Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 2 – Move Semantics

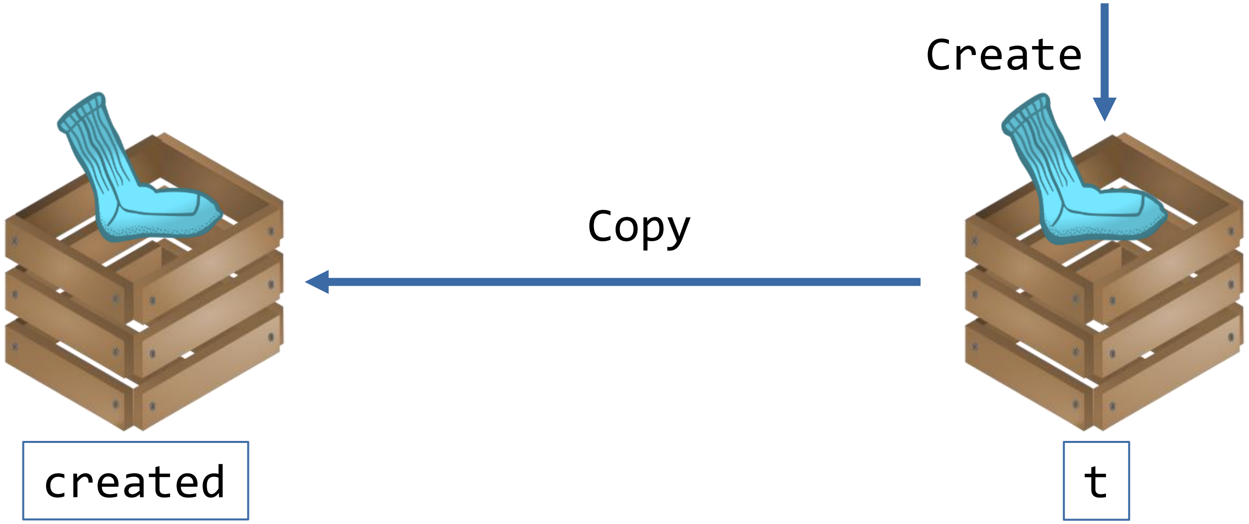
Thomas Corbat / Felix Morgner
Rapperswil / St. Gallen, 29.02.2024
FS2024

- **Topics:**
  - Motivation for Moving Elements
  - Rvalue References
  - Value Categories
  - Special Member Functions
  - Copy Elision

Goals

- **You can explain why move semantics exist**
- **You can apply the different types of references that are available in C++**
- **You know the value categories of C++**
- **You can determine the value category of an expression**
- **You can implement the special member functions that move objects**
- **You know what copy elision is and it is mandatory and when it is optional**

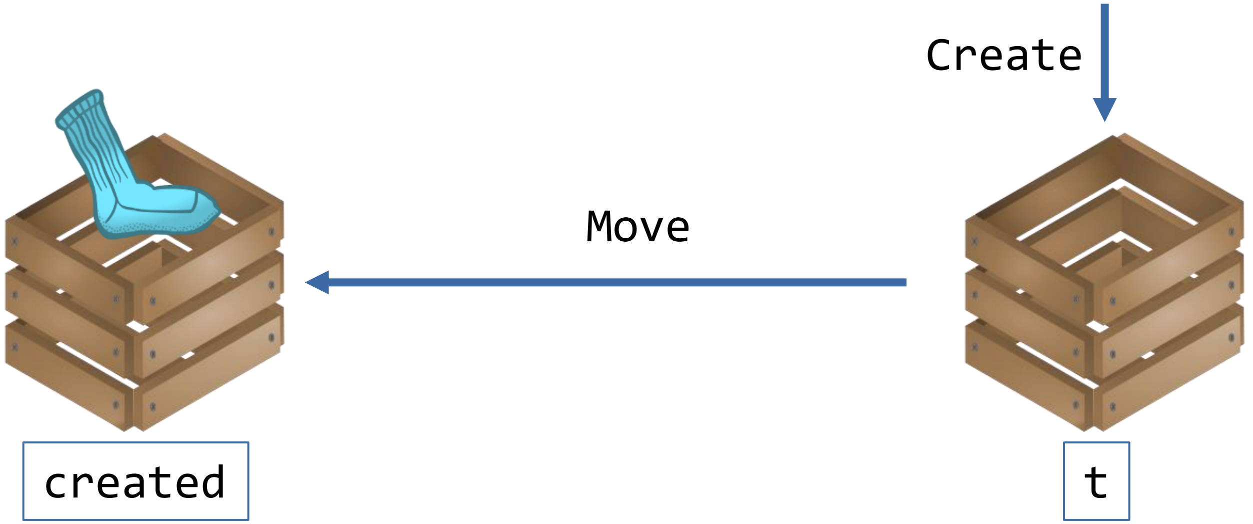
Motivation for Move Semantics




```
Create Thing
Copy Thing
```

```cpp
struct CopyableThing {
  CopyableThing() {
    std::cout << "Create Thing\n";
  }
  CopyableThing(CopyableThing const&) {
    std::cout << "Copy Thing\n";
  }
};

auto create() -> CopyableThing {
  CopyableThing t{};
  return t;
}

auto main() -> int {
  CopyableThing created = create();
}
```

