

# ZUSAMMENFASSUNG

## Lernziele

- Sicheres Beherrschen der wichtigen Sprachelemente und Bibliothekskomponenten von Standard – C++
- Abstraktion von Werten, Algorithmen, Objekte und Verhalten mit den Sprachmitteln von C++
- Grundlagen der funktionalen und generischen Programmierung und von Lambdas
- Beherrschung einer modernen C++ IDE (Cdevelop) mit Unit Testing und Refactoring

## Unterlagen / Bücher

- IFS HSR Wiki
  - o <https://wiki.ifs.hsr.ch/CPlusPlus/wiki.cgi?PlusPlus>
- Unterlagen auf dem [Skripteserver](#)

## Lerninhalte

- Kompilationsmodell, Übersetzungseinheiten, Modulkonzept in C++ mittels Präprozessor, Hello World
- Funktionen und Parameter, elementare Datentypen, einfache Ausgabe und Eingabe mittels Standard-Streams
- Initialisierung, Initialisierungslisten, Typ-Deduktion: const und auto, Referenzen
- Unit Testing mit CUTE, Standard-Exceptions anwenden
- Funktions-contracts
- Werte- und Definitionsbereiche
- Wichtige Typen und Algorithmen der Standardbibliothek: string, vector, copy, transform, Schleifen-Idiome, Range-for
- Anonyme Funktionen: Lambdas, Anwendungen mit Standardalgorithmen
- Einfache eigene Datentypen definieren (struct, class, enum), Typ-Aliase: using und typedef, Sichtbarkeit und Varianten (constructor/destructor)
- Generische Funktionen definieren (function template), Operator-Funktionen, decltype, constexpr, Funktions-Overload-Lookup und Namespaces
- Anwendung der Standardbibliothek mit eigenen Prädikaten und Funktoren, bzw. Lambdas
- Einfache generische Typen definieren (class templates), typename, Template-Aliases, eigene Wrapper für Standardcontainer, einfache Variadic Template Funktionen, static\_assert
- Resourcenmanagement – Lebensdauer, Referenzen(lvalue, rvalue), Move versus Copy, Forwarding, Lambdas mit Capture, Konstruktor-Destruktor-Regeln (=default, =deleted)
- Objektorientierte Programmierung mittels Vererbung und virtual Member Funktionen, Mehrfachvererbung, Mix-in Klassen und virtual Vererbung
- Einführung in weitere Container und Algorithmen der Standardbibliothek (während des ganzen Semesters)
- Objekte auf dem Heap mittels unique\_ptr und shared\_ptr. Problematik zyklischer Objektgeflechte mittels weak\_ptr lösen

<b>INTRODUCTION TO C++ .....</b>	<b>9</b>
WHY C++? .....	9
WHAT YOU WILL LEARN?.....	9
C++ MYTHEN .....	9
C++ WAHRHEITEN .....	9
A SMALL BUT COMPLETE C++ PROGRAM .....	9
C++ ISN'T JAVA .....	9
C++ HELLO WORLD.....	10
C++ SOURCE FILES .....	10
C++ COMPILATION MODEL .....	10
FILES FOR SAYHELLO.....	11
<i>Preprocessor run on main.cpp .....</i>	11
<i>Compiler run on (preprocessed) main.cpp.....</i>	11
<i>Linker creating the main executable.....</i>	11
C++ TERMINOLOGY .....	12
C++ VS. JAVA VARIABLES.....	12
STACK AND HEAP .....	12
SOME BAD CODE EXAMPLES .....	12
<b>C++ FUNKTIONEN UND MODULARIZATION.....</b>	<b>13</b>
DEFINING FUNCTIONS .....	13
DECLARE BEFORE USE .....	13
DECLARING FUNCTIONS.....	13
C++ MODULARIZING CODE .....	14
C++ UNIT TESTING SEPARATE MODULES.....	14
ONE DEFINITION RULE .....	14
A FIRST CLASS – TYPE DEFINITION .....	14
SEMICOLON AFTER }; IN CLASS DEFINITIONS.....	15
TESTS FOR CLASSES IN A LIBRARY .....	15
<b>VALUES AND STREAMS.....</b>	<b>16</b>
DEFINING VARIABLES & NAMEING .....	16
„CONST“ VARIABLES .....	16
WHERE TO PLACE VARIABLE DEFINITIONS?.....	16
TYPES FOR VARIANLES .....	17
<i>Built-in (numeric) types.....</i>	17
<i>Literal Values.....</i>	17
EXPRESSIONS.....	18
ACHTUNG – TYPEN KONVERSIONEN.....	18
ASSIGNMENT OPERATIONS .....	19
LOGIC AND RELATIONAL OPS .....	19
FLOATING POINT NUMBERS.....	19
STRINGS AND SEQUENCES .....	19
<i>Std::vector&lt;T&gt; and ArrayList&lt;T&gt; .....</i>	20
<i>Working with std::string .....</i>	20
<i>Preview References.....</i>	20
<i>Achtung: Sequence of Cells.....</i>	21
<i>„hello is not a std::string“ .....</i>	21
SIMPLE I/O.....	21

<i>Reading a std::string value</i> .....	21
<i>Reading a int value</i> .....	21
<i>Achtung: C develop und EOF</i> .....	22
<i>Robust reading an int value</i> .....	22
<i>An istream's States</i> .....	22
<i>Dealing with Invalid Input</i> .....	22
<i>Formatting Output</i> .....	23
<i>Reading and modifying</i> .....	23
<b>SIMPLE SEQUENCES</b> .....	<b>24</b>
<b>STD::VECTOR&lt;T&gt;</b> .....	<b>24</b>
<i>Overview</i> .....	24
<i>Bad Style Iteration</i> .....	24
<i>Element Iteration</i> .....	25
<b>C++ ITERATORS</b> .....	<b>25</b>
<b>ITERATION WITH ITERATORS</b> .....	<b>25</b>
<b>LOOPS AND ITERATORS</b> .....	<b>26</b>
<i>Example: Counting values</i> .....	26
<i>Special Iterators for I/O</i> .....	26
<b>ALGORITHMEN</b> .....	<b>28</b>
<i>Number of Elements</i> .....	28
<i>Generischer Algorithmus – for_each</i> .....	28
<i>Lambda Expressions</i> .....	29
<i>Finding Elements</i> .....	29
<b>STRUCTURE BEHAVIOR WITH FUNCTIONS</b> .....	<b>30</b>
<b>SCOPES</b> .....	<b>31</b>
<i>Namespaces</i> .....	31
<i>Using Declaration</i> .....	32
<i>Anonymouns Namespace</i> .....	32
<i>Inline Namespaces for Library Versioning</i> .....	32
<b>REFERENCES</b> .....	<b>32</b>
<i>References in C++</i> .....	32
<i>Local References</i> .....	33
<i>Kinds of reference parameters</i> .....	33
<i>Returning References</i> .....	33
<i>Parameter Passing – Returning Results</i> .....	33
<b>FUNCTION OVERLOADING</b> .....	<b>34</b>
<i>Default Arguments</i> .....	34
<i>Overloading Ambiguity</i> .....	34
<b>FUNCTIONS AS PARAMETERS</b> .....	<b>35</b>
<b>FUNCTIONALITY GUARANTEE</b> .....	<b>35</b>
<i>When could a function fail?</i> .....	35
<i>Ignore the error</i> .....	35
<i>Cover Error with a standard result</i> .....	36
<i>Error Value</i> .....	36
<i>Error Status Side Effect</i> .....	36
<i>Exceptions</i> .....	36
<i>Catching Exceptions</i> .....	36
<i>Predefined Exception Types</i> .....	37
<i>Silly Exceptions Example</i> .....	37

Functions with „narrow contract“ .....	37
Testing for Exceptions .....	38
<b>CLASSES AND OPERATOR OVERLOADING .....</b>	<b>39</b>
CLASSES .....	39
<i>A Good class</i> .....	39
<i>Strucute of a class</i> .....	39
<i>Inheritance</i> .....	41
<i>Separation of Declaration and Implementation</i> .....	42
<i>Using the class</i> .....	42
OPERATOR OVERLOADING .....	44
<i>Free Operator</i> .....	44
<i>Member Operator</i> .....	45
<i>Comparison Syntactic Sugar</i> .....	45
<i>Implementing All Comparisons</i> .....	45
<i>Sending Date to std::ostream</i> .....	46
<i>Reading Date from std::istream</i> .....	47
<i>Default Value</i> .....	48
<i>Date in Namespace</i> .....	50
ADL .....	51
<i>ADL Examples</i> .....	51
<i>ADL Issues (Where Intuition Fails)</i> .....	52
<i>Workaround for std::Resolution Problem</i> .....	52
<b>ENUMS .....</b>	<b>53</b>
ENUMERATION TYPES .....	53
SCOPES OF ENUMERATORS .....	53
<i>Unscoped enumeration (no class keyword)</i> .....	53
<i>Scoped enumeration (class keyword)</i> .....	53
ENUMERATION CONVERSION .....	54
DEFINING VALUES OF ENUMERATORS .....	54
OPERATOR OVERLOADING FOR ENUMERATIONS .....	55
SPECIFYING THE UNDERLYING TYPE .....	55
EXAMPLE .....	55
<b>ARITHMETIC TYPES (RING5) .....</b>	<b>56</b>
ARITHMETIC TYPES .....	56
COMPARISON OF RING5 .....	56
OPERATORS FOR RING5 .....	56
<i>Output Operator</i> .....	56
<i>Plus Operation</i> .....	57
<i>Multiplication Operation</i> .....	57
<i>Mixed Arithmetic</i> .....	57
<b>STANDARD CONTAINERS STL .....</b>	<b>59</b>
CATEGORIES OF STL CONTAINERS .....	59
COMMON FEATURES OF CONTAINERS .....	59
COMMON CONTAINER API .....	59
COMMON CONTAINER CONSTRUCTORS .....	60
STL ITERATOR CATEGORIES .....	60
<i>Input Iterator</i> .....	60

<i>Forward Iterator</i>	61
<i>Bidirectional Iterator</i>	61
<i>Special Case: Output Iterator</i>	61
<i>Why these iterator categories?</i>	61
<i>&lt;iterator&gt; functions</i>	61
<b>SEQUENCE CONTAINERS</b>	62
<i>Array</i>	62
<i>Double-ended Queue – std::deque</i>	62
<i>Double-linked List: std::list</i>	63
<i>Singly-linked List: std::forward_list</i>	63
<i>LIFO Adapter - std::stack</i>	63
<i>FIFO Adapter - std::queue</i>	64
<i>Adapter - std::priority_queue</i>	64
<i>Example Stack and queue</i>	64
<b>ASSOCIATIVE CONTAINERS AKA: SORTED (TREE-) CONTAINERS</b>	65
<i>Set of element – std::set</i>	65
<i>Map – std::map&lt;key, value&gt;</i>	65
<i>Multiset / Multimap</i>	66
<b>HASHED CONTAINERS – UNSORDERED_SET</b>	67
<i>Example – unordered_set</i>	67
<i>Example – unordered_map</i>	67
<b>STL ALGORITHMS</b>	68
<b>WHY SHOULD WE USE ALGORITHMS?</b>	68
<b>ALGORITHM BASICS</b>	68
<i>Iterators for Ranges</i>	68
<i>Iterators as Output of Ranges</i>	69
<i>Algorithm-for_each</i>	69
<i>Reading Algorithm Signatures</i>	69
<i>Functor</i>	70
<i>Predicates</i>	70
<i>Algorithm Examples</i>	70
<i>Heap Algorithms</i>	73
<b>FALLSTRICKE</b>	75
<i>Mismatching Iterator Pairs</i>	75
<i>Reserving Enough Space</i>	75
<i>Wenn Sie einen Iterator zur Spezifizierung der Ausgabe für einen Algorithmus verwenden, müssen Sie sicherstellen, dass genügend Speicherplatz zugewiesen wird.</i>	75
<i>Insert Iterators</i>	76
<i>Input Validation</i>	76
<b>ALGORITHM HEADERS</b>	76
<i>In the Standard Library</i>	76
<i>Non-Modifying Sequence Operations</i>	77
<i>Mutating sequence operations</i>	77
<i>Sorting and Related Operations</i>	77
<b>FUNCTORS AND PARAMETERIZING STL</b>	78
<b>FUNCTION OBJECTS</b>	78
<i>Example Functor with memory</i>	78
<b>FUNCTION/FUNCTOR TERMINOLOGY</b>	78
<b>GENERATOR FUNCTORS</b>	79

