# C++ | CPl

## Zusammenfassung

## CONTENTS

***Hinweis:** In der Typst-Datei kann mit `#let EXAM_MODE = true` zwischen der On- und Offlineversion von CPPReference gewechselt werden, um diese Links auch in der CAMPLA-Prüfungsumgebung nutzen zu können.*

## 1. INTRODUCTION

`main()` is the program entry function. Unlike Java, C++ provides ***functions***, not methods. Not all functions are bound to a class or object. Bound functions are called ***member-functions***.
Return types are written in front of the function name *(C style)* or as trailing return-types *(modern C++ style)* in declarations. `main()` implicitly returns 0.

```
// modern C++ function definition
auto main() → int { }
```

```
// classic C style function definition
int main() { }
```

The difficulties with C++ lie in the ***"permissiveness" to program C*** that still compiles as C++, the ***manual memory management*** *(no garbage collection)* and the ***"undefined behavior"*** in the C++ standard, where if conditions occur that aren't described in the standard, every compiler can do what it wants, leading to unpredictable non-deterministic results.

### 1.1. COMPILATION PROCESS

- ****.cpp files for source code:*** Also called "Implementation File". Contains function implementations and is the source of compilation - aka the "Translation Unit"
- ****.hpp (or *.h) files for interfaces (and templates):*** Also called "Header File". Contains declarations and definitions to be used in other implementation files *(shared variables, function signatures)*. Textual inclusion through a pre-processor with `#include "header.hpp"`. The pre-processor then "copies" the entire content of `header.hpp` into the file.

C++ is usually compiled into machine code. Unlike Java, there is ***no Virtual Machine overhead***. There are 3 phases of compilation:
- ***Preprocessor:*** Textual replacement of preprocessor directives (`#include`, `#define` etc.)
- ***Compiler:*** Translation of C++ code into machine code (source file to object file)
- ***Linker:*** Combination of object files and existing libraries into new libraries and executables

`sayhello.cpp` → ***Preprocessor*** → `sayhello.i` *(preprocessed source)* → ***Compiler*** → `sayhello.o` *(object code)* → ***Linker*** → `sayhello` *(binary)*

#### 1.1.1. Files of `sayhello`

**`main.cpp`**

```
#include "sayhello.hpp"
#include <iostream>

auto main() → int {
  sayHello(std::cout);
}
```

**`sayhello.hpp`**

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_
#include <iosfwd>

auto sayHello(std::ostream&) →
void;
#endif /* SAYHELLO_HPP_ */
```

**`sayhello.cpp`**

```
#include "sayhello.hpp"
#include <ostream>

auto sayHello(std::ostream& os) →
void {
  os << "Hi there!\n";
}
```

The ***preprocessor*** combines `main.cpp` and `sayhello.hpp` into the preprocessed source `main.i`. On this, the compiler creates the object file `main.o` for this translation unit in machine code. The Linker finally combines the translation units `main.o` and `sayhello.o` into the executable `sayhello`.

**`main.i`**

```
<content of iosfwd>
auto sayHello(std::ostream&) →
void;

<content of iostream>
auto main() → int {
  sayHello(std::cout);
}
```

**`main.o`**

```
010110101... (machine code)
<Defintion of main() which
calls sayHello>
```

**`sayhello.o`**

```
010110101... (machine code)
<Definition of sayHello()
which writes to the ostream
parameter>
```

**`sayhello Executable`**

```
010110101... (machine
code)

<executable program>
```

## 1.2. MODULARIZATION & TESTING

Code in C++ should be ***modularized into libraries*** to allow for ***unit testing***. `main()` is usually kept ***minimal,*** with only a few calls to library functions, as this code can't be unit tested. Using library functions requires ***`#include`***, normally at the beginning of the file. The names of macros provided by the ***Catch2 unit testing framework*** are written in ***uppercase.***

```
#include "sayhello.hpp"
#include <ostream>
auto sayhello(std::ostream & out) -
> void {
  out << "Hello world!\n";
}
```

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <iosfwd>
auto sayhello(std::ostream & out) -
> void;

#endif /* SAYHELLO_HPP_ */
```

```
#include
"sayhello.hpp"
#include <iostream>
auto main() → int {

sayhello(std::cout);
}
```

## 1.3. DECLARATIONS AND DEFINITIONS

All things with a name must be ***declared before usage*** *(e.g. function call, type of a variable, variables)*. ***Names*** for things concerning the ***preprocessor*** are conventionally written in ***uppercase.***

### 1.3.1. Declaring Functions

Declarations are usually put into a ***header file*** *(*.hpp)*, so other modules can ***access*** and call them. There can be ***multiple declarations*** of the same function.

```
auto          <function-name>(<parameters>) → <return-type>;
<return-type> <function-name>(<parameters>);
```

| *Term* | *Description* |
|---|---|
| Return Type | Every function either returns a value of a specified type or it has return type `void`. |
| Function Name | Identifier. Overloading allowed *(Multiple functions with the same name but different parameters)*. |
| Parameters | A list of $0$ to $N$ parameters. Each parameter has a type and an optional name. |
| Signature | Combination of name and parameter types. Used for overload resolution. No return type overloading. |

### 1.3.2. Defining & Implementing Functions

Specifies what the function does. Definitions are usually put into a source file *(*.cpp)*. There can only be ***one definition*** of the ***same function*** *(One Definition Rule)*. Functions with ***non-void return types*** must return a value on ***every code path*** or throw an exception. The compiler only throws a warning, not an error without a valid return statement, so this still compiles *(undefined behavior)*!

```
auto          <function-name>(<parameters>) → <return-type> { /* body */ }
<return-type> <function-name>(<parameters>)                 { /* body */ }
```

| *Term* | *Description* |
|---|---|
| Return Type, Function Name, Parameters, Signature | Same as for function declaration |
| Body | Implementation of the function with $0$ to $N$ statements |

#### One Definition Rule (ODR)

While a program element can be ***declared several times*** there can be ***only one definition*** of it. ***Consequences:*** There can be only one definition of the `main()` function or any other function with the same signature. There ***must be a definition for all elements*** that are used.

***`#include` guards are recommended*** in header files, so a function cannot be accidentally included multiple times over dependencies.

### 1.3.3. Include Guards

Use of specific preprocessor directives to ensure that a header file is ***only included once***. A code block within an include guard is skipped on subsequent inclusions. Without it, invalid code could be generated.

| ***Directive*** | ***Description*** |
|---|---|
| `#ifndef SYMBOL` | Checks whether the `SYMBOL` macro has already been defined.<br>If not, the block until `#endif` is included. |
| `#define SYMBOL` | Defines a macro named `SYMBOL` without any content. |
| `#endif` | Closes the conditional block opened by `#ifndef` |

```
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <iosfwd>
struct Greeter { /* ...
*/ };
#endif /* SAYHELLO_HPP_ */
```

## 1.4. DIFFERENCES C++ AND JAVA

| ***C++*** | ***Java*** |
|---|---|
| Allocates memory for variables on definition ***on the stack***. No ***explicit heap memory*** needed. No indirection and space overhead.<br>`Type name{};` | Objects are placed on the ***heap*** *(as references)*.<br>Exception: Primitive values (`int`, `float`, `boolean`).<br>`Type name = new Type();` |
| Assigning an object to another object results in ***two different objects*** on the stack.<br>`// Copied values`<br>`Point p1{1, 20}; point p2{p1};` | Because only a ***reference*** is stored, the reference points to the ***same data on the heap***, modifications affect both variables.<br>`// shared values`<br>`Point p1 = new Point(1,2); Point p2 = p1;` |

Due to C++'s ***allocation implementation***, functions can mix and match the two different types of parameters:

- ***Value Parameter:*** No side-effect on the call-site, because the elements get copied *(call by value)*
- ***Reference Parameter:*** Side-effect on the call-site. Needs to be explicitly defined with an &: `Point & x` *(call by reference)*.

A function has a ***side effect*** if it does more than reading its parameters and returning a value to its callee, i.e. modifying non-local variables, performing I/O or throwing errors.

# 2. VALUES AND STREAMS

## 2.1. VARIABLES

`<type> <variable-name>{<initial-value>}`: `int anAnswer{42}`, `int const theAnswer{42}`

Variables initialized with ***empty {}*** are initialized with the ***default value*** of this type. Using = or {} for initialization with a value we can have the ***compiler determine its type***: `auto const i = 5`; ***Uninitialized variables*** contain ***random*** values. Dangerous! Variables are best defined ***as close to their use as possible***.

***Naming Conventions:*** Begin variable names with a lower-case letter. Do not abbreviate unnecessarily.

### 2.1.1. `const`: Constants

*CPPReference: const and volatile*

Adding `const` in front of the name makes the variable ***only assignable at initialization*** - a ***constant***.

`int const theAnswer{42}`

It is ***best practice*** to use `const` whenever possible for ***non-member variables*** that don't need to be updated.

### 2.1.2. Name visibility / Scope

A variable defined within a block is ***invisible after*** the block ends. ***Redefining*** an existing variable inside a block is ***not*** an error in C++.

### 2.1.3. Types

CPPReference: Fundamental Types

- `short`, `int`, `long`, `long long` *(also available as `unsigned` variants)*
- `bool`, `char`, `unsigned char`, `signed char` *(are treated as integral numbers as well)*
- `float`, `double`, `long double`
- `void` is special, it is the type with ***no values***
- `std::string`, `std::vector` *(requires `#include` of the type definition)*

## 2.2. VALUES AND EXPRESSIONS

CPPReference: Expressions

| *Arithmetic Expressions* | *Logical Expressions* | *Bit-operators* |
|---|---|---|
| – **binary:** `+, -, *, /, %`<br>– **unary:** `+, -, ++, --`<br>*Unary have one, binary two operands* | – **ternary/conditional:** `? :`<br>– **binary:** `&&, \|\|`<br>– **unary:** `!` | – **binary:** `& \| ^ << >>` *(bitand, bitor, xor, shift)*<br>– **unary:** `~` *(complement)* |

Unusual literals: `5ull` *(unsigned long long)*, `0x1f` *(int32)*,`0.f` *(float)*, `1e9` *(double)* $10^9$, `42.E-12L` *(long double $42 * 10^{-12}$)*

### 2.2.1. Type Conversion

C++ provides ***automatic*** type conversion if values of different types are ***combined*** into an expression, ***unless in braced initialization*** like `int i{1.0}`.
***Division*** results of integers get ***rounded down*** (`double x = 45 / 8` evaluates to `5`). ***Integers*** can be automatically ***converted*** to `bool`: `0` is `true`, every other value `false`.
***Logical operators*** and conditional statements accept ***numeric values***; however `if(5)` is probably not useful. This can cause confusion, as `if(a < b < c)` does **not** test whether `b` is between `a` and `c`.

### 2.2.2. Floating Point Numbers

Use `double` instead of `float`. `float` is only needed if memory consumption is utmost priority and precision and range can be traded.
There are ***legal*** double values that are ***not numbers***: `NaN, +Inf, -Inf`. ***Comparing*** floating points for equality (`==`) is usually wrong, better check if it is in a certain range around the expected value.

## 2.3. STRINGS

CPPReference: Strings library

`std::string name{"Bjarne Stroustrup"};`
Type for representing ***sequences of char***. Only 8 bit, so ***no Unicode support***. Literals like `"ab"` are ***not*** of type `std::string`, but an ***array of*** `const` ***characters*** which is `null` terminated. The type of `"ab"` is therefore `char const[3]`.
But `"ab"s` is an `std::string`. This requires `using namespace std::literals`:

```
auto printName(std::string name) → void {
  using namespace std::literals;
  std::cout << "my name is: "s << name;
}
```

### 2.3.1. Capabilities

- `std::string` objects are ***mutable***, unlike in Java where `String` objects cannot be modified.
- You can ***iterate*** over the contents of a string.

```
auto toUpper(std::string & value) → void {
  std::transform(cbegin(value), cend(value), begin(value), ::toupper);
}
```

This changes the content of the ***original*** string object.

### 2.3.2. Example

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

auto askForName(std::ostream & out) -> void {
  out << "What is your name? ";
}

auto inputName(std::istream & in) -> std::string {
  std::string name{};
  in >> name;
  return name;
}

auto sayGreeting(std::ostream & out, std::string name) -> void {
  out << "Hello " << name ", how are you?\n";
}

auto main() -> int {
  askForName(std::cout);
  sayGreeting(std::cout, inputName(std::cin));
}
```

## 2.4. INPUT AND OUTPUT STREAMS

*CPPReference: Input/output library*

`std::string` and built-in types represent ***values***. Can be copied and passed-by-value. There is ***no need*** to allocate memory ***explicitly*** for storing the chars. Some objects aren't values, because they can't be copied *(i.e. I/O streams)*. So, these ***functions taking a stream object*** must take it as a ***reference***, because they ***provide a side-effect*** to the stream.

### 2.4.1. `std::cin` and `std::cout`

*CPPReference: std::cin*, *CPPReference: std::cout*

`std::cin` and `std::cout` *(character in/out)* are ***predefined globals***. Should ***only*** be used in the `main()` function.

– ***The bitwise "shift" operators*** read into variables or write values to an output: `std::cin >> x; std::cout << x;`
– ***Multiple values*** can be streamed at once: `std::cout << "the value is " << x << '\n';`
– The stream object is always the ***first element*** in a statement, no stream after the first shift operator.
– Streams have a ***state*** that denotes if I/O was successful or not. Only ***`.good()`*** streams actually do I/O. You need to ***`.clear()`*** the state in case of an error.

### 2.4.2. Reading a `std::string` Value

Reading a `std::string` can ***not go wrong***, unless the stream is already `!good()`. Reads until the first whitespace. The content of the `std::string` is ***replaced***. Maybe it is ***empty*** after reading.

```
#include <istream>
#include <string>
auto inputName(std::istream & in) -> std::string {
  std::string name{}; in >> name; return name;
}
```

### 2.4.3. Reading an `int` Value

Reads the first non-whitespace character, regardless if it is a number or not. ***No error recovery***, One wrong input puts the stream into ***status fail***. Characters ***remain*** in input.

***Boolean Conversion:*** `if (in >> age)` is the `istream` object itself. It converts to `true` if the last reading operation has been successful.

*(More robust version see next page)*

```
#include <istream>
auto inputAge(std::istream & in) -> int {
  int age{-1}; if(in >> age) { return age; } return
-1;
}
```

### More robust reading an `int` Value

Read a line with `getline` and parse it ***as an integer*** until a int is read successfully or `EOF` *(end of file)*.

Read operation in while condition acts as a ***"did the read work?" check***.

Use an ***`std::istringstream`*** as intermediate stream so we can try to parse as `int` after the original `istream` has been read with `getline()`.

```
#include <istream>
auto inputAge(std::istream & in) → int {
  std::string line{};
  while(getline(in, line)) {
    std::isstringstream is {line}; int age{-1};
    if (is >> age) { return age; }
  }
  return -1;
}
```

### 2.4.4. Chaining Input Operations

Result of `in >> age` is the ***`istream` object*** itself. Multiple ***subsequent reads*** are possible. If a previous read already ***failed***, ***subsequent reads*** will fail ***as well***.

```
#include <istream>
auto readSymbols(std::istream & in) → std::string {
  char symbol{}; int count{-1};
  if(in >> symbol >> count) {
    return std::string(count, symbol);
  }
  return "error";
}
```

### 2.4.5. Stream handling on the terminal

If the application is waiting for `EOF` and the input is coming from the terminal, you need to terminate the stream by pressing **CTRL+D**. CTRL+Z terminates the whole application, similar to CTRL+C.

### 2.4.6. An `std__istream's` States

*CPPReference: Stream State Flags and Accessors*

A stream can have ***different states***, depending on what the stream was fed last. A stream always starts as `good()`.

| ***State Bit Set*** | ***Query*** | ***Entered*** |
|---|---|---|
| `<none>` | `is.good()` | initial, `is.clear()` |
| `failbit` | `is.fail()` | input formatting failed |
| `eofbit` | `is.eof()` | trying to read at end of input |
| `badbit` | `is.bad()` | unrecoverable I/O error |

Formatted input on stream is ***must*** check for `is.fail()` and `is.bad()`. If failed, `is.clear()` the stream and ***consume invalid input characters*** before continue. When reading from a `fail`-ed stream, nothing happens.

### 2.4.7. Dealing with Invalid Input

```
auto inputAge(std::istream & in) → int {
  while (in.good()) { // check for good stream state
    int age{-1}; if (in >> age) { return age; } // return if successfully read
    in.clear(); // remove fail flag to continue reading
    in.ignore(); // skip the char that caused the fail (isn't a number)
  } return -1; // return on EOF
}
```

### 2.4.8. Formatting Output

*CPPReference: Input/output manipulators*

There are different ***manipulators*** that can format values for input & output.

```
#include <iostream>
#include <iomanip>
#include <ios>
#include <cmath>

auto main() → int {
  std::cout << 42 << '\t' // '\t' = Tab character
            << std::oct << 42 << '\t' // octal system output
            << std::hex << 42 << '\n'; // hexadecimal system ouput
  std::cout << 42 << '\t' // std::hex is sticky, this is still in hex
            << std::dec << 42 << '\n';
  std::cout << std::setw(10) << 42 // minimal line width, not sticky
            << std::left << std::setw(5) << 43 << "*\n";
            // ...43 if without left, 43... with left
  std::cout << std::setw(10) << "hallo" << "*\n";

  double const pi{std::acos(0.5) * 3};
  std::cout << std::setprecision(4) << pi << '\n';
  std::cout << std::scientific << pi << '\n';
  std::cout << std::fixed << pi * 1e6 << '\n';
}
```

**Code Output** *(◦ = whitespace)*

```
// 42◦◦◦◦◦◦52◦◦◦◦◦◦2a
// 2a◦◦◦◦◦◦42
// ◦◦◦◦◦◦◦◦4243◦◦◦*
// hallo◦◦◦◦◦*
// 3.142
// 3.1416e+00
// 3141592.6536
```

**Other useful manipulators**

***std::ws*** / ***std::skipws***
consumes/skips whitespace
***std::setfill()***
spacing char for `std::setw`
***std::left*** / ***std::right***
sets placement of fill chars
***std::boolalpha***
display booleans as text
***std::uppercase***
print text as uppercase

### 2.4.9. Unformatted I/O

*CPPReference: <cctype>*

The `<cctype>` header contains char conversions and char query functions like `std::tolower()` / `std::toupper()`. The `.get()` / `.put()` functions deal with one char at a time.

```
#include <iostream>
#include <cctype>
auto main() → int { char c{};
  while (std::cin.get(c)) {
    std::cout.put(std::tolower(c)); } }
```

### 2.4.10. The I/O headers: `<iosfwd>`, `<istream>`, `<ostream>`, `<iostream>`

*CPPReference: <iosfwd>, CPPReference: <istream>, CPPReference: <ostream>, CPPReference: <iostream>*

– ***iosfwd:*** Contains only the declarations for `std::istream` / `std::ostream`. ***Use in header files (.hpp).***
– ***istream / ostream:*** Contains implementation of the stream and operators. ***Use in source files (.cpp).***
– ***iostream:*** Contains `std::cin` / `std::cout`. ***Use only in the `main()` function.***

## 3. SEQUENCES AND ITERATORS

*CPPReference: std::array*

`std::array<T, N>` is a ***fixed-size*** container. It is not possible to shrink or grow an array after its creation.
– `T` is a ***template*** type parameter to specify the type of elements the array should contain.
– `N` is a ***positive integer*** to specify the number of elements in the array.

Both can be ***deduced*** from the initializer, so you can write `std::array name {1, 2, 3, 4, 5}`. This only works if you write out the elements inside.

```
#include <array>
std::array<int, 5> name{1, 2, 3, 4, 5};
```

The ***size*** of the array must be ***known*** at compile-time and ***cannot be changed***. Otherwise, it contains `N` default-constructed elements: `std::array<int, 5> emptyArray{}` contains 5 zeroes. The size can be queried using ***`.size()`***.

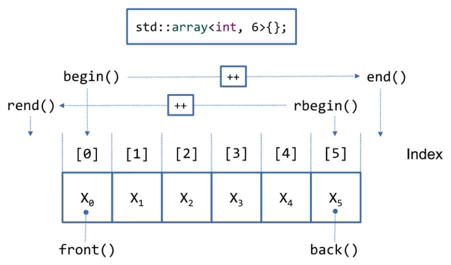
Elements of the array can be accessed via the ***subscript operator []*** or the ***`.at()` member function***. `.at()` throws an exception on invalid index access, while `[]` has undefined behavior.

Plain C-style arrays should be avoided, as they are only passed as pointers, thus the array size gets lost. This can lead to memory errors!
~~`int arr[]{1, 2, 3}`~~

### 3.0.1. Array Iterators

*CPPReference: std::array Iterators*

- ***`begin`:*** returns an Iterator to the first element of the array
- ***`end`:*** returns an Iterator to ***after*** the last element of the array
- ***`rbegin`:*** returns a reverse Iterator to the ***last*** element of the array
- ***`rend`:*** returns a reverse Iterator to ***before*** the first element of the array



Whenever a iterator is incremented, it will point to the next element in line. To access the element an iterator points to, the iterator needs to be ***dereferenced*** with the `*` operator *(indirection operator)*:

```
std::array arr{42, 1337, 666};
auto *iterator = arr.begin() + 1; // "auto *" because iterator is a pointer-like type
int secondElement = *iterator; // secondElement = 1337
```

***Reverse Iterators*** will iterate the array from the back, meaning the last element will be accessed first. The next element will be the second last element and so on.
All of the Iterators also have a ***const version*** *(`cbegin`, `cend`, `crbegin`, `crend`)* which returns a `const` Iterator, meaning the element the iterator points to can't be modified.

## 3.1. STD::VECTOR

*CPPReference: std::vector*

`std::vector<T>` is a ***Container***. There is ***no need*** to allocate the elements inside, as it already contains them *(unlike Java, where a `ArrayList` contains references to its elements)*. T is a ***template*** type parameter to specify the type of the elements to store.
`std::vector` can be initialized with a list of elements, but the list ***can be empty:*** `std::vector<double> vd{}`. When an ***initializer*** is given, the element type can be deduced.

