Extend current subarray
6          num // Make a new subarray
7      );
8      best_sum = std::max(best_sum, current_sum);
9  }
10 return best_sum;

```

### 7.3.2 Median of Two Circularly Sorted Logs

This uses Partition-Based Binary Search with some pivoted indexing.

Suppose you are given two vectors  $A, B$  which are rotations of sorted lists. This means there exists  $k_A, k_B$  such that

$$[A[k_A], A[k_A + 1], A[k_A + 2], \dots], [B[k_B], B[k_B + 1], A[k_B + 2], \dots]$$

are sorted. The rotation offset (which I call pivot) can then be computed by using binary search

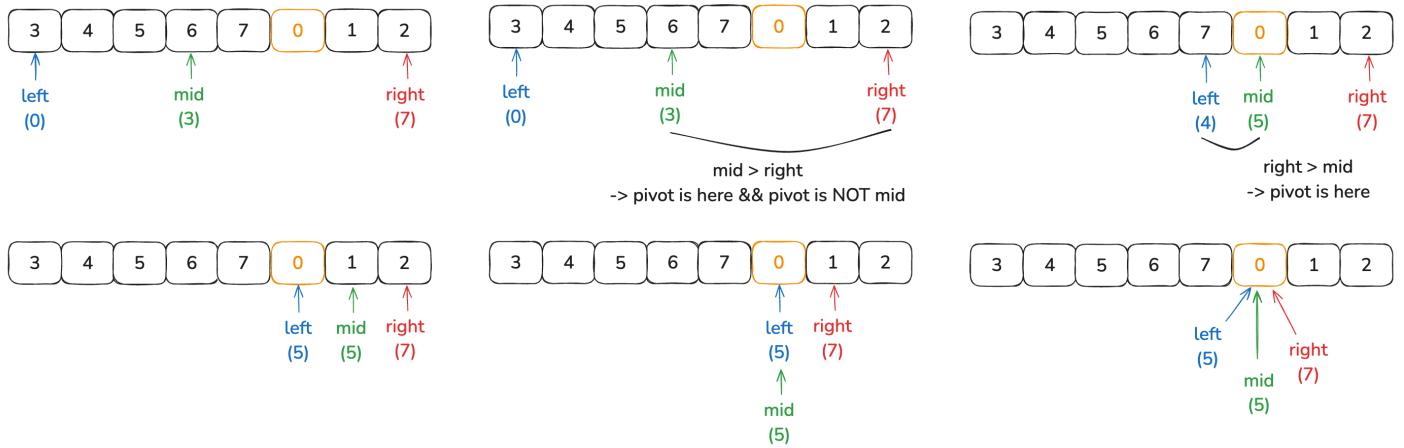


Figure 10: Pivot + partition-binary-search steps (circularly sorted arrays)

```

1  template <typename T>
2  auto compute_pivot(std::span<const T> xs) -> usize {
3      if (xs.empty()) { return 0zu; }
4
5      auto left = 0zu;
6      auto right = xs.size() - 1zu;
7      while (left < right) {
8          const usize mid = std::midpoint(left, right);
9
10         if (xs[mid] < xs[right]) {
11             // Pivot in RHS
12             right = mid;
13         }
14         else if (xs[mid] > xs[right]) {
15             // Pivot in LHS
16             left = mid + 1zu;
17         }
18         else {
19             assert(xs[mid] == xs[right]);
20             --right;
21         }
22     }
23
24     return left;
}

```

With the pivot we can then shift and get an element of the un-pivoted list:

```

1 template<typename T, typename IndexT>
2 auto get_rotated(std::span<const T> xs, IndexT shift, IndexT idx) {
3     return xs[static_cast<usize>(shift + idx) % xs.size()];
4 }
```

As a sidenote, this kind of indexing is expensive for efficient data structures like ring buffers, there we require the size to be a multiple of 2 and we just mask it off directly, but this can't be assumed here.

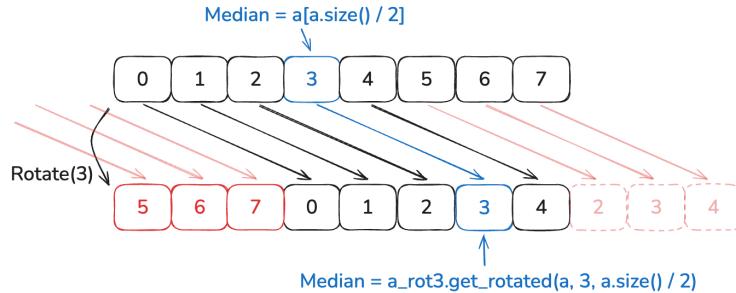


Figure 11: Median of Rotation

If one of the two lists is empty then this problem reduces to finding the pivot of the rotation of a sorted list, which by definition just means getting the middle element in the un-rotated list:

```

1 const auto pivot_A = compute_pivot(A);
2 if(B.empty()) {
3     assert(!A.empty());
4     return get_rotated(A, pivot_A, (A.size() - 1zu) / 2zu);
5 }
```

The general case first computes how many elements we want to have to the left of the median, which is

$$\text{num\_left} = (\text{A.size()} + \text{B.size()} + 1) / 2$$

compute bounds for what ranges in A this could be satisfied with and then doing binary search to find the desired result.

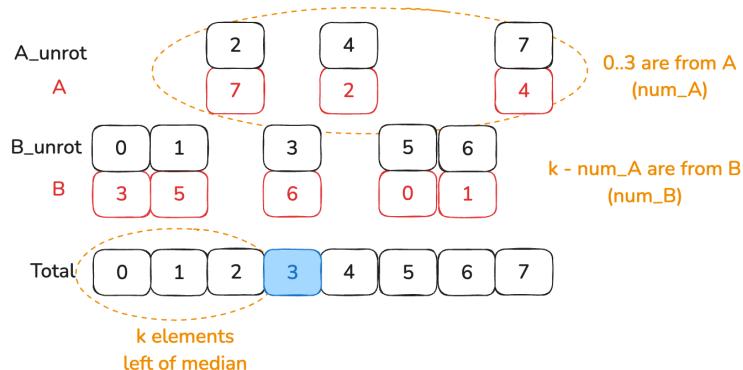


Figure 12: Visualisation of Bounds

We are interested in having  $\text{num\_left} = \text{num\_A} + \text{num\_B}$  elements in total where  $\text{num\_}\{\text{A}, \text{B}\}$  are the number of elements on the left partition. We exit when we have a valid partition, which is categorized by us preserving the ordering (left most element in A is left of right most element of B and v.v.).

