1 ToDo

1.1 TODO Add some Manipulators

2 General

Compilation Process

The C++ Compilation process consits of 3 steps:

1. Preprocessor (textual replacement in the files and execution of the #-strings
2. Compiler (compile each *.cpp file in to object code)
3. Linker (link all object code files to a library / executable / …

Best Practice Setup

A modern C++ project is divided in 3 subprojects:

- library: the whole application logic
- testing: project for Unit Testing (for example with CUTE)
- executable: small project with only a main function which calls the entrypoint in the library

Decleration / Definition

In C++ everything has to be declared before they can be used. Declarations are normally written in .h files. Multiple declaration for the same thing is sometimes allowed for example for functions. An example for a definition: `void sayHello(std::ostream &out);`
The definition is normally written in a .cpp file and contains the actual code. Each definition can only occure once. This is also called One Definition Rule. Example for a declaration: `void sayHello(std::ostream &out) { /* Body */ }`

ODR / One Definition Rule

In C++ everything can only have one definition. This is called the *One Definition Rule*.

Include Guards

In C++ declerations are included over the `#include` keyword. To prevent recursive includes the following code snippets is used. This is called *Include guard*:

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_
/* content */
#endif /* SAYHELLO_H_ */
```

Some compiler have the keyword `#pragma once`. This should not be used because it is not standard C++.

Value Semantic

C++ has a default value semantic. That means that every object is normally copied. In Java normally only the Reference to the object is copied.

References

In C++ references are not the same as in Java. References in C++ are just another name for a existing object. In a local context this is not useful. Used as parameters in function they can be very useful:

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

Casting

In modern CPP you use normally the `static_cast` operator.

```
enum class Month {
  Jan = 1, Feb, Mar, Apr, May, Jun, Jul,
      Aug, Sep, Oct, Nov, Dec
};

Month fromIntToMonth(int month) {
  if (month < 1 && month > 12) {
    throw std::invalid_argument{"moth␣
        must␣be␣between␣1␣and␣12"};
  }

  return static_cast<Month>(month);
}
```

Variable Definition

In modern CPP you should use the `{}` syntax to initialize new variables:

```
int anAnswer{42}; int zero{};
```

When you use the keyword `auto` (use it!) then the equal sign (`=`) should be used:

```
auto someType = 4;
```

Bit wise operations

When working with bit wise operations only unsigned integers should be used. Because the first bit in a signed integer indicates if it is a positive or negative number

Strings

In CPP exits two kinds of strings, the C style strings and the `std::string`. The C style should not be used anymore (when possible). A big different between Java and C# is that the string in CPP are mutable.
For each kind of string exits a string literal:

- `"ab"` is an array of const chars (is immutable)
- `"ab"s` is an std::string
  - this requires `using namespace std::literals;`

– this is sometimes used with `auto`

Input Streams

`std::cin` represents STDIN. When using `std::cin` with the `>>` operator it skips all kinds of white spaces. With `std::cin.get()` the white spaces are not skipped. The chars on a input stream ore only consumed if the reading was successfully. If does a read fail it goes in a fail state and all subsequent read fail as well. Before reading from a input stream object you should always check its state.

```
int inputAge(std::istream & in) {
  std::string line{};
  while (getline(in, line)) {
    std::istringstream is{line};
    int age{-1};
    if (is >> age) {
      return age;
    }
  }
  return -1;
}
```

Output Stream

The `std::cout` it the STDOUT. For modifying the behavior of the output stream so called Manipulators can be used.

Output Manipulators

The manipulators for output stream are defined in `<ios>` and `<iomanip>`

IO in CPP

The header files for IO are:

- *<iosfwd>* contains only declarations
- *<istream>* contains input implementation
- *<ostream>* contains output implementation
- *<iostream>* contains all of the previous inkl. `std::cout`, `std::in`, `std::cerr` (should only used in main)

Type Aliases

Using type aliases a long and complicated type could be asigned a shorter name. `using <alias-name> = <type>;`

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

When you should use references, when values

As a rule of thumb when to use C++ References go with the following:

1. value parameter - default
2. reference parameter - when side-effect is required at call side
3. const reference parameter - when type is large and no side-effects desired at call-site
4. const value parameter - just coding style, can prevent changing the parameter inside the function inadvertently (de. versehentlich)

When you should return references, when values

As a rule of thumb when to return C++ References

1. value - default
2. reference
   - return only a reference parameter or from a member function a member variable
   - **Never return a reference to a local variable!** (Undefined behavior)
3. by const value
   - **Do not do this**. It just annoys the caller

auto as return value

The `auto` keyword can be used as a return type. If `auto` is specified then the return value is deduced from return statements. You can also specify the return type with `auto` and append a `->` at the end with the return value. This is often used to align all function names.

```
auto middle(std::vector<int> const & c)
    -> int;
auto middle(std::vector<int> const & c)
    -> int {
  if (c.size() == 0) { throw new
      invalid_argument{}; }
  return c[c.size() / 2];
}
```

Default Arguments

```
// incr.h
void incr(int &var, unsigned delta = 1);

// incr.cpp
void incr(int &var, unsigned delta) {
  var += delta;
}
```

Default parameters can only be written at the end. Implicit overload of the function with fewer parameters. Default arguments can be omitted when calling the function.

Functions as parameters

```
void applyAndPrint(double x, double f(
    double)) {
  std::cout << "f" << x << ")␣=␣" << f(x
      ) << '\n';
}

void applyAndPrint(double x, double (&f)
    (double)) {
  std::cout << "f" << x << ")␣=␣" << f(x
      ) << '\n';
}

void applayAndPrint(double x, std::
    function<double(double)> f) { //
    recomended
  f(x);
}
```

A function parameter declared like `double f(double)` does not accept a lambda with a capture. Instead you should use `std::function<double(double)> f` this could be used for lambdas with captures. This is also the recommended way.

Lambdas with capture

If you want to assign a lambda to a variable you must use the `auto` keyword. Only the compiler knows the real type. In lambdas the parameter list can only omitted when the return type is not specified.

capturing a local variable by value:

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

capturing a local variable by ref:

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

capturing all local variables by value:

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

capturing all local variables by ref:

```
auto l = [&]() mutable {
  std::cout << ++x;
 };
```

`this` can be also captured

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

`x` is `auto` and is stored for all lambda calls (2, 4, 8, 16, …)

Exceptions

In CPP any copyable type can be thrown but you should only throw exception You do not need the *new* keyword. This would throw a pointer of an exception.

