0 Contents

1 Move Semantics

1.1 Copy

By default cpp will always create copies, this is good for memory safety etc, as you will not be returning null values, but it can be a runtime hit!
(There are some special types that can't be copied like mutexes etc)

```
// Copy contructor
class something {
  something(const something &other) {
    // copy values from other
  }
}
```

1.2 Move

Move constructor will *NOT copy values, instead, it will move these values into the new object, this is better for performance, but it requires more management from the programmer!*
Make sure to free the memory at the old object, otherwise you might be dealing with nullpointers!

```
Vector(Vector<T> &&vec)
    : size(vec.size), cap(vec.cap), data(std::move(vec.data)) {
  vec.data = nullptr;
} // yes this is the vector that you implemented kekw
```

In short, the move constructor makes a lot of sense when you have *Heap data*, aka if you have something like an array or a vector, then you will want to make sure to always use the move constructor if you can do so.
The default move constructor is as follows:

```
struct S {
  S(S && s) : member{std::move(s.member)}
  {...}
  M member;
};
```

1.3 Copy Assignment

Default copy assignment constructor:

```
struct S {
  auto operator=(S const& s) -> S& {
     member = s.member;
     return *this;
  }
  M member;
};
```

1.4 Move Assignment

Default move assignment constructor:

```
struct S {
  auto operator=(S&& s) -> S& {
    member = std::move(s.member);
    return *this;
  }
  M member;
};
```

1.5 Rvalue and Lvalue

lvalue T&: *variable with some location in ram*, either on the stack or on the heap.
rvalue T&&: *temporary value* that has no variable and no location in memory, it only exists in code.

```
int a = 5;
// 5 is an r value, it has no memory location
// a is an lvalue -> some address is set to 5

int b = 10;

int c = a + b;
// a + b is an rvalue -> value is 15, but no memory location for this calculation
// c is an lvalue -> some address is set to 5
```

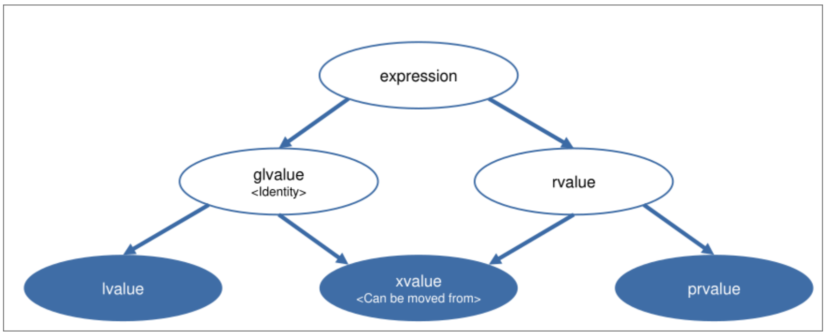
1.5.1 Convert lvalue to rvalue

By default you can't just use an lvalue as an rvalue, however, you can use *std::move* to explicitly convert an lvalue to an rvalue.
Note that in this case, you *can't use the old variable anymore, as the data has been moved! -> see rust*

```
auto consume(Food&& food) -> void;

auto fryBurger() -> Food;
auto fastFood() -> void {
  Food fries{"salty and greasy"};
  consume(fryBurger()); //call with rvalue
  consume(fries); //cannot pass lvalue to rvalue reference
  consume(std::move(fries)); //explicit conversion lvalue to xvalue
  Food&& burger = fryBurger(); //life-extension of temporary
}
```

1.6 Other value types

has identity?	can be moved from?	Value Category
Yes	No	lvalue
Yes	Yes	xvalue (expiring value)
No	No (Since C++17)	prvalue (pure rvalue)
No	Yes (Since C++17)	- (doesn't exist anymore)



- lvalue
  - address can be taken
  - Can be on the left-hand side of an assignment if modifiable
  - Can be used to initialize lvalue references
  - Examples: variables, function calls that return reference, increment and decrement operators, array index access if array is lvalue
  - all string literals
- prvalue
  - address can't be taken -> doesn't exist
  - cannot be on the left hand side of assignment
  - temporary "materialization" to xvalue
  - Examples: literals, false, nullptr, function call with non reference return type, postincrement and postdecrement!!
- xvalue
  - address cannot be taken
  - Cannot be used as left-hand operator of built-in assignment
  - Conversion from prvalue through temporary materialization
  - Examples: function calls with rvalue reference return type -> std::move, access of non-references members of an rvalue object, arra index access when array is rvalue

1.6.1 Temporary Materialization

Getting from something imaginary to something you can point to....
When this happens:

- binding a reference to a prvalue
- when accessing a member of prvalue
- when accessing an element of a prvalue array
- when converting a prvalue array to a pointer
- when initializing an std::initializer_list<T> from a braced-init-list
- Type needs to be complete and needs to have a destructor

```
struct Ghost {
  auto haunt() const -> void {
    std::cout << "booooo!\n";
  }
  //~Ghost() = delete;
};
auto evoke() -> Ghost {
  return Ghost{};
}
auto main() -> int {
  Ghost&& sam = evoke(); // bind reference to a prvalue
  Ghost{}.haunt(); // access member of prvalue
}
```

1.7 l and rvalue references

- lvalue reference made only of lvalues!!
  - type: T&
  - alias for a variable
  - can be used as function member type, local member/variable, return type
  - be aware of dangling references when returning!
- rvalue reference made of rvalues, prvalues or xvalues!
  - Type: T&&
  - when assigned to a name (for example inside of a function), then it is actually an lvalue!!
  - Argument is either a literal or a temporary object

```
      std::string createGlass() -> std::string;
  void fancyNameForFunction() {
    std::string mug{"cup of coffee"};
    std::string&& glass_ref = createGlass(); //life-extension of temporary
    std::string&& mug_ref = std::move(mug); //explicit conversion lvalue to rvalue
    int&&
    i_ref = 5;
    //binding rvalue reference to prvalue
  }
```

1.8 Binds

`T value{};` `std::cout << value;`	lvalue
`int value{};` `std::cout << value + 1;`	rvalue

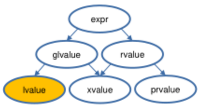
`auto foo(T& param) -> void {`
`  std::cout << param;`
`}`
	lvalue

`auto print(T&& param) -> void {`
`  std::cout << param;`
`}`
	lvalue

`auto create() -> T;` `create();`	rvalue

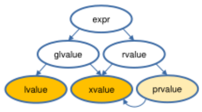
`T & create();` `create();`	lvalue
`T && create();` `create();`	rvalue
`T value{};` `std::cout << value + 1;`	depends on +
`T value{};` `T o = std::move(value);`	rvalue
`std::cout << "Hello";`	lvalue

- **lvalue Reference**
  - binds



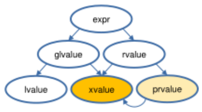

```
auto f(Type&) -> void;

Type t{};
f(t);
```

- **const lvalue Reference**
  - binds




```
auto f(Type const&) -> void;

Type t{};
f(t);
f(std::move(t));
f(Type{});
```

- **rvalue Reference**
  - binds




```
auto f(Type&&) -> void;

Type t{};
f(Type{});
f(std::move(t));
```

	`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 &)

	`S::m()`	`S::m() const`	`S::m() &`	`S::m() const &`	`S::m() &&`
`S s{};` `s.m();`	✓	✓	✓ (preferred over const &)	✓	✗
`S const s{};` `s.m();`	✗	✓	✗	✓	✗
`S{}.m();`	✓	✓	✗	✓	✓ (preferred over const &)
`S s{};` `std::move(s).m();`	✓	✓	✗	✓	✓ (preferred over const &)

1.9 Destructor

Whenever you need to write an explicit destructor, please make sure that you will not throw exeptions here. This can cause memory to not be freed, which.... well you guess what heppens In general you should make sure that *ANY form of memory management doesn't throw exceptions!!!*

1.10 Default Constructors and user defined Constructors

What you get

What you write

Where you want to be

Avoid if possible

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted (!)	defaulted (!)	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted (!)	not declared	not declared
copy assignment	defaulted	defaulted	defaulted (!)	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

The ! means that it is a standard library bug, don't use those defaulted ones!!!
Note that deleting a constructor will be the same as "user declared"!!

1.11 The problem with func(T const&)

When working with const T references, this implies that we can either *copy or move it*, this means we will not necessarily know what we get.
The only possible way without type deduction is an overload for both.

```
template <typename T>
  auto log_and_do(T const& param) -> void {
  //log
  do_something(param);
} // lvalue
template <typename T>
  auto log_and_do(T&& param) -> void {
  //log
  do_something(std::move(param));
} // lvalue and rvalue!!
```

Note, with more parameters, you would need x amount of overloads for each combination of parameters!!

2 Type Deduction

2.1 Forwarding Reference

A T&& is not always an rvalue! In some cases, it is a forwarding reference, which can be either an lvalue or an rvalue!!

```
template <typename T>
auto f(T && param) -> void;

// lvalue
int x = 23;
f(x);
// auto f(int & param) -> void; (inferred)

// rvalue
f(23);
// auto f(int && param) -> void; (inferred)
```

2.2 Rules for Type Deduction

```
// base function
template <typename T>
```

```cpp
auto f(T param) -> void;

// type usages with function instances and deduced T
int          x = 23; // f(x) = f(int param) -> T = int
int const   cx = x; // f(cx) = f(int param) -> T = int
int const& crx = x; // f(crx) = f(int param) -> T = int
char const * const ptr = /* something */; // f(ptr) = f(char const * param) -> T = char const*;
// -- ignore outermost const
// -- ignore reference types
// -- take base type

// base function 2
template <typename T>
auto f(T & param) -> void;

// type usages with function instances and deduced T
int          x = 23;  // f(x) = f(int& param) -> T = int
int const   cx = x;   // f(cx) = f(int const& param) -> T =int const
int const& crx = x;   // f(crx) = f(int const& param) -> T = int const
// -- ignore reference type

// base function 3
template <typename T>
auto f(T const& param) -> void;

// type usages with function instances and deduced T
int          x = 23; // f(x) = f(int const& param) -> T = int
int const   cx = x;  // f(cx) = f(int const& param) -> T = int
int const& crx = x;  // f(crx) = f(int const& param) -> T = int
// -- ignore reference types
// -- take base type

// base function 4
template <typename T>
auto f(T&& param) -> void;

// type usages with function instances and deduced T
int          x = 23; // f(x) = f(int& param) -> T = int&
int const   cx = x;  // f(cx) = f(int const& param) -> T = int const&
int const& crx = x;  // f(crx) = f(int const& param) -> T = int const&
//                   // f(27) = f(int&& param) -> T = int
// -- if param is an lvalue, then they become lvalue references
// -- otherwise rvalue, default rules for references
```

2.2.1 Deducing Initializer Lists

With initializer lists, you can't directly deduce the type as it will think T is the entire list, which is nonsense!

```cpp
template <typename T>
auto f(T param) -> void;
f({23}); //error

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

2.2.2 Deducing auto types

```cpp
autox = 23;          //auto is a value type
auto const cx = x;   //auto is a value type
auto& rx = x;        //auto is a reference type
auto&& uref1 = x;    //x is an lvalue, uref1 is int&
auto&& uref2 = cx;   //cx is an lvalue, uref2 is int const&
auto&& uref3 = 23;   //23 is an rvalue, uref3 is int&&

// special cases
auto init_list1 = {23};   //std::initializer_list<int>
auto init_list2{23};      //int, was std::initializer_list<int>
auto init_list3{23, 23}; //Error, requires one single argument
```

Note that auto type deduction works with parameters and return types, with the special cases like initializer list still applying!!

2.2.3 Type Deduction with Decltype

```cpp
int            x       = 23;
int const      cx      = x;
decltype(cx)   cx_too  = cx; //type of cx_too is int const
int&           rx      = x;
decltype(rx)   rx_too  = rx; //type of rx_too is int&

// these two are the only surprises! auto only gives the base type without reference, while the other gives the full reference
    type
auto just_x = rx; //type of just_x is int
decltype(auto) more_rx = rx; //type of more_rx is int&
```

decltype(auto) etc can also be used for returning something specific:

```cpp
// auto decltype
template <typename Container, typename Index>
decltype(auto) access(Container & c, Index i) {
  return c[i];
}

// specific decltype
template <typename Container, typename Index>
auto access(Container & c, Index i) -> decltype(c[i]) {
  return c[i];
}
```

Note we can only declare decltype(c[i]) as a trailing type! The reason for this is that c and i are only known AFTER the parameters!

2.2.4 Returns with decltype

```
decltype(auto) funcName() {
  int local = 42;
  return local; // decltype(local) => int
} // lvalue -> T
decltype(auto) funcNameRef() {
  int local = 42;
  int & lref = local;
  return lref; // int & -> bad (dangling)
} // lvalue reference -> T&
decltype(auto) funcXvalue() {
  int local = 42;
  return std::move(local); // int && -> bad (dangling)
} // rvalue reference -> T&&
decltype(auto) funcLvalue() {
  int local = 42;
  return (local); // int & -> bad (dangling)
} // lvalue reference -> T&
decltype(auto) funcPrvalue() {
  return 5; // int
} // prvalue -> T
```

2.3 Checking for r and l-values

We learned that we can solve the issue of multiple overloads with T&&, but what if we want to differentiate after the fact? std::forward!

```
template <typename T>
auto log_and_do(T&& param) -> void {
  //log
  do_something(std::forward<T>(param));
}

// example for implementation
template <typename T>
  decltype(auto) forward(std::remove_reference_t<T>& param) {
  return static_cast<T&&>(param);
}
// explanation
// this will check if we have an lvalue or not by trying to cast to an rvalue reference
// if & and && are casted, it will always result in &
// this means only an rvalue will result in an rvalue being returned, everything else will result in lvalue being returned
// this is called reference collapsing!
// example -> when T is int& the static cast will be int& && and hence collapsed to int&
// when T is int&& the static cast will be int&& && and hence collapsed to int&&
// when T is int, the static cast will be int&&, no collapse is needed here.
// note references are only checked for the type, the actual references are removed, as can be seen by the std::
    remove_reference_t
```

This means that forwards is essentially *a conditional cast to an rvalue reference!*

Rules for reference collapsing:

- & and & = &
- && and & = &
- & and && = &
- && and && = &&

2.3.1 std::move vs std::forward

While forward is the *conditional cast*, std::move is the *unconditional cast*! This means you will always receive an rvalue!