LAMBDA FUNCTION SYNTAX .....	79
<i>Recap Lambda Rules</i> .....	79
<i>Lambda Captures</i> .....	79
<i>Special Case mutable</i> .....	80
<i>Lambdas are functors</i> .....	80
<i>Lambdas in Member Functions</i> .....	80
STANDARD FUNCTOR TEMPLATE CLASSES .....	80
<i>Standard functor classes</i> .....	81
PARAMETRIZE ASSOCIATIVE CONTAINERS .....	81
<i>Example set&lt;string&gt; dictionary</i> .....	81
HOW TO KEEP FUNCTORS AROUND?.....	82
STD::FUNCTION<SIGNATURE> .....	82
<i>Example using std::function</i> .....	82
<i>Std::function with member functions</i> .....	83
WHAT IS A MEMBER-(FUNCTION) POINTER?.....	83
<i>Example Pointer to Members.*</i> .....	83
STD::FUNCTION CALLING MEMBER FUNCTION .....	83
CREATING YOUR OWN ITERATOR TYPES.....	84
<i>Example using Boost Iterator</i> .....	84
<b>FUNCTION .....</b>	<b>85</b>
MOTIVATION FOR TEMPLATES.....	85
<i>Template Myths</i> .....	85
WHY FUNCTION TEMPLATES .....	85
SPECIFY A FUNCTION TEMPLATE .....	86
<i>Using our MyMin::min() template</i> .....	86
TEMPLATE ARGUMENT'S CONCEPTS .....	86
<i>Concepts for min()'s T</i> .....	87
<i>What is the concept for max()'s T?</i> .....	87
FUNCTION TEMPLATE ARGUMENT DEDUCTION .....	87
<i>Surprising type deduction</i> .....	87
FUNCTION TEMPLATE OVERLOADS .....	88
FUNCTION OVERLOADS AND TEMPLATES .....	88
<i>Caution: Overloading function templates</i> .....	88
PROBLEM – VARYING NUMBER OF ARGUMENTS.....	88
<i>Defining a variadic template function</i> .....	89
<i>Implementing a variadic function</i> .....	89
<b>TEMPLATE CLASSES .....</b>	<b>90</b>
TEMPLATE CLASS – CLASS TEMPLATE .....	90
TEMPLATE CLASS SACK<T>.....	90
<i>Type aliases</i> .....	90
<i>Typename for dependent names</i> .....	91
<i>What Concept for Sack's T?</i> .....	91
<i>Members outside of class template</i> .....	91
TEMPLATE CLASS RULES .....	91
DEMO STATIC MEMBER VARIABLES IN TEMPLATE CLASS .....	92
CLASS TEMPLATE GOTCHAS .....	92
CLASS TEMPLATES THAT INHERIT .....	93
WHAT ABOUT A SACK WITH POINTERS? .....	93
<i>Template (partial) Specialization)</i> .....	93

<i>How to avoid creating a Sack&lt;int*&gt;.....</i>	93
<i>A Sack&lt;char const *&gt;.....</i>	94
TEMPLATE TERMINOLOGY .....	94
EXTENDING OUR STACK<T> .....	95
<i>Construct a Sack from Iterators .....</i>	95
<i>Surprise: Creation allows two ints.....</i>	95
<i>Extracting a std::vector from Sack.....</i>	96
<i>Filling a Stack with std::initializer_list&lt;T&gt; .....</i>	96
<i>Factory for Sack from an Initializer.....</i>	96
<i>More on using std::initializer_list&lt;T&gt;.....</i>	97
<i>Varying Sack's Container .....</i>	97
TEMPLATE TEMPLATE PARAMETERS.....	97
TEMPLATE-TEMPLATE ARGUMENT FACTORY.....	98
ADAPTING STANDARD CONTAINERS.....	98
<i>Example SafeVector&lt;T&gt; .....</i>	98
<b>MAIN (argc, argv) .....</b>	<b>99</b>
ÜBERGEBEN VON PROGRAMMARGUMENTEN .....	99
«NACKTE» ARRAYS ALS PARAMETER.....	99
<i>Pointer zu Arrays sind Random-Access Iteratoren .....</i>	99
<i>Initialisieren eines Arrays .....</i>	99
<i>Erschliessen der Array Grössen.....</i>	100
<i>Zusammenfassung «Nackte» Arrays .....</i>	100
<b>DYNAMIC HEAP MEMORY MANAGEMENT .....</b>	<b>101</b>
HEAP MEMORY .....	101
C++ HEAP MEMORY BASICS .....	101
<i>C++ Richtige Heap Memory Nutzung .....</i>	101
STD::UNIQUE<PTR> .....	101
<i>Ressourcenmanagement mit unique_ptr&lt;T&gt;.....</i>	102
<i>Für C Pointer .....</i>	102
<i>Richtlinien für unique_ptr.....</i>	102
SHARED_PTR<T> + MAKE_SHARED<T>() .....	102
<i>Für was sind Sie? .....</i>	103
<i>Ressourcenmanagement mit shared_ptr&lt;T&gt; .....</i>	103
<i>Klassenhierarchien mit shared_ptr&lt;T&gt; .....</i>	103
<i>Interessante Seiteneffekte.....</i>	104
<i>Zusammenfassung von shared und weak Pointers .....</i>	104
BEISPIEL ZU POINTERS (ZEIGERN) – TEILE DER ERSTEN BIBLISCHEN FAMILIE .....	104
<i>Schritte und Effekte .....</i>	104
<i>Erster Schritt auf dem shared_ptr .....</i>	105
<i>Zweiter Schritt, töten von Abel .....</i>	105
<i>Das Problem, dass der shared_ptr nicht zugreifbar ist .....</i>	106
<i>Dritter Schritt, Adam kann sterben .....</i>	107
<i>Das Problem der zyklischen Objektreferenzen .....</i>	107
<i>Vierter Schritt, alle können sterben .....</i>	108
<b>INHERITANCE AND DYNAMIC POLYMORPHISM .....</b>	<b>109</b>
GRÜNDE FÜR VERERBUNG .....	109
VERERBUNG FÜR DYNAMISCHE BINDUNG .....	109
VERERBUNGSSYNTAX .....	109

Syntax für Mehrfachvererbung (Nicht Prüfungsstoff) .....	109
Initialisierung der Basisklassen.....	110
Basisklasse «ctor» bevor der Member Initialisierung.....	110
Wann Vererbung schlecht ist.....	110
VERERBUNG UND SICHTBARKEIT .....	110
Sichtbarkeitsregeln.....	111
DYNAMISCHER POLYMORPHISMUS .....	111
Wichtig .....	111
Wann wird es in C++ gebraucht.....	111
Hierarchie von iostremas (Vereinfacht) .....	111
Potentielle Probleme mit Vererbung und Pass by Value.....	112
Member Hiding Problem .....	112
Virtuelle Member Funktionen.....	113
Beispiel zu Vererbung und Dynamischer Polymorphismus.....	114
Abstrakte Basis Klassen – Pure Virtual.....	115
Gebrauch von Polymorphen Klassen .....	116
Wieso brauchen wir das «virtual»-Schlagwort?.....	116
DESIGN GUIDELINES.....	116
Vererbung und Dynamischer Polymorphismus.....	116
Regeln zum Überschreiben von «virtual» Members .....	116

# Introduction to C++

## Why C++?

C++ basiert auf einem ISO Standard. Aktuell ist C++14 in Verwendung. Teile davon wurden an der HSR entwickelt. C++ funktioniert auf nahezu allen Plattformen. Von Mikrocontrollern bis hin zu Mainframes. Die Grundlagen für C++ können ganz leicht auch auf andere Programmiersprachen adaptiert werden.

## What you will learn?

- Modern C++ usage
- Effective use of Standard Library
- Modern Unit Testing with CUTE
- Using Cdevelop

## C++ Mythen

- C++ sei weniger effizient als C
- C++ setzt voraus, dass man Pointers richtig verwendet.
- C++ Programme stürzen ab und ist ein Sicherheitsleck
- C++ haben Memory Leaks
- Klartexteditoren sind die besten Editoren zur Bearbeitung von Code
- C++ Code kann nicht mittels Unit Tests getestet werden.

## C++ Wahrheiten

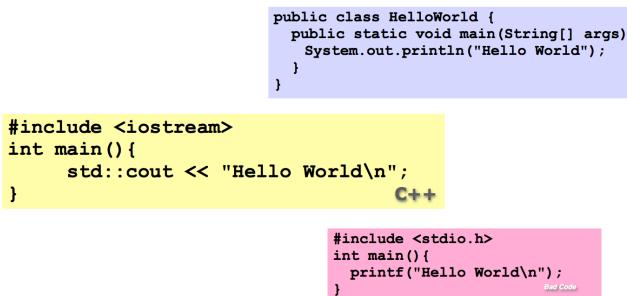
Im C++ Standard gibt es ein „undefined behavior“ definiert. Die Compiler Nachrichten können teilweise sehr schwierig zu verstehen sein. Grundsätzlich ist es auch möglich C zu programmieren und es C++ zu nennen. Gegenüber anderen Sprachen wie Java oder .NET gibt es in C++ keine Garbage Collection. Es wird daher nicht aufgeräumt. Zuletzt ist zu sagen, dass der IDE Support noch nicht gleich gut wie bei Java ist.

## A small but complete C++ program

Die Funktion `int main(){}` ist die Programms Einstiegs Funktion. C++ stellt Funktionen zur Verfügung. (nicht so wie Java, die Methoden hat). Es ist zu beachten das nicht alle Funktionen auf ein Objekt gebunden sind. Die Rückgabetypen müssen vor den Funktionsnamen in den Deklarationen angegeben werden. Die Klammern () nach dem Namen markieren eine Funktion. Die {} Klammern markieren den Funktionsbody.

## C++ isn't Java

Die Syntax ist sehr ähnlich im Vergleich zu Java. Java hat sie von C++ gestohlen, welche Sie von C gestohlen hat. Weitere Vergleiche sind in den Self-Study Unterlagen zu finden.



The image shows a code editor interface with three panes. The top pane contains Java code:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

The bottom-left pane contains C++ code:

```
#include <iostream>
int main(){
    std::cout << "Hello World\n";
}
```

The bottom-right pane contains C code:

```
#include <stdio.h>
int main(){
    printf("Hello World\n");
}
```

## C++ Hello World

Was ist hier falsch? (Siehe Bild)

```
=====
// Name      : helloworld.cpp
// Author    :
// Version   :
// Copyright : Your copyright notice
// Description: Hello World in C++, Ansi-style
=====

#include <iostream>
using namespace std;           bad practice, very bad in global scope

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
    return 0;                         redundant
}                                     using global variable! really bad :-(

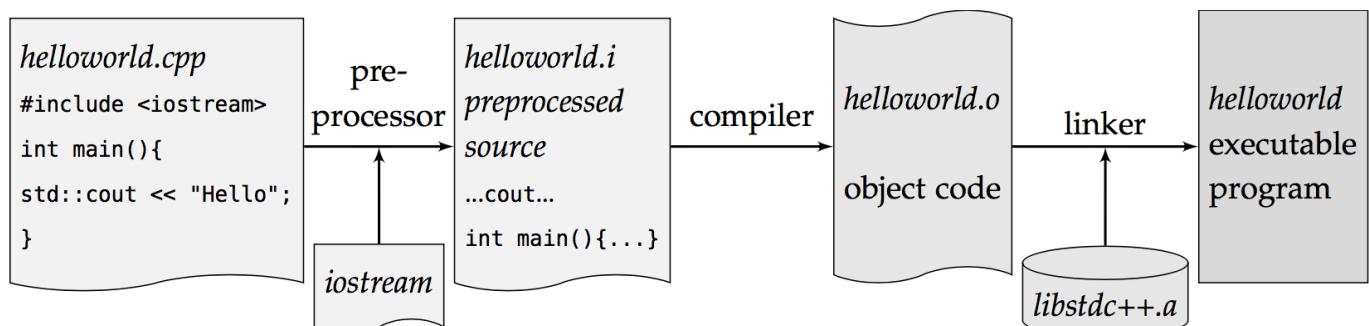
belongs into version management system
ridiculous comment
inefficient, redundant
```

## C++ source files

\*.cpp Dateien sind für den Quellcode. Hauptsächlich ist es als Implementations-Datei angedacht. Es sind darin Funktionsdefinitionen zu finden. Zudem bildet es die Quelle für die Kompilation.