```
throw std::invalid_argument{"Description
    "};
```

In an Exception are no meta information available.

In CPP you have only `try` and `catch`. `finally` does not exists. The principle is throw by value, catch be `const` reference. This avoids unnecessary coping and allows dynamic polymorphism for class types.

The sequence of catches is significant because the first match wins. To catch all you can use the ellipsis `(...)`. This must be the last catch otherwise the later catches would never be executed.

To rethrow an exception just throw it again.

```
try {
  f("do␣something");
}
catch (std::invalid_argument const &c) {
  // do something
}
catch (...) {
  // catch all
}
```

Undefined Behaviour in Call Sequence

It is not defined in which order the function are evaluated. It could be that the first `inputName` and then the seconds is processed. It could be also the other way be. If you are not sure use variables to store the output of the function and use the variable as argument for the function call.

```
sayGreeting(std::cout,
            inputName(std::cin),
            inputName(std::cin));
```

3 STL Containers

3.1 Sequence Containers

Sequence Containers

All elements are accessible in the same order as they were inserted. To find an element in the container linear time ($O(n)$) is needed.

- `std::vector<T>` in `<vector>`
- `std::deque<T>` in `<deque>`
- `std::list<T> (double linked list)` in `<list>`
- `std::forward_list<T> (single list)` in `<forward_list>`
- `std::array<N, T>` in `<array>`

Attention: Both list has *NO* index access.
When you use sequence containers in CPP you should normally use the `std::vector` or the `std::deque` because they are very efficient. Except you use it in a bad way.

Array
CPP has two types of arrays:

- plain C-Arrays (should not be used)
- `std::array` (defined in `<array>`)

When you need arrays always use `std::array` instead of the C style arrays. For `std::array` you do not need to handle raw pointers. Additional `std::array` has some convenient features like storing the length of the array.
The size has to be known at compile time.

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

The std::array has two possibles to access the elements inside:

- using the `at()` function, this throws an exception on invalid index access
- using the `[]` operation, this has undefined behavior on invalid index access

vector
The vector in CPP is a dynamic sized container. To use it the header file `<vector>` needs to be included. The elements can be access with `at()` and `[]`. To insert a new element at the end the function `push_back()` is used. Also the function `insert()` exists to insert an element somewhere in the vector.

```
std::vector<int> v{};
v.push_back(2);
auto it = v.begin();
v.insert(it, 3);
```

Sequence Adapters in STL
In the std library exits LIFO and FIFO Adapters for the sequence containers. They are based on a real / full container. But they implement only the required functions.
LIFO Adapter:

- `std::stack` in `<stack>`

FIFO Adapter:

- `std::queue` in `<queue>`

The `std::priority_queue` is also just an adapter.

3.2 Associative Containers

Associative Containers
The elements in an Associative Container are **always** stored in sorted order. Because of that the `find()` member method runs in logarithmic time. So you should always use the `find()` method of the member itself.

- `std::set<T>` in `<set>`
- `std::multiset<T>` in `<set>`
- `std::map<K, V>` in `<map>`
- `std::multimap<K, V>` in `<map>`

These containers allow the use of functors to manipulate how the data should be stored.

```
#include <set>
#include <functional>

int main() {
  std::set<int, std::greater<>>
      reverse_int_set{};
}
```

Set
You can not modify the elements inside the set. This could break the order inside the data structure.
You use `.find()` / `.count()` member function because it is a lot faster. `.count(element)` will return 0 or 1. To check if the set contains an element use `.count(element) = 1`=
Map
The iterator access returns a `std::pair<key, value>`. The index operator inserts a *new* entry automatically if the key is not availabel.
To check if the set contains an element use `.count(element) = 1`=
Multiset
The multiset has three special member functions:

- `equal_range()`: returns a range (`std::pair`) containing all elements with the given key in the container
- `lower_bound()`: returns an iterator pointing to the first element that is *not less* than the key.
- `upper_bound()`: returns an iterator pointing to the first element that is *greater* than the key.

Multimap
The multimap has three special member functions:

- `equal_range()`: returns a range (`std::pair`) containing all elements with the given key in the container
- `lower_bound()`: returns an iterator pointing to the first element that is **not less** than the key.

- `upper_bound()`: returns an iterator pointing to the first element that is **greater** than the key.

3.3 Hashed Containers

The elements in a Hashed Container are stored in an unspecified order. The `.find()` member function runs in constant time ($O(1)). Therefore, you should use the member function to find elements inside a hashed container

- `std::unordered_set` in `<unordered_set>`
- `std::unordered_map` in `<unordered_map>`

Implementing your own Hash Function is hard and prone to failures. Stick to standard types like `std::string` for the keys.

3.4 Iterators

Iterator Kinds In CPP exits two kinds of iterators:

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

Input Iterator
The input iterator can only go in one direction and can only read. The iterator can be copied. But after an increment all other copies are invalid. But `*it++` is explicit allowed by the standard.

Forward Iterator
The forward iterator can do whatever an input iterator can. Additional it can:

- changing the "current" element (as long the container / element are not const)
- the iterator copy can be kept for later references

Bidirectional Iterator
The bidirectional iterator can do whatever the forward iterator can do. Additional it can:

- go backwards (`it--`)

Random Access Iterator
The random access iterator can do whatever the bidirectional iterator can do. Additional it can:

- directly access elements at index (positive / negative)
- go n steps forward / backwards
- subtract two iterators to get the distance
- compare with relational operators (`<`, `<=`, `>`, `>=`)

Output Iterator
The output iterator can only write a value to the current element. But this is only possible once (`*it=value`). After that the iterator has to be incremented (`it++`). The output iterator has not end-iterator.
Most other iterators can act also as an output iterator. Except the underlying container is `const`. The Associative Containers are an exception because they return only a read-only iterator.

Stream Iterator
Streams can not be used directly with the algorithms from the standard library. I have to create an iterator for a stream:

