Contents

1 Move Semantics

What is move semantics? Move Semantics is when you do not copy an object, but you move it. A small object lives on the stack (see Smart Pointers in the CPP STL) and points to a huge amount of data on the heap. Moving means, you create a new small object on the stack let it point to the huge amount of data and the old small object does not point anymore to the data.

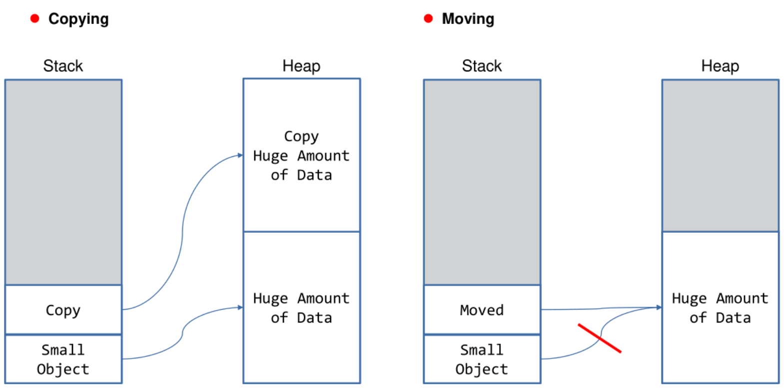


Figure 1: Copy vs move semantics

C++ References In C++ references are not the same as in Java. In C++ we have two kinds of references:

- lvalue references
- rvalue references

- `void scale(Point point)`: No side-effect (call by value)
- `void scale(Point & point)`: Has side-effect (call by ref - lvalue reference)
- `void scale(Point && point)`: rvalue reference

lvalue reference lvalue references in C++ are just another name for a existing object. The original must exists as long as it is referred to. Do not return locales as reference)! lvalue references binds to an lvalue.
The synatx for a lvalue reference is `T &`.

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

Listing 1: lvalue reference example

rvalue reference

rvalue references can extend the life-time of an temporary. The synatx for an rvalue reference is `T &&`. rvalue references binds to an rvalue (xvalues, prvalues)

```
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, does not compile
  consume(std::move(fries));   //explicit conversion lvalue to
       xvalue
  Food&& burger = fryBurger(); //life-extension of temporary
```

Listing 2: rvalue reference example

Different value categories

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

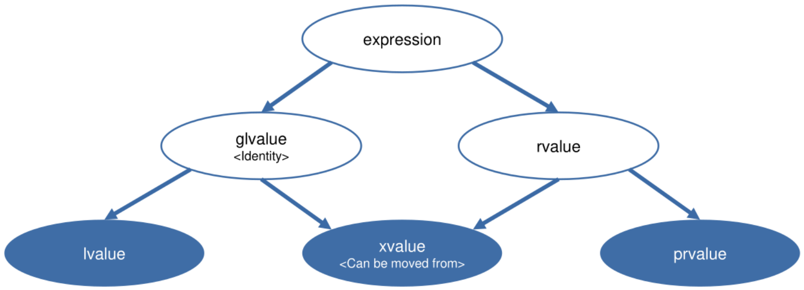


Figure 2: Kinds of expressions

lvalues Everything which belongs to the lvalue category can be bind by a lvalue reference.

- 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 for an lvalue:

- names of variables and parameters
- function call with return type of lvalue reference to class type (`std::cout << 23`)
- build-in prefix increment/decrement expressions (++a)
- array index access (`arr[0]`), when `arr` is an lvalue
- all string-literals by definition (`"name"`)
  - does not included user-defined literals like `"name"s` or `"name"sv`

prvalues

- address can not be taken
- can not be left-hand side argument of built-in assignment operators
- temporary materialization when a gvalue is required
  - conversion to xvalue

Examples for prvalues:

- literals: `23`, `false`, `nullptr`
- function call expression of non-reference return type: `int std::abs(int n)`
- serveral operators for built-in types, like post-increment / decremnt expressions: x++

temporary materialization (prvalue to xvalue) happens:

- when binding a reference to a prvalue (see 1 in ??)
- when accessing a member of a prvalue (see 2 in ??)
- 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

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

Listing 3: Example for prvalue to xvalue conversion

xvalues

- address can not be taken
- can not be taken
- can not be used as left-hand operator of built-in assignment
- conversion from prvalue throught temporary materialization

Examples for an xvalue:

- 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]`), when `arr` is an rvalue

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

```
X{}.member;
```

Overload for references The overload for lvalue parameters impose ambiguities (you can not use lvalue and rvalue overloads).

	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 &)

Figure 3: Overloard resolution for free 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 &)

Figure 4: Overloard resolution for member functions

Move constructor The move constructor is a type of constructors (Which kind of constructor exits in CPP?) in CPP to move all members to an new object.

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

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

Copy constructor The copy constructor is a type of constructors (Which kind of constructor exits in CPP?) in CPP to copy all members to an new object.

```
struct Date {
  Date(Date const &)
};
```

Destructor The Destructor is the counter part to the constructor (Which kind of constructor exits in CPP?). It must release all resources and is not allowed to throw an exception. If you program properly you will hardly ever need to implement it yourself. It is called automatically for local instances at the end of the block. Must not rhow exceptions!

```
class Date {
  ~Date();
};
```

Copy assignment If you need to implement the copy assignment operator you should follow the following points:

- 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;
  }
};
```

Listing 4: Example for copy assignment

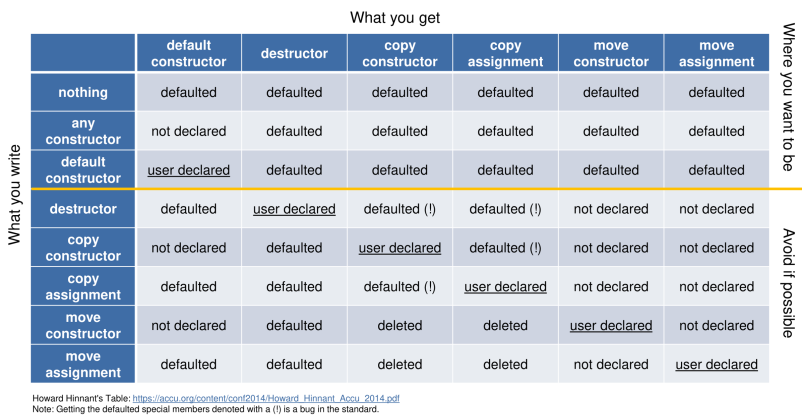
Move assignment Usually you don't need to implment this at all. If you have to, following the pattern is usually recommended:

- avoid self-move
- swap the `this` object with the parameter

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

Listing 5: Example for a move assignment

What special members 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	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

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.

Figure 5: What special member function do you get?

Copy elision In some cases the compiler is required to elide (omit) specific copy/move operations Regardless of the side-effects of the corresponding special member functions!

- The omitted copy/move special member functions need not exist
- If they exist, their side-effects are ignored

If initialization, when the initializer is a prvalues (see 1 in **??**) and when a function call returns a prvalue the compiler must omit the copy / move operations (see 2 in **??**).

