

# C++ Spick

---

## ##TODO

- Demo verstehen (V12, S. 11)
- V12, S.21
- Templates-Erweiterungen (V12)
- Ausgaben-Formatierung (Oct, dec usw.)
- Beispiele
  - Ring5 (V6, S31, auch Übung)
- const Beschreiben, unterschied const Methoden und const Variablen
- Initialisierung mit () vs {}
- Cute
- Klassen
  - Interface
  - Enum
    - Scoped
    - unscoped
- Beispiel für Anonymous Namespace?
- Call by value / call by reference
- Files lesen / schreiben
- Dekonstruktor ist nicht virtual
- Testat indexable einfügen

## Table of Contents *generated with [DocToc](#)*

- [Variablen](#)
- [Typen](#)
  - [Literele](#)
- [Operatoren](#)

- Reihenfolge
- Funktionen
  - Scopes Summary
  - Parameter Passing - Return Values
  - Function Overloading
  - Default Arguments
  - Funktionen als Parameter
- Include Files
  - Include Guard
    - Beispiel mit 3 Files
  - Wichtige includes
- Const/non-const und Value/Reference
- Kommandozeilenargumente übergeben
- Memory (Heap)
  - Beispiel: Eltern und Kinder
- Move
- Namespaces
- CUTE TODO ist das nötig?
- Using
- Streams
  - Beispiel: Date read() implementieren
- Iterators
  - Spezielle Iteratoren für I/O
  - Kategorien
  - Spezialfunktionen
  - `std::istream_iterator`
  - `std::istreambuf_iterator`
  - `std::ostream_iterator`
  - `std::ostreambuf_iterator`
  - `std::reverse_iterator`
- Containers

- `std::vector`
- `std::array`
- `std::deque`
- `std::list`
- `std::forward_list`
- `std::stack`
- `std::queue`
- `std::priority_queue`
- `std::set`
- `std::multiset`
- `std::map`
- `std::multimap`
- `std::unordered_set`
- `std::unordered_map`
- Nackte Arrays
  - Initialisieren
  - Länge erkennen mit Type Deduction
- Lambdas
  - Captures und Parameter
    - Spezialfall mutable
- Functor
  - Als Parameter
- Algorithms
  - Ranges
  - Beispiele aus Vorlesung
  - Suffix-Versionen
  - Heap-Algorithmen
  - Fallen
  - Tabelle
- Klassen
  - Beispielklasse Date

- Access Specifier
- Member Variables
- Static Member-Variablen
- Konstruktor
  - Spezielle Konstruktoren
  - Implementation
  - Konstruktor mit `std::istream &`
  - Konstruktoren wieder default machen/löschen
- Destruktoren
- Vererbung
- Implementation
- Benutzung
- Member-Funktionen
  - Static Member-Funktionen
- Operator-Overloading
  - Beispiel: Date vergleichbar machen
  - Beispiel: Date an `std::cout` senden
- Vererbung
  - Mehrfachvererbung
  - Initialisierung
  - Sichtbarkeit
  - Object Slicing
  - Probleme mit Vererbung und pass-by-value
  - Virtual
  - Abstrakte Klassen
- Argument Dependent Lookup (ADL)
- Enums
  - Wert festlegen
  - Typ festlegen
- Contract/Exceptions
  - Exceptions

- Exceptions abfragen mit CUTE
- Templates
  - Function Templates
  - Concepts
  - Overloads
  - Variadic Template Function
  - Class Templates
    - Erben in Template Classes
    - Spezialisierung
    - Template Terminologie
    - Sack mit Initializer List füllen
    - Container variieren
    - Templates als Adapter

# Variablen

---

```
<type> <name> {<value>;
```

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

Initialisierung kann weggelassen werden, wird aber nicht empfohlen.  
Leere Klammern bedeuten Default-Initialisierung Mit `=` können wir den Compiler den Typ entscheiden lassen (nicht mit geschw. Klammern kombinieren)

```
auto const i = 5
```

Mit `const` **muss** initialisiert werden (mit geschweiften Klammern). Mit

`constexpr` wird der Wert zur Compile-Zeit festgelegt. Nicht vergessen:

## As const as possible

```
int const theAnswer{6*7}  
double constexpr pi{3.14}
```

C++ definiert den Begriff des lvalue und rvalue. Man darf beispielsweise nur lvalues inkrementieren

```
x = 6 * 7  
x // lvalue  
6 * 7 // rvalue  
x++ // ok  
5++ // nicht ok
```

Liste des Bösen:

- eine Variable *darf* innerhalb eines Blocks neu verwendet werden, dies ist kein Fehler
- globale Variablen

# Typen

---

Eingebaute Typen (ohne include)

- bool
- char, unsigned char, *wchar\_t*, *char16\_t*, *char32\_t*
- short, int, long, long long
- unsigned short, unsigned, unsigned long, unsigned long long
- float, double long double
- weitere

# bool

---

In Bedingungen können alle Werte stehen, die zu bool oder Zahlen konvertiert werden können. Dabei gilt der Wert 0 als false.

## Literale

---

- U/L für Integer (unsigned/long), Gross-/Kleinschreibung egal
- Exponenten mit E für float/double
- "ab"s macht einen String aus "ab", benötigt `using namespace std::literals`

```
void sayHello(){  
    using namespace std::literals;  
    out << "hello"s;  
}
```

## Operatoren

---

- Wie aus Java bekannt
- `and, or, not` sind alternative Schreibweisen für `&&, ||, !`
- `bitand, bitor, xor` sind `&, |, ^`

## Reihenfolge

---

Achtung: in einer einzelnen Expression, wenn die Funktionsaufrufe nur durch Komma getrennt sind, ist die Reihenfolge undefiniert.

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|--|--|--|--|

| Precedence | Operator           | Description                             | Associativity |
|------------|--------------------|---|---------------|
| 1          | ::                 | Scope resolution                        | Left-to-right |
| 2          | a++ a--<br>-       | Suffix/postfix increment and decrement  |               |
|            | type()<br>type{}   | Functional cast                         |               |
|            | a()                | Function call                           |               |
|            | a[]                | Subscript                               |               |
|            | . ->               | Member access                           |               |
| 3          | ++a --a<br>a       | Prefix increment and decrement          | Right-to-left |
|            | +a -a              | Unary plus and minus                    |               |
|            | ! ~                | Logical NOT and bitwise NOT             |               |
|            | ( type )           | C-style cast                            |               |
|            | a                  | Indirection (dereference)               |               |
|            | &a                 | Address-of                              |               |
|            | sizeof             | Size-of <sup>[note 1]</sup>             |               |
|            | new<br>new[]       | Dynamic memory allocation               |               |
|            | delete<br>delete[] | Dynamic memory deallocation             |               |
| 4          | . ->               | Pointer-to-member                       | Left-to-right |
| 5          | ab<br>a/b<br>a%b   | Multiplication, division, and remainder |               |



|    |   |  |               |
|----|---|--|---------------|
| 6  | <code>a+b</code><br><code>a-b</code>        | Addition and subtraction   |               |
| 7  | <code>&lt;&lt;</code> <code>&gt;&gt;</code> | Bitwise left shift and right shift   |               |
| 8  | <code>&lt;</code> <code>&lt;=</code>        | For relational operators <code>&lt;</code> and <code>≤</code> respectively |               |
|    | <code>&gt;</code> <code>&gt;=</code>        | For relational operators <code>&gt;</code> and <code>≥</code> respectively |               |
| 9  | <code>==</code> <code>!=</code>             | For relational operators <code>=</code> and <code>≠</code> respectively    |               |
| 10 | <code>a&amp;b</code>                        | Bitwise AND  |               |
| 11 | <code>^</code>                              | Bitwise XOR (exclusive or)   |               |
| 12 | <code> </code>                              | Bitwise OR (inclusive or)  |               |
| 13 | <code>&amp;&amp;</code>                     | Logical AND  |               |
| 14 | <code>  </code>                             | Logical OR   |               |
| 15 | <code>a?b:c</code>                          | Ternary conditional <sup>[note 2]</sup>                                    | Right-to-left |
|    | <code>throw</code>                          | throw operator   |               |
|    | <code>=</code>                              | Direct assignment (provided by default for C++ classes)                    |               |
|    | <code>+=</code> <code>-=</code>             | Compound assignment by sum and difference                                  |               |
|    |   | Compound   |               |

|    |   |   |               |
|----|---|---|---------------|
|    | <code>*=</code><br><code>/=</code><br><code>%=</code>     | assignment by product, quotient, and remainder            |               |
|    | <code>&lt;&lt;=</code><br><code>&gt;&gt;=</code>          | Compound assignment by bitwise left shift and right shift |               |
|    | <code>&amp;=</code><br><code> =</code><br><code>^=</code> | Compound assignment by bitwise AND, XOR, and OR           |               |
| 16 | <code>,</code>  | Comma   | Left-to-right |

# Funktionen

Wenn der Block Scope endet `}` ist die Lebenszeit vorbei

`}` collects garbage

Achtung: in Funktionen können Variablen Shadowing machen, dies ist nicht verboten (wie in Java)

## Scopes Summary

- Global Scope ::
  - named Namespaces ::name:: (may be nested)
    - anonymous namespace (hides name from linker)
      - class scope (members)
        - function scope (parameters)
          - block scope (local variables)
            - temporaries (subexpression results)

# Parameter Passing - Return Values

- Pass by value: `f(type par)` (bevorzugt)
- Pass by const ref: `f(type const & par)` (bevorzugt wenn nichts verändert wird)
- Pass by reference: `f(type & par)`
- Return by value: `type f()`
- return by reference: `type & f(); type const &g`

**Achtung:** return by reference nur wenn die Referenzen als Parameter bereits herein gekommen sind, sonst führt es zu dangling references.

**Aufpassen mit Referenzen:** Wenn Parameter als Referenzen reinkommen, haben Änderungen darauf natürlich auch Einfluss auf die Originalvariable. Ebenso **NIE** eine lokale Variable als Referenz zurückgeben. Beim Abräumen des Stack geht diese flöten und die HSR brennt ab. Es sind **einzig** die eigenen Parameter wieder als Referenz zurückzugeben.

|           | value  | reference  |
|-----------|--|--|
| non-const | <code>Word(std::string value)</code> <ul style="list-style-type: none"><li>- Argument wird kopiert</li><li>- Änderungen im Wert haben keine Auswirkungen auf Aufrufer-Seite</li><li>- Für primitive und kleine Typen</li></ul> | <code>Word(std::string &amp; value)</code> <ul style="list-style-type: none"><li>- Argument wird aus dem Speicher as-is benutzt</li><li>- Änderungen im Wert haben Auswirkungen auf Aufrufer-Seite</li><li>- Benutzt wenn die Seiteneffekte gewünscht sind</li></ul> |
|           | <code>Word(std::string const value)</code>   | <code>Word(std::string const &amp; value)</code>   |

|       |   |  |
|-------|---|--|
| const | <ul style="list-style-type: none"><li>- Argument wird kopiert</li><li>- Wert kann nicht verändert werden</li><li>- Für primitive und kleine Typen</li></ul> | <ul style="list-style-type: none"><li>- Argument wird aus dem Speicher as-is benutzt</li><li>- Wert kann nicht verändert werden</li><li>- Kann für grössere Objekte verändert werden</li></ul> |
|-------|---|--|

## Function Overloading

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

Nur wenn die Parameter-Typen unterschiedlich sind oder eine andere Anzahl haben, nicht der Rückgabewert. Overload wird zur Compile-Zeit entschieden.