```
// std::forward
template <typename T>
  decltype(auto) forward(std::remove_reference_t<T>& param) {
  return static_cast<T&&>(param);
} // will collapse dynamically
// std::move
template <typename T>
decltype(auto) move(T&& param) { // param is always T&& !!!
  return static_cast<std::remove_reference_t<T>&&>(param);
} // will always collapse to && and && meaning && is returned
```

3 Lambdas

3.1 From lambda to actual code

```
// lambda
int i0 = 42;
auto missingMutable = [i0] {return i0++;};

// compiler code
struct CompilerKnows {
  auto operator()() const -> int {
    return i0++;
  }
  int i0;
};
```

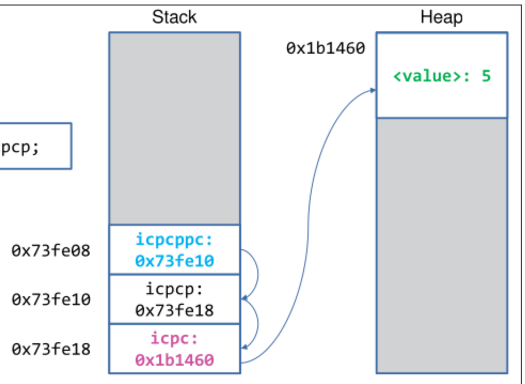
4 Memory Management and Heap

4.1 Pointers

Funny pointer consty fun.

- **Pointers can be const**
  - **const** is on the RIGHT side of the *

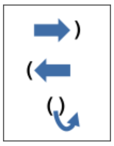
```
int const * const * * const icpcppc = &icpcp;
```



4.1.1 Reading a pointer declaration

int const	* const	*	* const	icpcppc;
(5)	(4)	(3)	(2)	(1)

icpcppc is a const pointer to a pointer to a const pointer to a const int
(1) (2) (3) (4) (5)




```
void (* f)(int &, double)
(4)  (2)(1)     (3)
```

```
f is)
(a pointer to
a function, taking a reference to int and a double, returning
void
```

```
int (*f(int(*)(int))) (int);
```

f is a function
this function takes a pointer to a function as argument (1)
and returns a pointer to a function (2)
(1) this function takes an int and returns an int
(2) this function takes an int and returns an int

```
using alias = int(*)(int);
alias f(alias);
```

4.1.2 nullptr

The nullptr has a more specific meaning than either 0 or NULL,
other than 0, it has no implicit conversion to integral type, unlike 0, and also ensures no mistakes with overloads -> again integral.
There is also the implicit conversion from nullptr to T*

```
int* test = nullptr;
float* test2 = nullptr;

// lol
void* something = nullptr;
int* no = (int*) something;
```

4.2 Const

By default the const keyword needs to be on the right, the only exception is the first type on the left!

```
int const i; //both declarations
const int i; //are the same
```

const applies to its left neighbor; only if there is no left neighbor it applies to its right neighbor

Be careful with left const assignments when using aliases!

```
// Extract the int const * part
using alias = int const *;
alias const icpc; // works well

// Extract the int * const part
using alias = int * const;
const alias cipc; // this is bs! Compiles however!
```

4.3 mutable

The mutable keyword is always used on the variable itself!

```
// the value at mutable_const_int_pointer is constant
// however the pointer itself is not!
// the mutable keyword here is only used for const functions -> can be used inside of them
class Something {
  mutable const int * mutable_const_int_pointer;
}
```

4.4 New

```
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}};
}
```

4.5 Delete

Every new needs to be accomodated with a delete, *deleting twice will lead to undefined behavior!*.
However, deleting the nullptr is well defined, it does nothing.

```
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};
  //pp member access with pp->
  //pp is the pointer value
  delete pp; //destructor
}
```

Using delete with [] will delete arrays.

```
struct Point {
  Point(int x, int y) :x {x}, y {y}{}
  int x, y;
}
auto funWithPoint(int x, int y) -> void {
  Point * arr = new Point[2]{{0, 0},{x, y}};
  //element access with [], e.g. arr[1]
  //arr points to the first element
  delete[] arr; //destructors
} // this also deletes multidimensional arrays!!
```

4.5.1 Placement new

This takes a ptr where *currently no element is placed* and creates a new class instance of choice in this pointer.
This means that you can potentially create a pointer to a smaller instance. It just needs to be suitable, aka big enough, so bigger objects won't work!!

```
struct Point {
  Point(int x, int y):x {x}, y {y}{}
  int x, y;
};
auto funWithPoint() -> void {
  auto ptr = new Point{9, 8};
  // must release Point{9, 8}
  // release can be done with ptr->~NewTest();
  // or with std::destroy_at(ptr);
  new (ptr) Point{7, 6};
  delete ptr;
}
```

4.5.2 Placement Destroy

There is no proper placement destroy, instead there is the *regular destructor, but that one doesn't work with primitive built-in types, so instead use std::destroy_at.*

```
struct Resource {
  Resource() {
    /*allocate resource*/
  }
  ~Resource() {
    /*deallocate resource*/
  }
};
auto funWithPoint() -> void {
  auto ptr = new Resource{};
  ptr->~Resource();
  new (ptr) Resource{};
  delete ptr;
}
```

4.5.3 Non Default Constructible Types

This refers to types that do not have a constructor with no parameters. -> defualt constructor
With these types we can't use new TypeName, instead we need to allocate memory explicitly like this:

```
struct Point {
  Point(int x, int y); // default deleted!
  ~Point();
  int x, y;
};

// allocate memory
auto memory = std::make_unique<std::byte[]>(sizeof(Point) * 2);

// initialize
new (memory.get()) Point{1, 2};
```

Accessing these individually is tedious, how about e helper?

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

auto memory = std::make_unique<std::byte[]>(sizeof(Point * 2));
Point * first = &elementAt(memory.get(), 0);
new (first) Point{1, 2};
Point * second = &elementAt(memory.get(), 1);
new (second) Point{4, 5};

// make sure to also destroy it manually!
// it ain't rust so get shit on
// order is irrelevant for the memory management itself.
std::destroy_at(second);
std::destroy_at(first);
```

You have to destroy the memory manually however!
The reason for this is that each object might have heap allocated memory itself, this is NOT guaranteed to be cleaned up.

4.5.4 New and Delete are fucking operators...

```cpp
struct not_on_heap {
  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, never called, but should come in pairs
  }
  static auto operator delete[](void *ptr) -> void noexcept {
    // do nothing, never called, but should come in pairs
  }
  // just no
  // but you can create your own allocators
  // or simply make sure that noone ever calls new or delete with your types, kekw
};
```

4.5.5 Typical Problems with memory

```cpp
auto foo() -> void {
  int * ip = new int{5};
  //exit without deleting
  //location ip points to
}
```


```cpp
auto foo() -> void {
  int * ip = new int{5};
  delete ip;
  delete ip;
}
```


```cpp
auto foo() -> void {
  int * ip = new int{5};
  delete ip;
  int dead = *ip;
}
```


```cpp
auto bar() -> void;

auto foo() -> void  {
  int * ip = new int{5};
  bar(); //exception?!
  delete ip;
}
```

```cpp
auto foo(int * p) -> void  {
  //is it up to me to
  //delete p? likely not
}
```

```cpp
auto create() -> int * {
  int * ip = new int{5};
  return ip;
}
auto foo() -> void {
  int * ip = create();
  //My turn to delete?
  //Probably yes
}
```

5 Static vs Dynamic Polymorphism

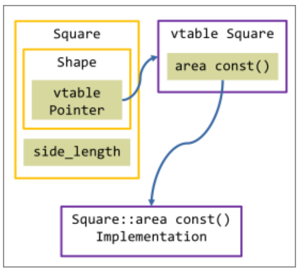
5.1 Static

- faster at runtime
  no need to ckeck or cast function, just use it
- slower at compile time
  each implementation used will be made with macros
- syntax checking is off -> lsp limitation in c++
- larger binaries -> more code

5.2 Dynamic

The problem is displayed as follows:

```cpp
struct Shape {
  virtual unsigned area() const = 0;
  virtual ~Shape();
};
struct Square : Shape {
  Square(unsigned side_length)
  : side_length{side_length} {}
  unsigned area() const {
    return side_length * side_length;
  }
  unsigned side_length;
};

decltype(auto) amountOfSeeds(Shape const & shape) {
  auto area = shape.area();
  return area * seedsPerSquareMeter;
};
```

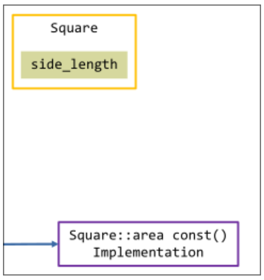


The problem is that we need to cast when using these functions. Once again you can see the shit that is inheritance as it forces this conveluted casting style of writing code.

5.2.1 Comparison to static

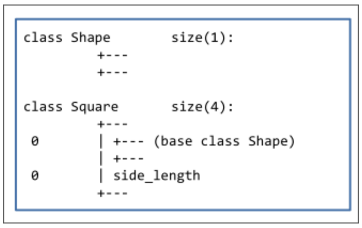
```cpp
struct Square {
  Square(unsigned side_length)
  : side_length{side_length} {}
  unsigned area() const {
    return side_length * side_length;
```

```cpp
  }
  unsigned side_length;
};

template<typename ShapeType>
decltype(auto) amountOfSeeds(ShapeType const & shape) {
  auto area = shape.area();
  return area * seedsPerSquareMeter;
}

// instance -> not written by programmer -> made by compiler
// decltype(auto) amountOfSeeds(Square const & shape) {
//   auto area = shape.area();
//   return area * seedsPerSquareMeter;
// };
```

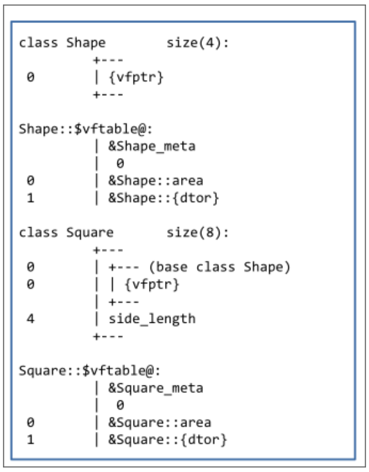


The only downside it that you can't use this with dynamic types, but once again this is why you don't use this crap. Remember the pain that was in your rust game, the same thing would happen here.

5.2.2 Dynamic Dispatch Virtual Table



```
class Shape       size(1):
        +---
        +---

class Square      size(4):
        +---
 0      | +--- (base class Shape)
        | +---
 0      | side_length
        +---
```



```
class Shape       size(4):
        +---
 0      | {vfptr}
        +---

Shape::$vftable@:
        | &Shape_meta
        |  0
 0      | &Shape::area
 1      | &Shape::{dtor}

class Square      size(8):
        +---
 0      | +--- (base class Shape)
 0      | | {vfptr}
        | +---
 4      | side_length
        +---

Square::$vftable@:
        | &Square_meta
        |  0
 0      | &Square::area
 1      | &Square::{dtor}
```

Note that the size(1) in the first figure is simply there, because in C++ each object needs to be differentiable.
This means that you need some sort of address to do that. If this object doesn't actually exist, then there will be no size, as can be seen in the square.

6 Substitution Failure

The template itself does not throw a compilation error, meaning that if the template itself can't be done with a specific type, then we simply ignore the type for this template. However, if we then go ahead and use this function somewhere and this specific type didn't work with this function, then we will receive a compiler error. This is also the reason why the lsp is so bad at showing errors when it comes to templates.

```cpp
template <typename T>
auto increment(T value) -> T {
  return value.increment();
} // here string is just not considered as string has no .increment

increment("pingpang"); // error bro
```

You can use the dropping of instances of templates by using functions that only work on certain types in order to protect against using with strange types.

6.1 Type Traits

Compares two types according to traits
Note: These only work in *Templates, Parameters and Return Types*, NOWHERE else!

```cpp
template <typename T, typename U>
struct is_same : false_type {
  // inherits
  // static constexpr bool value = false;
};
template <typename T>
struct is_same<T, T> : true_type {
  // inherits
  // static constexpr bool value = true;
};
template <typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;
```

- std::is_same<T,U> compares the 2 types
- std::is_same_v<T,U> same but results in bool -> ::value
- std::is_same_t<T,U> same but results in type -> ::type
- std::is_class<T> Checks to see if type is a class type
- std::is_same_v<T> same but results in bool -> ::value
- std::negation_v<T> negates the value
- std::is_reference<T> checks if type is a reference type
- std::is_constructible_v<T> checks if compiler is constructible

```cpp
#include <type_traits>
```

```cpp
struct S{};

auto main() -> int {
  std::is_class<S>::value; // true
  std::is_class<int>::value; // false
}
```

- std::enable_if<bool, T> checks if type is of given type
- std::enable_if_t<bool, T> -> ::type

```cpp
template <bool expr, typename T = void>
struct enable_if{
  template <bool expr,typename T = void>
  struct enable_if{};

  template <typename T>
  struct enable_if<true, T> {
    using type = T;
  };

  template <bool expr,typename T = void>
  using enable_if_t = typename enable_if<expr, T>::type;
};

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

Possibilities of application:

```cpp
template <typename T>
auto increment(T value) -> std::enable_if_t<std::is_class_v<T>, T> {
  return value.increment();
}

template <typename T>
auto increment(std::enable_if_t<std::is_class_v<T>, T> value) -> T {
  return value.increment();
} // enable_if as parameter, impairs type deduction

template <typename T, typename = std::enable_if_t<std::is_class_v<T>, void>>
auto increment(T value) {
  return value.increment();
} // would be void per default
```

6.1.1 Constructors and type checks

```cpp
template <typename T>
struct Box {
  Box() = default;
  template <typename BoxType, typename = std::enable_if_t<std::is_same_v<Box, BoxType>>>
  explicit Box(BoxType && other)
  : items(std::forward<BoxType>(other).items) {}
  // only matches when entered type can be made into .items
  explicit Box(size_t size)
  : items(size) {}
  //...
  private:
    std::vector<T> items{};
};
```

The problem is that with forward, the matching gets *eager*, this means that int would match to the BoxType && other, resulting in an error since int doesn't have .items
This is just an example, do not use this over proper copy and move constructors

7 Requires C++20

This is the solution to the previously complicated way of handling template type requirements
It can be done in these two ways:

```cpp
// after template, works for structs, classes and functions
template<typename T>
requires true // or anything that can resolve to bool
auto function(T argument) -> void {}

// after return type, only works for functions
template<typename T>
auto function(T argument) -> void requires true {}


// explicit example
template <typename T>
requires std::is_class_v<T>
auto function(T argument) -> void {}
```

7.1 Requires as function

Sequence of actions:

```cpp
requires {
  // Sequence of requirements
}
```

Requires with parameters:

```cpp
requires ($parameter-list$) {
  // Sequence of requirements
}
```

Example:

```cpp
template <typename T>
requires requires (T const v) { v.increment(); }
auto increment(T value) -> T {
  return value.increment();
}
// yes, you need two requires.....
```

7.2 Subtype Requirements

```cpp
template<typename T>
requires {
  typename BoundedBuffer<T>::value_type;
  typename BoundedBuffer<T>::size_type;
  typename BoundedBuffer<T>::reference;
  typename BoundedBuffer<T>::const_reference;
}
```

7.3 Compund Requirements

```cpp
template <typename T>
requires requires (T const v) {
  { v.increment() } -> std::same_as<T>;
} // check if the return of the check to v.increment type == T
auto increment(T value) -> T {
  return value.increment();
}
```

7.4 Concept Keyword

These are essentially just traits...