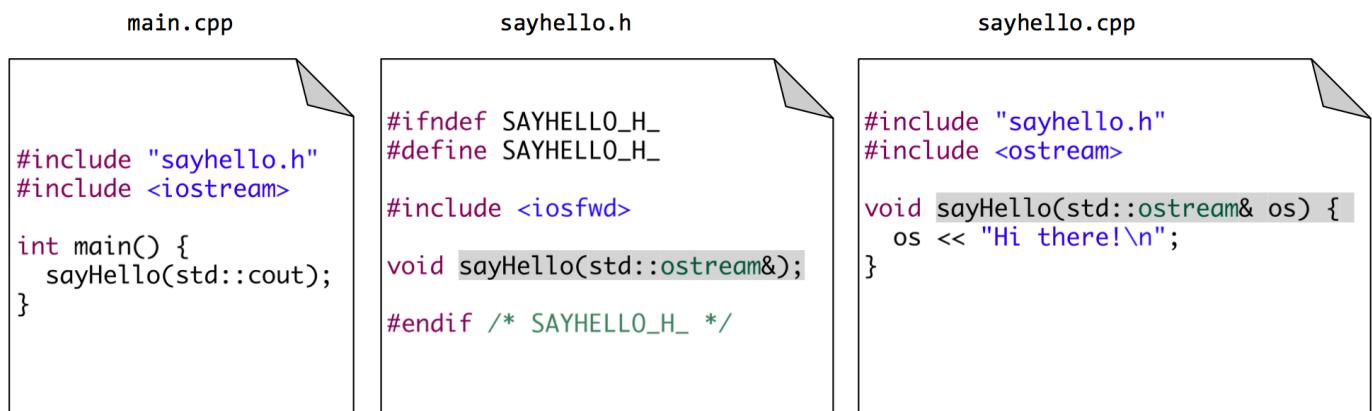
\*.h sind Dateien für Interfaces und Templates. In den Dateien sind Deklarationen und Definitionen, welche in den Implementationsdateien verwendet werden. Eine textuelle Inclusion erfolgt durch den Preprozessor. Die Dateien werden über #include „header.h“ eingebunden. Wie in C.

## C++ compilation model

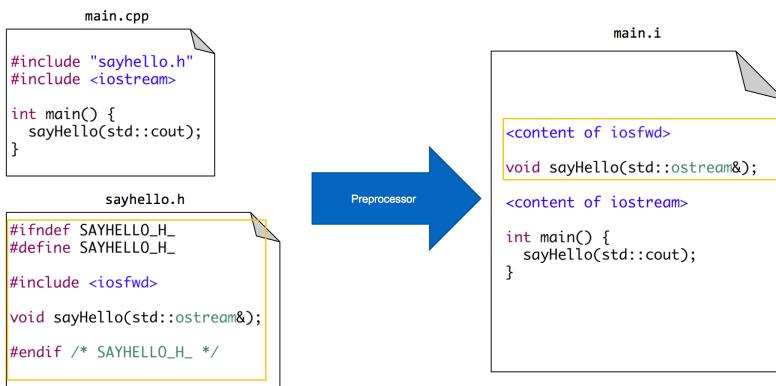


C++ Code wird normalerweise in Maschinen-Code kompiliert. Dabei gibt dabei keinen JVM Overhead. Das Modell besteht aus drei Phasen: Processor, Compiler und Linker. Die Bibliotheken (Libraries) stellen wiederverwendbare Module von Binary Code zur Verfügung.

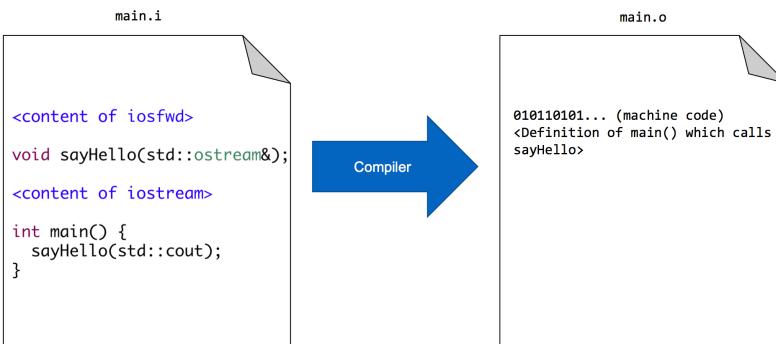
## Files for sayhello



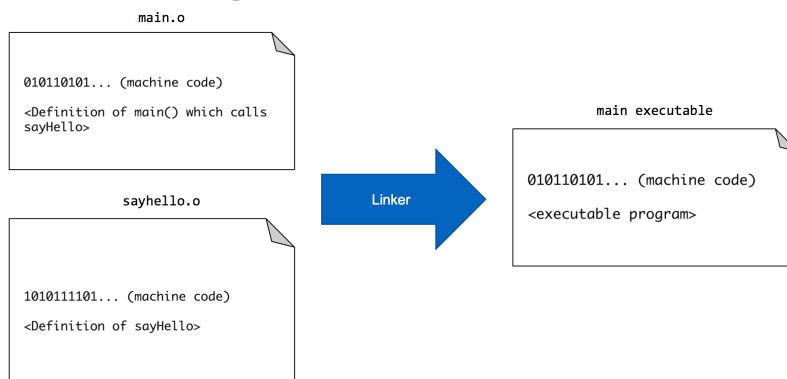
### Preprocessor run on main.cpp



### Compiler run on (preprocessed) main.cpp



### Linker creating the main executable



## C++ Terminology

**Value** (Abstrakte) Elemente von einem bestimmten Typ – 42

**Type** Range von Werte und ihr Verhalten – int

**Variable** Bezeichneter Werthalter von einem bestimmten Typ – int const i{42};

**Expression** Errechnet einen Wert von einem bestimmten Typ – (2+4)\*7

**Statement** Ausführbarer Teil von einer Funktion – while(true);

**Declaration** Führt einen Namen und den dazugehörigen Typ ein – int foo();

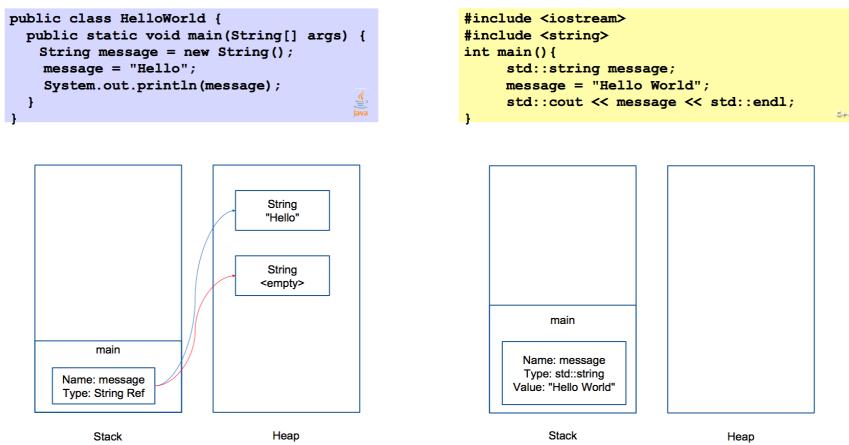
**Definition** Definiert einen Namen und was es ist – int J;

**Function** Eine Ausführbarer Teil, welcher aufgerufen werden kann – void bar(){};

## C++ vs. Java Variables

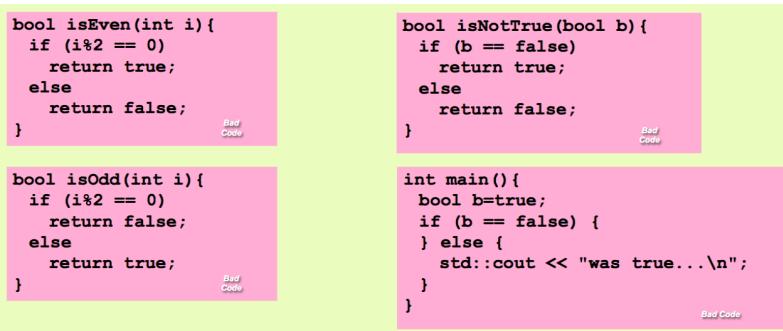
Die Java Objekt Variablen sind auf dem Heap abgespeichert bis auf die primitiven Typen (int, string, char, boolean). In C++ wird Memory bei der Definition der Variablen bereitgestellt. C++ benötigt keine explizite Heap Verwendung.

## Stack and Heap



## Some Bad Code Examples

Nutze niemals folgende Boolean-Ausdrücke im Code. In der Prüfung gibt es dafür sogar Abzüge (auch in Java).



# C++ Funktionen und Modularization

## Defining Functions

**Schema für eine Funktion**      return\_type function(parameter) { /\*body\*/ }

Bei int main() handelt sich um eine spezielle Funktion. Sie hat ein implizites return Statement, welches 0 zurückgibt, wenn keiner mitgegeben wird.

Alle anderen Funktionen mit einem non-void Typ müssen einen Wert zurückgeben. Als Beispiel **sayhello.cpp**.

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

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

## Declare before Use

Alle Dinge mit einem Namen, welche in einem C++ Programm benutzt werden, müssen deklariert werden, bevor Sie benutzt werden können. So zum Beispiel eine Funktion welche ich aufrufe, einen Typ von einer Variable oder einer Variable, welche ich benutze. Für Bibliothekscode werden solche Deklarationen in header files ausgelagert.

#include <iostream> um std::cout zu nutzen.

## Declaring Functions

**Schema für eine Funktion**      return\_type function(parameter);

Der Funktionsbody wird dabei weggelassen. Solche Deklarationen werden häufig in header files ausgelagert. Der Preprocessor bietet einen #include-Schutz, dieses ist nötig wenn der Header Typendefinitionen erhält. Als Beispiel **sayhello.h**.

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_

#include <iostream>
void sayHello(std::ostream &out);

#endif /* SAYHELLO_H_ */
```

## C++ Modularizing Code

Die Bibliotheksfunktionen einer separaten Zusammenstellung erlauben Unit Testing. Die Deklaration findet in header files statt. Die Verwendung von solchen Funktionen setzt einen Include voraus.

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

#ifndef SAYHELLO_H_
#define SAYHELLO_H_

#include <iostream>
void sayHello(std::ostream &out);

#endif /* SAYHELLO_H_ */

#include "sayhello.h"
#include <iostream>
int main(){
    sayHello(std::cout);
}
```

## C++ Unit Testing separate Modules

- CUTE library test project

```
#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"
#include "sayhello.h"

void testSayHelloSaysHelloWorld() {
    std::ostringstream out{};
    sayHello(out);
    ASSERT_EQUAL("Hello, world!\n", out.str());
}

void runAllTests(int argc, char const *argv[]){
    cute::suite s;
    //TODO add your test here
    s.push_back(CUTE(testSayHelloSaysHelloWorld));
    cute::xml_file_opener xmlfile(argc, argv);
    cute::xml_listener<cute::ide_listener> lis(xmlfile.out);
    cute::makeRunner(lis, argc, argv)(s, "AllTests");
}

int main(int argc, char const *argv[]){
    runAllTests(argc, argv);
    return 0;
}
```

- C++ library project

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_

#include <iostream>
void sayHello(std::ostream &out);

#endif /* SAYHELLO_H_ */

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

32

## One Definition Rule

Während ein Programmelement mehrmals deklariert werden kann ohne Probleme, darf es nur einmal definiert werden.

### Konsequenzen

- Es darf nur eine main() Funktion haben
- Es müssen Definitionen für alle Elemente vorhanden sein, welche benutzt werden.
- #include-Schütze sind von Vorteil.