```

1 const auto num_A_min = std::max(0, num_left - B.size());
2 const auto num_A_max = A.size();
3 assert(num_A_min <= num_A_max);
4 while(true) {
5     const auto num_A = (num_A_min + num_A_max + 1) / 2;
6     const auto num_B = num_left - num_A;
```

```

7
8  constexpr auto long_min = std::numeric_limits<long>::min();
9  constexpr auto long_max = std::numeric_limits<long>::max();
10
11 const auto A_left = (num_A == 0) ? long_min : get_rotated(A, pivot_A, num_A - 1);
12 const auto A_right = (num_A == A.size()) ? long_max : get_rotated(A, pivot_A, num_A);
13
14 const auto B_left = (num_B == 0) ? long_min : get_rotated(B, pivot_B, num_B - 1);
15 const auto B_right = (num_B == B.size()) ? long_max : get_rotated(B, pivot_B, num_B);
16
17 if(A_left <= B_right && B_left && A_right) { // Partition is Valid
18     return std::max(A_left, B_left);
19 }
20
21 if(A_left > B_right) {
22     num_A_max = num_A - 1;
23 } else {
24     assert(B_left > A_right);
25     num_B_max = num_A + 1;
26 }
27

```

### 7.3.3 Lazy Deletion Stack

This uses the Lazy Delete / Tombstone pattern to allow efficiently deleting large parts of a stack to implement `remove_lower` and `remove_upper`, which cut out (potentially) large parts of the stack. Aside from tombstoning a `std::map` is used (as opposed to `std::unordered_map`) to avoid having to do a linear sweep over the whole stack when deleting objects.