```cpp
template <typename T>
concept Incrementable = requires (T const v) {
  {
    v.increment()
  } -> std::same_as<T>;
}; // potential to use || or && to chain requires!!
```

7.4.1 Usage

These are the same:

```cpp
template <Incrementable T>
auto increment(T value) -> T {
  return value.increment();
}

template <typename T>
requires Incrementable<T>
auto increment(T value) -> T {
  return value.increment();
}
```

7.5 AutoTemplates

You can use the auto keyword to automatically use templates:

```cpp
// both are the same
auto function(auto argument) -> void {}

template <typename T>
auto function(T argument) -> void {}
```

7.5.1 Problems with auto templates

```cpp
auto function(auto arg1, auto arg1) -> void {}

// ignored!!!!
template <typename T>
auto function(T arg1, T arg2) -> void {}

// chosen, the auto automatically converts to this
template <typename T1, typename T2>
auto function(T1 arg1, T2 arg2) -> void {}
```

7.5.2 Concept with auto

```cpp
// both are the same
auto increment(Incrementable auto value) -> T {
  return value.increment();
}

template <Incrementable T>
auto increment(T value) -> T {
  return value.increment();
}
```

8 Compile Time Evaluation

8.1 Legacy

global const variables are essentially the same thing as const in rust -> evaluated at compile time.

```cpp
size_t const SZ = 6 * 7; // evaluated during compilation
double x[SZ]; // -> x == SZ
```

8.2 Constant Expression Contexts

These are the contexts where compile time evaluation is possible:

- non-type template arguments

```
std::array<Element, 5> arr{}
```

- array bounds

```
double matrix[ROWS][COLS]{}
```

- Case expressions

```
switch(value) {
case 42: // ...
}
```

- Enumerator Initializer

```
enum Light {
Off = 0, On = 1
};
```

- static_assert

```
static_assert(order == 66);
```

- constexpr variables

```
constexpr unsigned pi = 3;
```

- constexpr if statements

```
if constexpr (size > 0) {
// ..
}
```

- noexcept

```
Blob(Blob &&) noexcept(true);
```

8.3 static_assert

Can be used to check things on compile time -> e.g. tests during compile time.
Note, the compilation fails if the assert fails!

```
// usage: static_assert(condition, message(optional))
static_assert(isGreaterThanZero(Capacity));
static_assert(sizeof(int) == 4, "unexpected size of int");
```

8.4 constexpr/constinit

variables evaluated at compile time -> literal values 5,6,"asdf",constexpr functions.

```
constexpr unsigned pi = 3;
constinit unsigned pi = 3;
```

- scopes
  local, namespace, global -> static
- constexpr variables are const
- constinit variables are not const!

8.5 constexpr functions

Can't use exceptions, which are shit either way!
Functions that are evaluated at compile time:

```
constexpr auto factorial(unsigned n) {
//...
}
```

Note, these functions can only have variables of literal type, and these MUST be initialized before used.
You can also:

- use loops
- branches -> if can be evaluated at compile time -> no exceptions!
- can only call constexpr functions
- allocate new memory with new,delete or unique pointers etc.
- use constexpr functions as virtual functions in classes and structs

Note that you can use constexpr in non constexpr contexts, in this case it will try to evaluate this constexpr at compile time if possible

8.5.1 Consteval

These are essentially the same thing as constexpr functions, but they will *always* evaluate at compile time, this means it can only be used in constexpr contexts.

```
consteval auto factorial(unsigned n) {
  auto result = 1u;
  for (auto i = 2u; i <= n; i++) {
    result *= i;
  }
  return result;
}
constexpr auto factorialOf5 = factorial(5);
auto main() -> int {
  static_assert(factorialOf5 == 120);
}
```

8.6 Undefined behavior and compiler

Interestingly enough the compiler will prevent undefined behavior in the compilation itself, instead it will just stop the compilation and return an errormessage.
Note that for compile time evaluations, should code not actually reach invalid code, then it will just work, since that code was not reached.... another template fun thing with c++...

```
constexpr auto throwIfZero(int value) -> void {
  if (value == 0) {
    throw std::logic_error{""};
  }
}
constexpr auto divide(int n, int d) -> int {
  throwIfZero(d);
  return n / d;
}
constexpr auto five = divide(120, 24);
// this is not reached, if it would be then it would not compile
constexpr auto failure = divide(120, 0);
```

8.7 Literal Types

- Trivial Destructor
- special literal types:
  - Lambdas
  - References
  - Arrays of Literal Types
  - void
  - int, double, pointers, enums, literal strings, strings, etc.

You can create your own literal type:

```
template <typename T>
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 auto length() const -> T {
    auto squares = x() * x() +
    y() * y() +
    z() * z();
    return std::sqrt(squares);
  }
  constexpr auto x() -> T& {
    return values[0];
  }
  constexpr auto x() const -> T const& {
    return values[0];
  }
  //...
};
```

- at least one constexpr or consteval constructor
- trivial destructor
- const and non-const functions possible
- note that only constexpr or consteval functions are done at compile time!
- Can be a template
- Other functions don't *need* to be constexp or consteval!

8.7.1 Compile Time template class computation

```
template <size_t n>
struct fact {
  static size_t const value{(n > 1)?
};
n * fact<n-1>::value : 1};
  template <>
  struct fact<0> { // recursion base case: template specialization
  static size_t const value = 1;
};
TEST(testFactorialCompiletime) {
  constexpr auto result = fact<5>::value;
  ASSERT_EQUAL(result, 2 * 3 * 4 * 5);
}
```

8.7.2 Captures in Lambdas are also literal types

```
constexpr auto cubeVolume(double x) {
  // x is literal
  auto area = [x] {return pi * x * x;};
  return area() * x;
}
constexpr auto cV = cubeVolume(5.0);
```

8.7.3 Variable declaration as templates

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

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

// the idea is that you can have recursive variable declarations.... wtf?
```

8.8 User Defined Literal-Suffixes

8.8.1 Problem

```cpp
template <typename Unit>
struct Speed {
  constexpr explicit Speed(double value)
  : value{value}{};
  constexpr explicit operator double() const {
    return value;
  }
  private:
  double value;
};
```

Quite verbose

Example	Valid
`Speed<Unit::kmh> s{5.0};`	Yes
`Speed<Unit::kmh> s = 5.0;`	Non-explicit
`auto s = Speed<Unit::kmh>{5.0}`	Yes
`auto s = 5.0;`	Not a speed object

8.8.2 Solution

This is basically an overload on a literal ending → This means you can create something like numbers with endings like kph, kilo or whatever you need.
In order to achieve this, you need to overload the "" operator ending with your literal.
Also note, The literal needs to be in a namespace in order to avoid confusion.
This suffix should only have these literals inside of it, nothing else!

```cpp
namespace velocity::literals {
  constexpr inline auto operator"" _kph(unsigned long long value) -> Speed<Kph> {
    return Speed<Kph>{safeToDouble(value)};
  }
  constexpr inline auto operator"" _kph(long double value) - Speed<Kph> {
    return Speed<Kph>{safeToDouble(value)};
  }
  auto speed1 = 5.0_kph;
  auto speed2 = 5.0_mph;
  auto speed3 = 5.0_mps;
}
```

This version is used to avoid wrappers that add a lot of boilerplate code and makes it hard to use as you need to unwrap the wrappers. Note, you can only overload a set of types -> (unsigned long long), (char const* , std::size_t), (char const *)
I assume only literal types???

8.8.3 String as suffix

```cpp
auto operator"" _suffix(char const *, std::size_t len) -> TYPE

namespace mystring {
  inline auto operator"" _s(char const *s, std::size_t len) -> std::string {
    return std::string { s, len };
  }
}
// ...
using namespace mystring;
auto s = "hello"_s;
s += " world\n";
std::cout << s;
```

Or you can convert integers and floats to string:

```cpp
auto operator"" _suffix(char const *) -> TYPE

// this takes the non 0 terminated strings
namespace mystring {
  inline auto operator"" _s(char const *s) -> std::string
    return std::string { s };
  }
}
```

Note, these can't be constexpr!

8.8.4 Compile Time User Defined Suffixes

```cpp
// variadic version of suffix operator
template <char ...Digits> requires (is_ternary_digit(Digits) && ...)
constexpr auto operator"" _ternary() -> unsigned long long {
  return ternary_value<Digits...>;
}

constexpr auto three_to(std::size_t power) -> unsigned long long {
  return power ? 3ull * three_to(power - 1) : 1ull;
}

template <char ...Digits>
extern unsigned long long ternary_value;

// handle 0
template <char ...Digits>
constexpr unsigned long long ternary_value<'0', Digits...> {
  ternary_value<Digits...>
};

// handle 1
template <char ...Digits>
constexpr unsigned long long ternary_value<'1', Digits...> {
  1 * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};

// handle 2
template <char ...Digits>
constexpr unsigned long long ternary_value<'2', Digits...> {
  2 * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};
```

```
// handle base case
template<>
constexpr unsigned long long ternary_value<>{0};
```

Newer versions of c++ also allow a more concise version

```
constexpr auto is_ternary_digit(char c) -> bool {
  return c == '0' || c == '1' || c == '2';
}
constexpr auto value_of(char c) -> unsigned {
  return c - '0';
}
template <char D, char ...Digits>
constexpr ternary_value<D, Digits...> {
  value_of(D) * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};
```

8.8.5 Default Suffixes

- string -> s
- std::complex -> i,il,if (imaginary numbers)
- std::chrono::duration -> ns,us,ms,s,min,h (time)

9 Multithreading and Mutexes

9.1 std::thread

This is a replacement for the POSIX API which is rather dated, and does not lead to clean code.
It mostly works, but some things need to fall back to POSIX, or microtroll API on windoof.

```
#include <thread>

auto main() -> int {
  // just like in rust -> create thread with closure/lambda
  std::thread greeter {
    [] { std::cout << "Hello, I'm thread!" << std::endl; }
  };
  // join the main thread -> blocking
  greeter.join();
}
```

9.1.1 Functors

You can also use a struct/class as parameter to pass into the std::thread.
This essentially means defining a struct with the function operator()() -> which essentially means turning it into a lambda with data attached.

```
#include <thread>

struct Functor {
  auto operator()() const -> void {
    std::cout << "Functor" << std::endl;
  }
};

auto function() -> void {
  std::cout << "Function" << std::endl;
  // return value ignored -> aka only void supported
}

auto main() -> int {
  std::thread functionThread{function};
  std::thread functorThread{Functor{}};
  functorThread.join();
  functionThread.join();
}
```

- Default consructible
- return values are ignored -> not supported within std::thread

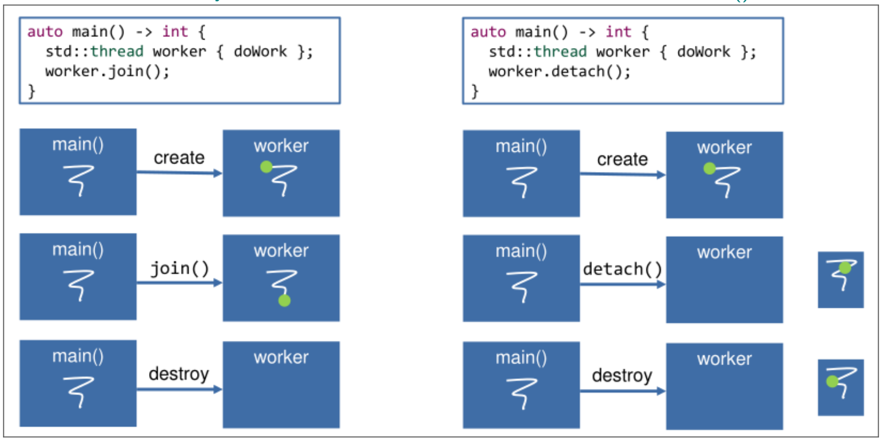
9.2 Passing arguments to threads

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


// usage
auto fibonacci(std::size_t n) -> std::size_t {
  if (n < 2) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
auto printFib(std::size_t n) -> void {
  auto fib = fibonacci(n);
  std::cout << "fib(" << n << ") is "
  << fib << '\n';
}
auto main() -> int {
  std::thread function { printFib, 46 };
  std::cout << "waiting..." << std::endl;
  function.join();
}
```

- std::thread constructor takes a function/functor/closure and arguments to forward
- passing arguments either by value, or you have to make sure references live long enough -> hello rust :)
- capturing references creates shared data -> no check for singular mutability

9.3 Destroying threads

Since we are not using the POSIX API, we need another way of ending the thread, and just like every c++ thing, it does not do this automatically.
This means that you need to do this on your own with either *join() or detach()*.
The first will attach the thread to the main thread, meaning the main thread will cascade destroy the thread.
The second will destroy the thread at the next available frame from the main() thread.



Note, without .join() or .detach() your program is killed in order to avoid undefined behavior(for once).

9.3.1 Dangers

```cpp
auto startThread() -> void {
  using namespace std::chrono_literals;
  std::string local{"local"};
  std::thread t{[&] {
    std::this_thread::sleep_for(1s);
    std::cout << local << std::endl;
  }};
  t.detach();
}
auto main() -> int {
  using namespace std::chrono_literals;
  startThread();
  std::this_thread::sleep_for(2s);
}
// problem, main thread can terminate before second thread -> therefore cout is no longer available
// cout is a global that is created with the main thread, therefore it is now a dangling reference!
```

- detach or join can't be called inside destructors! -> exception problems
- unjoined and undetached threads can be destroyed with *std::terminate()*
- when using .detach(), make sure you no longer use references from that thread -> danling references, nullptr... FUN

9.4 std::jthread

Thread that will automatically call .join().

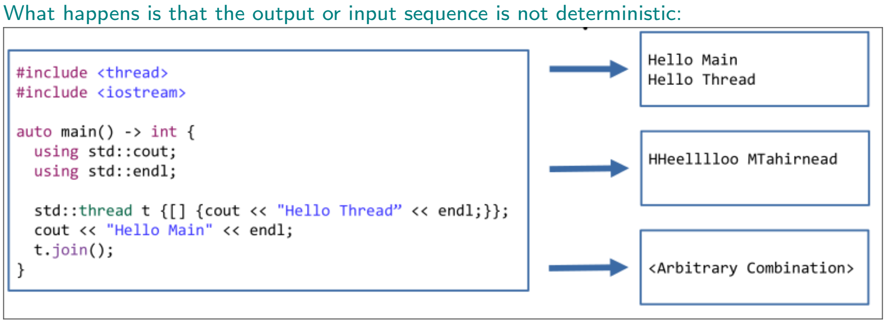
```cpp
auto main() -> int {
  std::jthread t {[]{
    std::cout << "Hello Thread"<< std::endl;
  }};
  std::cout << "Hello Main" << std::endl;
}
```

This thread can also be stopped by other threads with *thread.request_stop()*