## A first class – Type definition

Die „**struct**“ und „**class**“ Stichworte sind äquivalent. **Struct** bedeutet public, während **class** für private zu verwenden ist. Die Sichtbarkeit ist nicht für jedes einzelne Element zu definieren.

```
#ifndef HELLO_H_
#define HELLO_H_
#include <iostream>

struct Hello {
    void sayHello(std::ostream &out) const;
};

#endif /* HELLO_H_ */
```

## Implementationsnamen einer Memberfunktion mit einem Klassennamen

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

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

## Objektinstanzierung und Memberfunktionsaufruf

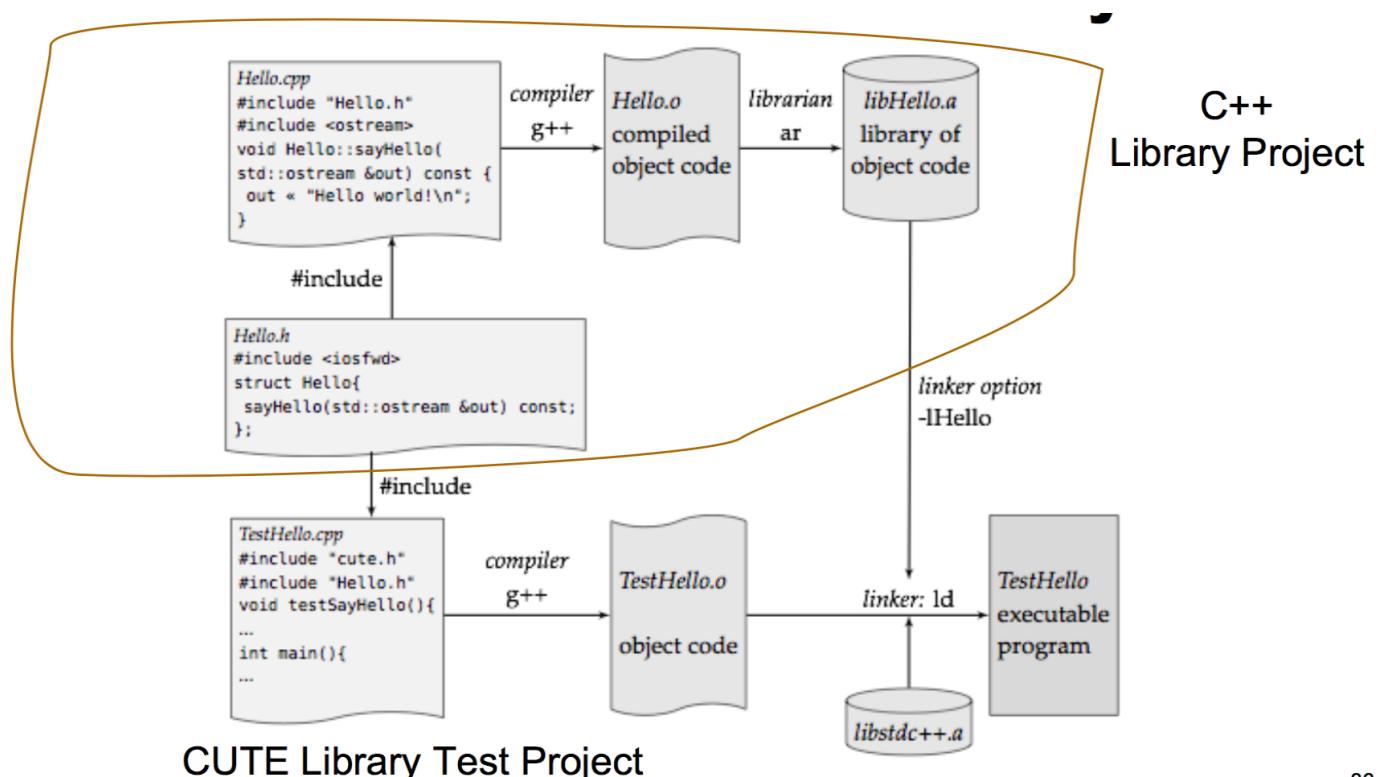
```
#include "Hello.h"
#include <iostream>

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

## Semicolon after }; in class definitions

Wenn man das **Semicolon** nach einer Klassendefinition weglässt, werden sehr „strange“ Fehlermeldungen angezeigt. Meist sind sie meilenweit vom vergessenen **Semicolon** entfernt.

## Tests for Classes in a Library



# Values and Streams

## Defining Variables & Nameing

```
<type> <name> {<value>};
  int anAnswer{42};
  int const zero{};
```

bleibt unveränderlich

Eine Definition einer Variable besteht aus dem Typ, dem Namen und einem initialen Wert.

Initialisierungen für non-const Variablen können weggelassen werden, dies ist aber bad Practice und in einigen Fällen auch gefährlich.

Leere {} bedeuten heißt, dass eine Default Initialisierung stattfindet.

Wenn wir = bei der Initialization nutzen können wir den Compiler entscheiden lassen, welchen Typ es hat.

```
auto const i = 5;
```

Namen sollten grundsätzlich klein beginnen. Der Name sollte beschrieben, für was Sie gebraucht wird. Abkürzungen sollten, wenn immer möglich vermieden werden. One Letter Namen sollen nur innerhalb von Iterationen verwendet werden.

## „const“ Variables

Type immer links von const angeben

```
int const theAnswer{6*7};
```

Wert muss unbedingt angegeben werden

Das Hinzufügen des const-Schlüsselwortes vor dem Namen macht die Variable eine Einzelzuweisungsvariable, aka, einer Konstante. Das Schlüsselwort const wird auch in anderen Kontexten angezeigt. Es bezeichnet etwas Unveränderliches. Einige Konstanten müssen zur Kompilierzeit gefixt sein. Um das zu erreichen kann das Schlüsselwort constexpr genutzt werden.

```
double constexpr pi{3.14};
```

Grundsätzlich sollte const wenn immer möglich genutzt werden. Viel Code brauchen Werte, die in den meisten Fällen aber gar nicht geändert werden. Es hilft zudem dabei „Bad Practices“ zu verhindern. Zum Beispiel die selbe Variable für verschiedene Zwecke zu verwenden. Mit der Verwendung von „const“ wird der Code „sicherer“.

## Where to place Variable definitions?

- So spät als möglich
  - o Alles am Anfang zu definieren, ist schon lange obsolet.
- Scoping Regeln sind ähnlich wie bei Java. Eine Variable die innerhalb eines Blocks definiert ist, ist unsichtbar nach dem Ende.
  - o Versuchen Sie aber Namenskonflikte zu vermeiden. Kleine gleichen Namen innerhalb eines Blockes.
- Keine globalen Variablen (Speicher) machen. Das ist ein Designfehler. Wann dann nur const.
-

## Types for Variables

C++ hat eine Reihe von built-in Typen. Dies meisten davon sind Nummer. Void ist eine Ausnahme. Es ist ein Typ ohne Werte. Bool und char werden ebenfalls als ganze Zahl behandelt.

Zudem bietet die Standardbibliothek eine Vielzahl von Typen für verschiedene Zwecke (definiert als Klasse). Zum Beispiel std::string oder std::vector.

### Built-in (numeric) types

- These are usable without any header `#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`
- Note: there are no "unsigned" floating points
- Plus some more not relevant now

alles was ohne include möglich ist.

`unsigned` sind nur positiv und lassen keine negativen Werte zu.  
`double` ist standard für die Fließkommazahlen

### Literal Values

- Literal
  - Type? Value (in decimal)?
- `'a', '\n', '\x0a'` `char`
  - `1, 42L, 5LL, int{}` `int` Wert Typ  
long long long long
  - `1u, 42ul, 5ull` `unsigned long long`  
unsigned unsigned long
  - `020, 0x1f, 0xFF` `octal (0)` `hex(0x)` hex als unsigned long long
  - `0.f, .33, 1e9, 42.E-12L, .31` `float` `double` `10 hoch` double oder ein long double (wegen L am Ende)
  - `"hello", "\012\n\"` character arrays
  - `R"(\root.hsr\skripte\)"`
  - `"hiho"s, std::string{}`  
String Literal wird zu einem std::string{}

**Arithmetic**

links oder rechts

- binary: + - \* / %(modulo)

nur auf einer seite

- unary: + - **++ --**

**Logic****prefix/postfix**

gleich wie JAVA

- ternary: ?: was bedeutet dies genau?

- binary: **&&** and **||** or links oder rechts

- unary: ! not nur auf einer seite

**Bit-operators**

- binary: & | ^ << >>

- unary: ~ compl

**What are the values?**

ganz divison, als keine Kommazahl

- **(5 + 10 \* 3 - 7 / 2)**

automatische typen bestimmung

- **auto x = 3 / 2;** ==> 1

- **auto y = x % 2 ? 1 : 0;**

**shortcut evaluation**

falls erster Teil falsch, wird der zweite Teil gar nicht angewendet.

- use unsigned types for these!

- **bitand bitor xor** are & | ^

**Achtung – Typen Konversionen**

C++ bietet eine automatische Typenkonversion, wenn Werte von verschiedenen Typen in einer Expression kombiniert werden. So wird aus 45.0 / 8 beispielsweise 5.625(hinterer Teil wird umgewandelt).

- **double x = 45/8;**

in einfachen Fällen wird eine Konvertierung, aber erst zum Schluss  
also 5 oder 5.0

Die Division von Integers runden nicht. Division mit Null führt zu undefined behavior. Dies ist ebenfalls bei der Operation mit Modulo Null der Fall.

```
/ ergibt inf
-x / ergibt -inf
0.0 / 0 ergibt NaN.
```

left: lvalue → **x = 6 \* 7;** right: rvalue

Eine Zuweisung braucht eine Variable auf der Linken Seite (lvalue). Auf der rechten Seite ist das rvalue ( $6*7 = 42$ , wäre daher ein nicht möglicher Fall). Die meisten binären Operatoren können zu kürzeren Anweisungen kombiniert werden um den Code kürzer zu machen.

• **a += b; c /= d; x >>= 2;**

Ein Increment/Decrement ist nur auf einem lvalue möglich.

• **a++; ++b; 5++**

Variablen können inkrementiert werden, Zahlen nicht.

## Logic and Relational Ops

Relationale Operatoren vergleichen die Werte. Dazu zählen  $<$   $>$   $\leq$   $\geq$   $=$  und  $\neq$ . Als Werte geben sie true oder false zurück. Alles was ungleich 0 ist, ist true.

Logische Operatoren und bedingte Anweisungen sind großzügig numerische Werte als Aussage der Wahrheit zu akzeptieren.

• **if (5);  
while (1);  
std::cout << (!x%2 ? "even" : "odd ");**

Precedence beachten. ! greift in diesem Fall vor dem Modulo ==> en.cppprecedence.com/net  
In diesem Fall ist eine Klammer nötig.

**if(a < b < c)...** compiles, but what does it mean?

In diesem Fall kommt es auf die Reihenfolge an. Daher wird von links nach rechts weitergefahrt. a < b wird zuerst ausgewertet und im Anschluss noch mit < c.

14

## Floating Point Numbers

Wenn immer möglich sollten doubles genutzt werden. Sie nutzen die aktuelle Hardware am effizientesten und sind der Standard für Literale. Float sollte nur gebraucht werden, wenn die Memory Verbrauch oberste Priorität hat. Zum Beispiel bei sehr grossen Datensätzen.

Es ist anzumerken, dass es auch legale double Werte gibt, welche keine Nummern sind. NaN, +Inf und -Inf. NaN ist nirgendswo genau gleich mit etwas anderen, es ergibt daher immer false.

Floating Points mit == zu vergleich ist grundsätzlich falsch. Mit dem CUTE ASSERT\_EQUAL(exp,act) wird automatisch ein Delta zur Verfügung gestellt welches den Vergleich unterstützt.

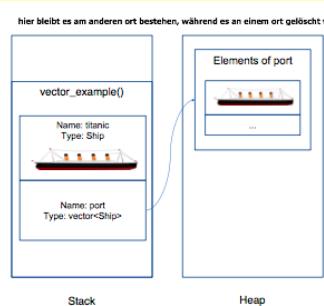
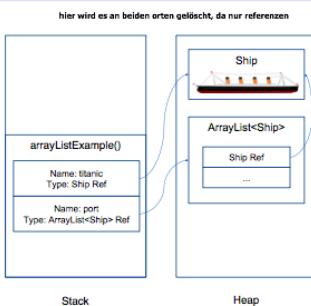
## Strings and Sequences

Std::string ist der C++ Typen für die Darstellung von Sequenzen von chars. (only 8 bit). Der Unicode Support ist anders gegenüber von Java. Ketten wie „ab“ sind nicht von diesem Typ aber „ab“'s ist es (setzt using namespace std::literals voraus).

Std::vector ist ein homogener Behälter, der eine Folge von Werten des Typs darstellt. Fast alle Typen können ein Vectorelement T sein. Es ist nicht nötig den Platz für individuelle Elemente zu reservieren. C++ hat Kopien von Elementen und nicht Referenzen.

```
public class SomeClassWeActuallyDontNeed {
    public static void arrayListExample() {
        Ship titanic = new Ship("RMS Titanic");
        ArrayList<Ship> port = new ArrayList<>();
        port.add(titanic);
    }
}
```

```
#include "Ship.h"
#include <vector>
void vector_example() {
    Ship titanic("RMS Titanic");
    std::vector<Ship> port();
    port.push_back(titanic);
}
```



## Working with std::string

```
#include <iostream>
#include <string>
void askForName(std::ostream &out) {
    out << "What is your name? ";
}
std::string inputName(std::istream &in) {
    std::string name{}; initialisieren
    in >> name; einlesen, in geht in name ein.
    return name; Variable name zurückgeben
}
void sayGreeting(std::ostream &out, std::string name) {
    out << "Hello " << name << ", how are you?\n";
}
int main() {
    askForName(std::cout);
    sayGreeting(std::cout, inputName(std::cin));
}
```

## Preview References

std::string und eingebauten Typen stellen Werte dar. Diese können kopiert werden. Es ist keine explizite Speicherzuweisung nötig um chars abzuspeichern.

Einige Objekte sind eine Werte, weil sie nicht kopiert werden können. Zum Beispiel die Streams für I/O.

Die Funktionen müssen das Stream Objekt als Referenz übernehmen, da sie ja einen Effekt auf den Stream haben wollen.

Referenzparameter sind mit einem & markiert. Im Gegensatz zu C++ werden in Java alle Objekte als Referenzen weitergegeben.

## Achtung: Sequence of Cells

Statements werden bei einem ; (Semicolon) sequenziert. Innerhalb einer einzelnen Expression, wie zum Beispiel einem Funktionsaufruf ist die Bewertung einer Sequenz nicht definiert.

