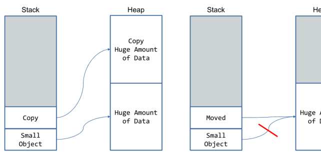
C++ Advanced | CPlA
Zusammenfassung

1. MOVE SEMANTICS

- Copying
- Moving



Sometimes, it is desirable to avoid copying values around for *performance reasons*.
– With a *copy*, the stack and heap data is copied. The new element is completely independent of the old one.
– With a *move*, only the stack data is copied. The old heap pointer gets deleted and the new one is attached to the same heap data. *Attention:* The old stack data is still valid, but in a indeterminate state, risk of dangling pointers.

Ownership Transfer: Resources of expiring values can be transferred to a new owner. Might be *more efficient* than a (deep) copy and destroying the original. Might be feasible when copying is not *(e.g. std::unique_ptr)*. Examples of such resources: Heap Memory, Locks, (File) Handles.

1.1. EXAMPLE MOVE CONSTRUCTOR (TESTAT 1)

```
struct ContainerForBigObject {
 ContainerForBigObject() // Default constructor
  : resource{std::make_unique<BigObject>()} {} // make_unique creates heap obj. & pointer

 ContainerForBigObject(ContainerForBigObject const& other) // Copy constructor
  : resource{std::make_unique<BigObject>(*other.resource)} {}
  // Creates a copy of the heap object and a new pointer to the copy

 ContainerForBigObject(ContainerForBigObject&& other) // Move constructor
  : resource{std::move(other.resource)} {}
  // New pointer points to the same heap object, 'other' is valid, but indeterminate state

 // Copy assignment operator
 auto operator=(ContainerForBigObject const& other) → ContainerForBigObject& {
   resource = std::make_unique<BigObject>(*other.resource); // Same as copy constructor
   return *this; // The 'this' object gets returned
 }

 // Move assignment operator
 auto operator=(ContainerForBigObject&& other) → ContainerForBigObject& {
   using std::swap; // Search swap in namespace scope, fallback to std-implementation
   swap(resource, other.resource); // Pointers get swapped
   //resource = std::move(other.resource) is possible too, same as move constructor
   return *this;
 }
private:
 std::unique_ptr<BigObject> resource;
};
```

1.2. LVALUE AND RVALUE REFERENCES

Lvalue: Everything that has an identity *(a name)*. The address can be taken. A lvalue reference can be used as function parameter type *(no copy happens, but side-effects on argument are possible)*, Member/local variable or as a return type *(returned object must survive, i.e. a reference parameter or vector.at())*. Beware of dangling references!
Rvalue: Disposable values without an address or a name. Either a literal, a temporary object or an explicitly converted lvalue. If an rvalue is passed to a rvalue reference parameter, it gets a name and turns into an lvalue. See Chapter 2.2. "Perfect forwarding with std::forward".

lvalue references	*rvalue references*
Binds to an lvalue *(everything with a name)*	Binds to an rvalue *(temporary objects, literals)*
T &	T &&
The original must exist as long as it is referred to.	Can extend the life-time of a temporary.

`auto modify(T& t) → void {`
`  // manipulate t`
`}`
`auto lvalueRefExample() → void {`
`  T t = 5;`
`  modify(t); // modify(5) would not`
`  // work because 5 is not a reference`
`  T& ir = t; // rarely used`
`}`
	`auto createT() → T;`
	`// 't' can be "stolen" (not needed anymore)`
	`auto consume(T&& t) → void {`
	`  // manipulate t`
	`}`
	`auto rvalueRefExample() → void {`
	`  consume(T{}); // new temporary T as param`
	`  T&& t = createT(); // rarely used`
	`}`

The value which is the lvalue/rvalue is marked as [value]

lvalue examples	*rvalue examples*
`// Has a name` `T value {}; std::cout << [value];`	`// Temporary object without a name` `int value{}; std::cout << [value + 1];`

`// Function calls returning a lvalue ref`
`[std::cout << 23]; // returns 'std::cout &'`
`[vec.front()]; // returns 'T &'`
	`// Function call returning a value type`
	`std::abs(int n); // returns 'int'`

`// Built-in prefix inc/dec expressions` `++a; // returns 'T &'`	`// Built-in postfix inc/dec expressions` `a++; // returns 'T', the value without +1`

`// lvalue ref, but has a name`
`auto foo(T& param) → void {`
`  std::cout << [param]; }`
	`// Temporary T without a reference`
	`auto create() → T; [create()];`

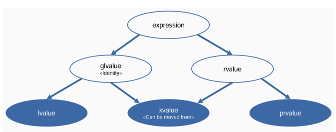
`// rvalue ref, but has a name`
`auto print(T&& param) → void {`
`  std::cout << [param]; }`
	`// Transformation into rvalue with move`
	`T value{}; T o = [std::move(value)];`

`// References have an address` `T& create(); [create()];`	`// xvalue, binds to rvalue references` `T&& create(); [create()];`
`// String literals are always lvalues` `std::cout << ["Hello"];`	**Rule of thumb:** Does element keep living? ✓ lvalue *(only copy)*, ✗ rvalue *(copy & move possible)*

```
// depends on the implementation of +
T value{}; std::cout << [value + 1];
```

1.3. VALUE CATEGORIES

CPPReference: Value categories



A value is always either a lvalue, xvalue or prvalue. Lvalue does not always mean "on the left side of an assignment": `int const a = 0; a = 7; // error` *(left side can't always be assigned)*

Has identity?	*Can be moved from?*	*Value category*
Yes	No	lvalue
Yes	Yes	xvalue *(expiring value)*
No	Yes	prvalue *(pure rvalue)*
No	Yes *(Since C++17)*	– *(does not exist anymore)*

Xvalue: Expiring Value. Address cannot be taken, cannot be used as left-hand operator of built-in assignment. Conversion from prvalue through *temporary materialization*. Conversion from lvalue through `std::move(x)`.
Examples: Function call returning rvalue ref *(i.e. std::move(x))*, access of non-ref members of rvalue object:
`X x1{}, x2{}; consume(std::move(x1)); std::move(x2).member; X{}.member;`

1.3.1. Temporary Materialization

Getting from something imaginary to something you can point to *(a value getting an address)*. Transformation from prvalue to xvalue. Requires a destructor. Happens...
– when *binding a reference to a prvalue (1)*,
– when *accessing a member of a prvalue (2)*,
– when accessing an element of a prvalue array *(int value = (int[]){10, 20}[1]; // value = 20)*,
– when converting a prvalue to a pointer *(const int &&ref = (5+3); const int *ptr = &ref;)*
– when initializing an `std::initializer_list<T>` from a braced-init-list. *(std::vector{1,2,3};)*

```
struct Ghost {
  auto haunt() const → void { std::cout << "booo!\n"; } // ~Ghost() = delete; would error
};
auto evoke() → Ghost { return Ghost{}; }
auto main() → int { Ghost&& sam = evoke(); /* (1) */ Ghost{}.haunt(); /* (2) */ }
```

const lvalue reference: Binds lvalues, xvalues and prvalues: `auto f(T const &) → void`

1.4. OVERLOAD RESOLUTION FOR FREE FUNCTIONS

	f(S)	f(S &)	f(S const &)	f(S &&)
S s{}; f(s);	✓	✓ *(preferred over const&)*	✓	✗
S const s{}; f(s);	✓	✗	✓	✗
f(S{});	✓	✗	✓	✓ *(preferred over const&)*
S s{}; f(std::move(s));	✓	✗	✓	✓ *(preferred over const&)*

The overload for value parameters imposes ambiguities. For deciding two lvalue reference overloads, the const-ness of the argument is considered.

1.5. OVERLOAD RESOLUTION FOR MEMBER FUNCTIONS

	S::m()	S::m() const	S::m() &	S::m() const &	S::m() &&
	Value Members		Reference Members		
S s{}; f(s);	✓	✓	✓ *(preferred over const&)*	✓	✗
S const s{}; s.m();	✗	✓	✗	✓	✗
S{}.m();	✓	✓	✗	✓	✓ *(preferred over const&)*
S s{}; std::move(s).m();	✓	✓	✗	✓	✓ *(preferred over const&)*

Reference and non-reference overloads cannot be mixed. The reference qualifier affects the this object and the overload resolution. const && is theoretically possible, but an artificial case.

1.6. SPECIAL MEMBER FUNCTIONS

CPPReference: The rule of three/five/zero
– **Constructors:** Default Constructor, Copy Constructor, Move Constructor *(Called on variable initialization)*.
– **Assignment Operators:** Copy Assignment, Move Assignment *(Called on variable reassignment)*
– **Destructors** *(Called automatically when variable goes out of scope to clean up the objects resources)*

It is normally not necessary to implement these yourself ("*rule of zero*"), but if a destructor or a copy function is needed, all three are needed ("*rule of three*"). For further optimization, the move functions should also be implemented ("*rule of five*"). Copy Constructor/Assignment should be marked const *(they don't modify this)*.

Assignment operators must be *member functions*. Move operations *must not throw exceptions*, thus aren't allowed to allocate memory. Use the default implementation of the special member functions whenever possible.

1.6.1. Move Constructor S(S &&)

CPPReference: Move constructor
Takes the guts out of the argument and moves them to the constructed object. Leaves the argument in *valid but indeterminate state*. Don't use the argument after it has been moved from until you assign it a new value. **Default behavior:** Initialize base classes and members with move initialization. `S(S && s) : s(std::move(v.m))`

1.6.2. Copy/Move assignment auto operator=(S const &) → S& / auto operator=(S &&) → S&

CPPReference: Copy assignment, CPPReference: Move assignment
Copies/Moves the argument into the this object. Executed when the variable to copy/move to has *already been initialized*. Must be a member operator. **Default behavior:** Initializes base classes and members with copy-assignment/move-assignment.

1.6.3. Destructor ~S()

CPPReference: Destructors
Deallocates resources held by the this object. No parameters. Must not throw *(is noexcept)*. Must be implemented if a custom copy/move is defined. **Default behavior:** Calls destructor of base classes and members.

1.6.4. Copy-Swap-Idiom

Allows efficient implementation of the copy-assignment operator. Utilizes the copy constructor to create a temporary object, and exchanges its contents with itself using a non-throwing swap. Therefore, it *swaps the old data with new data*. The temporary object is then destructed automatically. It needs a copy-constructor, a destructor and a swap function to work. Should be marked noexcept.

```
struct S {
  // swap() member function - called by non-member swap() below
  auto swap(S& other) noexcept → void {
    using std::swap; // Fall back to std::swap if no user defined swap is available
    swap(member1, other.member1); // Calls std::swap or custom swap of member variable
  }
  // Copy assignment
  auto operator=(S const& s) → S& {
    if (std::addressof(s) ≠ this) { // Avoids unnecessary self-copy
      S copy = s; // Can throw exception, this-object stays intact
      swap(copy);
    }
    return *this;
  }
  // Move assignment
  auto operator=(S&& s) → S& { // could be noexcept
    if (std::addressof(s) ≠ this) {
      swap(s); // 's' now contains value of this-object, but is okay because dead soon
    }
    return *this;
  }
}; // Destructor is implicit

auto swap(S& lhs, S& rhs) noexcept → void {
  lhs.swap(rhs); // Calls S::swap() above
}
```

1.6.5. What you write vs. what you get

Watch out for the combinations in red. "defaulted!" are bugs, should be implemented yourself. You want to stay above the black line, avoid going below it! When using multiple constructor / assignment needs, defaulted takes precedence over defaulted!, undeclared over defaulted. Defining a constructor / assignment method as = default; makes the implicit default explicitly user decl.

write/get	*default ctor*	*destructor*	*copy ctor*	*copy assign*	*move ctor*	*move assign*
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any ctor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default ctor	user decl.	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user decl.	defaulted!	defaulted!	undeclared	undeclared
copy ctor	undeclared	defaulted	user decl.	defaulted!	undeclared	undeclared
copy assign	defaulted	defaulted	defaulted!	user decl.	undeclared	undeclared
move ctor	undeclared	defaulted	deleted	deleted	user decl.	undeclared
move assign	defaulted	defaulted	deleted	deleted	undeclared	user decl.