```
auto create() -> S {
  return S{};
}
auto main() -> int {
  S s = S{S{}};               // 1
  S new_sw{create()};         // 2.1
  S * sp = new S{create()};  // 2.2
}
```

Listing 6: copy elision examples

The compiler is also allowed to optimize in throw and catch. To be sure to avoid copies, catch by `const &`.

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


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

Listing 7: copy elision examples try catch

Copy elision rules

- 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 type comparison)
  - the object is constructed in the location of the return value (insted of moved or copied)
- throw Expression
  - return expression is a local variable from the innermost surround try block
  - 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)

2 Type Deduction

Forwarding reference A forwarding reference can bind to an rvalue (prvalues, xvalues) and lvalues depending on the context. To create a forwarding reference, the function must be a template function. If the function is part of a template class, the function must introduce a new template.

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

template <typename T>
struct S {
T member

// no forwarding reference
auto f(T && param) -> void {}
}

// forwarding reference
template <typename E>
auto g(E && param) -> void {}
}


int main() {
  int x = 23;
  f(x)  // lvalue

  f(23) // rvalue
}
```

Listing 8: Example for forwarding references

Type deduction

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

Listing 9: Context

Deduction of type `T` depends on the structure of `ParamType`. We have three cases:

1. `ParamType` is value type (`auto f(T param) -> void`) (Rules for type deduction with value type in CPP)
2. `ParamType` is reference (`auto f(T & param) -> void`) (Rules for type deduction with reference type in CPP)
3. `ParamType` is a forwarding reference (`auto f(T && param) -> void`) (Rules for type deduction with forwarding reference type)

Type deduction with `std::initializer_list` does not work.

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

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

Listing 10: std::initializer$_{list}$ in type deduction

Type deduction with value types

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

f(<expr>);
```

Listing 11: Context

Steps:

1. `<expr>` is a reference type: ignore the reference
2. ignore `const` of `<expr>` (outermost)
3. pattern match `<expr>`'s type against ParamType to figure out `T`

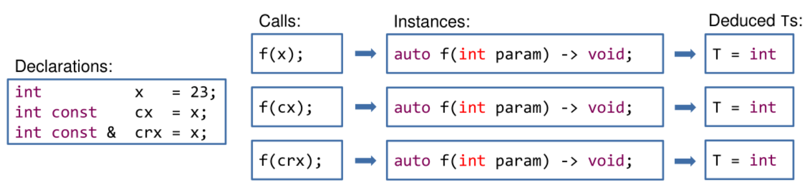


Figure 6: Example for type deduction with value type

Type deduction with reference types

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

f(<expr>);
```

Listing 12: Context non const

Steps:

1. `<expr>` is a reference type: ignore the reference
2. Pattern match `<expr>`'s type against ParamType to figure out T

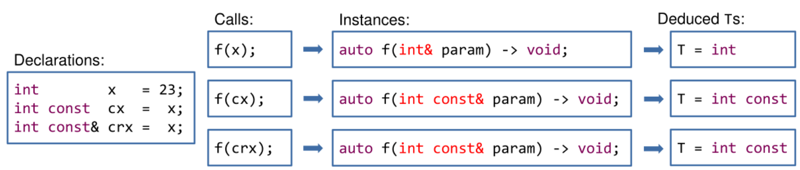


Figure 7: Example for type deduction with reference type

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

f(<expr>);
```

Listing 13: Context const

Steps:

1. `<expr>` is a reference type: ignore the reference
2. Pattern match `<expr>`'s type against ParamType to figure out T

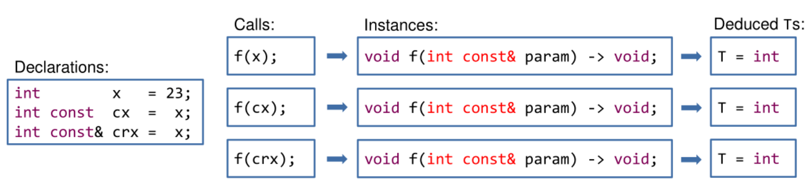


Figure 8: Example for type deduction with const reference type

Type deduction with forwarding reference types

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

f(<expr>);
```

Listing 14: Context

Steps:

1. `<expr>` is an lvalue: `T` and `ParamType` become lvalue references.
2. Otherwise (if `<expr>` is an rvalue): Rules for references apply (Rules for type deduction with reference type in CPP)

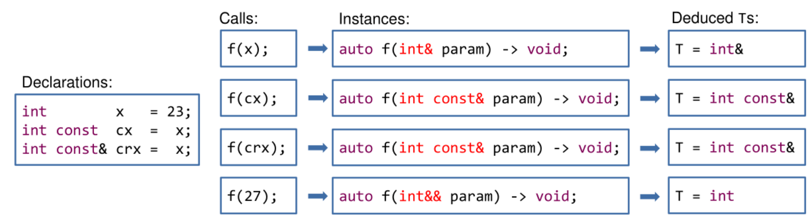


Figure 9: Example for type deduction with forwarding reference type

Type deduction with auto The type deduction with `auto` is the same as with templates (Rules for type deduction in CPP). `auto` takes the place of `T`.

```
auto x = 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&&

auto init_list1 = {23};    //std::initializer_list<int>
auto init_list2{23};       //int, was std::initializer_list<int
    >1
auto init_list3{23, 23};   //Error, requires one single
    argument
```

`auto` can also be used as return type as well for parameter declarations in labmdas and functions.

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

// results in this
template <typename T1, typename T2>
auto function(T1 arg1, T2 arg1) -> void {}

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

Listing 15: auto as function parameter

Type deduction with decltype `decltype` can be applied to an expression: `decltype(x)`. `decltype` represents the type of the applied expression.

```
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&

auto           just_x  = rx; // type of just_x is int
decltype(auto) more_rx = rx; // type of more_rx is int&
```

Listing 16: decltype examples

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

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

rules for `decltype(auto)`:

- unparenthesized variable name or data member: `T`, Type of the expression (retains reference)
- expression of value category xvalue: `T&&`, rvalue reference
- expression of value category lvalue: `T&`, lvalue reference
- expression of value category prvalue: `T`, value type of the expression

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

Listing 17: Example for decltype(auto)

Perfect forwarding

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

Listing 18: Forwarding Reference example

In **??** param is a forwarding reference. However, `param` is always an lvalue and `std::move(param)` always an rvalue. If `T` is of reference type we want to pass an lvalue otherwise we want to pass an rvalue. You can do this using `std::forward`.

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

Content c{};
log_and_do(c);
auto log_and_do(Content& param) -> void {
  do_something(std::forward<Content&>(param));
}

log_and_do(Content{});
auto log_and_do(Content&& param) -> void {
  do_something(std::forward<Content>(param));
}
```

Listing 19: Forwarding Reference using std::forward example

TODO Inside std::forward

3 Heap Memory Management

Pointers Pointer are things which point to a thing somewhere in the memory.

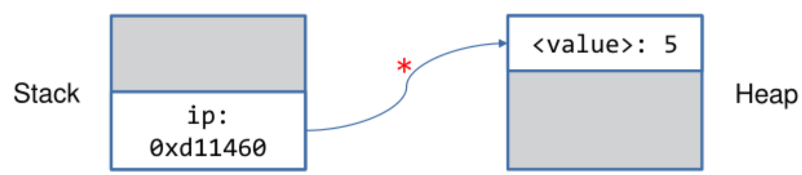


Figure 10: Point from stack to heap

To create an array on the heap you have to use the following synatx:

```
auto arr = new int[5]{};
// or
int * arr2 = new int[5]{};
// arr and arr2 are pointesr to the first element
```

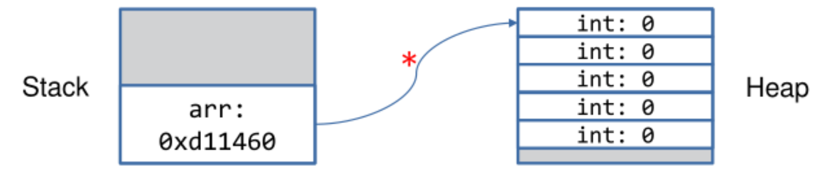


Figure 11: Pointer to array

If you want to create a null pointer (a pointer which points nowhere), you should use `nullptr` instead of `0` or `NULL` (`NULL` is just an alias to `0`).

