C++ Cheatsheet

Daniel Sinkin

February 16, 2026

Contents

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::Expects` and `gsl::Ensures`) as well as a new more stable assert syntax in form of `contract_assert`.

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

```
auto div_round_down_pos(std::int32_t a, std::int32_t b) -> std::int32_t
{
    gsl::Expects(a > 0);
    gsl::Expects(b > 0);

    const std::int32_t r = a / b;
    assert((r + 1) * b > a);

    Ensures(r * b <= a);
    return r;
}
```

Listing 2: C++26: Contract syntax

```
auto div_round_down_pos(std::int32_t a, std::int32_t b) -> std::int32_t
    pre  (a > 0)
    pre  (b > 0)
    post (r * b <= a)
{
    contract_assert((r + 1) * b > a);
    const std::int32_t r = a / b;
    return r;
}
```

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

```
auto& [_, value] = f(); // Structured binding

Mutex m1{}, m2{};
{
    std::scoped_lock _{m1, m2}; // RAII utils
}
```

2.2 C++23

2.2.1 std::expected

Allows for functional / Rust style errors as values.

Listing 4: C++26: Anonymous values

```
enum class MyError {
    negative_number;
    zero_division;
```

```
}

std::expected<float, PositiveDivisionError> my_div(float a, float b) {
    if((a * b) <= 0.0f) {
        return std::unexpected{MyError:negative_number};
    }
    if(b == 0.0f) {
        return std::unexpected{MyError:zero_division};
    }
    return a / b;
}

int main() {
    auto res = my_div(5.0, 3.0);
    if(!res) {
        switch(res.err()) {
            case MyError::negative_number: {/*...*/}
            case MyError::zero_division: {/*...*/}
        }
    } else {
        float value{*res};
        /* ... */
    }
}
```

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

```
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

```
float x = 12.52343232f;
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

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

2.3.3 Ranges

2.3.4 Coroutines

```
co_await task;
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

```
using namespace std::chrono;
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

```
#include <array>

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

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

```
#include <array>

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

```
std::mutex m1, m2;
{
    std::scoped_lock locks{m1, m2};
}
```

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

```
constexpr int sum_up_to(int n)
{
    int result = 0;

    for (int i = 1; i <= n; ++i)   // constexpr loop (C++14)
    {
        if (i % 2 == 0)            // constexpr conditional (C++14)
        {
            result += i;
        }
    }

    return result;
}
```

2.6 C++11

2.6.1 Value Semantics

Added move semantics and rvalue references.

Listing 13: Move constructors, Move assignment

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

Listing 14: RAII

```
struct File {
    File(const char* path);
    ~File();
    File(File&&);
    File& operator=(File&&);
};
```

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)

```
constexpr float k_pi = 3.14f;

constexpr int square(int x) {
    return x * x;
}
```

2.6.3 Lambdas

Listing 16: C++11 Lambda

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

2.6.4 std::unordered_map

2.6.5 std::lock_guard

Listing 17: Lock Guard

```
Mutex x;
{
    std::lock_guard<Mutex> guard{x};
}
```

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

```
int x = 5;
x; // This id-expression is an L value
(x); // L value
&x; // prvalue of type int*

1 + 1; // prvalue (of type int)

struct Foo {/*...*/}
Foo foo{}; // declaration not an expression, so no value cateogry

foo; // lvalue
f(foo); // passes an L value
f(std::move(foo)); // X value
f(Foo{}); // passes an PR value
```

3.1.1 Perfect Forwarding

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

```
template <class T>
constexpr T&& forward(std::remove_reference_t<T>& t) noexcept
{
    return static_cast<T&&>(t);
}

template <class T>
constexpr T&& forward(std::remove_reference_t<T>&& t) noexcept
{
    static_assert(!std::is_lvalue_reference_v<T>,
                  "bad forward: cannot forward an rvalue as an lvalue");
    return static_cast<T&&>(t);
}
```

```
class Foo{ /*...*/ };

void kind(T) { println("1"); }
void kind(const T&) { println("2"); }
void kind(T&&) { println("3"); }

template <typename T>
void bad_forward(T &&arg)
```

```
{
    kind(arg);
}

template <typename T>
void good_forward(T &&arg)
{
    kind(std::forward<T>(arg));
}

int main()
{
    Foo f{};
    bad_forward(f);          // prints: kind(Foo&)
    bad_forward(Foo{});      // prints: kind(Foo&)
    good_forward(f);         // prints: kind(Foo&)
    good_forward(Foo{});     // prints: kind(Foo&&)
}
```

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

```
class Foo {
    Foo() {}
    ~Foo() {...} // Destructor
    Foo(const Foo&) {...} // Copy Constructor
    Foo(Foo&&) {...} // Move Constructor
    Foo& operator=(const Foo&) {...} // Copy assignment constructor
    Foo& operator=(Foo&&) {...} // Move assignment constructor
}
```

3.2.2 Rule of Zero

Given that implementingi 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.

```
class RAII {
    RAII(std::string_view name) : name_(name), data_(static_cast<int*>(std::malloc(8 *
    sizeof(int)))) {
        std::println("Allocated Memory '{}'", name_);
    }
    ~RAII() {
        std::free(data_);
        std::println("Deallocated Memory '{}'", name_);
    }
private:
```

```
    std::string name_{};
    int* data_{};
};
```

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

```
auto func() -> void {
    std::println("Starting Function");
    RAII func_scope("Function Scope");
    std::println("Starting inner scope");
    {
        RAII inner_scope("Inner Scope");
    }
    std::println("Finished inner scope");
    std::println("Throwing exception");
    throw std::runtime_error("");

    std::println("Finished function");
}

int main() {
    std::println("Starting Program");
    try {
        func();
    } catch (...) {
        std::println("Caught exception");
    }
    std::println("Finishing Program");
}
/*
Starting Function
Allocated Memory 'Function Scope'
Starting inner scope
Allocated Memory 'Inner Scope'
Deallocated Memory 'Inner Scope'
Finished inner scope
Throwing exception
Deallocated Memory 'Function Scope'
Caught exception
Finishing Program
*/
```

`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.

```
using Data = std::array<int, 32>;
class TrackMemory {
public:
    TrackMemory() {
        std::println("Empty Constructor");
    }
    TrackMemory(const TrackMemory &other) : data_(other.data_) {
        std::println("Copy constructor");
    }
    TrackMemory(TrackMemory &&other) noexcept : data_(std::move(other.data_)) {
        std::println("Move constructor");
    }
private:
    Data data_{};
};
```

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

```cpp
auto no_nrvo(bool return_a) -> TrackMemory {
    TrackMemory a{}, b{};
    if (return_a) { return a; }
    return b;
}
```

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

```cpp
auto possible_nrvo() -> TrackMemory {
    TrackMemory a{};
    return a;
}
```

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

```cpp
auto forced_nrvo() -> TrackMemory {
    return TrackMemory{};
}
```