```cpp
auto main() -> int {
  // stop token makes it possible to check how many times stop was requested
  // e.g. thread safe, and you can define how many times this will work
  // request stop is blocking
  // stop token only works with jthread
  std::jthread t {[](std::stop_token token){
    // check if stop was requested
    while (!token.stop_requested()) {
      std::this_thread::sleep_for(100ms);
    }
    std::cout << "Stop requested"<< std::endl;
  }};
  std::this_thread::sleep_for(2s);
  // request stop
  t.request_stop();
}
```

9.4.1 iostream and threads

IOstream is done via buffers, this is *not threadsafe*, however it also does not create data races.
What happens is that the output or input sequence is not deterministic:



9.5 Current Thread

There is a helper namespace *std::this_thread* which holds helper functions for the current thread:

```
auto main() -> int {
  using std::cout;
  using std::endl;
  using namespace std::chrono_literals;
  std::thread t { [] {
    std::this_thread::yield();
    cout << "Hello ID: "
         << std::this_thread::get_id()
         << endl;
    std::this_thread::sleep_for(10ms);
  }};
  cout << "main() ID: "
       << std::this_thread::get_id()
       << endl;
  cout << "t.get_id(): "
       << t.get_id()
       << endl;
  t.join();
}
```

- get_id()
  returns ID of current thread by the OS
  This ID is unique
- sleep_for(time)
  suspends the thread for a duration
- sleep_until(time_point)
  sleep until a certain time point
- yield()
  Allows OS to schedule other threads

9.6 Mutexes

- lock(): blocking
- try_lock(): non-blocking
- unlock(): non-blocking
- try_lock_for(duration): non-blocking
  try to lock for a specific duration
- try_lock_until(time): non-blocking
  try to lock until specific time

Mutex versions:

- std::mutex
  standard mutex, not recursive, not timed
- std::recursive_mutex
  recursive mutex -> allows multiple nested acquire operations of the same thread
  not timed
- std::timed_mutex
  timed, not recursive -> allows try_lock_for() etc.
- std::recursive_timed_mutex
  timed and recursive

		Recursive: No	Recursive: Yes
Timed	No	`std::mutex`	`std::recursive_mutex`
	Yes	`std::timed_mutex`	`std::recursive_timed_mutex`

9.6.1 Read Locks

As already covered extensively with rust, multiple reads are allowed, but not multiple writes.
Hence c++ std::thread also provides read shared locks:

- lock_shared()
- try_lock_shared()
- try_lock_shared_for(duration)
- try_lock_shared_until(time)
- unlock_shared()

9.6.2 Mutex helper functions

- std::lock_guard
  for *single* mutex
  locks when constructed
  unlocks when destructed
- std::scoped_lock
  for *multiple* mutex
  locks when constructed
  unlocks when destructed
- std::unique_lock
  defered timed locking
  allows explicit locking and unlocking
  unlocks when destructed (in case still locked)
- std::shared_lock
  wrapper for shared mutexes
  allows explicit locking and unlocking
  unlocks when destructed (in case still locked)

9.6.3 Example for thread save queue

```
template <typename T,
typename MUTEX = std::mutex>
struct threadsafe_queue {
  using guard = std::lock_guard<MUTEX>;
  auto push(T const &t) -> void {
```

```cpp
    guard lk{mx};
    q.push(t);
  }
  T pop() { /* later */ return T{};}
  auto try_pop(T & t) -> bool {
    guard lk{mx};
    // note the use of q instead of this
    // function from queue used!
    if (q.empty()) return false;
    t = q.front();
    q.pop();
    return true;
  }
  auto empty() const -> bool{
    guard lk{mx};
    return q.empty();
  }
  private:
  // mutable needed in the empty function
  mutable MUTEX mx{};
  std::queue<T> q{};
};
```

- Makes every member function mutually exclusive
- delegates functionality to std::queue
- scoped lock pattern
  automatically locks and unlocks
- strategized locking pattern
  template parameter for mutex type
  could also be null_mutex(boost)

9.6.4 Multiple Locks with std::scoped_lock

```cpp
// can't be noexcept, because locks might throw
auto swap(threadsafe_queue<T> & other) -> void {
if (this == &other) return;

std::scoped_lock both{mx, other.mx};

argumentsstd::swap(q, other.q);
// no need to swap mutex or condition variable
}
```

- acquires multiple locks in the constructor
- avoids deadlocks by relying on internal sequence
- blocks until all locks could be acquired
- Class template argument deduction avoids the need for specifying the template arguments

9.6.5 Multiple Locks without std::scoped_lock

```cpp
// can't be noexcept, because locks might throw
auto swap(threadsafe_queue<T> & other) -> void {
if (this == &other) return;

// std::defer_lock prevents immediate locking
lock my_lock{mx, std::defer_lock};
lock other_lock{other.mx, std::defer_lock};

// blocks until all locks are acquired
std::lock(my_lock, other_lock);

std::swap(q, other.q);
// no need to swap mutex or condition variable
}
```

- acquires multiple locks in a single cell
- avoids deadlocks
- blocks untl all locks could be acquired
- can also be done with try_lock -> in that case no blocking

9.7 std::condition_variable

- Waiting for the condition
  - wait(mutex)requires surrounding loop
  - wait(mutex, predicate) loops internally
  - timed: wait_for and wait_until
- notifying a (potential) change
  - notify_one
  - notify_all
- std::unique_lock as condition -> releases lock

```cpp
template <typename T,
typename MUTEX = std::mutex>
struct threadsafe_queue {
  using guard = std::lock_guard<MUTEX>;
  using lock = std::unique_lock<MUTEX>;
  auto push(T const & t) -> void {
    guard lk{mx};
    q.push(t);
    // like jafuck -> other thread can activate
    notEmpty.notify_one();
  }
  auto pop() -> T {
    lock lk{mx};
    // wait for condition
    notEmpty.wait(lk, [this] {
      return !q.empty();
    });
    T t = q.front();
```

```
    q.pop();
    return t;
  }
  private:
    mutable MUTEX mx{};
    std::condition_variable notEmpty{};
    std::queue<T> q{};
};
```

9.8 Containers

All current standard containers are *NOT thread safe*, this means that we will have to build thread safe versions of it.
Note that accessing a singular element from a container is not a data race -> as singular elements are different from each other.
Concurrent uses of containers are dangerous by default!
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

9.9 Returns from a thread

9.9.1 Shared state

We can return shared state, but this is not intuitive:

```
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};
    shared = 42;
    finished.notify_all();
  }};
  std::this_thread::sleep_for(1s);
  auto lock = std::unique_lock{mutex};
  finished.wait(lock);
  std::cout << "The answer is:"
            << shared << '\n';
  thread.join();
}
```

9.9.2 std::future and std::promise

Future represent result that maybe compute asynchronously:
- wait(): blocks until available
- wait_for(timeout): blocks until available or timed out
- wait_until(time): blocks until available or timepoint has been reached
- get(): blocks until available and returns the result value or throws if the future contains an exception

Promises are one origin of futures:
- get_future(): obtain a future
- set_result(value): sets the associated futures result
- set_exception(err): sets the associated exception

Usage:

```
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();
}
```

9.10 std::async

```
// definition:
template<typename Function, typename ...Args>
auto async(Function&& f, Args&&... args) -> std::future<...>;

// usage:
auto main() -> int {
  auto the_answer = std::async([] {
    // Calculate for 7.5 million years
    return 42;
  });
  std::cout << "The answer is: " << the_answer.get() << '\n';
}
```

- Schedules the execution of the lambda (*CAN BE IN SAME THREAD!*)
- returns an std::future that will store the result
- get() waits for the result to be available

9.10.1 std::async::launch and std::async::deferred

std::async::launch: forces the async to *definitely use a new thread!*
std::async::deferred: defers execution until the result is obtained from the std::future
By default it does either of the two, so just make sure to define it!
std::aync::launch

```
auto main() -> int {
  // new thread guaranteed
  auto the_answer = std::async(std::launch::async, [] {
    // Calculate for 7.5 million years
    return 42;
  });
  std::cout << "The answer is: " << the_answer.get() << '\n';
}
```

std::async::deferred

```cpp
auto main() -> int {
  // lazy evaluation, simply returns the 42 as soon as ready, in this case instantly!!
  auto the_answer = std::async(std::launch::deferred, [] {
    // Calculate for 7.5 million years
    return 42;
  });
}
```

C++ Advanced

Fabio Lenherr

summary

Table of Contents

1. Memory Model and Atomics

1.1. The C++ standard

1.1.1. C++ Standard defines an abstract machine

- describes how a program is executed
- Abstracts away platform specifics
- Represents the "minimal viable computer" requried to execute a valid C++ program

1.1.2. The C++ abstract machine defines:

- in what order initialization takes place
- in what order a program is executed
- what a thread is
- what a memory location is
- how threads interact
- what constitutes a data race

1.1.2.1. Memory Location

An object of scalar type:
- arithmetic
- pointer
- enum
- std::nullptr -> pointer to 0

1.1.2.2. Conflict

- two expression evaluations run in parallel and both access the same **Memory Location**

-> one writes, the other reads -> see rust borrow checker

1.1.2.3. Data Race

The program contains two conflicting actions -> **Undefined Behavior**

1.2. Memory Model

- defines when the effect of an operation is visible to other threads -> when change happens globally
- How and when operations might be reordered -> reordering of code lines

Note: Reads/Writes in a single statement are "unsequenced" -> they are not in guaranteed order, see SQL problems

```
std::cout << ++i << ++i; // output not deterministic!
```

1.2.1. Visibility of effects

- sequenced-before: within a single thread
  as code was written
  Note, this is only guaranteed for multi line statements, not for single line statements like cout!! -> see Note above
- happens-before: either sequences-before or inter-thread happens-before
- synchronizes-with: inter-thread sync.

1.2.2. Ordering

This is the way code is executed -> as written ? Can the compiler reshuffle for optimizations?

- Sequentially-consistent: “intuitive” and the default behavior
  as code was written
  When you change away from this, make sure you actually **NEED** another ordering‼
- Acquire/Release: weaker guarantees than Sequentially-consistent
  can reshuffle
- Consume (*discouraged*): slightly weaker than acquire-release *can reshuffle*
- Relaxed: No guarantees besides atomicity! *can reshuffle*

1.3. Atomics

Definition:

```
template<typename T>
struct atomic;

class atomic_flag;
```

- Template class to create atomic types
- *Atomics are guaranteed to be data-race free!*
- Several spezializations in the standard library
- Most basic atomic: std::atomic_flag
  - Guaranteed to be lock-free
  - clear() - sets the flag to false
  - test_and_set() -set flag to true and return old value
- Other atomics might use locks internally
  - check with is_lock_free()

Usage:

```
auto outputWhenReady(std::atomic_flag & flag,
                     std::ostream & out) -> void {
  while (flag.test_and_set())
    yield();
  out << "Here is thread: “
      << get_id()
      << std::endl;
  flag.clear();
}
auto main() -> int {
  std::atomic_flag flag { };
  std::thread t { [&flag] {
    outputWhenReady(flag, std:: cout);}
  };
  outputWhenReady(flag, std::cout);
  t.join();
}
```

The reason why only these 2 are implemented -> clear and test_and_set, is that these 2 functions are supported by *all* CPUs!

1.3.1. Operations on Atomics

Using your own types with std::atomic is possible, but they must be **trivially-copyable** -> no container in container etc.

`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 (even when current value == `expected`). `compare_exchange_strong` cannot fail spuriously, but might be slower		

spezializations like std::atomic can provide things like ++, –, +=, etc.

1.4. Applying Memory Orders

All atomic operations take an additional argument to specify the memory order -> (std::memory_order)

- std::memory_order::seq_cst
- std::memory_order::acquire
- std::memory_order::release
- std::memory_order::acq_rel
- std::memory_order::relaxed
- std::memory_order::consume

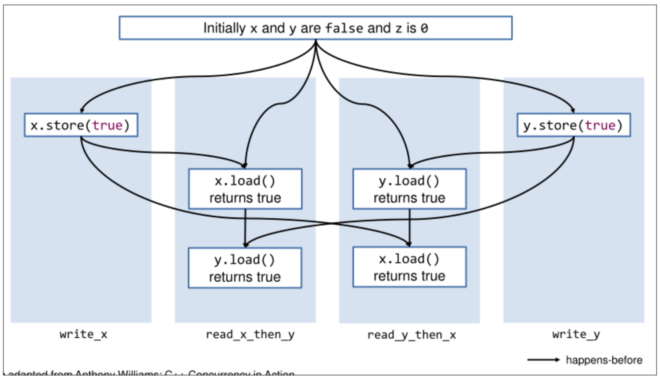
```cpp
auto outputWhenReady(std::atomic_flag & flag,
  std::ostream & out) -> void {
                                                  while
(flag.test_and_set(std::memory_order::seq_cst))
    yield();
  out << "Here is thread: "
      << get_id()
      << std::endl;
  flag.clear(std::memory_order::seq_cst);
}
auto main() -> int {
  std::atomic_flag flag { };
  std::thread t { [&flag] {
    outputWhenReady(flag, std::cout);}
  };
  outputWhenReady(flag, std::cout);
  t.join();
}
```

1.5. Sequentially-Consistent Mode

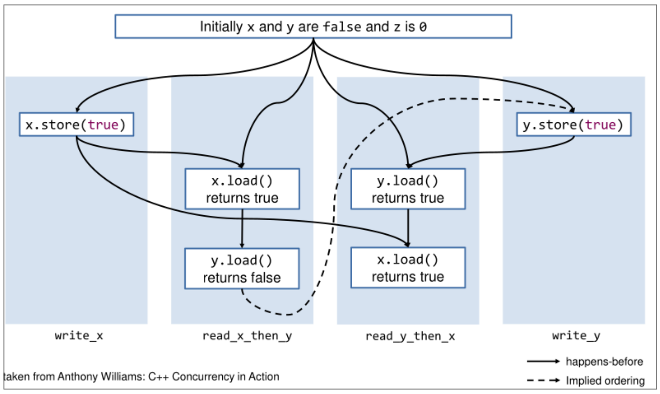
- Sequential Consistency
  - Global execution order of operations
  - every thread observes the same order
- Memory order flag
  - std::memory_order::seq_cst
- Default behavior
- The latest modification (in the global execution order) will be available to a read