```
std::ostream_iterator<int>{std::cout, ", "}
std::istream_iterator<int>{std::cin, ", "}
```

The output stream iterator (`ostream_iterator`) does not have an `end()` marker (theoretically you can print forever on this stream). With this stream iterators type aliases are interesting.
The default constructed `std::istream_iterator<T>` marks the EOF. Normally an `istream_iterator` uses the `>>` operator for input. Only the `std::istreambuf_iterator<char>` uses the `std::istream::get()` function.

```
#include <sstream>
#include <iterator>
#include <algorithms>

int main() {
  std::istringstream str2("1 3 5 7 8 9 10");
  auto it = std::find_if(std::
      istream_iterator<int>(str2),
                         std::
                             istream_iterator
                             <int>(), //
                             EOF
                         [](int i){
                             return i%2
                             == 0;});
}
```

$\text{const}_{\text{iterator}}$ vs const iterator
If you declare an iterator as `const` (`const auto it`) you are not allowed to modify the iterator itself. Therefore, you can not increment the iterator (`it++`). Normally this is not what you want.
The $\text{const}_{\text{iterator}}$ itself is *NOT* `const`. But the

underlying element is `const` and can not be modified. The functions `cbegin()` and `cend()` return such an iterator.

Iterators for inserting

If you use an iterator to specifying the output for an algorithm you need to make sure that enough space is allocated. For this you could use the following functions to create an iterator:

- `std::back_inserter`: creates an iterator which uses the `push_back` member function
- `std::front_inserter`: creates an iterator which uses the `push_front` member function
- `inserter`: creates an iterator which uses the `insert` member function

4 Algorithms

Why should you use STL algorithm

Writing your own algorithms, especial loops, is error-prone. Additional it is difficult to read and understand what your code does. Also, the algorithms in the STL are often faster than your own algorithms.

algorithms

The CPP standard library offers us a lot of algorithms. All of them take iterators as an argument to iterate over the containers.

The algorithms are defined in:

- `<algorithm>`: `std::count`, `std::for_each`, …
- `<numeric>`: `std::accumulate`, `std::reduce`, …

– these algorithms are not restricted to numbers. Often only the plus / minus has to be overloaded.

Functor

A function is a type that provides the call operator.

```
struct caselessCompare {
  auto operator()(std::string rhs, std::
      string lhs) const -> bool {
    /* do some stuff */
    return true;
  }
};
```

The call operator can be overloaded multiple times. It can take any number of parameters and have an arbitrary return type.

Many algorithms in the STL take a function as an argument. Sometimes you provide just a Lambda. For very simple lambdas the `<functional>` has already some functors implemented:

Predicate

Predicates are functions / lambdas which return a boolean (or a type convertible to boolean). Predicates are used to check if a condition is met.

Remove Elements from STL

In CPP you use The Erase-Remove Idiom for removing elements from a STL container. The function `std::remove` does *NOT* actually remove the elements. Instead, it moves the "not-removed" elements to the front and returns an iterator to the end of the "new" range. Go get rid of the "removed" elements usually the `.erase()` member function must be called.

Erase-Remove Idiom In data structures which are based on arrays, such as `std::vector`, removing every single element would be very time intensive. To avoid this problem first all elements which should **not** be deleted are moved to the front. After that only once the `erase` function is called and all deleted elements are erased / removed.

Heap

The STL itself has no heap data structure. Instead, it uses a sequenced container to operate on. The following functions always create a max heap, but the behavior could be changed using functors:

- `make_heap()`: constructs a max heap
- `pop_heap()`: removes the largest element from a max heap
- `push_heap()`: adds an element to a max heap
- `sort_heap()`: turns a max heap into a range of elements sorted in ascending order

for loops

In CPP exists three types of for loops:

- index based, do not used this, except you need the index
- range base loops
- iterator based loops

This code is legacy and should not be used anymore. Instead use the algorithms from the standard library.

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

Try to avoid programming your won loops (if possible): oftern algorithms faster, the correctness is proofed, and the readability of the code is better.

lambdas
Lambdas in CPP have the following syntax:

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

std::for_each(v.begin(), v.end(), [&out
    ](auto x) {
  out << "print:␣" << x << '\n';
 });
```

Normally the return type does not have to be specified because the compiler can detect it correctly. Lambdas are internally implemented as functors.