Moving Elements




```
Create Thing
Move Thing
```

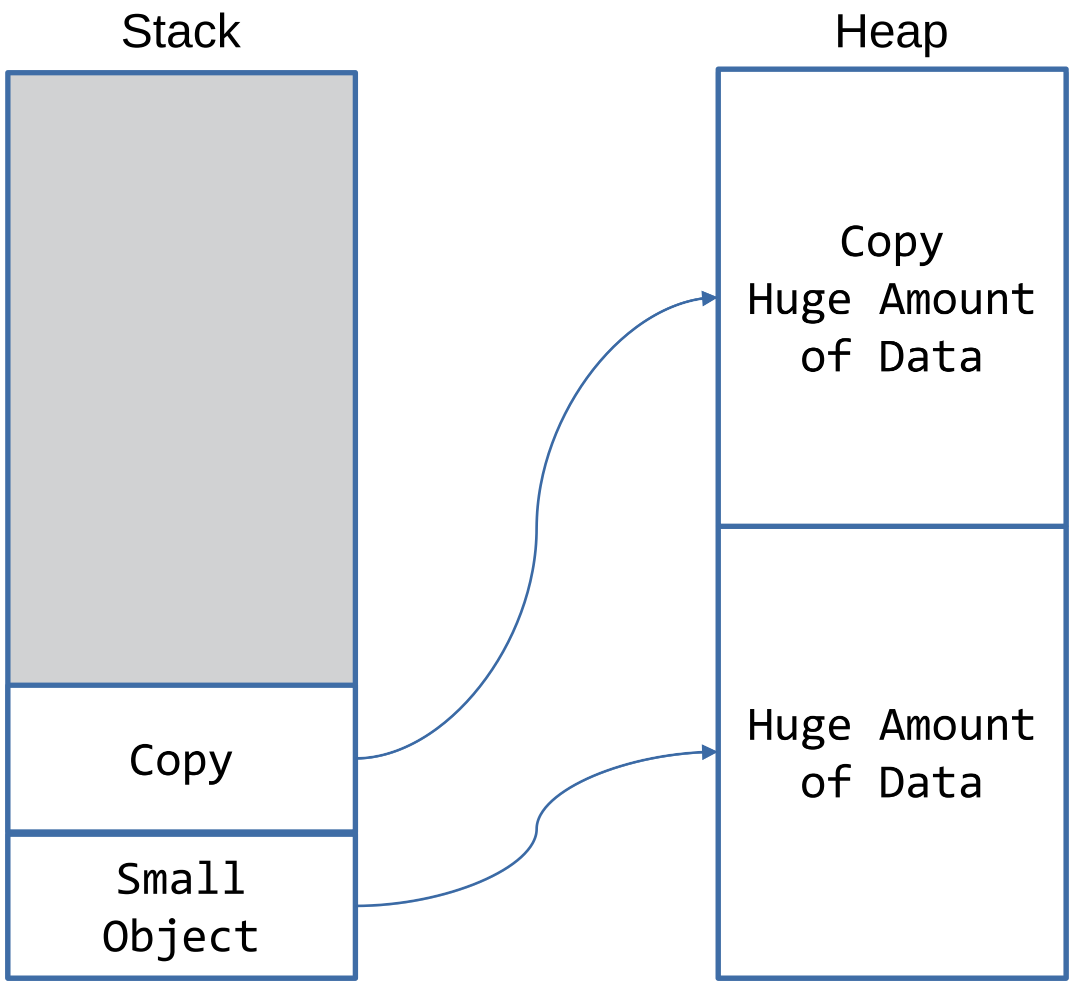
```cpp
struct MoveableThing {
  MoveableThing() {
    std::cout << "Create Thing\n";
  }

  MoveableThing(MoveableThing&&) {
    std::cout << "Move Thing\n";
  }
};

auto create() -> MoveableThing {
  MoveableThing t{};
  return t;
}

auto main() -> int {
  MoveableThing created = create();
}
```

- **Sometimes it is desirable to avoid copying values around for performance reasons**