```
void sayGreeting(std::ostream &out,
                  std::string name1,
                  std::string name2){
    out << "Hello " << name1 << ", do you love "
        << name2 << "?\n";
}

int main() {
    askForName(std::cout);
    sayGreeting(std::cout, inputName(std::cin),
                inputName(std::cin));
}

```

Verhalten ist nicht ganz bekannt.  
die Reihenfolge der Aufrufe von inputName ist nicht genau bekannt.



„hello is not a std::string“

```
auto s = "hello"; using namespace std::literals;
auto s = "hello"s;
```

Char Array  
==> wichtig

Wegen C, haben normale C++ String nicht den Typ std::string. Dies ist kein Problem, ausser wenn man auto nutzt um die Variable zu definieren.

Am besten std::String nutzen um auf der sicheren Seite zu sein.

```
auto t = std::string{"Peter"};
std::cout << s << t ;
```

## Simple I/O

Reading a std::string value

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

Ein Lesen von einem std::string kann nicht fehlschlagen, es sei denn der Stream ist nicht bereits nicht gut (!good()).

Reading a int value

```
int inputAge(std::istream& in) {
    int age{-1};
    if (in >> age) Implizite Abfrage des Zustandes von .good()
        return age;
    return -1;
}
```

Es gibt keine Fehlerbehebung. Ein falscher Input setzt den Stream auf den Status fail. Die Charaters bleiben auf dem Input.

## Achtung: Cdevelop und EOF

```
#include <iostream>
int main() {
    size_t count{0};
    char c{};
    while (std::cin >> c) ++count;
    std::cout << count << "\n";
}
```

```
$ mycharcount < input.txt
42
$ mycharcount
12345
<CTRL-D>
6
$ running in a shell/Terminal also
works and allows input redirection
```

Wenn ein Programm geschrieben wird, welches den ganzen Input nimmt, muss es mit Ctrl-D (Linux) oder Ctrl-Z (Windows) terminiert werden. Dazu muss vielleicht der Focus neu auf die Cdevelop Console gesetzt werden. Um die aktuelle Linie zu senden, muss Enter gedrückt werden.

### Robust reading an int value

```
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;
}
```

- Read a line and parse it as an integer until OK or EOF
- Use an `istringstream` as intermediate stream

### An `istream`'s States

bit	query	entered
<none>	<code>is.good()</code>	initial, <code>is.clear()</code>
failbit	<code>is.fail()</code>	formatted input failed
eofbit	<code>is.eof()</code>	end of input reached
badbit	<code>is.bad()</code>	unrecoverable I/O error

Ein formatierter Input auf dem Stream `is` muss überprüfen ob `is.fail()`. Wenn dies fehlschlägt wird der Stream mit `is.clear()` gecleared und es werden die ungültigen Eingabezeichen verbraucht, bevor fortgefahren wird.

### Dealing with Invalid Input

```
int inputAge(std::istream& in) {
    while (in) {
        int age{-1};
        if (in>>age)
            return age;
        in.clear(); // remove fail flag
        in.ignore(); // one char
        // alt: in.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        // ignores whole line
    }
    return -1;
}
```

```
#include <iostream>
#include <iomanip>
```

I/O Manipulators:  
setw(n), setprecision(n)

```
std::cout << 42 << '\t'
      << std::oct << 42 << '\t'
      << std::hex << 42 << '\n';
std::cout << 42 << '\t' // std::hex is sticky
      << std::dec << 42 << '\n';
std::cout << std::setw(10) << 42
      << std::left << std::setw(5) << 43 << "*\n";
std::cout << std::setw(10) << "hallo" << "*\n";

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

29

## Reading and modifying

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

Es ist ein ganz einfaches Programm, welches den Input in Kleinbuchstaben umwandelt. Get() und put() sind unformatierte I/O funktionen. Mit den Shiftoperationen << oder >> werden die Abstände ignoriert, während mit get(c) die Abstände ebenfalls mitgenommen werden.

# Simple Sequences

Std::vector, iteration, algorithms, lambda

## Allgemein

{} ist ein wichtiges Zeichen in C++ und wird als Garbage Collector verwendet.

Std::vector<T>

## std::vector<T>

```
std::vector<int> v{1,2,3,4,5};
```

<> Angabe des Datentyps

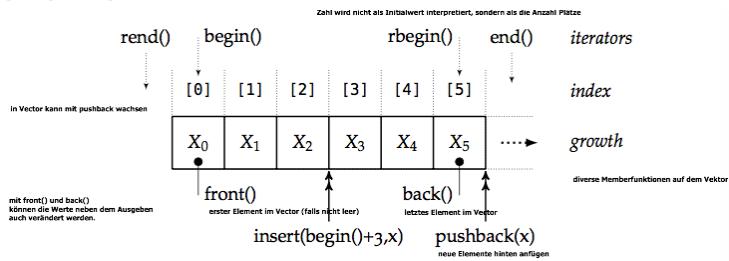
{ } Liste von Initialwerten

In C++ ist der std::vector ein wirklicher Container, wo auch alle die Elemente drin sind (im Gegensatz zu Java, wo mit Referenzen gearbeitet wird.). Eine Allokation der Elemente ist daher nicht möglich.

T ist der Template Typen Parameter, welcher ein Placeholder für den Typ angibt.

Ein Vektor kann mit einer Liste von Elementen initialisiert werden. Andernfalls ist Sie leer. Teilweise gibt es auch Stellen, wo am Ende Runde Klammern anstatt Eckige Klammern angeben werden.

## Overview



Ein Vector kann mit der pushback-Methode wachsen. Mit front() und back() können die Werte neben dem Ausgeben auch verändert werden.

Front() gibt das erste Element zurück, falls es nicht leer ist. Back() entsprechend das letzte.

Ein Vektor kann zudem mit einer Anzahl Plätze initialisiert werden. In diesem Fall sind dann runde Klammern zu gebrauchen.

```
std::vector<X> v(6);
```

Parentheses (n) for number of elements

Zahl wird nicht als Initialwert interpretiert, sondern als die Anzahl Plätze

## Bad Style Iteration

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

< muss stehen, da er sonst eins zu weit geht.  
schlechte Praxis

Man kann einen Vector gleich indexen wie ein Array. Aber Achtung es werden keine „Bounds Checks“ genmacht. Wenn er also eines zu weit geht, gibt es undefined behavior.

Mit der at() Member Funktion werden „Bounds Checks“ durchgeführt. So Schleifen zu machen, ist dennoch nicht Best Practice. Zudem ist die at()-Variante langsamer.

```
diese Angabe ist unbedingt nötig.
for (std::vector<int>::size_type i=0; i < v.size(); ++i){
    std::cout << v.at(i) << '\n';
}
```

at ist langsamer als Index Operator, macht aber einen bounds-checked.

Entspricht etwa  
der foreach Schleife  
in Java

```
auto = Passender Datentypen wird benutzt
for(auto const i:v){
    std::cout << "element: " << i << '\n'; i gibt eine Kopie
}
```

Es entspricht etwa der foreach-Schleife in Java. Es hat den Vorteil, dass keine Index Fehler auftreten können. Es funktioniert mit allen Container, sogar auch mit Valuelisten. Wenn immer möglich sollte der Elementtyp als const definiert werden.

Um Elemente zu verändern, muss dem Loop eine Referenzvariable mitgegeben werden mit &.

& mit und ist nur eine Referenz vorhanden und die Werte lassen sich verändern

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

## C++ Iterators

**Metapher** Man geht spazieren.

Jeder Container hat Iteratoren. Es gibt immer ein Paar von Iteratoren, welche den Anfang und das Ende der Iteration angeben. Eine hasNext-Abfrage findet nicht statt.

**begin(v), end(v) or v.begin(), v.end()**

Die C++ Iteratoren kennen das Ende der Iterationen nicht, sie müssen es wie folgt vergleichen:

**iterator != end(v)**

Mit \*iterator kann auf das aktuelle Element zugegriffen werden. Mit ++iterator wird zum nächsten Element weitergegangen.

### Iteration with Iterators

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

Ausgaben und gleichzeitige Modifikation

Schlechtes Beispiel, da beides gleichzeitig passiert.

Benutzt prefix++, startet bei begin(v) und vergleicht mit dem end(v). Zugegriffen wird mit \*it. Die Elemente können also auch bearbeitet werden. Es ist aber ein schlechtes Beispiel da gleichzeitig eine Modifikation sowie eine Ausgabe statt findet

Nur read-only garantieren kann man mit cbegin()/cend().

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

Elemente können nicht verändert werden, es werden allfällige Fehler zurückgeben.

## Loops and Iterators

Wenn immer möglich sollten Loops verhindert werden. Im Gegensatz zu anderen Sprachen hat C++ in seiner Standardbibliothek bereits viele Loops drin, welche wiederverwendet werden können.

Statt den eigenen Code zu schreiben, können diese Algorithmen aufgerufen werden.

Iteratoren sind der Kleber zwischen den Container und den Algorithmen.

### Example: Counting values

```
size_t count_blanks(std::string s){
    size_t count{0};
    for (size_t i=0; i < s.size(); ++i)
        if (s[i] == ' ') ++count;
    return count;
}
```

```
count(begin(s),end(s), ' ')
```

Der Algorithmus zählt wie viel etwas in einem Range vorkommt. Dies funktioniert mit einem std::string, std::vector und allen Ranges mit einem Paar Iteratoren.

### Summe aller Values in einem Vector

```
std::vector<int> v{5,4,3,2,1};
std::cout << accumulate(begin(v),end(v),0)<< " = sum\n";
```

### Special Iterators for I/O

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

#### std::ostream\_iterator<T>

Gibt Werte vom Typ T auf den gegebenen std::ostream aus. Es wird kein end() Marker für den Output gebraucht. Es endet, wenn der Inputrange zu Ende ist.

#### Std::istream\_iterator<T>

Liest Werte von T vom gegebenen Stream std::istream. Der Enditerator ist default auf istream\_iterator<T>{}. Es endet wenn der istream nicht länger good() ist.

Dazu muss der Header <iterator> included werden.

#### Using alias declaration

```
using input=std::istream_iterator<std::string>; ==> alias definieren
input eof{}; (definiert neuen Namen, für das was rechts steht.)
```

Die Streamiteratoren haben eine grosse Länge, was es eher unübersichtlich macht. Der Typ alias kann dabei helfen. Dazu kann der Iterator mit einem using als alias angegeben werden.

Und können auf die weiteren Funktionen die wesentlich kürzeren Namen übergeben werden. Es ist vor allem lohnenswert, wenn die mehrmals gebraucht werden.

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

```
using input=std::istreambuf_iterator<char>;
```

Der istream\_iterator nutzt den Operator >> für den Input. Das heisst es werden Blanks und White Spaces ignoriert. Für eine perfekte Kopie müssen wir auch den Rest haben.

Für das gibt es istreambuf\_iterator<char>, welche mit istream::get() arbeitet.

### copy with\_istream\_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);
}
```

18

### copy with\_istreambuf\_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);
}
```

19

### Exteding a vector

Es gibt einige Möglichkeiten einen Vektor zu vergrössern. Normalerweise geschieht dies mit der push\_back(val) Funktion. Dies kann aber auch mit einem Iterator als Ziel für die Copy gemacht werden. In den meisten Fällen kommt es aber dabei zu undefined behavior.

```
copy(begin(v), begin(v) + 2, back_inserter(v));
```

=> undefined behavior in den meisten Fällen.

### Filling a std::vector from input

Um einen Vektor von einem Stream zu füllen kann entweder copy mit einem back\_inserter(V) verwendet werden, was unter der Haube ein push\_back benutzt, oder man erstellt den Vektor direkt mit zwei Iteratoren.

