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*

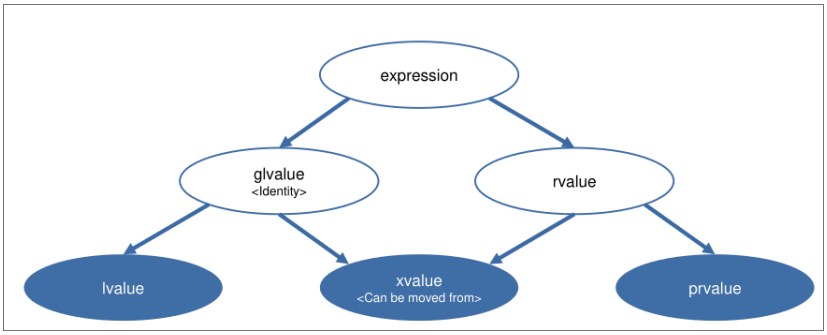
```
auto consume(Food&& food) -> void;

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

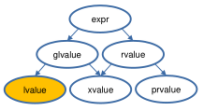
`auto foo(T& param) -> void {`
`  std::cout << param;`
`}`
	lvalue

`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

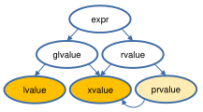
- **lvalue Reference**
  - binds




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

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

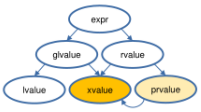
- **const lvalue Reference**
  - binds




```
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

2023_02_28_02_42_32.png

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"!!