1.7. COPY ELISION

CPPReference: copy elision
In some cases, the compiler is required to elide *(omit)* specific copy/move operations regardless of the side-effects of the corresponding special member functions ("*Mandatory elision*"). The omitted operations need not exist. This happens...
– In *initialization*, when the initializer is a *prvalue*: `S s = S{S{}};` *(Only 1 constructor call, 0 copy operations)*
– When a function call *returns a prvalue*: *(S{} gets initialized directly in new_sw() instead of in create())*

```
auto create() → S { return S{}; }
auto main() → int { S new_sw{create()}; S * sp = new S{create()}; }
```

1.7.1. Further optional elisions

– **NRVO:** *Named Return Value Optimization.* Return type is a value type, return expression is a local variable of the return type. The object is constructed in the location of the return value. The constructors *must still exist* - even if they are elided. `return std::move()` prevents NVRO.
– **throw Expression:** Return expression is a local variable from the innermost surrounding try block. The object is constructed in the location where it would be moved or copied to.
– **catch Clause:** If the caught type is the same as the object thrown, it accesses the object directly.

1.7.2. Example

```
auto create() → S { S s{}; std::cout << "\t --- create() ---\n"; return s; }
auto main() → int {
  std::cout << "\t --- S s{create()} ---\n"; S s{create()};
  std::cout << "\t --- s = create() ---\n"; s = create();
}
```

Disabled elision *(C++14)*	**Only mandatory elision** *(C++17)*	**With constructor elision** *(C++17)*
2x Move in s{}, 1x Move in s =	*1x Move in s{}, 1x Move in s =*	*0x Move in s{}, 0x Move in s =*

`--- S s{create()} ---`
`Constructor S()`
`--- create() ---`
`Constructor S(S&&)`
`Constructor S(S&&)`
`--- s = create() ---`
`Constructor S()`
`--- create() ---`
`Constructor S(S&&)`
`operator =(S&&)`
	`--- S s{create()} ---`
	`Constructor S()`
	`--- create() ---`
	`Constructor S(S&&)`
	`// no create() to s`
	`--- s = create() ---`
	`Constructor S()`
	`--- create() ---`
	`Constructor S(S&&)`
	`operator =(S&&)`
		`--- S s{create()} ---`
		`Constructor S()`
		`--- create() ---`
		`// no return to create()`
		`// no create() to s`
		`--- s = create() ---`
		`Constructor S()`
		`--- create() ---`
		`// no return to create()`
		`operator =(S&&)`

1.8. LIFE-TIME EXTENSION

The life-time of a temporary can be extended by *const lvalue references* or *rvalue references*. Extended life-time ends at the end of the block. It is not transitive *(Reference Return → Dangling Reference → Undefined Behavior)*.

```
struct Demon { /* ... */ };
auto summon() → Demon { return Demon{}; } // Creates a demon
auto countEyes(Demon const&) → void { /* ... */ }

auto main() → int {
  summon(); // Demon dies at the end of the statement
  countEyes(Demon{}); // Demon lives long enough for count_eyes to finish
  // life-time can be extended by const & → flaahgun lives until end of block
  Demon const& flaahgun = summon();
  // life-time can also be extended by && → laznik lives until end of block
  Demon&& laznik = summon();
} // flaahgun and laznik die here
```

2. TYPE DEDUCTION

2.1. FORWARDING REFERENCES AND TYPE DEDUCTION

CPPReference: Template argument deduction, CPPReference: Reference declaration – Forwarding Reference
In some contexts, T&& does not necessarily mean rvalue reference. *Exceptions:* auto && or T&& when template type deduction applies for type T. Here, lvalues can also bind to T&& as a T&. Only works for Method Templates, not Class Templates, and not with "T &&", not "std::vector<T> &&" or "T const &&".

```
template <typename T>          // T and ParamType can be different types!
auto f(ParamType param) → void;     f(<expr>);
```

Deduction of type T depends on the structure of the type of the corresponding parameter ParamType:

2.1.1. ParamType is a value type (T)

*(e.g. auto f(T param) → void) Note: Pointers (T *) are also value types*
1. <expr> is a reference type: ignore the reference
2. Ignore the rightmost const of <expr> *(char const * const → char const *)*
3. Pattern match <expr>'s type against ParamType to figure out T

```
int x = 23; int const cx = x; int const & crx = x; char const * const ptr = ...;

// calls   // instances                          // deduced Ts
f(x);      auto f(int param) → void;             T = int
f(cx);     auto f(int param) → void;             T = int
f(crx);    auto f(int param) → void;             T = int
f(ptr);    auto f(char const * param) → void;    T = char const *
```

2.1.2. ParamType is a reference (T&)

(e.g. auto f(T & param) → void or auto f(T const & param) → void)
1. <expr> is a reference type: ignore the reference
2. Pattern match <expr>'s type against ParamType to figure out T

```
int x = 23; int const cx = x; int const & crx = x;

// calls   // instances                          // deduced Ts
f(x);      auto f(int& param) → void;            T = int
f(cx);     auto f(int const& param) → void;      T = int const
f(crx);    auto f(int const& param) → void;      T = int const
```

Examples for Const References: *auto f(T const & param) → void*

```
// calls   // instances                          // deduced Ts
f(x);      auto f(int const& param) → void;      T = int
f(cx);     auto f(int const& param) → void;      T = int
f(crx);    auto f(int const& param) → void;      T = int
```

2.1.3. ParamType is a forwarding reference (T&&)

(e.g. auto f(T && param) → void)
1. <expr> is an lvalue: T and ParamType become lvalue references *(First 3 examples turn into T&)*
2. Otherwise *(if <expr> is an rvalue)*: Rules for references apply *(Last example)*

```
int x = 23; int const cx = x; int const & crx = x;

// calls    // instances                               // deduced Ts
f(x);       auto f(int & param) → void;                T = int &
f(cx);      auto f(int const & param) → void;          T = int const &
f(crx);     auto f(int const & param) → void;          T = int const &
f(23);      auto f(int && param) → void;               T = int
f("OST");   auto f(char const (&) [4] param) → void;   T = char const (&) [4]
```

2.1.4. Type Deduction and Initializer Lists

When an initializer_list is used for type deduction, an error occurs: `f({23});`
For this to work, a separate template is needed:

```
template <typename T>
auto f(std::initializer_list<T> param) → void;
f({23}); // T = int, ParamType = std::initializer_list<int>
```

2.1.5. Type Deduction for auto

CPPReference: Placeholder type specifiers: auto/decltype(auto)
Same deduction as above. auto takes the place of T:

```
auto x = 23;          // T → int           auto is a value type
auto const cx = x;    // T → int const     auto is a value type
auto& rx = x;         // T& → int &        auto is a reference type

auto&& uref1 = x;     // T&&: x → int (lvalue)         → uref1 = int&
auto&& uref2 = cx;    // T&&: cx → int const (lvalue)  → uref2 = int const&
auto&& uref3 = 23;    // T&&: 23 → int&& (rvalue)      → uref3 = int&&

// Special cases
auto init_list1 = {23};     // std::initializer_list<int>
auto init_list2{23};        // int, was std_initializer_list<int> before C++17
auto init_list3{23, 23};    // Error, requires one single argument
```

auto Return Type Deduction: auto can be used as return type and for parameter declarations. Body must be available to deduce the type. Multiple auto parameters are considered different types.

2.1.6. Type Deduction for decltype

CPPReference: decltype specifier
Represents the declared type of a named expression. decltype(auto) allows deduction of (inline) function return types, but *does not strip references (&) or const like plain auto*. Can take an expression for specifying trailing return types.

```
int             x        = 23;
int&            rx       = x;
decltype(rx)    rx_too   = rx;   // rx_too → int&
auto            just_x   = rx;   // just_x → int
decltype(auto)  more_rx  = rx;   // more_rx → int&
template <typename Container, typename Index>
auto access(Container & c, Index i) → decltype(auto) { return c[i]; } // Requires body
auto access(Container &c, Index i) → decltype(c[i]) { ... } // Body not required (.hpp)
```

2.1.7. Type Deduction in Lambdas

Slides: Page 38

2.2. PERFECT FORWARDING WITH STD::FORWARD

CPPReference: std::forward, Slides Page 25 onward #include <utility>

```
template <typename T>
decltype(auto) forward(std::remove_reference_t<T>& param) {
  return static_cast<T&&>(param);
}
```

std::forward is a *conditional cast to an rvalue reference*. This allows arguments to be treated as what they originally were: lvalues remain lvalues and rvalues remain rvalues.
– If T is of *value type*, T && is an *rvalue reference* in the return expression. *(int → int&&)*
– If T is of *lvalue reference type*, the resulting type is an *lvalue reference to an lvalue reference* *(e.g. T = int & → T && would mean "int & &&" which can be collapsed into "int &").*

```
template<typename T>
static auto make_buffer(T && value) → BoundedBuffer<value_type> {
  BoundedBuffer<value_type> new_buffer{};
  new_buffer.push(std::forward<T>(value));
  return new_buffer; }
```

2.3. STD::MOVE

CPPReference: std::move, Slides Page 43 onward #include <utility>

```
template <typename T>
decltype(auto) move(T&& param) {
  return static_cast<std::remove_reference_t<T>&&>(param);
}
```

Does not actually move objects. It's just a *unconditional cast to an rvalue reference*. This allows resolution of rvalue reference overloads *(auto f(T&&))* and move-constructor/-assignment operators. *Caution! Moving a const object or a non-const object without a move ctor/assignment results in a copy, not move operation!* To allow for moving, there should be no const member in a class.

Reference Collapsing: "T& &", "T& &&" and "T&& &" become "T&", "T&& &&" becomes "T&&".

3. HEAP MEMORY MANAGEMENT

Lifetime on Stack: Deterministic, local variables get deleted automatically upon leaving their scope `{ int a; }`
Lifetime on Heap: Creation and deletion happens *explicitly* with new and delete *(Dangerous, avoid!)*.
`auto foo() → void { auto ip = new int{5}; /* ... */ delete ip; }`
Rules: Delete every object you allocated, do not delete an object twice or access a deleted object.

3.1. EXPLICIT LIFE-TIME MANAGEMENT

Global and local variables have life-time implicitly managed by the program flow. Some resources can be allocated and deallocated explicitly. This is *error-prone*. **Guideline:** Always wrap explicit resource management in an object which has its implicit life-time management (RAII).

3.2. POINTER SYNTAX

CPPReference: Pointer declaration

```
auto ip = new int{5};   // auto → int *
int v = * ip;           // accessing value: v == 5
```

Heap Array Access:

```
auto arr = new int[5]{};  // 5 needs to be compile-time-constant, auto → int *
int v = arr[4];           // accessing element (*arr + 4 pointer objects)
```

Direct Member Access (->):

```
struct S {
  auto member() → void { this→value = ...; } int value; // 'this': pointer to a instance
};
auto foo() → void {
  S* sp = new S{}; sp→member(); // = (*sp).member(); "." binds stronger than "*"
}
```

Pointer Parameters: Pointers can be used as parameters. Addresses can be taken with & *(Caution, operator could be overridden)* or std::addressof() *(Preferred, the rvalue overload is deleted to prevent taking its address)*.

```
auto foo(int* p) → void { }
auto bar() → void { int* ip = new int{5}; int local = 6; foo(ip); foo(&local); }
```

More complicated examples with pointer to pointer to int: Slides page 25

Const Pointers: const Pointer can't be modified, but the object behind it may. const is on the right side of the *. The declaration is read from right to left.
`int const * const * const icpcpc = &icpcp;`
"icpcpc is a *const pointer* to a *pointer* to a *const pointer* to a *const int*"

nullptr: Represents a null-Pointer. Is a *literal (prvalue)* and has type nullptr_t. Implicit conversion to any pointer type: T *. Prefer nullptr over 0 and NULL *(no overload ambiguity, no implicit conversion)*.