```
#include <vector>
std::vector<int> name{1, 2, 3, 4, 5}
```

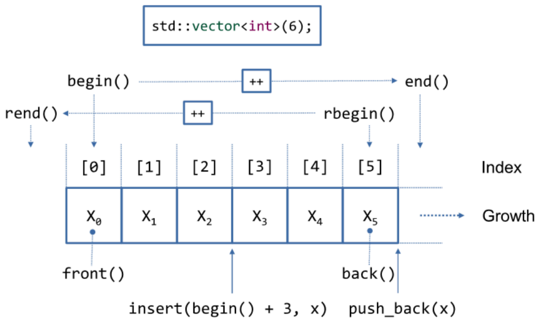
During ***initialization***, the initial size of the vector can be specified inside parenthesis
`std::vector<int>(6) /* size = 6 */ ≠ std::vector<int>{6} /* has one 6 inside */`

### 3.1.1. Vector Iterators

*CPPReference: std::vector Iterators*

In addition to the Iterators of `std::array` *(see "Array Iterators" (Page 9))*, `std::vector` has two additional functions to work with.

- ***`.insert(<iterator>, <value>)`:*** Insert a value at the position the iterator points to. All succeeding elements are moved one position up *(inefficient!)*.
- ***`.push_back(<value>)`:*** Inserts a value at the end of the vector *(more efficient)*



```
std::vector vec{1, 2, 4};
vec.insert(vec.begin() + 2, 3); // vec = {1, 2, 3, 4}
```

### 3.1.2. Array vs. Vector

Arrays are ***stack allocated***, Vectors are ***heap allocated***. Arrays should be used when the number of elements doesn't change, otherwise Vectors should be used.

## 3.2. ITERATION

### 3.2.1. Index-based Iteration

Vectors can be accessed via ***`for`-Loop***. The type of the index variable is ***`size_t`*** *(Works on all OS and platforms, `int` may cause problems)*. Using `.at()` prevents undefined behavior on invalid index access. ***Caution: Only use if the actual index value is required!***

```
for (size_t i = 0; i < v.size(); ++i) {
  std::cout << v.at(i) << '\n';
}
```

### 3.2.2. Element Iteration (Range-Based for, foreach)

***No index error*** possible, works with ***all containers***, even value lists `{1,2,3}`

| | ***`const`*** ***Element cannot be changed*** | ***`non-const`*** ***Element can be changed*** |
|---|---|---|
| ***`reference`*** *(marked with & operator)* ***Element in Vector is accessed*** *for big elements and changes to the original* | `for (auto const & cref : v) {`<br>`  std::cout << cref << '\n';`<br>`}` | `for (auto & ref : v) {`<br>`  ref *= 2;`<br>`}`<br>*Modifies elements in the original container* |
| ***`copy`*** ***Loop has own copy of the element*** | `for (auto const ccopy : v) {`<br>`  std::cout << ccopy << '\n';`<br>`}`<br>*constant copy is rarely used* | `for (auto copy : v) {`<br>`  copy *= 2;`<br>`  std::cout << copy << '\n';`<br>`}`<br>*Modifies elements in the copied container* |

### 3.2.3. Iteration with Iterators

A ***range-based for-loop*** uses ***iterators*** internally. They can also be used in a for-loop, but only useful if the iterator itself is required inside the loop. Otherwise ranged-for-loops or algorithms are preferred for memory safety reasons.

Start with `std::begin(vec)` and compare if the current iterator is not equal to `std::end(vec)`. The current element can be accessed with `*iterator`; if the iterator and container are non-`const`, elements can also be modified this way. To have read-only access to the container, use `std::cbegin(vec)` and `std::cend(vec)`.

```
for (auto it = std::cbegin(v); it != std::cend(v); ++it) {
  std::cout << *it << ", ";
}
```

For more Information on Iterators, see “Iterators” (Page 36).

## 3.3. ITERATOR ALGORITHMS

Algorithms perform ***frequently used operations on ranges and containers***, such as counting values, copying or searching for elements. Each algorithm takes iterators as arguments, the range(s) of elements to apply an algorithm to is specified by them.

For a more in-depth look at algorithms, see chapter “STL Algorithms” (Page 37).

Containers ***cannot*** be used with algorithms ***directly***. Iterators ***connect*** containers and algorithms.

| ***Container*** | ***Iterators*** | ***Algorithms*** |
|---|---|---|
| `std::vector<T>`<br>`std::string`<br>`std::set<T>`<br>`std::map<K,V>`<br>`...` | `std::begin()`<br>`std::end()`<br>`std::rbegin()`<br>`std::rend()`<br>`...` | `std::count(b, e, val)`<br>`std::ranges::count(r, val)`<br>`std::find(b, e, val)`<br>`std::accumulate(b, e, start)`<br>`std::copy(b, e, b_target)`<br>`...` |

### 3.3.1. Using Iterators with Algorithms

*CPPReference: Algorithms library*, *CPPReference: Iterator library* `#include <algorithm>`
*CPPReference: std::begin, std::cbegin*, *CPPReference: std::end, std::cend*

***Avoid programming your own loops!*** The corresponding algorithm is ***more correct***, ***more readable*** and has ***better performance***.

To also support ***containers*** and other data types that do not have a `.begin()`/`.end()` etc. member function *(such as plain-C-arrays)* the iterator library provides `std::begin()`/`std::end()` etc. These are functionally the same as the member functions and can for the most part be used interchangeably.

### 3.3.2. Basic Examples of Algorithm use

#### Counting values

*CPPReference: std::count*, *CPPReference: std::ranges::count*
Returns the number of occurrences of a value in range. Works with all ranges denoted by a pair of iterators.

```cpp
#include <algorithm>
#include <iterator>
auto count_blanks(std::string s) → size_t {
  return std::count(s.cbegin(), s.cend(), ' ');
}
```

#### Summing up values

*CPPReference: std::accumulate*
Applies the + operator to elements, requires the initial value *(here `0`)*.

```cpp
#include <algorithm>
#include <iterator>
#include <numeric>
std::vector<int> v{5, 4, 3, 2, 1};
std::cout << std::accumulate(std::cbegin(v), std::cent(v), 0) << " = sum\n";
```

#### Number of elements in range

*CPPReference: std::distance*
Containers provide a `size()` member function, useful if you only have iterators as `size()` may be unavailable inside an algorithm. Both `size()` and `std::distance()` provide the same value.

```cpp
#include <algorithm>
#include <iterator>
void printDistanceAndLength(std:string s) {
  std::cout << "distance: " << std::distance(s.begin(), s.end()) << '\n';
  std::cout << "in a string of length: " << s.size() << '\n';
}
```

### 3.3.3. `std::for_each` Algorithm

*CPPReference: std::for_each*
The most basic algorithm. Like a `for` statement, executes an action ***for each element in a range***. The ***last argument*** is a ***function*** that takes one parameter of the element `type` *(in the example below, one `int`)*. Most of the time, the function is a ***lambda.***

```cpp
#include <algorithm>
#include <iterator>
auto print(int x) → void {
  std::cout << "print: " << x << '\n';
}
auto printAll(std::vector<int> v) → void {
  std::for_each(std::crbegin(v), std::crend(v), print); // for each element, print() is run
}
```

```
auto values = std::vector{3,0,1,4,0,2};
auto f = [](auto v) {};
std::for_each(begin(values), end(values), f) // f is applied to each element
```

### 3.3.4. Lambda Functions Basics

Using `std::cout` outside `main` is discouraged. If we want to print to a given `std::ostream`, we need to use a ***lambda structure***. For more detailed information about Lambda Functions, see “Lambda Functions” (Page 17).

```
[<capture>](<parameters>) → <return-type> { <statements> }
```

| *Term* | *Definition* |
|---|---|
| Capture | Variables from outside the lambda to access inside of the lambda. Can either be copies or references *([&x]: by reference, [x]: by value, [=]: all local variables by value, [&]: all local values as references)* |
| Parameters | New variables to be used inside the lambda. When used with algorithms, there is usually one parameter that contains the current element of the range/container. |
| Return Type | Return type of the lambda function. Can be ***omitted*** if `void` or consistent return statements in the body. |

A ***lambda expression*** creates a function object that can be passed to an algorithm. ***Capture names variables*** are taken from the surrounding scope.

```
auto printAll(std::vector<int> v, std::ostream & out) → void {
// Lambda captures "out", can be used inside lambda
std::for_each(std::cbegin(v), std::cent(v), [&out](auto x) /* → no return-type */ {
  out << "print: " <<  x << '\n'; }); }
```

### 3.3.5. `std::ranges`

*CPPReference: Ranges library, CPPReference: Constrained Algorithms (list of all ranges algorithms)* `#include <algorithms>`

A lot of times, inside an algorithm we ***iterate*** from the ***start to the end*** of a container. So the first two parameters of an algorithm are ***`begin()`*** and ***`end()`***. To simplify this, most algorithms have a version in the `std::ranges` namespace, where only the container is taken as an argument.

```
#include <algorithm>
auto printAll(std::vector<int> v, std::ostream & out) → void {
  // No v.begin() and v.end(), just "v" is enough :)
  std::ranges::for_each(v, [&out](auto x) {
    out << "print: " << x << '\n'; }); }
```

#### Appending Elements to an `std::vector<T>`

*CPPReference: std::vector<>.push_back(), CPPReference: std::vector<>.insert()*

- ***Append:*** `v.push_back(<value>)` *(Append at the back, relatively efficient)*
- ***Insert anywhere:*** `v.insert()(<iterator-position>, <value>)` *(Has to move succeeding elements, inefficient)*

When using the ***`std::copy`*** algorithm, the ***target*** has to be an ***iterator*** too.
*CPPReference: std::copy, CPPReference: std::ranges::copy*

```
std::copy(<input-begin-iterator>, <input-end-iterator>, <output-begin-iterator>);
std::ranges::copy(<input-range>, <output-begin-iterator>);
```

***Caution:*** Using `begin()` or `end()` as the output begin iterators are not allowed, because they can’t insert values in a container. Additionally `end()` is not allowed, since it is outside of the allocated memory. Instead, we can use an ***`std::back_inserter`***, which performs `push_back()` for us:

```
std::vector<int> source{1, 2, 3}, target{};
// Use either ranges or non-ranges copy()
std::copy(source.cbegin(), source.cend(), std::back_inserter(target));
std::ranges::copy(source, std::back_inserter(target));
```

## Filling an `std::vector<T>` with values

*CPPReference: std::fill*, *CPPReference: std::ranges::fill*

Requires a vector with ***existing elements*** to be overwritten, or to create a vector directly filled with the wanted elements.

***Manual resize of vector***

```
std::vector<int> v{};
v.resize(10); // set size of vector to 10
std::fill(std::begin(v), std::end(v), 2);
std::ranges::fill(v, 2);
```

***Initialize vector with correct size***

```
std::vector<int> v(10); // Caution: () not {}!
std::fill(std::begin(v), std::end(v), 2);
std::ranges::fill(v, 2);
// Use either ranges or non-ranges fill()
```

## Filling an `std::vector<T>` with different values

*CPPReference: std::generate*, *CPPReference: std::ranges::generate*, *CPPReference: std::generate_n*, *CPPReference: std::ranges::generate_n*

The algorithms ***`std::generate()`*** and ***`std::generate_n()`*** fill a range with computed values:

***`back_inserter` on empty vector***

```
std::vector<double> powerOfTwos{};
double x{1.0};
std::generate_n(
  std::back_inserter(powerOfTwos),
  5, // Number of elements
  [&x] {return x *= 2.0; })
```

***Fill vector with specified size***

```
std::vector<double> powerOfTwos(5);
double x{1.0};
std::ranges::generate(
  powerOfTwos,
  [&x] {return x *= 2.0;}
);
```

***`std::iota()`*** *(named after the Greek letter* I*)* fills a range with ***subsequent values*** *(1,2,3)*:
*CPPReference: std::iota*

```
#include <numeric>
#include <iterator>
#include <algorithm>
std::vector<int> v(100);
std::iota(std::begin(v), std::end(v), 1); // fills v with numbers 1-100
```

## Finding and Counting Elements

*CPPReference: std::find, std::find_if*, *CPPReference: std::ranges::find, std::ranges::find_if*

***`std::(ranges::)find()`*** and ***`std::(ranges::)find_if()`*** return an iterator to the first element that matches the value or condition. If no match exists, the end of the range is returned.

```
#include <iterator>
#include <algorithm>
auto zero_it = std::ranges::find(v, 0); // find first 0
if (zero_it == std::end(v)) {
  std::cout << "no zero found \n";
}
```

***`std::(ranges::)count()`*** and ***`std::(ranges::)count_if()`*** return the number of matching elements in a range.
*CPPReference: std::count, std::count_if*, *CPPReference: std::ranges::count, std::ranges::count_if*

```
#include <iterator>
#include <algorithm>
std::cout << std::ranges::count(v, 42) << " times 42\n";
auto isEven = [](int x) { return !(x % 2); };
std::cout << std::ranges::count_if(v, isEven) << " even numbers\n";
```

## 3.4. ITERATORS FOR I/O

Streams ***cannot*** be used with algorithms directly. Instead, ***`std::ostream_iterator`*** and ***`std::istream_iterator`*** are used to take ***multiple values*** from the `istream` or put multiple values on the `ostream`.

### 3.4.1. `std::ostream_iterator`

*CPPReference: std::ostream_iterator*

A `std::ostream_iterator` can be used to ***copy*** the values of a container to a `std::ostream`. It can also take a optional ***delimiter character*** to separate the output values. In the example below, the vector values are printed with a comma and a space between them.

```
std::ranges::copy(v, std::ostream_iterator<int>{std::cout, ", "});
```

The ***output*** is of the type `T` given to the `std::ostream`. There is no `end()` marker needed for the output, id ends when the input range ends.

### 3.4.2. `std::istream_iterator`

*CPPReference: std::istream_iterator*

`std::istream_iterator<T>` ***reads*** values of type `T` from the given `std::istream`. To mark the ***end of the input*** for an algorithm that requires it, a empty `std::istream_iterator<T>{}` is needed. It ends when the stream is no longer `good()` *(i.e. no more characters in input or characters that can't be assigned to T)*.

```
std::istream_iterator<std::string> in{std::cin};
std::istream_iterator<std::string> eof{}; // dummy stream that acts as in.end()
std::ostream_iterator<std::string> out{std::cout, " "};
std::copy(in, eof, out); // writes chars from input directly to output
```

***`std::ranges::istream_view<T>`*** combines `in` and `eof`:

*CPPReference: std::ranges::istream_view*

```
std::ranges::istream_view<std::string> in{std::cin};
std::ostream_iterator<std::string> out{std::cout, " "};
std::ranges::copy(in, out);
```

### 3.4.3. Type Alias

*CPPReference: Type alias*

Type names can be given alias names. Useful if ***long type names*** occur ***more than once***.

```
using <alias-name> = <type>;
```

```
using input = std::istream_iterator<std::string>;
input eof{};
input in{std::cin};
std::ostream_iterator<std::string> out{std::cout, " "};
std::copy(in, eof, out);
```

**Unformatted Input: `std::istreambuf_iterator`**

*CPPReference: std::istreambuf_iterator*

`std::istream_iterator` skips whitespaces. For an ***exact copy***, we need ***`std::istreambuf_iterator<char>`***. Works only with ***char-like*** types.

```
using input = std::istreambuf_iterator<char>;
input eof{};
input in{std::cin};
std::ostream_iterator<char> out{std::cout, " "};
std::copy(in, eof, out);
```

**Filling an `std::vector<T>` from Standard Input**

*CPPReference: std::back_inserter*

***With `back_inserter`***

```
using input = std::ranges::istream_view<int>;
std::vector<int> v{};
std::ranges::copy(input{std::cin},
std::back_inserter(v));
```

***Construct vector from iterators***

```
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> const v{input{std::cin}, eof};
```

# 4. FUNCTIONS AND EXCEPTIONS

## 4.1. FUNCTIONS

| | `const` ***Parameter cannot be changed*** | `non-const` ***Parameter can be changed*** |
|---|---|---|
| `reference` Argument on call-site is accessed | `auto f(std::string const & s) → void {`<br>`  // no modification`<br>`  // efficient for large objects`<br>`}`<br>*For non-copyable objects like Streams* | `auto f(std::string & s) → void {`<br>`  // modification possible`<br>`  // side-effect also at call-site`<br>`}`<br>*When side-effect is required at call-site* |
| `copy` Function has its own copy of the parameter | `auto f(std::string const s) → void {`<br>`  // no modification`<br>`  // used for maximum constness`<br>`}`<br>*Could prevent changing the parameter inadvertently* | `auto f(std::string s) → void {`<br>`  // modification possible`<br>`  // side-effect only locally`<br>`}`<br>*Default for Parameters* |

The ***call-site*** always looks the same: `std::string name{"John"}; f(name);`

It is ***not possible*** to pass a `const` argument to a non-`const` reference, because the compiler can't guarantee that the object will not be changed in the non-`const` function!

### 4.1.1. Function Return Type

Use ***by value***. The default for return types.
`auto f() → type;` This creates a temporary at the call-site.

```
auto create() → std::string {
  std::string name{"John"};
  return name;
}
```

```
auto main() → int {
  std::string name = create();
  // Temporary stored in the name of 'create'
}
```

***Other ways*** to specify a return type:

– ***Const value return type*** `auto f() → type const;`
  Annoying to deal with: The value that the caller owns cannot be modified. This should be avoided!
– ***Reference return type:*** `auto f() → type &;`
  Modifiable reference, i.e. for accessing elements in a container
– ***Const reference return type:*** `auto f() → type const &;`
  Read-only view of an object

Never return a reference to a ***local variable*** *(Variable goes out of scope when method ends, leads to undefined behavior)*!

***The trailing return-type could be omitted.*** The actual return type will then be deduced from the return statements in the function's body.

### 4.1.2. Call Sequence in argument evaluation

Within a single expression *(i.e. function call)* the sequence of evaluation is ***undefined behavior***! A later function call could be executed before an earlier one.

```
auto sayGreeting(std::ostream &out, std::string name1, std::string name2) → void {
  out << name1 << " says Hello to " << name2;
}

int main() { // the second inputName call could be run before the first
  sayGreeting(std::cout, inputName(std::cin), inputName(std::cin));
}
```

## 4.2. FUNCTION OVERLOADING

The ***same function name*** can be used for ***different functions*** if the functions have ***different parameter numbers*** or ***types***. Just different ***return types*** does not count and leads to ***ambiguity***. The resolution of overloaded methods happens at compile time *(Ad hoc polymorphism)*

For Overloading in Templates, see "Template Overloading" (Page 44).

```
auto incr(int & var) → void;
auto inct(int & var, unsigned delta) → void;
```

**Overloading Ambiguity**

```
auto factorial(int n) → int { ... }
auto factorial(double n) → double { ... }
factorial(3); factorial(1e2); // OK
factorial(10u); factorial(1e1L); // Compiler doesn't know what to cast to
```

### 4.2.1. Default Arguments

A ***function declaration*** can provide ***default arguments*** for its parameters ***from the right*** *(It is not valid to have a non-default argument after a default argument)*. Default arguments can be ***omitted*** when calling the function. The default value ***should not be repeated*** in the ***function definition***.

```
// declaration with default value
auto incr(int & var, unsigned delta = 1) → void;
// definition without default value
auto incr(int & var, unsigned delta) → void;

int counter{0};
// function calls
incr(counter); incr(counter, 5);
```

## 4.3. FUNCTIONS AS PARAMETERS

Functions are ***"first class" objects*** in C++. This means, they can be ***passed as arguments*** to other *(higher-order)* functions and they can be ***kept in reference variables***. **Drawback:** A function parameter declared in this way does ***not accept*** a ***lambda with a capture*** *(the variables in brackets from outside the lambda)*.

```
// 2nd Parameter: function f with a double parameter and double as return type
auto applyAndPrint(double x, auto f(double) → double) → void {
  std::cout << "f(" << x << ") = " << f(x) << '\n';
}

// reference variables
auto (&ref)(double) → double // modern C++ with trailing return type
double (&ref)(double) // classic style with return type in front
```

### 4.3.1. Modern approach: `std::function` Template

*CPPReference: std::function*

`#include <functional>`

This also allows passing ***lambdas with captures***.

```
// 2nd Parameter: function f with a double parameter and double as return type
auto applyAndPrint(double x, std::function<auto(double) → double> f) → void {
  std::cout << "f(" << x << ") = " << f(x) << '\n';
}
```

```
auto main() → int {
  double factor{3.0};
  auto const multiply = [factor](double value){
    return factor * value;
  };
  applyAndPrint(1.5, multiply);
}
```

Function template methods can also be ***stored in variables*** with these types:

```
std::function<auto(double) → double>
std::function<double(double)>
```

The ***type definition*** could also be written in a shorter syntax:

```
auto applyAndPrint(double x, std::function<auto(double) → double> f) → void { ... } //old
auto applyAndPrint(double x, std::function<double(double)) → void { ... } //new
// syntax: return type ( parameter list )
```

## 4.4. LAMBDA FUNCTIONS

*CPPReference: Lambda expressions*

Lambda Functions are ***inline*** functions.

- ***`auto`*** for function variable to store Lambda function in
- ***`[]`*** introduces the Lambda function *(can contain captures to access specific or all variables from scope: see below)*
- ***`(Parameters)`*** as with other functions, but optional if empty.
- ***Trailing return type:*** Usually deduced and thus optional, but can be explicitly specified to automatically cast the return value
- ***Body*** with block statements.