## Default Arguments

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

Implizites Overloading. Wenn es n default Argumente gibt, gibt es n+1 Versionen der Funktion.

## Funktionen als Parameter

Funktionen sind First-Class-Parameter in C++

```
double specific(double y) {  
    return y;  
}
```

```
void printfunc(double x, double f(double)) {  
    std::cout << f(x);  
}  
  
int main() {  
    printfunc(1, specific);  
}
```

In C nur als Function Pointer möglich

## Aufruf Sequenz von Funktionen

---

```
void main() {  
    askForName(std::cout);  
    sayGreeting(std::cout, inputName(std::cin), inputName(st  
}
```

Es ist nicht klar welche Funktion inputName zuerst aufgerufen wird.

## Include Files

---

Zu Beachten:

- Eigene Includes immer zuoberst!
- Funktionen in Header-Files typischerweise nur als Deklaration.
- Wenn kurze Funktion (oder Template): `inline`
- Klassen-Member-Funktionen sowie Templates sind implizit inline.
- includes mit Anführungszeichen für includes aus dem gleichen Verzeichnis

- includes mit spitzen Klammer für includes aus Libraries und anderen Verzeichnissen

## Include Guard

---

Um die ODR (One Definition Rule) nicht zu verletzen, verwendet man Guard Statements. Include Guards werden vom Preprocessor bearbeitet. Dies bedeutet, dass man für Klassen/Methoden in verschiedenen Namespaces und Files, auch verschiedene Include Guards verwenden sollte (Faustregel: Ein disjunkter Include Guard pro Header).

```
#ifndef SAYHELLO_H_  
#define SAYHELLO_H_  
#include <iosfwd>  
void sayHello(std::ostream &out);  
#endif /* SAYHELLO_H_ */
```

## Beispiel mit 3 Files

Hello.h

```
#ifndef HELLO_H_  
#define HELLO_H_  
#include <iosfwd>  
  
struct Hello {  
    void sayHello(std::ostream &out) const;  
};  
#endif /* HELLO_H_ */
```

Hello.cpp

```
#include "Hello.h"
#include <ostream>

void Hello::sayHello(std::ostream &out) const {
    out << "Hello world!\n";
}
```

main.cpp

```
#include "Hello.h"
#include <iostream>

int main() {
    Hello hello{};
    hello.sayHello(std::cout);
}
```

## Wichtige includes

```
#include <algorithm> // für alle Algorithms ausser die in <memory>
#include <functional>
#include <iterator>
#include <numeric>
#include <stdexcept> // exception
#include <string> // std::string
// IO
#include <iosfwd> // Vorwärtsdeklaration von istream und ostream
#include <istream> // istream Definition und Implementation
#include <ostream> // ostream Definition und Implementation
#include <iostream> // istream, ostream und cin, cout
#include <sstream> // string stream
// Pointers
#include <memory> // pointers
```

# Kommandozeilenargumente übergeben

---

Main kennt folgende zwei Definitionen:

- `int main()`
- `int main(int argc, char * argv[])`
  - `argc` hat die Anzahl Argumente drin.
  - `argv[]` ist ein Array von Char-Pointern (ein Array von "Strings"), also die einzelnen Argumente.
    - Die eigentlichen Argumente beginnen erst bei `argv+1`, im ersten Element steht der Programmname. Es endet bei `argv+argc`.

## Memory (Heap)

---

Man könnte das Memory selber mit `new` allozieren. Das ist aber böse.

Ebenso sollte man nie plain-Pointers verwenden.

`std::unique_ptr<T>` und `std::make_unique`

Für unshared Heap-Memory, oder für lokales was auf den Heap muss (z.B. Stack zu klein).

```
std::unique_ptr<int> aFactory(int i) {  
    return std::make_unique<int>(i);  
}  
  
auto answer = aFactory(42);
```



```
// Transfer of Ownership
auto answer2=std::move(answer)
```

Die Pointer haben immer nur einen Owner und können nicht kopiert werden (nur by value zurückgegeben werden).

Wenn man C-Pointer (z.B. von gewissen Funktionen) als `unique_ptr` verpackt, werden sie beim `}` automatisch ge-free-d.

Weil immer nur einem der Pointer gehört, wird irgendwann garantiert das Memory aufgeräumt.

**`std::shared_ptr<T>` und `std::make_shared<T>`**

Funktionieren mehr wie die Java-Referenzen. Können kopiert und umhergereicht werden, der letzte löscht das Licht aus und löscht das allozierte Objekt. `make_shared<T>` sorgt dafür, dass die public Konstruktorparameter von T benutzt werden können

Wenn etwas wirklich auf den Heap muss, Factory verwenden

```
#include <memory> // oder <boost/shared_ptr.hpp>
#include <string>
struct A {
    A(int a, std::string b, char c){}
};

std::shared_ptr<A> A_factory() {
    return std::make_shared<A>(5, "hi", 'a');
}

// Usage
int main() {
    auto an_a=A_factory();
    auto b = an_a; // second pointer to same object
    A c{*b} // copy constructor
```

```
    auto another = std::make_shared<A>(c); // copy-construction
}
```

Wenn Instanzen einer Klassenhierarchie durch `shared_ptr<base>` repräsentiert werden, aber durch `make_shared<concrete>()` erstellt werden, muss der Destruktor nicht mehr virtual sein. `shared_ptr` mekrt sich den konkreten Destruktor.

Man kann in ein zirkuläres Dependency-Problem rennen. Um das zu umgehen, braucht man `weak_ptr` um diese zu brechen.

```
enable_shared_from_this<T> && shared_from_this()
```

Problem: Will ein Objekt von sich selbst ein `shared_ptr` erstellen, muss die Klasse von `enable_shared_from_this<T>` erben.

```
#include <memory>

struct Person : public std::enable_shared_from_this<Person> {
    std::shared_ptr<Person> getMe() { return shared_from_this(); }
};
```

## Beispiel: Eltern und Kinder

`shared_ptr` Zyklen: `weak_ptr/shared_from_this`

Eine Personenklasse soll erstellt werden:

- Jede Person kennt ihre Eltern (Vater, Mutter) wenn noch am Leben
- Jede Person kann verheiratet werden
- Jede Person kennt ihre Kinder

- Mutter und Vater kennen beide ihre Kinder

Damit das sauber funktioniert:

- Alle lebenden Objekte in einer separaten Datenstruktur als `shared_ptr`
- Abhängigkeiten durch `weak_ptr`

Wenn man sie nun aus der Datenstruktur der lebenden Objekte entfernt, werden die Objekte zerstört. Der `weak_ptr` hat zwar eine Referenz, muss aber konkretisiert werden (zu einem `shared_ptr`) und bei diesem Schritt kann erkannt werden dass das eigentliche Objekt weg ist.

==> [Biblische Familie](#)

# Namespaces

Es gibt den globalen Namespaces, `::`, zum Beispiel `::read`. Sub-Namespaces sind erlaubt.

Ein Namespace kann mehrere Male geöffnet und geschlossen werden. Mit `using` kann man Namen von anderen Namespaces in den eigenen Scope importieren

Beispiel

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

```

    void bar() {
        foo(); // 1
        subdemo::foo(); // 2
    }
}
void demo::foo() {//1} // definition
int main() {
    using demo::subdemo::foo;
    foo() // 2
    demo::foo() // 1
    demo::bar()
}

```

Dazu gibt es noch den anonymen Namespace, wenn man den Namen weglässt. Damit kann man Sachen ausserhalb des Files verstecken.

```

//File 1
namespace {
    void doit() {
        //do something
    }
}

void print() {
    doit();
}
//File 2
void caller() {
    print();
    //doit(); linker error
}

```

# CUTE TODO ist das nötig?

---

CUTE kennt viele Makros und man sollte sie auch nutzen. Will man zum Beispiel die relationalen Operatoren prüfen, dann nicht einfach `ASSERT_EQUAL(true, a < b)` sondern `ASSERT_LESS(a, b)` .

# Using

- "Alias" mit `using input=std::istream_iterator<std::string>`
- Parent-Members in Namespace übernehmen, z.B. Konstruktoren mit `using std::set<T, COMPARE>::set;`
- Funktionen eines Namespaces in den scope übernehmen z.B. `using namespace std`
- Klasse eines Namespaces in den Scope übernehmen z.B. `using std::string; string s{"abc"}`

# Streams

Streams haben einen Status, der anzeigt ob I/O erfolgreich war oder nicht

- Nur `.good()` Streams können noch I/O
- Nach einem Fehler (`.fail()`) muss man den Zustand mit `.clear()` wieder löschen, die ungültigen Eingaben rausholen (`.ignore()`)und weiterfahren

istream Zustände:

| bit     | query                  | entered                |
|---------|------------------------|------------------------|
| failbit | <code>is.fail()</code> | formatted input failed |
| eofbit  | <code>is.eof</code>    | end of input reached   |

|        |        |                         |
|--------|--------|-------------------------|
| badbit | is.bad | unrecoverable I/O error |
|--------|--------|-------------------------|

| ios_base::iostate<br>flags |         |        | basic_ios accessors |        |       |       |               |
|----------------------------|---------|--------|---------------------|--------|-------|-------|---------------|
| eofbit                     | failbit | badbit | good()              | fail() | bad() | eof() | operator bool |
| false                      | false   | false  | true                | false  | false | false | true          |
| false                      | false   | true   | false               | true   | true  | false | false         |
| false                      | true    | false  | false               | true   | false | false | false         |
| false                      | true    | true   | false               | true   | true  | false | false         |
| true                       | false   | false  | false               | false  | false | true  | true          |
| true                       | false   | true   | false               | true   | true  | true  | false         |
| true                       | true    | false  | false               | true   | false | true  | false         |
| true                       | true    | true   | false               | true   | true  | true  | false         |

Beispiel: robustes Einlesen eines int, mit istream als Zwischenstream

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

# Beispiel: Date read() implementieren

**.read()** implementieren Precondition: std::istream ist im .good()-State. Wenn wir kein Datum extrahieren können, setzen wir std::istream in den fail-State.

Wenn der Input nicht verwendet werden kann, wird das Objekt nicht überschrieben.