We then run those

```cpp
int main() {
    std::println("Impossible NRVO:");
    auto tm = no_nrvo(0);
    // Empty Constructor
    // Empty Constructor
    // Move constructor
    std::println("\nPossible NRVO:");
    auto tm2 = possible_nrvo();
    // Empty Constructor
    std::println("\nForced NRVO:");
    auto tm3 = forced_nrvo();
    // Empty Constructor
}
```

3.4 EBO (Empty Base Optimisation)

```cpp
struct alignas(16) EmptyAligned {};

class NoEBO {
private:
    double data_{};
    EmptyAligned empty_interals_{};
};

class WithEBO {
private:
    double data_{};
    [[no_unique_address]] EmptyAligned empty_internals_{};
};

static_assert(std::is_empty_v<EmptyAligned>);
static_assert(sizeof(EmptyAligned) == 16);
static_assert(alignof(EmptyAligned) == 16);
static_assert(sizeof(NoEBO) == 32);
static_assert(sizeof(WithEBO) == 16);
```

3.5 Idioms and Patterns

3.5.1 Two-pointer merge

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:

```cpp
std::vector<int> a{/*...*/};
std::vector<int> b{/*...*/};
std::vector<int> c{};
c.reserve(a.size() + b.size());

auto it_a = a.begin();
auto it_b = b.begin()
while(it_a != a.end() && it_b != b.end()) {
    if(*it_a <= *it_b) {
        c.push_back(*it_a);
        ++it_a;
    } else {
        c.push_back(*it_b);
        ++it_b;
    }
}
while(it_a != a.end()) {
    c.push_back(*it_a);
    ++it_a;
}
while(it_b != b.end()) {
    c.push_back(*it_b);
    ++it_b;
}
```

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.

```cpp
struct DeleteableInt{
    int value{};
    bool is_deleted{false};
};

std::vector<DeletableInt> vec{};
vec.resize(100);
for(auto i = 0; i < vec.size(); ++i) {
    if(i & 1 == 0) {
        vec[i].is_deleted = true;
    }
}
```

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

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

```

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

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 >= 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, ..., 0`.

Listing 20: Reverse loop for unsigned indices (safe)

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

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 decalre 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 specificied (standard only guarantees it within one translation unit (top-to-bottom)).

Listing 21: a.cpp

```
extern std::string g_name;

std::string make_greeting() { return "Hello " + g_name; }

std::string g_greeting = make_greeting();
```

Listing 22: b.cpp

```
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

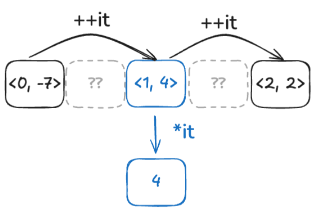
3.9.1 Variadic Templates

3.9.2 Template Metaprogramming

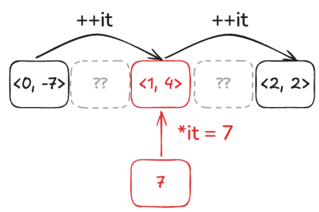
4 STL (Standard Templating Library)

4.1 Iterators

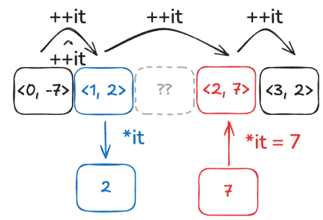
4.1.1 Iterator Categories



Input iterator

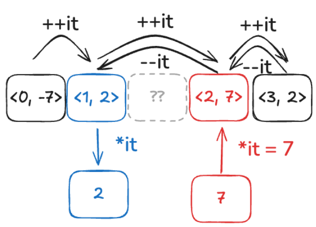


Output iterator

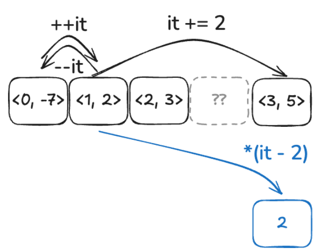


Forward iterator

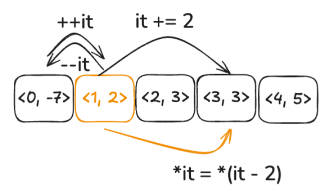
Figure 1: Single-pass and forward iterators



Bidirectional iterator



Random-access iterator



Contiguous iterator

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:

```
\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 mathematicl elements of vector spaces and physics vectors. A better name would for example be DynamicArray.

Listing 23: Vector class

```
template <typename T>
class VectorTrivial
{
    static_assert(std::is_trivially_copyable_v<T>);

public:
    using size_type = std::size_t;
    /*Rule of 5*/
    /*Public API*/
private:
    T* start_{};
    T* end_{};
    T* cap_{};
    /*Private API*/
};
```

5.1.1 Rule of 5

Listing 24: Rule of 5

```
VectorTrivial() = default;
explicit VectorTrivial(size_type n)
{ // Allocate with fixed capacity
    start_ = static_cast<T*>(std::malloc(n_capacity * sizeof(T)));
    if(!start_) {/*...*/}
    end_ = start_;
    cap_ = start_ + n_capacity;
}

VectorTrivial(const Vector& other)
{ // Copy Constructor
    const auto n_elems = other.size();
    const auto n_capacity = other.capacity();

    start_ = static_cast<T*>(std::malloc(n_capacity * sizeof(T)));
    if(!start_) {/*...*/}
    std::memcpy(start_, other.start_, n_elems * sizeof(T));
    end_ = start_ + n_elems;
    cap_ = start_ + n_capacity;
}

VectorTrivial(Vector&& other) noexcept
    : start_(other.start_), end_(other.end_), cap_(other.cap_)
{ // Move Constructor
    other.start_ = nullptr;
    other.end_ = nullptr;
```

```
    other.cap_ = nullptr;
}

VectorTrivial& operator=(const Vector& other)
{ // Copy Assignment Constructor
    if(this == &other)
    {
        return *this;
    }

    const auto n_elems = other.size();
    const auto n_capacity = other.capacity();o

    auto new_start = static_cast<T*>(std::malloc(n_capacity * sizeof(T)));
    if(!new_start) { /* ... */ }

    std::memcpy(new_start, other.start_, n_elems * sizeof(T));

    std::free(start_);
    start_ = new_start;
    end_ = start_ + n_elems;
    cap_ = start_ + n_capacity;
    return *this;
}

VectorTrivial& operator=(Vector&& other) noexcept
{ // Move Assignment Constructor
    if(this == &other)
    {
        return *this;
    }

    std::free(start_);
    start_ = other.start_;
    end_ = other.end_;
    cap_ = other.cap_

    other.start_ = nullptr;
    other.end_ = nullptr;
    other.cap_ = nullptr;
    return *this;
}
~VectorTrivial()
{ // Destructor
    std::free(start_);
}
```

5.1.2 Public API

```
[[nodiscard]] auto size() const noexcept -> size_type
{
    return static_cast<size_type>(end_ - start_);
}
[[nodiscard]] auto capacity() const noexcept -> size_type
{
    return static_cast<size_type>(cap_ - start_);
}
[[nodiscard]] auto empty() const noexcept -> bool
{ // More efficient than checking size() == 0
    return end_ == start_;
}
void push_back(const T& v)
{ // push copy of element to the back, growing if necessary
```