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

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

Konstruktur, welcher gerade alle Zahlen in den Vektor einliest.  
const kann verwendet werden, da er bereits abgefüllt ist.

```
std::vector<int> v(10);           10 Stellen mit 0 initialisieren.  

fill(begin(v),end(v),2);         alle Stellen mit 2 befüllen.
```

Ein Einfüllen von einem Vector setzt voraus, dass die Stellen bereits vorhanden sind und überschrieben werden können. Entweder nutzt man dazu `v.resize(10)` oder man erstellt einen Vektor mit 10 Elementen. Für die Angabe der Länge müssen runde Klammern angegeben werden.

### Kürzere Schreibweise für den obigen Ausdruck

```
std::vector<int> v(10,2);        oder die kürzere Schreibweise.
```

### Filling a std::vector with diffrent values values

<pre>std::vector&lt;double&gt; w{}; double x{2.0}; generate_n(std::back_inserter(w),5,[&amp;x]{return x*=2.0;});</pre> <p>• <code>generate()</code> and <code>generate_n()</code> fill a range with computed values</p> <ul style="list-style-type: none"> <li>either use <code>back_inserter</code> or a non-empty container</li> <li><code>iota()</code> fills a range with subsequent values (1,2,3,...)</li> <li>header <code>&lt;numeric&gt;</code> defines <code>iota()</code></li> </ul> <pre>std::vector&lt;int&gt; v(100); iota(begin(v),end(v),1);</pre>	<small>nur so, kann es innerhalb von Lamda geändert werden. Eigen Implementation</small> <pre>std::vector&lt;double&gt; w{}; generate_n(std::back_inserter(w),5, [x=2.0] mutable {return x*=2.0;});</pre> <p>defines Variable x () needed with mutable</p> <p>Kürzere Schreibweise für <code>generate_n...</code></p> <ul style="list-style-type: none"> <li>From C++14 on, lambdas allow defining variables in the capture list. However, to vary the value, the lambda must be marked with the keyword "mutable"</li> <li>more details see later.</li> </ul>
--	--

## Algorithmen

### Number of Elements

Jeder Container hat eine `size()` Memberfunktion. Was aber, wenn wir nur ein Paar von Iteratoren haben. In diesem Fall hilft uns die Memberfunktion `std::distance` weiter. Sie rechnet die Anzahl von Elementen in einem Range aus.

```
std::cout << "distance: "<< distance(begin(s),end(s)) << '\n';  
std::cout << "in a string of length: "<< s.size()<< '\n';
```

### Generischer Algorithmus - for\_each

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

`[]()` ist ein Lambdaausdruck, welche eine Funktion erstellt. Das Lambda kann aus als Funktion regulär angegeben werden und dann mit dem Aufruf mitgegeben werden. Funktion sind first class Werte und haben auch einen Typ.

```
void print(int x){  
    std::cout << "print:" << x++ << '\n';  
}  
...  
for_each(cbegin(v),crend(v),print);
```

```
[lambda_capture] ==> Klammern sind immer nötig.  

(parameters)->return_type{  

    statements  

}  

==> () dürfen weggelassen werden.  

[]() beschreibt eine Funktion die keine  

Parameter hat und nichts zurückgibt.  

==> Aufruf der Funktion durch ()
```

==> kann auch in eine Variable gespeichert werden.  
 auto l = []();  
 Aufruf mit l();

Ein Lambdaausdruck erstellt on the fly ein Funktionsobjekt, welches an einen Algorithmus weitergegeben werden kann. Die erstellte Funktion wird innerhalb vom Algorithmus aufgerufen.

Erfassen Sie Variablen(captures), die aus dem umgebenden Bereich genommen wurde oder definieren Sie neue. Mit =copy, mit &reference. Umbenennen ist ebenfalls möglich.

Die Parameter sind wie Funktionsparameter. Sie vorhanden, kann einfach **auto** benutzt werden.

### Finding Elements

```
auto zero_it=find(begin(v),end(v),0);  

if (zero_it == end(v)){  

    std::cout << "no zero found \n";  

}
```

find() und find\_if() geben einen Iterator zurück, welcher auf das erste Element zeigt welche mit der Bedingung übereinstimmt. Wenn es keines gibt, wird das Ende des Ranges zurückgegeben.

Etwa gleich funktionieren da count() und count\_if(), welche die Anzahl übereinstimmender Elemente zurückgeben in einem Range.

```
std::cout << count(begin(v),end(v),42)<<" times 42\n";  

std::cout << count_if(begin(v),end(v),  

    [](int x){return 0==(x%2);} )<<" even numbers\n";
```

Gereade Zahlen im Vector.

# Structure Behavior with Functions

Defining Functions, Parameters and Arguments, Visibility and Lifetime, Functionality Guarantees

## A good Function

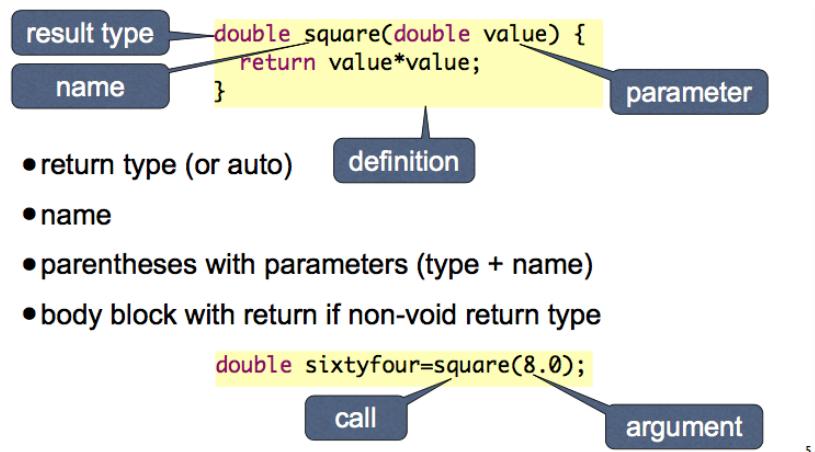
Macht eine Sache und wird auch nach dieser genannt. Sie hat zudem nur weniger Parameter (bis maximal 5). Zudem besteht Sie nur aus wenigen Zeilen von Code ohne riesige nestete Kontrollstrukturen. Des weiteren gibt Sie eine Sicherheit über das Result (sie gibt einen gültigen Wert zurück).

## Refactoring von „Bad Code“ zu „Good Code“

Eclipse steht einige Refactorings zur Verfügung, um Funktionen besser zu machen

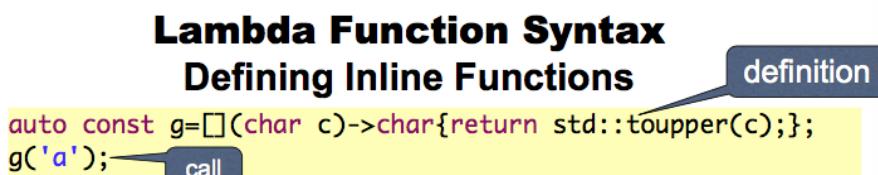
- Rename – provide better name
- Extract Function – simpler functions (Currently under Improvement)

## Function Syntax (repeated)



5

## Lambda Function Syntax - Defining Inline Functions



Auto const wird genutzt um die Funktion so abzuspeichern, dass sie später aufgerufen werden kann. [] eröffnet die Lambda Funktion, welche allfällige Captures beinhaltet. Zudem Parameter wie bei anderen Funktionen sowie ein Body mit den Code Statements.

## Header files and Functions

Normalerweise sind nur die Funktionsdeklarationen in den Headerdateien. Die Funktionsdefinitionen sind dann in den .cpp files.

Eine kurze, oft aufgerufene Funktion (genannt inline function) kann aber in den Headerfiles plaziert werden. Das Inline Wort macht, dass die One Definition Rule nicht verletzt wird. Die Class Member Funktionen sind direkt implementiert und sind implizit inline. Dies gilt für template functions ebenfalls.

```
inline double square(double value) {
    return value*value;
}
```

## Scopes

Jede Funktion macht einen eigenen Scope auf. Auf gleicher Ebene können nicht die gleichen Namen verwendet werden. Zudem öffnet {} einen neuen Scope.

```
void showScopingRules(int i, double d){
    unsigned j{1};
    // can not use name i instead of j
    std::cout << i << "\n";
{
    char i{'d'}; // Dieses i ist nach dem Scope Ende nicht mehr gültig.
    // parameter i not accessible but d is
    std::cout << i << " " << d << "\n";
} collects garbage
++i; // that is the parameter i
     // no longer shadowed
for (unsigned i=0;i<j;++i){ // another i
    std::cout << i << "\n"; // Hat keinen Effekt auf das äussere
}
std::cout << i << "\n";
// parameter i again
}
```

Die Scoping Regeln sind ähnlich wie bei anderen Programmiersprachen.

Die Parameter sind sichtbar innerhalb des Funktionsblock.

In C++ ist die Lifetime wichtig. Wenn ein Blockscope fertig ist, die Lifetime ist fertig. () „collects the garbage“). Dies ist ein Keyfeature von C++.

Bei redefinieren von den gleichen Namen ist Vorsicht geboten. Es kommt zu Shadowing und dies ist im Gegensatz zu Java kein Error.

## Namespaces

Namespaces sind Scopes für die Gruppierung und um Kollisionen mit den Namen zu verhindern (One Definition Rule). Daher ist der selbe Namen in unterschiedlichen Scops möglich.

Die Funktionen sind auf einem „namespace-level scope“ definiert oder direkt als Klassenmembers (inline). Namespace Präfix wäre zum Beispiel std::getline(s).

Es gibt auch einen globalen Namenspace mit :: als Prefix. Die führenden :: werden aber meist weggelassen. Sub-Namespace sind möglich und Nesting ist ebenfalls erlaubt.

```
neuer Namensraum öffnen
namespace demo{
void foo(); //1
namespace subdemo{
void foo()/*2*/}
} // subdemo
} // demo
namespace demo{
void bar(){
    foo(); //1
    subdemo::foo(); //2
}
}
void demo::foo()/*1*/ // definition
int main(){
    using demo::subdemo::foo; nur einzelne Funktionen
    foo(); //2 mit using namespace könnte alle eingebunden
    demo::foo(); //1 werden.
    demo::bar();
}
```

Die Namespaces können nur außerhalb von Klassen und Funktionen definiert werden.

Nesting ist möglich.

Ein Namespace mehrmals geöffnet und geschlossen werden um Definitionen und Deklarationen über mehrere Files zu ermöglichen.

## Using Declaration

Es importiert einen Namen von einem Namespace in den aktuellen Scope. Dies ist möglich mit Funktionen, Typen und Variablen.

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

Der Name kann so ohne Namespace-Prefix genutzt werden. Es ist nützlich, wenn der Name sehr oft verwendet wird. Alternativ kann using benutzt werden für Typen, wenn deren Namen lang ist.

```
using str=std::string;
str t{"short alias"};
```

Solche Deklarationen sollten in einem Headerfile unterbunden werden. Entweder im CPP File oder am besten im Funktionsbody.

## Anonymous Namespace

Nur wenn man im CPP File Definitionen hat, welche man niemand anderen zur Verfügung stellen will. (Helperfunktionen verstecken.)

```
#include <iostream>
namespace { // anonymous
void doit(){ // can not be called
    // outside this file
    std::cout << "doit called\n";
}
} // anonymous namespace ends
void print(){ // callable from other
    // parts if declared
    doit();
    std::cout << "print called\n";
}
```

```
void caller(){
    void print(); // declare print
    print();
    void doit(); // declare doit
    //doit();      // linker error
}
```

## Inline Namespaces for Library Versioning

Wir von uns nicht gebraucht → Siehe Vorlesungsfolien.

## References

### References in C++

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

Eine Referenz ist ein Alias für eine Variable oder einen Wert. Das Original muss existieren, solang darauf referenziert wurde. Am sinnvollsten ist es für Funktionsparameter. Es kann so verhindert werden, dass grosse Objekte kopiert werden müssen (const reference). Oder als nicht-const als eine Variable mit Side-Effects.

```

int i{42};
int &ri{i}; // must initialize ref
int const &cri{i}; // const alias
int const &cr{6*7}; // extend lifetime of 6*7
ri = 43; // changes i, ri only an alias
//--cri ; // doesn't work -> const
int &&rvi{3*14}; // extends lifetime of 3*14
//int &&rvri{i}; // impossible
int &&rvri{std::move(i)}; // steal i's content
  Inhalt wegschieben
  
```

bad code!  
exposition  
only

- &Prefix für lvalue Referenzen müssen initialisiert werden, sonst gibt es null Referenzen
- const &prefix für const Referenzen
- mit std::move(var) wird der Inhalt weggeschoben und ist nicht länger verfügbar.