```
auto g = [](char c) → char {
  return std::toupper(c);
};
g('a'); // Returns 'A'
```

**Capturing a local variable by value**

Local copy ***lives as long as the lambda lives***. It is ***immutable***, unless lambda is declared `mutable`. The lambda can be passed to other functions, as the captured variable is bound to the lambda.

```
int x = 5;  // stays 5
auto l = [x]() mutable {
  std::cout << ++x;
};
```

**Capturing a local variable by reference**

***Allows modification*** of the captured variable. ***Side-effect is visible*** in the surrounding scope, but referenced variable ***must live at least as long*** as the lambda lives, else null-reference possible. The ***mutability*** depends on if the referenced object is mutable. If if the captured variable is a local variable, problems are caused when this lambda is passed to other functions *(variable out of scope)* .

```
int x = 5; // changed by l
auto const l = [&x]() {
  std::cout << ++x;
};
```

**Capturing all (referenced) local variables by value**

Variables used in the lambda will be ***copied***. They are still `const` unless the lambda is `mutable`

```
int x = 5;  // stays 5
auto l = [=]() mutable {
  std::cout << ++x;
};
```

**Capturing all (referenced) local variables by reference**

Variables used in the lambda will be ***accessible*** in the lambda. Will allow modification, unless originally declared `const`

```
int x = 5; // changed by l
auto const l = [&]() {
  std::cout << ++x;
};
```

**Specify new local variable inside capture**

New variable in capture has type `auto`. Can be modified if lambda is `mutable`. The ***specified value*** is only used in the ***definition***, ***not*** in the ***function call***.

```
auto squares = [x=1]() mutable
{
  std::cout << (x *= 2);
};
```

**Capturing `this` pointer**

Allows accessing and modifying members of the class.

```
struct S {
  auto foo() → void {
    auto square = [this] {
      member *= 2;
    };
  }
private: int member{};
};
```

## 4.5. FAILING FUNCTIONS / ERROR HANDLING

Functions can fail when ***a contract cannot be fullfilled:***

– ***Precondition is violated:*** Negative index, divisor is zero, etc. Usually caller provided wrong arguments.
– ***Postcondition could not be satisfied:*** Resources for computation not available, cannot open a file, …

### 4.5.1. Functionality Guarantees (Contract)

What to do if a function cannot fulfill its purpose?

1. ***Ignore the error*** and provide potentially undefined behavior *(Relies on the caller to satisfy all preconditions. Viable only if not dependent on other resources. Most efficient and no checks needed but hard to handle for the caller. Should be done carefully!)*

2. ***Return a standard result*** to cover the error *(Reliefs the caller, can hide underlying problems. Often better if caller can specify its own default value)*

```
auto inputNameWithDefault(std::istream & in, std::string const & def = "anon") → std::string
{
  std::string name{}; in >> name; return name.size() ? name : def;
}
```

3. ***Return an error code*** or error value *(Only feasible if result domain is smaller than return type. POSIX: Error Code `'-1'`. Burden on the caller to check the result.)*

```
auto contains(std::string const & s, int number) → bool { // "artificial" npos value
  auto substring = std::to_string(number); return s.find(substring) ≠ std::string::npos;
}
```

A ***more graceful way*** to handle this situation is to use ***`std::optional<T>`***: It can either ***contain a value or not***. It encodes the possibility of failure in the type system. Requires explicit access of the value at the call site *(checking the boolean `has_value()`)*

```
std::optional<std::string> name = inputName(std::cin);
if (name.has_value()) { std::cout << "Name: " << name.value(); }
```

4. ***Provide an error status*** as a side-effect *(Requires reference parameter, annoying, when error variable must be provided)*

```
auto connect(std::string url, bool& error) → int {
  // set error when an error occured
}
```

5. ***Throw an exception*** *(Prevent execution of invalid logic by throwing an exception)*

```
void sayGreeting(std::ostream & out, std::string name) {
  if (name.empty()) { throw std::invalid_argument{"Empty name"}; }
  out << "Hello " << name << ", how are you?\n";
}
```

#### 4.5.2. Function with “Narrow Contract”

Functions that have a ***precondition*** on their caller. When not all possible argument values are useful for the function. Do ***not*** use exceptions as a second means to return values.

```
auto squareRoot(double x) → double {
  if (x < 0) {
  throw std::invalid_argument
    {"sqare_root imaginary"}; }
  return std::sqrt(x);
}
```

### 4.6. EXCEPTIONS

*CPPReference: Diagnostics library*, *CPPReference: <stdexcept>*

```
#include <stdexcept>
```

#### 4.6.1. Throwing Exceptions

Any *(copyable)* type can be thrown. There are ***no means to specify*** what could be thrown, but you should always throw exceptions *(either prediefined from* `<stdexcept>` *or derived from* `std::exception`*)*. There is also ***no meta-information*** available: ***no stack trace***, ***no source position*** of throw. Exception thrown while exception is propagated results in ***program abort***.

```
throw std::invalid_argument{"Description"};
throw 15;
// Do not use "throw new ..."
// This will throw a pointer and cause problems
```

#### 4.6.2. Catching Exceptions

Try-catch block as in Java. Principle: ***Throw by value, catch by const reference***. Avoids unnecessary copying, allows dynamic polymorphism for class types.
The ***sequence*** of catches is significant. ***First match wins.*** Catch all with `(...)` must be the last catch. Caught exceptions can be ***rethrown*** with `throw`. C++ does not have a `finally` clause.

```
try {
  throwingCall();
} catch (type const & e) {
  // Handle type exception
} catch (type2 const & e) {
  // Handle type2 exception
} catch (...) {
  // Handle other exception types
}
```

#### 4.6.3. Exception Types

*CPPReference: Exception Categories*

The Standard Library has some ***pre-defined exception types*** that you can use in `<stdexcept>`. All have a constructor parameter for the “reason” of type `std::string`. `std::exception` is the base class.

`std::exception, std::runtime_error, std::logic_error, std::out_of_range, std::invalid_argument, ..`

#### 4.6.4. Testing for Exceptions with Catch2

– ***`REQUIRE_THROWS(<code>)`:*** Tests that any type of exception is thrown

```
TEST_CASE("throw any exception on negative square_root") {
  REQUIRE_THROWS(square_root(-1.0));
}
```

– ***`REQUIRE_THROWS_AS(<code>, <exception_type>)`:*** Tests that a specific type of exception is thrown

```
TEST_CASE("throw std::out_of_range on empty vector at()") {
  std::vector<int> empty_vector{};
  REQUIRE_THROWS_AS(empty_vector.at(0), std::out_of_range);
}
```

– ***`REQUIRE_THROWS_WITH(<code>, <string or string matcher>)`:*** Test for exception message content

```
TEST_CASE("parseInt throws with message") {
  REQUIRE_THROWS_WITH(parseInt("one"), "parse error - invalid digits in 'one'");
}
```

#### 4.6.5. Keyword `noexcept`

*CPPReference: noexcept specifier, CPPReference: noexcept operator*

Functions can be declared to explicitly ***not throw*** an exception with the ***`noexcept`*** keyword. If an exception is thrown from a `nowexception` function, the program ***will terminate***.

```
auto add(int lhs, int rhs) noexcept → int {
  return lhs + rhs;
}
```

#### 4.6.6. Summary

A ***good function*** does ***one thing well*** and is named after that *(High cohesion)*. Has only ***few parameters***, consists of only a few lines without deeply nested control structure. ***Provides guarantees*** about its result and is ***easy to use***. Pass ***parameters*** and return ***results*** by ***value***, unless there is a good reason not to.

# 5. CLASSES AND OPERATORS

## 5.1. CLASSES

*CPPReference: Classes*

A ***good class*** does ***one thing well*** *(High cohesion)* and is named after that. It consists of ***member functions*** with ***only a few lines***. Has a ***class invariant*** *(Consistency, provides a guarantee about its state)*. Is ***easy to use*** without complicated protocol sequence requirements.

```
class <GoodClassName> {
  <member variables>
  <constructors>
  <member functions>
};
```

### 5.1.1. Declaration / Implementation Example

A class ***defines a new type***. At the ***end*** of a class definition, a semicolon is required.
The definition/declaration is in a ***header file*** *(*.hpp)* and the implementation in a ***source file*** *(*.cpp)*. In the implementation, the member functions are not wrapped in a class `class xy { ... }`, instead every function has the corresponding class name as a prefix `xy::`

```
// File Date.hpp
#ifndef DATE_HPP_ // include guard
#define DATE_HPP_

class Date { // keyword for defining a class
    // Member Variables
    int year, month, day;
  public: // access specifier
    // Constructor:
    Date(int year, int month, int day);
    // Member Function 1
    static auto isLeapYear(int year) → bool;

  private: // access specifier
    // Member Function 2
    auto isValidDate() const → bool;
}; // Don't forget this semicolon!
#endif // include guard
```

```
// File Date.cpp
#include "Date.hpp"

// Constructor
Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
    /* ... */
}

// Member Function 1
auto Date::isLeapYear(int year) → bool {
  /* ... */
}

// Member Function 2
auto Date::isValidDate() const → bool {
  /* ... */
}
```

**Using the Class**

```
#include "Date.hpp"

auto dating() → void {
  Date today{2016, 10, 19};
  auto thursday{today.tomorrow()}; // Copy
  Date::isLeapYear(2016); // Static Member Function

  Date invalidDate{2016, 13, 1};
}
```

## 5.2. DECLARATION IN THE HEADER FILE

CPPReference: Class Declaration, CPPReference: Source file inclusion

### 5.2.1. Include Guard

Ensures that the content of a header file is only included ***once***. Eliminates ***cyclic dependencies*** of `#include` directives. Prevents violation of the ***one definition rule***, see "One Definition Rule (ODR)" (Page 3).

```
#ifndef <name>
#define <name>
#endif
```

CPPReference: keyword struct, CPPReference: keyword class

There are two different keywords for defining a class: `class` and `struct`. Their only difference is the ***default visibility*** of their member functions and variables: `private` for `class`, `public` for `struct`.

### 5.2.2. Access specifiers

CPPReference: Access specifiers

– ***`private:`*** visible only inside the class, for hidden data members
– ***`protected:`*** also visible in subclasses
– ***`public:`*** visible from everywhere, for the interface of the class

It is possible to declare ***multiple blocks*** of the same access specifier, but best practice is to ***only use one block.***

### 5.2.3. Member variables

Have a ***type*** and a ***name***: `<type> <name>`. Do ***not*** make member variables ***`const`***, as it prevents ***copy assignment***. The definition order specifies the initialization order of the class members.

### 5.2.4. Static Members

CPPReference: Static members

Classes can also have static member functions/variables that don't need an instance to be called/accessed. In the header, they are defined with the `static` keyword.

```
class Date {
  static const Date myBirthday;
  static Date today{};
  static auto isLeapYear(int year) → bool;
}
```

### 5.2.5. Constructors

CPPReference: Constructors

A constructor *(often shortened to `ctor`)* is a ***function with the name of the class*** that can be called to create an instance of this class. It is a ***special member function***. It has ***no return type*** and can have an ***initializer list*** for member initialization.

```
<class name>() {}
<class name>(<parameters>)
  : <initializer-list>
{}
```

The ***member initializer list*** can take the parameters and directly assign them to member variables. The initialization ***order*** depends on the order of the members inside of the class, not the order in the initializer list. There can be more code after the initializer list, for example to perform validation of the parameters before assigning them. If there is no code after it, empty {} are still required!

```
Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
    if (month < 1 || month > 12) {
      throw std::invalid_argument
        {"Invalid month!"};
    }
    // more error checking for day & year
  }
```

#### Implicit Special Constructors

CPPReference: Default Constructor,
CPPReference: Copy Constructor

***Default Constructor:*** `Date d{};`
The Default Constructor is a constructor that can be called with ***no arguments.*** It is ***implicitly available*** if there are no other declared constructors. It has to ***initialize*** member variables with default values. If

```
class Date {
  public:
    Date(int year, int month, int day);
    // Default-Constructor
    Date(); // implicit
    Date() = default; // explicit

    // Copy-Constructor
```

there are other constructors, it can be ***explicitly*** made available with the keyword `default`.

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

***Copy Constructor:*** `Date d2{d};`
The copy constructor can be called with an object of the same class and copies the content of the argument. Is implicitly called when an object is assigned to a new variable. ***Copies*** all member variables. Has one `<own-type> const &` parameter. Is ***implicitly available***, unless there is a move constructor *(C++ Advanced topic)* or assignment operator. Usually no need for explicit implementation.

### 5.2.6. Defaulted Constructor

`<ctor-name>() = default;`
If any constructor is implemented, the implicit default constructor is no longer available. If it is still desired to keep it, instead of reimplementing it manually, it can be ***defaulted***. This adds it back with the same behavior as when it was implicitly available. Defaulting is also possible for default destructor, copy/move constructor and copy/move assignment operator.

```
class Date {
  int year{9999}, month{12}, day{31}
  // explicitly re-add default ctor
  Date() = default;
  Date(int year, int month, int day);
};
```

**Type-conversion Constructor**

`explicit <ctor-name>(<OtherType>);`
Constructors with a ***single argument*** or with default arguments for all parameters after the first can be called with any type as its argument, as long as it is ***implicitly convertible*** to the specified type *(i.e. a `double` argument for a `int` parameter).* This implicit conversion can cause errors. To disable this, constructors like this can be declared ***`explicit`***, so only the specified type will be taken as an argument.

```
class Date {
  public:
    Date(int year, int month, int day);
    // Type-conversion Constructor
    // Is marked 'explicit' to prevent implicit
    // conversion of the 'year' parameter
    // i.e. from double or char
    explicit Date(
      int year, int month = 1, int day = 1);
};
```

**Initializer List Constructor**

`Container box{item1, item2, item3};`
Has one `std::initializer_list<T>` parameter. Does ***not*** need to be marked ***`explicit`***. Initializer List constructors are preferred if a variable is initialized with `{}`:

```
std::vector v(4,10)  // returns 10, 10, 10, 10
std::vector v{4, 10} // returns 4, 10
```

```
struct Container {
  Container() = default;
  // Initializer List Constructor
  Container(
    std::initializer_list<Element> elements);
  private:
    std::vector<Element> elements{};

};
```

**Deleted Constructors**

`<ctor-name>() = delete;`
To ***delete implicit constructors***, you can delete them by adding the keyword `delete`. Possible for default constructor/destructor, copy/move constructor and copy/move assignment operator.

```
class Banknote {
  int value;
  // Delete default copy constructor
  // Instances can't be copied anymore
  Banknote(Banknote & const) = delete;
};
```

**Delegating Constructors**

Constructors can call ***other*** constructors, similar to Java. The Constructor call has to be in the ***member initializer list***.

```
Date::Date(int year, int month, int year)
  : Date{year, Month(month), day} {}
```

```
class Date {
  // ...
  Date(int year, Month month, int day);
  Date(int year, int month, int day);
};
```

**Destructor**

*CPPReference: Destructor*

`~Date();`

A destructor *(often shortened to `dtor`)* is the ***counterpart*** to the constructor. Must ***release*** all resources. Is implicitly available, must ***not*** throw an exception, because if it does, the whole program gets terminated. Is called ***automatically*** at the end of the block for local instances.

```
class Date {
  public:
    Date(int year, int month, int day);
    ...
    // Destructor
    ~Date();
};
```

## 5.3. IMPLEMENTATION IN THE SOURCE FILE

### 5.3.1. Constructors and Default Initialization

- ***Establish Invariant:*** Properties for a value of the type that are always valid. A `Date` instance always represents a valid date. All *(public)* member functions assume and keep it intact.
- ***Initialize all Members:*** Constructors only create a valid instance. Use initializer lists and the default values if possible / necessary, see “Constructors” (Page 21).

```
#include "Date.hpp"
Date::Date(int year, int month, int day)
    : year{year}, month{month}, day{day} {
  if (!isValidDate()) {
    throw std::out_of_range{"invalid date"};
  }
}
Date::Date() : Date{1980, 1, 1} { }

Date::Date(Date const & other)
  : Date{other.year, other.month, other.day} {}
```

The ***Default Value*** should be created by the default Constructor. Initialize all variables with `{}`!

`Date::Date():year{9999}, month{12}, day{31} {}`

***Member variables*** can have a ***default value*** assigned. These values are used if the member is not present in the initializer list of the constructor. Useful if multiple constructors initialize class similarly. ***`NSDMI` = Non-Static Data Member Initializers.***

```
class <classname> { // general syntax
  <type> <membername>{<default-value>};
};
class Data { // example
  int year{9999}, month{12}, {31};
}
```

### 5.3.2. Implementing Member Functions

- ***Don’t violate invariant***: Leave object in valid state.
- ***Implicit `this` object***: Is a pointer, member access with arrow `->`.
  `this→` can usually be omitted, only necessary when a naming ambiguity exists.
- ***Declare `const` if possible!***
- Must ***not modify members*** and can only call `const` functions if `const`.

Otherwise ***access*** to ***all*** other members.

```
// Date.cpp
#include "Date.hpp"
auto Date::isValidDate() const → bool {
  if ( day ⩽ 0) { return false; }
  switch (month) {
    case 1: case 3: case 5: case 7: case 8:
    case 10: case 12:
      return day ⩽ 31;
    case 4: case 6: case 9: case 11:
      return this→day ⩽ 30;
    case 2:
      return day ⩽ (isLeapYear(year) ? 29:28);
    default:
      return false;
  }
}
```

### 5.3.3. Implementing Static Member Functions

No `this` object, cannot be `const`.
No `static` keyword in the implementation.

Call with `<classname>::<member>()`:
`Date::isLeapYear(2016);`

```
#include "Date.hpp"
auto Date::isLeapYear(int year) → bool {
  if (year % 400 == 0) { return true; }
  if (year % 100 == 0) { return false; }
  return year % 4 == 0;
}
```

### 5.3.4. Implementing Static Member Variables

No `static` keyword in implementation. `static const` member can be initialized directly in the header.

Access outside of the class with name qualifier: `<classname>::<member>`

```
// File Date.hpp
static Date myBirthday;
// File Date.cpp
Date const Date::myBirthday{1996, 21, 10};
// File Any.cpp
#include "Date.hpp"
auto printBirthday() → void {
  std::cout << Date::myBirthday;
}
```

### 5.3.5. Inheritance

*CPPReference: Derived Classes*

Base classes are specified after the name. `class <name> : [visibility] <base1>, ..., <baseN>`.
***Multiple inheritance*** is possible, but should be avoided.
Inheritance can specify a ***visibility***, limits the maximum visibility of the inherited members *(i.e. private inheritance turns all `public` and `protected` members of the base class `private`)*
If ***no visibility*** is specified, the default of the inheriting class is used *(`class` → `private`, `struct` → `public`)*.

If the subclass is not a class but a struct, the keyword `"public"` is not needed.

```
class Base {
  private:
    int onlyInBase;
  protected:
    int baseAndInSubclasses;
  public:
    int everyoneCanFiddleWithMe;
};
class Sub : public Base {
  // can see baseAndInSubclasses and
  // everyoneCanFiddleWithMe
}
```

### 5.3.6. Sequence

The Sequence of initialization is important, if there are multiple base classes. The base class constructor should come ***before*** the initialization of members.

```
class DerivedWithCtor : public Base1, public Base2 {
  int mvar;
public:
  DerivedWithCtor(int i, int j) : Base1{i}, Base2{}, mvar{j} {}
};
```

For more details on inheritance, see "Dynamic Polymorphism" (Page 53).
For Template Syntax of classes, see "Class Templates" (Page 45).

## 5.4. OPERATOR OVERLOADING

*CPPReference: Operator Overloading*

`#include <compare>`

Custom operators can be ***overloaded*** for user-defined types. Declared like a function, with a special name.

```
auto operator op(<parameters>) → <returntype>
  <returntype> operator op(<parameters>)
```

Operators can be implemented to ***simplify*** the handling with classes. For example, you can override the == operator to see if two dates in the `Date` class are equal, or override the relational comparison operators <, >, ⩽, ⩾ to order dates.
***Operators should be implemented reasonably!*** Their semantic should be natural and lead to no surprises for the user.
Unary operators *(like ! or ++)* take one, binary operators *(like < or +=)* take two parameters. The second parameter *(often called **right hand side (`rhs`)**)* does not necessarily need to be the same type as the first *(often called **left hand side (`lhs`)**)*. If the operator is implemented inside of a class, the left-hand side is given ***implicitly through `this`***.

### 5.4.1. Three-Way-Comparison

CPPReference: `std::compare_three_way`

Before C++20, all relational operators <, >, <=, >= and equality operators ==, != had to be implemented ***separately,*** leading to a lot of boilerplate code.

The ***three-way-comparison operator <=>*** *(informally called Spaceship Operator)* can be implemented to provide all relational comparisons at once. It has a ***special return type*** based on whether how strongly comparable the elements are, see "Ordering" (Page 27).
***The equality operator ==*** still needs to be implemented ***manually*** due to differing return types, however it can be implemented by calling the spaceship operator. It also implicitly overrides != for inequality.