5 Class

A good class
A good class has:

- high cohesion
- avoids deeply nested control structures
- provides a guarantee about its state
- properties for a value of the type that are always true *(establish invariant)* if error during constructing throw exception

classes and structs
Normally you write the class declaration in the header file and the implementation is in the Header file. The only difference between the keyword `class` and `strcut` is the default visibility. `class` is per default private, `struct` public. Additional to `public` and `private` there exits also the `protected` keyword.
It is possible to declare your member variables as `const` but often this is the wrong way because this would prevent copy assignments.

```
/* Date.h */
struct Date {
  Date(int year);
  bool isLeapYear(int year);
  bool isValidDate() const;
};

/* Date.cpp */
Date::Date(int year)
  : year{year} {/**/}


bool Date::isLeapYear(int year) { /**/ }

bool Date::isValidDate() const { /**/ }
```

Inheritance
In CPP multiple inheritance is possible. To inherit from a class you write the following code:

```
class MyClass : public MyBase, private
    MyBase2 {
  int mvar;

public:
  MyClass(int i, int j) :
    MyBase{i}, MyBase2{}, mvar{j} {}
};
```

Normally the visibility is `public`. But you can also set `protected` or `private`. This specifies the maximum visibility of the elements inherit from the base class.

Kinds of constructors
In CPP we have the following constructors which we can use in classes:

- default constructor
- copy constructor
- move constructor
- type conversion constructor
- list constructor

How you should initialize member variables In the constructor you should do it like in the following snippet. If you do it in an other way, the values are first default constructed and then reassigned.

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

If your class has some default values, you can initialize it directly in the header file. This is called NSDMI (Non Static Data Member Initializers):

```
class Date {
  int year{9999}, month{12}, day{31};

  /**/
};
```

Reimplement the default behaviour (constructor)
Using the following syntax you can create the default constructor type again after creating a custom one.

```
class Date {
  Date(int year);
  Date() = default;
};
```

Using `delete` the implicit constructor type is removed

```
class Date {
  Date() = delete;
};
```

Copy Constructor
The copy constructor is a type of constructors in CPP to copy all members to an new object.

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

Move Constructor
The move constructor is a type of constructors in CPP to move all members to an new object.

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

Type Conversion Constructor
The Type Conversion constructor is a type of constructors in CPP to convert the input in the object. It is recommended to set the `explicit` keyword. If not set it could lead to unexpected conversions.

```
class Date {
  explicit Date(std::string const &);
};
```

List Constructor
The List Constructor is a type of constructors in CPP to fill a container with content.

```
class Container {
  Container(std::initializer_list<
      Element> elements);
};

Container box{item1, item2, item3};
```

Destructor
The Destructor is the counter part to the constructor. It must release all resources and is not allowed to throw an exception. If you program properly you will hardly ever need to implement it yourself. It is called automatically for local instances at the end of the block.

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

Member Function
It is a good practice to set as often as possible a member function as `const`. This prevents you from changing `this`. It is obvious that you can not call from a `const` member function a non-const member function.

```
/* Date.h */
class Date {
  bool isValidDate() const;
};


/* Date.cpp*/
bool Date::isValidDate() const { /**/ }
```

Static Member Function
Static Member Functions have some restrictions in CPP. In a static member function you can not access `this` (does not exits) and the function can not be `const` (again, `this` not exits). The `static` keyword is only allowed in the declaration (header file).

```
/* Date.h */
struct Date {
  static isValidDate(int year);
};

/* Date.cpp */
Date::isValidDate(int year) { /**/ }
```

Static Member Variable
Similar to the static Member functions the `static` keyword occurs only in the header file. If the variable is `const` the variable can be initialized directly in the header file. If not it must be initialized in the `cpp` file. But you should try to prevent non const static variables because they are a kind of global variables.

```
/* Date.h */
struct Date {
  static const int value{2};
  static int nonConst;
};

/* Date.cpp */
Date::nonConst = 4;
```

Overriding Operators
To override an operator in CPP you use the following syntax:

- `<returnype> operator op(<parameters>);`

```
// free ooperator
Date operator+(Date const &rhs, Date
    const &lhs) { /* */ }


struct Date {
  /**/

  Date operator+(Date const &lhs) { /*
      */ }
};
```

Non Overloadable Operators
In CPP the following operators are **not** overloadable:

- `::`
- `.*`
- `.`
- `?`

Free Operators
Free Operators are operator overloads outside of the class definition.

```
Date operator+(Date const &rhs, Date
    const &lhs) { /* */ }
```

In this function it is *not* possible to access the private / protected elements inside the class.

Friend functions

Functions with the keyword `friend` can only be declared inside a class declaration. But the the function is *not* part of the class itself. It's a *free* function which can also access the private member of a function.

This kind of function should only be used in exception cases. For example for implementing the stream operators.

```
// Switch.h
class Switch {
  int state;

public:
  Switch();
  auto pressButton() -> void;

  friend auto operator<<(std::ostream &
      out, Switch sw) -> std::ostream&;
};

// Switch.cpp
Switch::Switch() : state{0} { }

auto Switch::pressButton() -> void {
  ++current;
  this->state = current % 3;
}

auto operator<<(std::ostream &out,
    Switch sw) -> std::ostream& {
  std::array<std::string, 3> arr = { "
      off", "on", "blinking" };
  out << arr[sw.state];
  return out;
}
```

Member Operators

Member Operators are operator overloads inside of the class definition.

```
struct Date {
  /**/

  Date operator+(Date const &lhs) { /*
      */ }
};
```

In this function it is possible to access the private / protected elements inside the class.