Pointer vs. Reference

Pointers ...	*References ...*
can be nullptr	are always bound to an object
can be changed *(if not const)*	cannot be rebound to another object
require dereferencing with "*" or "->"	allow member access by "."
can be dangling *(pointing to a destroyed object)*	can be dangling *(referencing a destroyed object)*

Use raw pointers only to explicitly *model the possibility of a nullptr* (reference is a check like `auto foo(int* ptr) → void { if (ptr) { ... } }`) and for *modelling borrowing* only. Else, use *smart pointers*.

3.3. MEMORY ALLOCATION WITH NEW

`new <type> <initializer>`

Allocates memory for an instance of <type>. Returns a pointer to the object or array created on the heap of type <type> *. The arguments in the <initializer> are passed to the constructor of <type>. *Memory Leak if not removed with delete*. Avoid manual allocation, use RAII instead.

```
struct Point{ Point(int x, int y) : x{x}, y{y}{} int x, y; };
auto createPoint(int x, int y) → Point* {
  return new Point{x, y}; // constructor
}
auto createCorners(int x, int y) → Point* {
  return new Point[2]{{0, 0}, {x, y}};
}
```

3.3.1. Placement new

`new (<location>) <type> <initializer>`

Used for placing elements on the heap in the location of a deleted element. Does *not* allocate new memory *(Undefined Behavior!)*. The memory of <location> needs to be suitable for construction of a new object and any element there *must be destroyed before*. Calls the Constructor for creating the object at the given location and returns the memory location. Better use std::construct_at().

```
auto ptr = new Point{9, 8};
// deconstruct Point here...
new (ptr) Point{7, 6};
delete ptr;

// Better:
std::construct_at(ptr, 7, 6);
```

3.4. MEMORY DEALLOCATION WITH DELETE

`delete <pointer>`

Deallocates the memory of a single object pointed to by the <pointer>. Calls the Destructor of the destroyed type. delete nullptr does nothing. *Deleting the same object twice is Undefined Behavior!*

```
struct Point{ Point(int x, int y) : x{x}, y{y}{} int x, y; };
auto funWithPoint(int x, int y) → void {
  Point * pp = new Point{x,y};
  delete pp; // calls destructor and releases memory
}
```

3.4.1. Placement delete

Does not exist, but a destructor can be called explicitly.

`S * ptr = ...; ptr->~S();`

Destroys the object, but does *not* free its memory. Called like any other member function. Better use std::destroy_at(ptr). Use this for non-default-constructible types.

3.4.2. Array Memory Deallocation with delete[]

`delete[] <pointer-to-array>`

Deallocates the memory of an array pointed to by the <pointer-to-array>. Calls the Destructor of the destroyed objects. Also deletes *multidimensional arrays*. Not necessary to know exact amount of elements in the array. Undefined behavior if non-array is deleted with delete[].
`Point *arr = new Point[2]{{0, 0}, {3, 2}}; /* do stuff */ delete[] arr;`

3.5. NON-DEFAULT CONSTRUCTIBLE TYPES

A type is non-default-constructible when there is *no explicit or implicit default constructor*.
To create arrays of NDC types, allocate the plain memory and initialize it later *(Slides page 36 onward)*.

```
auto memory = std::make_unique<std::byte[]>(sizeof(Point) * 2 /* Array size */);
auto location = reinterpret_cast<Point*>(memory.get()); // Address of first element
std::construct_at(location, 1, 2); // Equivalent to arr[0] = Point{1, 2}
auto value = elementAt(memory.get(), 0); // Access value via helper function
std::destroy_at(location);

auto elementAt(std::byte* memory, size_t index) → Point& { // helper function
  return reinterpret_cast<Point*>(memory)[index];
}
```

Don't use an element if it is uninitialized and destroy them before the memory is deallocated.

Use a std::byte array as memory for NDC Elements.
– **Static:** `std::array<std::byte, no_of_bytes> values_memory;` *(on stack, size known at compile-time)*
– **Dynamic:** `std::unique_ptr<std::byte[]> values_memory;` *(on heap, size known at run-time)*

3.6. CLASS-SPECIFIC OVERLOADING OF OPERATOR NEW/DELETE

Overloading new and delete for a class can inhibit heap allocation. This can be used to provide efficient allocation. Is useful with a memory pool for small instances or if thread-local pools are used. Can log or limit number of heap-allocated instances. *But in general, not advisable.*

```
struct not_on_heap { // Prevents heap allocation of this class
  static auto operator new(std::size_t sz) → void * { throw std::bad_alloc{}; }
  static auto operator new[](std::size_t sz) → void * { throw std::bad_alloc{}; }
  static auto operator delete(void *ptr) → void noexcept { /* do nothing */ }
  static auto operator delete[](void *ptr) → void noexcept { /* do nothing */ }
};
```

3.7. READING DECLARATIONS

– Declarations are read *starting by the declarator* (name)
– First *read to the right* until a *closing parenthesis* is encountered
– Second *read to the left* until an *opening parenthesis* is encountered
– Third *jump out of the parentheses* and start over

Specifiers right to the declarator: Array Declarator ([]) and Function Parameter List ((<parameter declarations>))
Specifiers left to the declarator: References (&&, &), Pointers (*) and Types (int)
const: Applies to its left neighbor, if there is no left neighbor, it applies to its right neighbor.
`const int i; int const i;` Should always be written to the right of the type to avoid surprises *(i.e. with aliases)*.
`void (* f)(int &, double)`
f is a pointer to a function that takes a reference to int and a double, returning void.
`int const * (* f[2][3]) [5];`
f is an *array of 2 elements* of *arrays of 3 elements of pointers* to *arrays of 5 elements of pointers* to *const int*.
`int (* f(int(*)(int))) (int);`
f [1] is a function that takes a pointer to a function [2] and returns a pointer to a function [3]. The function in the argument takes an int as argument [4] and returns an int [5]. The returned function takes an int as argument [6] and returns an int [7]. Always use type aliases on cases like this!

3.8. RESOURCE MANAGEMENT WITH RAII

Resource Acquisition Is Initialization is an alternative to allocating and deallocating a resource explicitly. Wraps allocation and deallocation in a class, uses regular constructor/destructor. Cleaned up at end of scope.

3.8.1. std::unique_ptr and std::make_unique

CPPReference: std::unique_ptr, CPPReference: std::make_unique
`std::unique_ptr<char> cPtr = std::make_unique<char>('*');`
Wraps a plain pointer, has zero runtime overhead. A custom deleter could be supplied if required. Always use make_unique for creation. Can create unbound arrays, but not fixed size arrays.

3.8.2. Container Member Function emplace

CPPReference: std::vector<T,Allocator>::emplace, CPPReference: std::stack<T,Container>::emplace
Constructs elements directly in a container, more efficient than moving them. Not available for std::array.
`std::stack<Point> vec{}; vec.emplace(3, 5); // std::vector requires position argument`

4. ITERATORS & TAGS

4.1. TAGS FOR DISPATCHING

CPPReference: std::...iterator_tag
If the *same operation* can be implemented more/less efficiently depending on the capabilities of the argument, *tags can be used to find the "best" implementation*.
Tags are used to *mark capabilities* of associated types. They do not contain any members.

```
// provides travelThroughSpace(), the base functionality
struct SpaceDriveTag{};
template<typename> struct SpaceshipTraits { using Drive = SpaceDriveTag; };
// provides travelThroughSpace() and travelThroughHyperspace(), specialized template
struct SubspaceDriveTag : SpaceDriveTag{};
template<> struct SpaceshipTraits<GalaxyClassShip> { using Drive = SubspaceDriveTag; };

template <typename Spaceship>
auto travelToDispatched(Galaxy destination, Spaceship& ship, SpaceDriveTag) → void {
  ship.travelThroughSpace(destination); // for Spaceships with SpaceDriveTag
}
template <typename Spaceship>
auto travelToDispatched(Galaxy destination, Spaceship& ship, SubspaceDriveTag) → void {
  ship.travelThroughHyperspace(destination); // for Spaceships with SubspaceTag
}
template <typename Spaceship>
auto travelTo(Galaxy destination, Spaceship& ship) → void {
  typename SpaceshipTraits<Spaceship>::Drive drive{}; // get the Spaceship's drive tag
  travelToDispatched(destination, ship, drive); // call overloaded function
}
```

4.2. ITERATORS

Different algorithms require different strengths of iterators. Iterators capabilities can be described at compile time with tag types.
– **OutputIterator:** Write results *(to console, file etc.)*, without specifying an end *(used on std::ostream)*.
  operator * returns an *lvalue reference* for assignment of the value.
– **InputIterator:** Read sequence once *(used on std::istream)*.
  operator * returns *const lvalue reference*, or *rvalue*.
– **ForwardIterator:** Read/write forward sequence, multi-pass *(used on std::forward_list linked-list)*.
  const operator * returns const lvalue reference or rvalue, non-const operator * returns lvalue
– **BidirectionalIterator:** Read/write, forward and backward *(used on std::list double-linked-list)*.
– **RandomAccessIterator:** Read/write/indexed sequence *(used on std::vector)*.

You need to implement the members required by your iterator_tag.

```
struct IntIterator { // Provide these member types to align with STL iterators
  using iterator_category = std::input_iterator_tag; // iterator category
  using value_type = int; // type of element the iterator iterates over
  using difference_type = ptrdiff_t; // specifies iterator distance
  using pointer = int *; // pointer type of the elements iterated over
  using reference = int &; // reference type of the elements iterated over
};
```

4.2.1. iterator_traits<>

CPPReference: std::iterator_traits
STL algorithms often want to *determine the type* of some specific thing related to an iterator. However, not all iterator types are actually classes. Default iterator_traits just pick the type aliases from those provided.

```
template <typename InputIterator, typename Tp>
auto count(InputIterator first, InputIterator last, const Tp& value)
  → typename iterator_traits<InputIterator>::difference_type { ... }
```

Specialization of iterator_traits also allows "naked pointers" to be used as iterators in algorithms.

```
template <typename _Tp> struct iterator_traits<Tp*> { /* Provide regular iter usings */ }
```

4.2.2. Problems with the Stream Input Iterator

Reference Member and Default Constructor
When implementing a input iterator, we need to be able to create an EOF iterator. This dirty hack works, but the global variable to initialize the reference in an anonymous namespace is bad for multi-threading.

```
namespace { // global variable to initialize the reference in an empty namespace
  std::istringstream empty{}; // pseudo default
}
IntInputter::IntInputter() : input { empty } {
  // guarantee the empty stream is !good()
  input.clear(std::ios_base::eofbit); // mark empty stream as EOF
}
```

Dereferencing and Equality

```
// Dereferencing with * reads the value from the input
auto IntInputter::operator*() → IntInputter::value_type {
  value_type value{};
  input >> value;
  return value; }
// Stream iter comparisons only make sense for testing if they can still be read from
auto IntInputter::operator==(const IntInputter & other) const → bool {
  return !input.good() && !other.input.good(); }
```

4.2.3. Custom Iterator Example (Testat 2)

```
template<typename V>
struct iterator_base {
  using value_type = BoundedBuffer::value_type;
  using reference = BoundedBuffer::reference;
  using const_reference = BoundedBuffer::const_reference;
  using size_type = BoundedBuffer::size_type;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::random_access_iterator_tag;

  auto operator==(iterator_base const & other) const -> bool { ... }
  auto operator<=>(iterator_base const & other) const -> std::strong_ordering { ... }

  auto operator*() const -> decltype(auto) { return Buffer->elementAt(Index); }
  auto operator->() const -> decltype(auto) { return &Buffer->elementAt(Index); }
  auto operator[](difference_type index) const -> decltype(auto) { ... }

  auto operator++() -> iterator_base & { Index++; return *this; }
  auto operator++(int) -> iterator_base {
    auto const copy = *this;
    ++(*this);
    return copy; }

  auto operator--() -> iterator_base & { Index--; return *this; }
  auto operator--(int) -> iterator_base {
    auto const copy = *this;
    --(*this);
    return copy; }

  auto operator+(difference_type n) const -> iterator_base {
    auto copy = *this;
    copy += n;
    return copy; }

  auto operator-(difference_type n) const -> iterator_base {
    return this->operator+(-n); }

  auto operator+=(difference_type n) -> iterator_base & { Index += n; return *this; }
  auto operator-=(difference_type n) -> iterator_base & { return this->operator+=(-n); }

  auto operator-(iterator_base const & other) const -> difference_type { ... }
private: difference_type Index{}; BoundedBuffer* Buffer;
};

using iterator = iterator_base<Container<T>>;
using const_iterator = iterator_base<Container<T> const>;
```

Boost would generate operator++(int), operator--(int), operator+(difference_type n), operator-(difference_type n) with implementation shown above. Change signature to `struct iterator_base : boost::operators_impl::random_access_iterator_helper<iterator_base<V>, V> {}`

5. ADVANCED TEMPLATES

Pros of static polymorphism	**Cons of static polymorphism**
– Happens at compile-time	– Longer compile-times
– Faster execution time *(no dynamic dispatch required, easier to optimize)*	– Template code has to be known when used *(body needs to be in HPP file)*
– Type checks at compile-time	– Larger binary size *(copy of the used parts for each template instance)*

A polymorphic call of a *virtual function* *(inheritance overloading)* requires lookup of the target function. *Non-virtual calls* *(template overloading)* directly call the target function. This is *more efficient*.