## Header:

```
class Date {
    int year, month, day;
public:
    std::istream & read(std::istream & is) {
        int year{-1}, month{-1}, day{-1};
        char sep1, sep2;
        //read values
        is >> year >> sep1 >> month >> sep2 >> day;
        try {
            Date input{year, month, day};
            //overwrite content of this object (copy-ctor)
            (*this) = input;
            //clear stream if read was ok
            is.clear();
        } catch (std::out_of_range & e) {
            //set failbit
            is.setstate(std::ios::failbit | is.rdstate());
        }
        return is;
    }
};

inline std::istream & operator>> std::istream & is, Date & d {
    return d.read(is);
}
```

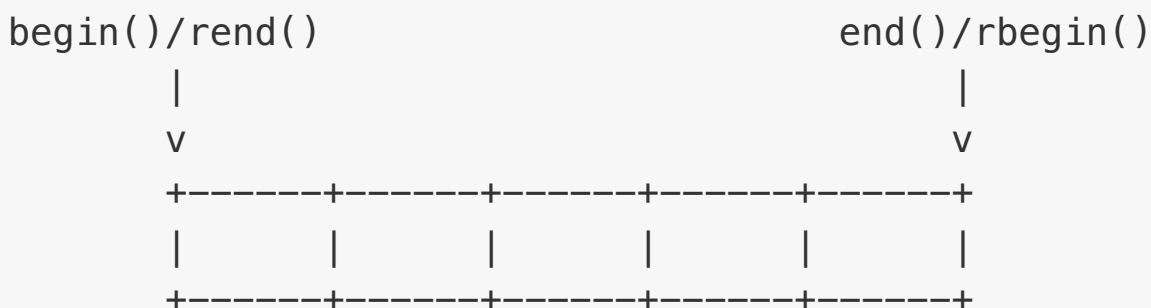
# Iterators

Include für alle Iterators: `#include <iterator>`

- Jeder Container bietet Iteratoren
- Es gibt immer ein Paar von Iteratoren, `begin(v)` und `end(v)`
- Es gibt die "allgemeine" Version wie oben, oder die spezialisierte Version `v.begin()/v.end()`. Im Zweifelsfall die spezialisierte Version verwenden.
- C++-Iteratoren kennen das Ende nicht. Man kann aber gegen das Ende vergleichen `iterator != v.end()`
- Auf Elemente mittels `*` zugreifen `*iterator`
- Nächster Schritt des Iterators: `++iterator`

Achtung, das Ende ist **vor** `end`.

Um read-only zu garantieren sollte `cbegin()/cend()` verwendet werden. Wenn man von "hinten" beginnen möchte, gibt's `rbegin()/rend()`.



Wenn man Iteratoren speichern will, am besten Typ `auto`.

Beispiele:



```

void testRIterators() {
    std::vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto it = v.rbegin();
    it++;
    ASSERT_EQUAL(*it, 9);
}

int main() {
    std::vector<int> v {1,2,3,4,5,6,7,8,9};
    std::cout << *(--v.end()) << ", " << *(--v.rend());
    // Output 9, 1
}

int main() {
    std::vector<int> v {1,2,3,4,5,6,7,8,9};
    std::cout << *v.begin() << ", " << *(--v.end())
               << ", " << *v.rbegin() << ", " << *(--v.rend());
    // Output 1, 9, 9, 1
}

```

Liste des Bösen

- Eigene Loops

## Spezielle Iteratoren für I/O

**`std::ostream_iterator<T>`** gibt Werte vom Typ **T** an den gegebenen **`std::ostream`** aus Endet wenn Input-Range fertig ist  
`copy(begin(v), end(v), std::ostream_iterator<int>{std::cout, ", "});`

**`std::istream_iterator<T>`** liest Werte vom Typ **T** vom gegebenen **`std::istream`** Endet wenn der istream nicht länger **good** ist

Diese nutzen aber beide intern den **`operator>>`** für Input. Der

überspringt Leerzeichen und White Spaces. Für eine perfekte Kopie bräuchten wir auch den Rest. Dies geht mit

```
istreambuf_iterator<char> .
```

## Beispiele Kopieren mit istream\_iterator

```
#include <iterator>
#include <iostream>
#include <algorithm>
#include <string>
int main() {
    using input=std::istream_iterator<std::string>;
    input eof{};
    input in{std::cin};
    std::ostream_iterator<std::string> out{std::cout, " "};
    copy(in, eof, out)
}
```

## Kopieren mit istreambuf\_iterator

```
#include <iterator>
#include <iostream>
#include <algorithm>
int main() {
    using input=std::istreambuf_iterator<char>;
    input eof{};
    input in{std::cin};
    std::ostream_iterator<char> out{std::cout};
    copy(in, eof, out)
}
```

# Kategorien

---

Grund für verschiedene Kategorien: Verschiedene Algorithmen brauchen spezielle Iteratoren z.B. random access

## Input Iteratoren

- Das aktuelle Element kann mehrmals ausgelesen werden
- Kann Iteratoren vergleichen
- Ein Zugriff der Art `*it++` führt dazu, dass alle bisherigen Kopien von `it` obsoletiert werden (siehe auch [C++Reference - InputIterator](#)).
- single-pass

## Forward Iterator

- Das Element kann gelesen und verändert werden (außer der Container oder die Elemente sind `const`)
- Kann nicht rückwärts lesen
- Der Iterator kann aber kopiert werden für spätere Referenz
- multi-pass

## Bidirectional Iterator

- Das Element kann gelesen und verändert werden (außer...)
- Kann vorwärts und rückwärts gehen
- Random Access sind bidirectional, aber können auch indexen
- multi-pass

## Output Iterator

- Kann einen Wert schreiben, aber nur einmal und muss danach inkrementiert werden
- single-pass

# Spezialfunktionen

Mit `distance` kann man die Anzahl Hops zählen, bis man zum anderen Iterator kommt. Mit `advance` kann man n Male hüpfen.

| category       |         | properties  |
|----------------|---------|---|
| all categories |         | <i>copy-constructible</i> ,<br><i>copy-assignable</i><br>and <i>destructible</i>  |
|                |         | Can be incremented  |
| Bidirectional  | Input   | Supports equality/inequality comparisons  |
|                |         | Can be dereferenced as an <i>rvalue</i>   |
|                | Output  | Can be dereferenced as an <i>lvalue</i> (only for <i>mutable iterator types</i> ) |
|                |         |   |
|                | Forward | <i>default-constructible</i>  |
|                |         | Multi-pass:<br>neither dereferencing nor incrementing affects dereferenceability  |
| Random Access  |         | Can be  |

|  |   |
|--|---|
|  | decremented   |
|  | Supports arithmetic operators + and -                               |
|  | Supports inequality comparisons (<, >, <= and >=) between iterators |
|  | Supports compound assignment operations += and -=                   |
|  | Supports offset dereference operator ([])                           |

Nicht möglich mit const Iterator

## std::istream\_iterator

Iterator für istream. Benutzen für formatierte Eingaben, z.B. doubles, ints.

Beispiel:

```
int main () {
    double value1, value2;
    std::cout << "Please, insert two values: ";
    std::istream_iterator<double> eos;
    std::istream_iterator<double> iit (std::cin);
    if (iit != eos) value1=*iit;
    ++iit;
    if (iit != eos) value2=*iit;
    std::cout << value1 << "*" << value2 << "=" << (value1*val
```

```
}
```

## std::istreambuf\_iterator

Iterator für istream. Benutzen für unformatierte Eingaben, z.B. char für char. Beispiel:

```
int main () {
    std::istreambuf_iterator<char> eos{};
    std::istreambuf_iterator<char> iit (std::cin.rdbuf());
    std::string mystring{};
    std::cout << "Please, enter your name: ";
    while (iit!=eos && *iit!='\n') mystring+=*iit++;
    std::cout << "Your name is " << mystring << ".\n";
}
```

TODO: Syntax-Highlighting stolpert über zweitletzte Zeile. Fix mit \*\*

## std::ostream\_iterator

Iterator für ostream. Formatierte Ausgabe. Beispiel:

```
int main () {
    std::vector<int> myvector{0,1,2,3,4,5,6,7,8,9};
    std::ostream_iterator<int> out_it (std::cout, ", ");
    std::copy ( myvector.begin(), myvector.end(), out_it );
}
```

Output: "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

# std::ostreambuf\_iterator

---

Iterator für ostream. Unformatierte Ausgabe, char für char. Beispiel:

```
int main () {
    std::string mystring ("Some text here...\n");
    std::ostreambuf_iterator<char> out_it (std::cout);
    std::copy ( mystring.begin(), mystring.end(), out_it);
}
```

# std::reverse\_iterator

---

Iterator in umgekehrter Reihenfolge. Funktioniert nur bei Bidirectional oder Random Access. Benutzung entweder mit rbegin(), rend() auf einem Container oder wie folgt:

```
int main () {
    std::vector<int> myvector{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    using iter_type = std::vector<int>::iterator;
    iter_type from = myvector.begin();
    iter_type until = myvector.end();
    std::reverse_iterator<iter_type> rev_until (from);
    std::reverse_iterator<iter_type> rev_from (until);
    while (rev_from != rev_until) std::cout << ' ' << *rev_from
}
```

Output: " 9 8 7 6 5 4 3 2 1 0"

# Containers

---

## Funktionen aller Container (ausser wenn anders vermerkt):

begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend(), size(), empty()

Statt `operator[]` wird `.at` empfohlen wenn verfügbar, da es dort Bound Checks gibt (HSR brennt nicht ab)

## Iterieren "foreach"

```
for(auto const i:v) {  
    std::cout << "element: " << i << "\n";  
}
```

Um auch ändern zu können, braucht man eine Referenz als Loop-Variable

```
for(auto &j:v) {  
    j *= 2;  
}
```

# std::vector

---

Entspricht ArrayList in java

**Include / Initialisieren:** `#include <vector> / std::vector<int> v{};`, alternativ mit Angabe der Grösse `std::vector<int> v(6)` oder gleich mit 2 füllen `(6,2)`

Ebenso kann er aus zwei Iteratoren konstruiert werden.

**Iterator:** random access



## Insert:

```
insert (const_iterator position, const value_type& val);
```

```
push_back (const value_type& val);
```

```
back_inserter(v) // Ziel für Copy
```

Mit unterschiedlichen Werten füllen:

```
std::vector<double> w{};  
generate_n(std::back_inserter(w),5,  
    [x=2.0]() mutable {return x *= 2.0;}  
);
```

## Delete:

```
erase (const_iterator position);
```

```
erase (const_iterator first, const_iterator last);
```

```
pop_back();
```

**Get/Change:** operator[]

# std::array

---

Wrapper über Klassischen C array mit length Feld

**Include/Initialisieren:** `#include <array>/std::array<int, 6> a{{1, 2, 3, 4, 5}};` // 6=size, Achtung doppelte geschw. Klammern!