```
class Date {
  int year, month, day;
 public:
  auto operator<=>(Date const& right) const -> std::strong_ordering {
    // the left hand side has an implicit 'this->'
    if (year != right.year) { return year <=> right.year; }
    if (month != right.month) { return month <=> right.month; }
    return day <=> right.day;
  }
  auto operator==(Date const& right) const -> bool {
    // implemented by calling <=> and checking if result is equal
    // '*this' to get the value of the current/lhs object
    return (*this <=> right) == std::strong_ordering::equal;
  }
};
```

The compiler can ***generate*** the three-way-comparison operator by ***defaulting*** it. The default compares every member of both objects in definition order with the spaceship operator. This implicitly generates the equality operator as well.

```
class Date {
  int year, month, day;
 public:
  // First compares year, then month, then day with the <=> operator
  auto operator <=> (Date const& right) const = default; // name "right" is optional
  // auto operator <=> (Date const& right) const -> std::strong_ordering = default
}
```

### 5.4.2. Free Operators

Operators are called free operators when they are implemented ***outside*** of a class. While ***inside*** of a class, the first parameter was given implicitly by the `this` pointer, free operators need to specify it explicitly.
***There are some limitations:*** Assignment can only be implemented as a member operator, while the << and >> operators dealing with streams can only be implemented as free operators.

```
class Date {
  int year, month, day;
 public: auto operand<(Date const& right) const → bool;
};

inline auto operator >(Date const& left, Date const& right) → bool { return right > left; }
// ...
```

### 5.4.3. Examples: Stream and input/output operators

To input or output data from/to a class, the << and >> operators are often ***overloaded.*** They must be implemented as ***free operators*** and require a reference to their respecting stream type as their ***first parameter*** *(`std::istream&` / `std::ostream&`)* and a object reference to read/write from/to as their ***second parameter***. Their return type is the same stream again, so multiple consecutive writes/reads are possible *(Chaining)*.

The operators also use the ***`inline` keyword***. This is because the definition inside of the header can appear in multiple translation units and the linker may see it multiple times. Normally, this would cause a compile error, but with `inline` we ask the linker not to worry about it and "just pick one".

**Print class members**

***"<<"*** Must be a ***free function***. To keep the class ***encapsulation intact***, the printing is delegated to a member function, so the operator does not need to access private class members directly *(often done via `friend` operator – bad design!)*. The second parameter with the object reference and the whole operator itself can be `const`, as nothing is modified.

```cpp
// Date.hpp
#include <ostream>
class Date {
  int year, month, day;
 public:
  auto print(std::ostream& os) const -> void {
    os << year << "/" << month << "/" << day;
  }
};
inline auto operator <<(std::ostream& os, Date const& date)
  -> std::ostream& {
  date.print(os); return os;
}
```

```cpp
// Any.cpp
#include "Date.hpp"
#include <iostream>

auto printBirthday() -> void {
  std::cout << Date::myBirthday;
}
```

**Create new instance by reading from input**

***">>"*** must be a ***free function***. When reading input, it is always a good idea to ***validate*** that input. Unlike the << operator, the object reference parameter and the whole operator itself ***cannot be `const`***, as `date` is modified by assigning it a new `Date` instance.

```cpp
// Date.hpp
#include <ostream>
class Date {
  int year, month, day;
 public:
  auto print(std::ostream& os) const -> void {
    os << year << "/" << month << "/" << day;
  }
};
inline auto operator >>(std::istream& is, Date& date)
  -> std::istream& {
  int year{-1}, month{-1}, day{-1};
  // discard vars to get rid of the date seperators
  char sep1, sep2;
  is >> year >> sep1 >> month >> sep2 >> day;
  try {
    date = Date{year, month, day};
    is.clear();
  } catch (std::out_of_range const& e) {
    // validation inside the 'Date' ctor failed
    is.setstate(std::ios::failbit);
  }
  return is;
}
```

```cpp
// Any.cpp
#include "Date.hpp"
#include <iostream>

auto readDate() -> Date {
  Date date{};
  std::cin >> date;
  return date;
}
```

### 5.4.4. Ordering

CPPReference: std::strong_ordering, CPPReference: std::weak_ordering, CPPReference: std::partial_ordering

The three-way-comparison returns a ***ordering type*** instead of a bool. There are different types of orders to choose from depending on the elements to compare.

**Strong Order**

```
auto operator <=> (Date const& right) const -> std::strong_ordering;
```

Values that are equivalent are ***indistinguishable***.
Either "a < b", "a == b" or "a > b" must be true.
*(For example, ints or Dates.)*

- std::strong_ordering::less for a < b
- std::strong_ordering::equivalent or std::strong_ordering::equal for a == b
- std::strong_ordering::greater for a > b

**Weak Order**

```
auto operator <=> (Date const& right) const  -> std::weak_ordering;
```

Values that are equivalent ***may be distinguishable***.
Either "a < b", "a == b" or "a > b" must be true.
*(For example strings, when letter case is ignored, i.e. Hello and hello are equivalent, but not equal)*

- std::weak_ordering::less for a < b
- std::weak_ordering::equivalent for a == b
- std::weak_ordering::greater for a > b

**Partial Order**

```
auto operator <=> (Date const& right) const -> std::partial_ordering;
```

Values that are equivalent ***may be distinguishable***.
"a < b", "a == b" and "a > b" can all be false.
*(For example double, as NaN always compares false with itself)*

- std::partial_ordering::less for a < b
- std::partial_ordering::equivalent for a == b
- std::partial_ordering::greater for a > b
- std::partial_ordering::unordered for none of the above

# 6. NAMESPACES AND ENUMS

## 6.1. NAMESPACES

CPPReference: Namespaces

Namespaces are ***scopes*** for grouping and preventing name clashes. The ***same name*** for classes, functions etc. in different scopes is possible *(boost::optional and std::optional can coexist)*. ***Nesting*** of namespaces is possible *(std::literals::chrono_literals)*, allows hiding of names.
The ***global namespace*** hast the :: prefix. Can be ***omitted*** if unique *(::std::cout is usually equal to std::cout)*.

### 6.1.1. Example

Namespaces can only be defined ***outside*** of classes and functions. The same namespace can be opened and closed multiple times *(i.e. to split a namespace over multiple files)*. Qualified names are used to access names in a namespace: demo::subdemo::foo().
A name with a leading :: is called a ***fully qualified name***.

```
namespace demo {
auto foo() → void; // 1
namespace subdemo {
auto foo() → void { /* 2 */ }
} // subdemo
} // demo

namespace demo {
auto bar() → void {
  foo(); // 1
  subdemo::foo(); //2
} } // demo
auto demo::foo() → void { /* 1 */ }

auto main() → int {
  using demo::subdemo:foo;
  foo(); // 2
  demo::foo(); // 1
  demo::bar();
}
```

**Using Declarations**

```
using std::string; string s{"no std::"};
```

***Imports*** a name from a namespace into the ***current scope***. That name can then be used without a namespace prefix. Useful, if the name is used very often.
It is also possible to give the namespace an ***alias***:
`using input = std::istream_iterator<int>;`
Don't put ***`using namespace std`*** into your header file to avoid ***"namespace pollution"***.

### 6.1.2. Anonymous Namespaces

The name after the `namespace` keyword can be ***omitted*** to turn it into an ***anonymous namespace***. Every implementation can only be accessed from ***inside this file***. This ***hides module internals*** like helper functions and constants. While the namespace doesn't have a "public" name, the compiler gives it an ***unique identifier*** internally. Anonymous namespaces should only be used in source files *(.cpp)*

```
namespace { // anonymous namespace
// can't be called outside this file
auto doit() → void { ... }
} // anonymous namespace ends

auto print() → {
  // callable if declared in header
  doit();
}
```

### 6.1.3. Argument Dependent Lookup

*CPPReference: Argument-dependent lookup*

Types and *(non-member)* functions belonging to that type should be placed in a ***common namespace***. When the compilers encounters an ***unqualified function*** it looks into the namespace in which that function is ***defined*** to ***resolve*** it *(i.e. it is not necessary to write `std::` in front of `for_each` when the first argument is `std::vector::begin()`).*

Functions and operators are ***looked up*** in the namespace of the type of their arguments first, so unqualified operator calls don't allow explicit namespace qualification: ~~`std::cout calendar::<< birthday`~~

**Example**

```
// Adl.hpp
namespace one {
  struct type_one{};
  auto f(type_one) → bool { /* ... */ } //(1)
}
namespace two {
  struct type_two{};
  auto f(type_two) → void { /* ... */ } //(2)
  auto g(one::type_one) → void { /* ... */ } //(3)
  auto h(one::type_one) → void { /* ... */ }
}
auto g(two::type_two) → void { /* ... */ } //(4)
```

```
// Adl.cpp
#include "Adl.hpp"
int main() {
  one::type_one t1{};
  f(t1); //(1)
  two::type_two t2{};
  f(t2); //(2)
  // error: t1 → one, no checks for 'two'
  h(t1);
  two::g(t1); //(3)
  g(t1); //Argument type does not match(4)
  g(t2) //(4)
}
```

### 6.1.4. Putting `Date` in a namespace

The `Date` class should be put in a namespace to group it with its operators and functions. Using types and and functions from Date now require qualification.

**Date.hpp**

```
namespace calendar {
class Date {
  int year, month, day;
public:
  auto tomorow() const → Date
}; }
```

**Date.cpp**

```
#include "Date.hpp"
auto calendar::Date::tomorrow() const → Date {
  // ...
}
```

### 6.1.5. Issues

Templates might ***not pick up*** a global operator << in an algorithm call using `ostream_iterator` if the value output is from ***namespace std*** too *(i.e `std::vector<int>`)*. This would require to put both the `ostream` and `std::vector<int>` in a `namespace std`-block. But this is ***not allowed*** by the C++ standard. To work around this, a ***new class*** inheriting from `std::vector<int>` has to be created with inherited constructors. A simple alias is insufficient. But it is generally ***not recommended*** to derive from standard containers in general.

## 6.2. ENUMS

*CPPReference: Enumeration declaration*

Enumerations are useful to ***represent types with only a few values***. An enumeration creates a new type that can easily be ***converted*** to an ***integral*** type. The ***individual values*** *(enumerators)* are ***specified*** in the type. Unless specified explicitly, the values start with 0 and increase by 1.

**Unscoped enum**

Has no `class` keyword. Used without qualifier.
Can ***implicitly converted*** to `int`.
Enumeration leaks into surrounding scope, best used as a member of a class.

**Scoped enum**

Has a `class` keyword. Requires qualifier.
Requires a `static_cast<int>`.
Enumeration does not leak into surrounding scope.

```
// unscoped enum
enum <name>  { <enumerators> };
enum DayOfWeek {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
// 0,   1,   2,   3,   4,   5,   6
};
// implicit conversion to int
int day = Sun; // 6

// scoped enum
enum class <name> { <enumerators> };
enum class DayOfWeek{
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
// 0,   1,   2,   3,   4,   5,   6
};
// no implicit conversion to int
int day = static_cast<int>(DayOfWeek::Sun);

// from int to enum always requires manual cast
// no type safety, invalid values possible
DayOfWeek tuesday = static_cast<DayOfWeek>(1);
```

### 6.2.1. Operator Overloads for Enumerations

Operators can be ***overloaded*** for `enums`. Prime candidates for overloading are the ***prefix increment*** `++i` and ***postfix increment*** `i++` operators. If both should be implemented, the postfix requires a pseudo-argument, so the compiler can distinguish the signatures.

```
// Prefix increment
auto operator++(DayOfWeek& d) -> DayOfWeek {
  int day = (d + 1) % (Sun + 1);
  d = static_cast<DayOfWeek>(day);
  return d;
}
```

```
// Postfix increment
auto operator++(DayOfWeek& d, int) -> DayOfWeek
{
  DayOfWeek ret{d};
  if (d == Sun) { d = Mon; }
  else {
    d = static_cast<DayOfWeek>(d + 1); } }
```

Another popular application is the `<<` operator: Since enumerator names are not mapped automatically to their original name, a lookup table is often provided by the output operator to ***get an Enumeration as string***.

```
auto operator<<(std::ostream& out, Month m) -> std::ostream& {
  static std::array<std::string, 12> const monthNames {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
    out << monthNames[m - 1]; return out;
}
```

### 6.2.2. Defining Values of Enumerators

With `=`, ***values can be specified*** for enumerators. Subsequent enumerators get value incremented.

***Different enumerators*** can have the ***same value***.

In the example on the right, `may` doesn't have a "long name" version and is missing in the second half. This is why `june` requires a new assignment.

```
enum Month {
  jan = 1, feb, mar, apr, may, jun, jul, aug,
  sep, oct, nov, dec,
  january = jan /*1 */, february /*2*/,
  march, april, june = jun /*5*/, july,
  august, september, october, november,
  december
}
```

### 6.2.3. Specifying the Underlying Type

Enumerations can ***specify*** the ***underlying type*** by inheritance. The underlying type can be ***any integral type***. This allows ***forward-declaring*** enumerations, which can be used to hide implementation details if defined as a class member.

```cpp
enum class LaunchPolicy : unsigned char {
  sync = 1;      // Enum values specified in
  async = 2;     // powers of 2 are often used
  gpu = 4;       // as bitmasks: 1 = 0b001
  process = 8;   // 2 = 0b010, 4 = 0b100
  none = 0;
}
```

### 6.2.4. Example

```cpp
// Statemachine.hpp
#ifndef STATEMACHINE_HPP_
#define STATEMACHINE_HPP_

struct Statemachine {
  Statemachine();
  auto processInput(char c) -> void;
  auto isDone() const -> bool;
private:
  enum class State : unsigned short;
  State theState;
};
#endif
```

```cpp
// Statemachine.cpp
#include "Statemachine.hpp"
#include <cctype>
enum class Statemachine::State : unsigned short {
  begin, middle, end
};
Statemachine::Statemachine() : theState {State::begin} {}
auto Statemachine::processInput(char c) -> void {
  switch (theState) {
    case State::begin:
      if(!isspace(c)) {theState = State::middle;} break;
    case State::middle:
      if(!isspace(c)) {theState = State::end;} break;
    case State::end:
      break; // Ignore input
  }
}
auto StateMachine::isDone() const -> bool {
  return theState == State::end;
}
```

## 6.3. ARITHMETIC TYPES

*CPPReference: Fundamental types*

The arithmetic types are divided into two categories: ***integral types*** *(which include character and boolean types)* and ***floating-point types***. It is not recommended to implement your own arithmetic type, but here is a basic example anyway.

### 6.3.1. Ring5: Arithmetic Modulo 5

The basics are provided: An invariant *(The member variable is in range $[0, 4]$)*, an accessor to the value and a explicit constructor. We also implement the default equality operator, a custom output operator and custom + and += operators

```cpp
struct Ring5 {
  explicit Ring5(unsigned x = 0u) : val{x & 5} {} // constructor
  auto value() const -> unsigned { return val; } // acessor
  auto operator==(Ring5 const& r) const -> bool = default;
  auto operator+=(Ring5 const& r) -> Ring5& {
    val = (val + r.val) % 5; return *this; // where the magic happens
  }
  auto operator+(Ring5 const& r) const -> Ring5 { // uses += operator for result
    Ring5 lvalue = *this; lvalue += r; return lvalue;
  }
private:
  unsigned val;
}
auto operator<<(std::ostream& out, Ring5 const& r) -> std::ostream {
  out << "Ring5{" << r.value() << '}'; return out;
}
```

### 6.3.2. Adding mixed arithmetic

If we want to add `Ring5` and `int`, we have two possibilities:

- ***Implement all parameter combinations for the + operator:*** Causes code duplication overhead
  `operator+(Ring5, unsigned); operator+(unsigned, Ring5)`
- ***Make constructor non-explicit:*** Might cause problems with automatic conversion.

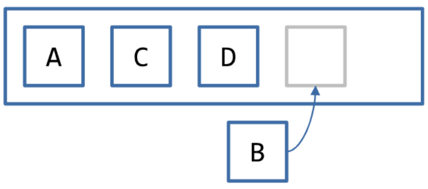
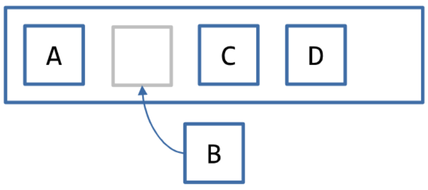
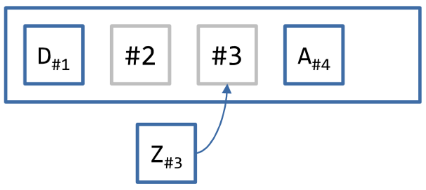
Both options have their own downsides. Pick your poison!

# 7. STANDARD CONTAINERS AND ITERATORS

## 7.1. STANDARD CONTAINERS

*CPPReference: Containers library*

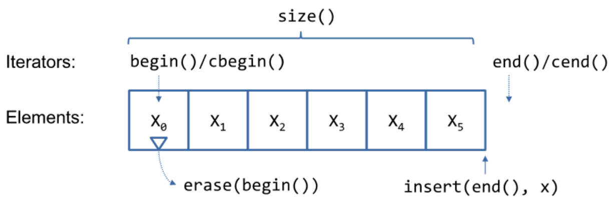
The standard library provides ***different categories*** of containers:

| ***Sequence Containers*** | ***Associative Containers*** | ***Hashed Containers*** |
|---|---|---|
| Elements are accessible ***in order*** as they were inserted / created.<br><br>Find in ***linear time*** $O(n)$ through the algorithm `find`.<br><br>**Examples:**<br>– `std::vector`<br>– `std::array`<br>– `std::list`<br><br> | Elements are accessible in ***sorted*** order.<br><br>`find` as a member function in ***logarithmic time*** $O(\log(n))$ .<br><br>**Examples:**<br>– `std::set`<br>– `std::map`<br>– `std::multimap`<br><br> | Elements are accessible in ***unspecified*** order.<br><br>`find` as member function in ***constant time*** $O(1)$.<br><br>**Examples:**<br>– `std::unordered_set`<br>– `std::unordered_map`<br>– `std::unordered_multimap`<br><br> |

### 7.1.1. Common Features of Containers

All containers have a similar basic interface.

| ***Member Function*** | ***Purpose*** |
|---|---|
| `begin(), end()` | Get iterators for algorithms and iteration in general |
| `erase(iter)` | Removes the element at position the iterator `iter` points to |
| `insert(iter, value)` | Inserts `value` at the position the iterator `iter` points to |
| `size(), empty()` | Check the size of the container |



Containers can be ***default-constructed***, ***copy-constructed*** from another container of the same type, ***equality compared*** if their elements can be compared and ***emptied with `clear()`***.

```cpp
std::vector<int> v{};
std::vector<int> vv{v}; //copy of v
if (v == vv) { v.clear(); }
```

### 7.1.2. Common Container Constructors

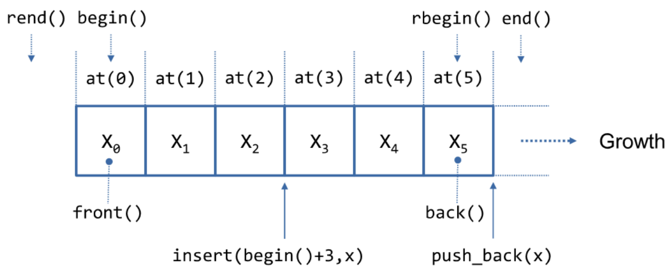
- ***Construction with initializer List:*** `std::vector<int> v{1,2,3,5,7,11};`
- ***Construction with a number of elements:*** `std::list<int> l(5, 42); // 5 list elements with value "42"`
  Can provide a default value. Often needs parenthesis instead of {} to avoid ambiguity from list of values initialization.
- ***Construction from a range given by a pair of iterators:*** `std::deque<int> q{cbegin(v), cend(v)};`
  Might need parenthesis instead of {} *(rare)*

### 7.1.3. Sequence Containers

*CPPReference: Containers library - Sequence Containers*

`std::vector<T>`, `std::deque<T>`, `std::list<T>`, `std::array<N,T>`
***Order*** is defined in order of inserted/appended elements. Lists are good for ***splicing*** and ***"in the middle" insertions***. `std::vector`/`std::deque` are ***efficient*** unless bad usage.

Can all ***grow***, except `std::array` because it is a fixed-size container.



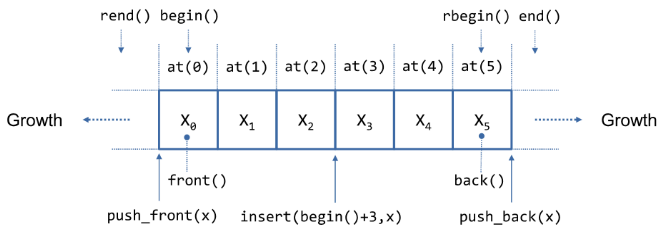
#### Double-Ended Queue: `std::deque<T>`

*CPPReference: `std::deque`*

`#include <deque>`

Like a vector but with additional, efficient front insertion/removal *(`push_front(x)`, `pop_front()`, `push_back(x)`, `pop_back()`)*.

```
std::deque<int> q{begin(v), end(v)};
q.push_front(42);
q.pop_back();
```



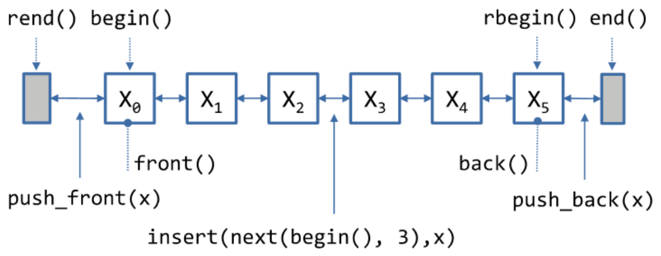
#### Double-Linked List: `std::list<T>`

*CPPReference: `std::list`*

`#include <list>`

Efficient ***insertion*** in ***any*** position *(`push_front(x)`, `insert(next(begin(), 3, x))`, `push_back(x)`)*.
Lower efficiency in ***bulk*** operations, requires member-function call for `sort()` etc. Only ***bi-directional*** iterators - no index access!

