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

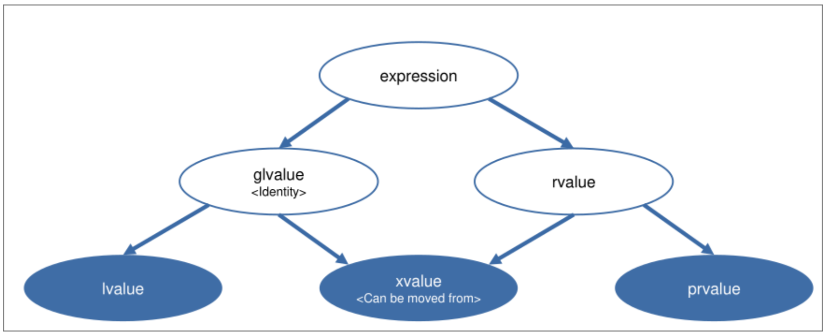
By default you can't just use an lvalue as an rvalue, however, you can use *std::move* to explicitly convert an lvalue to an rvalue.
Note that in this case, you *can't use the old variable anymore, as the data has been moved! -> see rust*

```
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

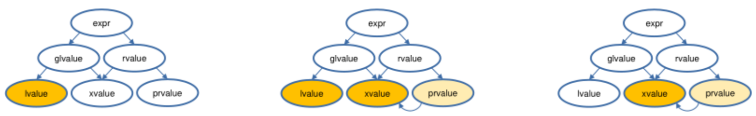
- lvalue reference made only of lvalues!!
  - type: T&
  - alias for a variable
  - can be used as function member type, local member/variable, return type
  - be aware of dangling references when returning!
- rvalue reference made of rvalues, prvalues or xvalues!
  - Type: T&&
  - when assigned to a 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	`T & create();` `create();`	lvalue
`int value{};` `std::cout << value + 1;`	rvalue	`T && create();` `create();`	rvalue
`auto foo(T& param) -> void {` `std::cout << param;` `}`	lvalue	`T value{};` `std::cout << value + 1;`	depends on +
`auto print(T&& param) -> void {` `std::cout << param;` `}`	lvalue	`T value{};` `T o = std::move(value);`	rvalue
`auto create() -> T;` `create();`	rvalue	`std::cout << "Hello";`	lvalue

- **lvalue Reference**
  - binds
- **const lvalue Reference**
  - binds
- **rvalue Reference**
  - binds




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

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

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

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

```
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

	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

Where you want to be

Avoid if possible

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>
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!

```
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

```
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

```
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:

```
// 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

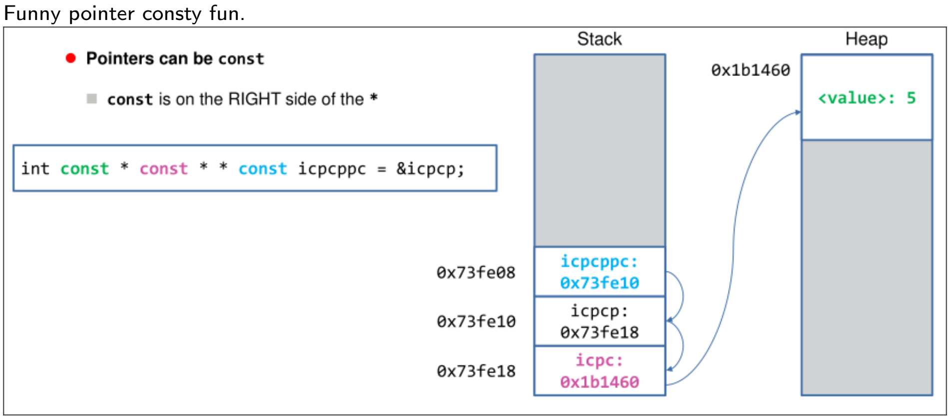
```
// lambda
int i0 = 42;
auto missingMutable = [i0] {return i0++;};

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

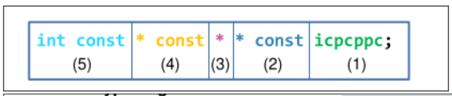
4 Memory Management and Heap

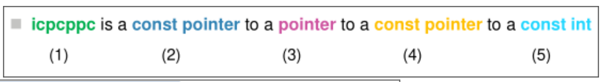
4.1 Pointers

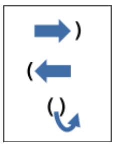
Funny pointer consty fun.

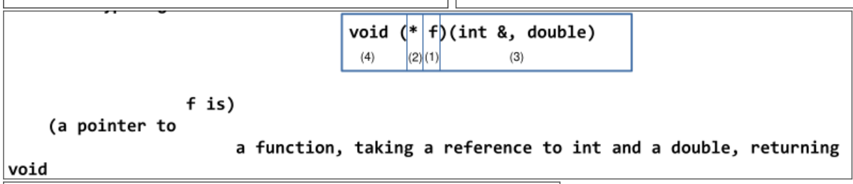


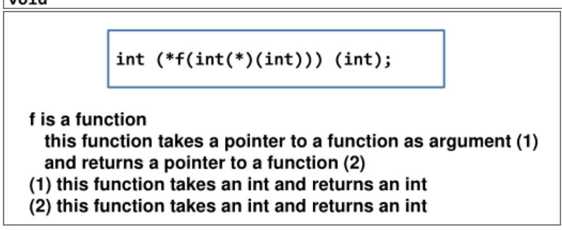
4.1.1 Reading a pointer declaration

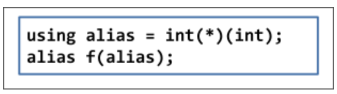












4.1.2 nullptr

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...

```
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

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

DANGER
Memory Leak

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

DANGER
Double Delete

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

DANGER
Invalid Access

```
auto bar() -> void;

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

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

```
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

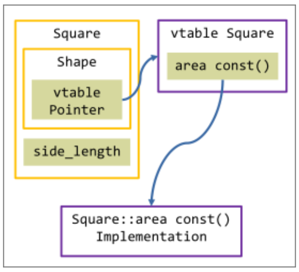
- faster at runtime
  no need to ckeck or cast function, just use it
- slower at compile time
  each implementation used will be made with macros
- syntax checking is off -> lsp limitation in c++
- larger binaries -> more code

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;
};
```



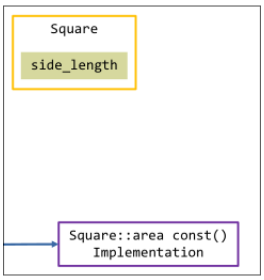
The problem is that we need to cast when using these functions. Once again you can see the shit that is inheritance as it forces this conveluted casting style of writing code.

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;
  }
  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.

```
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!

```
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

```
#include <type_traits>

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

```
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...

```
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:

```
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:

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

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

7.5.1 Problems with auto templates

```
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

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

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