5.1. SFINAE (SUBSTITUTION FAILURE IS NOT AN ERROR)

CPPReference: SFINAE
Is used to eliminate overload candidates by substituting return type and parameters. During overload resolution the template parameters in a template declaration *(i.e. T)* are substituted with the deduced types *(i.e. int)*. This may result in template instances that cannot be compiled *(i.e. calling a member function on a value type)*. If the substitution of template parameter fails, that overload candidate is *discarded*.
decltype(<return-expr>) as return type checks if the overload candidate would work, but this approach is infeasible, because functions can be void and it's not elegant for complex bodies.

5.1.1. Type traits

CPPReference: Type Traits, CPPReference: std::integral_constant #include <type_traits>
The standard library provides many predefined checks for type traits. A trait contains a boolean value. Usually available in a _v *(returns the bool result)* and non-_v variant *(returns the integral constant)*.

Example: std::is_class
CPPReference: std::is_class

```
std::is_class<S>::value;    std::is_class_v<S>;    // both true
std::is_class<int>::value;  std::is_class_v<int>;  // both false
```

std::enable_if / std::enable_if_t
CPPReference: std::enable_if
std::enable_if_t takes an expression and a type. If the expression evaluates to true, std::enable_if_t represents the given type, otherwise, it does *not* represent a type.

```
auto main() → int {
  std::enable_if_t<true, int> i; // int
  std::enable_if_t<false, int> error; } // no type, compiler error
```

std::enable_if can be applied at different places *(marked with "[]", only one needs to be used)*:

```
template <typename T, [typename = enable_if_t<is_class_v<T>, void>]>
auto increment([enable_if_t<is_class_v<T>, void> value]) // impairs type deduction
  → [enable_if_t<is_class_v<T>, T>] {
  return value.increment(); }
```

5.2. TEMPLATE PARAMETER CONSTRAINTS AND CONCEPTS

CPPReference: Constraints and Concepts
Provide a means to *specify* the *characteristics of a type* in template context. Better error messages, more expressive SFINAE.

5.2.1. Keyword requires

Allow constraining template parameters. requires is followed by a compile-time constant boolean expression. Is either placed after the template parameter list or after the function template's declarator.

```
template <typename T>
requires std::is_class_v<T> // either here...
auto function(T argument) → void requires std::is_class_v<T> /* or here */ { ... }
```

requires Expression
requires also starts an expression that evaluates to bool depending whether they can be compiled.

```
requires ($parameter-list$) { /* sequence of requirements */ }
```

```
template <typename T>
requires requires (T const v) { v.increment(); } // compiles if v has a increment function
auto increment(T value) → T { return value.increment(); }
```

Type Requirements
Check whether a type exists. Starts with typename keyword. Useful for nested types like in Bounded Buffer

```
requires { typename $type$ }
```

```
requires { typename BoundedBuffer<int>::value_type; }
```

Compound Requirements
Check whether an expression is valid and can check constraints on the expression's type. The return-type-requirement is optional. Needs to be a valid type constraint, regular types can't be used.

```
requires (T v) { { $expression$ } → $type-constraint$; }
```

```
template <typename T> // We can't use T as return type in requires, it is not a constraint
requires requires (T const v) { v.increment() } → std::same_as<T>; }
auto increment(T value) → T { return value.increment(); }
```

5.2.2. Keyword concept

Specifies a named type requirement. Conjunctions (&&) and disjunctions (||) can be used to combine constraints *(i.e. requires std::integral<T> || std::floating_point<T>)*.

```
template <typename T>
concept Incrementable = requires (T const v) { { v.increment() } → std::same_as<T> };
```

Named requirements can then be used in template parameter declarations or as parts of a requires clause.

```
template <Incrementable T> // either here...
requires Incrementable<T> // ...or here
auto increment(T value) → T { return value.increment(); }
```

5.2.3. Abbreviated Function Templates

auto can be used as parameter type instead of a template declaration.

```
template <Incrementable T>
auto increment(T value) → T { return value.increment(); }
// is equivalent to Terse Syntax
auto increment(Incrementable auto value) → T { return value.increment(); }
```

If there are two auto arguments, two template typenames T1, T2 get created.

5.2.4. STL Concepts

CPPReference: Concepts library #include <concepts>
The STL has predefined constraints: **std::equality_comparable** *(can type be ==/≠ compared?)* **std::default_initializable** *(can type be default constructed?)* **std::floating_point** *(is type a floating point?)*

6. COMPILE-TIME COMPUTATION

6.1. CONSTANT EXPRESSION CONTEXT

These expressions always need to be defined at compile-time.
– Non-type template arguments *(std::array<Element, 5> arr{};)*
– Array bounds *(double matrix[ROWS][COLS]{};)*
– Case expressions *(switch(value) { case 42: /* ... */ })*
– Enumerator initializers *(enum Light { Off = 0, On = 1 };)*
– Static Assertions *(static_assert(order == 66);)*
– constexpr / constinit variables *(constexpr unsigned pi = 3;)*
– constexpr if-statements *(if constexpr (size > 0) { })*
– noexcept expressions *(f(bool b) noexcept(true);)*

6.2. CONSTEXPR / CONSTINIT

static const variables in namespace scope of built-in types initialized with a constant expression are usually put into ROMable memory, if at all. Allowed in constant expression context. No complicated computations, no guarantee to be done at compile-time.
constexpr / constinit Variables are *evaluated at compile-time*. They are initialized by a constant expression and *require a literal type* *(primitive data type without heap allocation)*. They can be used in *constant expression contexts*. Possible contexts are local scope, namespace scope and static data members.
– *constexpr* variables are const, read only at run-time
– *constinit* variables are non-const. They need to be initialized at compile-time, but can be changed at run-time.

6.2.1. constexpr Functions

– Can have *local variables of literal type*. The variables must be initialized before usage.
– Can use loops, recursion, arrays, references, branches
– Can contain branches that rely on *run-time only features*, if branch is not executed during compile-time computations *(e.g. throw. Compilation error if throw is reached while compiling.)*
– Can *only call constexpr functions*.
– Are usable in constexpr and non-constexpr contexts *(during runtime)*
– Can allocate dynamic memory that is cleaned up by the end of the compilation *(since C++20)*
– Can be virtual member functions *(since C++20)*

```
constexpr auto factorial(unsigned n) → unsigned { /* needs to have a body */ }
```

6.2.2. consteval Functions

Are usable in constexpr contexts only *(can't be called during runtime)* and implicitly const.

```
consteval auto factorial(unsigned n) {
  // ...
  return result;
}
constexpr auto factorialOf5 = factorial(5); // works

auto main() → int {
  static_assert(factorialOf5 == 120); // works
  unsigned input();
  if (std::cin >> input) {
    std::cout << factorial(input); // error, function cannot be used at runtime
} }
```

6.2.3. Undefined Behavior

There is *no undefined behavior during compile-time*. Instead, there will be a compilation error. If constexpr evaluation does not reach an invalid statement, the code is still valid.

6.3. LITERAL TYPES

CPPReference: Named Requirements - LiteralType
Literal types are built-in scalar types *(like int, double, pointers, enums)*, Structs with some restrictions *(must have constexpr destructor and constexpr / consteval constructor)*, Lambdas, References, Arrays of literal types and void *(for functions with side effects on literals)*.
Literal Types can be used in constexpr functions, but only constexpr member functions can be called on values of literal type. Non-const member functions can modify the object.

6.3.1. Literal Class Type Example

```
template <typename T> // Can be a template
class Vector {
  constexpr static size_t dimensions = 3;
  std::array<T, dimensions> values{};
public:
  constexpr Vector(T x, T y, T z) : values{x, y, z} {} // constexpr constructor
  constexpr auto length() const → T { // constexpr const member function
    auto squares = x() * x() + y() * y() + z() * z();
    return std::sqrt(squares);
  }
  constexpr auto x() → T& { return values[0]; } // constexpr non-const member function
  constexpr auto x() const → T const& { return values[0]; }
}; // implicit default destructor is also constexpr

constexpr auto create() → Vector<double> {
  Vector<double> v{1.0, 1.0, 1.0}; v.x() = 2.0; return v; }
constexpr auto v = create();
auto main() → int {
  //v.x() = 1.0; // possible if v was constinit
  auto v2 = create(); v2.x() = 2.0; } // v2 is a regular variable, can be modified
```

6.3.2. Compile-Time Computation with Variable Templates

(for Class Template see slides page 25)
Variable templates can be constexpr/constinit and defined recursively. *Usage:* Template-ID

```
template <size_t N>
constexpr size_t factorial = factorial<N-1> * N;

template <> // Base case
constexpr size_t factorial<0> = 1;
```

6.3.3. Captures as Literal Types

Capture types *(the types returned by lambda expressions)* are literal types as well. The can be used as types of constexpr variables and in constexpr functions.

6.4. USER-DEFINED LITERALS
CPPReference: User-defined literals

```
operator"" _UDLSuffix()"
```

Allows integer, floating-point, character, and string literals to produce objects of user-defined type by defining a user-defined suffix. The suffix must start with an underscore. It allows to add dimension, conversion, etc. If possible, define UDL operator functions as `constexpr`. Usually a *conversion function* like `safeToDouble()` and a *conversion operator* is needed. *Rule:* put overloaded UDL operators that belong together in a separate namespace.

```cpp
namespace velocity::literals {
constexpr inline auto operator"" _kph(unsigned long long value) → Speed<Kph>
  { return Speed<Kph>{safeToDouble(value)}; } // user defined literal operator, can be
                                             // called with Speed s = 80_kph;
template<typename T>
concept arithmetic = std::is_arithmetic_v<T>; // allows ints & floats in + operator

static constexpr inline auto safeToDouble(long double value) → double {
  if (value > std::numeric_limits<double>::max()
   || value < std::numeric_limits<double>::min()) {
      throw std::invalid_argument{"Value must be within double range"};
  }
  return static_cast<double>(value);
}
} // namespace velocity::literals

struct Speed {
  constexpr explicit Speed(double value) : value{value} {};
  constexpr explicit operator double() const { return value; } // conversion to double
  auto constexpr operator +(arithmetic auto rhs) → decltype(rhs) { return value + rhs }
private:
  double value{};
}
auto constexpr operator +(arithmetic auto lhs, Speed rhs) → decltype(lhs)
  { return rhs + lhs; } // Make + operator commutative (value + Speed calls Speed + value)
```

Signatures: `unsigned long long` for integral constants, `long double` for floating point constants, `(char const *, size_t len)` → `std::string` for string literals, `char const *` for a raw UDL operator *(i.e. to convert ints and floats to strings, is not constexpr)*.