```
std::list<int> l{5, 1};
```



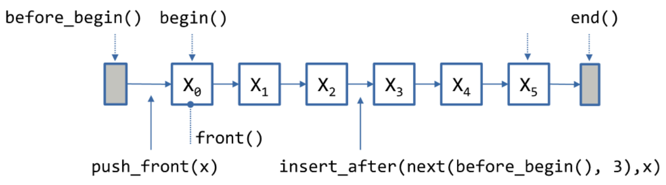
#### Singly-Linked List: `std::forward_list<T>`

*CPPReference: `std::forward_list`*

`#include <forward_list>`

Efficient insertion ***after*** any position, but clumsy with iterator to get "before" position.
Only ***forward-iterators***, clumsy to search and remove, use member-functions instead of algorithms. ***Avoid!***
Better use `std::list` or even better `std::vector`

```
std::forward_list<int> l{1, 2, 3, 4, 5, 6};
```

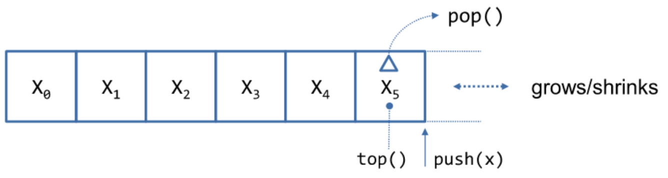


**LIFO Adapter: `std::stack`**

*CPPReference: `std::stack`*

#include <stack>

Uses `std::deque` and limits its functionality to stack operations. Delegates to `push_back()`, `back()` and `pop_back()`. Iteration ***not possible***. No longer a container!

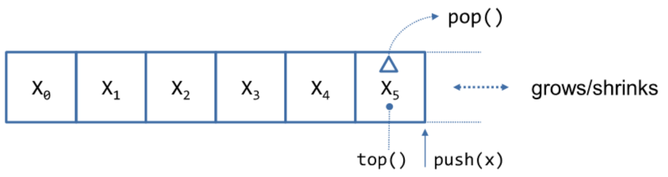


```
std::stack<int> s{};
s.push(42);
std::cout << s.top(); // Get value on stack top
s.pop(); // Removes value without returning it
```

**Adapter: `std::priority_queue`**

*CPPReference: `std::priority_queue`*

Like `std::stack`, but keeps elements partially sorted as (binary) heap. Requires element type to be ***comparable.*** `top()` element is always the smallest. No longer a container!

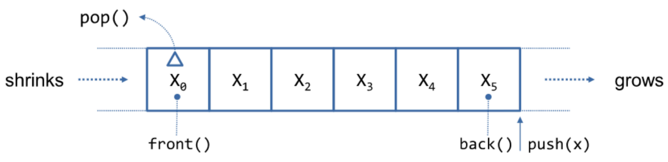


**FIFO Adapter: `std::queue`**

*CPPReference: `std::queue`*

#include <queue>

Uses `std::deque` and limits its functionality to queue operations. Delegates to `push_back()` and `pop_front()`. Iteration ***not possible***. No longer a container!



```
std::queue<int> q{};
q.push(42);
std::cout << q.front(); // get value
q.pop(); // remove value without returning it
```

### 7.1.4. Example: Stack and Queue

```
#include <iostream>
#include <deque>
#include <stack>
#include <string>

auto main() → int {
  std::stack<std::string> lifo{}; // Last in, first out
  std::queue<std::string> fifo{}; // First in, first out
  for (std::string s : { "Fall", "leaves", "after", "leaves", "fall" }) {
    lifo.push(s); fifo.push(s);
  }
  while (!lifo.empty()) {
    std::cout << lifo.top() << ' '; // fall leaves after leaves Fall
    lifo.pop();
  }
  std::cout << '\n';
  while (!fifo.empty()) {
    std::cout << fifo.front() << ' '; // Fall leaves after leaves fall
    fifo.pop();
  }
}
```

### 7.1.5. Associative Containers

*CPPReference: Containers library - Associative Containers*

Are ***sorted tree containers***. Allow searching by content, not by sequence *(Search by key, access key or key-value pair)*.

| | ***Only Key storable*** | ***Key-Value Pair storable*** |
|---|---|---|
| ***Unique Key*** | `std::set<T>` | `std::map<K,V>` |
| ***Multiple Equivalent Keys*** | `std::multiset<T>` | `std::multimap<K,V>` |

Associative containers allow an ***additional template argument*** for the comparison operation. It must be a functor class returning a binary predicate that is ***irreflexive*** and ***transitive.*** Own functors can provide special sort order *(i.e caseless-string comparison)*. The sorting requirement must be fulfilled *(i.e. ⩾ is not allowed, because it is reflexive!)*

```
std::set<int, std::greater> reversed_int_set{}
```

For more details about functors and predicates, see “Functor” (Page 38).

**Set of Elements: `std::set`**

*CPPReference: `std::set`*

`#include <set>`

Stores elements in ***sorted order*** *(ascending by default, can be overwritten by the 2nd parameter)*. Iteration walks over elements in order. Keys cannot be modified through iterators because this would destroy the sorting and invalidate the current iterator.

Use member-functions for `.find` and `.count`.

Initializer does not need to be sorted. `s.contains(x)` is a quick check, if `x` is present in `std::set`.

```
std::set<int> values{7,1,4,3,2,5,6}
```

```
#include <iostrem>
#include <set>
auto filterVowels(std::istream& in,
 std::ostream& out) → void {
  std::set const v{'a','e','o','u','i','y'};
  char c{};
  while (in >> c) {
    if (!v.contains(c)) { out << c; }
  }
}
auto main() → int {
  filterVowels(std::cin, std::cout);
}
```

**Map of Key-Value-Pairs: `std::map`**

*CPPReference: `std::map`*

`#include <map>`

Stores key-value pairs in ***sorted order***. Sorted by key in ascending order. Order can be overwritten by the 3rd template parameter.

The ***indexing operator [ ]*** inserts a new entry automatically if the given key is not present. Returns the value ***by reference***.

When using an ***iterator,*** the item returned is a `std::pair<key, value>`. Use `.first` to access the key and `.second` for the value.

```
std::map<char, size_t> vowels{
  {'a',3},{'e',8},{'i',5},{'o',4},{'u',2} }
```

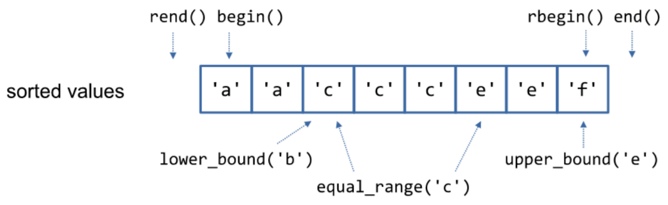
```
auto countVowels(std::istream& in,
std::ostream& out) → void {
  std::map<char, size-t> v{
    {'a',0},{'e',0},{'i',0},{'o',0},{'u',0}};
  char c{};
  while (in >> c) {
    if (v.contains(c)) {
      ++v[c];
      std::for_each(cbegin(v), cend(v), [&out]
       (auto const& e) {
        out << e.first <<" = "<<e.second<<'\n';
      });
     }
   }
 }
```

#### `std::multiset` and `std::multimap`

*CPPReference: `std::multiset`, CPPReference: `std::multimap`*

Multiple equivalent keys allowed. Use `equal_range()` or `lower_bound()`/`upper_bound()` member function/ algorithms to find boundaries of equivalent keys.

```
std::multiset<char>
letters{'a','a','c','c','c','e','e','f'};
letters.lower_bound('b'); // iter to elem 2
letters.upper_bound('e'); // iter to elem 7
// pair with iter to elem 2 & iter to elem 5
letters.equal_range('c');
```



*Example prints all the same words on the same line, new words on new line.*

```
#include <set> #include <map>
```

```
auto sortedStringList(std::istream& in,
std::ostream&out) -> void {
  using inIter =
     std::istream_iterator<std::string>;
  using outIter =
     std::ostream_iterator<std::string>;
  std::multiset<std::string> w{inIter{in},
     inIter{} };
  copy(cbegin(w), cent(w), outIter(out, "\n"));
  auto current = cbegin(w);
  while (current != cend(w)) {
    auto endOfRange = w.upper_bound(*curent);
    copy(current,endOfRange,outIter{out,", "});
    out << '\n';
    current = endOfRange;
  }}
```

### 7.1.6. Hashed Containers

*CPPReference: Containers library - Unordered Associative Containers*

#### `std::unordered_set`

*CPPReference: `std::unordered_set`*

More ***efficient lookup***, no sorting. Usage is almost equivalent to `std:set`, except for lack of ordering.

Don't use `std::unordered_set` with your own types.

```
#include <unordered_set>
```

```
auto main() -> int {
  std::unordered_set<char> const v
   {'a','e','o','u','i','y'}
  using in = std::istreambuf_iterator<char>;
  using out = std::ostreambuf_iterator<char>;
  remove_copy_if(in{std::cin}, in{},
   out{std::cout},[&](char c) {
       return v.contains(c);
   }); }
```

#### `std::unordered_map`

*CPPReference: `std::unordered_map`*

Usage is almost equivalent to `std:map`, except for lack of ordering.

Don't use `std::unordered_map` with your own types.

```
#include <unordered_map>
```

```
auto main() -> int {
  std::unordered_map<std::string, int> w{};
  std::string s{};
  while (std::cin >> s) { ++w[s]; }
  for (auto const& p : w) {
    std::cout<<p.first<<" = "<<p.second<<'\n';
  }
```

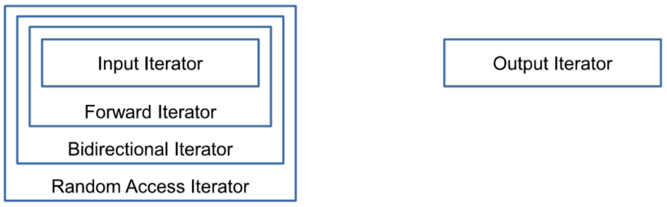
## 7.2. ITERATORS

*CPPReference: Iterator Library*

### 7.2.1. STL Iterator Categories

Different containers support iterators of different capabilities. Categories are formed around ***"increasing power"***

– ***std::input_iterator*** corresponds to `istream_iterator`'s capabilities
– ***std::ostream_iterator*** is an `output_iterator`
– ***std::vector<T>*** provides `random_access` iterators



Some algorithms ***only work*** with ***powerful iterators*** *(For example, `std::sort()` requires a pair of random access iterators to jump backwards and forwards)*. Some algorithms can be ***implemented better*** with more powerful iterators *(For example, `std::advance()` or `std::distance()` are faster with a random access iterator than a forward iterator)*.

**`const_iterator` ≠ `const` Iterator!**

Declaring an iterator ***`const`*** would not allow modifying the iterator object, meaning the iterator cannot be incremented with `++` or `std::next()`.
`cbegin()` and `cend()` return ***`const_iterators`***. This does ***not*** imply the iterator to be `const`, but the ***elements*** the iterator walks over are `const` and therefore can't be modified.

### 7.2.2. Input Iterator

*CPPReference: `std::input_iterator`*

Supports ***reading*** the "current" element. Allows for ***one-pass input algorithms***.

Can ***not*** step ***backwards.*** Models the `std::istream_iterator` and `std::istream`.

Can be ***compared*** with `==` and `!=` to other iterator objects of the same type: `It`. Can be ***copied.*** After incrementing *(Calling ++)*, all other copies are ***invalid***.

```
struct input_iterator_tag{};
auto operator* () -> Element;
auto operator++() -> It&;
auto operator++(int) -> It;
auto operator==(It const&) -> bool;
auto operator!=(It const&) -> bool;
auto operator= (It const&) -> It&;
It(It const&); //copy constructor
```

### 7.2.3. Forward Iterator

*CPPReference: `std::forward_iterator`*

Can do whatever an input iterator can, plus …
Supports ***changing*** the "current" element, unless the container or its element are `const`.

Can ***not*** step backwards, but can keep iterator copy around for later reference.
Models the `std::forward_list` iterators.

```
struct forward_iterator_tag{};
auto operator*() -> Element&;
// Otherwise has the same operators as the
// input iterator
```

### 7.2.4. Bidirectional Iterator

*CPPReference: `std::bidirectional_iterator`*

Can do whatever a forward iterator can, plus …
Can go ***backwards,*** allows for ***forward-backward-pass*** algorithms.

Models the `std::set` iterators.

```
struct bidirectional_iterator_tag{};
auto operator--() -> It&;
auto operator--(int) -> It;
// Otherwise has the same operators as the
// forward iterator
```

### 7.2.5. Random Access Iterator

CPPReference: std::random_access_iterator

Can do whatever a bidirectional iterator can, plus ... ***Directly access*** element at index *(Offset to current position)*: Distance can be positive or negative.

***Go n steps*** forward or backward. ***Subtract*** two iterators to get the distance. ***Compare*** with ***relational operators*** *(<, ≤, >, ≥)*

Models the `std::vector` iterators.

```
struct random_access_iterator_tag{};
auto operator[](distance) → Element&;
auto operator+(distance) → It;
auto operator+=(distance) → It&;
auto operator-(distance) → It;
auto operator-=(distance) → It&;
auto operator-(It const &) → distance;
// relational operators, like <
// Otherwise has the same operators as the
// bidirectional iterator
```

### 7.2.6. Output Iterator

CPPReference: std::output_iterator

Can ***write*** value to current element, but only once *(`*it = value`)*. After that, an increment is required.

Most ***other iterators*** can also ***act as output iterators***, unless the underlying container is ***`const`***. **Exception:** associative containers only allow read-only iteration.

***No comparison*** and end to an out range is not queryable.
Models the `std::ostream_iterator`.

```
struct output_iterator_tag{};

auto operator*() → Element&;
auto operator++() → It&;
auto operator++(int) → It;
```

### 7.2.7. Iterator Functions

CPPReference: Iterator library - Iterator Operations

- ***`std::distance(start, goal)`:*** counts the number of "hops" iterator `start` must make until it reaches `goal`. Efficient for random access iterators, for other iterators it needs to traverse the iterator.
- ***`std::advance(itr, n)`:*** lets `itr` "hop" `n` times. Requires a step, no default step size. Modifies the argument iterator. Returns void. Efficient for random access iterators. Allows negative `n` for bidirectional iterators.
- ***`std::next(itr, n)`:*** lets `itr` "hop" `n` times. Has a default step size of 1. Makes a copy of the argument *(returns a input iterator pointing to the n-th element)*.

# 8. STL ALGORITHMS

CPPReference: Algorithm library

It is almost always better to use an algorithm instead of a loop.
- ***Correctness:*** It is much easier to use an algorithm correctly than implementing loops correctly.
- ***Readability:*** Applying the correct algorithm expresses your intention much better than a loop.
- ***Performance:*** Algorithms might perform better than handwritten loops.

## 8.1. BASICS

Algorithms work with ***ranges*** specified by iterators. They usually take 1 or 2 ranges as the input (start, end) and 1 as the output (start only, end is not required). However, there are some things that need to be kept in mind when working with iterators, see "Pitfalls" (Page 41).

### 8.1.1. Iterator for Ranges

- ***First:*** Iterator pointing to the first element
- ***Last:*** Iterator pointing beyond the last element
- ***If `First == Last`:*** the Range is Empty

```
std::vector<int> values{54, 23, 17, 95, 85};
std::xxx(begin(values), end(values), ...);
```

### 8.1.2. Iterators as Output of Ranges

Streams need a wrapper to be used with algorithms

- std::ostream → std::ostream_iterator<T>
- std::istream → std::istream_iterator<T>
- Default-constructed std::istream_iterator<T> marks EOF

```
auto redirect(std::istream& in,
    std::ostream& out) → void {
  using in_iter = std::istream_iterator<char>;
  using out_iter = std::ostream_iterator<char>;
  std::copy(in_iter{in}, in_iter{},
    out_iter{out});
}
```

### 8.1.3. Reading Algorithm Signatures

Each algorithm has a name, parameters and a return type. The description specifies the requirements. Algorithms work with the iterator categories *(See "STL Iterator Categories" (Page 36))*

```
/*             Template-Header            */
template<class InputIt, class UnaryFunction>
UnaryFunction    for_each    (InputIt first, InputIt last, UnaryFunction f);
/* Returntype */ /* Name */  /*               Parameters               */
```

## 8.2. FUNCTOR

*CPPReference: Standard Library Header <functional>*

`#include <functional>`

A functor is a type that ***provides a call operator***. An object / instance of that type can be called like a function. They can hold a ***state*** between calls, like closures in functional languages. ***Lambdas*** are realized with functors internally.

```
auto average(std::vector<int> values) → int  {
  auto acc = Accumulator{};
  // for_each returns acc again
  return std::for_each(begin(values),
   end(values), acc).average();
}
```

```
struct Accumulator {
  int count{0}; int accumulatedValue{0};7
  auto operator()(int value) → void {//Functor
    count++; accumulatedValue += value;
  }
  int average() const; int sum() const;
}
auto average(std::vector<int> values) → int {
  Accumulator acc{};
  for(auto v : values) { acc(v); }
  return acc.average();
}
```

### 8.2.1. Predicate

A function or a lambda returning `bool`. For checking a criterion / condition.

| ***Unary Predicate*** | ***Binary Predicate*** |
|---|---|
| Has ***one*** Parameter.<br>`auto is_odd = [](auto i) → bool { return i % 2 };` | Has ***two*** Parameters.<br>`auto divides = [](auto i, auto j) → bool { return !(i % j); };` |

### 8.2.2. Standard Functor Template Classes

Lamdas make applying `transform` etc quite easy:
`transform(v.begin(), v.end(), v.begin(), [](auto x){return -x;});`

However, the STL provides standard Functor Classes, which make it even easier.
`transform(v.begin(), v.end(), v.begin(), std::negate<>{});`

**Binary arithmetic and logical**

```
– plus<>        // (+)
– minus<>       // (-)
– divides<>     // (/)
– multiplies<>  // (*)
– modulus<>     // (%)
– logical_and<> // (&&)
– logical_or<>  // (||)
```

**Unary**

```
– negate<>      // (-)
– logical_not<> // (!)
```

**Binary Comparison**

```
– less<>          // (<)
– less_equal<>    // (⩽)
– equal_to<>      // (==)
– greater_equal<> // (⩾)
– greater<>       // (>)
– not_equal_too<> // (≠)
```

### 8.2.3. Example: `set<string>` for dictionary

```cpp
#include <set>
#include <algorithm>
#include <cctype>
#include <iterator>
#include <iostream>
struct caseless {
  using string = std::string;
  // binary predicate with strings as Ranges, lambda as binary predicate on char
  auto operator()(string const & l, string const & r) const → bool {
    // run a lexographical compare on the two strings
    return std::lexicographical_compare(l.begin(), l.end(), r.begin(), r.end(),
      // make each char lowercase before comparing them
      [](char l, char r){ return std::tolower(l) < std::tolower(r); });
  }
};
auto main() → int {
  using std::string;
  // pass predicate functor as template argument, the strings are now sorted
  using caseless_set = std::multiset<string, caseless>;
  using in = std::istream_iterator<string>;
  auto const word_list = caseless_set{in{std::cin}, in{}};
  auto out = std::ostream_iterator<string>{std::cout, "\n"};
  copy(word_list.begin(), word_list.end(), out);
}
```

## 8.3. ALGORITHM EXAMPLES

**Syntax:**

- ***first1:*** Iterator to the start of the first range *(usually `.begin`/`std::begin()`)*
- ***last1:*** Iterator to the end of the first range *(usually `.end()`/`std::end()`)*
- ***first2:*** Iterator to the start of the second range *(usually `.begin`/`std::begin()`)*
- ***out_first:*** Iterator to the start of the output range *(usually `.begin`/`std::begin()`)*
- ***c:*** The container itself
- ***unary_op/binary_op:*** Unary/binary function, lambda or functor to apply to the range
- ***comp:*** Custom comparison function. Usually optional.
- ***init:*** A initial value

Almost all algorithms have a `std::ranges` variant, where `first1` and `last1` can be replaced with `c`. This does not work on `std::istream`, as it needs a dummy stream that acts as the end, see "`std::ranges`" (Page 12) and "Iterators for I/O" (Page 14)

### 8.3.1. `transform`

*CPPReference: `std::transform`, CPPReference: `std::ranges::transform`*

```cpp
std::transform(first1, last1, [first2], out_first, [unary_op|binary_op]);
```

Mapping one *(or two of same size)* range to new values and store the result in a new range. Lambda, Function or Functor for map operation, input and output types can be different.
*(Example: Put chars from `letter` count times into `combined`)*

```cpp
auto counts = std::vector{3,0,1,4,0,2};
auto letters = std::vector{'g','a','u','y','f','o'};
auto combined = std::vector<std::string>{};
auto times = [](auto i, auto c) { return std::string(i, c); };
std::transform(begin(counts), end(counts), begin(letters),
  std::back_inserter(combined), times);
```

### 8.3.2. `merge`

*CPPReference: std::merge*, *CPPReference: std::ranges::merge*

```
std::merge(first1, last1, first2, last2, out_first, [comp]);
```

Merge two ***sorted*** ranges into a output range. Undefined behavior if ranges are unsorted.