```cpp
// thread1
auto write_x() {
  x.store(true);
}
// thread2
auto write_y() {
  y.store(true);
}
// thread3
auto read_x_then_y() {
  while (!x.load());
  if (y.load()) ++z;
}
// thread3
auto read_y_then_x() {
  while (!y.load());
  if (x.load()) ++z;
}
```

Z == 2



Z == 1



1.6. Acquire/Release Mode

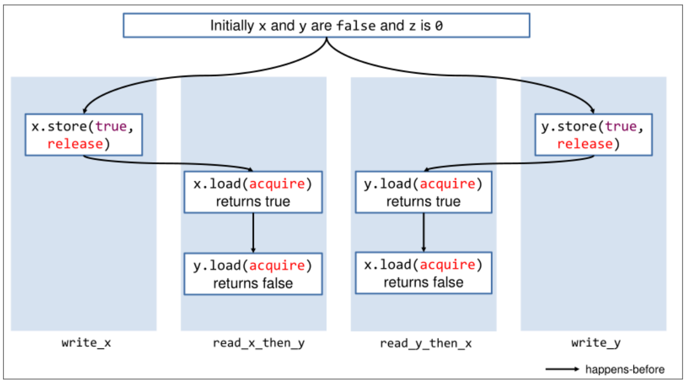
Has 3 different versions:
- std::memory_order::acquire
  - no reads or writes in the current thread can be reordered before this load
  - All writes in other threads that release the same atomic are visisble in the current thread
- std::memory_order::release
  - No reads or writes in the current thread can be reordered after this store
  - All writes in the current thread are visible in the other threads that acquire the same atomic
- std::memory_order::acq_rel
  - Works on the latest value

    This is the way to always receive the up-to-date value!

Usage:

```
x.test_and_set(std::memory_order::acq_rel);
```

Z == 0



The issue here is that you no longer have a guaranteed order, as you can see, the atomicity is given, but the order in which x and y are written or read is not consistent.
Note, the ordering of x and y alone is still ok, but not the ordering of both combined.

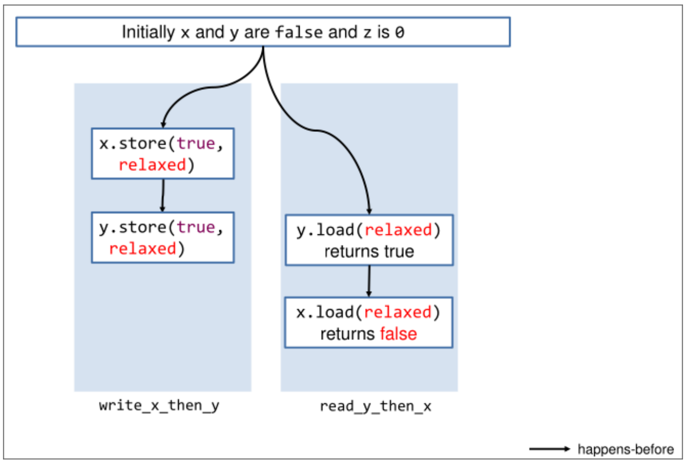
1.7. Relaxed Ordering Mode

- No promises about sequence whatsoever
- No data-races for atomic values -> **the only guarantee**
- Order of observable effects can be inconsistent
  - load and store operations can happen in parallel

- May be more "efficient" on certain platforms
  - less synchroniztion means less pipeline stalling or waiting for memory loads
- Extremely difficult to get right! You will need to prove the correctness of this program!

```cpp
#include <atomic>
#include <thread>
#include <cassert>
std::atomic<bool> x{};
std::atomic<bool> y{};
std::atomic<int> z{};
auto write_x_then_y() -> void {
  x.store(true, std::memory_order::relaxed);
  y.store(true, std::memory_order::relaxed);
}
auto read_y_then_x() -> void {
  while(!y.load(std::memory_order::relaxed)); // Spin
  if(x.load(std::memory_order::relaxed))
  ++z;
}
auto main() -> int {
  auto a = std::thread{write_x_then_y};
  auto b = std::thread{read_y_then_x};
  a.join();
  b.join();
  assert(z.load() != 0);
}
```

Z == 0



Here no order guarantees are given at all, aka even the order of x and y themselves can be shuffled!!

1.8. Release/Consume Ordering

- Don't fucking use, even the standard tells you to not use this!
- Similar to Acquire/Release
  - Introduces data-dependency concept
    - dependency-ordered-before
    - carries-a-dependency-to
    - Only dependent data is synchronized
    - Subtle difference == hard to use

1.9. Custom Types with std::atomic

```cpp
struct SimpleType {
  int first;
  float second;
}; // ok
```

```

// some that do not work:
struct NonTrivialCCtor {
  NonTrivialCCtor(NonTrivialCCtor const&) {
    std::cout << "copied!\n";
  } // ERROR: can't create nontrivial constructor -> only trivial copy allowed
};

struct NonTrivialMember {
  int first;
  std::string second;
}; // strings can't be copied trivially -> not usable!!
```

- You can not have a custom copy constructor
- You can not have a custom move constructor
- You can not have a custom copy assignment operator
- You can not have a custom move assignment operator
- **Object can only be accessed as a whole**
  - no member access operator in std::atomic

2. Bibs and Bobs (volatile and Interrupts)

2.1. Volatile

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

- Semantics different from dotnot and Jafuck
- volatile prevents the compiler from optimizing the reads/write on this variable
- **Prevents reordering in the same thread *by the compiler!***
  - Hardware might still reorder instructions, can't be solved by software
- Useful when accessing memory-mapped hardware
  - **never use it for inter-thread communication!**
- Currently there are proposals to reduce/remove volatile from the language -> replacement with library functionality and cleanup semantics

Again, whatever you learned in Parprog, this is different in CPP, it is only a flag to tell the compiler to not optimize, it doesn to do anything for thread synchronization or blocking!

2.2. Interrupts

- Events coming from the OS/CPU
- Can be suppressed by the platform
- When an interrupt occurs, a previously registered function is called
  - such functions are called Interrupt Service Routines (ISRs)
  - ISRs should generally be short and must run to completion
- After the intterupt was handled, execution of the program resumes

2.2.1. Interrupts and Shared Data

- Data sharing between ISRs and normal programs need to be protected
  - All access must be atomic
  - Modiciations need to become visible
- Volatile helps regarding visibility
  - Supresses compiler optimizations -> makes sure read happens!
- Interrupts may need to be disabled temporarily to guarantee atomicity
- Refer to your hardware manual for specific details on how to deal with interrupts

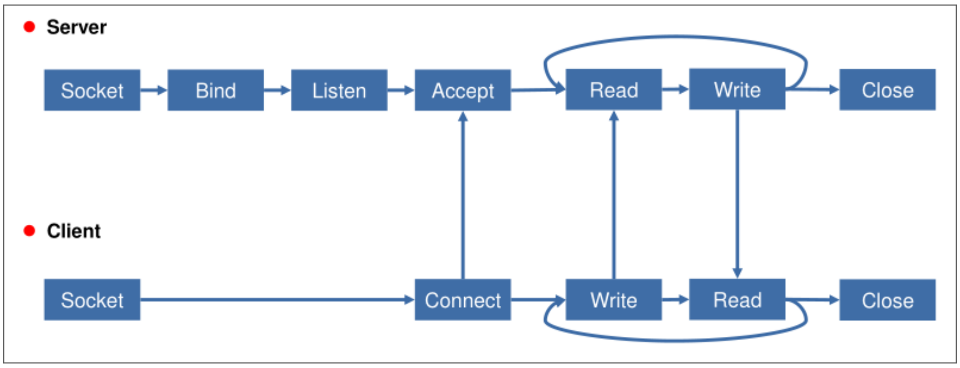
2.2.2. Interrupts Example

- On AVR-based Arduinos, interrupts cannot be interrupted
  - Other platforms support so-called Interrupt-Nesting (e.g. ARM, Risc V, …)
- **Before accessing shared data, interrupts must be disabled and enabled afterwards**
  - noInterrupts() – disable interrupts
  - interrupts() – enable interrupts
- **Interrupts sources can be "external", e.g.pins on the board**
  - check hardware manual..

```
constexpr byte ledPin = 13;
constexpr byte switchPin = 2;
volatile bool ledState = LOW;
void toggleLed() {
  ledState = !ledState;
}
void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(switchPin),
  toggleLed,
  CHANGE);
}
void loop() {
  noInterrupts();
  digitalWrite(ledPin, ledState);
  interrupts();
}
```

3. Network

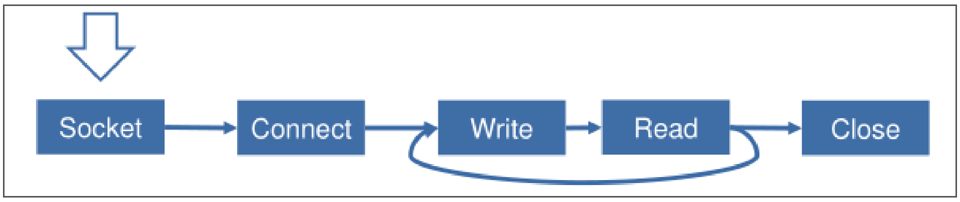
3.1. Sockets



3.2. Client example with TCP socket

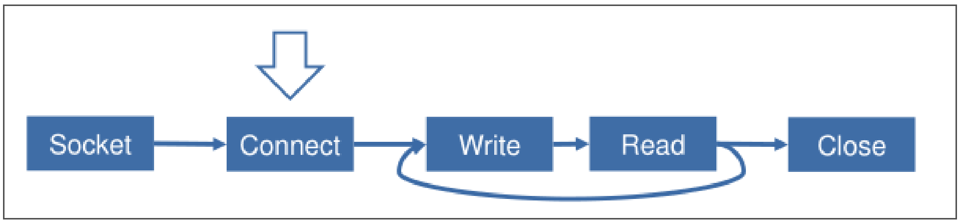
3.2.1. Creation of Socket

```
// create a context
// create a socket and add the context to it
asio::io_context context{};
asio::ip::tcp::socket socket{context};
```



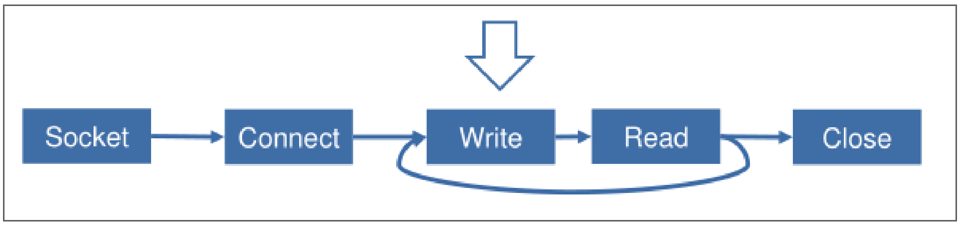
3.2.2. Connect to Socket

```
// with IP
auto address = asio::ip::make_address("127.0.0.1");
auto endpoint = asio::ip::tcp::endpoint(address, 80);
socket.connect(endpoint);

// with domain
asio::ip::tcp::resolver resolver{context};
auto endpoints = resolver.resolve(domain, "80");
asio::connect(socket, endpoints);
// we might have multiple answers here -> multiple DNS entries
```



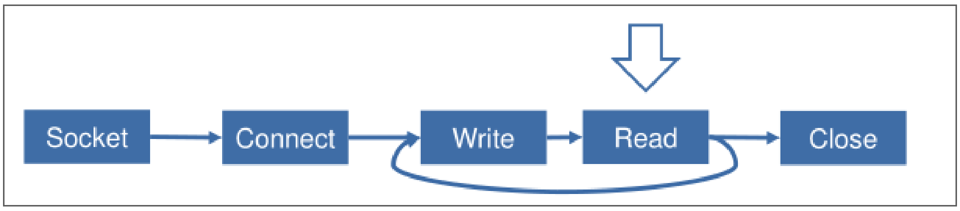
3.2.3. Write to socket

```
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()));
```



3.2.4. Read from socket

```
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);
```



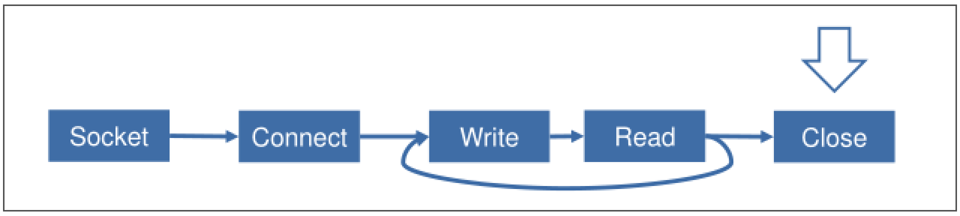
3.2.4.1. Advanced Read

- **asio::read also allows you to specify completion conditions**
  - asio::transfer_all: Default, transfers all available data or until buffer is full
  - asio::transfer_at_least(std::size_t bytes): Read at least x number of bytes, but may transfer more
  - asio::transfer_exactly(std::size_t bytes): self explanatory
- **asio::read_until allows you to specify conditions on the data being read**
  - simple matching of characters or strings
  - more complex matching using std::regex
  - allows you to specify a callable object -> expects std::pair<iterator,bool> operator()(iterator begin, iterator end)
  - **may read more! -> check the number of bytes returned!**

3.2.5. Close socket

- shutdown() closes the read/write stream associated with the socket, indicating to the peer that no more data will be received/sent.
- The destructor of the socket cancels all pending operations and destroys the object