6.4.1. Template UDL Operator

```
template <char...>
auto operator""_suffix() → TYPE
```

Has an empty parameter list and a variadic template parameter. Characters of the literal are template arguments. Often used for interpreting individual characters. Since C++20, the template UDL operator works with string literals as well *(example and compile-time steps in slides on page 42 - 44)*.

Standard Literal Suffixes: Do not have leading underscore: `std::string` *(s)*, `std::complex` *(i, il, if)*, `std::chrono::duration` *(ns, us, ms, s, min, h)*

6.5. PREPROCESSOR
`hello.cpp` → `preprocessor` → `hello.i` → `compiler` → `hello.o` → `linker` → `hello` *(executable program)*

Object-like Macros

```
#define identifier replacement-list new-line
```

Identifier is a unique name, by convention in ALL_CAPS. Is valid until `#undef NAME`. *Replacement-list* is a possible empty sequence of preprocessor tokens. *New-line* terminates the replacement list.
Example: `#define NUMBER_OF_ROWS 5`

Function-like Macros

```
#define identifier ( identifier-list?, ?...? ) replacement-list new-line
```

Features an optional parameter list, containing only names. Params with a #-prefix turn into string literals.

Includes: Textual inclusion of another file. `#include "path"` for including a header file from the same project or workspace, `#include <path>` for external includes.
Conditional Includes: Enable a section depending on a condition. *(example and macros in slides on page 52 - 61)*.

```
#if constant-expression new-line        #ifdef identifer new-line
#elif constant-expression new-line      #ifndef identifier new-line
#else new-line
#endif new-line
```

7. MULTI-THREADING & MUTEX
`std::thread` to explicitly run a new thread, *`std::async`* to easily wrap a computation *(possibly with a result)*, *`std::mutex`* and co. to facilitate synchronization, *pthreads* is legacy. No portability guarantee.

7.1. API OF STD::THREAD
CPPReference: std::thread `#include <thread>`

```cpp
auto main() → int { std::thread greeter { [] { /*lambda*/ } }; greeter.join(); }
```

A new thread is created and started automatically. Creates a new execution context. `join()` waits for the thread to finish. Besides *lambdas*, *functions* or *functor objects* can also be executed in a thread. The *return value* of the function is *ignored*. Threads are *default-constructible* and *moveable*. *Caution:* Program terminates if thread gets destructed without calling `join()` before!

```cpp
struct Functor {
  auto operator()() const → void { std::cout << "Functor" << std::endl; }
};
auto function() → void { std::cout << "Function" << std::endl; }
auto main() → int {
  std::thread functionThread{function};
  std::thread functorThread{Functor{}};
  functorThread.join(); functionThread.join(); }
```

Streams: Using global streams does not create data races, but sequencing of characters could be mixed.

`std::this_thread` **helpers:** *`get_id()`* *(An ID of the underlying OS thread)*, *`sleep_for(duration)`*/*`sleep_until(time_point)`* *(Suspends thread for/until time)*, *`yield()`* *(Allows OS to schedule another thread)*.

7.1.1. Passing arguments to std::thread

```
template<class Function, class... Args>
explicit thread(Function&& f, Args&&... args);
```

The *`std::thread` constructor* takes a function/functor/lambda and arguments to forward. You should pass all arguments *by value* to avoid data races and dangling references. *Capturing by reference* in lambdas creates shared data as well *(if you have to use them, don't declare them as mutable)*!

```cpp
auto fibonacci(std::size_t n) → std::size_t { ... }
auto printFib(std::size_t n) → void { auto fib = fibonacci(n); /* print ... */ }
auto main() → int { std::thread function { printFib, 46 }; function.join(); }
```

Before the `std::thread` object is destroyed, you must *`join()`* *(wait until finished)* or *`detach()`* *(detach from the main thread and run in the background)* the thread, otherwise you get a runtime error.

7.2. STD::JTHREAD
CPPReference: std::jthread `#include <thread>`

RAII wrapper that automatically calls `join()`. Also supports external stop requests *(Similar to Cancellation Token in C#. `t.request_stop()` sends the request, with `stop_requested()` it can be checked if a stop request has been received)*.

7.3. INTER-THREAD COMMUNICATION
CPPReference: std::mutex, CPPReference: std::shared_mutex `#include <mutex>`

Communication happens with *mutable shared state*. *Problem:* Data Race. *Solution:* Locking the shared access or make access atomic.

All *mutexes* provide the following operations:
- **Acquire:** `lock()` - blocking, `try_lock()` - non-blocking
- **Release:** `unlock()` - non-blocking

Two properties specify the capabilities:
- **Recursive:** Allow multiple nested acquire operations of the same thread *(prevents self-deadlock)*
- **Timed:** Also provide timed acquire operations *(`try_lock_for(duration)`, `try_lock_until(time)`)*

Reading operations don't need exclusive access. Only concurrent writes need exclusive locking. Use `std::shared_mutex`/`std::shared_timed_mutex` with `lock_shared()` to *read with multiple threads*. Shared mutex also have the exclusive lock methods from a regular mutex.

7.3.1. Acquiring / Releasing Mutexes
CPPReference: std::lock_guard, CPPReference: std::scoped_lock, CPPReference: std::unique_lock `#include <mutex>`
CPPReference: std::shared_lock, CPPReference: std::condition_variable `#include <shared_mutex>`

Usually you use a lock that manages the mutex:
- *`std::lock_guard`:* RAII wrapper for a single mutex. Locks immediately when constructed, unlocks when destructed.
- *`std::scoped_lock`:* RAII wrapper for multiple mutexes. Locks immediately when constructed, unlocks when destructed. Acquires multiple locks in the scope of construction, avoids deadlocks by relying on internal sequence. Blocks until all locks could be acquired. *(`std::scoped_lock both{mx, other.mx};`)*
- *`std::unique_lock`:* Mutex wrapper that allows deferred and timed locking. Similar interface to timed mutex, allows explicit locking/unlocking. Unlocks when destructed.
- *`std::shared_lock`:* Wrapper for shared mutexes. Allows explicit locking/unlocking, unlocks when destructed.
- *`std::condition_variable`:* Condition. Waits for the condition and then notifies a potential change. Wait will go into sleep again if the actual condition has not been met yet. *(`wait(<mutex>, <predicate>)`, `wait_for()`, `wait_until()`, `notify_one()`, `notify_all()`)*

Standard Containers and Concurrency: There is no thread-safety wrapper for standard containers. Access to different individual elements from different threads is *not* a data race. Almost all other concurrent uses of containers are *dangerous*. `shared_ptr` copies to the same object can be used from different threads, but accessing the object itself can race if non-const *(reference counter is atomic)*.

7.3.2. Thread-safe Guard Example (Testat 3)
Scoped Lock Pattern: Create a lock guard that (un)locks the mutex automatically. Every member function is mutually exclusive because of scoped locking pattern. *Strategized Lock Pattern:* Template Parameter for mutex type.

```cpp
template <typename T,
          typename MUTEX = std::mutex, typename CONDITION = std::condition_variable>
struct threadsafe_queue {
  using guard = std::lock_guard<MUTEX>;
  using lock = std::unique_lock<MUTEX>;
  template <typename ForwardType>
  auto push(ForwardType &&t) → void {
    guard lk{mx}; q.push(std::forward<ForwardType>(t)); not_empty.notify_one();
  }
  auto pop() → T {
    lock lk{mx}; // wait requires timed locking therefore unique_lock
    not_empty.wait(lk, [this]{ return !q.empty(); }); // checked once, no busy wait
    T t = q.front();
    q.pop();
    return t;
  }
  auto try_pop(T & t) → bool {
    guard lk{mx}; if (q.empty()) { return false; } t = q.front(); q.pop(); return true;
  }
  // call container empty, not this→empty, would cause deadlock
  auto empty() const → bool { guard lk{mx}; return q.empty(); }
private:
  std::queue<T> q{};
  mutable MUTEX mx{}; // mutable to unlock in const member functions
  CONDITION not_empty{};
}; // Mutex & condition_variable don't need to be swap()-ed. But notify them of changes!
```

7.4. RETURNING RESULTS FROM THREADS
We can use shared state to "return" results. *Acquire lock* in producer, *write* the shared result, *wait* for the result, *read* the result. We cannot communicate exceptions!

```cpp
auto main() → int {
  auto mutex = std::mutex{}; auto finished = std::condition_variable{}; auto shared = 0;
  auto thread = std::thread{[&]{
    std::this_thread::sleep_for(2s);
    auto guard = std::lock_guard{mutex}; // Lock the mutex
    shared = 42; finished.notify_all(); // Wakes up all threads waiting on condition
  }}; // Mutex is unlocked when guard goes out of scope
  std::this_thread::sleep_for(1s);
  auto lock = std::unique_lock{mutex}; // Lock the mutex
  finished.wait(lock); // Release mutex, wait until thread unlocks mutex, lock it again
  std::cout << "The answer is: " << shared << "\n"; thread.join(); }
```

7.4.1. std::future
CPPReference: std::future `#include <future>`

Future represents results that may be computed asynchronously. They allow us to *wait until the result is available*.
- `wait()`: blocks until available
- `wait_for(<timeout>)`: blocks until available or timeout elapsed,
- `wait_until(<time>)`: blocks until available or the timepoint has been reached

...and then *get the result*
- `get()`: blocks until available and returns the result value or throws

Their destructor may wait for the result to become available.

7.4.2. std::promise
CPPReference: std::promise `#include <future>`

Promises are the origin of futures. They allow us to *obtain* a future using `get_future()` and *publish* results or errors *(`set_result(<result_value>)` – set the associated future's result to result_value, `set_exception(<exception pointer>)` – set the associated future's exception)*.

```cpp
auto main() → int {
  using namespace std::chrono_literals;
  std::promise<int> promise{}; auto result = promise.get_future();
  auto thread = std::thread{
    [&]{std::this_thread::sleep_for(2s); promise.set_value(42); }
  };
  std::this_thread::sleep_for(1s);
  std::cout << "The answer is: " << result.get() << "\n"; thread.join(); }
```

7.5. STD::ASYNC
CPPReference: std::async `#include <future>`

Ready made *solution for computing asynchronously*. It allows us to *return our result* from our computation function. Additionally, it *catches all exceptions* and propagates them.

```
template<typename Function, typename ...Args>
auto async(Function&& f, Args&&... args) → std::future</* implicitly from f */>;
```

```cpp
auto main() → int {
  auto the_answer = std::async([] { /* calculate a while ... */ return 42; });
  std::cout << "The answer is: " << the_answer.get() << "\n"; }
```

This function *returns a `std::future`* that will store the result. `get()` waits for the result to be available. `std::async` can take an argument of type `std::launch` *(launch policy)*: *`std::launch::async`* launches a new thread and executes it regardless of if we need the result or not, *`std::launch::deferred`* defers execution until the result is obtained from the `std::future` *(lazy evaluation)*, which computes the result on the thread calling `get()`. The default policy is `std::launch::async | std::launch::deferred` *(environment dependent)*.

8. MEMORY MODEL & ATOMICS
The *C++ Standard* defines an *abstract machine* which describes *how a program is executed*. *Platform specifics* are *no longer relevant* with this abstraction. Represents the *"minimal viable computer"* required to execute a C++ program. The abstract machine defines in *what order initialization* takes place and in *what order* a program is executed. It defines what a *thread* is, what a *memory location* is, how threads *interact* and what constitutes a *data race*.