```

1 class LazyDeleteStack {
2 public:
3     auto push(int value) -> void {
4         const usize idx = data_.size();
5         data_.push_back({value, false});
6         value_to_idx_[value].push_back(idx);
7     }
8
9     auto pop() -> void {
10        while (!data_.empty()) {
11            const auto top = data_.back();
12
13            const auto entry = data_.back();
14            data_.pop_back();
15            value_to_idx_[entry.value].pop_back();
16
17            if (!top.is_removed) {
18                // break on first alive object deleted
19                break;
20            }
21        }
22    }
23
24    auto remove_lower(int value) -> void {
25        auto it = value_to_idx_.begin();
26        const auto end = value_to_idx_.lower_bound(value);
27        while (it != end) {
28            for (auto idx : it->second) {
29                data_[idx].is_removed = true;
30            }
31            ++it;
32        }
33    }
34
35    auto remove_upper(int value) -> void {
36        auto it = value_to_idx_.upper_bound(value);
37
38        for (auto idx : it->second) {
39            data_[idx].is_removed = true;
40        }
41    }
42
43    const auto is_empty() const {
44        return data_.empty();
45    }
46
47    const auto size() const {
48        return data_.size();
49    }
50
51    const auto front() const {
52        return data_.front();
53    }
54
55    const auto back() const {
56        return data_.back();
57    }
58
59    const auto begin() const {
60        return data_.begin();
61    }
62
63    const auto end() const {
64        return data_.end();
65    }
66
67    const auto rbegin() const {
68        return data_.rbegin();
69    }
70
71    const auto rend() const {
72        return data_.rend();
73    }
74
75    const auto begin() {
76        return data_.begin();
77    }
78
79    const auto end() {
80        return data_.end();
81    }
82
83    const auto rbegin() {
84        return data_.rbegin();
85    }
86
87    const auto rend() {
88        return data_.rend();
89    }
90
91    const auto front() {
92        return data_.front();
93    }
94
95    const auto back() {
96        return data_.back();
97    }
98
99    const auto size() {
100       return data_.size();
101    }
102
103    const auto is_empty() {
104        return data_.empty();
105    }
106
107    const auto begin() {
108        return data_.begin();
109    }
110
111    const auto end() {
112        return data_.end();
113    }
114
115    const auto rbegin() {
116        return data_.rbegin();
117    }
118
119    const auto rend() {
120        return data_.rend();
121    }
122
123    const auto front() {
124        return data_.front();
125    }
126
127    const auto back() {
128        return data_.back();
129    }
130
131    const auto size() {
132        return data_.size();
133    }
134
135    const auto is_empty() {
136        return data_.empty();
137    }
138
139    const auto begin() {
140        return data_.begin();
141    }
142
143    const auto end() {
144        return data_.end();
145    }
146
147    const auto rbegin() {
148        return data_.rbegin();
149    }
150
151    const auto rend() {
152        return data_.rend();
153    }
154
155    const auto front() {
156        return data_.front();
157    }
158
159    const auto back() {
160        return data_.back();
161    }
162
163    const auto size() {
164        return data_.size();
165    }
166
167    const auto is_empty() {
168        return data_.empty();
169    }
170
171    const auto begin() {
172        return data_.begin();
173    }
174
175    const auto end() {
176        return data_.end();
177    }
178
179    const auto rbegin() {
180        return data_.rbegin();
181    }
182
183    const auto rend() {
184        return data_.rend();
185    }
186
187    const auto front() {
188        return data_.front();
189    }
190
191    const auto back() {
192        return data_.back();
193    }
194
195    const auto size() {
196        return data_.size();
197    }
198
199    const auto is_empty() {
200        return data_.empty();
201    }
202
203    const auto begin() {
204        return data_.begin();
205    }
206
207    const auto end() {
208        return data_.end();
209    }
210
211    const auto rbegin() {
212        return data_.rbegin();
213    }
214
215    const auto rend() {
216        return data_.rend();
217    }
218
219    const auto front() {
220        return data_.front();
221    }
222
223    const auto back() {
224        return data_.back();
225    }
226
227    const auto size() {
228        return data_.size();
229    }
230
231    const auto is_empty() {
232        return data_.empty();
233    }
234
235    const auto begin() {
236        return data_.begin();
237    }
238
239    const auto end() {
240        return data_.end();
241    }
242
243    const auto rbegin() {
244        return data_.rbegin();
245    }
246
247    const auto rend() {
248        return data_.rend();
249    }
250
251    const auto front() {
252        return data_.front();
253    }
254
255    const auto back() {
256        return data_.back();
257    }
258
259    const auto size() {
260        return data_.size();
261    }
262
263    const auto is_empty() {
264        return data_.empty();
265    }
266
267    const auto begin() {
268        return data_.begin();
269    }
270
271    const auto end() {
272        return data_.end();
273    }
274
275    const auto rbegin() {
276        return data_.rbegin();
277    }
278
279    const auto rend() {
280        return data_.rend();
281    }
282
283    const auto front() {
284        return data_.front();
285    }
286
287    const auto back() {
288        return data_.back();
289    }
290
291    const auto size() {
292        return data_.size();
293    }
294
295    const auto is_empty() {
296        return data_.empty();
297    }
298
299    const auto begin() {
300        return data_.begin();
301    }
302
303    const auto end() {
304        return data_.end();
305    }
306
307    const auto rbegin() {
308        return data_.rbegin();
309    }
310
311    const auto rend() {
312        return data_.rend();
313    }
314
315    const auto front() {
316        return data_.front();
317    }
318
319    const auto back() {
320        return data_.back();
321    }
322
323    const auto size() {
324        return data_.size();
325    }
326
327    const auto is_empty() {
328        return data_.empty();
329    }
330
331    const auto begin() {
332        return data_.begin();
333    }
334
335    const auto end() {
336        return data_.end();
337    }
338
339    const auto rbegin() {
340        return data_.rbegin();
341    }
342
343    const auto rend() {
344        return data_.rend();
345    }
346
347    const auto front() {
348        return data_.front();
349    }
350
351    const auto back() {
352        return data_.back();
353    }
354
355    const auto size() {
356        return data_.size();
357    }
358
359    const auto is_empty() {
360        return data_.empty();
361    }
362
363    const auto begin() {
364        return data_.begin();
365    }
366
367    const auto end() {
368        return data_.end();
369    }
370
371    const auto rbegin() {
372        return data_.rbegin();
373    }
374
375    const auto rend() {
376        return data_.rend();
377    }
378
379    const auto front() {
380        return data_.front();
381    }
382
383    const auto back() {
384        return data_.back();
385    }
386
387    const auto size() {
388        return data_.size();
389    }
390
391    const auto is_empty() {
392        return data_.empty();
393    }
394
395    const auto begin() {
396        return data_.begin();
397    }
398
399    const auto end() {
400        return data_.end();
401    }
402
403    const auto rbegin() {
404        return data_.rbegin();
405    }
406
407    const auto rend() {
408        return data_.rend();
409    }
410
411    const auto front() {
412        return data_.front();
413    }
414
415    const auto back() {
416        return data_.back();
417    }
418
419    const auto size() {
420        return data_.size();
421    }
422
423    const auto is_empty() {
424        return data_.empty();
425    }
426
427    const auto begin() {
428        return data_.begin();
429    }
430
431    const auto end() {
432        return data_.end();
433    }
434
435    const auto rbegin() {
436        return data_.rbegin();
437    }
438
439    const auto rend() {
440        return data_.rend();
441    }
442
443    const auto front() {
444        return data_.front();
445    }
446
447    const auto back() {
448        return data_.back();
449    }
450
451    const auto size() {
452        return data_.size();
453    }
454
455    const auto is_empty() {
456        return data_.empty();
457    }
458
459    const auto begin() {
460        return data_.begin();
461    }
462
463    const auto end() {
464        return data_.end();
465    }
466
467    const auto rbegin() {
468        return data_.rbegin();
469    }
470
471    const auto rend() {
472        return data_.rend();
473    }
474
475    const auto front() {
476        return data_.front();
477    }
478
479    const auto back() {
480        return data_.back();
481    }
482
483    const auto size() {
484        return data_.size();
485    }
486
487    const auto is_empty() {
488        return data_.empty();
489    }
490
491    const auto begin() {
492        return data_.begin();
493    }
494
495    const auto end() {
496        return data_.end();
497    }
498
499    const auto rbegin() {