```
socket.shutdown(asio::ip::tcp::socket::shutdown_both);
socket.close();
```



3.3. Data Sources/Buffers

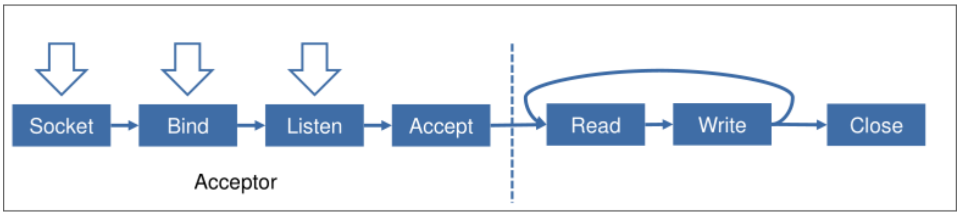
- **ASIO does not manage memory for you**
- **Fixed size buffers using asio::buffers**
  - must provide at least as much memory as you would like to read
  - Can use several standard containers as a backend
  - Pointer + Size combinations are also available
- Dynamically sized buffers using asio::dynamic_buffer()
  - for use with std::strng and std::vector -> heap
  - automatic resize as known by vector!
- Streambuf buffers using asio::streambuf
  - works with std::istream and std::ostream -> IO

3.4. Server example with TCP socket

3.4.1. Creating socket

- an acceptor is a special socket responsible for establishing incoming connections
- In ASIO the acceptor is bound to a given local endpoint and **starts listening automatically**

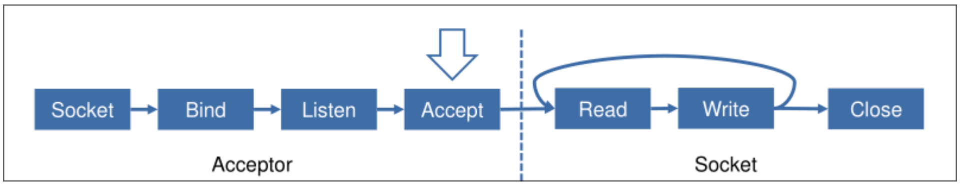
```
asio::io_context context{};
asio::ip::tcp::endpoint localEndpoint{asio::ip::tcp::v4(), port};
asio::ip::tcp::acceptor acceptor{context, localEndpoint};
```



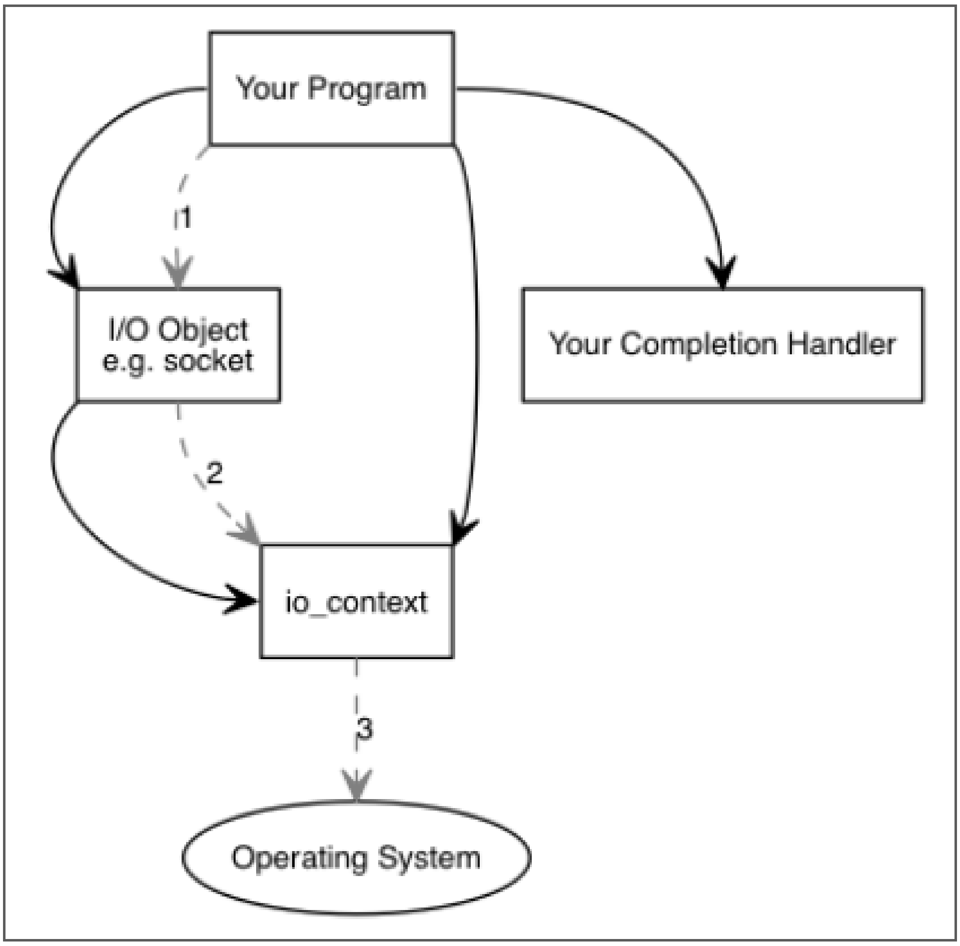
3.4.2. Accepting Connections

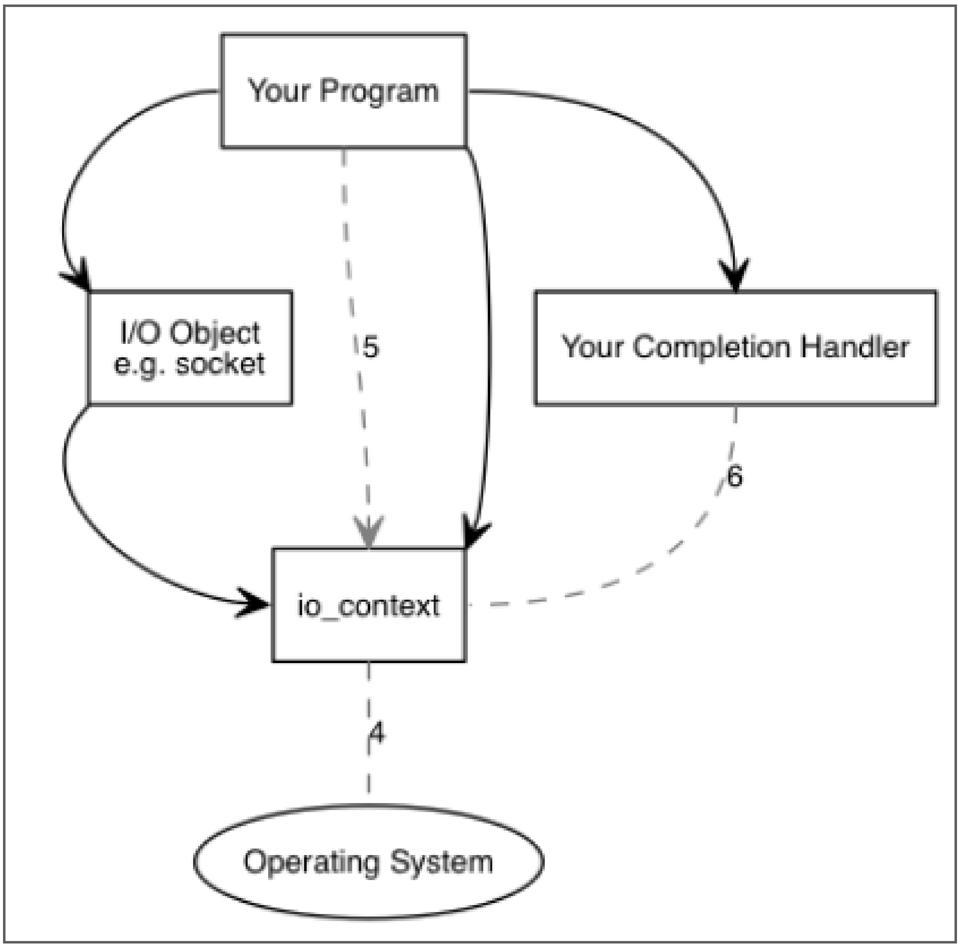
- The accept() member function blocks until a client tries to establish a connection
- it returns a new socket through which the connected client can be reached

```
asio::ip::tcp::endpoint peerEndpoint{};
asio::ip::tcp::socket peerSocket = acceptor.accept(peerEndpoint);
```



3.4.2.1. Handling multiple requests at once

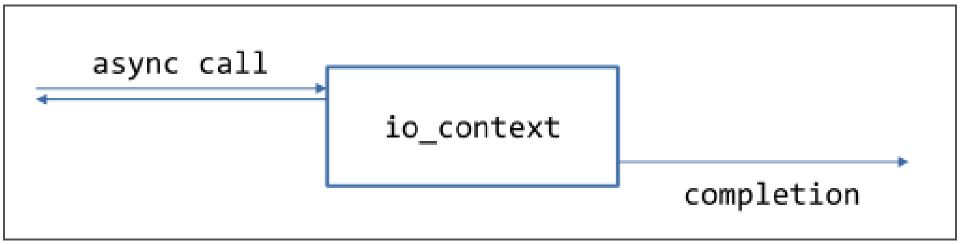




- The program invokes an async operation on an I/O object and passes a completion handler as a callback
- The I/O object delegates the operation and the callback to its io_context
- The operating system performs the asynchronous operation
- The OS singals the io_context that the operation has been completed
- When the program calls io_context::run() the remainin asynchronous operations are performed (wait for the result of he OS)
- Still inside io_context::run() the completion handler is called to handle the result (or error) of the asynchronous operation

3.5. Asynchronous Read/Write on Sockets

- Async read operations
  - asio::async_read
  - asio::async_read_until
  - asio::async_read_at
- Async write opertions
  - asio::async_write
  - asio::async_write_at
- These functions **return immediately**
- The operation is processed by the executor associated with the streams asio::io_context
- A completion handler is called when the operation is done



3.5.1. Example Asynchronous Read

- asio::async_read_until
  - reads from async stream into a buffer until specific character is encountered
  - then calls handler
  - completion handler takes asio::error_code and size_t as parameters

```cpp
auto readCompletionHandler = [] (asio::error_code ec, std::size_t length) {
  //...
};
asio::async_read_until(socket, buffer, '\n', readCompletionHandler);
```

3.5.2. Example Asynchronous Write

- asio::async_write writes data from a buffer to async stream until all data bas been written or error occurs
- then calls completion handler
  - completion handler takes asio::error_code and size_t as parameters

```cpp
auto writeCompletionHandler = [] (asio::error_code ec, std::size_t length) {
  //...
};
asio::async_write(socket, buffer, writeCompletionHandler);
```

3.5.3. Async Acceptor Overview

```cpp
struct Server {
  using tcp = asio::ip::tcp;
  Server(asio::io_context & context, unsigned short port)
        : acceptor{context, tcp::endpoint{tcp::v4(), port}}{
    accept();
  }
private:
  auto accept() -> void {
    auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
      if (!ec) {
        auto session = std::make_shared<Session>(std::move(peer));
        session->start();
      }
      accept();
    };
    acceptor.async_accept(acceptHandler);
    }
  tcp::acceptor acceptor;
};
```

3.5.4. Asynchronous Acceptor -> accept()

- creates 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 accepting new inbound connection attempts
- the accept handler is registered to handle the next accept asynchronously

```cpp
auto accept() -> void {
  auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
    if (!ec) {
      auto session = std::make_shared<Session>(std::move(peer));
      session->start();
    }
    accept();
  };
  acceptor.async_accept(acceptHandler);
}
```

3.5.5. Asychronous Acceptor (constructor)

```
Server(asio::io_context & context, unsigned short port)
      : acceptor{context, tcp::endpoint{tcp::v4(), port}}{
  accept();
}
```

- creates the sever
- initialized the acceptor with the given context and port
- calls accept for registering the accept handler for the next incoming conenction -> does not block

3.5.6. Asychronous Acceptor (Use)

```
auto main() -> int {
  asio::io_context context{};
  Server server{context, 1234};
  context.run();
}
```

- creates an io_context
  - has an associated executor that handles the async calls
- create the server on port 1234 (see above)
- Run the executor of the io_context until no async operation is left
  - Since we already have an async_accept request pending this operation does not return immediately
  - We will keep the this run() call busy
- **It is important that the server object llives as long as async operations on it are processed**

3.6. Sessions with Async IO

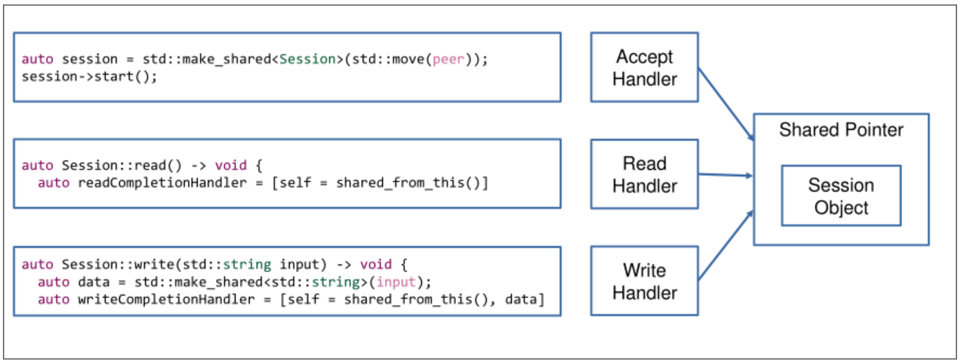
```
struct Session : std::enable_shared_from_this<Session> {
  explicit Session(asio::ip::tcp::socket socket);
  auto start() -> void {
    read();
  }
private:
  auto read() -> void;
  auto write(std::string data) -> void;
  asio::streambuf buffer{};
  std::istream input{&buffer};
  asio::ip::tcp::socket socket;
};
```

- constructor stored the client connection
- start(): initiates the first async read
- read(): invokes async reading
- write(): invokes async writing -> called by handler in read
- the fields store the data of the session
- enable_shared_from_this This is needed in order to keep the session alive -> e.g. lives until function is done via incrementing reference counting
  The session is defined as std::shared_ptr:

  ```
  if (!ec) {
  auto session = std::make_shared<Session>(std::move(peer));
  session->start();
  }
  ```

The handler then increments the reference counter:

```
//In the accept handler
void Session::read() {
auto handler = [self = shared_from_this()](error_code ec, size_t length) {
  // ..
}
```



3.7. Async Operations without Callbacks

- specify special objects as callbacks
- asio::use_future
  - returns std::future<T>
  - Errors are communicated via exception in future
- asio::detached
  - ignores the result of the operation -> fire and forget
- asio::use_awaitable (cpp 20, probably not usable)
  - returns asio::awaitable<T> which can be awaited in coroutine

3.8. Signal Handling

```
#include <asio.hpp>
#include <csignal>
#include <iostream>

auto main() -> int {
  auto context = asio::io_context{};
  auto signals = asio::signal_set{context, SIGINT, SIGTERM};
  signals.async_wait([&](auto error, auto sig) {
    if (!error) {
      std::cout << "received signal: " << sig << '\n';
    } else {
      std::cout << "signal handling aborted\n";
    }
  });
  context.run();
}
code
```

- waits for an event and executes completion handler specified
- note, you can't wait for SIGKILL and SIGSTOP, these actions will always happen

3.9. Accessing Shared Data