```
    if (end_ == cap_) { grow_(); }
    *end_ = v;
    ++end_;
}
void push_back(T&& v)
{ // take ownership of element and push it to the back, growing if necessary
    if (end_ == cap_) { grow_(); }
    *end_ = std::move(v);
    ++end_;
}
```

5.1.3 Private API

```
// 2.0 on MSVC, 1.5 on GCC and Clang
constexpr double k_resize_factor = 2.0;
constexpr double k_initial_capacity = 1;
{
    const auto n_elems = size();
    const auto old_cap = capacity();
    const auto new_cap = old_cap * k_resize_factor;
    const auto new_cap = (old_cap == 0) ? 8 : new_cap;

    void* new_mem = std::realloc(start_, new_cap * sizeof(T));
    if (!new_mem) { /* handle allocation failure */ }

    start_ = static_cast<T*>(new_mem);
    end_ = start_ + n_elems;
    cap_ = start_ + new_cap;
}
```

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-trival 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.

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

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.

```cpp
// Take ownership over a pointer
UniquePtr(pointer_type ptr) { ptr_ = ptr; }

// Copying is disallowed
UniquePtr(const UniquePtr&) = delete;
UniquePtr& operator=(const UniquePtr&) = delete;

// Moving is allowed
UniquePtr(UniquePtr&& other) noexcept(/*Deleter must be noexcept moveable and move constructible*/)
    : ptr_(other.release()), deleter_(std::move(other.deleter_)) {}

UniquePtr& operator=(UniquePtr&& other) {
    if(*this == other) {
        return this;
    }
    if(ptr_) {
        deleter_(ptr_);
    }

    ptr_ = other.release();
    deleter_ = std::move(other.deleter_);
    return *this;
}

~UniquePtr() {
    if(ptr_) {
        deleter_(ptr_);
    }
}
```

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.

```cpp
auto release() -> pointer_type {
    auto ptr = ptr_;
    ptr_ = nullptr;
    return ptr;
}

auto get() const -> pointer_type {
    return ptr_;
}
```

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 tempalted over the underlying type. There must be two overloads, one for arrays and one for non-arrays.

```cpp
template <class T>
struct DefaultDelete {
    auto operator()(T* p) -> void {
        delete p;
    }
}

template <class T>
struct DefaultDelete<T[]> {
    auto operator()(T* p) -> void {
        delete[] p;
    }
```

```
}
```

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.

```
template <class T, class...Args>
    requires (!std::is_array(T))
auto make_unique(Args&&...args) -> UniquePtr<T> {
    return UniquePtr<T>(new T(std::forward<Args>(args)...));
}

template <class T, class...Args>
    requires (std::is_array(T) && std::extent_v<T> == 0)
auto make_unique(Args&&...args) -> UniquePtr<T> {
    using  U = std::remove_extent_t<T>;
    return UniquePtr<U>(new U[n]());
}

template <class T, class...Args>
    requires (std::is_array(T) /*&& bounds side > 0*/)
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.

```
template <class T>
class Optional{
public:
    /*Rule of 5*/
    /*Public API*/
private:
    bool is_filled_{false};
    alignas(T) std::byte storage[sizeof(T)];
    /*Private API*/
};
```

5.4.1 Public API

```
auto release() -> void {
    if(is_filled_) {
        *ptr()->~T();
        is_filled_ = false;
    }
}
template <class...Args>
auto emplace(Args...args) -> void {
    release();
    ::new (static_cast<void*>storage())
        T(std::forward<Args>(args));
    is_filled_ = true;
}
explicit bool() const noexcept {
    return is_filled_;
}
[[nodiscard]] auto has_value() const noexcept -> bool {
```

```
    return is_filled_;
}
[[nodiscard]] auto value() & -> T& {
    return **ptr();
}
[[nodiscard]] auto value() const & -> const T& {
    return **ptr();
}
[[nodiscard]] auto value() & -> T&& {
    return **ptr();
}
[[nodiscard]] auto value() const & -> const T&& {
    return **ptr();
}
```

5.4.2 Rule of 5

```
Optional() = default();
Optional(const T& t) {
    emplace(t);
}
Optional(T&& t) {
    emplace(t);
}
Optional(const Optional& other) {
    if(other.is_filled_) {
        emplace(*other.ptr());
        is_filled_ = true;
    }
}
Optional(Optional&& other) {
    if(other.is_filled_) {
        emplace(*other.ptr());
        is_filled_ = true;
    }
}
~Optional() {
    reset();
}
Optional& operator=(const Optional& other) {
    if(other == *this) {
        return *this;
    }
    if(other.is_filled_) {
        emplace(*other.ptr());
        is_filled_ = true;
    }
    return *this;
}
Optional& operator=(Optional&& other) {
    if(other == *this) {
        return *this;
    }
    if(other.is_filled_) {
        emplace(*other.ptr());
        is_filled_ = true;
    }
    return *this;
}
```

5.4.3 Private API

```cpp
auto ptr() -> T& {
    return std::launder(reinterpret_cast<T>(storage_));
}
auto ptr() const -> const T& {
    return std::launder(reinterpret_cast<const T>(storage_));
}
auto storage() -> T* {
    return storage_;
}
auto storage() const -> const T* {
    return storage_;
}
```

5.5 Tree

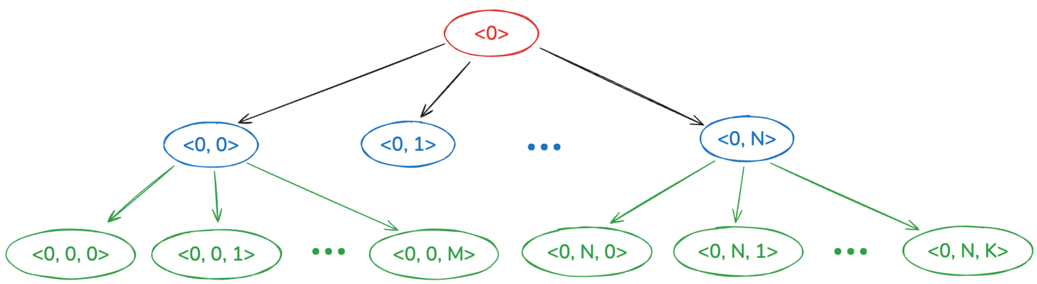


Figure 3: General tree (arbitrary number of children per node)

```cpp
struct TreeNode {
    int value{};
    std::vector<std::unique_ptr<TreeNode>> children{};
};
```

5.6 Binary Tree

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

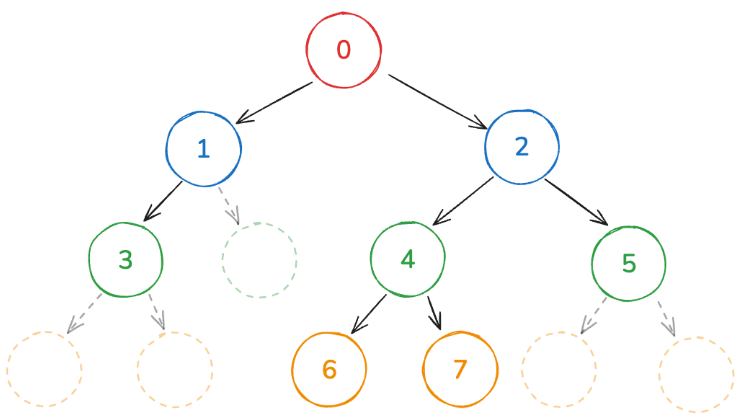


Figure 4: Binary tree (each node has at most two children)

```cpp
struct BinaryTreeNode {
    int value{};
    std::unique_ptr<std::unique_ptr<TreeNode>> left{};
    std::unique_ptr<std::unique_ptr<TreeNode>> right{};
};
```

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.