```
auto bar(int i) -> void;
auto bar(S* ps) -> void;
//calls
bar(0);        //bar(int)
bar(NULL);     //surprising also bar(int)
bar(nullptr);  //bar(S*)
```

Allocate memory To allocate new memory on heap, use the following syntax:

- `new <type> <initializer>`

However, you can not allocate an array of non-default constructible types. However, you can allocate plain memory (`std::byte[]`) and initialize it later using Placement 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}};
}

auto defaultPoints(int x, int y) -> Point* {
  return new Point[2]{}; // does not work, because no default
      constructor
```

```
}
```

Listing 20: Allocate new memory on heap

```
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)
    ; // allocate plain memory
Point * first = &elementAt(memory.get(), 0);
new (first) Point{1, 2}; // create new Point object in the
    alread allocated memory
Point * second = &elementAt(memory.get(), 1);
new (second) Point{4, 5}; // create new Point object in the
    alread allocated memory

std::destory_at(second);
std::destory_at(first);
```

Listing 21: Example for a non default constructible type on the heap

Deallocate memory To deallocate memory you have to distinguied between array and single objects:

- memory deallocation: `delete <pointer>`
- array memory deallocation: `delete[] <pointer-to-array>`
  - this deallocates also multidimensional arrays

Deleting a `nullptr` does nothing However, deleting the same object twiche is **Undefined Behaviour**.

Placement new If you already allocated some memory, you can construct an object to this location.

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

This instruction does **NOT** allocate newy memory.
The same can also be done using the `std::construct_at` from the std library.

```
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}
  new (ptr) Point{7, 6};
  delete ptr;
}
```

Placement delete If you want to destruct an object but you dont want to free the memory, you can call the destructor manually.

- `ptr->~S()`

This instruction does **NOT** free the memory
The same can also be done using the `std::destroy_at` from the std library.

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

Listing 22: Example for placement new and delete

Stack only class If you want to prevent, that the user creates your data structure on the heap, you have to override the `new` and `delete` operators. The `new` operator should throw an exception, while the `delete` operator does nothing.

```
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
  }
};
```

Listing 23: Example struct with overriden operators

4 Iterator und Tags

What is a type tag A tag type is a class, which is only used to mark capabilities of associated types. Suach a tag type does not contain ayn members. It is also possible to derive tag types from each other to "inherit" the capabilities.

```
//Provides travelThroughSpace
struct SpaceDriveTag{};
//Provides travelThroughSpace and travelThroughHyperspace
struct HyperspaceDriveTag : SpaceDriveTag{};
//Provides travelThroughSpace and travelImprobably
struct InfniteProbabilityDriveTag : SpaceDriveTag{};
```

Tags for dispatch You should implement traits! You implement a trait using templates and set the tag. For each concrete implementation you can now create a template specialization and set tag accordingly. Then you create one public function (`travelTo`), which calls the dispachted functions with the tag (`travelToDispatch`).

```
struct SpaceDriveTag {};
struct HyperspaceDriveTag : SpaceDriveTag {};
struct InfiniteProbabilityDriveTag : SpaceDriveTag {};

struct MultiPurposeCrewVehicle;
struct GalaxyClassShip;
struct HeartOfGoldPrototype;

// Every Spaceship can travel through space
template <typename>
struct SpaceshipTraits {
  using Drive = SpaceDriveTag;
};

template <>
struct SpaceshiTraits<GalaxyClassShip> {
  using Drive = HyperspaceDriveTag;
};

template <typename Spaceshipt>
auto travelToDispatch(Galaxy destination, Spaceship& ship,
    SpaceDriveTag) -> void {
  ship.travelThroughSpace(destination);
}

template <typename Spaceshipt>
auto travelToDispatch(Galaxy destination, Spaceship& ship,
    InfiniteProbabilityDriveTag) -> void {
  while(destination != ship.travelImprobably());
}

template <typename Spaceship>
auto travelTo(Galaxy destination, Spaceship& ship) -> void {
  typename SpaceShipTraits<SpaceShip>::Drive drive{};
  travelToDispatch(destination, ship, drive);
}
```

Listing 24: Tag for dispatching example

Iterator Tags In CPP exits two kinds of iterators:

- input iterator
- forward iterator
- bidirectional iterator
- random access iterator
- output iterator

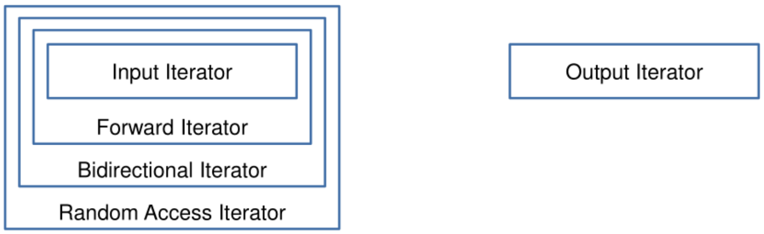


Figure 12: STL Iterator Categories

Iterator member types

```
struct IntIterator {
  using iterator_category = std::input_iterator_tag;
  using value_type = int;
  using difference_type = ptrdiff_t;
  using pointer = int *;
  using reference = int &;
}
```

Listing 25: Example member types for an iterator

TODO Iterator$_{traits}$<>

TODO Using Boost iterator

5 Advanced Templates

Static polymorphism Static polymorphism happens at compile-time and is therefore faster in the execution and the type checks happens at compile-time. In C++ static polymorhism works using templates. At compile time every call to the template, with an different type, is copied and the template parameter replaced with the type.
The drawback is, the compiler has to generate larger binary, the template code has to be known when used and the compile-time is longer.

```
template <typename T>
auto f(T t) -> void {
  // do something
}

f(int);
// generates this
auto f(int t) -> void {
  // do something
}
```

Listing 26: Example for a C++ template

Dynamic polymorphism Dynamic polymorphism is when you have a inheritance hirarchy and you call a virtual function on the object. This will then call the actual implementation on the object (not necessary the function on the base object). The dynamic dispatch is performed using a vtable.

```
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(); // calls area function on Square
  return area * seedsPerSquareMeter;
};

amountOfSeeds(Square{1});
```

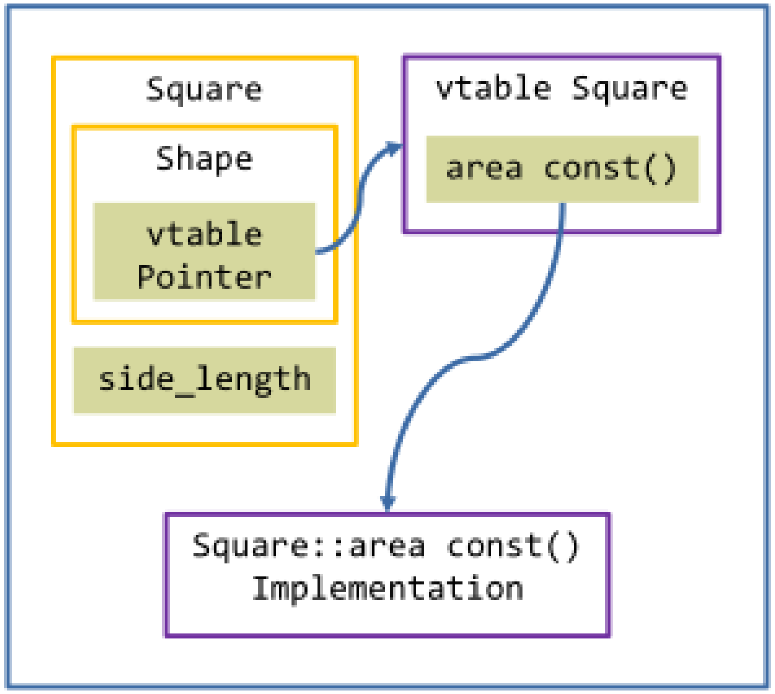


Figure 13: Dynamic Dispatch

SFINAE SFINAE stands for *Substitution Failure Is Not An Error*. When the C++ compiler performs overload resolution, the template parameters in a template declaration are substituted with the deducted types. This can result in template instances that can not be compiled. If the substituion of template parameters fails that overload candidate is discareded.