500        return data_.rbegin();
501    }
502
503    const auto rend() {
504        return data_.rend();
505    }
506
507    const auto front() {
508        return data_.front();
509    }
510
511    const auto back() {
512        return data_.back();
513    }
514
515    const auto size() {
516        return data_.size();
517    }
518
519    const auto is_empty() {
520        return data_.empty();
521    }
522
523    const auto begin() {
524        return data_.begin();
525    }
526
527    const auto end() {
528        return data_.end();
529    }
530
531    const auto rbegin() {
532        return data_.rbegin();
533    }
534
535    const auto rend() {
536        return data_.rend();
537    }
538
539    const auto front() {
540        return data_.front();
541    }
542
543    const auto back() {
544        return data_.back();
545    }
546
547    const auto size() {
548        return data_.size();
549    }
550
551    const auto is_empty() {
552        return data_.empty();
553    }
554
555    const auto begin() {
556        return data_.begin();
557    }
558
559    const auto end() {
560        return data_.end();
561    }
562
563    const auto rbegin() {
564        return data_.rbegin();
565    }
566
567    const auto rend() {
568        return data_.rend();
569    }
570
571    const auto front() {
572        return data_.front();
573    }
574
575    const auto back() {
576        return data_.back();
577    }
578
579    const auto size() {
580        return data_.size();
581    }
582
583    const auto is_empty() {
584        return data_.empty();
585    }
586
587    const auto begin() {
588        return data_.begin();
589    }
590
591    const auto end() {
592        return data_.end();
593    }
594
595    const auto rbegin() {
596        return data_.rbegin();
597    }
598
599    const auto rend() {
600        return data_.rend();
601    }
602
603    const auto front() {
604        return data_.front();
605    }
606
607    const auto back() {
608        return data_.back();
609    }
610
611    const auto size() {
612        return data_.size();
613    }
614
615    const auto is_empty() {
616        return data_.empty();
617    }
618
619    const auto begin() {
620        return data_.begin();
621    }
622
623    const auto end() {
624        return data_.end();
625    }
626
627    const auto rbegin() {
628        return data_.rbegin();
629    }
630
631    const auto rend() {
632        return data_.rend();
633    }
634
635    const auto front() {
636        return data_.front();
637    }
638
639    const auto back() {
640        return data_.back();
641    }
642
643    const auto size() {
644        return data_.size();
645    }
646
647    const auto is_empty() {
648        return data_.empty();
649    }
650
651    const auto begin() {
652        return data_.begin();
653    }
654
655    const auto end() {
656        return data_.end();
657    }
658
659    const auto rbegin() {
660        return data_.rbegin();
661    }
662
663    const auto rend() {
664        return data_.rend();
665    }
666
667    const auto front() {
668        return data_.front();
669    }
670
671    const auto back() {
672        return data_.back();
673    }
674
675    const auto size() {
676        return data_.size();
677    }
678
679    const auto is_empty() {
680        return data_.empty();
681    }
682
683    const auto begin() {
684        return data_.begin();
685    }
686
687    const auto end() {
688        return data_.end();
689    }
690
691    const auto rbegin() {
692        return data_.rbegin();
693    }
694
695    const auto rend() {
696        return data_.rend();
697    }
698
699    const auto front() {
700        return data_.front();
701    }
702
703    const auto back() {
704        return data_.back();
705    }
706
707    const auto size() {
708        return data_.size();
709    }
710
711    const auto is_empty() {
712        return data_.empty();
713    }
714
715    const auto begin() {
716        return data_.begin();
717    }
718
719    const auto end() {
720        return data_.end();
721    }
722
723    const auto rbegin() {
724        return data_.rbegin();
725    }
726
727    const auto rend() {
728        return data_.rend();
729    }
730
731    const auto front() {
732        return data_.front();
733    }
734
735    const auto back() {
736        return data_.back();
737    }
738
739    const auto size() {
740        return data_.size();
741    }
742
743    const auto is_empty() {
744        return data_.empty();
745    }
746
747    const auto begin() {
748        return data_.begin();
749    }
750
751    const auto end() {
752        return data_.end();
753    }
754
755    const auto rbegin() {
756        return data_.rbegin();
757    }
758
759    const auto rend() {
760        return data_.rend();
761    }
762
763    const auto front() {
764        return data_.front();
765    }
766
767    const auto back() {
768        return data_.back();
769    }
770
771    const auto size() {
772        return data_.size();
773    }
774
775    const auto is_empty() {
776        return data_.empty();
777    }
778
779    const auto begin() {
780        return data_.begin();
781    }
782
783    const auto end() {
784        return data_.end();
785    }
786
787    const auto rbegin() {
788        return data_.rbegin();
789    }
790
791    const auto rend() {
792        return data_.rend();
793    }
794
795    const auto front() {
796        return data_.front();
797    }
798
799    const auto back() {
800        return data_.back();
801    }
802
803    const auto size() {
804        return data_.size();
805    }
806
807    const auto is_empty() {
808        return data_.empty();
809    }
810
811    const auto begin() {
812        return data_.begin();
813    }
814
815    const auto end() {
816        return data_.end();
817    }
818
819    const auto rbegin() {
820        return data_.rbegin();
821    }
822
823    const auto rend() {
824        return data_.rend();
825    }
826
827    const auto front() {
828        return data_.front();
829    }
830
831    const auto back() {
832        return data_.back();
833    }
834
835    const auto size() {
836        return data_.size();
837    }
838
839    const auto is_empty() {
840        return data_.empty();
841    }
842
843    const auto begin() {
844        return data_.begin();
845    }
846
847    const auto end() {
848        return data_.end();
849    }
850
851    const auto rbegin() {
852        return data_.rbegin();
853    }
854
855    const auto rend() {
856        return data_.rend();
857    }
858
859    const auto front() {
860        return data_.front();
861    }
862
863    const auto back() {
864        return data_.back();
865    }
866
867    const auto size() {
868        return data_.size();
869    }
870
871    const auto is_empty() {
872        return data_.empty();
873    }
874
875    const auto begin() {
876        return data_.begin();
877    }
878
879    const auto end() {
880        return data_.end();
881    }
882
883    const auto rbegin() {
884        return data_.rbegin();
885    }
886
887    const auto rend() {
888        return data_.rend();
889    }
890
891    const auto front() {
892        return data_.front();
893    }
894
895    const auto back() {
896        return data_.back();
897    }
898
899    const auto size() {
900        return data_.size();
901    }
902
903    const auto is_empty() {
904        return data_.empty();
905    }
906
907    const auto begin() {
908        return data_.begin();
909    }
910
911    const auto end() {
912        return data_.end();
913    }
914
915    const auto rbegin() {
916        return data_.rbegin();
917    }
918
919    const auto rend() {
920        return data_.rend();
921    }
922
923    const auto front() {
924        return data_.front();
925    }
926
927    const auto back() {
928        return data_.back();
929    }
930
931    const auto size() {
932        return data_.size();
933    }
934
935    const auto is_empty() {
936        return data_.empty();
937    }
938
939    const auto begin() {
940        return data_.begin();
941    }
942
943    const auto end() {
944        return data_.end();
945    }
946
947    const auto rbegin() {
948        return data_.rbegin();
949    }
950
951    const auto rend() {
952        return data_.rend();
953    }
954
955    const auto front() {
956        return data_.front();
957    }
958
959    const auto back() {
960        return data_.back();
961    }
962
963    const auto size() {
964        return data_.size();
965    }
966
967    const auto is_empty() {
968        return data_.empty();
969    }
970
971    const auto begin() {
972        return data_.begin();
973    }
974
975    const auto end() {
976        return data_.end();
977    }
978
979    const auto rbegin() {
980        return data_.rbegin();
981    }
982
983    const auto rend() {
984        return data_.rend();
985    }
986
987    const auto front() {
988        return data_.front();
989    }
990
991    const auto back() {
992        return data_.back();
993    }
994
995    const auto size() {
996        return data_.size();
997    }
998
999    const auto is_empty() {
1000       return data_.empty();
1001    }
1002
1003    const auto begin() {
1004        return data_.begin();
1005    }
1006
1007    const auto end() {
1008        return data_.end();
1009    }
1010
1011    const auto rbegin() {
1012        return data_.rbegin();
1013    }
1014
1015    const auto rend() {
1016        return data_.rend();
1017    }
1018
1019    const auto front() {