**Iterator:** random access

**Insert:** -

**Delete:** -

**Get/Change:** `operator[]`

## std::deque

---

Die Double Ended Queue funktioniert ähnlich wie ein vector aber zusätzlich noch `push_front()`, `pop_front()` Methoden.

**Include / Initialisieren:** `#include <deque> / std::deque<int> d{};`

**Iterator:** random access

**Insert:**

```
insert(const_iterator position, const value_type& val);
```

```
push_back (const value_type& val);
```

```
push_front(const value_type& val);
```

**Delete:**

```
erase (const_iterator position);
```

```
erase (const_iterator first, const_iterator last);
```

```
pop_back(); pop_front();
```

**Get/Change:** `operator[]`

## std::list

---

Double Linked List

**Include / Initialisieren:** `#include <list> / std::list<int> l{};`

**Iterator:** bidirectional

**Insert:**

```
insert(const_iterator position, const value_type& val);
```

```
push_back (const value_type& val);
```

```
push_front(const value_type& val);
```

**Delete:**

```
erase (const_iterator position);
```

```
erase (const_iterator first, const_iterator last);
```

```
pop_back(); pop_front();
```

**Get:** `front(); back; // für mittlere Elemente Iterators benutzen`

## std::forward\_list

---

Singly Linked List

**Include/Initialisieren:** `#include<forward_list>/std::forward_list<int> l{};`

**Iterator:** forward

**Insert:**

```
insert_after(const_iterator position, value_type& val);
```

```
push_front(const value_type& val);
```

### Delete:

```
erase_after (const_iterator position);
```

```
pop_front();
```

**Get:** `front();` //für andere Elemente Iterators benutzen

## std::stack

---

LIFO (last in first out), **keine Iteratoren!**

**Include / Initialisieren:** `#include <stack> / std::stack<int> s{};`

**Iterator:** -

**Insert:** `push(const value_type& val);`

**Delete:** `pop();`

**Get:** `top();` // peek() in java

## std::queue

---

FIFO (first in first out), **keine Iteratoren!**

**Include / Initialisieren:** `#include <queue> / std::queue<int> {};`

**Iterator:** -

**Insert:** `push(const value_type& val);`

**Delete:** `pop();`

**Get:** `front();`

## **std::priority\_queue**

---

Pop nimmt grösstes Element aus der Queue, **keine Iteratoren!**

**Include / Initialisieren:** `#include <queue> /  
std::priority_queue<int> {};`

**Iterator:** -

**Insert:** `push(const value_type& val);`

**Delete:** `pop();`

**Get:** `top();`

## **std::set**

---

Entspricht TreeSet in Java, aufsteigend sortiert ohne Duplikate  
(Sortierung als zweiter Template-Parameter möglich)

**Include / Initialisieren:** `#include <set> / std::set<int> s{}; ODER  
AUCH std::set<int, std::less> s{};`

**Iterator:** bidirectional (immer const)

**Insert:** `insert (const value_type& val);`

**Delete:**

`erase (const_iterator position);`

`erase (const_iterator first, const_iterator last);`

**Get:** iterators

## std::multiset

---

Aufsteigend sortiert mit Duplikaten

**Include / Initialisieren:** `#include <set> / std::multiset<int> s{};`

**Iterator:** bidirectional (immer const)

**Insert:** `insert (const value_type& val);`

**Delete:**

`erase (const_iterator position);`

`erase (const_iterator first, const_iterator last);`

**Get:** iterators

## std::map

---

Entspricht TreeMap in Java, aufsteigend sortiert mit Key und Value

**Include / Initialisieren:** `#include <map> / std::map<Key, Value> s{};`

**Iterator:** bidirectional über std::pair

**Insert:** `insert (std::pair<Key, Value>{"key", "value"});`

```
std::map<std::string, size_t> words;  
std::string s;
```

```
while (std::cin >> s)
    ++words[s]; //erstellt automatisch einen Eintrag
```

**Delete:**

```
erase (const_iterator position);
```

```
erase (const_iterator first, const_iterator last);
```

**Get:** `map[key]`

## std::multimap

---

Map mit mehreren Elementen mit gleichen Keys

**Include / Initialisieren:** `#include <map> / std::multimap<Key, Value> s{};`

**Iterator:** bidirectional über `std::pair`

**Insert:** `insert (std::pair<Key, Value>{"key", "value"});`

**Delete:**

```
erase (const_iterator position);
```

```
erase (const_iterator first, const_iterator last);
```

**Get:** `multimap.find(Key)` -> iterator über alle values

## std::unordered\_set

---

HashSet in Java, nicht benutzen → C++ advanced

# std::unordered\_map

---

HashMap in Java, nicht benutzen → C++ advanced

## Nackte Arrays

---

Wenn man mit nackten Arrays arbeitet, kann man diese trotzdem mit Iteratoren verwenden:

- Anfang: `array`
- Ende: `array+size` .

## Initialisieren

---

```
int five[5]{}; // 5 zeros
int four[4] = {1, 2, 3, 5};
double d[4]{1.0, 2.0}; // d[2] = d[3] = 0.0
double m[2][3]{ {1, 2, 3} , {4, 5, 6} };
char s[6]{"hello"}; // 5 chars + '\0'
```

Achtung: mehrdimensionale Arrays nicht mit Komma schreiben. Der `operator,` evaluiert alle Subexpressions sequentiell und ist nur nützlich, wenn der erste Teil einen Seiteneffekt hat.

## Länge erkennen mit Type Deduction

---

```
template <typename T, unsigned N>
// () benötigt
void printArray(std::ostream & out, T const (&x)[N]) {
```



```
copy(x, x+N, std::ostream_iterator<T>{out, ", "});  
}
```

Die Type Deduction presst das dann so rein dass in N die Grösse steht. Wenn man ein Array mittels Liste erstellt, wird die Grösse automatisch festgelegt.

**Nackte Arrays sollten nicht verwendet werden. Ersetzen mit `std::array`**

# Lambdas

Wenn es alle möglichen Klammern hat, ist es wahrscheinlich ein Lambda.

```
[lambda_capture]  
(parameters)->return_type {  
    statements  
}
```

Beispiel:

```
auto const g=[](char c){return std::toupper(c);}; // beide S  
g('a');
```

- Parameter sind wie Funktionsparameter, auto möglich. Klammer kann auch weggelassen werden
- return\_type kann weggelassen werden wenn void oder die return-Statements im Typ konsistent sind (Compiler erkennt). **Dann aber auch den Pfeil weglassen.**

# Captures und Parameter

- Capture benennt Variablen die vom umgebenden Scope genommen werden oder erstellt sogar neue
  - `=` copy
  - `&` reference
  - `var=value` erzeugt neue Capture-Variable mit Wert
  - Können auch kombiniert werden
    - `=, &out` capturet alle by copy, aber `out` by reference
    - `&,=x` capturet alle by reference, aber `x` by copy/value
    - Guideline: alle Captures explizit machen
  - Typ wird abgeleitet
  - Wenn man Member-Variablen captured, sind diese automatisch by reference, auch wenn man `[=]` automatisch by copy macht.
    - this ist schon eine Reference
- Parameter sind wie Funktionsparameter, auto möglich
- **Captures werden zur Definitionszeit festgelegt, Parameter zur Aufrufzeit**

```
void testLambdaCapture() {  
    int n{5};  
    auto lambdaCapture=[n]() {return n;};  
    n++;  
    ASSERT_EQUAL(lambdaCapture(), 5);  
    ASSERT_EQUAL(n, 6);  
}
```

```
void testLambdaCaptureReference() {  
    int n{5};  
    auto lambdaCapture=[&n]() {return n;};  
    n++;  
    ASSERT_EQUAL(lambdaCapture(), 6);  
}
```

```

    ASSERT_EQUAL(n, 6);
}

void testLambdaParameter() {
    int n{5};
    auto lambdaCapture=[](int m){return m;};
    n++;
    ASSERT_EQUAL(lambdaCapture(n), 6);
    ASSERT_EQUAL(n, 6);
}

```

## Spezialfall mutable

Variables die by copy gecaptured werden sind im Lambda const, ausser wenn das Lambda als `mutable` gekennzeichnet wird. Das Lambda bekommt seine eigene Kopie der Variable oder das Lambda definiert die Variable gleich neu.

## Functor

Ein Functor ist eine Klasse, die einen Call-Operator hat: `operator()`. Gegenüber einer normalen Funktion hat sie Speicher zur Verfügung, der auch bleibt.

Der Functor kann auch gleich mit `doNothingfunctor{}()` konstruiert und gleich aufgerufen werden.

```

struct Accumulator {
    int count{0};
    int accumulated_value{0};
    void operator()(int value) {
        count++;
    }
}

```

```

        accumulated_value += value;
    }
    int average() const;
    int sum() const;
};

// Achtung nicht Implementation von oberem
int average(std::vector<int> values) {
    Accumulator acc{};
    for(auto v: values) { acc(v); }
    return acc.average();
}

```

Lambdas werden intern mit Functors realisiert.

Es gibt ein paar Standard Functors, wie z.B. `std::greater<>{} in <functional>` .

## 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_to<>` (`!=`)

Diese lassen sich dann in manchen Containern als Vergleichsoperator mitgeben, allerdings nur wenn wie irreflexiv und transitiv sind (ah ja.)

## Als Parameter

`std::function<SIGNATURE>;` , also zum Beispiel

`std::function<bool(int)> apredicate{};`

### Beispiel

```
void apply(std::ostream& out, std::function<bool(int)> f) {
    if(f) {
        //safe to use f
    } else {
        //empty function holder
    }
}

int main() {
    std::function<bool(int)> f;
    f = [](int i) {return i%2};
    apply(std::cout, f);
    f = nullptr;
    apply(std::cout, f);
}
```

Die Signatur wird geprüft.

# Algorithms

---

Ein paar Beispiele: Jeder Container hat `size()`. Was wenn man nur zwei Iteratoren hat? `std::distance(begin, end)`

`for_each` (halbböse):

```
for_each(begin(v), end(v), [](auto x) {  
    std::cout << x++ << "\n";  
});
```

Wenn man eine Funktion `print(int x)` hat, geht auch  
`for_each(begin(v), end(v), print);`

## Ranges

---

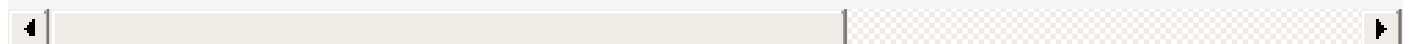
Algorithmen brauchen oft Ranges, die man mit Iteratoren angibt.