Substitution failure might happen in:

- function return type
- function parameter
- template parameter decleration
- and expressions in the above

However, errors in the instance body are still errors.

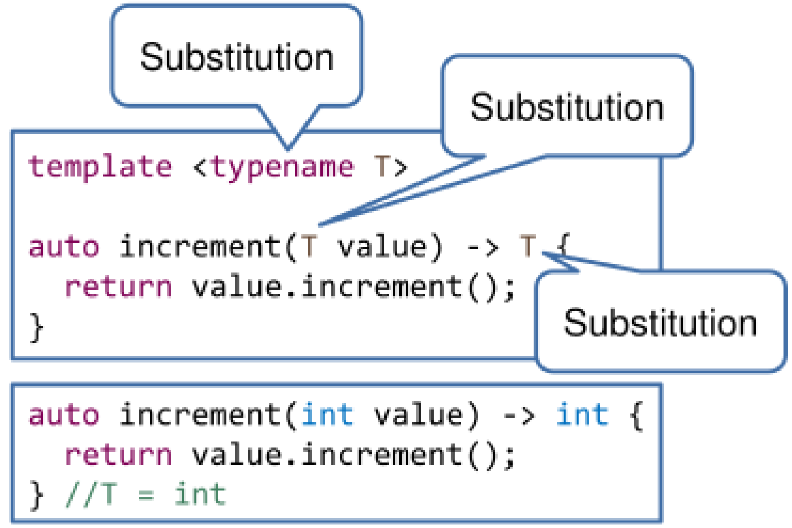


Figure 14: Positions for substituion

How to use SFINAE? One solution might be to use `decltype` as trailing return type. However, this does not always work (for example `void` as return type). Instead you should use type traits from the `<type_traits>` header.

```cpp
template <typename T>
auto increment1(T value) -> std::enable_if<std::is_class_v<T>,
    T> {
  return value.increment();
}

template <typename T>
auto increment2(std::enable_if_t<std::is_class_v<T>, T> value)
    -> T {
  return value.increment();
}

template <typename T, typename = std::enable_if_t<std::
    is_class_v<T>, void>>
auto increment3(T value) {
  return value.increment();
```

```
}
```

Listing 27: Example useage for SFINAE

Concepts Contraints are used to specify the characteristics of a type in template contextes. `requires` clauses allow constraining template parameters. `requires` is followed by a compile-time constatnt boolean expression.

```
template <typename T>
requires std::is_class_v<T>
auto function1(T argument) -> void {}

template <typename T>
auto function2(T argument) -> void requires std::is_class_v<T>
    {}

template <typename T>
requires requires (T const v) { v.increment(); } // requires
    requires is not an error
auto increment(T value) -> T {
  return value.increment();
}

// SFINAE
template <typename T>
auto function3(T value) -> std::enable_if<std::is_class_v<T>,
    T> {}
```

Listing 28: constraints vs SFINAE

requires expressions The `requires` keyword can also be used to start an expression that evaluates to bool. You can perform the following things:

- **simpel requirements** are statements that are true when they can be compiled
- **type requirements** check wheter a specific tpye exists
- **compound requirements** checks constraints on an expressions type
- **nested requirements** contain further (nested) requires expression

```
// simple requirements
template <typename T>
requires requires (T const v) { v.increment(); }
auto increment(T value) -> T {
  return value.increment();
}
```

Listing 29: simple requirements example

```
// type requirements
template <typename T>
requires {
  typename BoundedBuffer<int>::value_type;
  typename BoundedBuffer<int>::size_type;
  typename BoundedBuffer<int>::reference;
  typename BoundedBuffer<int>::const_reference;
}
```

Listing 30: type requirements example

```
// compound requirements
template <typename T>
requires requires (T const v) {
  { v.increment() } -> std::same_as<T>;
}
auto increment(T value) -> T {
  return value.increment();
}

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

Listing 31: Compound requirements exmaple

Deduction guides Deduction Guides are used to tell the compiler how to translate a set of constructor arguments into template parameters for the class.
In the following code snippet I have to tell the compiler how to map the Template Parameter `Iter`. Otherwise, the compiler would not know that `Iter` should be an iterator. It could be also an `int` or anything else.

```
template <typename T>
class Sack {
  //...
  template <typename Iter>
  Sack(Iter begin, Iter end) : theSack(begin, end) {}
  //...
};

// deduction guide
template <typename Iter>
Sack(Iter begin, Iter end) -> Sack<typename std::
    iterator_traits<Iter>::value_type>;
```

6 Compile Time Computation

constexpr contexts

- non-type template arguments: `std::array<Element, 5> arr{};`
- array bounds: `double matrix[ROWS][COLS]{};`
- case expressions
- enumerator initializers
- static$_{\text{assert}}$: `static_assert(order = 66)=`
- `constexpr` variables: `constexpr unsigned pi = 3;`
- `constexpr if` statements: `if constexpr (size > 0){}`
- `noexcept`

constexpr vs constinit `consexpr` variables are `const` while `constinit` variables are non-const. However, both are initialized at compile time.

```
constexpr unsigned pi = 3;
constinit unsigned pi2 = 3;
```

Listing 32: constexpr and constinit initialization

constexpr functions `constexpr` functions can be used to perform operations at compile-time. A `constexpr` function can:

- have local variables of literal types
- use loops, recursion, arrays, references
- can contain branches with run-time features, if branch is not executed during compile-time computation
- allocate dynamic memory (`new` / `delete`) that is cleandup by the end of the compilation
- be a virtual member function
- can only `constexpr` functions
- can not use exception handling on executed path

```
constexpr factorial(unsigned n) -> void {
  int local1;
  LiteralType local2{};
  std::string local3{};
}

consexpr auto allocate() -> int* {
  return new int{};
} //requires corresponding delete somewhere

struct Base {
  constexpr virtual auto modify() -> void;
};
```

Listing 33: Examples for constexpr function usage

consteval functions `consteval` functions are usable in `constexpr` contexts (see What are constant expression contexts in CPP?) only.

```
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);
  unsigned n;
  std::cin >> n;
  //std::cout << factorial(n); // does not compile
}
```

Listing 34: consteval example

UB in constexpr The compiler will prevent Undefined Behaviour during constexpr evaluation and generates a compiler error.

literal types A literal type is one of the following:

- built-in scalar types (`int`, `double`, pointers, enumerations)
- structs with some restrictions
  - trivial destructor (non-user-defined)
  - with a `constexpr` / `constval` constructor
- lambdas
- references
- arrays of literal types
- void

Literal types can be usedin in constexpr functions, but only `constexpr` member function can be called on values of literal types.

literal class types To be a literal type the class must have:

- a trivial destructor (non-user defined)
- at least one constexpr / constval constructor
- constexpr / constval member functions (only those are usabel in constexpr context)

It is also possible to have non-constexpr constructors as well as non-constexpr member functions.