### Kinds of reference parameters

#### Lvalue Referenzen std::ostream &out

es gibt dabei keine Kopie. Es wird direkt darauf gearbeitet. Es sind daher Parameter mit Side Effekten.

#### Const Referenzen std::vector<int> const &v

Für grössere Parameter ohne Side Effekte. Vektor wird mitgegeben, die Funktion darf es aber nicht abändern.

#### Rvalue Referenzen std::vector<int> &&

Für Move Semantic und Perfect Forwarding. Details in C++ Advanced.

### Returning References

```

std::ostream &print(std::ostream &out, int value) {
    out << value;
    return out; // OK, allows chaining
}
  
```

Es sollten nur Referenzen zurückgegeben werden, welche wir über die Argumente als Referenzen erhalten haben. Dies gewährt Lifetime, weil der Aufrufer ja das Original besitzt.

Es sollten nicht Referenzen zu lokalen Variablen zurückgegeben werden.

```

int & doNeverDoThis_ReturnReferenceToLocalVariable(){
    int number{42};
    return number; // really really bad and wrong!!!
}
  
```



### Parameter Passing – Returning Results

#### Pass by Value/Return by Value

```

int plusOne(int i){
    return i+1;
}
  
```

Es wird ein temporärer Wert an die Funktion übergeben. Mit Hintergrund findet eine Kopie/Moving statt. Diese Methode ist in den heutigen Compiler am effizientesten. Es kann nicht falsch laufen, da keine Side Effekte und Daingling.

## std::string serialize(std::vector<int> const &v);

Keine Einschränkungen auf dem Aufrufargument. Es findet, dass kopiert wird. Vielleicht wird aber eine Copy Funktion innerhalb genutzt. Zurückgeben mit einer Const-Referenz ist potentiell gefährlich.

### Pass by Reference

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

Es wird eine Variable (lvalue) auf der Aufrufseite benötigt.

Grundsätzlich ist dies für Elemente welche nicht kopiert oder verschoben werden können (std::istream, std::ostream).

### Function Overloading

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

RETURN Typ muss gleich sein, so wie in JAVA.

Der selbe Funktionsnamen kann für verschiedene Funktionen benutzt werden, wenn die Parameter unterschiedlich sind. Der Return Typ muss aber gleich sein.

Die Auflösung des Overloads findet zur Compile Zeit statt = static Polymorphism.

Der interne Name der Funktion beinhaltet auch die Parametertypen als Information (nicht die Parameternamen!).

### Default Arguments

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

Eine Funktionsdeklaration kann auf Default Parameter mitgeben. Die Default-Parameters müssen im Headerfile stehen. Die Definition muss/sollte nicht wiederholt werden. Ein = vor dem Wert ist nötig. Im Hintergrund findet ein impliziter Overload der Funktion mit weniger Argumenten statt.

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

### Overloading Ambiguity

```
int factorial(int n){  
    if (n > 1) return n * factorial(n-1);  
    return 1;  
}  
  
double factorial(double n) {  
    double result=1;  
    if (n < 15)  
        return factorial(int(n));  
    while(n > 1) {  
        result *= n;  
        --n;  
    }  
    return result;  
}
```



```
void demoAmbiguity() {  
    /*  
        factorial(10u); // ambiguous  
        factorial(1e11); // ambiguous  
    */  
    std::cout << factorial(3) << "\n";  
    std::cout << factorial(1e2) << "\n";  
}
```

## Functions as Parameters

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

Funktionen (Referenzen) sind „first class“ in C++. Sie können als Argumente mitgegeben werden. Diese wird in Referenzvariablen gehalten. In C würde dies mit Function Pointers realisiert werden.

### Funktions Referenz Typen Syntax

**double (&h)(double)**

### Functionality Guarantess

Wenn eine Funktion den Auftrag nicht erfüllen kann, gibt es verschiedene Ansätze damit umzugehen.

1. Denn Error ignorieren und potentiell Undefined behavior zur Verfügung stellen
2. Ein Standard Resultat zurückgeben um den Error abzudecken
3. Ein Error Code oder Error Wert zurückgeben
4. Ein Error Status als Side Effekt zur Verfügung stellen
5. Eine Exception werfen

### When could a function fail?

#### Precondition

- negative Indizes
- Falsche Argumente werden mitgegeben

#### Postcondition

- Ein File kann nicht geöffnet werden
- Zugriff auf Ressource ist nicht möglich.

#### Ignore the error

```
std::vector<int> v{1,2,3,4,5};
v[5] = 7; // sic!
```



Verlässt sich darauf, dass der Aufrufer alle Voraussetzungen erfüllt. Dies ist nur realisierbar, wenn es nicht von anderen Ressourcen abhängt. Es ist die effizienteste Umsetzung, da keine unnötigen Kontrollen vorhanden sind. Einfach für den Ersteller, schwieriger für den Aufrufer. Es sollte bewusst und konsequent getan werden.

## Cover Error with a standard result

```
std::string inputName(std::istream &in) {
    std::string name{};
    in >> name;
    return name.size() ? name : "anonymous";
}
```

Entlastet den Aufrufer davon von der Notwendigkeit sich darum zu kümmern, dass alles richtig ist. Dies natürlich nur, wenn auch mit dem Standardwert weitergefahren werden kann.

Es ist aber möglich das es darunterliegende Probleme versteckt, deren Debugging dann in Alpträumen enden kann.

Meist ist es besser, wenn der Aufrufer den Defaultwert mitgeben kann.

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

## Error Value

Dies ist nur möglich, wenn der Ergebnisbereich kleiner ist das der Bereich des Rückgabetyps. Es wäre also ein Wert vorhanden, welcher genutzt werden kann. Meistens wird dafür -1 verwendet. Der An-

## Error Status Side Effect

Setzt einen Referenz Parameter voraus. Dies kann ein Objekt der Memberfunktionen sein. Alternativ kann auch eine Globale Variable verwendet werden (sollte man nicht). Ein Beispiel ist die Variable errno.

## Exceptions

Eine Exception werfen würde man mit throw value. Jeder kopierbare Typ kann geworfen werden. Es gibt keine Möglichkeit um zu spezifizieren, was geworfen werden könnte. Zudem sind keine Meta-Info verfügbar als Teil der Exception. (Kein Stack Trace oder Position).

Wenn eine Exception geworfen wird, während die Exception propagiert wird, kommt es zu einem Programmabbruch.

## Catching Exceptions

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

Dafür sind wie in anderen Sprachen die try-catch Blöcke vorhanden. Im try kann der ganze Funktionsbody stehen. Während die Exception beim Wert geworfen wird, findet der Catch mit einer Referenz statt. Die Reihenfolge der angegebenen Sequenz ist wichtig, da der erste Treffer genommen wird. Am Ende kann ein catch\_all die restlichen Exceptions auffangen.

```
catch(...) { }
```

```
throw std::logic_error{"logic"}
```

In der Standard Bibliothek sind einige vordefinierte Exceptiontypen vorhanden, welche verwendet werden können. Std::logic\_error ist der meist genutzt, es gibt aber noch domain\_error, invalid\_argument, length\_error und out\_of\_range. Zudem gibt es noch runtime\_errors: range\_error, overflow\_error und underflow\_error. Alle haben einen String für die Angabe des Grundes. Std::Exception ist die Basisklasse. Der Grund kann mit e.what() abgefragt werden.

### Silly Exceptions Example

```
#include <iostream>
#include <cstdlib> // srand, rand, time functions -> deprecated don't use
#include <stdexcept>
void mightthrow_logic_error(){
    if (rand()%2) throw std::logic_error{"logic"};
}
void must_be_called_with_greater_3(unsigned i){
    if (i <= 3) throw std::invalid_argument{"too small"};
}
int main() try {
    srand(time(0)); // randomize rand() - deprecated don't use
    mightthrow_logic_error();
    must_be_called_with_greater_3(rand()%6);
    if (rand()%2) throw std::string{"error"};
    if (rand()%2) throw 15;
    throw "hallo";
} catch (std::logic_error const &e) {
    std::cout << "logic error: \'" << e.what() << "\'\n";
    throw; // re-throw the caught exception
} catch (std::exception const &e) {
    std::cout << "exception: \'" << e.what() << "\'\n";
} catch (int const &i) {
    std::cout << "exception int: <" << i << ">\n";
} catch (std::string const &s) {
    std::cout << "exception string: \'" << s << "\'\n";
} catch (...) {
    std::cout << "unknown exception value\n";
}
```

Throwing std exception

function try-catch block

can throw any value

catch by reference

re-throw current exc.

what() delivers string

catch all of the rest

37

### Functions with „narrow contract“

```
double square_root(double x){
    if (x<0)
        throw std::invalid_argument{"square_root imaginary"};
    return std::sqrt(x);
}
```

Für Funktionen, welche eine Voraussetzung für den Aufrufer haben. Zum Beispiel wenn nicht alle möglichen Argumentwerte für die Funktion nützlich sind.

Benutzen Sie nicht Exceptions als zweite Bedeutung für Returnwerte. Sonst wird das catch zu „come from“ und der Wurf zu „go to“.

## Testing for Exceptions

```
void testSquareRootNegativeThrows(){
    ASSERT_THROWS(square_root(-1.0),std::invalid_argument);
}
```

```
void testEmptyVectorAtThrows() {
    std::vector<int> empty_vector{};
    ASSERT_THROWS(empty_vector.at(0),std::out_of_range);
}
```

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

# Classes and Operator Overloading

## Classes

### A Good class

GoodClassName.h

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

- Tuet eine Sache gut und ist nach diesem benannt. (Single Responsibility)
- Beinhaltet Member Funktionen mit nur weniger Zeilen Code (nicht zu lang)
  - o Lange Kontrollstrukturen sollen vermieden werden
- Für die Klasse sind Eigenschaften gegegen, welche immer war sind. Dies muss nach dem Konstruktor gelten. Das heisst, dass zum Beispiel nur ein gültiges Datum herauskommt und durch die Klasse auch nicht ungültig gemacht wird.
- Einfach zu Nutzen ohne komplizierten Protokollsequenzen

## Strucute of a class

Eigenes Header-File

```
#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} {/*...*/}
    static bool isLeapYear(int year) {/*...*/}
private:
    bool isValidDate() const {/*...*/}
};

#endif /* DATE_H_ */
```

Eine Klasse definiert einen eigenen Typ.

Die Klasse ist normalerweise in einem Header File definiert.

Am Ende der Klassendefinition ist ein Semicolon nötig.

## Include Guard

```
#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} {/*...*/}
    static bool isLeapYear(int year) {/*...*/}
private:
    bool isValidDate() const {/*...*/}
};

#endif /* DATE_H_ */
```

Der Include Guard stellt sicher, dass der Inhalt des Header Files nur einmal importiert/include wird. Es verhindert Zyklen.

Zudem stellt es sicher das die One Definition Rule nicht verletzt wird.

## Anweisungen

#ifndef <name>, #define <name> und #endif

## Class Head

```
class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year(year), month(month), day(day) /*...*/
    static bool isLeapYear(int year) /*...*/
private:
    bool isValidDate() const /*...*/
};
```

Es gibt zwei Schlagwörter, mit welchen sich Klassen definieren lassen. Einerseits `class`, andererseits `struct`.

Während die Standardmässige Sichtbarkeit der Memberfunktion bei `class` `private` ist, ist Sie bei `struct` `public`. Die Grundstruktur bleibt die selbe.

## Access specifier

```
public:
    Date(int year, int month, int day)
        : year(year), month(month), day(day) /*...*/
    static bool isLeapYear(int year) /*...*/
private:
    bool isValidDate() const /*...*/
};
```

Es wird zwischen drei Access Specifier unterschieden:

### Private:

Nur sichtbar innerhalb der Klassen (und seinen Friends). Für versteckte Daten.

### Protected:

sichtbar auch in den Subklassen

### Public:

Sichtbar von überall, für das Interface der Klasse

Die Angabe findet in Blöcken statt. Alle Methoden in diesem Block haben dieselbe Sichtbarkeit.

## Member variables

Alle Daten, die ein Element dieser Klasse representieren.

```
class Date {
    int year, month, day; ==> Implizit privat
public:
```

Sie haben einen Typ sowie einen Namen. Wenn immer möglich sollten sie `const` gemacht werden. Es sollten keine Variablen hinzugefügt werden um zwischen Memberfunktionen zu kommunizieren. Sie sind sehr zu testen. Nutze dafür besser Parameter.

## Constructor

```
class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year(year), month(month), day(day) /*...*/
    wenn dies nicht getan wird, wird es default initialisiert.
```

Funktion mit dem Namen der Klasse, eine spezielle Member Funktion. Die Funktion `<class name>()` hat keine Rückgabewert. Inhalt ist eine Initializer List für die Member Initialization.