1020        return data_.front();
1021    }
1022
1023    const auto back() {
1024        return data_.back();
1025    }
1026
1027    const auto size() {
1028        return data_.size();
1029    }
1030
1031    const auto is_empty() {
1032        return data_.empty();
1033    }
1034
1035    const auto begin() {
1036        return data_.begin();
1037    }
1038
1039    const auto end() {
1040        return data_.end();
1041    }
1042
1043    const auto rbegin() {
1044        return data_.rbegin();
1045    }
1046
1047    const auto rend() {
1048        return data_.rend();
1049    }
1050
1051    const auto front() {
1052        return data_.front();
1053    }
1054
1055    const auto back() {
1056        return data_.back();
1057    }
1058
1059    const auto size() {
1060        return data_.size();
1061    }
1062
1063    const auto is_empty() {
1064        return data_.empty();
1065    }
1066
1067    const auto begin() {
1068        return data_.begin();
1069    }
1070
1071    const auto end() {
1072        return data_.end();
1073    }
1074
1075    const auto rbegin() {
1076        return data_.rbegin();
1077    }
1078
1079    const auto rend() {
1080        return data_.rend();
1081    }
1082
1083    const auto front() {
1084        return data_.front();
1085    }
1086
1087    const auto back() {
1088        return data_.back();
1089    }
1090
1091    const auto size() {
1092        return data_.size();
1093    }
1094
1095    const auto is_empty() {
1096        return data_.empty();
1097    }
1098
1099    const auto begin() {
1100        return data_.begin();
1101    }
1102
1103    const auto end() {
1104        return data_.end();
1105    }
1106
1107    const auto rbegin() {
1108        return data_.rbegin();
1109    }
1110
1111    const auto rend() {
1112        return data_.rend();
1113    }
1114
1115    const auto front() {
1116        return data_.front();
1117    }
1118
1119    const auto back() {
1120        return data_.back();
1121    }
1122
1123    const auto size() {
1124        return data_.size();
1125    }
1126
1127    const auto is_empty() {
1128        return data_.empty();
1129    }
1130
1131    const auto begin() {
1132        return data_.begin();
1133    }
1134
1135    const auto end() {
1136        return data_.end();
1137    }
1138
1139    const auto rbegin() {
1140        return data_.rbegin();
1141    }
1142
1143    const auto rend() {
1144        return data_.rend();
1145    }
1146
1147    const auto front() {
1148        return data_.front();
1149    }
1150
1151    const auto back() {
1152        return data_.back();
1153    }
1154
1155    const auto size() {
1156        return data_.size();
1157    }
1158
1159    const auto is_empty() {
1160        return data_.empty();
1161    }
1162
1163    const auto begin() {
1164        return data_.begin();
1165    }
1166
1167    const auto end() {
1168        return data_.end();
1169    }
1170
1171    const auto rbegin() {
1172        return data_.rbegin();
1173    }
1174
1175    const auto rend() {
1176        return data_.rend();
1177    }
1178
1179    const auto front() {
1180        return data_.front();
1181    }
1182
1183    const auto back() {
1184        return data_.back();
1185    }
1186
1187    const auto size() {
1188        return data_.size();
1189    }
1190
1191    const auto is_empty() {
1192        return data_.empty();
1193    }
1194
1195    const auto begin() {
1196        return data_.begin();
1197    }
1198
1199    const auto end() {
1200        return data_.end();
1201    }
1202
1203    const auto rbegin() {
1204        return data_.rbegin();
1205    }
1206
1207    const auto rend() {
1208        return data_.rend();
1209    }
1210
1211    const auto front() {
1212        return data_.front();
1213    }
1214
1215    const auto back() {
1216        return data_.back();
1217    }
1218
1219    const auto size() {
1220        return data_.size();
1221    }
1222
1223    const auto is_empty() {
1224        return data_.empty();
1225    }
1226
1227    const auto begin() {
1228        return data_.begin();
1229    }
1230
1231    const auto end() {
1232        return data_.end();
1233    }
1234
1235    const auto rbegin() {
1236        return data_.rbegin();
1237    }
1238
1239    const auto rend() {
1240        return data_.rend();
1241    }
1242
1243    const auto front() {
1244        return data_.front();
1245    }
1246
1247    const auto back() {
1248        return data_.back();
1249    }
1250
1251    const auto size() {
1252        return data_.size();
1253    }
1254
1255    const auto is_empty() {
1256        return data_.empty();
1257    }
1258
1259    const auto begin() {
1260        return data_.begin();
1261    }
1262
1263    const auto end() {
1264        return data_.end();
1265    }
1266
1267    const auto rbegin() {
1268        return data_.rbegin();
1269    }
1270
1271    const auto rend() {
1272        return data_.rend();
1273    }
1274
1275    const auto front() {
1276        return data_.front();
1277    }
1278
1279    const auto back() {
1280        return data_.back();
1281    }
1282
1283    const auto size() {
1284        return data_.size();
1285    }
1286
1287    const auto is_empty() {
1288        return data_.empty();
1289    }
1290
1291    const auto begin() {
1292        return data_.begin();
1293    }
1294
1295    const auto end() {
1296        return data_.end();
1297    }
1298
1299    const auto rbegin() {
1300        return data_.rbegin();
1301    }
1302
1303    const auto rend() {
1304        return data_.rend();
1305    }
1306
1307    const auto front() {
1308        return data_.front();
1309    }
1310
1311    const auto back() {
1312        return data_.back();
1313    }
1314
1315    const auto size() {
1316        return data_.size();
1317    }
1318
1319    const auto is_empty() {
1320        return data_.empty();
1321    }
1322
1323    const auto begin() {
1324        return data_.begin();
1325    }
1326
1327    const auto end() {
1328        return data_.end();
1329    }
1330
1331    const auto rbegin() {
1332        return data_.rbegin();
1333    }
1334
1335    const auto rend() {
1336        return data_.rend();
1337    }
1338
1339    const auto front() {
1340        return data_.front();
1341    }
1342
1343    const auto back() {
1344        return data_.back();
1345    }
1346
1347    const auto size() {
1348        return data_.size();
1349    }
1350
1351    const auto is_empty() {
1352        return data_.empty();
1353    }
1354
1355    const auto begin() {
1356        return data_.begin();
1357    }
1358
1359    const auto end() {
1360        return data_.end();
1361    }
1362
1363    const auto rbegin() {
1364        return data_.rbegin();
1365    }
1366
1367    const auto rend() {
1368        return data_.rend();
1369    }
1370
1371    const auto front() {
1372        return data_.front();
1373    }
1374
1375    const auto back() {
1376        return data_.back();
1377    }
1378
1379    const auto size() {
1380        return data_.size();
1381    }
1382
1383    const auto is_empty() {
1384        return data_.empty();
1385    }
1386
1387    const auto begin() {
1388        return data_.begin();
1389    }
1390
1391    const auto end() {
1392        return data_.end();
1393    }
1394
1395    const auto rbegin() {
1396        return data_.rbegin();
1397    }
1398
1399    const auto rend() {
1400        return data_.rend();
1401    }
1402
1403    const auto front() {
1404        return data_.front();
1405    }
1406
1407    const auto back() {
1408        return data_.back();
1409    }
1410
1411    const auto size() {
1412        return data_.size();
1413    }
1414
1415    const auto is_empty() {
1416        return data_.empty();
1417    }
1418
1419    const auto begin() {
1420        return data_.begin();
1421    }
1422
1423    const auto end() {
1424        return data_.end();
1425    }
1426
1427    const auto rbegin() {
1428        return data_.rbegin();
1429    }
1430
1431    const auto rend() {
1432        return data_.rend();
1433    }
1434
1435    const auto front() {
1436        return data_.front();
1437    }
1438
1439    const auto back() {
1440        return data_.back();
1441    }
1442
1443    const auto size() {
1444        return data_.size();
1445    }
1446
1447    const auto is_empty() {
1448        return data_.empty();
1449    }
1450
1451    const auto begin() {
1452        return data_.begin();
1453    }
1454
1455    const auto end() {
1456        return data_.end();
1457    }
1458
1459    const auto rbegin() {
1460        return data_.rbegin();
1461    }
1462
1463    const auto rend() {
1464        return data_.rend();
1465    }
1466
1467    const auto front() {
1468        return data_.front();
1469    }
1470
1471    const auto back() {
1472        return data_.back();
1473    }
1474
1475    const auto size() {
1476        return data_.size();
1477    }
1478
1479    const auto is_empty() {
1480        return data_.empty();
1481    }
1482
1483    const auto begin() {
1484        return data_.begin();
1485    }
1486
1487    const auto end() {
1488        return data_.end();
1489    }
1490
1491    const auto rbegin() {
1492        return data_.rbegin();
1493    }
1494
1495    const auto rend() {
1496        return data_.rend();
1497    }
1498
1499    const auto front() {
1500        return data_.front();
1501    }
1502
1503    const auto back() {
1504        return data_.back();
1505    }
1506
1507    const auto size() {
1508        return data_.size();
1509    }
1510
1511    const auto is_empty() {
1512        return data_.empty();
1513
```