```
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 -> {
    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];
  }
  // ...
}
```

Listing 35: Example for a literal class type

user-defined literals User-defined literals are a way to construct a class from a literal type.

```
auto speed1 = Speed<Kph>{5.0};
auto speed2 = Speed<Mph>{5.0};
auto speed3 = Speed<Mps>{5.0};

// vs.

auto speed1 = 5.0_kph;
auto speed2 = 5.0_mph;
auto speed3 = 5.0_mps;
```

Listing 36: User-defined literals in action

create user-defined literals To create a user-defined literal you have to overwrite to operator for it. The UDLSuffix could lexically be any identifier, but must start with an underscore. Other suffixes are reserved for the standard.

Also put your UDL overloads that belong together in a sperate namespace and import them when required using `using namespace`.

```
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)};
  }
}

namespace mystring {
  inline auto operator"" _s(char const *s, std::size_t len) ->
       std::string {
    return std::string {s, len };
  }

  // works only for integral and floating literals
  // 42_ss becomes std::string{"42"}
  inline auto operator"" _ss(char const *s) -> std::string {
    return std::string { s }
  }
}
```

Listing 37: user-defined literal for Speed

7 Threading and Mutexes

Threads In CPP we have many different classes for threads:

- `std::thread`: is started automatically (How to use std::thread in CPP?)
- `std::jthread`: automatically calls `join`

```
auto main() -> int {
  std::thread greeter {
    []{ std::cout << "Hello, I'm thread!\n" }
  };
  greeter.join();
}
```

Listing 38: threads in CPP

std::thread A `std::thread` is started automatically after its creation. To run a thread you have to provide a lambda, a function or a functor object which can be executed in a thread. Return values are ignored.

If possible you should pass all arguments by value to avoid data races and danglign references. If the thread goes out of scope you have to `join` or `detach` the thread.

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

struct function() -> void {
  std::cout << "Function" << std::endl;
}

auto main() -> int {
  std::thread functionThread{function};
  std::thread functorThread{Functor{}};
  std::thread lambdaThread{
    []{ std::cout << "Lambda" << std::endl; }
  };

  lambdaThread.join();
  functorThread.join();
  functionThread.join();
}
```

Listing 39: std::thread example

Wake up a thread

1. `notifyAll()`, `notify()`
2. InterruptedException (The InterruptedException in Java)
3. Spurious Wake up (falsely wake up POSIX Thread API)

Mutexes In the C++ standard library we have four kinds of mutexes. They can be devided into recursive and timed.

recursive allow multiple nested acquire operations of the same thread (prevents self-deadlock)

timed also provides timed acquire operations (`try_lock_for` / `try_lock_until`)

Each mutex has multiple operations defined (more for read-locking - `_shared`):

- `lock()` - blocking
- `try_lock()` non-blocking
- `unlock()` - non-blocking
- `try_lock_for(<duration>)` - only for timed
- `try_lock_until(<time>)` - only for timed

After each lock you must unlock the mutex, otherwise deadlocks can occurs. To prevent this problem you can use RAII wrappers (What are the different locks for mutexes?).

read-write locks Mutual exclusion is too strong when only reading happens. Mutual exclusion is only required when minimal one thread wants to write.

Parallel	Read	Write
Read	Yes	No
Write	No	No

```
var rwLock = new ReentrantReadWriteLock(true);
rwLock.readLock().lock();
// read-only accesses
rwLock.readLock().unlock();
rwLock.writeLock().lock();
// write (and read) accesses
rwLock.writeLock().unlock();
```

different locks Instead of locking and unlocking manually a mutex, you can use RAII wrappers to automatically lock and unlock a mutex.

```
template <typename T, tpyename MUTEX = std::mutex>
struct threadsafe_queue {
  using gurad std::lock_guard<MUTEX>;

  auto push(T const &t) -> void {
    guard lk{mx}; // acquire lock
    q.push(t);
  } // release lock

  auto empty() const -> bool {
    guard lk{mx};
    return q.empty();
  }

  auto swap(threadsafe_queue<T> & other) -> void {
    if (this == &other) return;

    std::scoped_lock both{mx, other.mx}; // lock multiple
        mutexes
    std::swap(q, other.q);
    // no need to swap mutex or condition variable
  }

private:
  mutable MUTEX mx{}; // must be mutable, otherwise you could
      not call empty
  std::queue<T> q{};
}
```

Listing 40: Example usage for RAII wrappers

lock and conditions In lock & condition you have a monitor (How does the monitor lock work?), but instead of one queue you have a queue for each condition. This has the benefit that not always all threads must be woken up. Only the threads where the condition may be now fulfilled.

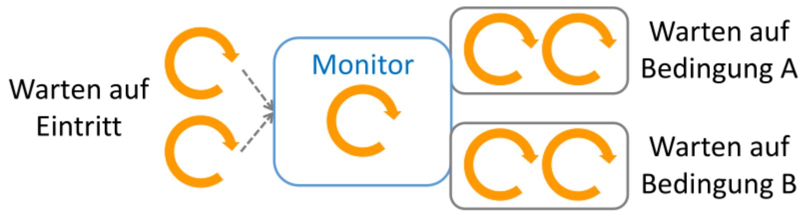


Figure 15: Lock & Conditions

TODO std::condition$_{\text{variable}}$

Return value from thread In C++ we have *futures* and *promise* to communicate results from a thread to another. The `std::future` represent results that may be computes asynchronously, while the `std::promise` is a possible origin for a future.

```
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); // communicate result
  }};

  std::this_thread::sleep_for(1s);

  std::cout << "The answer is: " << result.get() << '\n';
  thread.join();
}
```

Listing 41: Example usage for future and promise

std::async Performing intesive computations on a different thread is a common task. The C++ standard library uses for this the `std::async` class. `std::async` returns a `std::future`. Based on the *launch policy* the execution is scheduled on the current thread (`std::launch:deferred`) or in a new thread (`std::launch::async`). You should always the *launch policy* because the standard does not define it uniquely.

```
auto main() -> {
  auto the_answer = std::async(std::launch::async, [] {
    return 42;
  });

  std::cout << "The answer is: " << the_answer << '\n';
}
```

Listing 42: Example usage for std::async

The `std::launch::async` policy performs the operation, regardless if the result is required. Attention: the destructor will block until the result is availabel, but not in the `deferred` policy. The `std::launch::deferred` policy performs the operation only if you need the result. But the calculation is performed in the calling thread.

8 Memory Model and Atomic

Memory Location In C++ a memory location must be one of the following types:

- arithmetic
- pointer
- enum
- std::nullptr

Memory Model Visibility of effects:

sequenced-before withing a single thread

happens-before either sequenced-before or inter-thread happens-before

synchronizes-with inter-thread sync

Reads / Writes in a single statement are **unsequenced**!

For the memory ordering see The memory ordering in CPP.