Memory Location: An object of scalar type *(Arithmetic, pointer, enum, `std::nullptr`. These values have the same amount of bits as the architecture defines, like 64-bit)*.
Conflict: Two expression evaluations run in parallel. Both access the same Memory Location *(at least one write, one read)*
Data Race: The program contains two conflicting actions. Undefined Behavior!

8.1. MEMORY MODEL
The C++ *Memory Model* defines when the effect of an operation is visible to other threads and how and when operations might be reordered. The Memory orderings define when effects become visible.
- *Sequentially-consistent* *(Same as code ordering and the default behavior)*,
- *Acquire/Release* *(Weaker guarantees than sequentially-consistent)*,
- *Consume* *(Discouraged, slightly weaker than acquire-release)* and
- *Relaxed* *(No guarantees besides atomicity)*.

Visibility of effects: *(If one thread modifies a variable, under what conditions is another thread guaranteed to see that modification?)* *sequenced-before:* *(A comes before B within a single thread)*, *synchronizes-with:* *(inter-thread sync, event that makes another thread see A, i.e. with Mutex)*, *happens-before* *(A happens-before B if A is sequenced-before or synchronizes-with B)*.

Read/Writes in a single statement are "unsequenced": `std::cout << ++i << ++i; // output unknown`

8.1.1. Atomics
CPPReference: std::atomic, CPPReference: std::atomic_flag, CPPReference: std::memory_order `#include <atomic>`

Template class to create atomic types. Atomics are guaranteed to be *data-race free*. There are several specializations in the standard library. The most basic type *`std::atomic_flag`* is lock-free.

```cpp
auto outputWhenReady(std::atomic_flag & flag, std::ostream & out) → void {
  // start critical section
  while (flag.test_and_set()) /* set flag to true and returns old value */ yield();
  out << "Here is thread: " << get_id() << std::endl;
  flag.clear(); // sets the flag to false
  // end critical section
}

auto main() → int {
  std::atomic_flag flag { };
  std::thread t { [&flag] { outputWhenReady(flag, std::cout);} };
  outputWhenReady(flag, std::cout); t.join(); }
```

When creating your own atomic type with `std::atomic<T>`, the atomic member operations are:
- `void store(T)` *(set the new value)*
- `T load()` *(get the current value)*
- `T exchange(T)` *(set the new value and return the old one)*
- `bool compare_exchange_weak(T & expected, T desired)` *(compare expected with current value, if equal replace the current value with desired, otherwise replace expected with current value. May spuriously fail)*
- `bool compare_exchange_strong(T & expected, T desired)` *(cannot fail spuriously)*

Applying Memory Orders: All atomic operations take an additional argument to specify the memory order *(Type `std::memory_order`)*. Example: `flag.clear(std::memory_order::seq_cst);`
- **Sequential Consistency (`seq_cst`):** Global execution order of operations. Every thread observes the same order. This is the Default behavior. The latest modification will be available to a read operation.
- **Acquire (`acquire`):** No reads or writes in the current thread can be reordered *before* this load. All writes in other threads that *release the same atomic* are visible in the current thread. *Not guaranteed to see latest write* (Half Fence), but ordering is consistent.
- **Release (`release`):** No reads or writes in the current thread can be reordered *after* this store. All writes in the current thread are visible in other threads that *acquire the same atomic*.
- **Acquire/Release (`acq_rel`):** Guaranteed to work on the latest value unlike Acquire, used for Read-Modify-Write operations (Full Fence) e.g. `test_and_set(...)`.
- **Relaxed (`relaxed`):** Does not give promises about sequencing. No data-races for atomic variables. Order can be inconsistent *(parallel `load()`/`store()` (Lost Updates))*, but may be more efficient. Difficult to get right.
- **Release/Consume:** Do not use! Data-dependency, hard to use. Better use `acquire`.

8.1.2. Custom Types with std::atomic
Custom types need to be trivially copyable. You *cannot have* a custom copy ctor, move ctor, copy assignment or move assignment. Object can only be accessed as a whole. No member access operator.

8.2. VOLATILE

```
volatile int mem{0};
```

Volatile in C++ is *different* from volatile in Java and C#. Load and store operations of volatile variables must *not be elided*, even if the compiler cannot see any visible side-effects within the same thread. Prevents the compiler from *reordering* within the same thread *(but the hardware might reorder instructions anyway)*. Useful when accessing *memory-mapped hardware*. *Never use for inter-thread communication!*

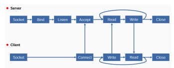
8.3. INTERRUPTS
Interrupts are *events originating from underlying system* which *interrupt the normal execution* flow of the program. Depending on the platform, they *can be suppressed*. When an interrupt occurs, a *previously registered function* is called *("Interrupt Service Routines" – ISRs)*. Should be *short* and must *run to completion*. After the interrupt was handled, execution of the program resumes.

Data *shared* between an ISR and the normal program execution needs to be *protected*. All accesses must be *atomic*, modifications need to become *visible*. *`volatile`* helps because it suppresses compiler optimizations. Interrupts may need to be *disabled temporarily* to guarantee atomicity.

9. NETWORK & ASYNC
Sockets are an abstraction of endpoints for communication.
- **TCP Sockets** are reliable, stream-oriented and require a connection setup
- **UDP Sockets** are unreliable, datagram-oriented and do not require a connection setup.

Connection-Oriented Communication Pattern using Sockets:

Socket: Create a new communication end point
Bind: Attach a local address to a socket *(IP, Port)*
Listen: Announce willingness to accept connections
Accept: Block caller until a connection request arrives
Connect: Actively attempt to establish a connection
Send: Send some data over the connection
Receive: Receive some data over the connection
Close: Release the connection



9.1. DATA SOURCES / BUFFERS
Transmit/Receive functions need *sources* or *destination* buffers. The ASIO library doesn't manage memory! **Fixed Size Buffers:** `asio::buffer()`. Must provide at least as much memory as will be read. Can use several standard containers as a backend. Pointer + Size combinations are available.
Dynamically Sized Buffers: `asio::dynamic_buffer()` Use if you do not know the required space and with `std::string` and `std::vector`.
Streambuf Buffers: `asio::streambuf`. Works with `std::istream` and `std::ostream`

9.2. ASIO LIBRARY
CPPReference: Extensions for networking `#include <asio.hpp>` *(if installed)*

9.2.1. Example: Synchronous TCP Client Connection with ASIO
Socket: All ASIO Operations require an I/O context. Create a TCP Socket using the context. There are asynchronous and synchronous functions to communicate with sockets.

```cpp
asio::io_context context{}; // multiple contexts possible
asio::ip::tcp::socket socket{context}; // multiple sockets per context possible
```

Connect: If the IP address is known, an endpoint can be constructed easily. `socket.connect()` tries to establish a connection to the given endpoint.

```cpp
auto address = asio::ip::make_address("127.0.0.1");
auto endpoint = asio::ip::tcp::endpoint{address, 80}; socket.connect(endpoint);
```

A resolver resolves the host names to endpoints. `asio::connect()` tries to establish a connection.

```cpp
asio::ip::tcp::resolver resolver{context};
auto endpoints = resolver.resolve(domain, "80"); asio::connect(socket, endpoints);
```

Write: `asio::write()` sends data to the peer the socket is connected to. It returns when all data is sent or an error occurred *(`asio::system_error`)*.

```cpp
std::ostringstream request{};
request << "GET / HTTP/1.1\r\n";
request << "Host: " << domain << "\r\n"; request << "\r\n";
asio::write(socket, asio::buffer(request.str())); // Blocks until data is sent
```

Read: `asio::read()` receives data sent by the peer the socket is connected to. It returns when the read-buffer is full, when an error occurred or when the stream is closed. The error code is set if a problem occurs, or the stream has been closed *(`asio::error::eof`)*.

```cpp
constexpr size_t bufferSize = 1024; std::array<char, bufferSize> reply{};
asio::error_code errorCode{};
auto readLength = asio::read(socket, asio::buffer(reply.data(), bufferSize), errorCode);
```

Advanced Reading: `asio::read` also allows to specify conditions on the data being read.
- *`asio::transfer_all()`* *(Default behavior, transfer all data or until buffer is full)*
- *`asio::transfer_at_least(std::size_t bytes)`* *(Read at least bytes number of bytes, may transfer more)*
- *`asio::transfer_exactly(std::size_t bytes)`* *(Read exactly bytes number of bytes)*

`asio::read_until()` allows to specify conditions on the data being read.
- Simple matching of characters or strings *(read until "\n")* or more complex matching using `std::regex`
- Allows to specify a callable object *(predicate → can be iterated over, returns sent data if predicate true. Expects `std::pair<iterator, bool> operator()(iterator begin, iterator end)`)*

Close: `shutdown()` closes the read/write stream associated with the socket. The destructor cancels all pending operations and destroys the object.

```cpp
socket.shutdown(asio::ip::tcp::socket::shutdown_both); // close read and write end
socket.close();
```

9.2.2. Example: Synchronous TCP Server with ASIO
Socket, Bind & Listen: An *acceptor* is a special socket responsible for establishing incoming connections. Is bound to a given local end point and starts listening automatically.

```cpp
asio::io_context context{};
asio::ip::tcp::endpoint localEndpoint{asio::ip::tcp::v4(), port}; //uses an available IPv4
asio::ip::tcp::acceptor acceptor{context, localEndpoint};
```

Accept: `accept()` blocks until a client tries to establish a connection *(with connect)*. It returns a new socket through which the connected client can be reached.

```cpp
asio::ip::tcp::endpoint peerEndpoint{}; // information about client
asio::ip::tcp::socket peerSocket = acceptor.accept(peerEndpoint);
```

9.2.3. Async communication
Handling multiple requests simultaneously: Using synchronous operations blocks the current thread. *Asynchronous Operations* allow further processing of other requests while the async operation is executed. Most I/O operations have asynchronous I/O operations.

1. The program *invokes an async operation* on an I/O object *(socket)* & *passes a completion handler* callback.
2. The I/O object *delegates* the operation and the callback to its *`io_context`*.
3. The OS *performs* the asynchronous operation.
4. The OS *signals* the `io_context` that the operation has been *completed*.
5. When the program calls `io_context::run()` the *remaining* asynchronous operations are *performed* *(wait for the result of the operation in the system)*.
6. Still inside the `io_context::run()` the *completion handler* is called to handle the *result* of the asynchronous operation.

9.2.4. Asynchronous Read/Write on Sockets
- *Async read operations:* `asio::async_read`, `asio::async_read_until`, `asio::read_at` *(certain position in buf.)*
- *Async write operations:* `asio::async_write`, `asio::async_write_at`

They return immediately. The operation is processed by the executor associated with the stream's `asio::io_context`. A completion handler is called when the operation is done.

```cpp
// Reads from async stream into buffer until '\n' is reached, then calls completion handl.
auto readCompletionHandler = [] (asio::error_code ec, std::size_t len_of_read) { /*...*/ }
asio::async_read_until(socket, buffer, '\n', readCompletionHandler);

// Writes data from a buffer to async stream until data has been written
// or an error occurs. Then calls the completion handler.
auto writeCompletionHandler = [] (asio::error_code ec, std::size_t len_of_write) {/*...*/}
asio::async_write(socket, buffer, writeCompletionHandler);
```

9.2.5. Asynchronous Acceptor
Create an *accept handler* that is called when an incoming connection has been established. The second parameter is the socket of the newly connected client. A `session` object is created on the heap to handle all communication with the client. `accept()` is called to *continue new inbound connection attempts*. The accept handler is registered to handle the next accept asynchronously.