## Beispiele aus Vorlesung

---

**transform** Eine Range zu neuen Values umwandeln, oder zwei Ranges derselben Grösse. Input: `counts` : 3, 0, 1, 4, 0, 2 Input: `letters` : g, a, u, y, f, o Output: ggg, , u, yyyy, , oo

```
std::vector<std::string> combined{};  
auto times = [](int i, char c) {return std::string(i, c);};  
std::transform(begin(counts), end(counts), begin(letters), s
```



**merge** Zwei Ranges sortiert mergen

**accumulate** Kann auch so gewürdigt werden dass es nicht-numerisches unterstützt, z.B. Listen.

**Erase/Remove** Wenn man Elemente löschen will, so macht man das immer in zwei Teilen

1. Finden, was zu löschen ist. Technisch: alles zu Löschende ans Ende verschieben und Iterator auf erstes zu löschendes Element zurückgeben: `remove`
2. Danach löscht man von diesem gegebenen Iterator bis zum `end` :  
`erase`

```
// removes all elements with the value 5  
v.erase( std::remove( v.begin(), v.end(), 5 ), v.end() );
```

## Suffix-Versionen

---

`_if` Von manchen gibts noch eine `_if` -Version, die noch ein Predicate nimmt

`_n` Vielfach statt dem `last` -Iterator einfach eine Angabe wie oft.

## Heap-Algorithmen

---

Bauen die Container so um, dass sie einem Heap entsprechen

### Operationen

- `make_heap` :  $3 \cdot N$  Vergleiche
- `pop_heap` :  $2 \cdot \log(N)$  Vergleiche
- `push_heap` :  $\log(N)$  Vergleiche

- `sort_heap` :  $N \cdot \log(N)$  Vergleiche

## Fallen

---

- Die Iteratoren müssen natürlich zum selben Range gehören, sonst brennt die HSR ab.
- Bei den copy-Algorithmen muss genügend Platz im Ziel sein, sonst brennt die HSR ab. Wenn man sich nicht darum kümmern will: `back_inserter`, `front_inserter` oder `inserter`.
- Manche Operationen machen die Iteratoren ungültig, zum Beispiel ein Push-Back auf einem Vector. Der end-Iterator zeigt dann nicht mehr auf den richtigen Ort.

## `std::lexicographical_compare`

---

```
struct caselessCompare {  
    bool operator()(std::string const &left, std::string const &right) const {  
        return std::lexicographical_compare(left.begin(), left.end(),  
            right.begin(), right.end(),  
            [](char l, char r) {  
                return std::tolower(l) < std::tolower(r);  
            });  
    }  
};
```

## Tabelle

---

|  |
|--|
| <b>Non-modifying sequence operations</b> |
|--|



|  |  |
|--|--|
| Defined in header <code>&lt;algorithm&gt;</code> |  |
| all_of any_of none_of<br>(C++11) (C++11) (C++11) | checks if a predicate is true for all, any or none of the elements in a range    |
| for_each   | applies a function to a range of elements  |
| for_each_n<br>(C++17)                            | applies a function object to the first n elements of a sequence                  |
| count count_if                                   | returns the number of elements satisfying specific criteria                      |
| mismatch   | finds the first position where two ranges differ                                 |
| equal  | determines if two sets of elements are the same                                  |
| find find_if find_if_not<br>(C++11)              | finds the first element satisfying specific criteria                             |
| find_end   | finds the last sequence of elements in a certain range                           |
| find_first_of                                    | searches for any one of a set of elements  |
| adjacent_find                                    | finds the first two adjacent items that are equal (or satisfy a given predicate) |
| search   | searches for a range of elements   |
| search_n   | searches for a number of consecutive copies of an element in a range             |
| <b>Modifying sequence operations</b>             |  |
| Defined in header <code>&lt;algorithm&gt;</code> |  |
| copy copy_if<br>(C++11)                          | copies a range of elements to a new location                                     |

|                                 |  |
|---------------------------------|--|
| copy_n<br>(C++11)               | copies a number of elements to a new location                                      |
| copy_backward                   | copies a range of elements in backwards order                                      |
| move<br>(C++11)                 | moves a range of elements to a new location  |
| move_backward<br>(C++11)        | moves a range of elements to a new location in backwards order                     |
| fill                            | copy-assigns the given value to every element in a range                           |
| fill_n                          | copy-assigns the given value to N elements in a range                              |
| transform                       | applies a function to a range of elements  |
| generate                        | assigns the results of successive function calls to every element in a range       |
| generate_n                      | assigns the results of successive function calls to N elements in a range          |
| remove remove_if                | removes elements satisfying specific criteria                                      |
| remove_copy<br>remove_copy_if   | copies a range of elements omitting those that satisfy specific criteria           |
| replace replace_if              | replaces all values satisfying specific criteria with another value                |
| replace_copy<br>replace_copy_if | copies a range, replacing elements satisfying specific criteria with another value |
| swap                            | swaps the values of two objects  |
| swap_ranges                     | swaps two ranges of elements   |
|                                 |  |

|   |  |
|---|--|
| iter_swap                                       | swaps the elements pointed to by two iterators                                   |
| reverse   | reverses the order of elements in a range  |
| reverse_copy                                    | creates a copy of a range that is reversed                                       |
| rotate  | rotates the order of elements in a range   |
| rotate_copy                                     | copies and rotate a range of elements  |
| random_shuffle shuffle<br>(until C++17) (C++11) | randomly re-orders elements in a range   |
| sample<br>(C++17)                               | selects n random elements from a sequence  |
| unique  | removes consecutive duplicate elements in a range                                |
| unique_copy                                     | creates a copy of some range of elements that contains no consecutive duplicates |

## Partitioning operations

Defined in header `<algorithm>`

|                           |  |
|---------------------------|--|
| is_partitioned<br>(C++11) | determines if the range is partitioned by the given predicate          |
| partition                 | divides a range of elements into two groups                            |
| partition_copy<br>(C++11) | copies a range dividing the elements into two groups                   |
| stable_partition          | divides elements into two groups while preserving their relative order |
| partition_point           | locates the partition point of a                                       |

|  |   |
|--|---|
| (C++11)  | partitioned range   |
| <b>Sorting operations</b>                          |   |
| Defined in header <code>&lt;algorithm&gt;</code>   |   |
| is_sorted<br>(C++11)                               | checks whether a range is sorted into ascending order                                   |
| is_sorted_until<br>(C++11)                         | finds the largest sorted subrange   |
| sort   | sorts a range into ascending order  |
| partial_sort                                       | sorts the first N elements of a range   |
| partial_sort_copy                                  | copies and partially sorts a range of elements  |
| stable_sort  | sorts a range of elements while preserving order between equal elements                 |
| nth_element  | partially sorts the given range making sure that it is partitioned by the given element |
| <b>Binary search operations (on sorted ranges)</b> |   |
| Defined in header <code>&lt;algorithm&gt;</code>   |   |
| lower_bound  | returns an iterator to the first element <i>not less</i> than the given value           |
| upper_bound  | returns an iterator to the first element <i>greater</i> than a certain value            |
| binary_search                                      | determines if an element exists in a certain range                                      |
| equal_range  | returns range of elements matching a specific key                                       |

## Set operations (on sorted ranges)

Defined in header `<algorithm>`

|                                       |  |
|---------------------------------------|--|
| <code>merge</code>                    | merges two sorted ranges                           |
| <code>inplace_merge</code>            | merges two ordered ranges in-place                 |
| <code>includes</code>                 | returns true if one set is a subset of another     |
| <code>set_difference</code>           | computes the difference between two sets           |
| <code>set_intersection</code>         | computes the intersection of two sets              |
| <code>set_symmetric_difference</code> | computes the symmetric difference between two sets |
| <code>set_union</code>                | computes the union of two sets                     |

## Heap operations

Defined in header `<algorithm>`

|                                       |   |
|---------------------------------------|---|
| <code>is_heap</code><br>(C++11)       | checks if the given range is a max heap                             |
| <code>is_heap_until</code><br>(C++11) | finds the largest subrange that is a max heap                       |
| <code>make_heap</code>                | creates a max heap out of a range of elements                       |
| <code>push_heap</code>                | adds an element to a max heap                                       |
| <code>pop_heap</code>                 | removes the largest element from a max heap                         |
| <code>sort_heap</code>                | turns a max heap into a range of elements sorted in ascending order |

## Minimum/maximum operations

Defined in header `<algorithm>`

|  |   |
|--|---|
| <code>max</code>                       | returns the greater of the given values                                     |
| <code>max_element</code>               | returns the largest element in a range                                      |
| <code>min</code>                       | returns the smaller of the given values                                     |
| <code>min_element</code>               | returns the smallest element in a range                                     |
| <code>minmax</code><br>(C++11)         | returns the smaller and larger of two elements                              |
| <code>minmax_element</code><br>(C++11) | returns the smallest and the largest elements in a range                    |
| <code>clamp</code><br>(C++17)          | clamps a value between a pair of boundary values                            |
| <code>lexicographical_compare</code>   | returns true if one range is lexicographically less than another            |
| <code>is_permutation</code><br>(C++11) | determines if a sequence is a permutation of another sequence               |
| <code>next_permutation</code>          | generates the next greater lexicographic permutation of a range of elements |
| <code>prev_permutation</code>          | generates the next smaller lexicographic permutation of a range of elements |

## Numeric operations

Defined in header `<numeric>`

|                              |  |
|------------------------------|--|
| <code>iota</code><br>(C++11) | fills a range with successive increments of the starting value |
| <code>accumulate</code>      | sums up a range of elements                                    |

|                                     |   |
|-------------------------------------|---|
| inner_product                       | computes the inner product of two ranges of elements                          |
| adjacent_difference                 | computes the differences between adjacent elements in a range                 |
| partial_sum                         | computes the partial sum of a range of elements                               |
| reduce<br>(C++17)                   | similar to std::accumulate , except out of order                              |
| exclusive_scan<br>(C++17)           | similar to std::partial_sum , excludes the ith input element from the ith sum |
| inclusive_scan<br>(C++17)           | similar to std::partial_sum , includes the ith input element in the ith sum   |
| transform_reduce<br>(C++17)         | applies a functor, then reduces out of order                                  |
| transform_exclusive_scan<br>(C++17) | applies a functor, then calculates exclusive scan                             |
| transform_inclusive_scan<br>(C++17) | applies a functor, then calculates inclusive scan                             |

# Klassen

Es gibt zwei Keywords, `struct` und `class` . Die sind äquivalent, ausser

- `struct` ist standardmässig public
- `class` ist standardmässig private

Eine gute Klasse kennt eine Klasseninvariante, d.h. dass eine Instanz sich immer in einem guten Zustand befindet. Falls eine Änderung diese Invarianz verletzt, wird sie^ entweder zurückgerollt oder zerstört. Aber

nicht im FUBAR-Zustand belassen.