```
std::vector r1{ /* ... */}; std::vector r2{ /* ... */ }
// initialize empty vector with correct size
std::vector d(r1.size() + r2.size(), 0);
std::merge(begin(r1), end(r1), begin(r2), end(r2), begin(d));
```

### 8.3.3. `accumulate`

*CPPReference: std::accumulate*

```
std::accumulate(first1, last1, init, [binary_op]);
```

Some numeric algorithms, like `accumulate`, can be used in non-numeric context. This function sums elements that are addable *(+ Operator)* starting at an initial value.

```
std::vector<std::string> months{"Jan", "Feb", ..., "Dec"};
auto accumulatedString = std::accumulate(
  next(begin(months)), // Second Element
  end(months), // end
  months.at(0), // first element, usually the neutral element
  [](std::string const & acc, std::string const & element) {
    return acc + ", " + element;
}); // "Jan, Feb, ..., "Dec"
```

## 8.4. `STD::REMOVE`, ERASE-REMOVE-IDIOM

*CPPReference: std::remove*, *CPPReference: std::erase*

```
std::remove(first, last, value);
```

***`std::remove`*** doest ***not*** actually ***remove*** the elements, it ***moves*** the "not-removed" elements to the ***front*** and ***returns an iterator*** to the end of the "new" range. The "removed" elements can still be dereferenced, but their behavior is undefined. To get ***rid*** of the "removed" elements, usually the ***`erase` member function*** is called.

```
auto values = std::vector{54, 13, 17, 95, 2};
auto is_prime = [](unsigned u) { ... };
auto removed = std::remove_if(
  begin(values), end(values), is_prime);
values.erase(removed, values.end());
```

## 8.5. _IF-VERSIONS OF ALGORITHMS

Some algorithms have a variation with the `_if` suffix. They take a predicate *(instead of a value)* to provide a condition.

```
auto numbers = std::set{1,2,3,4,5,6,7,8,9};
auto isPrint = [](auto u) { /* ... */ }
auto noOfPrimes = std::count_if(
  begin(numbers), end(numbers), isPrime);
```

**Algorithms with the `_if` Suffix**

| | | | |
|---|---|---|---|
| `count_if` | `find_if` | `replace_if` | `remove_if` |
| `copy_if` | `find_if_not` | `replace_copy_if` | `remove_copy_if` |

## 8.6. _N-VERSIONS OF ALGORITHMS

The _n suffix is related to a number provided instead of the "last" iterator.

```
auto numbers = std::set{1,2,3,4,5,6,7,8,9};
auto top5 = std::vector<int>(5);
std::copy_n(rbegin(numbers), 5, begin(top5));
```

**Algorithms with the _n suffix**

| | | | | |
|---|---|---|---|---|
| `search_n` | `fill_n` | `for_each_n` | `copy_n` | `generate_n` |

## 8.7. HEAP ALGORITHMS

*CPPReference: Algorithms – Heap operations*

A heap can be implemented on any ***sequenced container*** with ***random access iterators*** *(i.e. vector)*. Containers with the heap property in C++ are essentially ***balanced binary trees***.
***Guarantees:*** Top element is the largest, adding and removing elements have performance guarantees. Used for ***implementing priority queues***.

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(), v.end());
pop_heap(v.begin(), v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(), v.end());
sort_heap(v.begin(), v.end());
// Corresponding images in the slides
// Week 8, page 32 onwards
```

**Heap operations**

`make_heap`, `pop_heap`, `push_heap`, `sort_heap`.

***The most important operation is `make_heap`:*** It can be applied to any range with a random access iterator to turn it into a heap. It rearranges the elements to satisfy the max-heap property: ***Each parents note is greater or equal to the value of its children***. The first element in the container is the root node, with the second and third being its children and the next four elements being their child nodes.

The ***heap creation process*** starts at the bottom of the tree at the last non-leaf node to the root. Its value is then compared to its children and if one of the children is greater, the nodes will swap places. This continues until the root is reached and the heap is sorted.

## 8.8. PITFALLS

### 8.8.1. Mismatching Iterator Pairs

It is ***mandatory*** that the iterators specifying a range ***need to belong to the same range***. Otherwise, access of the value might result in undefined behavior.

```
std::vector<int> first{1, 2, 3};
std::vector<int> second{4, 5, 6};
auto func = [](int i) { ... };
std::for_each(std::begin(first), std::end(second), f); // iterators from different objects, UB!
```

### 8.8.2. Not reserving enough space

If you use an ***iterator*** for ***specifying the output*** of an algorithm, you need to make sure that ***enough space is allocated***.

It is possible to insert elements into a container ***without*** pre-allocating the required memory with the inserter functions `back_inserter`, `front_inserter` and `inserter` that call the respective member functions of the container.

```
std::set<unsigned> numbers {1,2,3,4,5,6,7,8,9};
std::vector<unsigned> primes{};
auto is_prime = [](unsigned u) { /* ... */ };
std::copy_if(begin(numbers), end(numbers), begin(primes), is_prime); // not enough space, UB!
std::copy_if(begin(numbers), end(numbers), back_inserter(primes), is_prime); // works :)
```

### 8.8.3. Input Invalidation

Some operations on containers ***invalidate its iterators***. ***Example:*** `std::vector<T>::push_back`. If the new `size()` is greater than the `capacity()`, then all iterators and references are ***invalidated.*** This means it is ***undefined behavior*** to do a `push_back()` on a container inside a `std::for_each` or similar.

# 9. FUNCTION TEMPLATES

CPPReference: Templates, CPPReference: Function templates

```
template <Template-Parameter-List> FunctionDefinition
```

Function Templates are the C++ way to ***create generic code*** that can work with different types. The keyword `template` is used for declaring a template. A template parameter is a ***placeholder for a type***, which can be used within the template as a type. A type template parameter is introduced with the `typename` keyword *(in older C++ standards, the `class` keyword was used, but it was changed as the type doesn't need to be a class)*

**The compiler …**

- ***resolves*** the function template *(checks which function (template) to use)*
- figures out the template ***argument(s)***
- ***instantiates*** the template for the arguments *(creates code with template parameters replaced)*
- ***checks*** the types for correct usage

## 9.1. TEMPLATE DEFINITION

CPPReference: Template parameters

Templates are usually ***defined*** *(not just declarated)* in a header file, because a compiler needs to see the whole template definition to create an instance.

***Type checking*** happens twice:

- ***During definition:*** Only basic checks are performed: Syntax and resolution of names independent of the template parameters.
- ***During template instantiation (writing code calling the template):*** The compiler checks whether the template arguments can be used as required by the template.

C++ Templates use ***duck-typing***: Every type can be used as argument as long as it supports the used operations.

### 9.1.1. Example Usage

```cpp
// Min.hpp
template <typename T>
auto min(T left, T right) -> T {
  return left < right ? left : right;
}
```

```cpp
// Smaller.cpp
#include "Min.hpp"
#include <iostream>

auto main() -> int {
  int first; int second;
  if (std::cin >> first >> second) {
    auto const smaller = min(first, second);
    std::cout << "Smaller of " << first
              << " and " << second
              << " is: " << smaller << '\n';
  }
}
```

## 9.2. TEMPLATE CONCEPTS

CPPReference: Constraints and concepts

A concept is the ***requirements a type must fulfill*** to be usable as an argument for a specific template parameter.
The requirements of the type `T` in the `min` template above are:

- ***Comparable with itself:*** The < operator is used to compare two elements of type `T`
- ***Copy/Move constructible:*** The template creates a new instance of `T` to return the result by value

C++20 allows to explicitly specify concepts to allow better checking of the template definition *(are all requirements fulfilled?)* and allows for easier to read error messages for failed template instantiations.

### 9.2.1. Example: What are the Concepts?

```
template<typename InputIt1, typename InputIt2, typename T>
auto inner_product(InputIt1 first1, InputIt2 last1, InputIt2 first2, T init) → T {
  while (first1 ≠ last1) { // check if still in range of first
    init = init + *first1 * *first2; // dereference the iterators, multiply the values & add to
sum
    ++first1; ++first2; // increment iterators of both ranges
  }
  return init;
}
```

| *InputIt1/InputIt2* | *init + *first1 * *first2* | *T* |
|---|---|---|
| – `*`: Dereferenceable<br>– `++`: Prefix increment<br>– `≠`: `InputIt1` with itself, result convertible to `bool` | – `*`: Multiplication on `*first1` and `*first2`<br>– `+`: Addition on `T` and result of above | – `=`: Assignable from result of `init + *first1 * *first2`<br>– Copy/Move constructable due to `return` as value |

## 9.3. ARGUMENT DEDUCTION

*CPPReference: Template argument deduction*

The compiler will try to figure out the function template's arguments from the call by pattern matching on the function parameter list. If the type is ***ambiguous***, it cannot figure out the arguments. For example, if there is a template `min(T left, T right)` and a function calls `min(1, 1.0)`, the compiler doesn't know if `T` should be `int` or `double`.

## 9.4. VARIADIC TEMPLATES

*CPPReference: Packs*, *CPPReference: `sizeof...`*, *CPPReference: Fold*

In specific cases, the number of template parameters might not be fix/known upfront. Thus the template shall take an arbitrary number of parameters, so called ***Variadic Templates***.

**Syntax: Ellipses everywhere.**

1. ***Template Parameter Pack:*** In template parameter list for an arbitrary number of template parameters
2. ***Function Parameter Pack:*** In function parameter list for an arbitrary number of function arguments
3. ***Number of template arguments:*** After `sizeof` to access the number of elements in template parameter pack
4. ***Pack Expansion:*** In the variadic template implementation after a pattern

```
template<typename First, typename...Types>// 1.
auto printAll(First const & first,
  Types const &...rest) → void { // 2.
  std::cout << first;
  if (sizeof...(Types)) { // 3.
    std::cout << ", ";
  }
  printAll(rest...); // 4. (Recursion)
}
auto printAll() → void { } // Base Case

// Usage
int i{42}; double d{1.25}; string name{"Nina"};
printAll(i, d, name);

// ...xy fold together
// xy... fold out
```

The example uses ***recursion*** to handle each function parameter one by one: The value in `first` gets printed and all others in `rest` are the arguments for the recursive call, where the first element of `rest` gets placed in `first`.

For each *(recursive)* function call, the complier creates an instance of the template where the template types and function types are replaced by their actual type. However, this does not work if there are ***zero parameters remaining:*** The template requires at least one argument (`first`) to be called. To work around this, we ***create a new non-template function with no arguments*** that does nothing. It acts as our recursive base case.

## 9.5. TEMPLATE OVERLOADING

Multiple function templates with the ***same name*** can exist, as long as they can be ***distinguished*** by their parameter list. An overload for pointers is possible. This way, the content of ***reference types*** can be compared instead of pointer addresses.

```
template <typename T> // regular template
auto min(T left, T right) → T {
  return left < right ? left : right;}

template <typename T> // overload for pointers
auto min(T * left, T * right) → T * {
  return *left < *right ? left : right;}
```

***Function templates*** and ***"normal" functions*** with the same name can coexist as well. When called with `std::string`, the pointer overload would only compare the first `char`s of the string, so a non-template function specifically for string comparisons can be created.

```
auto min(char const * left, char const * right)
  → char const * { // non-template function
  return std::string{left}<std::string{right}
    ? left : right;}
```

## 9.6. GENERIC LAMBDA

Operators and member functions can be templates too. Lambdas are internally converted to templates. ***Beware:*** Don't make operator templates too eagerly, you might end up with unexpected matches for other calls!

```
auto const printer = [&out](auto const & e) {
  out << "Element: " << e; }; // converted to:
struct __PrinterLambda {
  template <typename T>
  auto operator()(T const & e) const → void {
    __out << "Element: " << e;
  } std::ostream& __out; }
```

## 9.7. TEMPLATE GOTCHAS

### 9.7.1. Literals and references

Because strings are ***arrays of `chars`*** referenced on the heap, problems can occur if you try to ***compare*** two strings which do not have ***equal size***. With the use of ***String Literals***, this can be fixed *(Conversion into `std::string`)*.

```
template <typename T>
auto min(T const & left, T const & right)
  → T const & {
  return left < right ? left : right;
}
```

```
std::cout << min("C++", "Java");
// ⟶ error: no matching function for call to
// 'min(const char[4], const char[5])'. Fix:
using namespace std::string_literals;
std::cout << min("C++"s, "Java"s);
```

### 9.7.2. Matching and Overloading

Sometimes, the template might be a better match and ***overload*** your function you want to call. `const` and non-`const` values and parameter are prone to this.

```
std::string small{"aa"};
std::string capital{"ZZ"};
std::cout <<< min(small, capital) << '\n'; //ZZ
```

```
template <typename T>
auto min(T & left, T & right) → T {
  return left < right ? left : right;
}
//Gets overloaded by the template because
strings are not const:
auto min(std::string const & left, std::string
const & right) → std::string { /* ... */ }
```

### 9.7.3. Invalid Template

A temporary *(value not stored in a variable)* might become invalid, because the lifetime of temporaries ends at `";"`. `const &` can extend the lifetime of a temporary, but only if it is a temporary value as a result of the ***outermost expression***.

**Example:** `smaller` is a invalid reference because the arguments are only valid within the `min()` function.

```
template <typename T>
auto min(T const & left, T const & right) → T
const & {
  return left < right ? left : right;
}
std::string const & smaller = min("a"s, "b",s);
std::cout << "smaller is: " << smaller;
```

## 10. CLASS TEMPLATES

*CPPReference: Class template*

In addition to functions, class types can have template parameters as well.

```
template <Template-Parameter-List> class TemplateName { /* ... */ }
template <typename T> class Sack { /* ... */ }
```

A class template provides a type with ***compile-time parameters***. Data members can depend on template parameters. Function members are ***implicit template functions*** with the class' template parameters.

**Note:** Function members can be defined as template member functions with ***additional*** template parameters. They will then have the template parameter of their class, as well as the newly defined ones.

#### 10.0.1. Example Usage

```
template <typename T> // One template parameter
class Sack {
  using SackType = std::vector<T>;
  //"typename" keyword required, create new type
  // size_type -> dependent name
  using size_type = typename SackType::size_type;
  SackType theSack{};
public:
  auto empty() const -> bool {
    return theSack.empty(); }
  auto size() const -> size_type{
    return theSack.size(); }
  auto putInto(T const & item) -> void {
    theSack.push_back(item); }
  // Member function forward declaration
  auto getOut() -> T;
};
```

```
// Example for implementing member function
// outside of a class, requires 'template' KW
template <typename T>
auto Sack<T>::getOut() -> T {
  if (empty()) {
    throw std::logic_error{"Empty Sack"};
  }
  auto index = static_cast<size_type>(
    rand() % size()); // pick random element
  T returnvalue{theSack.at(index)};
  theSack.erase(theSack.begin() + index);
  return returnvalue;
}
// Concept for Sack's T:
// - T is assignable (implied by std::vector)
// - T is copyable (push_back & copy
//   constructor in on 'returnvalue')
```

### 10.1. TYPE ALIASES AND DEPENDENT NAMES

*CPPReference: Type alias*, *CPPReference: Dependent names*

It is common for template definitions to define type aliases in order to ***ease their use***. Less typing and reading, single point to change the aliased type. This could even be a template itself *(Outdated `typedef` keyword does not allow templates)* These are called ***Type Aliases*** or ***Alias Templates***.

```
using Typename = AliasedType; → using SackType = std::vector<T>;
```

#### 10.1.1. `typename` for Dependent Names

*CPPReference: `typename` keyword*

Within the template definition you might use names that are directly or indirectly ***depending*** on ***template parameter*** *(i.e. `std::vector<T>` depends on `T`)*. The compiler assumes that a name is either a variable or a function name. If a name should be interpreted as a type, you have to explicitly tell the compiler this with the ***`typename` keyword***. When the ***`typename` keyword*** is ***required,*** you should extract the type into a ***type alias***.

```
using size_type = typename SackType::size_type;
```

### 10.1.2. Example

```
// Accessing a member of a template parameter
template <typename T>
void accessTsMembers() {
  typename T::MemberType m{}; // keyword req.
  T::StaticMemberFunction(); // no keyword
  T::StaticMemberVariable; // no keyword
}
```

```
struct Argument {
  struct MemberType{};
  static auto StaticMemberFunction() → void;
  static int StaticMemberVariable;
}
```

```
// Indirect dependency, 'typename' necessary because 'size_type' depends on T
template<typename T>
class Sack { using size_type = typename std::vector<T>::size_type; }
```

### 10.1.3. Members Outside of Class Template

Members can be defined out ouf the class template. They still must be ***inline***, but it is ***implicitly*** inline as it is a function template.
For a ***full example*** see “Example Usage” (Page 45), right side.

```
template <typename T> // repeat template decl.
auto Sack<T>::getOut() → T // Member signature
Sack<T>:: // Template ID of sack as name scope
```

### 10.1.4. Rules

– Define class templates ***completely in header files***. The ***member functions*** can be directly in the class template *(recommended)* or as an inline function template in the ***same*** header file.

– When using language elements ***depending*** directly or indirectly on a ***template parameter***, you must specify ***`typename`*** when it is naming a type.

– ***`static` member variables*** of a template class can be defined in the header without violating ODR *(One definition rule)*, even if included in ***several*** compilation units. They can even be declared ***inside*** the class template, this requires the ***`inline`*** keyword. *(i.e. `inline static int member{sizeof(T)}`)*

## 10.2. INHERITANCE

When a class template ***inherits*** from another class template, ***name-lookup*** can be surprising!
**Rule:** Always use `this→` or the class `name::` to refer to inherited members in a template class. If the name could be a ***dependent name***, the compiler will not look for it when compiling the template definition *(Thus eventual unqualified variables/functions will be accessed, see example below)*. Checks might only be made for dependent names at template usage.

```
template <typename T>
struct Parent {
  auto foo() const → int {
    return 42;
  }
  static int const bar{43};
};
auto foo() → int {
  return 1;
}
double const bar{3.14};
```

```
template <typename T>
struct Child : Parent<T> {
  auto demo() const → void {
    out << bar;          // 3.14
    out << this→bar;     // 43
    out << Child::bar;   // 43
    out << foo();        // 1
    out << this→foo();   // 42
    out << Child::foo(); // 42
  }
}
```

## 10.3. PARTIAL SPECIALIZATION

*CPPReference: Partial template specialization*

Like function template overloads, we can provide “template specializations” for class templates. These can be ***partial*** still using a template parameter, but provide some arguments. Or complete ***explicit*** specializations, providing all arguments with concrete types.

One must declare the ***non-specialized*** template ***first.*** The ***most specialized version that fits is used***.

```
// Partial Specialization for all pointers
template <typename T>
struct Sack<T *>;
```

```
// Explicit Specialization for std::string
template <>
struct Sack<char const *>;
```

Class template specializations can have ***any content***, even no content at all. There is really no relationship apart from the template name.

### 10.3.1. Preventing Creation of a partial specialization

To prohibit instantiating a class is to prohibit the ability to its destructor. ***If an object cannot be destroyed, it cannot be created.*** This can be done by declaring its ***destructor*** as `= delete;`.

```
template <typename T>
struct Sack<T *> { ~Sack() = delete; }
// now a sack of pointers cannot be created
```

## 10.4. ADAPTING STANDARD CONTAINERS

Possible adaptations that could be implemented by you *(yes, you!)*

- ***`SafeVector:`*** no undetected out-of-bounds access
- ***`IndexableSet:`*** provide "[]"
- ***`SortedVector:`*** guarantee sorted order of elements

To build these extensions, create a template class inheriting from template base class and ***inherit constructors of the standard container*** *(instantiates the container directly when instantiating the extension class)*.

```
template<typename T>
struct SafeVector : std::vector<T> {
  using container = std::vector<T>;
  using container::container; // inherits constructors
  using size_type = typename container::size_type; //type alias
  using reference = typename container::reference;
  using const_reference = typename container::const_reference;
  reference operator[](size_type index) {
    return this→at(index);
  }
  const_reference operator[](size_type index) const {
    return this→at(index);
  }
}
```

***Caution:*** no safe conversion to base class, no polymorphism

For a full example about Extending a Template, see slides of week 10, page 29.

### 10.4.1. Extending the Sack Template

What should it be able to do?

- Create a `Sack<T>` using iterators to fill it
  ```
  std::vector values{1, 5 , 7, 12};
  Sack<int> sack{begin(values), end(values)};
  ```
- Create a `Sack<T>` of multiple default values
  ```
  Sack<unsigned> sack(10, 3);
  ```
- Create a `Sack<T` from a initializer list `Sack<char> charSack{'a', 'c', 'a', 'b'};`
- Obtain copy of contents to store in a `std::vector`
  ```
  Sack<int> sack{1, 2, 3};
  auto v = static_cast<std::vector<int>>(sack);
  ```
- Auto-deducing `T` for a `Sack<T>` from an initializer list
  ```
  Sack c{'n', 'g'}; Sack i{begin(v), end(v)};
  ```
- Allow to vary the type of the container to be used
  ```
  Sack<unsigned, std::set> sack{1, 3, 9};
  ```

## 10.5. DEDUCTION GUIDES

*CPPReference: Class template argument deduction (CTAD)*

Class template arguments can usually be ***determined by the compiler***. The behavior is similar to pretending as if there was a factory function for each constructor *(i.e. a `make_sack(T content)` that returns a `Sack<T>` with the content in it)*

### 10.5.1. User Provided Deduction Guides

In some cases, the compiler does deduct the ***wrong template***. Consider the example below:

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

```
TEST_CASE("suprisingDeduction") {
  std::vector values{1, 2, 3, 4, 5, 6};
  Sack sack{begin(values), end(values)};
  REQUIRE(sack.size() == values.size());
  // results in "2 == 6"
}
```

`Sack sack{begin(values), end(values)}` will not initialize the sack with the contents of the iterator range, but will place the two iterators themselves into the sack. This is because the compiler doesn't know which type the vector should contain – ***the template type `Iter` has no relation with `T`***. If the braces in the call are replaced with parenthesis, this works.