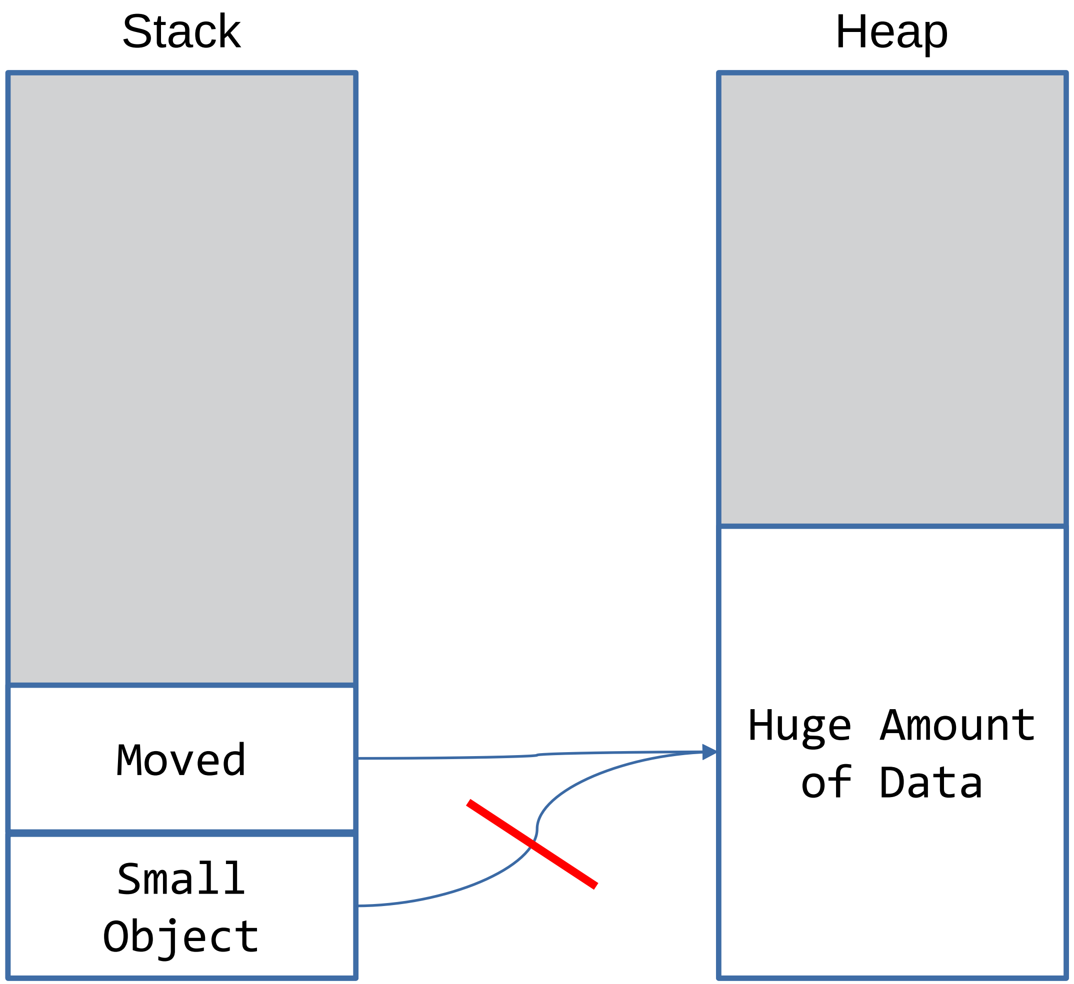
```
struct Planet {
  //gigantic type with data on heap
};

auto createPlanetsOfSolarSystem() -> std::vector<Planet> {
  std::vector<Planet> planets{};
  Planet earth{};
  planets.push_back(earth);         //copying a whole planet takes time
  planets.push_back(createMars()); //creates a mars temporary; copy that too?
  return planets;                  //copy everything once more?!
}
```

- **Is it really necessary to copy all those objects around?**

- **Copying**



- **Moving**



Example Move Constructor


```cpp
struct ContainerForBigObject {
  ContainerForBigObject()
      : resource{std::make_unique<BigObject>()} {}

  ContainerForBigObject(ContainerForBigObject const& other)
      : resource{std::make_unique<BigObject>(*other.resource)} {}

  ContainerForBigObject(ContainerForBigObject&& other)
      : resource{std::move(other.resource)} {}

  auto operator=(ContainerForBigObject const& other) -> ContainerForBigObject& {
    resource = std::make_unique<BigObject>(*other.resource);
    return *this;
  }
  auto operator=(ContainerForBigObject&& other) -> ContainerForBigObject& {
    using std::swap;
    swap(resource, other.resource);
    //resource = std::move(other.resource) is possible too
    return *this;
  }

private:
  std::unique_ptr<BigObject> resource;
};
```

- **Resources of expiring values can be transferred to a new owner**
  - This operation can be more efficient than creating a new (deep) copy and destroying the original
  - This operation might be feasible while a copy is not (e.g. `std::unique_ptr`)
- **Examples of such resources:**
  - Heap Memory
  - Locks
  - (File) Handles

Rvalue References

Ostschweizer
Fachhochschule

- **lvalue References**
  - Binds to an lvalue
  - Syntax: T &
  - The original must exist as long as it is referred to!

```
auto modify(T& t) -> void {
  //manipulate t
}

auto lvalueRefExample() -> void {
  T t = 5;
  modify(t);
  T& ir = t;
  //...
}
```

- **rvalue References**
  - Binds to an rvalue
  - Syntax: T &&
  - Can extend the life-time of a temporary

```
auto createT() -> T;

auto consume(T&& t) -> void {
  //manipulate t
}

auto rvalueRefExample() -> void {
  consume(T{});
  T&& t = createT();
  //...

}
```

References in C++: Recap Lvalue References

- **An lvalue Reference is an alias for a variable**
  - Binds an lvalue
  - Syntax: `T &`
  - The original must exist as long as it is referred to!
- **Can be used as**
  - Function parameter type (most useful: no copy and side-effect on argument possible)
  - Member or local variable (barely useful)
  - Return type (Must survive!)

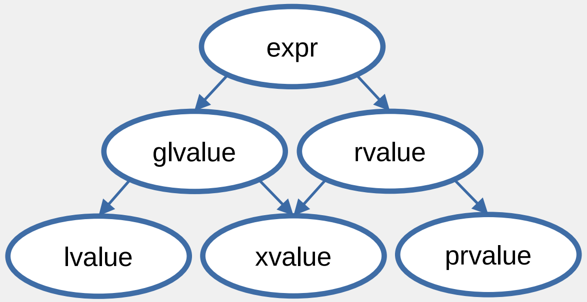
```
auto increment(int& i) -> void {
  ++i; // side-effect on argument
}
```

- Beware of dangling references: undefined behavior!