The constructor creates the server. It initializes its acceptor with the given `io_context` and port. It calls `accept()` for registering the accept handler and does not block. To use it, create an `io_context` and the `server`. The executor will run until no async operation is left thanks to `asio::async_accept()`. It is important that the server lives as long as async operations on it are processed.

```cpp
struct Server { using tcp = asio::ip::tcp;
  Server(asio::io_context &c, unsigned short port)
    : acceptor{c, tcp::endpoint{tcp::v4(), port}} { accept(); }
private:
  auto accept() → void {
    auto acceptHandler = [this](asio::error_code ec, tcp::socket peer) {
      if (!ec) { // start new session if no error has occurred
        auto session = std::make_shared<Session>(std::move(peer));
        session→start();
      }
      accept(); // call accept again to continue accepting new connections.
    }; // If not done only one client session request will ever be handled
    acceptor.async_accept(acceptHandler); // Register acceptHandler to handle next accept
  }
  tcp::acceptor acceptor;
};
auto main() → int {
  asio::io_context context{}; Server server{context, 1234}; context.run();
}
```

9.2.6. Session with Asynchronous IO
The constructor stores the socket with the client connection. *`start()`* initializes the first async read, *`read()`* invokes async reading. *`write()`* invokes async writing. The fields store the data of the session, *`enable_shared_from_this`* is needed because the session object would die at the end of the accept handler, therefore it needs to be allocated on the heap. The *handlers need to keep the object alive* by pointing on it on the heap. If there is no pointer to the object left, it gets deleted.

```cpp
struct Session : std::enable_shared_from_this<Session> {
  explicit Session(asio::ip::tcp::socket socket); auto start() → void { read(); }
private:
  auto read() → void; auto write(std::string data) → void;
  asio::streambuf buffer{}; std::istream input{&buffer}; asio::ip::tcp::socket socket;
};
// Code in Accept handler
auto session = std::make_shared<Session>(std::move(peer)); session→start();
accept(); // if not called, only single connection possible
// Code in Read handler
auto Session::read() → void { auto readComplHandl = [self = shared_from_this()] /*...*/ }
// Code in Write handler
auto Session::write(std::string input) → void {
  auto data = std::make_shared<std::string>(input);
  auto writeCompletionHandler = [self = shared_from_this(), data] /*...*/}
```

9.2.7. Async Operation without Callbacks
Async operations can work "without" callbacks. Specify "special" objects as callbacks.
- *`asio::use_future`:* Returns a `std::future<T>`. Errors are communicated via exception in future
- *`asio::detached`:* Ignores the result of the operation
- *`asio::use_awaitable`:* Returns a `std::awaitable<T>` that can be awaited in a coroutine. Complicated!

9.3. SIGNAL HANDLING
CPPReference: Standard library header <csignal> `#include <csignal>`

Most OS support signals. Signals provide asynchronous notifications. They are used to gracefully terminate a program, communicate errors, notify about traps *("it's a trap!")*

`SIGTERM` *(Termination requested)*, *`SIGSEGV`* *(Invalid memory access)*, *`SIGINT`* *(User interrupt)*, *`SIGILL`* *(Illegal instruction)*, *`SIGABRT`* *(Abnormal termination)*, *`SIGFPE`* *(Floating point exception)*

9.3.1. Signal Handling in ASIO
`asio::signal_set` defines a set of signals to wait for. Handlers can be set up with `signal_set.wait()`/`signal_set.async_wait()` that take a lambda. The signal handler receives the signal that occurred and an error if the wait was aborted: `[&](auto error, auto sig){...}` Useful to cleanly stop server applications.

9.4. ACCESSING SHARED DATA
Multiple async operations can be in flight. All completion handlers are dispatched through `asio::io_context` and run on a thread executing `io_context.run()`. Multiple threads can call `run()` on the same `asio::io_context`. This results in a possible data race.

9.4.1. Strands
Strands are a mechanism to *ensure sequential execution* of handlers.
- *Implicit strands:* If only one thread calls `io_context.run()` or program logic ensures only one operation is in progress at a time.
- *Explicit strands:* For multiple threads on the same `io_context`. Objects of type `asio::strand<...>`. Created using `asio::make_strand(executor)` or `asio::make_strand(execution_context)`. Applied to handlers using `asio::bind_executor(strand, handler)`.

```cpp
// globally accessible
auto results = std::vector<int> { }; auto strand = asio::make_strand(context);
// in connection class
asio::async_read(socket, asio::buffer(buffer),
  asio::bind_executor(strand, [&](auto error, auto bytes) { // wrap access in bind_executor
    auto result = parse(buffer); results.push_back(result); /* bye bye data race */ }));
```

10. ADVANCED LIBRARY DESIGN
10.1. EXCEPTION SAFETY
There is code that *handles* exceptions, code that *throws* exception, and *exception neutral code* *(does not throw, does not catch, just forwards exceptions)*. Exception safety is important in generic code that manages resources or data structures *(might call user-defined operations, must not garble its data structures and must not leak resources)*

The *deterministic lifetime model of C++* requires exception safety. When an exception is thrown, "stack unwinding" ends the lifetime of temporary and local objects. Throwing an exception while another exception is "in flight" in the same thread causes the program to terminate.

10.1.1. Safety Levels (from highest to lowest)
- *noexcept aka no-throw:* Will never throw an exception *(and is always successful)*. Very hard to achieve, sometimes even impossible, e.g. memory allocation. *(Examples: Swap, Move Constructor, Move Assignment Operator, `std::vector<T>::clear()`)*
- *Strong exception safety:* Operation succeeds and doesn't throw, or nothing happens but an exception is thrown *(transaction, compare DB transactions)*. Hard to achieve. *(Examples: Copy Constructor, Copy Assignment Operator (Copy-Swap Idiom))*
- *Basic exception safety:* Does not leak resources or garble internal data structures in case of an exception *(guarantees invariance)*, but operations might be only half-done *(i.e. if copy throws, but internal members have not been adjusted yet)*

```cpp
auto push(value_type const & elem) → void {
  value_type val{elem}; // might throw here due to copy operation failing
  tail_ = (tail_ + 1) % (capacity() + 1); elements_++; /*elements not yet changed*/}
```

- *No guarantee:* Often unintentional, but happens. Invalid or corrupted data when an exception is thrown.

A function can only be as exception-safe as the weakest sub-function it calls!

	Invariant OK	*All or Nothing*	*Will Not Throw*
No Guarantee	✗	✗	✗
Basic Guarantee	✓	✗	✗
Strong Guarantee	✓	✓	✗
No-Throw Guarantee	✓	✓	✓

10.1.2. noexcept Keyword
CPPReference: noexcept specifier

`noexcept` belongs to the function signature. You *cannot overload* on `noexcept`. *`noexcept(expression)`* can be used to determine the *"noexceptiness"* of an expression, without computing it. `noexcept(noexcept(f()))` means "Outer function is `noexcept`, if function `f()` is also `noexcept`".

The compiler might *optimize* a call of a `noexcept` function *better*. But if you throw an exception from a `noexcept` function, `std::terminate()` will be called.

10.1.3. Member Functions that should not throw
- *Destructors* must not throw when used during stack unwinding
- *Move construction* and *move assignment* better not throw *(This is why it often uses swap internally)*
- *swap* should not throw *(`std::swap` requires non-throwing move operations)*
- *Copying* might throw, when memory needs to be allocated

10.1.4. Standard Library Helpers
CPPReference: std::move_if_noexcept `#include <utility>`

It may be hard for a container to implement its move operations if the element type does not support `noexcept`-move. Use `std::move_if_noexcept` instead: If `noexcept`, then move.

```cpp
explicit Box(T && t) noexcept(noexcept(T(std::move_if_noexcept(t))))
  : value(std::move_if_noexcept(t)) { ... }
```

There are other helpers like `is_nothrow_...`:
`constructible`, `move_constructible`, `default_constructible`, `assignable`, `move_assignable`, `copy_assignable`, `destructible`, `copy_constructible`, `swappable`

10.1.5. Wide and narrow contracts
- *Wide Contract:* A function that can handle all argument values of the given parameter types successfully. It cannot fail and should be specified as `noexcept(true)`. `this`, global and external resources are also possible parameters.
- *Narrow Contract:* A function that has preconditions on its parameters, e.g. `int` parameter must not be negative. Even if not checked and no exception is thrown, those functions *should not be `noexcept`*. This allows later checking and throwing.

10.2. PIMPL IDIOM (POINTER TO IMPLEMENTATION IDIOM)
CPPReference: PImpl

Opaque/Incomplete Types: Name known *(declared)*, but not the content. Introduced by a forward declaration. Can be used for pointers and references, but it cannot dereference values or access members without a later definition.

```cpp
struct S; // Forward Declaration
auto foo(S & s) → void { foo(s); /* S s{}; is invalid */}
struct S{}; // Definition
auto main() → int { S s{}; foo(s); /* s can now be used */ }
```

Problem: Internal changes in a class' definition require clients to re-compile *(e.g. changing a type of a private member variable)*. **Solution:** Create a "Compilation Firewall": Allow changes to implementation without the need to recompile users. It can be used to shield client code from implementation changes, meaning you must not change header files your client relies upon.

Put in the "exported" header file a class consisting of a "Pointer to Implementation" plus all public members. Basically place all public definition in the header file.

With *`std::shared_ptr<class Impl>`* we can use a minimal header and hide all details in the implementation `WizardImpl`. The `Wizard` class called from the header delegates all calls to `WizardImpl`.

```cpp
// wizard.hpp: Minimal Header
class Wizard {
  std::shared_ptr<class WizardImpl> pImpl;
public:
  Wizard(std::string name = "Rincewind");
  auto doMagic(std::string wish)
    → std::string;
};
```

```cpp
// WizardImpl.cpp: Full implementation of
// WizardImpl (all private members, not shown
// here) and Wizard (PImpl & public methods)
Wizard::Wizard(std::string name) :
  pImpl{std::make_shared<WizardImpl>(name)}{}

auto Wizard::doMagic(std::string wish)
  → std::string {
  return pImpl→doMagic(wish);
}
```

With *`std::unique_ptr<class Impl>`* we need to define the destructor of `Wizard` after the definition of `WizardImpl`. The compiler can't move the destructor by himself.

```cpp
// wizard.hpp
class Wizard {
  std::unique_ptr<class WizardImpl> pImpl;
public:
  Wizard(std::string name);
  ~Wizard(); // define explicitly
  auto doMagic(std::string wish)
    → std::string; };
```

```cpp
// WizardImpl.cpp
class WizardImpl { /* ... */ };
// ...
Wizard::~Wizard() = default;
```

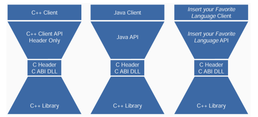
Because the default deleter of `std::unique_ptr` can't delete an incomplete type, we need to define the destructor explicitly.

10.2.1. Design Decisions with PImpl Idiom
How should objects be copied?
- *No copying - only moving:* `std::unique_ptr<class Impl>` *(declare destructor and move operations &= default)*
- *Shallow copying:* `std::shared_ptr<class Impl>` *(sharing the implementation)*
- *Deep copying:* `std::unique_ptr<class Impl>` *(with DIY copy constructor of Impl, default for C++)*

Never do `pImpl = nullptr`, and do not inherit from PImpl class.

11. HOURGLASS INTERFACES
11.1. APPLICATION BINARY INTERFACES (ABI)
ABIs define *how programs interact on a binary level* *(Names of structures and functions, calling conventions, Instruction sets)*. *C++ does not define* any specific ABI, because they are *tightly coupled* to the platform. They change between OSes, compiler versions, library versions, etc. Different STL implementations are *(usually)* incompatible.

Use *C(89)* as an "intermediate" layer, as "C frontend for our C++ code". This is a *extremely stable ABI*. No namespaces, no name mangling. But there are also no member functions, no exceptions, no templates.