***User-defined deduction guides*** that show the compiler when to use what template can be specified in the same scope as the template. Usually after the template definition itself. It might be necessary for a ***complex case***, for example if the constructor template parameters do not map directly to the class parameters. Most of the time, the deduction guide is also a template and looks similar to a free-standing constructor declaration.

```
TemplateName(ConstructorParameters) -> TemplateID;
```

**Example:**

```
template <typename Iter>
// Constructor signature                     // Deduced template instance
Sack(Iter begin, Iter end) -> Sack<typename std::iterator_traits<Iter>::value_type>;
// Meaning: Use this constructor if template type 'Iter' is a iterator of value types.
```

After adding the deduction guide, the test case above for deducing the template argument from iterators works correctly. But now, using the constructor for creating a Sack with n-times a value doesn't work anymore. An additional template is required so this constructor can be called again. No deduction guide is needed because the compiler can deduce T for `Sack<T>` from the second parameter.

```
Sack sack(10, 3u); // this calls the Iter templ
template<typename Iter>
// Fails, because 'unsigned' is not an iterator
Sack(unsigned begin, unsigned end) ->
Sack<typename
std::iterator_traits<unsigned>::value_type>
// Explicit constructor for n-times value sack
Sack(size_type n, T const & value)
  : theSack(n, value)
```

## 10.6. TEMPLATE TEMPLATE PARAMETER

A template can take other templates as parameters, a ***template template parameter***. This allows us to swap the underlying container of our `Sack`.
The template template parameter must specify the ***number of parameters***. But standard containers usually take more than just the element type. We can fix this by leaving the number of template parameters ***unspecified*** with `template<typename...>` to allow an arbitrary number of parameters.
Our ***`getOut()`*** function also needs a small rewrite to work with container without index access.

```
template <typename T,
  template<typename...> typename Container>
class Sack;

Sack<unsigned, std::set> aSack{1,2,3}

auto getOut() -> T { // generalize for all
  throwIfEmpty();    // types of container
  auto index = static_cast<size_type>
    (rand() % size());
  std::advance(it, index);
  T returnvalue{*it};
  theSack.erase(it); return returnvalue;
}
```

C++ allows ***default arguments*** for function and for template parameters:

```
template <typename T, template<typename...> typename Container = std::vector>
class Sack;
```

### 10.7. NON-TYPE TEMPLATE PARAMETERS

Useful for specifying ***compile-time values***. If the type of the non-type template parameter should be flexible, ***auto*** can be used.

```
template <typename T, std::size:t n>
// template <typename T, auto n> can be used as well
auto average(std::array<T, n> const & values) {
  auto sumOfValues = accumulate(begin(values), end(values), 0);
  return sumOfValues / n;
}
```

### 10.8. VARIABLE TEMPLATES

*CPPReference: Variable template*

It is also possible to specify a template for a variable. The template can be specialized and is usually a `constexpr`. The purpose is to provide compile-time predicates and properties of types, which is useful for template meta programming.

```
template<typename T> // cast pi to other types
constexpr T pi = T(3.14159265358979);
template<typename T> //for all types except int
constexpr bool = is_integer = false;
template<> // template for just int
constexpr bool = is_integer<int> = true;
```

### 10.9. BEST PRACTICES

- Create *(partial)* ***specialization*** if the class template should behave differently for specific arguments
- Specify ***type aliases*** to be expressive and have only a single location to adapt them
- Access ***inherited members*** from other class templates with ***this->*** or ***base::***
- ***Inherit constructors*** when deriving from a standard container
- ***Deduction guides*** help the compiler deducing the template arguments

## 11. HEAP MEMORY MANAGEMENT

Stack memory is ***scarce***. The heap memory might also be needed for creating object structures or for polymorphic factory function to class hierarchies. Example for the latter: If we have a function that creates instances of class `Circle` and the result should be stored in a variable of base class `Shape`, we can't just return a value, because the `Circle` part will just get "thrown away". Thus we need to return a pointer to the `Circle` instance.

***Always rely on libary classes for managing heap memory!***

***Resource Acquisition is Initialization*** (RAII) Idiom
- Allocation in the constructor
- Deallocation in the destructor
- Use RAII wrapper as value in local scope
- Destructor will be called when the scope is exited
  *("}", return or exception)*

```
struct RaiiWrapper {
  RaiiWrapper() { /* Allocate Resource */ }
  ~RaiiWrapper() { /* Deallocate Resource */ }
}
```

C++ allows ***allocating*** objects on the heap ***directly***, like in C. However, if done manually you are responsible for ***deallocation*** and risk undefined behavior *(Memory leaks, dangling pointers, double deletes)*! No garbage collection happens. ***Don't do this!***

```
auto ptr = new int{};
std::cout << *ptr << '\n';
delete ptr;
// Better: Use smart pointers (see below)
// Even better: Store value locally
// as value-type variable
```

C++ offers three types of ***type safe smart pointers:*** `std::unique_ptr` *(allows just one handler)*, `std::shared_pointer` *(allows multiple handlers)* and `std::weak_pointer` *(prevents circular dependencies from creating memory leaks)*. With these smart pointers, a manual call to `delete ptr;` is no longer required. But still: ***always prefer storing a value locally.***

## 11.1. STD :: UNIQUE_PTR<T>

*CPPReference: std :: unique_ptr* `#include <memory>`

The unique pointer is used for ***unshared heap memory***. Only a single owner exists. Pointer cannot be copied, but moved. This transfers ownership from one variable to another.

A unique pointer *(std :: unique_ptr<T>)* is obtained with `std::make_unique<T>()`. `std::make_unique<T>()` and `std::make_shared<T>()` are ***factory functions***.

```
auto factory(int i) → std::unique_ptr<X> {
  return std::make_unique<X>(i);
}
```

`unique_ptr`s are ***not suited*** for creating class hierarchies or data structures with multiple pointers *(i.e. double-linked-lists)*

### 11.1.1. Example

```
#include <iostream>
#include <memory>
#include <utility>

// transfer of ownership through return by value
auto create(int i) → std::unique_ptr<int> {
  return std::make_unique<int>(i);
}
auto main() → int {
  std::cout << std::boolalpha;
  auto pi = create(42);
  std::cout << "*pi = " << *pi << '\n'; // *pi = 42
  // bool is false if pi does not point to the int
  std::cout << "pi.valid? " << static_cast<bool>(pi) << '\n'; // true
  //explicit transfer of ownership from lvalue
  auto pj = std::move(pi);
  std::cout << "*pj = " << *pj << '\n'; // *pj = 42
  std::cout << "pi.valid? " << static_cast<bool>(pi) << '\n'; // false
} // pj goes out of scope, gets deallocated on heap (destructor is called)
```

When interfacing with C code, there may be functions that return pointers. These must be deallocated manually be calling `free(ptr)`. Wrapping these pointers in a `std::unique_ptr` ensures that they will be properly discarded.

### 11.1.2. Guidelines for `std::unique_ptr`

- ***As a member variable:*** To keep a polymorphic reference instantiated by the class or passed in as `std::unique_ptr` and transferring ownership *(i. e. a member variable that references a instance of `Cat` or `Dog` in base class `Animal`)*
- ***As local variable:*** To implement RAII. Can provide custom deleter function as second template argument to type that is called on destruction.
- ***`std::unique_ptr<T> const p {new T{}};` (Local):*** Cannot transfer ownership, cannot leak. But better use `std::make_unique<T>`.

## 11.2. STD :: SHARED_PTR<T>

*CPPReference: std :: shared_ptr* `#include <memory>`

`std::unique_ptr` allows ***only one*** owner and ***cannot*** be copied, but is only returned by value. `std::shared_ptr` works more like Java's references: It can be ***copied*** and ***passed around***. The last one ceasing to exist deletes the object.

You can create `std::shared_ptr` and associated objects of type `T` using `std::make_shared<T>(...)`. It allows all `T`'s public constructor's parameters to be used.

```
struct Article {
  Article(std::string title,
    std::string content);
  // ...
};
// Local variable stored on stack
Article cppExam{"How to pass?", "You can't"};
// Pointer to value stored on heap
std::shared_ptr<Article> abcPtr =
  std::make_shared<Article>("Alphabet", "ABC");
```

Use `std::shared_ptr` if you really need ***heap-allocated objects*** *(i.e. network graphs or trees)*, if you need to ***support run-time polymorphic container contents*** *(i.e. a list of `Animals` that can contain both `Cat` and `Dog`)*, class members that ***cannot*** be ***passed as reference*** *(i.e. members marked `static`)* or for Factory functions. But first check, if ***alternatives*** are viable: (`const`) references or plain member objects.

Copying/destroying a `std::shared_ptr` is slow due to the atomic reference counter.

The ***last*** `std::shared_ptr` handle ***destroyed*** *(by leaving the scope)* or ***manual reset*** *(by explicitly calling `reset()`)* will ***delete*** the allocated object. This is problematic with ***cyclic*** structures: When two objects reference each other, but there are no outside references, the objects cannot be reached anymore and should be deleted. They cannot be deleted however, because of their mutual references.

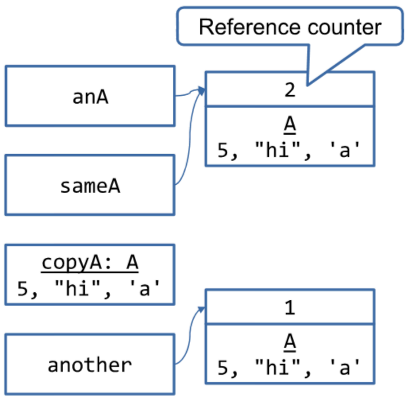
```
auto main() -> int {
  // Turn on
  auto light = std::make_shared<Light>();
  auto same = light;
  auto last = same;
  light.reset();
  same.reset();
  last.reset(); // Turn off
}
```

```
struct Light {
  Light() {
    std::cout << "Turn on\n";
  }
  ~Light() {
    std::cout << "Turn off\n";
  }
};
```

### 11.2.1. Example

If you really need to keep something explicitly on the heap, use a factory.

```
struct A {
  A(int a, std::string b, char c);
};
auto createA() -> std::shared_ptr<A> { // Factory function
  return std::make_shared<A>(5, "hi", 'a');
}
auto main() -> int {
  auto anA = createA();
  auto sameA = anA; // second pointer to the same object
  A copyA{*sameA}; // copy ctor
  auto another = std::make_shared<A>(copyA); // copy ctor on heap
}
```



### 11.2.2. Class Hierarchies

Use `std::ostream`, just as an example for a base class, and a very primitive factory function that creates an `ostream` which either prints to the console or to a file. The concrete type is required as template argument for `make_shared`.

```
auto os_factory(bool file) -> std::shared_ptr<std::ostream> {
  using namespace std;
  if (file) { return make_shared<ofstream>("hello.txt"); }
  else { return make_shared<ostringstream>(); }
}
aut main() -> int {
  auto out = os_factory(false);
  if (out) { (*out) << "hello world\n"; } // prints to console
  auto fileout = os_factory(true);
  if (fileout) { (*fileout) << "Hello, world!\n"; } // prints into file
}
```

### 11.2.3. Things to keep in mind when working with `shared_ptr`

– When the ***last*** `shared_ptr` handle is ***destroyed,*** the allocated object will be ***deleted.***
– If subclasses are stored in variables of type `std::shared_ptr<Base>` but are always created by a `std::make_shared<Sub>()`, the destructor no longer needs to be virtual, meaning you don't need to overload the destructor of the base class.
– `std::shared_pointer` can create cycles that ***cannot be cleared***, causing ***memory leaks***. Can be addressed with `std::weak_pointer`

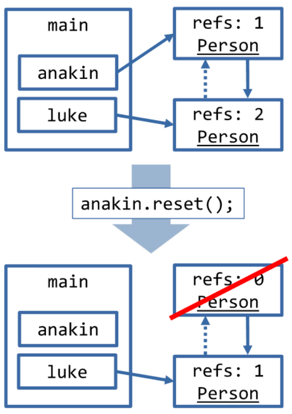
## 11.3. STD::WEAK_PTR

*CPPReference: std::weak_ptr*

We create a class `Person`. Each `Person` knows its `mother`, `father` and `child`. Each person can be married. This results in ***cycles*** – you cannot use values to store them, as that would mean copying `Persons` resulting in an infinite recursion. This task has to be solved with pointers.

To break the cycles, we can use ***`std::weak_ptr`***. They do not allow direct access to the object and do not count as a reference when determining if an object should be deleted. To aquire the object, `lock()` can be called on the `weak_ptr` to acquire a `std::shared_ptr`.

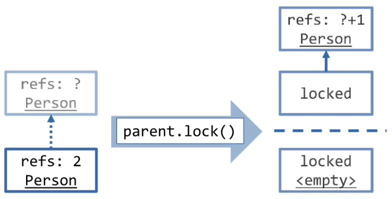
```
struct Person {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;
};
auto main() → int {
  auto anakin = std::make_shared<Person>();
  auto luke = std::make_shared<Person>();
  anakin→child = luke;
  luke→parent = anakin;
  anakin.reset(); // removes the ref to anakin
  // anakin now has no strong references
  // object on the heap will be deleted
}
```



### Checking liveness of locked pointer

A `weak_ptr` does not know whether the pointee is ***still alive***. `std::weak_ptr::lock()` returns a `std::shared_ptr` that either points to the alive pointee or is empty. Before accessing, verify that the pointer is ***valid.***

```
auot Person::aquireMoney() const → void {
  auto locked = parent.lock();
  if (locked) { // object is alive
    begForMoney(*locked);
  } else { // object is dead
    goToTheBank();
  } }
```



## 11.4. SELF-REFERENCING POINTERS: _FROM_THIS()

*CPPReference: std::enable_shared_from_this*

It would be nice if parents could spawn their own children *(no, not like that (unfortunately))*.
We need a `std::weak_ptr/std::shared_ptr<Person>` to the this-object to assign `child.parent`. By ***publicly deriving*** from ***`std::enable_shared_from_this<T>`***, the member functions ***`weak_from_this()`*** and ***`shared_from_this()`*** are provided. The returned object internally stores a `weak_ptr` to the this object.
**Caution!** *When using `class`, make sure to publicly inherit, otherwise you will run into memory errors like `segfault: bad_weak_ptr`*
*`class Car : public std::enable_shared_from_this<Car> {}`*

```
struct Person
  : std::enable_shared_from_this<Person> {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;
}

auto spawn() → std::shared_ptr {
  child = std::make_shared<Person>();
  child→parent = weak_from_this();
  return child;
}
```

### 11.4.1. Having multiple children

Smart pointers can be stored in standard containers, like `std::vector`. An alias for a `Person` pointer that can be used in the type itself requires a ***forward declaration***.

```
using person_ptr = std::shared_ptr<struct Person>;
struct Person {
  private:
  std::vector<PersonPtr> children;
  std::weak_ptr<Person> mother;
  std::weak_ptr<Person> father;
};
```

## 12. DYNAMIC POLYMORPHISM

C++ default mechanisms support ***value classes*** *(no reference classes)* with ***copying/moving*** and ***deterministic lifetime***. Operator and function overloading and templates allow ***polymorphic behavior at compile-time***. This is often ***more efficient*** and avoids indirection at run-time.

***Dynamic polymorphism*** needs ***object references*** or ***smart pointers*** to work. This results in ***syntax overhead***. The base interface must be a ***good abstraction*** and copying carries the danger of ***slicing*** *(Object is only copied partially)*.

Implementing ***design patterns*** for run-time flexibility. Client code uses an abstract interface and gets parameterized / called with reference to a concrete instance *(see image about the `std::ios` hierarchy below)*.

But: if ***run-time flexibility is not required***, templates can implement many patterns with compile-time flexibility as well.

### 12.0.1. Reasons for using Inheritance

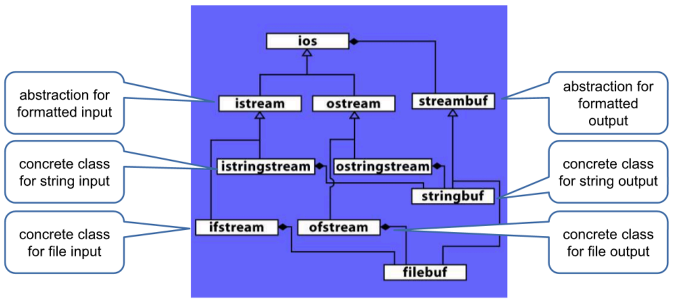
– ***Mix-in of functionality from empty base class:*** Often with own class as template argument, known as ***CRTP*** *(Curiously Recurring Template Pattern)* i.e. `boost::equality_comparable<T>`. No inherited data members, only added functionality *(A bit like C# constraints)*
  `struct Date : boost::equality_comparable<Date> { /* ... */ }`
– ***Adapting concrete classes:*** No additional own data members, convenient for inheriting member functions and constructors
```
template<typename T, typename Compare>
struct indexableSet : std::set<T, Compare> { /* ... */ }
```

### 12.1. INHERITANCE FOR DYNAMIC BINDING

Implementing a ***design pattern*** with dynamic dispatch. ***Provide*** common ***interface*** for a variety of dynamically changing or different ***implementations***, exchange ***functionality*** at run-time.

Base class/interface class provides a ***common abstraction*** that is used by clients.

For the inheritance Syntax, see “Inheritance” (Page 24).


ios
abstraction for formatted input
istream
ostream
streambuf
abstraction for formatted output
concrete class for string input
istringstream
ostringstream
stringbuf
concrete class for string output
concrete class for file input
ifstream
ofstream
filebuf
concrete class for file output

With interface inheritance, the base class must be `public`. Private inheritance is possible, but only useful for mix-in classes that provide a `friend` function. ***Most often, private base classes with members are wrong design!***

### 12.1.1. Initializing multiple base classes

Base constructors can be ***explicitly called*** in the member initializer list. If a constructor of a base class is omitted, its default constructor is called. The ***base class constructor*** should be ***placed before the initialization*** of subclass members. The compiler enforces this rule, even though you can put the list of initializers in the wrong order.

```
class DerivedWithCtor
  : public Base1, public Base2 {
  int myvar;
public:
  DerivedWithCtor(int i, int j)
  : Base1{i}, Base2{j}, myvar{j} {}
};
```

## 12.2. SHADOWING

If a function is reimplemented in a derived class, it ***shadows*** its counterpart in the base class. However, if ***accessed through a declared base object***, the shadowing function is ignored.

```cpp
struct Base {
  auto sayHello() const → void {
    std::cout << "Hi, I'm Base\n";
  }
};
struct Derived : Base {
  auto sayHello() const → void {
    std::cout << "Hi, I'm Derived\n";
  }
}
```

```cpp
auto greet(Base const & base) → void {
  base.sayHello();
}

auto main() → int {
  Derived derived{};
  greet(derived); // "Hi, I'm Base\n"
}
```

## 12.3. VIRTUAL MEMBER FUNCTIONS

Dynamic polymorphism requires base classes with ***`virtual`*** member functions. `virtual` is ***inherited*** and ***can be omitted in the derived class***. It is possible to mark an overriding function with ***`override`***. This does the same thing as `virtual`, except it ***throws an error*** if the function does not exist in the base class. To override a virtual function in the base class, the signature must be the same. ***Constness*** of the member function is ***part of the signature***.

```cpp
struct Base {
  virtual auto sayHello() const → void {
    std::cout << "Hi, I'm Base\n";
  }
};
struct Derived : Base {
  // auto sayHello() const override → void
  virtual auto sayHello() const → void {
    std::cout << "Hi, I'm Derived\n";
  }
}
```

```cpp
auto greet(Base const & base) → void {
  base.sayHello();
}

auto main() → int {
  Derived derived{};
  greet(derived); // "Hi, I'm Derived\n"
}
```

### 12.3.1. Calling `virtual` Member Functions

```cpp
struct Base { virtual auto sayHello() const → void; };
struct Derived : Base { auto sayHello() const → void; };
```

**Value Object**

Class type determines function, regardless of `virtual`. By passing as value, the inherited part gets left off. Just the base part of the object gets copied, see "Inheritance and Pass-by-Value" (Page 55).

```cpp
auto greet(Base base) → void {
  //always calls Base::sayHello
  base.sayHello();
}
```

**Reference**

Virtual member of derived class called through base class reference. By passing as reference, all *(child)* members are still there. The overridden methods can and will be used.

```cpp
auto greet(Base const & base) → void {
  //calls sayHello() of the actual type
  base.sayHello();
}
```

**Smart Pointer**

Virtual member of derived class called through smart pointer to base class.

```cpp
auto greet(std::unique_ptr<Base> base ) {
  //calls sayHello() of the actual Type
  base→sayHello();
}
```

**Dumb Pointer**

Virtual member of derived class called through base class pointer.

```cpp
auto greet(Base const * base) → void {
  //calls sayHello() of the actual type
  base→sayHello();
}
```

### 12.3.2. Abstract Base Class: Pure Virtual

There are ***no interfaces*** in C++. A pure virtual member function makes a class ***abstract***. To mark a virtual member function as pure virtual, it has ***zero assigned*** after its signature. ***No implementation*** needs to be provided for that function. Abstract classes cannot be instantiated.

```
struct AbstractBase {
  virtual void doitnow() = 0;
};
// cannot be instantiated:
AbstractBase create() {
  return AbstractBase{}; // does not work
}
```

## 12.4. DESTRUCTORS

Classes with virtual members require a ***virtual Destructor***. Otherwise when allocated on the heap with `std::make_unique<Derived>` and assigned to a `std::unique_ptr<Base>`, only the destructor of Base is called.