- **References for rvalues**
  - Binds only rvalues
  - Syntax: `<Type> &&`
- **Argument is either a literal, a temporary object or an explicitly converted lvalue**

```
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
}
```

Value Categories



- **CPL**
  - lvalue: expression on the left-hand side of an assignment (memory location)
  - rvalue: expression on the right-hand side of an assignment (value)
- **C++**
  - A little more complicated
  - lvalue: has identity
  - rvalue: does not have identity (temporaries and literals)
- **Example why lvalue does not always mean “can be on the left-hand side of an assignment”**

```
int a = 0;
a = 7; //a is an lvalue
       //ok
```

```
int const a = 0;
a = 7; //a is still an lvalue
       //not ok, const lvalue
```

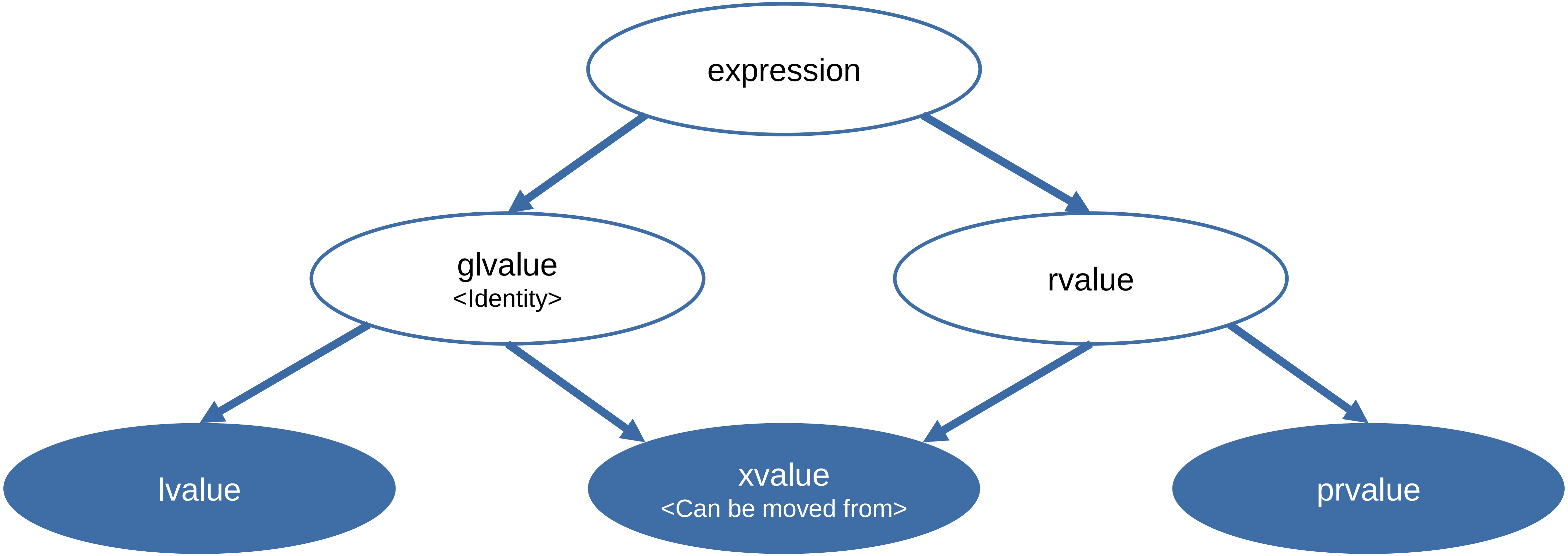
- **Example why rvalue does not always mean “cannot be on the left-hand side of an assignment”**
  - Not useful, but valid. `S{}` clearly is a temporary

```
#include <iostream>
#include <string>

struct S {
  auto operator=(std::string const& s) -> S& {
    std::cout << "got \"" << s << "\" assigned\n";
    return *this;
  }
};

auto main() -> int {
  S{} = "new value";
}
```

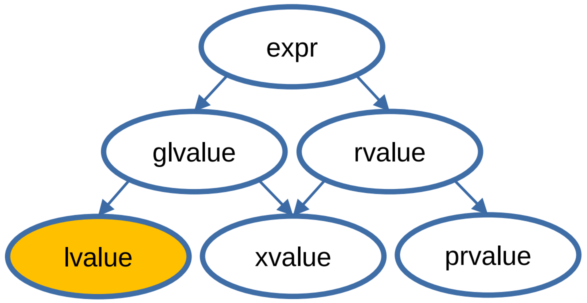
- **Every expression has**
  - (non-reference) Type
  - Value Category

- **Properties of a Value Category**
  - has identity (address can be taken)
  - can be moved from

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)


expression
glvalue
<Identity>
rvalue
lvalue
xvalue
<Can be moved from>
prvalue

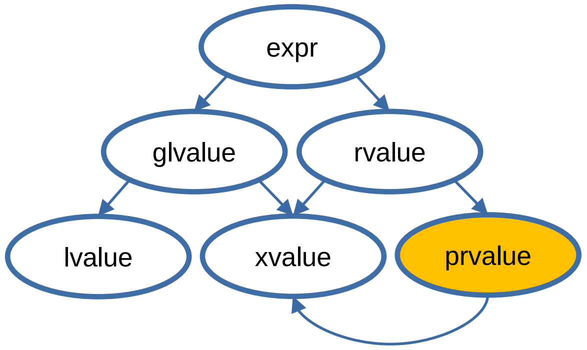
lvalues (has identity / cannot be moved from)