Boost to implement compare operators

Normally you have to implement all compare operators by your self. Using the library Boost you only have to implement the less operator (`<`). But in exchange you have to inherit from `boost::less_than_comparable<T>` (private is enough).

```
#include "boost/operators.hpp"
#include <tuple>


class Date : private boost::
    less_than_comparable<Date> {
  int year, month, day;

public:
  bool operator<(Date const & rhs) const
      {
    return std::tie(year, month, day) <
        std::tie(rhs.year, rhs.month, rhs
        .day);
  }
};
```

Overloading Stream Operators

The syntax for use the output operator is the following:

```
std::cout << Date::myBirthday;
```

This requires that the `ostream` has an overload of the output operator with `Date` or the function is free. We can not write an overload for `ostream` so only free operators are left. But a free function can not access the private members of the class. The solution to this problem are `friend` functions. The same applies for the input operator and `istream`.

```
// Switch.h
class Switch {
  int state;

public:
  Switch();
  auto pressButton() -> void;

  // output
  friend auto operator<<(std::ostream &
      out, Switch sw) -> std::ostream&;
  friend auto operator>>(std::istream &
      in, Switch &sw) -> std::istream&;
};

// Switch.cpp
Switch::Switch() : state{0} { }

auto Switch::pressButton() -> void {
  ++current;
  this->state = current % 3;
}

auto operator<<(std::ostream &out,
    Switch sw) -> std::ostream& {
  std::array<std::string, 3> arr = { "
      off", "on", "blinking" };
  out << arr[sw.state];
  return out;
}

auto operator>>(std::istream &in, Switch
    &sw) -> std::istream& {
  std::array<State, 3> arr = { State::
      off, State::on, State::blinking };
  int i{0};
  in >> i;
  i = i % 3;

  if (in) {
```

```
    sw.state = arr[i];
  } else {
    throw std::invalid_argument{"Could␣
        not␣read␣a␣switch␣from␣input"};
  }
  return in;
}
```

Factory Functions
Place your factory function as a static member or in the same namespace. The task of this function is to create the desired object and provide a default value. This function are normally called:

- `make_xxx()`
- `create_xxx()`

```
Date make_date(std::istream & in) {
  try {
    return Date{in};
  } catch(std::out_of_range const &) {
    return Date{9999, 12, 31};
  }
}
```

6 Namespaces

Namespaces
Namespaces are used to group classes, functions and other elements and to prevent name clashes. The same name can occur multiple times in different namespaces. Namespaces can only defined outside classes and functions. But the Namespace can be opened and closed multiple times. Normally you open a namespace in the header file and reopen it in the cpp file again.
The global namespace in CPP has the prefix `::` and can be ommited if unique.

Import Namespace
The keyword `using` is used to import a name from a namespace into the current scope. Using type aliases you can shorten the name if its too long. Using directives which import **ALL** names of a namespace should only be used locally (for example in a function).

```
#include <string>

auto foo() -> void {
  using namespace std;

  string a{"Hallo"};
}
```

Anonymous Namespace
Anonymous Namespaces are a special case of namespaces. This namespaces are used to hide module internals like:

- helper function and types
- constants

The anonymous namespace can not be imported by you. The anonymous namespace is imported directly after the namespace is closed. So the following elements can use the elements defined in the anonymous namespace. You should anonymous namespaces only in the .cpp file.

Argument Dependent Lookup / ADL
When the compiler encounters an unqualified function or operator call with an argument of a user-defined type it tries to resolve it. To resolve it checks the namespace of the arguments to find the function.
E.g it is not necessary to write `std::` in front of `for_each` when `std::vector::begin()` is an argument.

```
// Date.h
namespace calendar {
  class Date {
    // ...
  };

  bool isHoliday(Date const &);
}

// Holidays.cpp
#include "Date.h"

using Dates = std::vector<calendar::Date
    >;

void markHolidays(Dates const & dates) {
  for_each(begin(dates), end(dates), [](
      calendar::Date const & d) {
    if (isHoliday(d)) {} // this is
        found because of ADL
  });
}
```

7 Enums

Enums
In CPP there exits two kinds of enums:

- unscoped enums
- scoped enums

```
enum unscopedEnum {};
enum class scopedEnum {};
```

For enums the operators als can be overloaded

```
DayOfWeek operator++(DayOfWeek &); //
    prefix
DayOfWeek operator++(DayOfWeek &, int);
    // postfix
```

The enumerator names are not mapped automatically to their name. You have to provide a lookup table to print out the name.

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

Scoped / Unscoped

Unscoped enums can be converted to `int` implicit. Scoped enums can only explicitly casted. For the conversation from `int` to enum is always the `static_cast` required.

```
enum DayOfWeek { Sun, /* ... */ };
enum class Month { Jan, /* ... */ };

int day = Sun;
int month = static_cast<int>(Month::Jan)
    ;
```

Unscoped enums are best used as a memeber of a class. If the enum is a independent type then the scoped version should be used.

Specifying the Underlying type for an enum

In CPP you can specify which size the elements of the enum should have. *Attention*: This is *not* a inheritance, you only specify the underlying type.

```
enum class LaunchPolicy : unsinged char
    { /**/ };
```

8 Templates

8.1 Function Templates

inline keyword

> Because the meaning of the keyword inline for functions came to mean "multiple definitions are permitted" rather than "inlining is preferred", that meaning was extended to variables.

What are Function Templates

If you want to implement the `min` method have to implement this for `int`, `float`, `double` and so on. This is error prone because you would write the same code for each data type you want to support. In Programming Languages like C# or Java you have Generics to solve this problem. In CPP we have Function Templates for compile-time polymorphism.
These are normally defined in a header file because the compiler needs to see the whole template to create an instance.

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

How do Function Templates work

The process how the compiler generates code from a Function Template.
The compiler...:

1. resolves the function template
2. figures out the template arguments
3. creates code with template parameters replaced
4. checks the types for correct usage

Type Checking in Templates

The type checking in templates happens twice:

1. when the template is defined: Only basic checks are performed (syntax, name resolution)
2. when the template is instantiated (used): the compiler checks whether the template arguments can be used as required by the template

Valid Template Argument

CPP use duck-typing to check if template argument is valid for this template. If the code inside the function can be executed using the given type then the template is valid. Otherwise, a compilation error is thrown.
Example:
What are the requirements of the type T in the `min` function template?