## Beispielklasse Date

---

```
#ifndef DATE_H_
#define DATE_H_
class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year{year}, month{month}, day{day} // initialzer list
    {
        ...
    }
    static bool isLeapYear(int year) {
        ...
    }
private:
    bool isValidDate() const {
        ...
    }
}; //Semikolon am Ende der Klassendefinition nicht vergessen

#endif /* DATE_H_ */
```

## Access Specifier

---

- private
- protected (auch Subklassen)
- public

Visibilities können auch mehrmals verwendet werden



# Member Variables

---

Haben einen Typ und einen Namen. So const wie möglich.

```
<type> <name>
```

Sie sind im Header File, damit das Speicherlayout bekannt ist.

## Static Member-Variablen

---

**Im Header:** als `static` oder als `static const` deklarieren. `static const` dürfen auch gleich initialisiert werden:

```
Class Date {  
    static const Date myBirthday;  
    static const Date favoriteStudentsBirthday;  
    static const int zero{0};  
}
```

**Im Implementationfile:** kein `static`. Es dürfen auch const-Variablen hier initialisiert werden (aber auch nicht const):

```
#include "Date.h"  
Date const Date::myBirthday{1964, 12, 24};  
Date Date::favoriteStudentsBirthday{1995, 5, 10};
```

**Ausserhalb der Klasse:** mit `::`:

```
#include "Date.h"  
Date::favoriteStudentsBirthday = ...;
```

# Konstruktor

---

Der Konstruktor ist eine spezielle Member Funktion. Er hat **keinen** Rückgabewert. Es gibt eine Initializer-List für Member-Initialisierung

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

## Spezielle Konstrukturen

**Default Constructor** `Date(); / Date d{};`

Keine Parameter, implizit verfügbar wenn es keine anderen Konstrukturen gibt. Initialisiert die Member-Variablen mit Default-Werten

**Copy Constructor** `Date(Date const &); / Date d2{d};`

Hat einen `<own type> const &` Parameter. Implizit verfügbar (ausser es gibt einen expliziten Move-Konstruktor oder Assignment-Operator).

Kopiert alle Member-Variablen. Implementiert man normalerweise nicht selber.

**Move Constructor** `Date(date &&); / date d2{std::move(d)}`

Hat einen `<own type> &&` Parameter. Implizit verfügbar (ausser es gibt einen expliziten Copy-Konstruktor oder Assignment-Operator).

Verschiebt alle Member (d ist dann tot). Implementiert man normalerweise nicht selber.

**Typeconversion Constructor** `explicit Date(std::string const &);  
/ Date d{"19/10/2016"s}`

Hat einen `<other-type> const &` Parameter. Konvertiert den Input-Typ wenn möglich. `explicit` deklarieren, damit nicht versucht wird ein anderer Typ in diesen String (im Beispiel) hineinzupressen.

## Implementation

Der Konstruktor soll die Invariante etablieren und die Member initialisieren. Konstruktoren bauen nur valide Instanzen und werfen ansonsten Exceptions. Beim Default-Konstruktor ohne Parameter sollten sinnvolle Defaultwerte gesetzt werden

Date.cpp

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

## Konstruktor mit `std::istream` &

Man kann einen Konstruktor definieren, der explizit nur istreams entgegennimmt: `explicit Date(std::istream & in)`

Wenn das erstellen fehlschlägt, wird eine Exception geworfen.

```
Date::Date(std::istream & in)
    : year{}, month{}, day{}
{
    read(in); // read muss selber implementiert werden
    if(in.fail()) {
        throw std::out_of_range("invalid date");
    }
}
```

Man könnte natürlich auch eine Factory-Funktion machen.

## Konstruktoren wieder default machen/löschen

Wenn man einen eigenen Konstruktor gemacht hat, ist der Default-Konstruktor weg. Mit folgendem Befehl kann man ihn einfach wieder definieren:

```
<constructor-name>() = default;
```

Ebenso kann man Konstruktoren löschen:

```
<constructor-name>() = delete;
```

## Destruktoren

Genannt wie der Default-Konstruktor mit einem ~ zu Beginn: `~Date();`

Muss alle Ressourcen freigeben. Implizit verfügbar. Darf keine Exception werfen! Wird automatisch am Ende des Blocks für alle lokalen Instanzen aufgerufen.

TODO muss bei Klassen mit virtual Methoden immer virtual deklariert

werden.

# Implementation

---

Die eigentliche Implementierung sollte die Klasse im Header-File inkludieren und dann die Methoden implementieren. Wichtig: die Scope Specifier beachten

```
#include "Date.h"

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

bool Date::isLeapYear(int year) {
    ...
}

bool Date::isValidDate() const {
    ...
}
```

# Benutzung

---

```
#include "Date.h"

void foo() {
    Date today{2016, 10, 19};

    Date::isLeapYear(2016)
}
```

# Member-Funktionen

---

Dürfen die Invariance nicht verletzen.

Es gibt das implizite `this`-Objekt. `this` ist ein Pointer und muss darum vor dem Zugriff dereferenziert werden. Entweder mit `(*this).day` oder mit `this->day`. Wenn es im Scope der Methode nicht noch eine andere Variable "day" gibt kann auch einfach mit `day` darauf zugegriffen werden..

In einer const-Memberfunktion dürfen die Member nicht verändert werden. Es können nur const-Member aufgerufen werden.

## Static Member-Funktionen

Es gibt kein this-Objekt, können **nicht** const sein. Kein static Keyword.

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

## Implementation Header oder CPP File

Wenn die Methoden keine zusätzlichen Dependencies hat und die implementation klein und offensichtlich ist, können diese im Header implementiert werden.

## Member(Function)-Pointers

---

Man kann auf Member(Funktionen) referenzieren.

**Beispiel**

```

struct Klasse {
    void foo() const;
    void bar() const;
    int a;
    int b;
};

/**Funktion objekt und Funktion mitgeben, welche aufgerufen wird
void doit(void(Klasse::*mfunc)() const, Klasse const &instanz) {
    (instanz.*mfunc)(); //Klammern nötig
}

/**Funktion objekt und variable mitgeben, welche den Wert ändert
void change(int Klasse::*var, Klasse& instanz, int val) {
    instanz.*var = val;
}

int main() {
    Klasse x{1,2};
    //Referenzen sind auf die Klasse für MemberPointers!
    doit(&Klasse::foo,x);
    change(&Klasse::a,x,3);
}

```

## Operator-Overloading

When in doubt, do as the ints do

Wie eine Funktion deklariert, allerdings als mit speziellem Namen: ``operator();

Unäre/binäre Parameter haben einen bzw. zwei Parameter.

**Überladbare Operatoren:**

|  |  |  |  |  |  |
|--|--|--|--|--|--|
|  |  |  |  |  |  |
|--|--|--|--|--|--|

|    |     |     |       |        |          |
|----|-----|-----|-------|--------|----------|
| +  | -   |     | /     | %      | ^        |
| &  |     | ~   | !     | ,      | =        |
| <  | >   | <=  | >=    | ++     | --       |
| << | >>  | ==  | !=    | &&     |          |
| += | -=  | /=  | %=    | ^=     | &=       |
| =  | =   | <<= | >>=   | []     | ()       |
| -> | ->* | new | new[] | delete | delete[] |

## Nicht überladbare Operatoren

|    |    |   |              |
|----|----|---|--------------|
| :: | .* | . | : ? (ternär) |
|----|----|---|--------------|

## Beispiel: Date vergleichbar machen

year, month und day vergleichen. Wir nutzen den `operator<`. Immer `const &!` Rückgabewert `bool`. Achtung, hier wird eine Entwicklung gezeigt, damit man den Unterschied sieht.

Erste Variante: als freier Operator (in Klassenfile, aber unterhalb Klassendefinition). Hier zwei Parameter, Date lhs und Date rhs. `inline` verwenden, da im Header definiert.

```
class Date {
    int year, month, day; // private
};

// ACHTUNG DAS GEHT NICHT!!!
inline bool operator<(Date const & lhs, Date const & rhs) {
    lhs.year? // Geht nicht, kein Access auf private Member
}
```



Zweite Variante: als Member Operator (innerhalb Klassendefinition). Nur noch ein Parameter, Date rhs (implizites lhs/this). Ebenso implizites `inline` als Member. Da die Methode `const` ist, ist auch das `this` `const`.

```
class Date {
    int year, month, day; // private

    bool operator<(Date const & rhs) const {
        return year < rhs.year ||
            (year == rhs.year && (month < rhs.month ||
                (month == rhs.month && day == rhs.day)))
    };
};
```

**In Verwendung:** einfach den `<` Operator verwenden:

```
std::cout << "is d older? " << (d < Date::myBirthday);
```

**Syntactic Sugar für Vergleiche** `std::tie` kreiert ein Tupel. ( `<tuple>` )

```
return std::tie(year, month, day) < std::tie(rhs.year, rh
```

`std::tuple` (Header) bietet die folgenden Operatoren

- `operator==`
- `operator!=`
- `operator<`
- `operator<=`
- `operator>`
- `operator>=`

Der Vergleich ist immer komponentenweise von links nach rechts.

## Andere Vergleiche implementieren

```
class Date {
    int year, month, day; //private
public:
    bool operator<(Date const & rhs) const;
};

inline bool operator>(Date const & lhs, Date const & rhs) {
    return rhs < lhs;
}
inline bool operator>=(Date const & lhs, Date const & rhs) {
    return !(lhs < rhs);
}
inline bool operator<=(Date const & lhs, Date const & rhs) {
    return !(rhs < lhs);
}
inline bool operator==(Date const & lhs, Date const & rhs) {
    return !(lhs < rhs) && !(rhs < lhs);
}
inline bool operator!=(Date const & lhs, Date const & rhs) {
    return !(lhs == rhs);
}
```

Die ganzen Operatoren ausserhalb der Klasse sehen wie Boilerplate-Code aus. Deshalb gibts von Boost eine Klasse von der man erben kann, die genau diese zusätzlichen Operatoren bietet. `private` erben reicht aus.

Benötigt `<`, bietet

- `>`
- `<=`
- `>=`

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

class Date : private boost::less_than_comparable<Date> {
    int year, month, day; //private
public:
    bool operator<(Date const & rhs) const {
        return std::tie(year, month, day) < std::tie(rhs.year, rhs.month, rhs.day);
    }
};
```

## Beispiel: Date an std::cout senden

Was wir wollen:

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

Erste Variante: als freier Operator. Parameter: `std::ostream &` und `Date const &`. Rückgabewert: `std::ostream &` für Output Chaining

Problem: kann private Member nicht ansprechen.

Zweite Variante: als Memberfunktion. Problem: `std::ostream` darf nicht links sein in Calls. Andersherum würds zwar vom Compiler her gehen (`Date::myBirthday << std::cout`) aber das ist per Konvention falsch.

Dritte Variante: public `print(std::ostream & os) const` - Memberfunktion, die von `operator<<` aufgerufen wird. Jetzt funktioniert alles!