- **Address can be taken**
- **Can be on the left-hand side of an assignment if modifiable (i.e. non-`const`)**
- **Can be used to initialize an lvalue reference**
- **Examples**
  - Names of variables and parameters (`counter`)
  - Function call with return type of lvalue reference to class type (`std::cout << 23`)
  - Built-in prefix increment/decrement expressions (`++a`)
  - Array index access (`arr[0]`), wenn arr is an `lvalue`
  - All string-literals by definition (`"name"`)
    - This does not include user-defined (string) literals, like `"name"s` or `"name"sv`

prvalues (does not have identity / cannot be moved from)

- **Name: pure rvalue**
- **Address cannot be taken**
- **Cannot be left-hand side argument of built-in assignment operators**
- **Temporary materialization when a glvalue is required**
  - Conversion to xvalue
- **Examples:**
  - Literals: `23`, `false`, `nullptr`, …
  - Function call expression of non-reference return type: `int std::abs(int n)`
  - Several operators for built-in types, like post-increment/-decrement expressions: `x++`



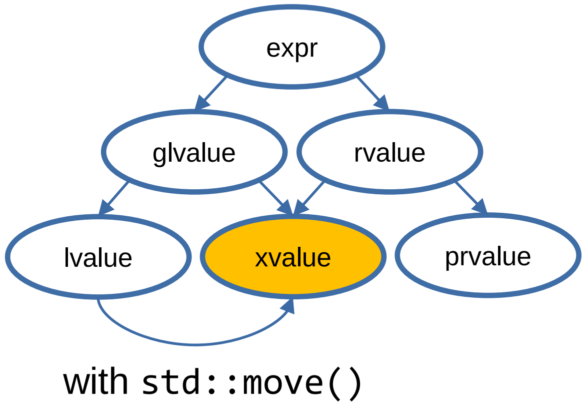
temporary materialization

- **Getting from something imaginary to something you can point to**
- **Prvalue to xvalue conversion happens...**
  - ... when binding a reference to a prvalue (1)
  - ... when accessing a member of a prvalue (2)
  - ... when accessing an element of a prvalue array
  - ... when converting a prvalue array to a pointer
  - ... when initializing an `std::initializer_list<T>` from a braced-init-list
- **Requires type to be complete and 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();    (1)

  Ghost{}.haunt(); (2)
}
```

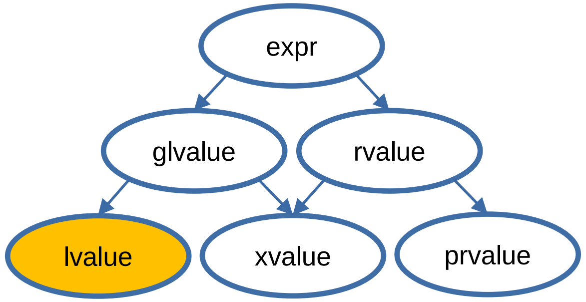
xvalues (has identity / can be moved from)

- **Name: expiring value**
- **Address cannot be taken**
- **Cannot be used as left-hand operator of built-in assignment**
- **Conversion from prvalue through temporary materialization**
- **Examples:**
  - Function call with rvalue reference return type, like std::move: `std::move(x)`
  - Access of non-reference members of an rvalue object
  - Array index access (`arr[0]`), wenn arr is an `rvalue`




```
X x1{}, x2{};
consume(std::move(x1));
std::move(x2).member;
X{}.member;
```

- **An lvalue Reference is an alias for a variable**
  - Syntax: `T &`
  - The original must exist as long as it is referred to!

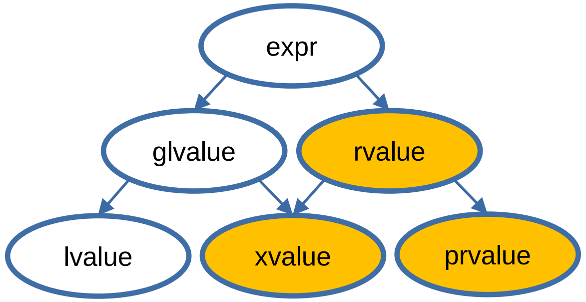


- **Can be used as**
  - Function parameter type (most useful: no copy and side-effect on argument possible)
  - Member or local variable (barely useful)
  - Return type (Must survive!)

```
auto increment(int& i) -> void {
  ++i; // side-effect on argument
}
```

- Beware of dangling references: undefined behavior!

- **References for rvalues**
  - Syntax: `T &&`
  - Binds to an rvalue (xvalue or prvalue)



- **Argument is either a literal or a temporary object**

```
std::string createGlass() -> std::string;

auto fancyNameForFunction() -> void {
  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
}
```

- **Beware: Parameters and variables declared as rvalue references are lvalues in the context of function bodies! (Everything with a name is an lvalue)**
- **Beware 2.0: `T&&/auto&&` is not always an rvalue reference! (We'll come to that later)**

?

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

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

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

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

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

lvalue or rvalue?

?