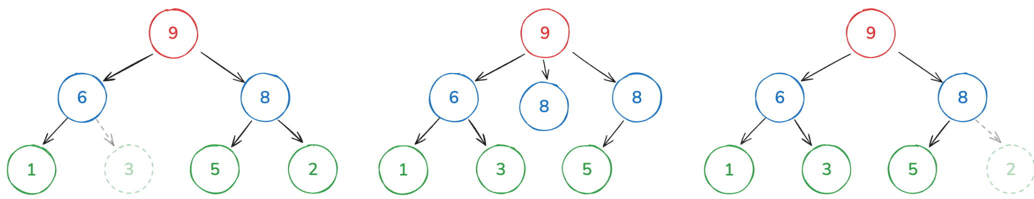


Figure 5: Left is a binary tree that is not complete, middle is not a binary tree, right is a complete binary tree

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

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

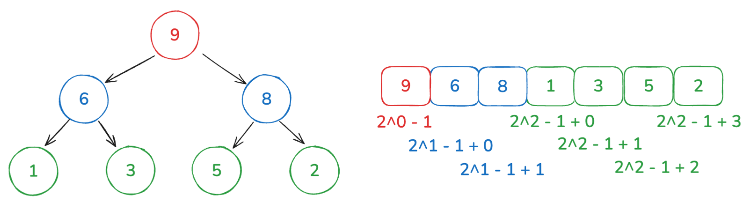


Figure 6: Complete binary tree (filled left-to-right)

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).

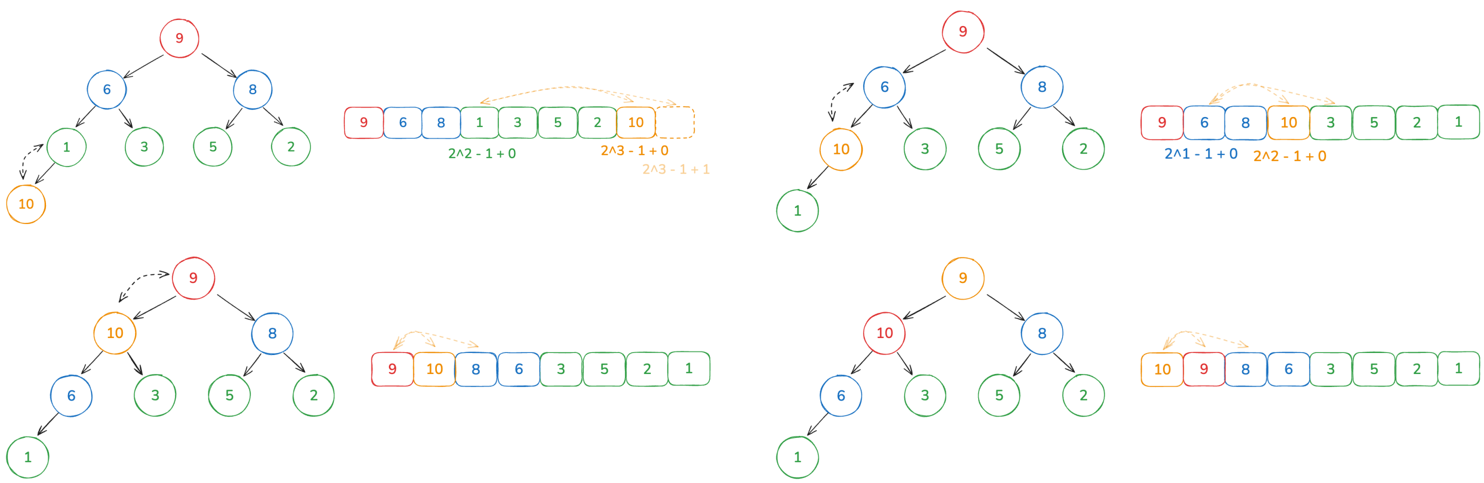


Figure 7: Max-heap insertion (heapify-up steps)

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-balancnig 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.

```
struct ControlBlock {
    std::atomic<usize> strong{};
    std::atomic<usize> weak{};
    void (*deleter)(void*);
    void* ptr;
}
```

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)

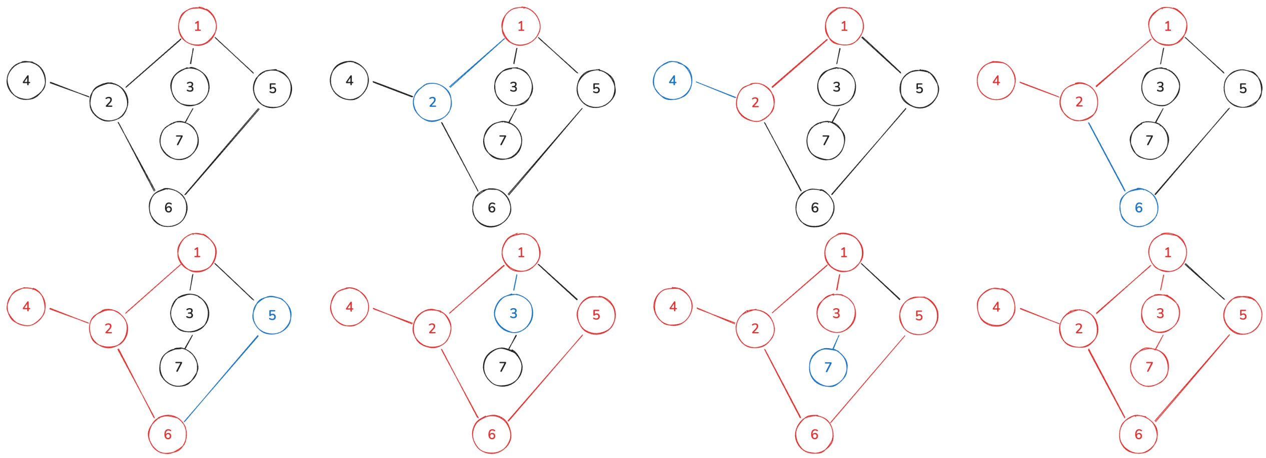


Figure 8: Depth First Search traversal steps (min-neighbor tie rule)