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

- T must be comparable with the `<` operator
- Copy / Move construcible, to return T by value

duck-typing

Duck-typing is concept where the type of variable is not describe using its class but using the existence of methods, properties, fields.

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Template Argument Deduction
The process of figuring out the correct template arguments from the call. The CPP compiler performs a pattern matching on the function parameter list for deducing the correct argument.
If ambiguities during the deduction occurs then a compiler error is throws.

Function Template Overloading
Using string literals in a normal template could be lead to wrong behavior. The `min` template below (**??**) is implemented correctly. But if you use it with string literals a comparison of the address is made. This is normally not what you want.

To prevent this behavior you could overwrite a template. Multiple function template with the same name can exist as long as they can be distinguished by their parameter list. It is also possible to write a *normal* function with the same name. The most concrete version is used in this case.

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

template <typename T>
T * min(T * left, T * right) {
  return *left < *right ? left : right;
}

char const * min(char const * left,
                 char const * right) {
  return std::string{left} < std::string
      {right} ? left : right;
}
```

8.1.1 Variadic Templates

Variadic Templates
In some cases the number of template parameters might not be fixed / known before. Thus, the template shall take an arbitrary number of parameters.

```
template<typename First, typename...
    Types>
void printAll(First const & first, Types
     const &...rest) {
  std::cout << first;
  if (sizeof...(Types)) {
    std::cout << ",␣";
  }
  printAll(rest...);
}
```

Pack Expansion
In a Variadic Template the last parameter is a function is called Function Parameter Pack. In the template declaration it is called Template Parameter Pack. This pack contains all remaining parameters.
Using `sizeof...(PackType)` the number of elements in the template parameter pack is returned.
To expand if `rest...` syntax is used. This expands to a comma seperated list of instances of the pattern (see example).
The pattern must contain at least one pack parameter. The base case for the recursion must exist **before** the templates is executed / generated. Otherwise an compilation error is thrown.

8.2 Class Templates

Class Template
Class Templates are very similar to function templates. Instead of function it is based on classes. Therefore, it enables us compile-time polymorphism for classes and structs.

```
template <typename T>
class Sack {
  using SackType = std::vector<T>;
  using size_type = typename SackType::
      size_type;
  SackType theSack{};
public:
  bool empty() const {
    return theSack.empty();
  }
  size_type size() const {
    return theSack.size();
  }
  void putInto(T const & item) {
    theSack.push_back(item);
  }
  T getOut(); // implementation
      somewhere else
};


// example for implementing member
    functions outside of a class
template <typename T>
inline T Sack<T>::getOut() {
  if (empty()) {
    throw std::logic_error{"Empty␣Sack"
        };
  }
  auto index = static_cast<size_type>(
      rand() % size());
  T retval{theSack.at(index)};
  theSack.erase(theSack.begin() + index)
      ;
  return retval;
}
```

Access to a type of the template parameter
Within a template definition sometimes you

want to use names witch are directly or indirectly depending on the template parameter. In the example below for example the `size_type` from `SackType`. In contrast to a variable or function name you have to tell the compiler that it is a type. This is done using the `typename` keyword.
In the example below the `size_type` of Sack and the `size_type` of `SackType` / `std::vector` are the same.

```
template <typename T>
class Sack {
  using SackType = std::vector<T>;
  using size_type = typename SackType::
      size_type;
  SackType theSack{};
}
```

static member variables in class templates

In a class template you can create a static variable. This does not violate the ODR. Since C++17 it is possible to declare the static variable inside the class template. However, this requires the `inline` keyword.

```
template <typename T>
struct staticmember {
  inline static int dummy{sizeof(T)};
};
```

Class Template Gotchas

When a class template inherits from another class template the name-lookup can be surprising! To prevent this behavior always use `this->` or `name::` to refer to an inherited member.

Class Template Specialization

Similar to the overloads in a function template you could specialize a class template. Thus, we provide different implementations based on the template parameter. It is possible to provide a partial specialization or an explicit specialization (a specialization for a concrete type).

How to prevent the compiler from creating specialiced class template

In our Sack example using class templates we do not want raw pointers (this could lead to dangling pointers). To prevent the compiler to create such an instance we could write the following code:

```
template <typename T>
struct Sack<T> {
  /* implementation */
};

/* */
template <typename T>
struct Sack<T *> {
  ~Sack() = delete;
};
```

Because the destructor is missing the compiler could not create such an instance (except you use `new` - DO NOT DO THIS).

How to inherit constructors in a class template

If you want to implement a class template which inherits from an other class template you may want to inherit its constructors. This is also done using the `using` keyword:

```
template<typename T>
struct safeVector : std::vector<T> {
  using container = std::vector<T>;
  using container::container; //or using
      std::vector<T>::vector; Inherit
      constructors
  using size_type = typename container::
      size_type;
  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);
  }
  // should also provide front/back with
       empty() check
};
```

Deduction Guides

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

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

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

template template parameters

A template can take templates as parameters. This parameter is called template template parameter.

```
// variadic template is required because
     many stl containers needs more than
    one element type
template<typename T, template<typename
    ...> typename Container>
class Sack;

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

Default for a template parameter

Sometimes you want to sepcify a default for a template parameter. This is done like in the following snippet.

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

non-type template parameters

In some cases you want not a type but a value as a parameter in the template. For example the `std::array` has one type and one non-type parameter. The second one is the size of the array.