```cpp
T value{};
std::cout << value;
```

lvalue

```cpp
int value{};
std::cout << value + 1;
```

rvalue

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

lvalue

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

lvalue

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

rvalue

```cpp
T& create();
create();
```

lvalue

```cpp
T&& create();
create();
```

rvalue

```cpp
T value{};
std::cout << value + 1;
```

depends on +

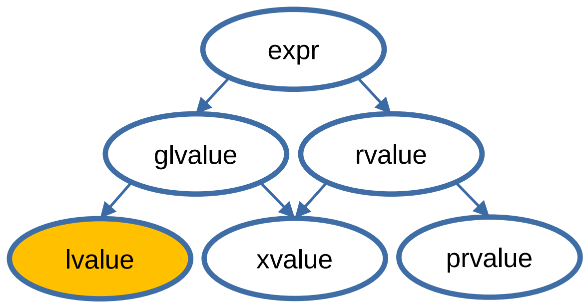
```cpp
T value{};
T o = std::move(value);
```

rvalue

```cpp
std::cout << "Hello";
```

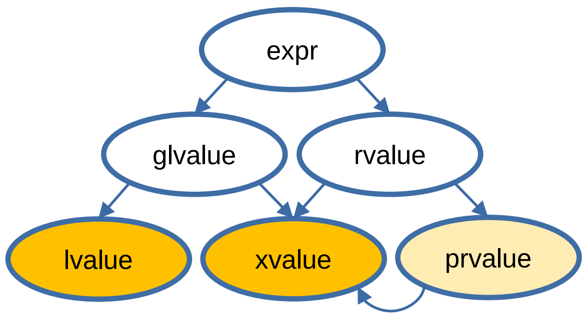
lvalue

- **lvalue Reference**
  - binds



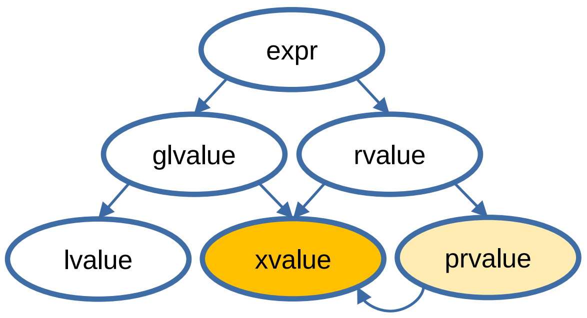

```
auto f(Type&) -> void;

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 &)

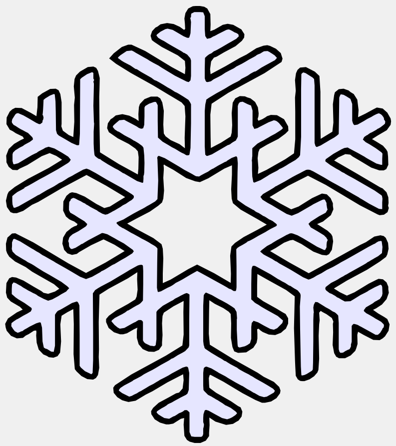
- **The overload for value parameters imposes ambiguities.**
- **For deciding between two lvalue reference overloads (const and non-const) the constness of the argument is considered.**

Overload Resolution for Member Functions

	`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 &)

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

Special Member Functions



Ostschweizer Fachhochschule

- **Constructors**
  - Default Constructor
  - Copy Constructor
  - Move Constructor
- **Assignment Operators**
  - Copy Assignment
  - Move Assignment
- **Destructor**
- **Advice: If possible, design your types in way that the default implementations work for them. Library developers might need to implement custom special member functions.**

```
struct S {
  S();
  ~S();
  S(S const&);
  auto operator=(S const&) -> S&;
  S(S&&);
  auto operator=(S&&) -> S&;
};
```