```
#include <ostream>
```

```

class Date {
    int year, month, day;
public:
    std::ostream & print(std::ostream & os) const {
        os << year << "/" << month << "/"

};

inline std::ostream & operator<<(std::ostream & os, Date const& date) {
    return date.print(os);
}

```

Dieses "Pattern" braucht man immer wieder, auch z.B. fürs Einlesen. In den Streams gibt es auch ein Beispiel fürs Einlesen von Dates.

# Vererbung

## Syntax

```

class Base {};
class Derived : public Base {};

```

Die Reihenfolge ist wichtig! Wenn man ein Interface erbt, muss die Base public geerbt werden. Private ist möglich, aber für Mix-Ins gedacht (Mix-Ins sind Klassen die nur Funktionen hinzufügen, z.B.

boost/operators.hpp)

## Mehrfachvererbung

```

class Base {};

```

```
struct MixIn{};
class MultipleBases : public Base, private MixIn {};
```

Die Basisklassen werden in Reihenfolge ihrer Angaben initialisiert. Normalerweise sind private Basisklassen falsches Design.

## Initialisierung

Die Base-Class-Konstruktoren-Aufrufe kommen vor die Member-Initializers in der Konstruktor-Initialzer-List (vor Body).

Es gibt kein `super()`. Die Klasse muss selber konstruiert werden, bevor der Body anfängt zu laufen. Die Dekonstruktoren werden dann von Base zu Super dekonstruiert

```
class DerivedWithCtor : public Base {
    //zuerst Basis Konstruktor aufrufen und dann können die i
    DerivedWithCtor(int i, int j):Base{i}, mvar{j} {}
};
```

## Member Hiding Problem


Überladene Memberfunktionen in abgeleiteten Klassen verstecken alle Funktionen mit selben Namen der Basis Klassen

```
struct Base {
    void foo(int i) const;
};
struct Derived:Base {
    void foo();
};
```

```
int main() {
    Derviced d{};
    //d.foo(31); //is hidden
}
```

## Lösung

```
struct Derived:Base {
    using Base::foo; //zuerst using damit die Funktionen ver
    void foo();
};
```



# Sichtbarkeit

---

Die Vererbung kann sogar eine Visibility haben. Dies beschränkt die **maximale** Visibility der geerbten Member.

Siehe Beispiel für Erklärung:

```
class Base {
protected:
    int a{0};
public:
    int b{1};
private:
    int c{2}; // auf private members kann nie aus Subklassen
}

class A: private Base {
    // A kann auf a und b zugreifen
    // Subklassen von A können a und b nicht verwenden
    // Ausserhalb der Klassenhyrarchie kann a und b nicht ve
```

```
}■
```

```
class B: protected Base {  
    // B kann auf a und b zugreifen  
    // Subklassen von B können a und b verwenden  
    // Ausserhalb der Klassenhierarchy kann a und b nicht ve
```

```
}■
```

```
class C: public Base { // public ist default  
    // C kann auf a und b zugreifen  
    // Subklassen von C können a und b verwenden  
    // Ausserhalb der Klassenhierarchy kann a aber nicht b v
```

```
}■
```

## Object Slicing

```
struct Base {  
    int a;  
    Base(int a): a{a} {}  
}■
```

```
struct Sub: Base {  
    int b;  
    Sub(int a, int b): Base{a}, b{b} {}  
}■
```

```
Sub sub{1,2};  
Base base{sub}; // ACHTUNG, Nur der Base Teil wird kopiert!
```

## Probleme mit Vererbung und pass-by-value

Wenn eine abgeleitete Klasse Funktionen definiert, werden die Base-Funktionen versteckt. Kann problematisch sein, vorallem mit const/non-const. **Achtung: dies gilt nicht wenn die Funktion einmal const (z.B. in Base) und einmal non-const (z.B. in Derived) gemacht wird.**

Um das zu umgehen macht man ein `using` auf die Base Class Funktionen, dann wird die Base-Funktion so behandelt als wär sie in der abgeleiteten Klasse und nimmt am Overloading teil.

TODO Beispiel

## Virtual

---

Wenn eine Funktion mit dynamischem Polymorphismus aufgerufen werden soll, muss sie in der Base-Klasse als `virtual` deklariert werden. Sie bleibt dann virtual in allen abgeleiteten Klassen, bis eine sie als `final` deklariert.

**Sobald Funktionen `virtual` deklariert wurden, muss der Destruktor auch `virtual` sein!**

```
class PolymorphicBase {
public:
    virtual void doit() { /* etwas */ }
};

class Implementor : public PolymorphicBase {
public:
    //can be marked with override. this will lead to an comp.
    void doit() { /* etwas anderes */ }
};
```



Welche Funktion tatsächlich aufgerufen wird kommt auch drauf an, ob man einen Wert oder eine Referenz hat:

- Value-Objekt: Klassentyp definiert aufgerufene Funktion, egal ob virtual
- Reference oder Pointer: Virtual Member der abgeleiteten Klasse wird aufgerufen (durch Referenzen auf Basisklasse). Aber nur für Funktionen die `virtual` sind.

Wieso braucht man virtual: C++ baut Sachen nur auf, wenn man sie tatsächlich braucht. Wenn man eine Funktion `virtual` deklariert, wird eine `vtable` für diese Funktion aufgebaut, die immer trackt welche Funktion denn jetzt tatsächlich aufgerufen werden soll.

**Beispiel: Bird** TODO Beispiel verkürzen

```
#include <iostream>
using std::cout;

struct Animal {
    void makeSound() {
        cout << "---\n";
    }
    virtual void move() {
        cout << "---\n";
    }
    Animal() {
        cout << "animal born\n";
    }
    ~Animal() {
        cout << "animal died\n";
    }
};

struct Bird: Animal {
    virtual void makeSound() {
        cout << "chirp\n";
    }
};
```

```

    }
    void move() {
        cout << "fly\n";
    }
    Bird() {
        cout << "bird hatched\n";
    }
    ~Bird() {
        cout << "bird crashed\n";
    }
};

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

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

```

```

    bird.move();
    animal.move();
    cout << "(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

```

## Erklärungen

A) *Konstrutoren werden Basis vor Derived ausgeführt* Copy führt nur Copy Konstruktor aus (Hier keine Ausgabe) \*Referenz keine Ausgabe

B) *Da nicht virtual werden eigene Aufgerufen* Object Slicing von Hummingbird zu bird. (Bird kennt nur noch eigene Methoden)

- C) \*move ist virtual darum führt Animal Pointer die von Hummingbird aus
- D) *Objekte welche später instanziiert wurden, werden zu erst abgeräumt*
- Derived Deonstruktor vor Base \*Referenzen rufen Deconsturktoren nicht auf

## Abstrakte Klassen

---

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

Abstrakte Funktionen nennt man auch "pure virtual". Wenn man keine Implementation anbietet und dies explizit sagen will: `= 0;`

## Argument Dependent Lookup (ADL)

---

Wenn man Funktionen ausserhalb der Klasse, aber im selben Headerfile nutzt, sollte man die Klasse und die Funktion mit demselben Namespace versehen. Wenn der Compiler eine nicht-qualified Funktion oder einen nicht-qualified Operator auffindet, schaut er sich den Namespace der Typen an, die involviert sind.

```
//adl.h  
  
namespace one {  
    struct type_one{};
```

```

    void f(type_one) {};
}

namespace two {
struct type_two{};
    void f(type_two);
    void g(one::type_one);
    void h(one::type_one);
}
void g(two::type_two);

//adl.cpp
#include "adl.h"

int main() {
    one::type_one t1{};
    f(t1); //one::f
    two::type_two t2{};
    f(t2); // two:f
    //h(t1) wird nicht gefunden
    two::g(t1);
    g(t1); //argument type does not match compile fehler
    g(t2); //ruft g ausserhalb namespace auf
}

```

## Pitfall

Ein Alias wie `using vec = std::vector<int>` führt nicht dazu, dass ADL im Globalnamespace sucht.

```

using vec = std::vector<int>; //is still in std namespace
using outv = std::ostream_iterator<vec>;
using out = std::ostream_iterator<int>;

//This shift operator is in the global namespace
std::ostream& operator<<(std::ostream & os, vec const & v) {

```

```

        copy(begin(v), end(v), out {os, ","});
        return os;
    }

    void work_only_with_shift_in_ns_std(std::ostream &os) {
        std::vector<vec> vv {{1,2,3},{4,5,6}};
        //copy findet den shift operator für std::vector<int> nicht
        copy(begin(vv), end(vv), outv{os, "\n"});
    }

```

Der Namespace std darf nicht verändert werden.

## Lösung

```

namespace X {
    struct vec : std::vector<int> {
        using vector<int>::vector; //ctors
    };
    //OPERATOR here
}

// use vector<X::vec> vv {{1,2,3},{4,5,6}};

```

# Enums

## Syntax

```

enum day_of_week {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
}

```

Alternativ noch mit class

```
enum class day of week (usw.)
```

Unterschied: ohne `class` leaken sie in den umgebenden Scope ( `day = date::Sat` ), am besten als Member einer Klasse genutzt. Mit `class` leaken sie nicht ( `day == date::day_of_week::Sat` ), und der darunterliegende Typ kann spezifiziert werden.

Die Enums starten normalerweise bei 0 und erhöhen sich um 1. Es ist möglich, die Operatoren zu überschreiben.

## Enum Conversion

Operatoren können ebenfalls für enums überschrieben werden

```
enum to int: int day = Sun; int to enum: dayOfWeek day =  
static_cast<dayOfWeek>(1);
```

## Wert festlegen

Mit `=` kann man den Wert festlegen. Folgende erhalten dann einfach den Wert + 1.

**Beispiel Monate** Bei den Monaten will man mit 1 beginnen, und man will auch Kurznamen. Dabei gibt es für Mai (May) aber keinen Kurznamen.

```
enum momth {  
    jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov,  
    january=jan, february, march, april, june=jun, july, aug  
};
```

# Typ festlegen

---

Der darunterliegende Typ kann jeder Ganzzahl-Typ sein (auch bool und char).

```
enum class launch_policy : unsigned_char
```

Wenn man den "Namen" eines Enums wieder will, muss man selber eine Lookup-Table implementieren.