```

37     while (it != value_to_idx_.end()) {
38         for (auto &idx : it->second) {
39             std::println("remove_upper_inner");
40             data_[idx].is_removed = true;
41         }
42         ++it;
43     }
44 }
45
46 auto print() const -> void {
47     for (auto i = data_.size(); i-- > 0;) {
48         const auto entry = data_[i];
49         if (!entry.is_removed) {
50             std::println("<{:3}>", entry.value);
51         }
52     }
53 }
54
55 private:
56     std::vector<StackEntry> data_{};
57     std::map<int, std::vector<usize>> value_to_idx_{};
58 };

```

### 7.3.4 Subarrays with Given Sum and Bounded Maximum

Suppose we are given an array `nums` with `n` elements and we are interested in counting the number of contiguous subarrays which sum to some `k` and whose elements are at most `M`.

First we note that whenever we see a value  $x > M$  we have a cut, so we can see this problem as summing up the number of contiguous subarrays per block, where blocks are separated by too large values. For example suppose

$$\text{nums} = [-1, 2, 1, 7, -1, 5, 2, 1, 2, -7],$$

$k = 2$  and  $M = 3$ , then

```

1 [ -1, 2, 1, 7, -1, 5, 2, 1, 2, -7]
2      ^ {-1, 2, 1}
3          ^ {-1}
4                  ^ {2, 1, 2, -7}

```

and  $F(\text{nums}, k, M) = f(-1, 2, 1, k) + f(-1, k) + f(2, 1, 2, -7, k)$  where  $F$  denotes the main entry point and  $f$  the counts per block.

Let  $\text{pref}(x)$  for an array  $x = \{x_1, x_2, x_3, \dots\}$  be defined as the cumulative sum

$$\text{pref}(x) = \{x_1, x_1 + x_2, x_1 + x_2 + x_3, \dots\}.$$

We can then efficiently evaluate the total sum of a subarray  $[x_i, \dots, x_k]$  by computing

$$\text{pref}(x)[k] - \text{pref}(x)[i - 1]$$

(of course you cache and don't recompute it every time). The big speedup improvement comes from inserting target values into a hash map and summing up the counts while having a single running total instead of storing the prefix array.

```

1 ++counts[pref - k];
2 out += counts[pref];

```

To avoid unnecessary insertions by using `[]` we use `.find` and obtain the following per-block logic:

```

1 if(auto it = counts.find(pref - k); it != counts.end()) {
2     out += it->second;
3 }
4 ++counts[pref];

```

A block ends when a value is  $> M$  so we get the following total solution

```

1 auto out = 0ll;
2 for(auto i = 0zu; i < nums.size(); ++i) {
3     const auto num = nums[i];
4     if(num > M) {
5         counts.clear();
6         pref = 0ll;
7         continue;
8     }
9     if(auto it = counts.find(perf - k); it != counts.end()) {
10        out += it->second;
11    }
12    ++counts[perf];
13 }
14 return out;

```

## 7.4 Maximize Profit with Task Deadlines and Multiple Servers

Suppose we have  $n$  tasks and  $m$  workers. Each worker can process one task in a single timestep. Tasks have a profit and a deadline, if a task is completed before its deadline then we get that profit. The problem is to figure out what the optimal assignment of workers is to maximise the total profit.

Because each task has the same duration the optimal strategy is a rolling greedy strategy. We first collect the lists of deadlines and profits into combined Task objects (alternatively one could do permutation on `std::iota` and use that as a permutation map, but this would only make sense for significantly larger lists).

```

1 template <typename DeadlineT, typename ProfitT>
2 struct TaskT {
3     DeadlineT deadline;
4     ProfitT profit;
5 };
6 template <typename DeadlineT, typename ProfitT>
7 auto collect_tasks(const std::vector<DeadlineT>& deadlines, const std::vector<ProfitT>& profits) ->
8     TaskT<DeadlineT, ProfitT> {
9     assert(deadlines.size() == profits.size());
10    std::vector<Task> out;
11    out.reserve(deadlines.size());
12    for(auto i = 0zu; i < deadlines.size(); ++i) {
13        out.emplace_back(deadlines[i], profits[i]);
14    }
15    return out;
16 }
17 using Task = TaskT<int, int>;

```

These tasks must be sorted by their deadlines (we can't use ranges as HackerRank is stuck on C++20)

```

1 std::vector<Task> tasks = collect_tasks(deadlines, profits);
2 std::sort(
3     tasks.begin(),
4     tasks.end(),
5     [] (const Task& a, const Task& b) -> bool {
6         return a.deadline < b.deadline;
7     });

```

To efficiently solve this problem we need to be able to quickly evict tasks if they are not worth doing. This is ideally solved by using a `MinHeap`:

```

1 using MinHeap = std::priority_queue<int, std::vector<int>, std::greater<int>>;
2 // using MaxHeap = std::priority_queue<int, std::vector<int>, std::less<int>>;
3 using MaxHeap = std::priority_queue<int>;

```

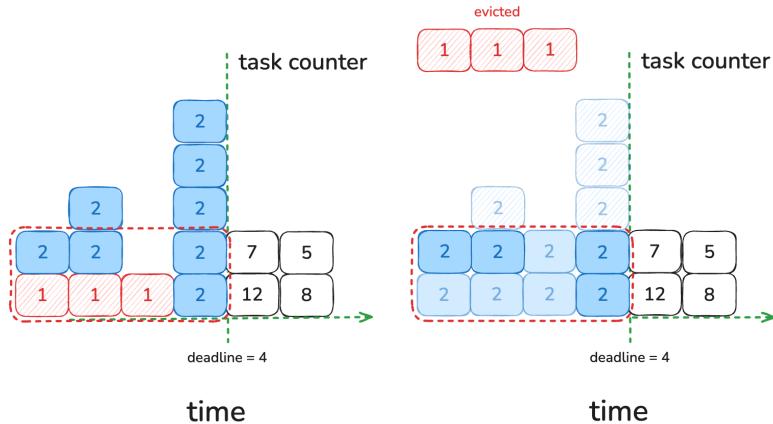


Figure 13: Maximize profit with task deadlines and multiple servers

What remains is to iterate through all tasks and check that current rolling window if a task should be evicted due to capacity. We track the total profits and greedily try to improve it.