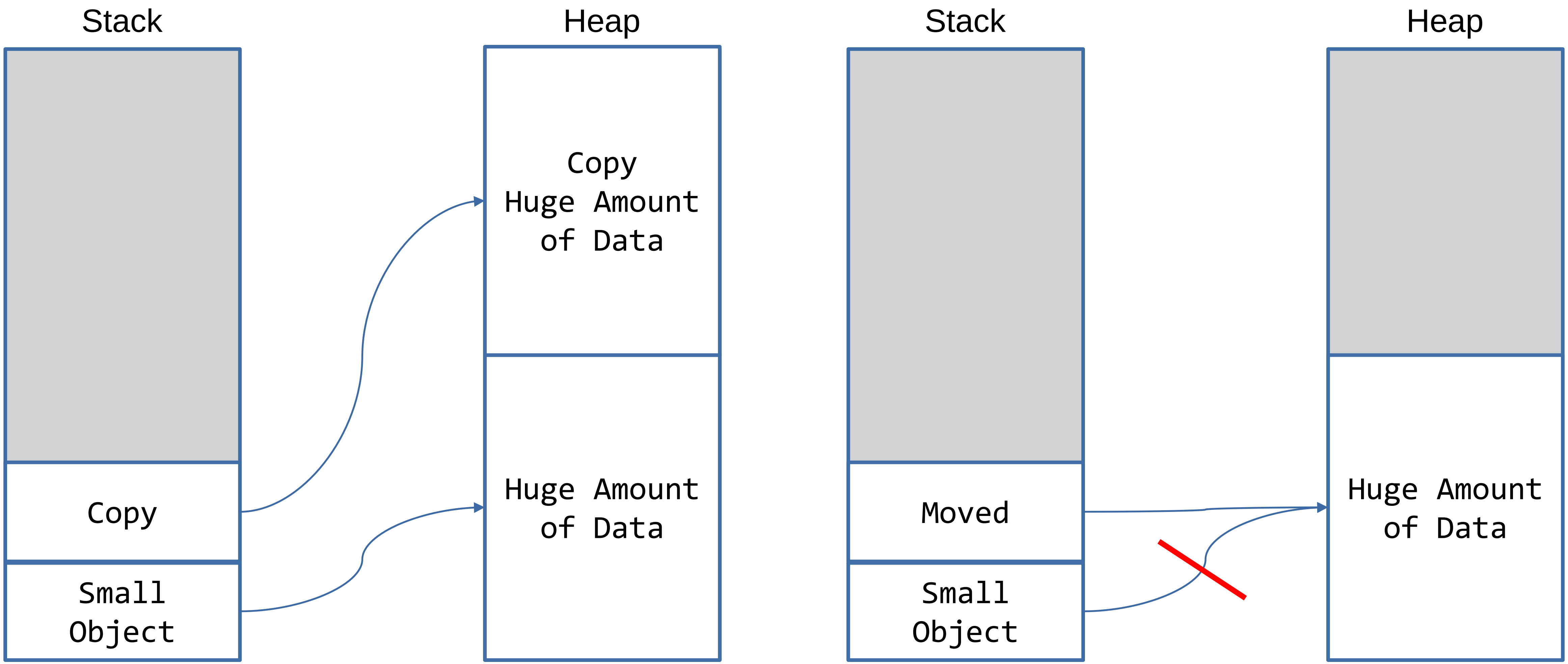
- **Move Constructor (Since C++11)**
  - Takes the entrails out of the argument and moves them to the constructed object
    - Leaves argument in valid but indeterminate state
    - Don't use the argument after it has been moved from until you assign it a new value
  - Signature: && parameter of the same type
  - Default Behavior (implicit or `=default`)
    - Initializes base-classes and members with move-initialization

`S(S &&)`

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

```
auto f(S param) -> void {
  S local{std::move(param)};
  // don't use param until
  // param = ...
}
```

- **Copy-Constructor**
- **Move-Constructor**



- **Copy Assignment Operator** `S & operator=(S const &)`
  - Copies the argument into the `this` object
  - Assignment Operator with `const &` parameter of the same type
  - Default Behavior (implicit or `=default`)
    - Initializes base-classes and members with copy-assignment

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

```
auto f(S param) -> void {
  S local{};
  local = param;
  ...
}
```

- **Move Assignment Operator** `S & operator=(S &&)`
  - Takes the entrails out of the argument and moves them to the `this` object
    - Leaves argument in valid but indeterminate state
    - Don't use the argument after it has been moved from until you assign it a new value
  - Assignment Operator with && Parameter of the same Type
  - Default Behavior (implicit or `=default`)
    - Assigns base-classes and members with move-assignment

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

```
auto f(S param) -> void {
  S local{};
  local = std::move(param);
  ...
}
```

- **Destructor** **`~S()`**
  - Deallocates resources held by the `this` object
  - Signature: `~<Class-Name>()`
  - No Parameters
  - Default Behavior (implicit or `=default`)
    - Calls destructor of base-classes and members
  - Must not throw exceptions! (is `noexcept`)

```
struct S {
  ~S() noexcept {…}
  M member;
};
```

```
<usually, you will not call
destructors explicitly>
Happens at end of scope: }
```

Exception in Destructor

- **Assignment operators must be member functions**
- **Move operations must not throw exceptions**
  - They shall not allocate new memory
  - Otherwise `std::swap` won't work reliably
  - More on the topic of exception guarantees later
- **Use the default implementation whenever possible**

```
struct S {
  auto operator=(S&& s) noexcept -> S&;
  S(S&& other) noexcept;
};
```

```
struct S {
  S() = default;
  ~S() = default;
  S(S const&) = default;
  auto operator=(S const&) -> S& = default;
  S(S&&) = default;
  auto operator=(S&&) -> S& = default;
};
```

What Special Member Functions Do You Get?

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	<u>user declared</u>	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	<u>user declared</u>	defaulted (!)	defaulted (!)	not declared	not declared
copy constructor	not declared	defaulted	<u>user declared</u>	defaulted (!)	not declared	not declared
copy assignment	defaulted	defaulted	defaulted (!)	<u>user declared</u>	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	<u>user declared</u>	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	<u>user declared</u>

Where you want to be

Avoid if possible

Howard Hinnant's Table: https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf
Note: Getting the defaulted special members denoted with a (!) is a bug in the standard.