```
auto main() -> int {
  auto context = asio::io_context{};
  // start some async operations
  through asio::io_context
  auto runners = std::vector<std::thread>{};
  for(int i{}; i < 4; ++i) {
    runners.push_back(std::thread{[&]{
```

```
      context.run();
    }});
  }
  io_context.run()
  asio::io_context!
  for_each(runners.begin(), runners.end(),
           [](auto & runner){
            runner.join();
           });
}
```

- Multiple async operations can be in flight -> reading rom multiple sockets
- All completion handlers are dispatched through asio::io_context
  - handlers run on a thread executing io_context.run()
  - multiple threads can call run() on the same asio::io_context
  - **could lead to data races!**

```
// example for potential datarace
// globally accessible
auto results = std::vector<int> { };
// in connection class
asio::async_read(socket, asio::buffer(buffer), [&](auto err, auto bytes) {
auto result = parse(buffer);
results.push_back(result); // << data race here
});
```

3.9.1. Strands

ensure sequential execution of signal handlers

- implicit strands
  - if only one thread calls io_context.run()
  - our program blocks calling multiple handlers
- explicit strands
  - objects of type asio::strand<>
  - created using asio::make_strand(executor)
  - asio::make_strang(execution_context)
  - applied to handlers using asio::bind_executor(strand, handler)

```
// same code as above, now with strands -> data race resolved
// 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 err, auto bytes) {
auto result = parse(buffer);
results.push_back(result); // <<< No more data race
}));
```

4. Exception Safety Levels

4.1. Types of code with exceptions

4.1.1.1. Exception throws

This code will throw exceptions

4.1.1.2. Exception handling

This code will handle exceptions

4.1.1.3. Exception neutral

This code will neither handle nor throw exceptions, it will simply forward all exceptions.

4.2. Levels of Safety

- noexcept -> no throw This code will never throw an exception Note, the code might still have throws internally -> but it will also handle them directly aka it will not throw towards the caller
- strong exception safety: Operation succeeds and no exception is thrown, or nothing happens and an exception is thrown
- basic exceptin safety: does not leak resources in case of exception, but might not be complete -> not all operations done
- no guarantee You have to make sure there are no data leaks, dangling pointers etc.
- function is only as exception safe as it’s weakest link.

4.2.1. No Guarantee

```
auto & operator=(BoundedBuffer const & other) {
  if (m_container != other.m_container) {
    m_capacity = other.m_capacity;
    // what if this allocation throws?
    m_container = new char[sizeof(T) * m_capacity];
    m_position = 0;
    m_size = 0;
    for (auto const & element : other){
      this->push(element); // what if a copy throws?
    }
  }
  return *this;
}
code
```

- Don’t use this
- Invalid or corrupted data when an exception is thrown
- easy to implement, kekw, aka no handling

4.2.2. Basic Guarantee of an Operation

```
template<typename...TYPE>
static auto make_buffer(const int size, TYPE&&...param) -> BoundedBuffer<value_type> {
  int const number_of_arguments = sizeof...(TYPE);
  if (number_of_arguments > size)
    throw std::invalid_argument{"Invalid argument"};
    // the only safety
  BoundedBuffer<value_type> buffer{size};
  buffer.push_many(std::forward<TYPE>(param)...);
  return buffer;
}
```

- no resource leaks
- invariants are ok

However, the push_many() function might also throw -> aka the operation might not be completed

```
auto push_many() -> void { }
template<typename FIRST, typename...REST>
auto push_many(FIRST && first, REST&&...rest) -> void {
  push(std::forward<FIRST>(first));
  push_many(std::forward<decltype(rest)>(rest)...);
}
auto push(value_type const & elem) -> void {
```

```
  if(full()) throw std::logic_error{"full"};
    auto pointer = reinterpret_cast<value_type*>(dynamic_container_) + tail_;
  new (pointer) value_type{elem}; // might throw due to copy
  tail_ = (tail_ + 1) % (capacity() + 1);
  elements_++;
}
```

Note the throw, this means the element might not be pushed

4.2.3. Strong Guarantee

```
auto & operator=(BoundedBuffer const & other) {
  if (this != &other) {
    BoundedBuffer copy {other}; // might throw
    swap(copy); // mustn't throw
  }
  return *this;
}
```

Problem: What happens when both functions throw?
How to guarantee that at least one function does not throw?

- hard to guarantee a sequence
- might need undo functions if exception happens
- function that executes effect may not throw

4.2.4. No-Throw

```
auto std::vector<T>::empty() const noexcept -> bool;
auto std::vector<T>::size() const noexcept -> size_type;
auto std::vector<T>::capacity() const noexcept -> size_type;
auto std::vector<T>::data() noexcept -> T *;
// all iterator factories begin(), end()...
auto std::vector<T>::clear() noexcept -> void;
```

- memory allocations can't do noexcept -> might always fail hence pushback and popback can't be noexcept

4.2.5. Overview

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

4.3. No Except

- noexcept(false): default
- noexcept == noexcept(true) -> shorthand
- noexcept(function()): asks if the function is noexcept -> returns bool
- noexcept can't be overloaded

```
auto function() noexcept -> void {
  //...
}

template<typename T>
```

```cpp
auto function(T t) noexcept(<expression>) {
  // the noexcept here checks if the expression is noexcept -> does NOT actually run it!
  // then proceeds to set this functions noexcept based on the expression -> same except level!
}

auto main() -> int {
  std::cout << "is function() noexcept? " << noexcept(function()) << '\n';
  // returns bool whether or not function is noexcept
}
```

4.3.1. Explosive example

```cpp
template <unsigned ChanceToExplode>
struct Liquid;
using Nitroglycerin = Liquid<75>;
using JetFuel = Liquid<10>;
using Water = Liquid<0>;
template <typename Liquid>
struct Barrel {
  Barrel(Liquid && content): content{std::move(content)} {}
  auto poke() noexcept(noexcept(std::declval<Liquid>().shake())) {
    content.shake();
  }
private:
  Liquid content;
};
```

4.3.2. Members should not throw

- destructors...
- move constructors
- swap
- any sort of allocation or memory moving

4.3.3. std::move_if_noexcept

```cpp
template <typename T>
constexpr typename std::conditional<
!std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
const T&,
T&&
>::type move_if_noexcept(T & x);
```

is_nothrow_constructible	is_nothrow_move_constructible	is_nothrow_move_assignable
is_nothrow_default_constructible	is_nothrow_assignable	is_nothrow_destructible
is_nothrow_copy_constructible	is_nothrow_copy_assignable	is_nothrow_swappable

this moves the value if the type has a no except move operation -> otherwise copy

Example:

```cpp
template<typename T>
class _box {
  T value;
  public:
  explicit _box(T const &t) noexcept(noexcept(T(t))) :
  value(t) {}
  explicit _box(T && t) noexcept(noexcept(T(std::move_if_noexcept(t)))) :
  value(std::move_if_noexcept(t)) {}
  auto get() noexcept -> T &{
    return value;
  }
};
```

4.3.4. Contracts

C++ has no native support for contracts, meaning it is only software engineering thing.
- narrow contract
  - function expectes specific parameters -> may only be in range 50-100 etc
  - other preconditions
  - might not work properly if you don't make sure to pass the right stuff
- wide contract
  - function accepts any paramter with the correct type -> does internal checking
  - will never fail
  - no undefined behavior (if done properly)

Example:

```cpp
// wide contract
auto size() const _GLIBCXX_NOEXCEPT -> size_type {
  return size_type(this->_M_impl._M_finish - this->_M_impl._M_start);
}
// narrow contract:
explicit BoundedBuffer(size_type capacity): startIndex { 0 },
                                            nOfElements { 0 },
                                            capacity { capacity },
                                            values { allocate(capacity) } {
  if (capacity == 0) {
    throw std::invalid_argument { "size must be > 0." };
  }
}
```

4.3.5. Compiler and Noexcept

- compiler might optimize better with noexcept -> no stack unwinding preparation
- compiler will not warn you if you use exceptions with noexcept....
  - in this case std::terminate() will be called to avoid udef

```cpp
struct Ball {};
auto barrater() noexcept -> void {
  throw Ball{};
  // terminate called!!
}
auto main() try -> int {
    barrater();
  } catch(Ball const & b) {
    std::cout << "caught the ball!";
  }
}
```

5. Pimpl Idiom

5.1. Opaque types

These are types that were declared first, but without definition.
Then later the actual definition is used:

```cpp
struct S; //Forward Declaration
auto foo(S & s) -> void {
  foo(s);
  //S s{}; //Invalid
}
struct S{}; //Definition
auto main() -> int {
  S s{};
```

```
  foo(s);
}
```

The same is done in C with void * -> casting to the actual type later (unsafe, kek)

```
template<typename T>
auto makeOpaque(T * ptr) -> void * {
  return ptr;
}
template<typename T>
auto ptrCast(void * p) -> T * {
  return static_cast<T*>(p);
}
auto main() -> int {
  int i{42};
  void * const pi {makeOpaque(&i)};
  cout << *ptrCast<int>(pi) << endl;
}
```

5.2. Pointer to Implementation

This is used in order to avoid recompilatin for each file that uses the library.
C++ is a kappa language, which somehow recompiles the entire file if you use a library function directly.

```
class Wizard { // all magic details visible
std::string name;
MagicWand wand;
std::vector<Spell> books;
std::vector<Potion> potions;
auto searchForSpell(std::string const & wish) -> std::string;
Potion mixPotion(std::string const & recipe);
auto castSpell(Spell spell) -> void;
auto applyPotion(Potion phial) -> void;
public:
Wizard(std::string name = "Rincewind") :
name{name}, wand{} {}
auto doMagic(std::string const & wish) -> std::string;
  //...
};
```

In other words, if you change something here, then you will need to recompile all files that use wizard, as it has been seen as a change in that file.
To fix this, make the wizard a pointer wrapper to the implementation, hence the name pimpl.

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

// wizard.cpp
//Implementation of Wizard
Wizard::Wizard(std::string name):
pImpl{std::make_shared<WizardImpl>(name)} {}
auto Wizard::doMagic(std::string wish) -> std::string {
  return pImpl->doMagic(wish);
}

// class moved here
#include "Wizard.hpp"
#include "WizardIngredients.hpp"
#include <vector>
#include <algorithm>
class WizardImpl {
  std::string name;
```

```
  MagicWand wand;
  std::vector<Spell> books;
  std::vector<Potion> potions;
  auto searchForSpell(std::string const & wish) -> std::string;
  auto mixPotion(std::string const & recipe) -> Potion;
  auto castSpell(Spell spell) -> void;
  auto applyPotion(Potion phial) -> void;
  public:
  WizardImpl(std::string name) : name{name}, wand{}{}
  auto doMagic(std::string const & wish) -> std::string;
  //...
};
```

Since we moved the entire class into a cpp, we no longer include the file in all other files -> hence, we no longer need to recompile everywhere.

5.2.1.1. Defining size -> for unique pointer

The problem with unique pointer, is that it requires the type to be sized, which the type WizardImpl isn't at this point.
So we need to define a destructor for the WizardImpl:

```
// wizard.hpp
class Wizard {
  std::unique_ptr<class WizardImpl> pImpl;
  // size needs to be known here for wizardimpl
public:
  Wizard(std::string name);
  ~Wizard();
  auto doMagic(std::string wish) -> std::string;
};

// wizard.cpp
//class WizardImpl {
  //...
};
//...
Wizard::~Wizard() = default;
// this line makes it work
// no implicit destructor for wizard -> at this point size for wizard impl is known
// now you can let c++ auto define the destructor for the wizard wrapper
```

No Copying – Only Moving	`std::unique_ptr<class Impl>` • Declare destructor & `=default` • Declare move operations & `=default`
Shallow Copying (Sharing the implementation)	`std::shared_ptr<class Impl>`
Deep Copying (Default for C++)	`std::unique_ptr<class Impl>` • with DIY copy constructor (use copy constructor of Impl)

- pimpl should generally not be nullptr -> or rather not possible to be nullptr, should have used rust
- don't inherit from the pimpl class
  - not that I ever would since inheritance is crap.

6. Hourglass Interfaces

6.1. ABI

- name mangling
  - used for things like overloading
  - seen in c++ or rust
- calling conventions

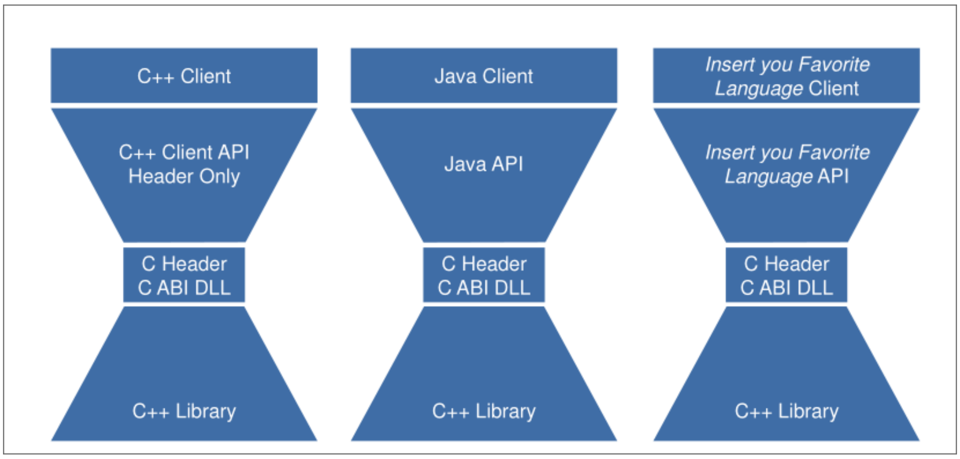
- instruction sets -> intel64, arm, risc-V
- c++ does not define a specific ABI as it is coupled to the platform
  - done by the compiler -> GCC/G++ for example
- ABIs change between OS, compiler, versions, library versions etc.

6.1.1. Comparison to C

- C also does not define a specific ABI
  - but it is more stable
    - no mangling
    - no namespaces
    - hence also no:
      - exceptions
      - templates
      - member functions on structs or classes

6.2. Idea

The general idea of the hourglass interface is that you have an ABI that will allow you to use any language in order to use libraries.
Aka you can use c++ libraries with rust if you so chose to, for whatever reason.



Here the C ABI, which is more stable than the c++ library is used, hence it should be expected to work.

6.2.1. 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();
  }
};
```

6.2.2. ABI

- represented by abstract pointers
- member functions are handled by taking the abstract pointer as first argument -> see python and rust
- requires factory and disposal functions to manage object lifetime
- strings are represented by char*

- exceptions do not work across C ABI
  - instead you can store the message of exceptions in parameters
  - errors need to be cleaned up when they are no longer used -> char* !