```

1 MinHeap counts{};
2 auto sum = 0l;
3 for(auto [deadline, profit] : tasks) {
4     counts.push(profit);
5     sum += profit;
6
7     const auto capacity = static_cast<i64>(num_workers) * static_cast<i64>(d);
8     if(counts.size() >= static_cast<usize>(capacity)) {
9         // Evict worst task
10        sum -= counts.top();
11        counts.pop();
12    }
13 }
```

# 8 Software Engineering

## 8.1 Testing, Unit Tests, TDD (Test Driven Development)

Good book on testing is 15.

The most famous testing frameworks for C++ are Catch2 and Google Test (also known as GTest).

Attributes of tests to focus on:

Falsifiable Tests set up falsifiable hypothesis. If your "Test Oracle" is your own code then your test is not falsifiable, as you just prove "The code we wrote is the code we wrote". That is NOT a unit test but an acceptance test (those are important for opaque and/or legacy code you don't understand).

Repeatable You get the same answer every time.

Replicable Your colleagues get the same answer as you do.

Accuracy Measurements are "right".

Precision Measurements are "informativ".

## 8.2 Design Patterns

### 8.2.1 Visitor Pattern

In C++ this can be implemented efficiently (no runtime overhead) by using `std::variant` and `std::visit`.

### 8.2.2 Strategy Pattern

### 8.2.3 CRTP (Curiously Recurring Template Pattern) Design Pattern

### 8.2.4 Type Erasure Design Pattern

## 9 Databases

### 9.1 MySQL

### 9.2 MongoDB

# 10 Computer Architecture, HPC (High Performance Computing)

## 10.1 Memory Hierarchy and Cache

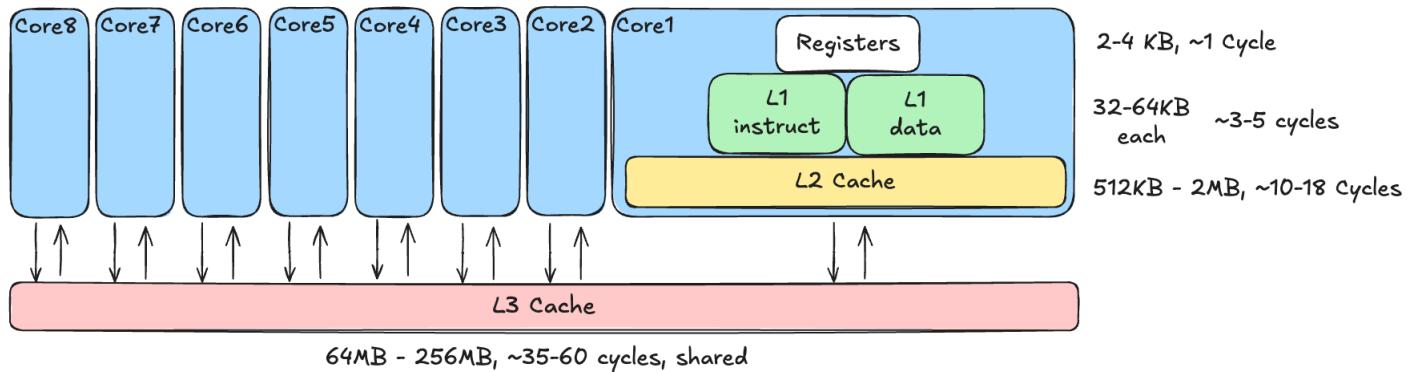


Figure 14: CPU cache hierarchy (registers, L1/L2/L3, main memory)

### 10.1.1 Cache Lines and Cache Locality

A **Word** is the width of a standard register, on modern machines this is usually **8 Byte**. The architecture defines the size of a pointer, usually it's either **4 Byte** (on 32 bit architecture) or **8 Byte** (on 64 bit architecture). A **Cache Line** is the smallest contiguous chunk of memory that the CPU can load at once. Usually it's the size of **8 Word**. We will assume that

$$\text{CacheLine} = 8 \text{ Word} = 8 * 8 \text{ Byte} = 64 \text{ Byte}.$$

### 10.1.2 Spatial vs Temporal Locality

Spatial Locality means that if you access some element then you probably will access nearby elements as well, this is referred to as spatial locality. For example if you pull a variable into cache on a cache miss you pull in an entire cacheline, if you iterate through a large number of elements sequentially (potentially strided) the CPU will see this and pre-fetch memory to be used when it's needed (pipelining).

```
1 for(auto i = 0zu; i < vec.size(); ++i) {  
2     // vec[i + 1] is already ready to be processed before this operation is done  
3     vec[i] += 1;  
4 }
```

Temporal Locality means that if you access some element then you will probably access it again soon, so it makes sense to cache it.

```
1 auto x = 1;  
2 for(auto i = 0zu; i < vec.size(); ++i) {  
3     x += 2; // x remains in L1 cache, doesn't get evicted.  
4     vec[i] += x;  
5 }
```

### 10.1.3 False Sharing

False sharing is a phenomenon that appears when you have two different threads write to elements on the same cache line. When thread a writes while thread b uses another value on the same cache line the cache line gets invalidated and so b has a cache miss and needs to pull the value into cache again despite it not having changed. To avoid this make sure that if two threads access memory often that they are not on the same cache line (easiest solution is to just pad to the next cache line, you can do that using the `std::hardware_destructive_interference_size` constant).

```
1 constexpr usize k_cacheline_size{std::hardware_destructive_interference_size};  
2  
3 struct Packed {
```

```

4     int a{};
5     int b{};
6 };
7
8 struct Separated {
9     alignas(k_cacheline_size) int a{};
10    alignas(k_cacheline_size) int b{};
11};
12
13 auto func_a(auto& Packed p) -> void {
14     p.a = p.a + 1; // invalidates cache line that Packed lives on
15 }
16
17 auto func_b(auto& Packed p) -> void {
18     p.b = p.b + 1; // invalidates cache line that Packed lives on
19 }
20
21 auto func_a_separated(auto& Separated p) -> void {
22     p.a = p.a + 1; // Doesn't invalidate cacheline
23 }
24
25 auto func_b_separated(auto& Separated p) -> void {
26     p.b = p.b + 1; // Doesn't invalidate cacheline
27 }

```

## 10.2 Memory Layout and Alignment

### 10.2.1 Alignment vs Size

### 10.2.2 Tight Packing vs Padding

### 10.2.3 AoS vs SoA vs AoSoA

Suppose you have the following representation of a physics owning entity represented as a fat struct

```

1 struct Transform {
2     Pos3 pos;
3     Dir3 scale;
4     Quaternion orientation;
5 };
6 struct Entity {
7     u32 id{};
8     Color3 color{};
9     u32 visual_mask{};
10    u32 hit_mask{};
11    std::string name{};
12    Transform transform{};
13    std::unique_ptr<RigidBody> body{};
14 };

```

If we store our entities as