- There are three different kinds of expression types in C++ (lvalue, xvalue, prvalue)
- The compiler must omit certain copy and move operations related to initialization from prvalues
- Objects/values can be copied, moved or passed by reference
- Good read about rvalue references and move semantics (state pre C++17): http://thbecker.net/articles/rvalue_references/section_01.html
- Interesting talk about the problems with move semantics (by Nicolai Josuttis): https://www.youtube.com/watch?v=PNRju6_yn3o

Implementing Assignment

Self-Study

Ostschweizer Fachhochschule

- **The C++ Core Guidelines recommend providing a `swap()` member function and free function for value-like types (C.83)**
  - This enables the Copy-Swap-Idiom
  - `using std::swap` allows to fall back on the `std::swap` implementation if no user defined `swap()` is available for the member

```
struct S {
  auto swap(S& other) noexcept -> void {
    using std::swap;
    swap(member1, other.member1);
    //…
  }
  //…
};

auto swap(S& lhs, S& rhs) noexcept -> void {
  lhs.swap(rhs);
}
```

- **Usually you don’t need to do this at all, but if you have to following this pattern is usually recommended**
  - Avoid self-copy
  - Use the copy constructor to create the copy of the argument
  - Swap the `this`-object with the copy (swapping is expected to be efficient)
  - Copy-Swap-Idiom

```
struct S {
  auto operator=(S const& s) -> S& {
    if (std::addressof(s) != this) {
      S copy = s;
      swap(copy);
    }
    return *this;
  }
  //…
};
```

- **Usually you don’t need to do this at all**
- **If you have to, following this pattern is usually recommended**
  - Avoid self-move
  - Swap the `this`-object with the parameter (swapping is expected to be efficient)

```
struct S {
  auto operator=(S&& s) -> S& {
    if (std::addressof(s) != this) {
      swap(s);
    }
    return *this;
  }
  //…
};
```

Copy Elision

```
struct S {
  S(S const& s) {
    //Why is this not called?!
  }
};
```

Self-Study

Ostschweizer Fachhochschule

- **In some cases the compiler is required to elide (omit) specific copy/move operations (regardless of the side-effects of the corresponding special member functions!)**
  - The omitted copy/move special member functions need not exist
  - If they exist, their side-effects are ignored
- **In initialization, when the initializer is a prvalue**
  - `S{}` is materialized in `s`
- **When a function call returns a prvalue (simplified)**
  - `S{}` is materialized in `new_sw`
  - `S{}` is materialized at the memory location return by `new`
    We will cover explicit memory management later

```
S s = S{S{}};
```

```
auto create() -> S {
  return S{};
}

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

- **In some cases the compiler is allowed to further optimize specific copy/move operations (regardless of the side-effects of the corresponding special member functions!)**
  - Named return value optimization

```
auto create() -> S {
  S s{};
  return s;
}

auto main() -> int {
  S s{create()};
  s = create();
}
```

- **The constructors must still exist – even if they are elided.**

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

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

Disabled elision (C++14):
`-fno-elide-constructors`

```
 --- S s{create()} ---
Constructor S()
 --- create() ---
Constructor S(S&&)
Constructor S(S&&)
 --- s = create() ---
Constructor S()
 --- create() ---
Constructor S(S&&)
operator =(S&&)
```

Disabled elision (C++17):
`-fno-elide-constructors`

```
 --- S s{create()} ---
Constructor S()
 --- create() ---
Constructor S(S&&)

 --- s = create() ---
Constructor S()
 --- create() ---
Constructor S(S&&)
operator =(S&&)
```

With elision (C++17):

```
 --- S s{create()} ---
Constructor S()
 --- create() ---


 --- s = create() ---
Constructor S()
 --- create() ---

operator =(S&&)
```

- **In throw expressions (Since C++11)**

```
try {
  throw S{7};
} catch (...) {
}
```

- **In catch clauses (Since C++11)**

```
try {
  throw S{7};
} catch (S s) {
}
```

- **Beware: The compiler is allowed to change observable behavior with this optimization!**
- **To be sure to avoid copies still catch by `const &`**

- **Is the following a good idea?**

```
auto create() -> S {
  S s{};
  return std::move(s);
}
```

- **While it sounds not that bad it prevents copy elision**

```
auto create() -> S {
  S s{};     //ctor
  return s;
}

auto foo() -> void {
  auto s = create();
} //dtor
```

```
auto create() -> S {
  S s{};     //ctor
  return std::move(s); //move ctor
} //dtor

auto foo() -> void {
  auto s = create();
} //dtor
```

Rules for Copy Elision

- **NRVO (Named Return Value Optimization)**
  - Return type is value type
  - Return expression is a local variable (more or less) of the return type
    - const is ignored for the type comparison
  - The object is constructed in the location of the return value (instead of moved or copied)
- **throw Expression**
  - Return expression is a local variable (more or less) from the innermost surrounding try block (if any)
  - The object is constructed in the location where it would be moved or copied
- **catch Clause**
  - If the caught type is the same as the object thrown, it access the object directly (as if caught by reference)
    - Must not change the observed behavior (except constructors/destructors)

Life-Time Extension

Self-Study

Ostschweizer Fachhochschule

- **Life-time of a temporary can be extended by "`const` lvalue reference" or "rvalue reference"**
- **Extended life-time ends at the end of the block**

```
struct Demon { /*...*/ };

auto summon() -> Demon{
  return Demon{};
}

auto countEyes(Demon const&) -> void { /*..*/ }

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

- **Extension of life-time is not transitive**

DANGER

Dangling References

```
auto bloodMagic() -> Demon const& {
  Demon breknok{};
  return breknok;
} //When blood_magic ends, breknok dies and will stay dead. All access will be Undefined Behavior!

auto animate(Demon const& demon) -> Demon const& {
  /*...*/
  return demon;
}

auto main() -> int {
  Demon const & breknok = blood_magic();     //You cannot keep demon from blood_magic alive!
  // -> Access to breknok would be Undefined Behavior
  Demon const & knoorus = animate(Demon{}); //You cannot keep demon passed through animate alive!
  // -> Access to knoorus would be Undefined Behavior
}
```