If you want a flexible type you can also use `auto`.

```
template <typename T, std::size_t n>
auto average(std::array<T, n> const &
    values) {
  auto sumOfValues = accumulate(begin(
      values), end(values), 0);
  return sumOfValues / n;
}
```

9 Heap Memory Management

RAII - Resource Acquisition Is Initialization

Resource Acquisition Is Initialization (RAII) is a concept used in programming. The concept says:

- allocation of resources should happen in the constructor
- deallocation should happen in the destructor

The benefit of this pattern is that in every case the resources are allocated correctly and deallocated.

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

How you should manage heap memory

In modern CPP you should never allocate memory directly on the heap (`new` keyword). Instead, you should use libraries for this (where possible). Those libraries normally implement the Resource Acquisition Is Initialization idiom.
The CPP STL provides a few types for heap management (smart pointers).

Smart Pointers

- std::unique$_{ptr}$<T> obtained with std::make$_{unique}$<T>()
- std::shared$_{ptr}$<T> obtained with std::shared<T>()

unique$_{ptr}$

The unique$_{ptr}$ is a smart pointer used for unshared memory. Only a single owner exists. It is often used to wrap to-be-freed pointers from C functions when interfacing legacy code. `std::unique_ptr` can not be used for class hierarchies.

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

std::unique_ptr<int> create(int i) {
  return std::make_unique<int>(i);
}

int main() {
  std::cout << std::boolalpha;
  auto pi = create(42);
  std::cout << "*pi = " << *pi << '\n';
  std::cout << "pi.valid? " <<
      static_cast<bool>(pi) << '\n';
  auto pj = std::move(pi);
  std::cout << "*pj = " << *pj << '\n';
  std::cout << "pi.valid? " <<
      static_cast<bool>(pi) << '\n';
}
```

shared$_{ptr}$

The shared$_{ptr}$ is a smart pointer. The shared$_{ptr}$ works similar to Javas references. It can be copied and passed around. The last one ceasing to exist deletes the object. You can create a `std::shared_ptr` using `std::make_shared<T>()`
Attention: All owners of the shared$_{ptr}$ can changed the value of it.

```
struct Article {
  Article(std::string title, std::string
      content);
  //..
};
```

```
Article cppExam{"How to pass CPl?", "In
    order to pass the C++ exam, you have
    to..."};
std::shared_ptr<Article> abcPtr = std::
    make_shared<Article>("Alphabet", "
    ABCDEFGHIJKLMNOPQRSTUVXYZ");
```

The shared$_{ptr}$ is used when you want to create your own object networks. Another situation is when you need run-time polymorphic or class members that can not be passed as reference, e.g., because of lifetime issues.

weak$_{ptr}$

The weak$_{ptr}$ is similar to the shared$_{ptr}$. But a weak$_{ptr}$ does not increment the counter of the shared$_{ptr}$. Therefore, using weak$_{ptr}$ you can break cyclic dependencies.

The function `std::weak_ptr::lock()` returns a `std::shared_ptr` that either points to the alive pointer or is empty.

```
struct Person {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;
  void Person::acquireMoney() const {
    auto locked = parent.lock();
    if (locked) {
      begForMoney(*locked);
    } else {
      goToTheBank();
    }
  }
};
```

weak$_{ptr}$ from this

It would be nice if parents could spawn their own children. Therefore, we need a `std::weak_ptr` to the `this` object which is assigned to `child.parent`. This could be done by inheriting from `std::enable_shared_from_this<T>`.

10 Dynamic Polymorphism

mix-in

Mix-in is something like a class which provide additional functionality without adding more data. In CPP these arte often classes which requires your own class as template argument.

```
struct Date : boost::equality_comparable
    <Date> {
  //...
};
```

shadowing member functions

If you reimplemented a function in a derived class it shadows its counterpart in the base class. Even if the signature does not match (constness is part of the signature). However, if access through a declared base object, the shadowing is ignored and only the base class members are availabel (static dispatch). The static dispatch is the default in CPP.

If you want in your subclass both members (from sub and base class) you need to *import* using the keyword `using` (see figure **??**).

The output of in figure **??** below is *Hi, I'm Base.*

virtual member functions

To use dynamic polymorphism the function requires keyword `virtual`. The `virtual` keyword can be omitted in the derived class. Similar to Javas annotation `@Override` you can add the `override` keyword to the function.

For a successful override the function signature must be the same (also the constness).

Call Virtual Member Functions

In CPP if you want dynamic polymorphism you need a reference or pointers. If you only have a value object of the base class also the base function is called.

```
struct Base {
  virtual void sayHello() const;
};

struct Derived : Base {
  void sayHello() const;
};

void greet(std::unique_ptr<Base> base) {
  //calls sayHello() of the actual type
  base->sayHello();
}
```

```
struct Base {
  virtual void sayHello() const;
};

struct Derived : Base {
  void sayHello() const;
};

void greet(Base const * base) {
  //calls sayHello() of the actual type
  base->sayHello();
}
```

Pure Virtual Function

In CPP does not exist interfaces. Using pure virtual functions you make a class abstract. This can sometimes behave similar to an interface.