6.2 BFS (Breadth First Search)

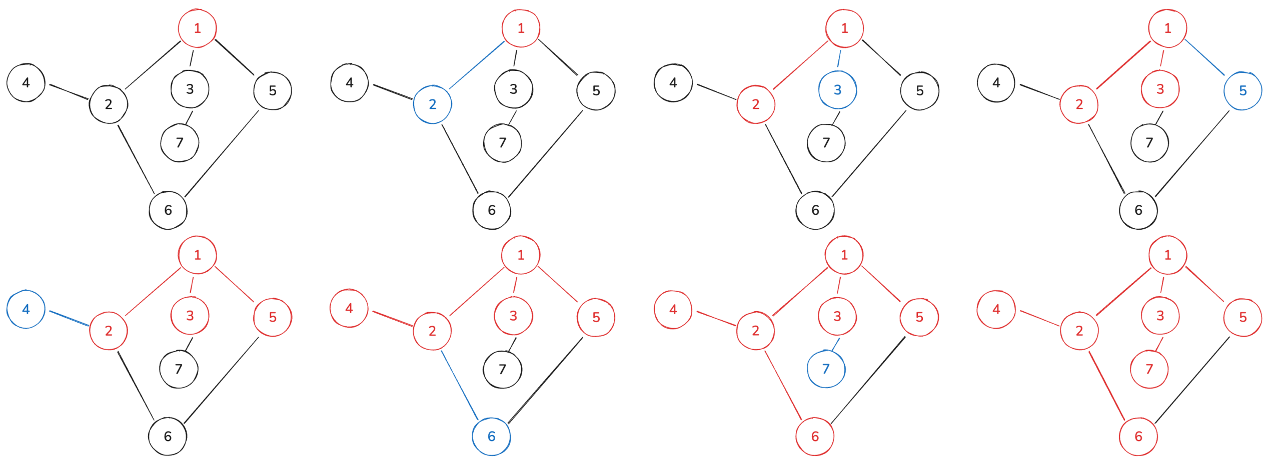


Figure 9: Breadth First Search traversal steps (min-neighbor tie rule)

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

$$\texttt{xs[i - a] = xs[i + a]}$$

and even palindrome

$$\texttt{xs[i - a] = xs[i + 1 + a]}$$

in the following way:

```cpp
const auto n = static_cast<int>(s.size());
auto palindrome_count = 0;
const auto expand_and_update = [&](int left, int right) -> void {
    while(left >= 0 && right < n && s[left] == s[right]) {
        --left;
        ++right;
        ++palindrome_count;
    }
};
for(auto i = 0; i < n; ++i) {
    expand_and_update(i, i); // Odd Palindrome
    expand_and_update(i, i + 1); // Even Palindrome
}
```

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.

```cpp
auto top_k_naive(const std::vector<int>& xs, int k) -> std::vector<int> {
    std::priority_queue<int> max_heap{};
    for(const auto x : xs) {
        heap.push(x);
    }
    std::vector<int> out;
    out.reserve(static_cast<usize>(k));
    for(auto i = 0zu; !heap.empty() && i < k; ++i) {
        out.push_back(heap.top());
        heap.pop();
    }
    return out;
}
```

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:

```cpp
auto top_k(const std::vector<int>& xs, int k) -> std::vector<int> {
    std::priority_queue<int, std::vector<int>, std::greater<int>> min_heap{};
    for(const auto x : xs) {
        min_heap.push(x);
        if(min_heap.size() > k) {
            // Pops the smallest element
            min_heap.pop();
        }
    }
    std::vector<int> out;
    out.reserve(static_cast<usize>(k));
    for(auto i = 0zu; !min_heap.empty() && i < k; ++i) {
        out.push_back(min_heap.top());
        min_heap.pop();
```

```
    }
    return out;
}
```

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

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

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

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

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. □

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.

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

7.1.4 Rolling Average

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

Claim. For all $k \geq 1$,

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

Proof by induction. **Base case.**

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

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

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

Then

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

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

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

$$= \frac{k\operatorname{avg}(k) + a_{k+1}}{k+1}. \tag{4}$$

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

Examples.

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

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

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

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

$$i + j = L, \qquad 0 \le i \le n, \qquad 0 \le j \le m.$$

Define boundary values (with sentinels):

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

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

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

When valid, the lower median is

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

We binary search i in the feasible range

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

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

7.1.6 Prefix Sums

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

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

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

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.

$$\sum_{j=3}^{5} x_j = x_3 + x_4 + x_5$$

$$= (x_1 + x_2 + x_3 + x_4 + x_5) - (x_1 + x_2)$$

$$= S(5) - S(2).$$

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 Unique Paths

This is basically THE intro DP / memoization problem. Suppose you start at `(0, 0)` and you can only go to the right or to the bottom. How many unique paths are there to get to `(m, n)`?

Idea is to memoize on every position, the value of very position is simply the sum of the value of going right once and of going going down once, this causes a lot of overcounting, with memoization we avoid this overcounting.

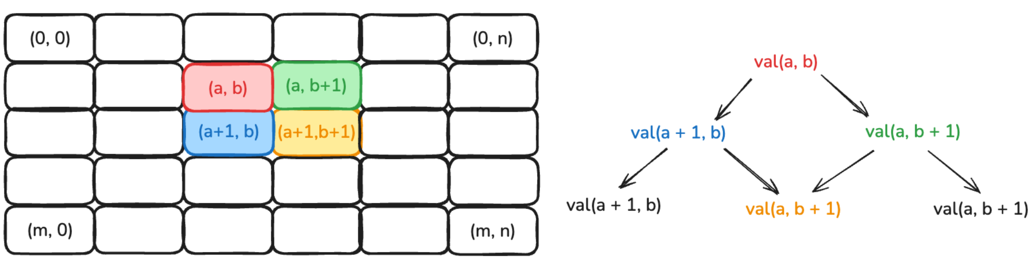


Figure 10: Unique Paths Memoization

```
constexpr auto k_not_init = -1;

class Solution {
    auto uniques_path(int m, int n) -> int {
        m_ = m;
        n_ = n;
        memo_.assign(m * n, k_not_init);

        return helper(0, 0);
    }

private:
```

```
    std::vector<int> memo_;
    int m_;
    int n_;

    auto get(int m, int n) -> usize {
        return m * n_ + n;
    }

    auto helper(int m, int n) -> int {
        if(m >= m_ || n >= n_) return 0;
        if(m == m_ && n == n_) return 1;

        const auto idx = get(m, n);
        if(memo_[idx] == k_not_init) {
            memo_[idx] = helper(m + 1, n) + helper(m, n + 1);
        }
        return memo_[idx];
    }
};
```

7.2.2 Merge sorted linked lists

Given two (ascendingly) sorted linked lists `a`, `b` we want to merge `b` into `a` such that a is still sorted. The main tricky part is to realise that adding a dummy node makes the merging much simpler.

```
auto merge2(ListNode* a, ListNode* b) -> ListNode* {
    ListNode dummy();
    ListNode* tail = &dummy;

    while(a && b) {
        if(a->val <= b->val) {
            tail->next = a;
            a = a->next;
        } else {
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    tail->next = a ? a : b;
    return dummy;
}
```

This can then easily be extended to merging arbitrarily many linked lists:

```
auto mergeK(std::vector<ListNode*> lists) -> ListNode* {
    if(lists.empty()) {
        return nullptr;
    }
    if(lists.size() == 1) {
        return lists[0];
    }
    auto curr = lists[0];
    for(auto i = 1zu; i < lists.size(); ++i) {
        curr = merge2(curr, lists[i]);
    }
    return curr;
}
```

If the lists don't alias then this is embarassingly parallel, can just do a type of tree reduce on the individual combinations.