Memory Ordering The memory orderings defi define when effects become visible:

sequentially-consistent intuitive and the default behaviour

acquire / relase weaker guarantees than sequentailly-consistent

consume (do not use this) slightl weaker than acquire-release
relaxed no guarantees besides atomicity

Atomics In C++ atomics are realised using the `atomics` template. atomics ar guaranteed to be data-race free. `std::atomic_flag` is guranteed to be lock-free. All other atomics might use locks internally (for example for custom types). However, they must be **trivially-copyable**.
All atomic operations take an additional argument to specify the memory ordering (The memory ordering in CPP):

- `std::memory_order::seq_cst`
- `std::memory_order::acquire`
- `std::memory_order::release`
- `std::memory_order::acq_rel`
- `std::memory_order::release`

```
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();
}
```

Listing 43: Example for atomic$_{\text{flag}}$

Seq-Cst Ordering Using sequential consistency all operations are globally ordered. Every thread will observe the same order.
The latest modification (in the global execution order) will be available to a read.

Acquire / Release Ordering Using the *acquire* ordering 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 visibile in the current thread.

```
x.load(std::memory_order::acquire);
```

Using the *release* ordering no reads or writes in the current thread can be reordered **after** this store. All writes in the current thread are visible in other threads that acquire the same atomic.

```
x.store(std::memory_order::release);
```

the *acquire/release* works on the latest value.

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

Note: It is not guaranteed that you always see the latest write in a read operation, but what you see is consistent according to the ordering above regarding the same atomic!

Releaxed Ordering The relaxed memory order does not give and guarantees about sequencing. It guarantees only no data-races for atomic variables.

```
x.store(true, std::memory_order::relaxed);
```

Volatile The `volatile` keyword prevents the compiler from optimizing the varaible, even if the compiler can not see any visible side-effects within the same thread. The compiler also does not reorder the instructions. However, the hardware might still reorder instructions.

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

9 Networking

Socket primites

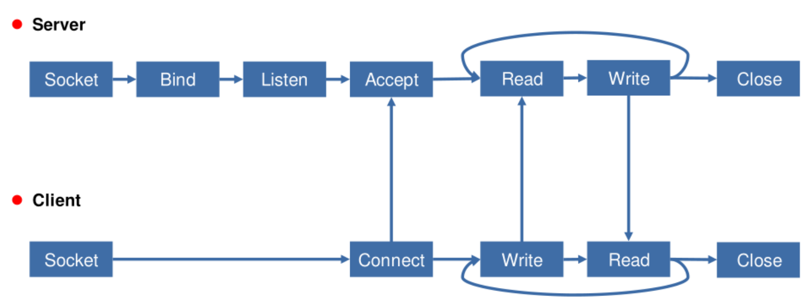


Figure 16: Connection-oriented communication pattern using sockets

Sync TCP client

```
auto main() -> int {
  asio::io_context context{};
  asio::ip::tcp::socket socket{context};
  auto address = asio::ip::make_address("127.0.0.1");
  auto endpoint = asio::ip::tcp::endpoint(address, 80);

  // asio::ip::tcp::resolver resolver{context};
  // auto endpoints = resolver.resolver(domain, "80"); // DNS
  socket.connect(endpoint);

  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()));

  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);

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

Listing 44: Client connection example using ASIO

Sync TCP server

```
auto main() -> int {
  asio::io_context context{};
  asio::ip::tcp::endpoint localEndpoint{asio::ip::tcp::v4(),
      port};
  asio::ip::tcp::accptor acceptor{context, localEndpoint};

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

Listing 45: Server example using ASIO

ASIO async operations

1. the application invokes an async operation on an IO object and passes a completion handler as a callback
2. the IO object delegates the operation and the callback to its `io_context`
3. the OS performs the async operation
4. the OS signals the `io_context` that the operation has been completed
5. when the program calls `io_context::run()` the remaining async operations are performed (wait for the result of the OS)
6. still inside `io_context::run()` the completion handler is called to handle the result / error of the async operation