11.1.1. Example: Wizard Class

```cpp
// C++ Library
struct Wizard {
  Wizard(std::string name = "Rincewind") : name{name}, wand{} {}
  auto doMagic(std::string const & wish) → char const *;
  auto learnSpell(std::string const & newSpell) → void;
  auto mixAndStorePotion(std::string const & potion) → void;
  auto getName() const → char const * { return name.c_str(); }
};
// C++ Client / Tests: Work normally as you would expect
using wizard_client::Wizard;
TEST(canCreateDefaultWizard) {
  Wizard const magician{};
  ASSERT_EQUAL("Rincewind", magician.getName());
}
```

Background C API: Abstract data types can be *represented by pointers*. Ultimate abstract pointer: `void *`. Member functions map to functions taking the abstract data type pointer as first argument. Requires *factory and disposal functions* to manage object lifetime. Strings can only be represented by `char *`. Make sure to not return pointers to temporary objects. Exceptions do not work across a C API, use a `Error` struct.

```cpp
// C Header / C API DLL (Wizard.h)
#ifdef __cplusplus // 'extern' only complies with a CPP, not C compiler
extern "C" { // mark header file as C to disable name mangling
#endif
typedef struct Wizard * wizard; // Wizard can only be accessed through pointers
typedef struct Wizard const * cwizard;
typedef struct Error * error_t; // Stores exception messages, needs to be cleaned up
wizard createWizard(char const * name, error_t * out_error); // Factory func to wrap ctor
void disposeWizard(wizard toDispose); // Factory function to wrap destructor
char const * error_message(error_t error); // Allocates error message on the heap
void error_dispose(error_t error); // Removes error message from the heap
char const * doMagic(wizard w, char const * wish, error_t *out_error);
void learnSpell(wizard w, char const * spell);
void mixAndStorePotion(wizard w, char const * potion);
char const *wizardName(cwizard w); // All member funcs take a wizard pointer as first arg
#ifdef __cplusplus
}
#endif
```

Parts of C++ that can be used in an extern "C" interface:
- Functions, but not templates. No overloading!
- C primitive Types *(char, int, double, void)*
- Pointers, including function pointers
- Forward-declared structs
- Unscoped Enums without class or base types
- If using from C you must embrace it with `extern "C"` when compiling it with C++ *(`extern "C" {}`)*

Implementing the Opaque Wizard Type: Wizard class must be implemented. To allow full C++ including templates, we need to use a "trampoline" class. It wraps the actual Wizard implementation.

```cpp
// Wizard.cpp
extern "C" {
struct Wizard { // C linkage trampoline
  Wizard(char const * name) : wiz{name} {}
  unseen::Wizard wiz;
};
```

```cpp
// WizardHidden.hpp
namespace unseen {
struct Wizard {
  /* see code from C++ library above */ };
}
```

Dealing with Exceptions: You can't use references in C API, you must use pointers to pointers. In case of an error, allocate error value on the heap. You must provide a disposal function to clean up. Internally, you can use C++ types, but you should return `char const *`, because the caller owns the object providing memory.
Creating Error Messages from Exceptions: Call the function body and catch expressions. Map them to an `Error` object, set the pointer pointed to by `out_error`. Passed out `out_error` must not be `nullptr`!
`*out_error = new Error{"Unknown internal error"}`
Error Handling at Client Side: Client-side C++ usage requires mapping error codes back to exceptions. Unfortunately, the exception type doesn't map through. But you can use a generic standard exception *(`std::runtime_error`)*. There is a dedicated RAII class for disposal. You could also use a *temporary object with throwing destructor*, but this is tricky because of possible leaking.

```cpp
// C++ Client API (Header Only)
struct ErrorRAII {
  ErrorRAII(error_t error)
    : opaque{error} {}
  ~ErrorRAII() {
    if (opaque) {
      error_dispose(opaque);
    }
  }
  error_t opaque;
};
```

```cpp
// Header continued
struct ThrowOnError{
  ThrowOnError() = default;
  ~ThrowOnError() noexcept(false) {
    if (error.opaque) { throw
std::runtime_error{error_message(error.opaque)};}
  }
  operator error_t*() { return &error.opaque; }
private:
  ErrorRAII error{nullptr};
};
```

```cpp
struct Wizard {
  Wizard(std::string const & who = "Rincewind")
    : wiz {create_wizard(who.c_str(), ThrowOnError{})} {}
    // C linkage trampoline
};
```

To complete `WizardClient.cpp`, call the C functions from `WizardClient.hpp` from global namespace *(i.e. `::do_magic()`)*. Delete the copy constructor & assignment to prevent a double free.

```cpp
struct Wizard {
  // ...
  auto doMagic(std::string, const &wish) → std::string {
    return ::do_magic(wiz, wish.c_str(), ThrowOnError{});
  }
private:
  Wizard(Wizard const &) = delete;
  Wizard & operator =(Wizard const &) = delete;
  wizard wiz;
}
```

11.2. JAVA NATIVE ACCESS (JNA)
JNA provides a *simple interface to C libraries*. Consists of a single JAR file and is cross-platform. For C++/Java Type mappings see slides page 31.

11.2.1. Loading Libraries

```java
public interface CplaLib extends Library {
  CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);
  void printInt(int number); // matches the function in the next chapter
}
```

Calling the loaded library handle *`INSTANCE`* is only by convention. The *loader* searches for a suitable library, first in the path specified by `jna.library.path`, otherwise in the system default library search path. Fallback is the class path.

11.2.2. Interfacing with Functions

```cpp
extern "C" { void printInt(int number); }
```

Function names and parameter types *must match*. However, the types are *not* validated. Parameter names don't matter.

11.2.3. Interfacing with Plain structs
(see example on slides starting at page 34)
Plain non-opaque `struct` types must *inherit from `Structure`*. You must override *`getFieldOrder()`* and you can use the tag-interface *`Structure.ByValue`*. You can access pointers to such types using *`getPointer()`*.

Opaque `struct` types should *inherit from `Pointer`* and provide a constructor using the `create...()` function. *Managing lifetime* is *not trivial*. Using `dispose...()` API functions in finalizers is not recommended. Either provide a dispose method on your Java type or implement `AutoClosable` and use your objects with `try-with-resources`.

11.2.4. Working with Raw Byte Arrays
(see example on slides starting at page 42)
Use *`IntByReference`* to retrieve the size of the buffer. Requires that the API supports it. *`getByteArray()`* copies the data from the buffer. Make sure to free the buffer either using an API `free...()` functions or `Native.free()` *(tends to crash on windows)*.

12. BUILD SYSTEMS
12.1. BUILD AUTOMATION
Good for *reproducibility*, *productivity*, *maintainability* and *shareability*. There are many IDEs which help build our projects. But sometimes, you don't want to rely on an IDE *(like on a build server, when sharing or reproducing with others)*

Do *not write your own scripts* for this process, because then every source file gets built *every time*, the commands tend to be *platform specific*, build order must be managed *manually* and scripts tend to become *messy* over time.

12.2. BUILD TOOLS
There are plenty of *build-tools:* GNU make, Scons, Ninja, CMake, autotools, ...
Features of Build Tools: Incremental builds, parallel builds, automatic dependency resolution, package management, automatic test execution, platform independence, additional processing of build products.
Different Classes of Build Automation Software:
- *Make-style Build Tools:* Run build scripts, produce your final products, often verbose, use a language agnostic configuration language
- *Build Script Generators:* Generate configurations for Make-style Build Systems or Build Scripts, configuration is independent of actual build tool, often have advanced features like download dependencies.

12.2.1. GNU make
Well-known tool to build all kinds of projects. *Many IDEs* "understand" make projects. The workflow description is in *Makefile via "Target" rules*. Each target may have one or more *prerequisites* and execute one or more *commands* to generate one or more *results*. Targets are then executed "top-down". A target is only executed if required.

```make
target: prereq_target
prereq_target: prereq_file other_target
  command_to_generate_output
other_target:
```

- *Pros:* Very generic, powerful pattern matching mechanism, builds only what is needed, when its needed
- *Cons:* Often platform-specific commands, need to specify how to do thins

12.3. BUILD SCRIPT GENERATORS
Define what we want to achieve, not how to do it. Work on a higher level, let the tool create the actual build configurations. Platform independent build specification, tool independent.

12.3.1. CMake
Has support for many languages and is platform independent.

```cpp
// main.cpp
#include <iostream>
int main() { std::cout << "Hello There!\n";}
```

```cmake
# CMakeLists.txt
project("my_app" LANGUAGES CXX)
add_executable("my_app" "main.cpp")
```

```
$ mkdir build # create separate build directory for the build outputs
$ cd build
$ cmake ..
$ cmake --build . # don't use "make" to build ("cmake && make" or just "make").
# Cmake uses the correct configured build tool, whereas make always assumes MakeFile
```

- *`cmake_minimum_required(...)`* sets the minimum required CMake version. This implicitly defines the available feature set.
- *`project(...)`* command defines the name of the project and which language we use
- *`add_executable(...)`* defines binaries
- *`target_compile_features(...)`* defines which language features are used by the target i.e. the C++ Standard or specific features. Prefer requiring standards rather than specific features!
- *`target_include_directories`* is used to define the include search path of the target. Default is non-system include path, specify `SYSTEM` to define path as being a system include path *(includes using <...>)*. Can be `PUBLIC` or `PRIVATE`
- *`set_target_properties(...)`* defines additional target properties
- *`add_library(...)`:* Defaults to static libraries, can be overridden at configuration time. All features, include paths and dependencies should be `PUBLIC`.
- *`target_link_libraries`* is used to define libraries required by a target. Can be `PUBLIC` or `PRIVATE`, applies `PUBLIC` features/dependencies/includes of the library

Variables can be defined using `set(VAR_NAME VALUE)`. They are referenced using `${VAR_NAME}$`. There are global variables like `PROJECT_NAME`, `PROJECT_SOURCE_DIR` or `PROJECT_BINARY_DIR`. Can be used in place of concrete values: `add_executable(${PROJECT_NAME} "source1.cpp" "source2.cpp" ...)`.

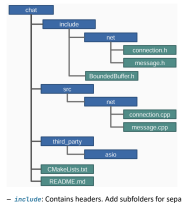
12.3.2. Testing with CMake
CMake includes CTest. Enable it with `enable_testing()`. Create a "Test Runner" executable.

```cmake
enable_testing()
add_executable("test_runner" "Test.cpp")
```

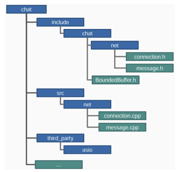
```
$ cmake ..  # configure build environment
$ cmake --build . # build the project
$ ctest --output-on-failure # run ctest
```

12.4. PROJECT LAYOUT
Best Practices: Separate headers from implementation files, group files by submodule / functionality. Be consistent!

Project Layout (General)



Project Layout (Libraries)



- *`include`*: Contains headers. Add subfolders for separate subsystems if needed
- *`src`*: Contains implementations. Subfolder layout should match `include` folder
- *`lib/third_party`:* Contains external resources like libraries
- *`test`*: Contains tests. Add above folders as necessary
- Build config files should be in the project root
- When creating a library, introduce another layer of nesting to avoid filename clashes in clients

13. MOVE SEMANTICS OUTPUT

```cpp
struct Example {
  Example(int a) { std::cout << "Ctr\n"; }
  Example(Example const&) { std::cout << "Copy Ctr\n"; }
  Example(Example&&) { std::cout << "Move Ctr\n"; }
  ~Example() { std::cout << "Dtor\n"; }
};
```

1. Creating new object with = still calls Constructor.

```cpp
Example a = 5; // "Ctr"
Example b = std::move(a); // "Move Ctr"
Example c = Example(std::move(f)); // "Move Ctr"
Example const d = c; // "Copy Ctr"
Example e = std::move(d); // "Copy Ctr" (Moving const not possible, fallback)
// "Dtor" 5 times
```

2. Constructor elides creation of temporaries (Optimization)

```cpp
auto defaultExample() → Example { return Example{5}; }
Example f{Example{5}}; // "Ctr"
Example g{defaultExample()}; // "Ctr"
// "Dtor" 2 times
```

3. Temporary lvalue in methods

```cpp
auto moveExample(Example e) { auto h = std::forward<Example>(e); } // "Move Ctr" "Dtor"
moveExample(g); // "Copy Ctr" "Dtor"
```