7.2.3 Rotting Fruit

```cpp
auto orangesRotting(vector<vector<int>>& grid) -> int {
    const auto n = static_cast<int>(grid.size());
    const auto m = static_cast<int>(grid[0].size());

    auto n_fresh = 0;
    std::queue<Pos> rotten_current_step{};
    std::queue<Pos> rotten_next_step{};
    for(auto y = 0; y < n; ++y) {
        for(auto x = 0; x < m; ++x) {
            const auto cell = grid[y][x];
            if(cell == k_fresh) {
                ++n_fresh;
            } else if (cell == k_rotten) {
                rotten_current_step.push({y, x});
            }
        }
    }

    const auto process_neighbor = [&](int y, int x) {
        if(y < 0 || y >= n || x < 0 || x >= m) return;
        if(grid[y][x] != k_fresh) return;
        grid[y][x] = k_rotten;
        --n_fresh;
        rotten_next_step.push({y, x});
    };
    auto time = 0;
    while(true) {
        while(!rotten_current_step.empty()) {
            const auto [y, x] = rotten_current_step.front();
            rotten_current_step.pop();

            process_neighbor(y, x + 1);
            process_neighbor(y, x - 1);
            process_neighbor(y + 1, x);
            process_neighbor(y - 1, x);
        }
        if(rotten_next_step.empty()) {
            break;
        }
        std::swap(rotten_current_step, rotten_next_step);
        ++time;
    }
    if(n_fresh > 0) {
        return k_fruit_remain_at_end;
    }
    return time;
}
```

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`.

```cpp
using OldNode = Node;
using NewNode = Node;

auto graph_deep_copy(OldNode* node) -> Node* {
    if(!node) return nullptr;

    std::unordered_map<OldNode*, NewNode*> old_to_new{};
    std::stack<OldNode*> to_visit{};
```

```
    old_to_new[node] = new NewNode(node->val);
    to_visit.push(node);
    while(!to_visit.empty()) {
        OldNode* curr = to_visit.top();
        to_visit.pop();

        NewNode* curr_cpy = old_to_new[curr];
        for(OldNode* nb : curr->neighbors) {
            if(!nb) continue;
            if(const auto it = old_to_new.find(nb); it == old_to_new.end()) {
                old_to_new[nb] = new NewNode(nb->val);
                to_visit.push(nb);
            }
            curr_cpy->neighbors.push_back(old_to_new[nb]);
        }
    }
    return old_to_new[node];
}
```

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 ne w subarray or extend the current one. There is never a reason to backtrack.

```
auto best_sum = std::numeric_limits<int>::lowest();
auto current_sum = 0;
for(const auto num : nums) {
    current_sum = std::max(
        current_sum + num, // Extend current subarray
        num // Make a new subarray
    );
    best_sum = std::max(best_sum, current_sum);
}
return best_sum;
```

7.3.2 Maximum Product Subarray

This is a variation on the "Maximum Subarray" problem where we consider products. This case is a bit more tricky for two reasons:

- Products with 0 become 0
- We must also track rolling min because product with negative number makes larger number MORE negative than smaller number (assuming both are positive)

```
auto max_product_subarray(const std::vector<int>& nums) -> int {
    auto max_prod = nums[0];
    auto min_prod = nums[0];
    auto best_prod = nums[0];

    for(auto i = 1zu; i < nums.size(); ++i) {
        const auto x = nums[i];

        if(x < 0) std::swap(max_prod, min_prod);

        max_prod = std::max(max_prod * x, x);
        min_prod = std::min(min_prod * x, x);

        best_prod = std::max(best_prod, max_prod);
    }
```

```
    return best_prod;
}
```

7.3.3 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], ...]`, `[B[k_B], B[k_B + 1], A[k_B + 2], ...]`

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

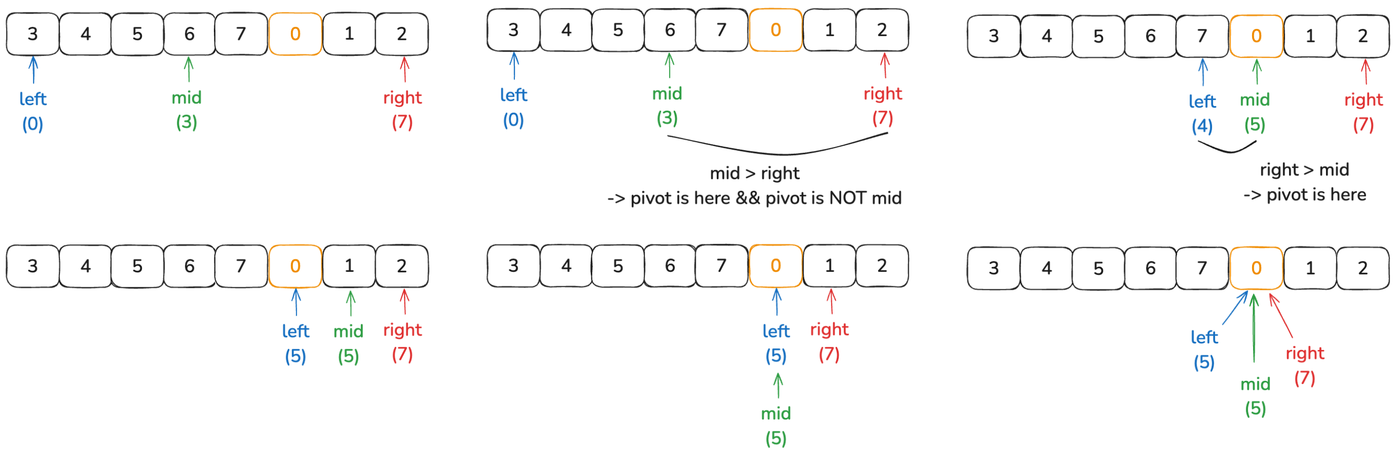


Figure 11: Pivot + partition-binary-search steps (circularly sorted arrays)

```cpp
template <typename T>
auto compute_pivot(std::span<const T> xs) -> usize {
    if (xs.empty()) { return 0zu; }

    auto left = 0zu;
    auto right = xs.size() - 1zu;
    while (left < right) {
        const usize mid = std::midpoint(left, right);

        if (xs[mid] < xs[right]) {
            // Pivot in RHS
            right = mid;
        }
        else if (xs[mid] > xs[right]) {
            // Pivot in LHS
            left = mid + 1zu;
        }
        else {
            assert(xs[mid] == xs[right]);
            --right;
        }
    }
    return left;
}
```

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

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

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.