```
std::vector<Entity> entities;
```

we have an (A)rray (o)f (S)tructs.

If we now want to update the position of all entities (for example by shifting them all, or syncing the transform with the underlying physics representation) then we'd have to load at least one **CacheLine** per entity which is a lot of wasted loading if we are only interested in the transform, for example

```
1 sizeof(Transform) == sizeof(Pos3) + sizeof(Dir3) + sizeof(Quaternion) == 3 * 4 + 3 * 4 + 4 * 4 == 40
```

(assuming everything consists of 32 bit floats) while

```

1 sizeof(Entity) == sizeof(u32) + sizeof(Color3) + sizeof(u32) + sizeof(u32)
2     + sizeof(std::string) + sizeof(Transform) + sizeof(std::unique_ptr)
3     == 4 + 3 * 4 + 4 + 4 + 3 * 8 + 40 + 8
4     == 96

```

Assuming things are aligned well we don't have to pull in 2 **CacheLine** but we still pull 24 Bytes of memory too much.

The code for shifting everything by {1.0f, 1.0f, 1.0f} is

```
1 const Vec3 shift{1.0f, 1.0f, 1.0f};  
2 for(auto& entity : entities) {  
3     entity.transform += shift;  
4 }
```

We instead can store the individual components contiguously in memory

```
1 TransformSOA {  
2     f32* pos_xs;  
3     f32* pos_ys;  
4     f32* pos_zs;  
5     f32* scale_xs;  
6     f32* scale_ys;  
7     f32* scale_zs;  
8     Quaternion* orientations;  
9 }
```

and now we can load in three **CacheLine** to update 8 elements at a time, so we have 3 cache misses per 16 updates in the worst case. The pointers can for example store to a ArenaAllocator or some other contiguous storage like **std::vector** or **std::pmr::vector**). The update code becomes (assuming number of entities is divisible by 8 to avoid having to deal with boundary behavior)

```
1 const auto shift_x = 1.0f;  
2 const auto shift_y = 1.0f;  
3 const auto shift_z = 1.0f;  
4 for(auto i = 0zu; i < n_entities; ++i) {  
5     pos_xs[i] += shift_x;  
6     pos_ys[i] += shift_y;  
7     pos_zs[i] += shift_z;  
8 }
```

A higher overhead but more SIMD aware storage method would be to use AoSoA, where we store arrays of blocks (here with element arrays of size 4 as I'm on NEON, can increase for AVX and AVX512):

```
1 struct TransformBlock {  
2     std::array<f32, 8> pos_x;  
3     std::array<f32, 8> pos_y;  
4     std::array<f32, 8> pos_z;  
5     /*...*/  
6 };
```

- 10.2.4 SIMD Alignment Requirements**
- 10.2.5 Practical Trade-offs (Bandwidth vs Compute)**
- 10.3 CPU Microarchitecture**
  - 10.3.1 Pipelining**
  - 10.3.2 Branch Prediction and Speculative Execution**
  - 10.3.3 Out-of-Order Execution**
- 10.4 GPU Architecture**
  - 10.4.1 Thread Blocks**
  - 10.4.2 Warps**
  - 10.4.3 Memory Coalescing**
- 10.5 CPU–GPU Interaction**
  - 10.5.1 Asynchronous Execution**
  - 10.5.2 Synchronization Primitives**
- 10.6 Benchmarking and Measurement**
  - 10.6.1 Sampling Benchmarks**
  - 10.6.2 Cache and Memory Profiling (perf, Valgrind)**

# 11 Concurrency

## 11.1 C++ Memory Model

### 11.1.1 Atomics

There is only one atomic data type which is required to be lock free, namely `std::atomic_flag`. Pretty much every core C++ data type has an atomic version, for example

```
std::atomic<bool>, std::atomic<int>, std::atomic<float>, std::atomic<double>
```

It is important to check if those have hardware support (i.e. work lock free), this can be done by checking the compile time constant

```
std::atomic<T>::is_always_lock_free
```

## 11.2 Multiple Threads

## 11.3 Multiple Processes

### 11.3.1 OpenMP

## 11.4 Data Structures

### 11.4.1 SPSC (Single Producer Single Consumer)

### 11.4.2 SPMC (Single Producer Multiple Consumer)

### 11.4.3 MPSC (Multiple Producer Single Consumer)

### 11.4.4 MPMC (Multiple Producer Multiple Consumer)

# 12 Operating Systems

## 12.1 Process, Thread

## 12.2 Scheduling

## 12.3 CPU Virtualisation

## 12.4 Memory Virtualisation

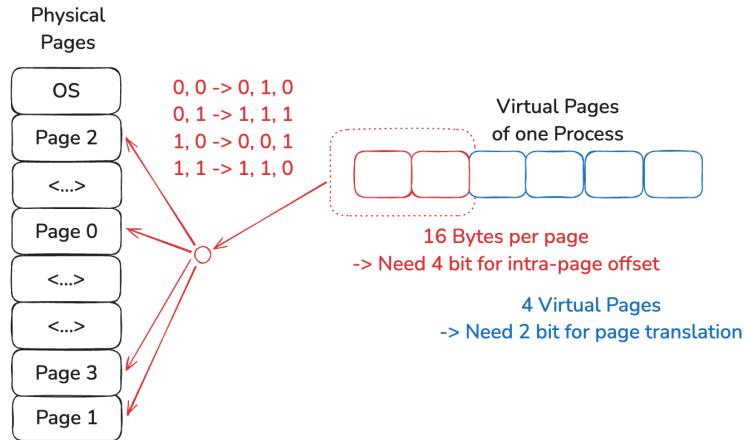


Figure 15: Virtual address translation: virtual page number + offset mapped to a physical frame number + offset

Paging as opposed to segmentation slices up the available virtual memory into fixed-size pieces.

## 13 Networking

13.1 OSI Model

13.2 UDP

13.3 TCP/IP

## 14 Trivia

### 14.1 Error #323 on GCC

There used to be a semi-famous issue where some compilers (notably GCC on x86) evaluated floating-point expressions using 80-bit x87 registers, which could lead to results that were “too correct” and therefore differ from other platforms and compilers (see Reference 15).

## 15 References

- Egor Suvorov.  
*Using Floating-point in C++: What Works, What Breaks, and Why.*  
CppCon 2025. <https://www.youtube.com/watch?v=m83TjrB6wYw>
- Klaus Iglberger, *C++ Software Design*. O'Reilly Media, 2022.
- Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books. <https://pages.cs.wisc.edu/~remzi/OSTEP/>
- Fedor Pikus, *The Art of Writing Efficient Programs*. Apress, 2021.
- Anthony Williams, *C++ Concurrency in Action* (2nd ed.), Manning Publications, 2019.
- Glenford J. Myers, *The Art of Software Testing*. Wiley, 1979.