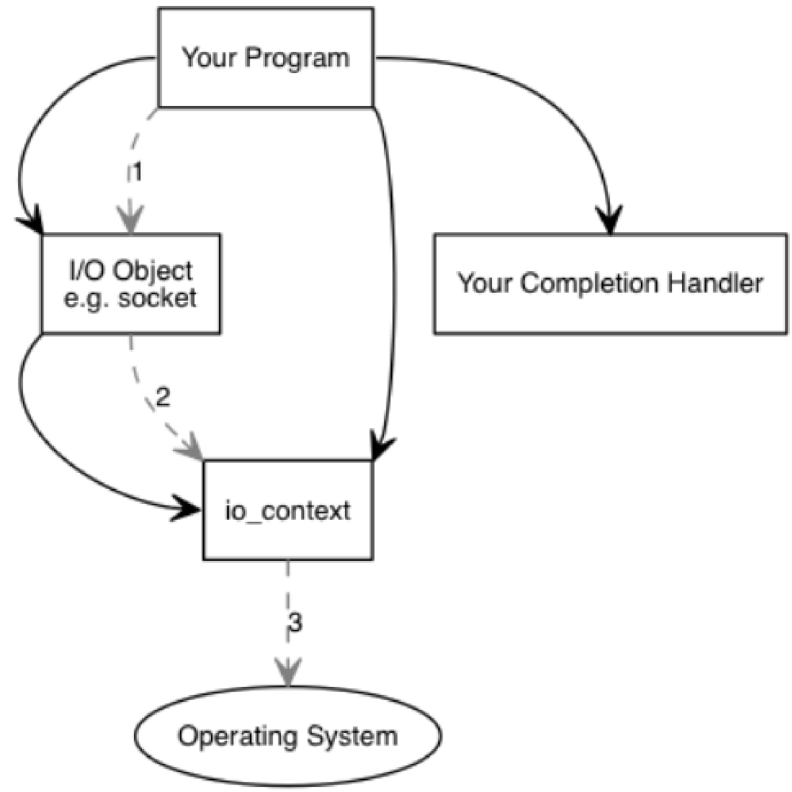


Figure 17: ASIO async operations part 1

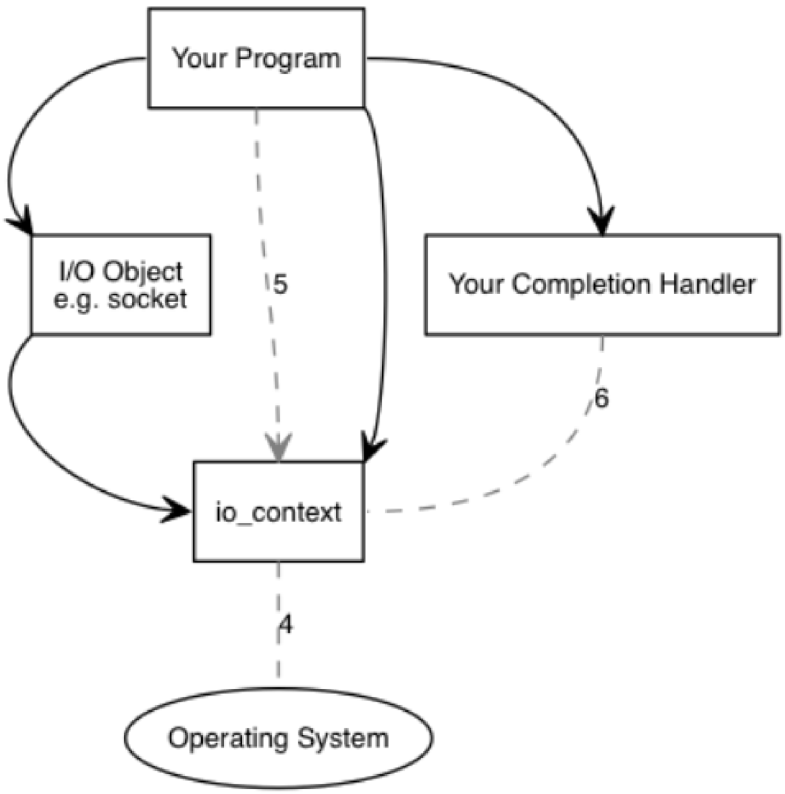


Figure 18: ASIO async operations part 2

ASIO async example Make sure the `Server` lives as long as async operations on it are proccessed

```
struct Session : std::enable_shared_from_this<Session> {
  explicit Session(asio::ip::tcp::socket socket);
  void start() {
        read;
  }

private:
  void read() {
        // using shared_from_this the session is kept alives
        auto handler = [self = shared_from_this()](error_code
            ec, size_t length) {}
  }
  void write(std::string data);

  asio::streambuf buffer{};
  std::istream input{&buffer};
  asio::ip::tcp::socket socket;
};

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

private:
  void accept() {
        auto acceptHandler = [this](asio::error_code ec, tcp::
            socket peer) {
          if (!ec) {
            auto session = std::make_shared<Session>(std::move
                (peer));
            session->start(); //without the shared_from_this,
                the session would die here
          }
          accept();
        };
        acceptor.async_accept(acceptHandler);
  }

  tcp::acceptor acceptor;
};
```

Listing 46: Async Example using ASIO

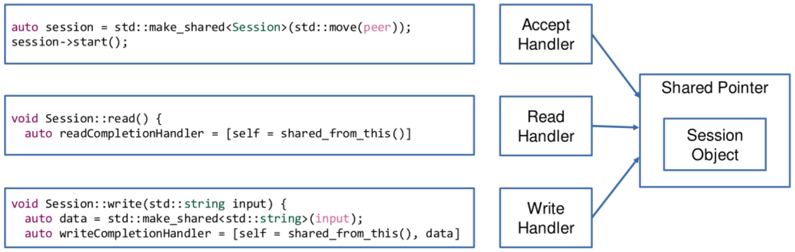


Figure 19: Keep session alive

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();
}
```

Listing 47: Example for signal handling using ASIO

10 Advanced Library Design

Exception Safety In generic code (CPP Template) you might call user-defined operations from the template argument. The user-defined operations must not garble the data structure or leak resources. At the same time, your template code is responsible that user-provided code does not suffer.

When an exception is thrown, "stack unwinding" destroys local and temporary objects. If an exception is thrown during unwinding the program is terminated using `std::terminate()`. To prevent this, you normally should not throw exceptions in the destructor.
To prevent all named problems you need to specify exception safty / exception guaranty.

Exception Safety Levels

noexcept / no-throw will never-ever throw an exception
strong exception safety operation succeeds and does not throw, or nothing happens but an exception is thrown (transaction)
basic exception safety does not leak resources or garble interanl data structures in case of an exception but might be incomplete
no guarantee undefined behaviour and garbled data lurking if exception is thrown (you dont want to go here)

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

noexcept keyword The `noexcept` keyword belongs to the function signature, but you can no overload on `noexcept`. The compiler might optimize a call of a `noexcept` function better because it is not required to provide the infrastructure of uniwinding the stack. However, if you throw in a `noexcept` environment the application terminates immediately.

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

template <typename T>
auto function(T t) noexcept(true) -> void {
  // ...
}


template <typename T>
auto function(T t) noexcept(false) -> void {
  // ...
```

```
}
```

Listing 48: Exaple for noexcept in signature

You can use the `noexcept` keyword also for conditions.

```
auto function2() noexcept(noexcept(function())) -> void {
  // function2 is noexcept when function() is also noexcept
}


auto main() -> int {
  std::cout << "is function() noexcept? " << noexcept(function
      ()) << '\n';
}
```

Listing 49: Example for conditional noexcept

Throwing in member functions The destructor should normally not throw an exception. During stack unwinding the destructor is called and if you then throw an exception the application terminates. Move construction, move assignment and `swap` should not throw. Copy can throw when new resources need to be allocated.

wide vs narrow contracts

A function that can handle all argument values of the given parameter types successfully has a "Wide Contract":

- it can not fail
- it should be specified as `noexcept(true)`
- `this` is also a parameter
- globals and external resources also (e.g. heap)

A function that has preconditions on its parameters has a narrow contract:

- i.e., `int` parameter must not be zero
- i.e., pointer parameter must not be `nullptr`
- even if not checked and no excption thrown, those function should not be `noexcept`
- this allows later checking and throwing if U.B.

TODO Standard Library Helpers

Opaque Types

For an opaque type we do not know anything about its structure but its name. This is achived using a forward declaration.

```
struct S; // Forward declaration
auto foo(S & s) -> void {
  foo(s);
  //S s{}; // Invalid
}

struct S{}; // Definition
auto main() -> int {
  S s{};
  foo(s);
}
```

PIMPL Idiom If you make changes to a class definition, the client must be recompiled. Even then, when the changes are not visible from outside. Using PIMPL this can be neglected.
In the *exported* header file you write a class consisting of a **Pointer to Implementaiton** and all public members of the real implementation.

```
// Wizard.hpp
class Wizard {
  // class WizardImpl is a forward declaration
  std::shared_ptr<class WizardImpl> pImpl;

public:
  Wizard(std::string name = "Rincewind");
  auto domagic(std::string wish) -> std::string;
};

// WizardImpl.cpp
class WizardImpl {
  std::string name;
  /// ...
public:
  WizardImpl(std::string name = "Rincewind") :
    name{name} {}
  auto doMagic(std::string const & wish) -> std::string {}
};

Wizard::Wizard(std::string name) :
  pImpl{std::make_shard<WizardImpl>(name)} {}
```

```cpp
auto Wizard::doMagic(std::string wish) -> std::string {
  return pImpl->doMagic(wish);
}
```

Listing 50: PIMPL example using $\text{shared}_{\text{ptr}}$

PIMPL with $\text{unique}_{\text{ptr}}$ If you want to implement the PIMPL idiom using an `unique_ptr` you must define the desturcotr (The Destructor in CPP) manually in the class declaration. This is required, because the default delter for the `unique_ptr` has to know, how big the implementation is. Therefore, you have to implement the destructor **after** the implementation again with `default`.

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

// WizardImpl.cpp
class WizardImpl {
  // ...
};


// Default the destructor
Wizard::~Wizard() = default;
```

Listing 51: PIMPL using $\text{unique}_{\text{ptr}}$

PIMPL and Copy

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)

11 Hour Glass Interface

Hour Glass Interface An hourglass interface is a way to expose your C++ library over a C ABI to another language (or C++).

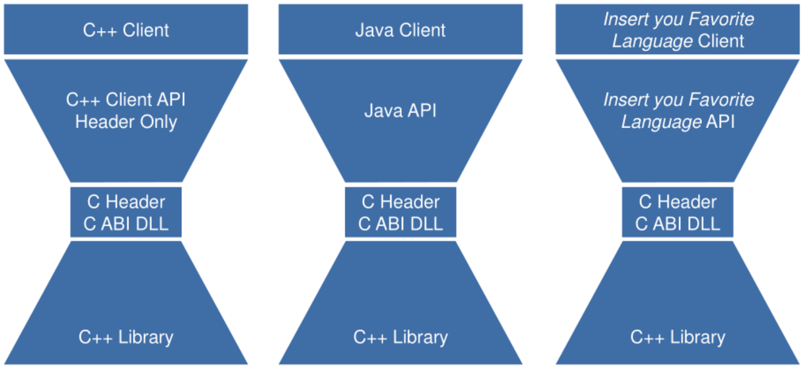


Figure 20: Hourglass Interface idea

C++ to C ABI Abstract data types can represented by pointers (`void *`). Member functions map to functions taking the abstract data type pointer as first argument. Constructors and destructors are replaced with factory and disposal functions. Strings can only represented by `char *`. Exception do not work across the C ABI.

C++ in extern C C++ has a lot more features than C has. Therefore, you can not use all features in an `extern "C"` interface:

- functions, but no template or variadic
- C primitive types (`char`, `int`, `double`, `void`)
- pointers, including function pointers
- forward-declared structs
  - pointers to those are opaque types
  - are used for abstract data types
- enums (unscoped - wouthout class or base type)