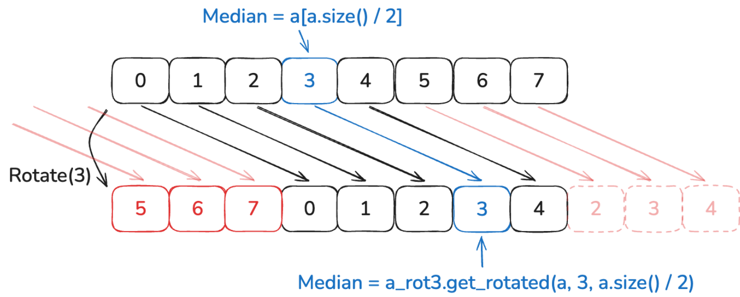


Figure 12: Median of Rotation

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:

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

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

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

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

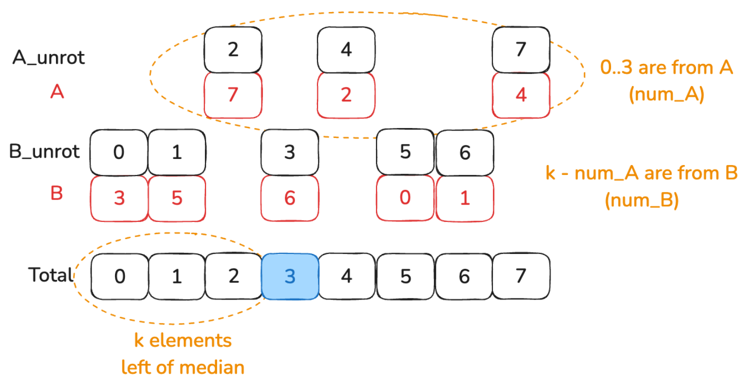


Figure 13: Visualisation of Bounds

We are interested in having `num_left = num_A + num_B` elements in total where `num_{A, 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.).

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

    constexpr auto long_min = std::numeric_limits<long>::min();
    constexpr auto long_max = std::numeric_limits<long>::max();

    const auto A_left  = (num_A == 0)        ? long_min : get_rotated(A, pivot_A, num_A - 1);
    const auto A_right = (num_A == A.size()) ? long_max : get_rotated(A, pivot_A, num_A);
```

```
    const auto B_left  = (num_B == 0)        ? long_min : get_rotated(B, pivot_B, num_B - 1);
    const auto B_right = (num_B == A.size()) ? long_max : get_rotated(B, pivot_B, num_B);

    if(A_left <= B_right && B_left && A_right) { // Partition is Valid
        return std::max(A_left, B_left);
    }

    if(A_left > B_right) {
        num_A_max = num_A - 1;
    } else {
        assert(B_left > A_right);
        num_B_max = num_A + 1;
    }
}
```

7.3.4 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.

```
class LazyDeleteStack {
public:
    auto push(int value) -> void {
        const usize idx = data_.size();
        data_.push_back({value, false});
        value_to_idx_[value].push_back(idx);
    }

    auto pop() -> void {
        while (!data_.empty()) {
            const auto top = data_.back();

            const auto entry = data_.back();
            data_.pop_back();
            value_to_idx_[entry.value].pop_back();

            if (!top.is_removed) {
                // break on first alive object deleted
                break;
            }
        }
    }

    auto remove_lower(int value) -> void {
        auto it = value_to_idx_.begin();
        const auto end = value_to_idx_.lower_bound(value);
        while (it != end) {
            for (auto idx : it->second) {
                data_[idx].is_removed = true;
            }
            ++it;
        }
    }

    auto remove_upper(int value) -> void {
        auto it = value_to_idx_.upper_bound(value);
        while (it != value_to_idx_.end()) {
            for (auto &idx : it->second) {
                std::println("remove_upper inner");
                data_[idx].is_removed = true;
            }
            ++it;
```

```
        }
    }

    auto print() const -> void {
        for (auto i = data_.size(); i-- > 0;) {
            const auto entry = data_[i];
            if (!entry.is_removed) {
                std::println("<{:3}>", entry.value);
            }
        }
    }

private:
    std::vector<StackEntry> data_{};
    std::map<int, std::vector<usize>> value_to_idx_{};
};
```

7.3.5 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 seperated by too large values. For example suppose

`nums = [-1, 2, 1, 7, -1, 5, 2, 1, 2, -7]`,

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

```
[ -1, 2,  1,  7, -1, 5, 2, 1, 2, -7]
              ^ {-1, 2, 1}
                     ^ {-1}
                               ^ {2, 1, 2, -7}
```

and `F(nums, k, M) = f(-1, 2, 1, k) + f(-1, k) + f2, 1, 2, -7, k` where `F` denotes the main entry point and `f` the counts per block.

Let `pref(x)` for an array `x = {x1, x2, x3, ...}` be defined as the cummulative sum

`pref(x) = {x1, x1 + x2, x1 + x2 + x3, ...}`.

We can then efficiently evaluate the total sum of a subarray `[xi, ..., xk]` by computing

`pref(x)[k] - perf(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.

```
++counts[pref - k];
out += counts[pref];
```

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

```
if(auto it = counts.find(perf - k); it != counts.end()) {
    out += it->second;
}
++counts[perf];
```

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

```
auto out = 0ll;
for(auto i = 0zu; i < nums.size(); ++i) {
    const auto num = nums[i];
    if(num > M) {
        counts.clear();
        pref = 0ll;
```

```
        continue;
    }
    if(auto it = counts.find(perf - k); it != counts.end()) {
        out += it->second;
    }
    ++counts[perf];
}
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).

```
template <typename DeadlineT, typename ProfitT>
struct TaskT {
    DeadlineT deadline;
    ProfitT profit;
};
template <typename DeadlineT, typename ProfitT>
auto collect_tasks(const std::vector<DeadlineT>& deadlines, const std::vector<ProfitT>& profits) ->
   TaskT<DeadLineT, ProfitT> {
    assert(deadlines.size() == profits.size());
    std::vector<Task> out;
    out.reserve(deadlines.size());
    for(auto i = 0zu; i < deadlines.size(); ++i) {
        out.emplace_back(deadlines[i], profits[i]);
    }
    return out;
}
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)