```
// define abstract wizard
typedef struct Wizard * wizard;
// const version
typedef struct Wizard const * cwizard;
// create function
wizard createWizard(char const * name, error_t * out_error);
void disposeWizard(wizard toDispose);
typedef struct Error * error_t;
char const * error_message(error_t error);
void error_dispose(error_t error);
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);
```

This can also be done in C++ with extern C:

```
#ifdef __cplusplus
extern "C" {
  #endif
  typedef struct Wizard * wizard;
  typedef struct Wizard const * cwizard;
  wizard createWizard(char const * name,
  error_t * out_error);
  void disposeWizard(wizard toDispose);
  // ...
  // Comments are ok too, as the preprocessor
  // eliminates them anyway
  #ifdef __cplusplus
}
#endif
```

- no overloading in extern
- only primitive types and pointers
- structs are not the same as in c++
- always forward declare structs!
- enums unscoped
- Note, not all things are unallowed, namespaces can be used, they will simply be put back into the global scope at compile time
  Aka in here, the compiler will make the code C compliant, therefore making sure the ABI stability is guaranteed.

6.2.3. Binding ABI to the library (Trampolin)

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

// wizard.hpp
namespace unseen {
  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();
    }
  };
```

```
}
```

This binds the implementation in C++ to the C ABI by using namespaces.
Aka c++ defines both without causing naming issues
The wrapper is just needed in order to use the c++ features like overloads and templates.
Hence we need a C compatible wrapper, which is just a constructor that will be bound to a struct in C.
Again note, it does not mean that you need to use straight C, just C ABI compatible code!

6.2.4. Dealing with Exceptions

- no references in C ABI -> pointer to pointer
- in case of error, allocate error value on heap -> string
  - provide disposal function as well
- no usage of c++ types
- it is safe to return char const * -> receiving function owns the error string
  - hence also needs to dispose of allocated memory!

6.2.4.1. Creating Error Messages

```
template<typename Fn>
bool translateExceptions(error_t * out_error, Fn && fn)
  try {
    fn();
    return true;
  } catch (const std::exception& e) {
    *out_error = new Error{e.what()};
    return false;
  } catch (...) {
    *out_error = new Error{"Unknown internal error"};
    return false;
}

wizard create_wizard(const char * name, error_t * out_error) {
  wizard result = nullptr;
  translateExceptions(out_error,[&] {
    result = new Wizard{name};
  });
  return result;
}
```

- translation from exceptions to bools etc. -> primitive types
- map the info to an error struct
- remember, no reference possible -> pointer to pointer (pointy point)

6.2.5. Client

6.2.5.1. Error Handling on the Client Side

```
// could be any language
// wizardclient.hpp
struct ErrorRAII {
  ErrorRAII(error_t error) : opaque {error} {}
  ~ErrorRAII() {
    if (opaque) {
        error_dispose(opaque);
    }
  }
  error_t opaque;
};

struct ThrowOnError {
  ThrowOnError() = default;
```

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

- map error codes back to exceptions if you would like to use exceptions
  - once again, exception types can't be mapped through! -> only messages
  - you could however use standard types, as you can store primitive types for resolution
- temporary object with destructor that can throw error ? wat
  - automatic type conversion passes adress of error content
  - make sure not to leak memory -> deallocation must succeed!

6.2.5.2. General Usage

```cpp
struct Wizard {
  // note the passing of the ThrowOnError
  // this will handle the storing of the error that can also propagate to the backend!
  Wizard(std::string const & who = "Rincewind") : wiz {createWizard(who.c_str(), ThrowOnError{})} {
  ~Wizard() {
    dispose_wizard(wiz);
  }
  auto doMagic(std::string const &wish) -> std::string {
    return ::do_magic(wiz, wish.c_str(), ThrowOnError{});
  }
  auto learnSpell(std::string const &spell) -> void {
    ::learn_spell(wiz, spell.c_str());
  }
  auto mixAndStorePotion(std::string const & potion) -> void{
    ::mix_and_store_potion(wiz, potion.c_str());
  }
  auto getName() const -> char const * {
    return wizard_name(wiz);
  }
private:
  Wizard(Wizard const &) = delete;
  Wizard & operator=(Wizard const &) = delete;
  wizard wiz;
};
```

6.2.5.3. DLL Hiding (Compiler Dependent)

```cpp
#define WIZARD_EXPORT_DLL [[gnu::visibility("default")]]
WIZARD_EXPORT_DLL
char const * error_message(error_t error);
WIZARD_EXPORT_DLL
void error_dispose(error_t error);
WIZARD_EXPORT_DLL
wizard create_wizard(char const * name,
error_t *out_error);
WIZARD_EXPORT_DLL
void dispose_wizard(wizard toDispose);
WIZARD_EXPORT_DLL
char const * do_magic(wizard w,
char const * wish,
error_t *out_error);
WIZARD_EXPORT_DLL
void learn_spell(wizard w, char const *spell);
WIZARD_EXPORT_DLL
void mix_and_store_potion(wizard w, char const *potion);
WIZARD_EXPORT_DLL
char const * wizard_name(cwizard w);
```

- use -fvisibility=hidden with GCC and clang in order to make library symbols hidden
  - see above: all symbols that should be visible must be marked as such
- can also be done with windoof: __declspec(dllexport)

6.3. Jafuck Natice Access (JNA)

- generates interfaces at runtime
- single jar
- cross platform -> compile once debug everywhere

6.3.1.1. Mappings

Native Type	Java Type
char	byte
short	short
wchar_t	char
int	int
bool (int)	boolean
long	NativeLong
long long (64-bit)	long
float	float
double	double
char *	String
some_type *	Pointer
struct xyz	Structure

6.3.1.2. Loading

```java
public interface CplaLib extends Library {
  CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);
}
```

- loader searches for suitable library -> libname.so libname.dylib libname.dll etc
  - first in path specified by jna.library.path
  - otherwise system default library path
  - fallback into classpath

6.3.1.3. Interfacing with Functions

```java
public interface CplaLib extends Library {
  CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);
  // function that is defined in ABI
  void printInt(int number);
}
```

- names and types must match!
  - see mappings!
- parameter names do not matter

6.3.1.4. Interfacing with non-opaque Structs

```cpp
extern "C" {
```

```
  struct Point {
    int x;
    int y;
  };
  void printPoint(Point point);
}
```

Can be translated to jafuck as follows:

```
// translation
public interface CplaLib extends Library {
  CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);
  public static class Point extends Structure implements Structure.ByValue {
    public int x, y;
    Point(int x, int y) {
      this.x = x;
      this.y = y;
  }
    @Override
    protected List<String> getFieldOrder() {
      return List.of("x", "y");
    }
  }
  void printPoint(Point point);
}

// USAGE
CplaLib.Point p = new CplaLib.Point(12, 90);
CplaLib.INSTANCE.printPoint(p);
```

- structs from C must override the getFieldOrder() function
  - this function is needed since jafuck reorders members as it wants, but in C it is fixed, this makes sure the mapping still works!
- can use tag-interface Structure.ByValue
- you can access references with getPointer -> remember, jafuck has a garbage collector, might be cleaned up -> nullptr

6.3.1.5. Interfacing with opaque Structs

```
extern "C" {
  typedef struct Unicorn * unicorn;
  unicorn createUnicorn(char * name);
  void disposeUnicorn(unicorn instance);
  void printUnicorn(unicorn unicorn);
}
```

This can be converted to jafuck as follows:

```
public interface CplaLib extends Library {
  CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);
  public static class Unicorn extends Pointer {
    Unicorn(String name) {
      super(Pointer.nativeValue(INSTANCE.createUnicorn(name)));
    }
    void dispose() {
      INSTANCE.disposeUnicorn(this);
    }
  }
  Pointer createUnicorn(String name);
  void disposeUnicorn(Unicorn instance);
  void printUnicorn(Unicorn unicorn);
}

// USAGE
CplaLib.Unicorn u = new CplaLib.Unicorn("freddy");
CplaLib.INSTANCE.printUnicorn(u);
u.dispose();
```

- for opaque structs, you should inherit from Pointer, wowie inheritance, fuck me
  - provide a constructor for this -> create()
- managing lifetimes is not trivial -> not automatic
  - provide a dispose method or implement AutoClosable and use it with try-with-resources (wat?)
  - using dispose() in finalizers is not recommended

6.3.1.6. Using Raw byte arrays

```
extern "C" {
  char * getData(int * size);
  void freeData(char * data);
}
```

Conversion to jafuck:

```
public interface CplaLib extends Library {
  CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);
  Pointer getData(IntByReference size);
  void freeData(Pointer data);
}

// USAGE
IntByReference size = new IntByReference();
Pointer data = CplaLib.INSTANCE.getData(size);
byte[] javaData = data.getByteArray(0, size.getValue());
CplaLib.INSTANCE.freeData(data);
for(byte b : javaData) {
  System.out.println(b);
}
```

- use IntByReference to retrieve the size of the buffer
  - requires that API supports this
- getByteArray() copies data from the buffer
- make sure to free the buffer (why even bother with java then……)
  - either with using free()
  - or Native.free()
    - apparently crashes on windoof because fuck you

7. Build Tools

7.1. Scripts

- easy to write
- platform dependent
- **builds each binary every time…**
- tend to become messy over time

7.2. Proper tools

list:
- cmake
- GNU make
- ninja
- meson
- Scons

Advantages:
- incremental builds

- parallel builds
- automatic dependency resolution (within project)
- package management
- automatic test execution
- platform independence
- additional processing of build products
  - something like signing

Note: not every tool will have all of the features above!

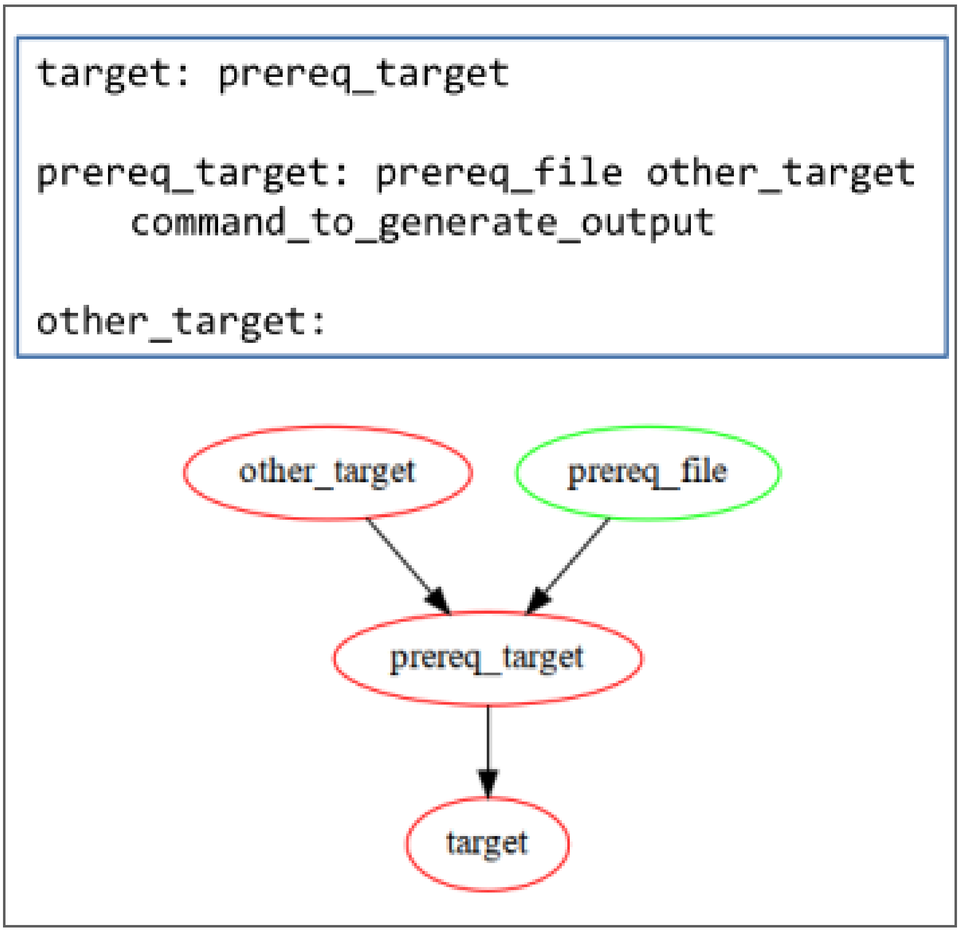
7.2.1. Classes of Tools

Make-style build tools:
- run build systems
- produce final product
- often verbose
- language agnostic config language -> works for C, c++, Jafuck etc.

build script generator:
- wrapper for Make
- automatic configuration (or at least tries to)
- downloading of packages and dependencies
- other advanced features

7.2.2. GNU Make

- understood by pretty much all IDEs
- can be used without an IDE as well
- rules via targets
  - each target can have prerequisites
  - executes one or more commands
  - generates one or more reuse
  - targets are executes top to bottom
  - target only executed if required, either dependency or manually called



Pros:
- very generic, runs everywhere
- powerful pattern matching mechanism
- builds only what is needed, and when it is needed

Negatives:
- often platform specific commands used inside targets
- you need to specify how to do things
  - can be quite a bit of code!
- may not feel native with some languages -> rather strange or unintuitive to use.

7.2.3. Build Script Generators

- define what to do now how
- work on a higher level
- lets you create actual build configurations
- platform independent build specification
- tool independent
  - can be used by IDEs or other editors
  - supports multiple build tools -> Make or more

7.2.3.1. Cmake

```
# sets minimum version
cmake_minimum_required(VERSION "3.12.0")
# set the name of the project and define language used
project("my_app" LANGUAGES CXX)
# add executables to build
```

```
add_executable("my_app"
  "main.cpp"
)
# add libraries to include
# THIS DEFAULTS TO STATIC LINKING!
# Use cmake -D BULD_SHARED_LIBS=YES for manual override to dynamic library!
# run this command once for configuration, then just build as usual
add_library("my_lib"
  "lib.cpp"
)
# add library to be included in the executable
target_link_libraries("my_app" PRIVATE "my_lib")
# set compiler to use
target_compile_features("my_app" PRIVATE
  "cxx_std_17"
)
// includes a directory
target_include_directories("test_runner" SYSTEM PRIVATE
"cute“
)
// adds a test
add_test("tests" "test_runner")
# set properties for targets
set_target_properties("my_app" PROPERTIES
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)
```

```
cmake -B build
cmake --build build
# for running the test
ctest --output-on-failure
```

- Public: functions from this can be used when using this entire project as a library e.g. re-export in rust
- private: functions from this can't be used when using this entire project as a library