**Output non-virtual:**

```
put into trash // ~Fuel()
```

**Output virtual:**

```
store          // ~Plutonium()
put into trash // ~Fuel()
```

***Alternative:*** `std::shared_ptr` can memorize the actual type and knows which destructor to call. Instead of using the keyword `virtual` on the base destructor, the call in the main function can be replaced with
`... = std::make_shared<Plutonium>();`

```
struct Fuel {
  virtual auto burn() → void = 0;
  // Option 1: non-virtual
  ~Fuel() { std::cout << "put into trash\n"; }
  // Option 2: virtual:
  virtual ~Fuel() { std::cout <<
    "put into trash\n"; }
};
struct Plutonium : Fuel {
  auto burn() → void { std::cout <<
    "split core\n"; }
  ~Plutonium() { std::cout << "store\n"; }
};
auto main() → int {
  std::unique_ptr<Fuel> surprise =
    std::make_unique<Plutonium>();
}
```

## 12.5. PROBLEMS WITH INHERITANCE

Inheritance can be bad, because it ***introduces a very strong coupling*** between subclasses and their base class. An API of base class must fit for all subclasses, which is ***very hard to get right***.

Conceptual hierarchies are often used as examples but are usually ***very bad software design***.

### 12.5.1. Inheritance and Pass-by-Value

Assigning or passing by value a derived class value to a base class variable / parameter incurs ***object slicing***. Only base class member variables are transferred.

```
struct Base {
  // ...
  virtual auto modify() → void { member++; }
  // ...
};
struct Derived : Base {
  using Base::Base;
  auto modify() → void { member += 2; }
};
```

```
auto modifyAndPrint(Base base) → void {
  base.modify();
  base.print(std::cout);
}
auto main() → int {
  Derived derived{25};
  modifyAndPrint(derived);
}
// Output: 26
```

### 12.5.2. Problems with Member Hiding

Member functions in derived classes ***hide*** base class member with the ***same name***, even if different parameters are used.

**Example:** `Derived::modify(int)` hides `Base::modify()`. By "`using`" the base class member the hidden name(s) become visible: `using Base::modify;`

```
struct Base {
  int member{};
  explicit Base(int initial);
  virtual ~Base = default;
  virtual void modify();
}
```

```
struct Derived : Base {
  using Base::Base;
  using Base::modify;
  void modify(int value) {
    member += value;
  } // hides base function
}  // w/o using Base::modify
```

```
auto main() → int {
  Derived derived{25};
  derived.modify();
}
```

### 12.5.3. Assignment through References

Assignment cannot be implemented properly for ***virtual inheritance structures***. When assigning to a reference variable of the base class, the base part of a derived object gets ***overwritten.***

```
struct Animal {
  virtual classIsNowAbstract() = 0;
}
struct Cat : Animal { /*...*/ }
```

```
Cat elvis{};

// only the animal part gets copied
Animal & animal = elvis;
```

To prevent object slicing in the base class, you can declare the copy-operations as deleted.

**Problematic Example**

```
using Page = int; // shortcut for demo purposes
struct Book {
    explicit Book(std::vector<Page> p) : pages{p} {};
    virtual auto currentPage() const → Page = 0;
    auto lastPage() const → Page {
        return pages.size();
    }
protected:
    std::vector<Page> pages;
};

struct Ebook : Book {
    using Book::Book;
    auto currentPage() const → Page {
        return currentPageNumber;
    }
    auto openPage(size_t page) → void {
        currentPageNumber = page;
    }
private:
    size_t currentPageNumber{1};
};

auto readPage(Page page) {
   std::cout << "Page read: " << page << '\n'; }
auto makeEbook(size_t pageCount) {
   return std::vector<Page>(pageCount); }
```

```
auto main() → int {
  Ebook dune{makeEbook(869)};
  Ebook lordOfTheRings{makeEbook(1137)};
  lordOfTheRings.openPage(1000);
  Book &bookRef = lordOfTheRings;
  std::cout << "LotR pages to read: "
            << bookRef.lastPage() << '\n';
  readPage(bookRef.currentPage());
  bookRef = dune; // only base part copied over!
  std::cout << "Dune pages to read: "
            << bookRef.lastPage() << '\n';
  readPage(bookRef.currentPage());
}
```

**Output**

```
LotR pages to read: 1137
Page read: 1000
Dune pages to read: 869
Page read: 1000
```

Only the ***Book part*** of dune got ***copied*** into `bookRef`, the ***Ebook part remained values*** from `lordOfTheRings`. dune now has an ***invalid page number***. This can be prevented by ***deleting copy operations*** in book:

```
struct Book {
  auto operator=(Book const &other) → Book& = delete;
  Book(Book& const other) = delete;
}
```

## 12.6. GUIDELINES

- You should only apply inheritance and virtual member functions ***if you know what you do***
- Do ***not*** create classes with ***virtual*** members ***by default***
- If you design base classes with polymorphic behavior, ***understand the common abstraction*** that they represent *(Do not provide too many members or too few, extract from existing classes the base after you see the commonality arise)*
- Follow the ***Liskov Substitution Principle*** *(If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction)*. The Base class states must be valid for subclasses, do not break invariants of the base class, don't change semantics unexpectedly.

### 12.6.1. Polymorphism Example

```
struct Animal {
  auto makeSound() → void { cout << "---, "; }
  virtual auto move() → void { cout << "---, "; }
  Animal() { cout << "animal born, "; }
  ~Animal() { cout << "animal died, "; }
};

struct Bird : Animal {
  virtual auto makeSound() → void { cout<< "chirp, "; }
  auto move() → void { cout << "fly, "; }
  Bird() { cout << "bird hatched, "; }
  ~Bird() { cout << "bird crashed, "; }
};

struct Hummingbird : Bird {
  auto makeSound() → void { cout << "peep, "; }
  virtual auto move() → void { cout << "hum, "; }
  Hummingbird() { cout << "hummingbird hatched, "; }
  ~Hummingbird() { cout << "hummingbird died, "; }
};

// Destructors of subclass get called first.
```

```
auto main() → int {
  cout << "\n(a)---------------\n";
    Hummingbird hummingbird;
    Bird bird = hummingbird;
    Animal & animal = hummingbird;
  cout << "\n(b)---------------\n";
    hummingbird.makeSound();
    bird.makeSound();
    animal.makeSound();
  cout << "\n(c)---------------\n";
    hummingbird.move();
    bird.move();
    animal.move();
  cout << "\n(d)---------------\n";
}
```

**Output:**

```
(a)---------------
animal born, bird hatched, hummingbird hatched,
(b)---------------
peep, chirp, ---,
(c)---------------
hum, fly, hum,
(d)---------------
bird crashed, animal died,
hummingbird died, bird crashed, animal died,
```

## 13. INITIALIZATION & AGGREGATES

In C++, there are six different types of initialization:
- ***Default*** Initialization
- ***Value*** Initialization
- ***Direct*** Initialization
- ***Copy*** Initialization
- ***List*** Initialization
- ***Aggregate*** Initialization

They have four general syntaxes, the kind depends on the context.
1. Nothing
2. `( expression list )`
3. `= expression`
4. `{ initializer list }`

### 13.1. DEFAULT INITIALIZATION

*CPPReference: Default initialization*

The simplest form of initialization: Simply ***don't provide an initializer***. The effect depends on the kind of entity to declare. Does not work with references! It also doesn't necessarily work with `const`, as the object must have a valid value. Default initialized entities ***can be dangerous*** *(i.e. default initialized variables contain a random value)*. ***Should be avoided due to possible undefined behavior!***

**Effects:**
- ***`static` variables*** are ***zero initialized*** first, then their type's default constructor is called. If the type cannot be default constructed, the program is ill-formed!
- ***Non-`static`*** integer and floating point variables are ***uninitialized***
- Objects of ***`class` types*** are constructed using their default constructor
- ***Member variables*** not in a constructor initializer list are ***default initialized***
- ***Arrays*** initialize all elements according to their type

```
int global_variable; // implicitly static
auto di_func() → void {
  static long local_static;
  long local_variable;
  std::string local_text;
}

struct di_class {
  di_class() = default;
  char member_var; // not in ctor init list
};

struct no_default_ctor {
  no_default_ctor(int x);
};
no_default_ctor static_instance; // error!

auto print_uninitialized() → void {
  int my_number; // undefined behavior
  std::count << my_number << '\n';
}
```

## 13.2. VALUE INITIALIZATION

CPPReference: Value initialization

Initialization is performed with empty `()` or `{}`. `{}` is preferred, since it works in more cases. It ***invokes the default constructor*** for class types.

```
auto vi_function() → void {
  int number{};
  std::vector<int> data{};
  int actually_a_function(); // error
}
```

## 13.3. DIRECT INITIALIZATION

CPPReference: Direct initialization

Similar to value initialization, except the `()` or `{}` ***contain a value***. If `{}` are used, direct initialization is only used if the object us not a `class` type. Otherwise, list initialization is used. Danger of ***most vexing parse*** with `()`, thus `{}` is preferred.

```
auto diri_function() → void {
  int number{69};
  std::string text("CPl");
  word vexing(std::string()); // danger!
}
```

### 13.3.1. Most Vexing Parse

The compiler can interpret the following statement in two different ways: `word vexing(std::string());`

- ***Initialization of a variable*** called `vexing` of type `word` with a value-initialized `std::string()`
- ***Declaration of a function*** called `vexing` that returns `word` and taking an unnamed pointer to a function returning an `std::string`

While the first one is what we would expect, the second is what the standard requires. Therefore, prefer `{...}`

## 13.4. COPY INITIALIZATION

CPPReference: Copy initialization

Initialization using `=`. If the object is a `class` type and the right-hand side has the same type, the object is constructed ***"in-place"*** if the right side is temporary *(i.e. a function call)*. Otherwise, the ***copy constructor*** is invoked.
If it is not a `class` type or does not have the same type, a suitable conversion sequence is searched for.

This also applies to `return` statements and `throw`/`catch` blocks.

```
auto string_factory() → std::string {
  return "";
}

auto ci_function() → void {
  std::string in_place = string_factory();
  std::string copy = in_place;
  std::string converted = "CPl";
}
```

## 13.5. LIST INITIALIZATION

CPPReference: List initialization

Uses non-empty `{}`. Two varieties: ***Direct List Initialization*** and ***Copy List Initialization***.
Constructors are selected in two phases:

1. Check for a constructor taking `std::initializer_list`.
2. Other suitable constructor is searched

Since the `std::initializer_list` constructor is preferred, you might run into ***problems***.

```
std::string directListInit{"Nina"};
std::string copyListInit = {"Jannis"};

auto usesInitializerList() → int {
  // creates a vector with values '10' and '42'
  // not with 10 times '42'
  std::vector<int> data{10, 42};
  return data[5]; // undefined behavior
}
```

**Initialization Overview Example**

```cpp
// Aggregate Initialization
std::array<char const *, 4> names{{"Freely", "Cally", "Sofieus", "Avren"}};

void print_names(std::ostream & out) {
  std::size_t name_count; // Default Initialization
  name_count = names.size(); // NONE! Copy Assignment

  for(int i = 0; i < name_count; ++i) { // Copy Initialization
    std::string name{names[i]}; // List Initialization
    out << name << '\n';
  }
}

int main() {
  std::size_t name_count(names.size()); // Direct Initialization
  std::cout << "will print " << name_count << " names\n";
  print_names(std::cout);
}
```

## 13.6. AGGREGATE TYPES

*CPPReference: Copy initialization*

An aggregate type is a class with certain restrictions regarding its content. Classes that do not have these elements are automatically considered aggregates:

– No ***user-declared*** or ***inherited constructors***
– No `private` or `protected` ***non-static data members*** *(private/protected functions are allowed)*
– No `private`, `protected` or `virtual` ***base classes***
– No `virtual` ***member functions***

***All arrays are automatically aggregates.*** The big ***advantage*** of aggregate types is that they can easily be initialized with a initializer list. *(like `std::vector`)* Mostly used for simple types *(reduces initializing code)* and if the class has no invariant.

**Valid Aggregate**

```cpp
struct Person {
  std::string name;
  int age{42};
  auto operator<(Person const &other) const
    -> bool { return age < other.age; }
  auto write(std::ostream &out) const -> void {
    out << name << ": " << age << '\n';
  }
};
auto main() -> int {
  person nina{"Nina", 28};
  nina.write(std::cout);
}
```

**Invalid Aggregate**

```cpp
struct db_entry {}; // base class
// no aggregate: private base class
struct Person : private db_entry {
  std::string name;
  int age{42};
  // no aggregate: user-declared constructor
  Person() = default;
  auto operator<(Person const &other) const
    -> bool { return age < other.age; }
};
// no aggregate: virtual function
virtual auto write(std::ostream& out) const
  -> void { out << name ": " << age << '\n'; }
```

### 13.6.1. Aggregate Initialization

*CPPReference: Aggregate initialization*

Aggregates can be ***initialized like `std::vector`*** with the member values in `{}`. This special type of List Initialization is called Aggregate Initialization.
The members and base classes are initialized from the initializers in the list. If more elements than members *(or base classes)* are provided, the program is ***ill-formed***. If less elements are provided, the uninitialized members use their member initializer, if they have any. Otherwise, they are initialized from empty lists.

```cpp
Person nina{"Nina"}; // age will be set to 42
```

## 14. TEMPLATE PARAMETER CONSTRAINTS

With Template Parameter Constraints, the ***requirements*** of template parameter can be specified. This allows for earlier detection of type violations in the template instantiation and leads to ***better error messages***.

As an example, the code to the right does not compile because the overload selection fails due to `int` not being a class.

1. Name `increment` is looked up
   `increment(unsigned) → increment` Template
2. Template arguments are deduced
   `increment(unsigned) → increment<int>(int)`
3. Best overload is selected
   `increment(int)`

```
error: request for member 'increment' in
'value', which is of non-class type 'int'
```

```
auto increment(unsigned i) → unsigned
  { return i++; }

template <typename T>
auto increment(T value) → T
  { return value.increment(); }

auto main() → int
  { return increment(42); }
```

### 14.1. SFINAE

*(Substitution Failure is not an Error)*

When searching for an overload, the template parameters are replaced with the deduced types. This may result in template instances that ***cannot be compiled*** or otherwise suboptimal selection. If the substitution of template parameter fails that overload candidate is ***discarded***. It only results in an error if there are no more remaining overloads. Errors in the function body still result in errors.

Substitution failure might happen in the

1. Function return type
2. Function parameter
3. Template parameter deduction
4. Expressions in the above

#### 14.1.1. `std::is_class`

*CPPReference: `std::is_class`*, *CPPReference: `std::enable_if`*　　`#include <type_traits>`

In the introductory example, the `increment` template should only be selected for class type arguments. ***`std::is_class_v<T>`*** is a variable template that returns `true` if `T` is a class. It can be used as `V` in the type template ***`std::enable_if_t<V, T>`*** that converts a boolean value `V` into type `T` if `true`, or does nothing if `false`.

We can use it to provoke template substitution failures *(e.g. in the parameter declaration)*:

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

This works, but it is the ***ugly old-school way*** of specifying type constraints. The modern way are Constraints & concepts.

### 14.2. CONSTRAINTS WITH A `REQUIRES` CLAUSE

*CPPReference: Constraints and concepts*

`requires` clauses allow ***constraining template parameters***. A `require` keyword is followed by a compile-time constant boolean expression. They are placed after the template parameter list or after the functions template declarator.

```
// Declaration after template params
template <typename T>
requires true
auto function(T argument) → void {}
```

```
// Declaration after function template declarator
template <typename T>
auto function(T argument) → void requires true {}
```

For example, a `requires` clause can be created with `std::is_class`

```cpp
template <typename T>
requires std::is_class_v<T>
auto function(T argument) → void {}

function(1);
```

```
error: no matching function for call to
'function(int)'
note: constraints not satisfied
note: the expression 'is_class_v<T>' [with T = int]
evaluated to 'false'
```

### 14.3. `REQUIRES` EXPRESSION

*CPPReference: requires expression*

The `requires` keyword can also be used to create a ***`requires` expression:*** A `requires` clause with ***multiple statements*** that evaluates to ***`bool`***.

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

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

**Types of requirements**

- ***Simple requirements:*** Statements that evaluate to true when compiled
- ***Type requirements:*** Check whether a specific type exists *(typically for nested types)*
- ***Compound requirements:*** Checks constraints on an expressions type
- ***Nested requirements:*** Contain further (nested) requires expressions *(not covered in CPl)*

**A note on C++ grammar and the double `requires`**

Most `requires` expressions are used within a `required` clause, meaning that you'll often see `requires requires (...) { ... }`.

***Why is the second `requires` necessary?***

A `requires`-clause says "This function should be considered in overload resolution if this condition is true" - it can take any constant boolean expression. It can be written as `requires(foo)`, where `foo` is a boolean expression.
A `requires`-expression just asks the compiler "Are these expressions well-formed?". `requires(foo f)` is a valid `requires` clause. So from what point on can the parser be sure that it is a `requires`-clause and not a expression?

```cpp
void bar() requires (foo) {
  // content
}
```

If `foo` is a type, then `(foo)` is a parameter list of a `requires` expression and everything in the `{}` is the body of this `requires` expression. If `foo` is not a type, then `foo` is an expression in a `requires` clause and everything in the `{}` is the regular function body of `bar()`. While the compiler could theoretically clear up this ambiguity by figuring out if `foo` is a type or not, the C++ comittee decided to require two `requires` to avoid this kind of context-sensitive parsing.

#### 14.3.1. Simple Requirements

Simple requirements are ***statements that are `true` when they can be compiled***. In the example, code that calls this template can only be compiled if `T` can be replaced with a type that has a `increment()` member function.

```cpp
requires (T v) {
  v.increment();
}
```

```cpp
template <typename T>
requires requires (T const v) { v.increment(); }
auto increment(T value) → T {
  return value.increment();
}
```

### 14.3.2. Type Requirements

Type requirements ***check whether a specific type exists***, typically for nested types. It starts with the `typename` keyword. It can be used to ***specify what kind of types can be passed*** as template arguments. In the example, we see the `max_value` algorithm that gets the largest value between the `begin` and `end` iterators. In its `requires` expression is specified that both arguments should be iterators that point to a value type.

```
requires {
  typename $type$
}
```

```
template <typename FwdIter>
requires requires {
  typename std::iterator_traits<FwdIter>::value_type;
}
auto max_value(FwdIter begin, FwdIter end)
  → std::optional<typename std::iterator_traits<FwdIter>::value_type> {
  auto max_pos = std::max_element(begin, end);
  if (max_pos == end) { return {}; }
  return *max_pos;
}
```

### 14.3.3. Compound Requirements

Compound requirements ***check whether an expression is valid*** and can check constraints on the expression's type. Similar to a simple requirement, but it can also optionally specify a return type requirement with a type requirement.

```
requires (T v) {
  { $expression$ } → $type-constraint$;
}
```

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

### 14.3.4. Named constraints with the `concept` keyword

Specifies a type requirement with a ***name that can be reused***. Typically, a `requires` expression is part of a `bool` expression. Conjunctions (&&) and disjunctions (||) can be used to combine multiple constraints.

```
template <typename T>
concept TypeRequirementName = $bool-expr$
```

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

Named constraints can be used in template parameter declarations or as part of a `requires` clause.

```
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();
}
```

## 14.4. ABBREVIATED FUNCTION TEMPLATES

Definitions of function templates can be ***shortened*** by using `auto` as the generic parameter type. With this, the `template` expression can be omitted.

```
// abbreviated template definition
auto function(auto argument) → void {}
```

```
// equivalent "normal" definition
template <typename T>
auto function(T argument) → void {}
```

This syntax can introduce conflicts when multiple function parameters are used.
What function will `auto function(auto arg1, auto arg2) → void` pick?

```
template <typename T>
auto function(T arg1, T arg2) → void {}
```

```
template <typename T1, typename T2>
auto function(T1 arg1, T2 arg2) → void {}
```

The problem can be avoided with the ***Terse Syntax for Constrained Parameters:*** Abbreviated function template parameters can be constrained too.

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

```cpp
// equivalent "normal" definition
template <Incrementable T>
auto increment(T value) -> T {
  return value.increment();
}
```

### 14.4.1. Example Concepts for Output

Here are two concepts *(not the template functions that implement them!)* that can be used to output values. Templates constrained by `Printable` can be used with classes that have a `print(std::ostream&)` member function, while templates constrained by `LeftshiftOutputtable` can only be used with types that overload the << operator to work with `std::ostream`.

```cpp
template <typename T>
concept Printable = requires (T const v, std::ostream& os) {
  v.print(os);
};

template <typename T>
concept LeftshiftOutputtable = requires (T const v, std::ostream& os) {
  {os << v} -> std::same_as<std::ostream&>;
};
```

## 14.5. OVERLOADING ON CONSTRAINTS

Function overloads with ***unsatisfied constraints*** are excluded from overload resolution as well.

In this example, we have a generic function `printAll()` with a variadic amount of parameters in `rest`. Depending on whether the type in `first` has a `print(std::ostream&)` member function or the << operator overloaded for outputting its value, the first or the second overload of `print()` overload will be called.
For example, `int` does not have a `print()` member function *(because int is not a class type)*, but it does have the << operator implemented, so the `print(LeftshiftOutputtable)` overload will be called.

```cpp
auto print(Printable auto const& printable) {
  printable.print(std::cout);
}

auto print(LeftshiftOutputtable auto const& outputtable) {
  std::cout << outputtable
}

auto printAll(auto const& first, auto const&... rest) -> void {
  print(first);
  if constexpr (sizeof...(rest)) {
    std::cout << ", ";
    printAll(rest...);
  }
}
```