```
enum class launch_policy : unsigned_char {  
    sync = 1,  
    async = 2,  
    gpu = 4,  
    process = 8,  
    none = 0  
}
```

# Contract/Exceptions

---

Wenn eine Funktion ihren Contract nicht erfüllen kann, kann man ein paar Sachen tun. Hier aufgeführt sind nur diese, welche mit C++ besonders Sinn machen oder deren Syntax speziell ist

- Ignorieren
- Standard-Resultat (z.B. anonymous)
  - Macht mehr Sinn, wenn der Aufrufer das Standardresultat im Fehlerfall bestimmen kann
- Error-Wert: Sentinel
- Error Status Side Effect: Parameter oder globale Variable verändern



- Exceptions

# Exceptions

---

Benötigen kein throw, es kann alles geworfen werden was kopierbar ist

```
throw 15; .
```

- Es gibt keine Möglichkeit zu spezifizieren was geworfen werden kann.
- Es gibt keine Checks, ob man eine Exception nicht auffängt.
- Es gibt keine Meta-Informationen
  - kein Stacktrace, keine Quellcode-Position
- Wenn eine Exception geworfen wird während eine Exception nach oben propagiert wird, bricht das Programm ab

Exceptions können natürlich gefangen werden

```
try {  
    ...  
} catch ( type const &e) { // als Referenz  
    ...  
}
```

throw by value, catch by const reference

Reihenfolge ist wichtig, der erste gewinnt. Ein Catch All `catch(...) {}` (das ist tatsächlich die Syntax) muss zuletzt sein.

Es gibt vordefinierte Exception Types in `<stdexcept>`

- `std::logic_error`
  - `std::domain_error`
  - `std::invalid_argument`

- `std::length_error`
- `std::out_of_range`
- `std::runtime_error`
  - `std::range_error`
  - `std::overflow_error`
  - `std::underflow_error`

Man kann als Konstruktorargument immer einen String als Grund angeben. Ebenso gibt es die `.what()` Member Funktion um den Grund zu erfragen

## Exceptions abfragen mit CUTE

---

Mit CUTE kann man die Exception mit

```
ASSERT_THROWS(square_root(-1.0), std::invalid_argument);
```

erfragen

## Templates

---

Templates erlauben es, den eigenen Code an Code zu adaptieren, den es beim Erstellen des eigenen Code noch garnicht gibt.

## Function Templates

---

Die Typen werden bei Function Templates automatisch erkannt! Es ist aber möglich, sie in den spitzen Klammern zu deklarieren um Unklarheiten zu beseitigen.

Header-File:

---

```

namespace MyMin { // namespace nicht zwingend
template <typename T>
T const & min(T const & a, T const & b) {
    return (a < b)? a : b;
}
}

```

Benutzung:

```

using MyMin::min;
using std::cout;
int i = 88;
cout << "min(i,42) = " << min(i,42) << '\n';
double pi = 3.1415;
double e = 2.7182;
cout << "min(e,pi) = " << min(e,pi) << '\n';
std::string s1 = "Hallo";
std::string s2 = "Hallihallo";

/* Es gibt auch std::min und wegen ADL wird ansonsten dieses
cout << "min(Hallo,Hallihallo) = " << MyMin::min(s1,s2)<<'\n';

/* Unterschiedliche Typen. Also festlegen!
Entweder im Template Argument (Template-Ebene) oder beim Aufruf
//min(2,pi); // compile error
min(static_cast<double>(2),pi);
min<double>(2,pi);

/* geht nicht wegen unterschiedlichen Längen
cout << "min(Peter,Toni) = " << min("Peter","Toni") << '\n';

/* ergibt Toni, weil hier Pointerwerte (Adressen) miteinander verglichen werden
Toni kommt später auf den Stack und hat damit auf x86-PCs eine höhere Adresse
cout << "min(Pete,Toni) = " << min("Pete","Toni") << '\n';

/* Lösung, damit tatsächlich Pete herauskommt und wir auch auf String Literals, "Toni"s */

```

```
cout << "min<std::string>(Pete,Toni) = " << min<std::string>
```

Random Fact: C-Style-Strings degenerieren zu einem Pointer, (es sind `char` -Arrays), wenn sie einer Funktion übergeben werden.

## Concepts

Das Ausrechnen der Bedingungen an den Typ nennt sich **Concepts**. Anders als in Java kann man den Typ von T nicht genauer festlegen (z.B. `implements Comparable<T>`). Wenn man sich unseren `min` -Code anschaut erkennt man folgendes

- `operator<(T const &, T const &)` muss definiert sein
- Oder `operator<(T,T)`
- Es muss einen Operator `bool` zurückgeben (wegen `?:` )

## Overloads

Wir mussten beim bisherigen `min` einen Workaround einbauen, wenn wir mit C-Style-Strings hereinkommen. Das bedingt aber das Wissen durch den Aufrufer. Um das zu umgehen, bauen wir einen Overload ein.

Jetziges Header-File:

```
namespace MyMin { // namespace nicht zwingend
template <typename T>
T const & min(T const & a, T const & b) {
    return (a < b)? a : b;
}
char const* min(char const* a, char const* b);
}
```

Und dazu noch ein Implementations-File:

```
#include "MyMin.h"
#include <string>
namespace MyMin {
char const* min(char const* a, char const* b) {
    return std::string(a) < std::string(b) ? a : b;
}
}
```

Sollte nicht zu oft verwendet werden. **Es gewinnt der spezifischste.**

## Variadic Template Function

Wenn man z.B. Funktionen wie printf implementieren will, kennt man die Anzahl Parameter zu Beginn nicht. Der `...`-Operator gehört zur Syntax:

- vor einem Namen: definiert den Namen als Platzhalter für variable Anzahl Elemente
- nach einem Namen: expandiert den Namen zu allen Elementen
- zwischen zwei Namen: definiert den hinteren Namen als Liste von Parametern, vom Typ des vorderen Namen

```
template<typename...ARGS> //any number of types
void variadic(ARGS...args) { //any number of arguments
    println(std::cout, args...); //expand parameters as arguments
}
```

Bei der Implementierung verwendet man die Rekursion

- Base Case ist 0 Argumente

- Rekursiver Fall ist 1 explizites Argument mit einem Schwanz als variadische Liste von Argumenten

```
//base case
void println(std::ostream& out) {
    out << "\n"
}

//Rekursion
template<typename Head, typename... Tail>
void println(std::ostream & out, Head const & head, Tail const &... tail) {
    out << head;
    if(sizeof...(tail)) { // expandiert, wenn sizeof grösser
        out << ", ";
    }
    println(out, tail...); // recurse on tail
}
```

## Class Templates

Bei Klassen-Templates muss das Template-Argument immer spezifiziert werden. Es gibt keine Auto Deduction. **Die Begriffe Template Class und Class Template bedeuten dasselbe.**

Funktionen haben die Class Template-Parameter, können aber auch weitere Parameter haben (also auch Function Templates)

Beispiel Sack:

```
template <typename T>
class Sack
{
    using SackType=std::vector<T>
```

```

    // dependent name: wir verwenden einen Typ gegen aussen,
    using size_type=typename SackType:size_type;
    SackType theSack{};
public:
    bool empty() const { return theSack.empty(); }
    // Weil wir den size_type ausgeben wollen, brauchen wir

    size_type size() const { return theSack.size(); }
    void putInto(T const & item) { theSack.push_back(item); }
};

// Member-Funktion ausserhalb der Template-Klasse müssen inkl.
template <typename T>
inline T Sack<T>::getOut() {
    if(!size()) { throw std::logic_error{"empty Sack"}; }
    // Pick random element
    auto index = static_cast<size_type>(rand() % size())
    T retval{theSack.at(index)};
    theSack.erase(theSack.begin()+index);
    return retval;
}

```

Es ist normal, dass Template Definitionen Typ-Aliase verwenden.  
Neuerdings mit using, früher mit typedef:

- `using newname=existingtype;`
- `typedef existingtype newname;`

Bei Mitgliedern ausserhalb des Class Templates (wie oben `getOut()`):

- Template Intro wiederholen
- Inline um ODR zu gewährleisten

Templates gehören immer ins Header-File! Grund: das .cpp-File kann man auch kompiliert als Objektcode abgeben. Den Header gibt man aber im Source-Code ab. Da der Compiler bei Templates mehr oder minder

Textersetzung macht (...), muss er das Template selber kompilieren können.

Funktionen immer direkt innerhalb der Klasse oder als inline-Funktion im Headerfile.

## Erben in Template Classes

Das Name-Lookup kann verwirrend sein. Immer `this->` oder alternativ `classname::` verwenden, damit man mit der richtigen Klasse spricht.

TODO: evtl. noch anderes von V12 S. 12?

## Spezialisierung

Man kann Class Templates auch spezialisieren (wie Function Templates), zum Beispiel zum Umgang mit Pointern. Es wird auch hier wieder am meisten spezialisierte genommen. Achtung, die Templates müssen nicht miteinander verwandt sein, das ist eigentlich eher Konvention...

```
// forward declaration
template <typename T> class Sack;

// braucht's
template <>
class Sack<char const *> { // Spezialisierung
    // zum Beispiel andere Implementierung
};

template <typename T>
class Sack<T*> { // Spezialisierung verbieten
    ~Sack() = delete; //dekonstruktor löschen um instanzierung
};
```



# Template Terminologie

## Template Definition

```
template <typename T>  
class Sack{...}
```

## Template Declaration

```
template <typename T>  
class Sack;
```

## Class Template Explicit Specialization

```
template<>  
class Sack<char const *> {...}
```

## Template Partial Specialization

```
template <typename T>  
class Sack<T *> {...}
```

## Class Template Member Specialization

```
template<>  
void Sack<char const *>::putInto(char const *p) {...}
```

## Sack mit Initializer List füllen

Wenn man den Sack mit einer Initializer List füllen will (z.B. `Sack<int>`

sack{1, 2, 3}; ), kann man die bereitgestellte

`std::initializer_list<T>` nutzen. Dazu definieren wir einen speziellen Konstruktor

```
Sack(std::initializer_list<T> il):theSack(il){ //theSack is  
}
```

## Sack mit Iteratoren füllen

Konstruktor für Iteratoren begin und end Iteratoren initialisieren.

```
//Konstruktor  
template <typename ITER>  
Sack(ITER b, ITER e) : theSack(b,e){ //theSack is a member v  
}
```

## Container variieren

Man könnte jetzt auch noch den Container variieren. Achtung: hier müssen dann evtl. die Iteratoren angepasst/verallgemeinert werden. Da Container oft mehr als ein Argument haben, machen wir es gleich variadisch. Und damit man nicht immer `std::vector` angeben muss, bauen wir einen Default ein.

Unsere Template-Definition schaut jetzt wie folgt aus (Klammerung beachten!, Keyword `class` ebenso)

```
template <typename T, template<typename...> class container=  
class Sack {  
    //...
```

```
}
```

Und dann z.B.: ``Sack listSack{1,2,3,4,5};`

## Templates als Adapter

Wenn man zum Beispiel einen SafeVector bauen will, der den Vektor implementiert, aber `operator[]` so implementiert dass ein Index Bounday Check stattfindet.

```
template <typename T>
class safeVector : public std::vector<T> {
    using Base=std::vector<T>;
public:
    using size_type=typename Base::size_type;
    using std::vector<T>::vector; // ctors übernehmen, using

    T const & operator[](size_type index) const {
        return this->at(index);
    }

    T & operator[](size_type index) {
        return Base::at(index); // ginge auch this->
    }

    // dann noch front/base
};
```