```
struct AbstractBase {
  virtual void doitnow() = 0;
};
```

Virtual Member Functions using smart pointers

If you have virtual members in your class then you need a virtual destructor if it is you used with a unique$_{ptr}$. Otherwise, only the destructor of the base is called.

The shared$_{ptr}$ memorize the actual type and know which destructor to call. Thus, the destructor is not required to be virtual.

Object Slicing

When in CPP a function takes a base class as value parameter, but you provide a subclass then *object slicing* occurs. That means, that the base is extracted / sliced from the subclass and passed to the function. Therefore, functions which were overwritten are not available (see figure **??**).

Somthing similar happens if you assign a subclass to a base class reference. The creation of the reference is OK. But the second assignment copies only the base class part into the base reference (see figure **??**). To prevent this, you can declare the copy-operations as deleted

```
struct Book {
  //...
  Book & operator=(Book const & other) =
      delete;
  Book(Book const & other) = delete;
};

// does not compile anymore because copy
    -operations are deleted
EBook designPatterns{writeEbook(395)};
EBook refactoring{writeEbook(430)};
refactoring.openPage(400);
Book & some = refactoring;
some = designPatterns;
readPage(some.currentPage());
```

Guidelines

- You should only apply inheritance and virtual member functions if you know what you do
- Do not (like the IDE) create classes with virtual members by default
- mark overwritten functions in sub classes with override
- If you design base classes with polymorphic behavior, understand the common abstraction that they represent
  - Extract from existing class(es) the base after you see the commonality arise
- Follow the Liskov Substitution Principle
- Do not break invariants of the base class
- Beware of unwanted member hiding
- Avoid object slicing
- Mark Destructors virtual if you have any other virtual member function

11 Initialization and Aggregates

Kinds of initialization

CPP has many of initialization.

- Default Initialization
- Value Initialization
- Direct Initialization
- Copy Initialization
- List Initialization
- Aggregate Initialization

Which kind is used is based on the context.

Default Initialization

The Default Initialization is the simplest one in CPP. You simply don't provide an initializer. The result depends on the kinds of entity we declare. Default Initialization could be dangerous because it sometimes behaves unexpected.

```
int global_variable; // implicitly
    static
void di_function() {
  static long local_static;
  long local_variable;
}
struct di_class {
  di_class() = default;
  char member_variable; // not in ctor
      init list
};
```

Static variables are zero initialized first and then their default constructor is called. If the type can not be default constructed the program does not compile. Non-static integral and floating point variables are uninitialized. Objects of class types are constructed using their default constructor. Member variables not in a ctor-init-list are default initialized.

Value Initialization

The Value Initialization is performed with empty `()` or `{}`. However, you should use the `{}` since it works in more cases (see example below). The Value Initialization just calls the default constructor for class types.

```
#include <string>
#include <vector>
void vi_function() {
  int number { };
  std::vector<int> data { };
  std::string actually_a_function(); //
      you define here a new function!!!
}
```

Direct Initialization

The Direct Initialization is one type of initialization in CPP. The Direct Initialization works similar to the value initialization. It uses non-empty `()` or `{}`. When you use `{}` this only applies if you want to initialize not a class type. Still you should prefer the `{}` syntax.

In the example below the last line is a function declaration of a function returning a word and taking an unnamed pointer to a function returning a string.

```
#include <string>
void diri_function() {
  int number{32};
  std::string text("CPl");
  word vexing (std::string());
```

```
}
```

Copy Initialization

To invoke the copy initialization the `=` is used.

- If the object has class type and the right-hand side has the same type:
  - if the right-hand side is temporary, the object is constructed "in-place"
  - otherwise, the copy constructor is invoked
- Otherwise, a suitable conversion is searched for.

This applies also to return / throw / catch statements.

```
#include <string>
std::string string_factory() { return ""
    ; }
void ci_function() {
  std::string in_place = string_factory
      ();
  std::string copy = in_place;
  std::string converted = "CPl";
}
```

List Initialization

The list initialization uses the non-empty `{}`. Further there exists two kinds of the list initialization:

- direct list initialization
- copy list initialization

The constructor is selected in two phases:

1. if there is a suitable constructor taking `std::initializer_list`, it is selected
2. otherwise, a suitable constructor is searched., it is selected

Attention: Since `std::initializer_list` is preferred, you might run into trouble.

```
std::string direct { "CPl" };
std::string copy = { "CPlA" };
std::vector<int> data {10, 42}; //
    creates [10, 42] and not 42x 10
std::vecotr<int> data2(10, 42); //
    create 42x 10
return data[5]; // Undefined Behaviour
```

Aggregate Initialization

The Aggregate Initialization is a special case of the list initialization. If the type is an aggregate then the members and base classes are initialized from the list. If you provide fewer initializers than there are bases and members the *uninitialized* members used the list from the declaration or the empty list.

```
struct person {
  std::string name;
  int age{42};
  bool operator<(person const & other)
      const {
    return age < other.age;
  }
  void write(std::ostream & out) const {
    out << name << ": " << age << '\n';
  }
};

int main() {
  person rudolf{"Rudolf", 32};
  rudolf.write(std::cout); // Rudolf: 32

  person hans{"Hans"};
  hans.write(std::cout); // Hans: 42
}
```

Aggregates in CPP

Aggregates are simple class types which are often used for DTOs. They can have other types as public base classes. They can provide member variables and functions. But they don't have a user-provide, inherited or explicit constructors. They also don't have any protected or private direct fields.

```
struct person {
  std::string name;
  int age{42};
  bool operator<(person const & other)
      const {
    return age < other.age;
  }
  void write(std::ostream & out) const {
    out << name << ": " << age << '\n';
  }
};
int main() {
  person rudolf{"Rudolf", 32};
  rudolf.write(std::cout);
}
```

What is a DTO

DTO is short for Data Transfer Object. That is a class which is only used to group information to transfer it from one point to another. Therefore, they normally don't have any invariant.

12 CUTE

CUTE

CUTE (C++ Unit Testing Easier) is a Unit Testing Framework for C++.

Exception In CUTE you can assert an thrown with `ASSERT_THROWS` or with a `try-FAILM()-catch()` construct:

```
void testForExceptionTryCatch() {
  std::vector<int> empty_vector{};
  try {
    empty_vector.at(1);
    FAILM("expected Exception");
```

```
} catch (std::out_of_range const &) {
  // expected
  }
}
```