```
#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);

#ifdef __cplusplus
}
#endif
```

Listing 52: Example for a extern C interface

Full C++ using extern C In the `extern C` interface we can only use a subset of C++ (What parts of CPP can be used in an extern C interface?). So that we can use full C++ we have to create a so-called trampolin class.

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

    unseen::Wizard wiz;
  };
}

// WizardHidden.hpp
namespace unseen {
  struct Wizard {
    // ...
    Wizard(std::string name = "Rincewind")
    : name{name}, wand{} {}

    auto doMagic(std::string const & wish) -> char const *;
    auto getName() const -> char const * {
      return name.c_str();
    }
  };
}
```

Listing 53: Example for a trampolin class

Exception in extern C In `extern C` we can not use exception and we must use pointers to pointers. If an error occurs (exception) we have to allocate error value on the heap and provide a disposal function to clean up error. To convert an exception to an error, we just capture everything and create an error when required. We use a pointer to a pointer as referenec to a pointer. Therefore, `out_error` must not be `nullptr`;
On the client side we can use RAII wrappers to convert errors again into exceptions.

```
// Wizard.cpp
extern "C" {
  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{"Unkown internal error"};
      return false;
    }

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

```
    });
    return result;
  }
}
```

12 Build Automation

Build Automation Software You can seperate into two classes of build automation software:

- Make-style build tools
  - run build scripts
  - produce your final products
- Build Script Generators
  - generate configuration for make-style build systems
  - configuration independent of actual build tool
  - advanced features (download dependencies, ...)

cmake When the flag is `PUBLIC`, a libraries or executables which links against it will inherit the properties. Therefore, libraries should be `PUBLIC` and executables `PRIVATE` because you cannot link against an executable.

```
cmake_minimum_required(VERSION "3.12.0")

project("my_lib" LANGUAGES CXX)

add_library("my_lib" "lib.cpp")
target_include_directories("my_lib" PUBLIC "includes")
target_compile_features("my_lib" PUBLIC "cxx_std_17")
set_target_properties("my_lib" PROPERTIES
                        CXX_STARNDARD_REQUIRED YES
                        CXX_EXTENSIONS NO)


add_executable("my_app" "app.cpp")
target_link_libraries("my_app" PRIVATE "my_lib")
```

Listing 54: Example for CMakeLists.txt

```
mkdir build
cd build
cmake ..
cmake --build .
```

Listing 55: Build cmake project

Project Layout

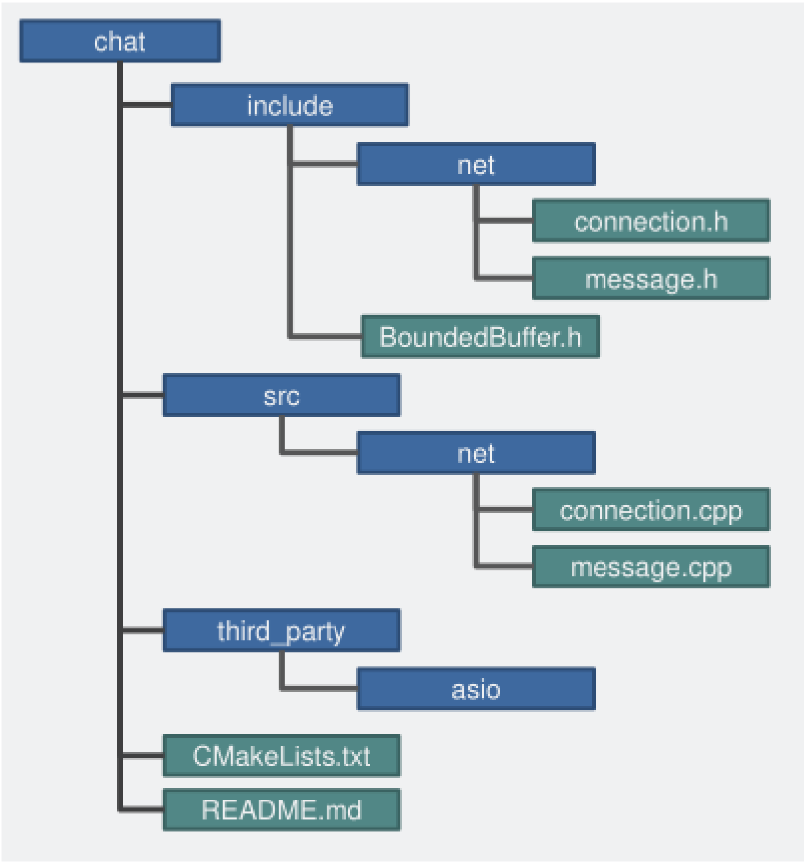


Figure 21: Recomended project layout for an application

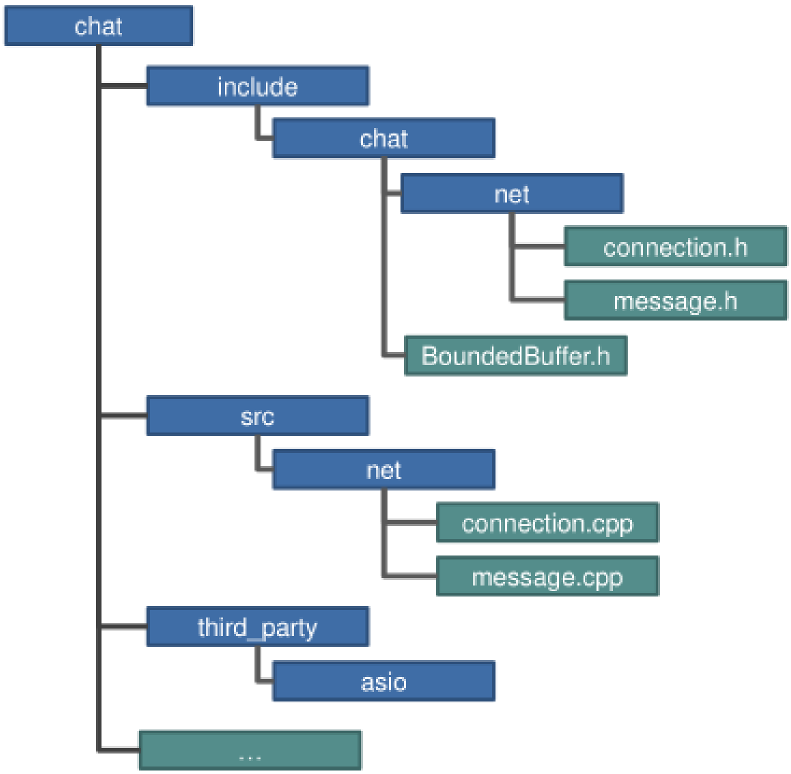


Figure 22: Recomended project layout for a library