```
std::vector<Task> tasks = collect_tasks(deadlines, profits);
std::sort(
    tasks.begin(),
    tasks.end(),
    [](const Task& a, const Task& b) -> bool {
        return a.deadline < b.deadline;
    });
```

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`:

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

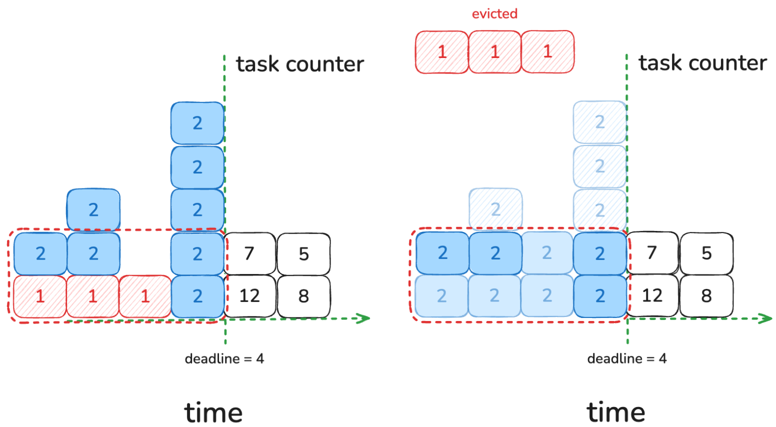


Figure 14: Maximize profit with task deadlines and multiple servers

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.

```
MinHeap counts{};
auto sum = 0l;
for(auto [deadline, profit] : tasks) {
    counts.push(profit);
    sum += profit;

    const auto capacity = static_cast<i64>(num_workers) * static_cast<i64>(d);
    if(counts.size() >= static_cast<usize>(capacity)) {
        // Evict worst task
        sum -= counts.top();
        counts.pop();
    }
}
```

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

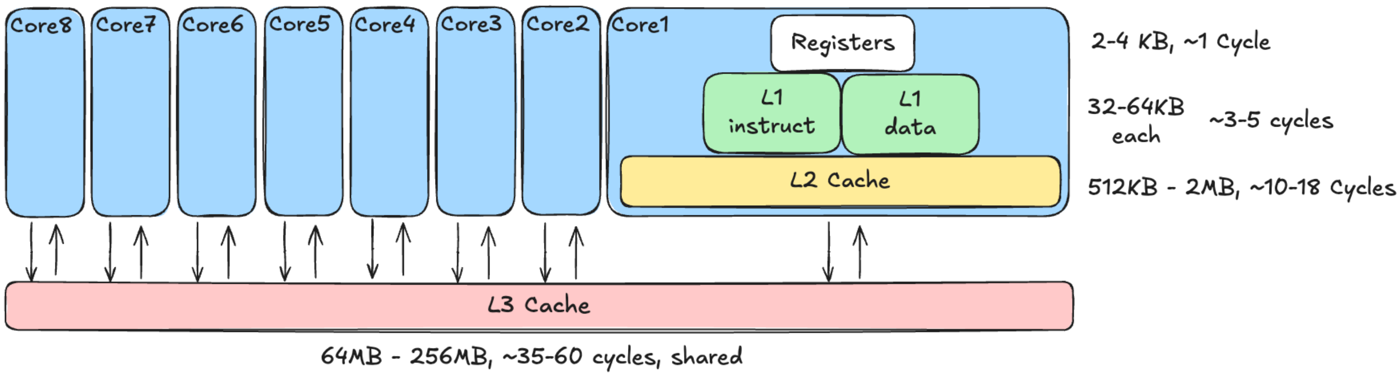


Figure 15: CPU cache hierarchy (registers, L1/L2/L3, main memory)

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

`CacheLine = 8 Word = 8 * 8 Byte = 64 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).

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

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

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

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.

```
constexpr usize k_cacheline_size{std::hardware_destructive_interference_size};

struct Packed {
```

```cpp
    int a{};
    int b{};
};

struct Seperated {
    alignas(k_cacheline_size) int a{};
    alignas(k_cacheline_size) int b{};
};

auto func_a(auto& Packed p) -> void {
    p.a = p.a + 1; // invalidates cache line that Packed lives on
}

auto func_b(auto& Packed p) -> void {
    p.b = p.b + 1; // invalidates cache line that Packed lives on
}

auto func_a_seperated(auto& Seperated p) -> void {
    p.a = p.a + 1; // Doesn't invalidate cacheline
}

auto func_b_seperated(auto& Seperated p) -> void {
    p.b = p.b + 1; // Doesn't invalidate cacheline
}
```

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

```cpp
struct Transform {
    Pos3 pos;
    Dir3 scale;
    Quaternion orientation;
};
struct Entity {
    u32 id{};
    Color3 color{};
    u32 visual_mask{};
    u32 hit_mask{};
    std::string name{};
    Transform transform{};
    std::unique_ptr<RigidBody> body{};
};
```

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 entites (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

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

(assuming everything consists of 32 bit floats) while

```cpp
sizeof(Entity) == sizeof(u32) + sizeof(Color3) + sizeof(u32) + sizeof(u32)
                  + sizeof(std::string) + sizeof(Transform) + sizeof(std::unique_ptr)
               == 4 + 3 * 4 + 4 + 4 + 3 * 8 + 40 + 8
               == 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

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

We instead can store the individual components contiguously in memory

```
TransformSOA {
    f32* pos_xs;
    f32* pos_ys;
    f32* pos_zs;
    f32* scale_xs;
    f32* scale_ys;
    f32* scale_zs;
    Quaternion* orientations;
}
```

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)

```
const auto shift_x = 1.0f;
const auto shift_x = 1.0f;
const auto shift_x = 1.0f;
for(auto i = 0zu; i < n_entities; ++i) {
    pos_xs[i] += shift_x;
    pos_ys[i] += shift_y;
    pos_zs[i] += shift_z;
}
```

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):

```
struct TransformBlock {
    std::array<f32, 8> pos_x;
    std::array<f32, 8> pos_y;
    std::array<f32, 8> pos_z;
    /*...*/
};
```

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

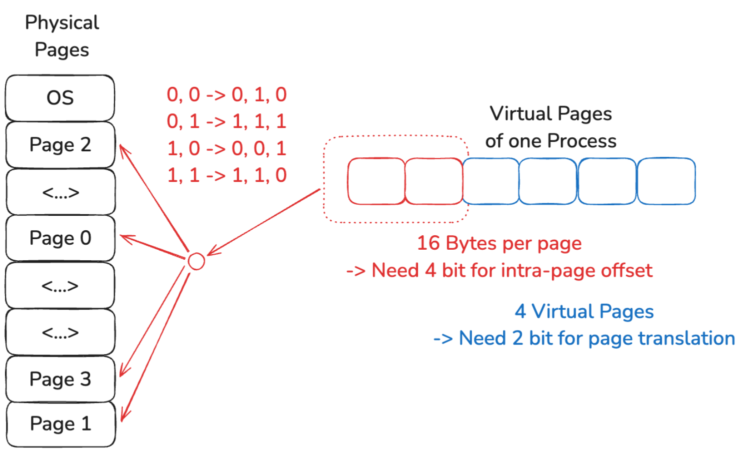


Figure 16: Virtual address translation: virtual page number + offset mapped to a physical frame number + offset

Paging as opposed to segmentation slices up the availiable 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.