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

```
//Default-Constructor
Date();
//Copy-Constructor
Date(Date const &);
```

```
//Move-Constructor
Date(Date &&);
//Typeconversion-Constructor
explicit Date(std::string const &);
```

//Destructor      in der Regeln nur ein Argument

### Default Constructor (ohne Parameter) – Date d{};

Er hat keine Parameter. Er ist implizit verfügbar, falls keiner definiert worden ist. Seine Funktion ist es die Variablen mit Defaultwerten zu initialisieren.

### Copy Constructor – Data d2{d}

Hat einen <own-type> const & parameter. Dieser ist implizit verfügbar, falls keiner explizit definiert wurde. Er ist immer dann zu verwenden, wenn wir ein bestehendes Objekt duplizieren wollen.

### Move Constructor – Date d2{std::move(d)};

Hat nur einen Parameter mit <own-type> &&. Es ist implizit verfügbar, falls er nicht explizit definiert wurde. Er verschiebt alle Members.

### Typeconversion Constructor – Date d{„19/10/2016“s};

Konstruktor um in den eignen Typ zu verwandeln. Er hat einem Parameter mit <own-type> const &. Er sollte explizit deklariert werden um unschöne Konversionen zu vermeiden.

### Destructor

```
//Destructor
~Date();
```

Er ist das Gegenstück zum Konstruktor. Benannt ist er gleich wie der Standard Konstruktor, beginnt aber mit einem Tilde-Zeichen(~). Er setzt alle Ressourcen frei. Implizit verfügbar, wer aber richtig programmiert hat dies immer selber zu definieren. Eine Exception darf nicht geworfen werden. Der Destructor wird automatisch am Ende des Blockes aufgerufen.

### Inheritance

Im Gegensatz zu Java sind mehrere Vererbungen möglich, es gibt aber keine Interfaces. Diese Basisklassen sind mit dem Namen spezifiziert.

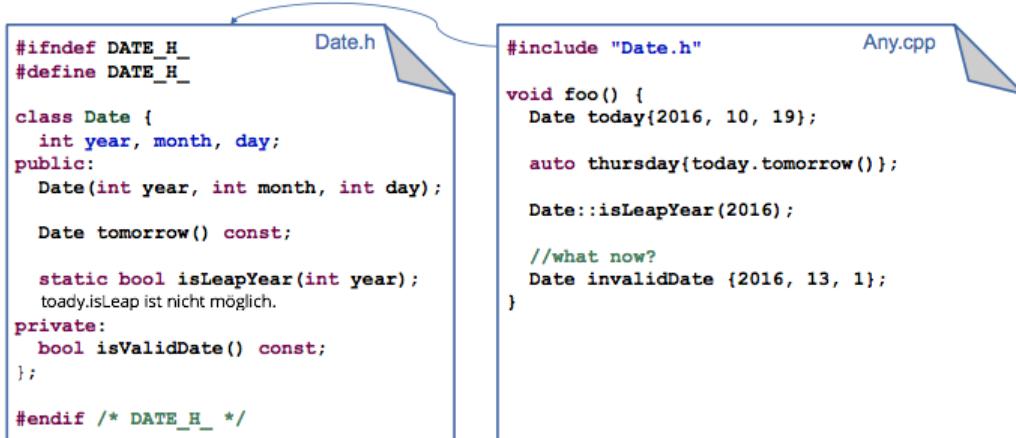
```
class Base {
public: private:
  int onlyInBase;
protected:
  int baseAndInSubclasses;
public:
  int everyoneCanFiddleWithMe
};
```

```
class Sub : public Base {
  //Can see baseAndInSubclasses and
  //everyoneCanFiddleWithMe
};
```

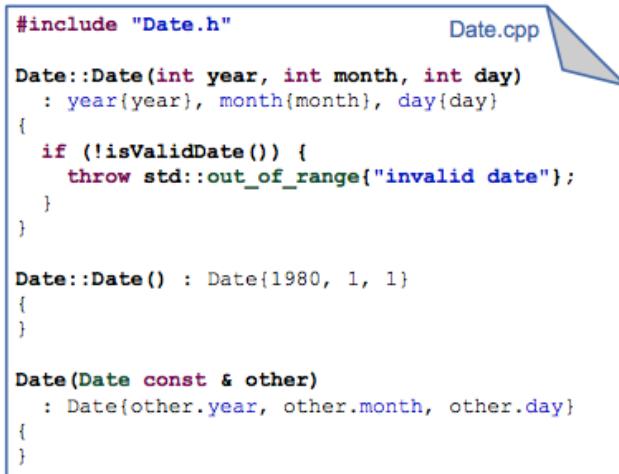
Auf der Vererbung kann die Sichtbarkeit angegeben werden. Es handelt sich aber um das Maximum. Private bleibt private auch wenn es mit Public angegeben ist.



## Using the class



## Implementing Constructors



## Invariant aufbauen

Alle Eigenschaften für ein Wert von einem Typ welche gültig sind. Das heisst. Eine Date-Instanz repräsentiert auch ein gültiges Datum. Alle public Funktionen nehmen dies an und halten die Gültigkeit intakt.

## Initialisierung aller Member

Er erstellt nur eine valide Instanz, andernfalls wird eine Exception geworfen. Er nutzt die Liste für die Initialisierung und Defaultwerte, wenn angegeben-

## Implementing Member Functions

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

Sie dürfen die Invariant nicht aufheben oder verletzen. Das Objekt soll in einem gültigen Zustand bleiben.

Implizit ist das this-Objekt verfügbar, welches ein Pointer ist und mit → genutzt werden kann.

Wenn immer möglich const verwenden-

Wenn Sie const ist, darf sie nicht die Member verändern.

## Implementing Static Member Functions

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

//or the unreadable version
bool Date::isLeapYear(int year) {
    return !(year % 4) &&
           ((year % 100) || !(year % 400));
}
```

Hier ist das this Objekt nicht verfügbar.

Zudem können die Funktionen nicht const sein.

In der Implementation ist das Schlüsselwort „static“ nicht vorhanden.

Aufgerufen werden können diese Funktionen nach dem Schema: <classname>::<member>(), also zum Beispiel Data::isLeapYear(2016);

## Static Member Variables

In der Implementation wird das Schlüsselwort „static“ ebenfalls nicht verwendet. Static const Member können direkt initialisiert werden. (im Headerfile). Zugriff von aussen geht über den Klassennamen sowie die Member, also <classname>::<member>.

```
#include "Date.h"                               Any.cpp
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
    Date::favoriteStudentsBirthday = ...;
}
```

```
#include "Date.h"                               Date.cpp
Date const Date::myBirthday{1964, 12, 24};
Wo liegt dieses Ding, wird damit beantwortet...
Date Date::favoriteStudentsBirthday{1995, 5, 10};
```

```
class Date {                                     Date.h
    static const Date myBirthday;
    static Date favoriteStudentsBirthday;
    static const int zero{0};

    //...
};
```

## Operator Overloading

Die Operatoren können mit eigenen Definitionen überladen werden. Die Deklaration erfolgt wie eine Funktion.

```
<returntype> operator<op>(<parameters>);
```

Bei Unären Operatoren muss ein Parameter mitgegeben werden, während bei den binären Operatoren zwei Operatoren mitgegeben werden.

Die Operatoren sollen mit Grund eingesetzt werden. Wenn Zweifel da sind, sollte es gemacht werden wie die „Ints“.

### ■ Overloadable Operators:

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

### ■ Non-Overloadable Operators:

::	.	*	.	:
----	---	---	---	---

## Free Operator

Beispiel      Das Datum vergleichbar machen.

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

void foo() {
    std::cout << Date::myBirthday;
    Date d();
    std::cin >> d;
    std::cout << "is d older? " << (d < Date::myBirthday);
}
```

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

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

Wir möchten also Jahr, Monat und Tag vergleichen. Der Free Operator hätte zwei Parameter vom Typ Date. Bei const &. Zurückgegeben würde ein Boolean.

Dies kann inline geschehen, wenn es im Header definiert wird. Dabei geht es darum, dass diese nicht mehrmals eingebunden werden. Inline verhindert, dass dies mehrmals geschieht.

Das Problem: Wir haben so keinen Zugriff auf die private Member der Klasse.

```
class Date { Date.h
    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)));
    }
};
```

```
#include "Date.h" Any.cpp
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
    Date d();
    std::cin >> d;
    std::cout << "is d older? " << (d < Date::myBirthday);
}
```

Für den Member Operator wäre nur noch ein Parameter nötig, welcher const & ist. Als Rückgabewert fungiert ebenfalls ein Boolean. Der Parameter bildet die rechte Seite der Operation.

Implizit ist das this-Objekt verfügbar, welches die linke Seite darstellt, welches durch den Qualifier ebenfalls const ist. Zugriff zu den privaten Members ist möglich.

### Comparison Syntactic Sugar

Std::tie erstellt ein Tupel und bindet die Argumente mit einer Ivaule-Referenz. Std::tuple stellt die Operatoren ==, !=, <, <=, > und >= zur Verfügung. Der Vergleich von std::tuple findet komponentenweise von links nach rechts statt.

```
#include "Date.h" Date.cpp
#include <tuple>

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

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

    bool operator<(Date const & rhs) const;
};
```

### Implementing All Comparisons

Sorgt für Transivität, Associativität und Kommunikativität. Duplikation sollte vermieden werden. Zudem sollte man auch Call Loops achten.

```
class Date { Date.h
    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);
    }
};
```

Boost stellt die Basisklassen zur Verfügung, wovon erbt werden kann. Die private Vererbung ist genügend für die Verwendung. Weitere Informationen sind unter [www.boost.org](http://www.boost.org) verfügbar.

```
#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);
    }
};
```

## Sending Date to std::ostream

### Als Free Function

Output Operator definiert als Free Operator. Parameter sind std::ostream & sowie Date const &. Zurückgegeben wird std::ostream & um den Output zu verändern. Hier ist es natürlich wieder so, dass die privaten Members nicht erreicht werden können.

```
#include "Date.h"          Any.cpp
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
}
```

In diese Klasse hinein schieben.

```
#include <iostream>           Date.h
class Date {
    int year, month, day; //private ⊗
};

inline std::ostream & operator<<(std::ostream & os, Date const & date){
    os << date.year << "/" << date.month << "/" << date.day; //Invalid
    return os;
}
```

### Als Member Function

Als Parameter wird nur std::ostream & benötigt. Ein Zugriff auf die privaten Member ist nun natürlich möglich. Zurückgegeben wird std::ostream & um den Output zu verändern.

```
#include "Date.h"          Any.cpp
#include <iostream>

void foo() {
    //Invalid (Compiler)
    std::cout << Date::myBirthday;
    //Invalid (by Convention)
    Date::myBirthday << std::cout;
}
```

```
#include <iostream>           Date.h
class Date {
    int year, month, day; //private ⊗
    std::ostream & operator<<(std::ostream & os) const
    {
        os << year << "/" << month << "/" << day;
        return os;
    }
};
```

Die Probleme hier sind nun das der Compiler std::ostream auf der linken Seite nicht akzeptiert. Der ostream auf der rechten Seite ist gemäss der Konvention falsch. Daher muss eine print() Member Function erstellt werden.

## Print() Member Funktion

Als Workaround. Die Funktion hat auch Zugriff auf die privaten Memberdaten. Sie kann mit dem Free Operator << aufgerufen werden.

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

```
#include "Date.h"      Any.cpp
#include <iostream>

void foo() {
    std::cout << Date::myBirthday;
}
```

## Reading Date from std::istream

Der Input Operator hat wie der Output Operator dieselben Probleme. Das wird hier eine read() Member Funktion implementiert. Der Operator ist >>, Parameter sind std::istream & und Date const & und std::istream & wird zurückgegeben um den Input zu ändern.

```
#include <iostream>
Date.h
class Date {
    int year, month, day;
public:
    std::istream & read(std::istream & is) {
        //Logic for reading values and verifying correctness
        return is;
    }
    inline std::istream & operator>>(std::istream & is, Date & date) {
        return date.read(is);
    }
}
```

```
Any.cpp
#include "Date.h"
#include <iostream>

void foo() {
    Date d{};
    std::cin >> d;
}
```

## Logik der read() Funktion

Als Bedingung um eine korrektes Datum zu erstellen, soll erwartet werden das der Stream in einem guten Zustand ist. Wenn der Extrakt des Datums fehlgeschlagen ist sollte der Stream auf fail gesetzt werden. Das this Objekte sollte nicht überschrieben, wenn der Input nicht verwendet werden kann um ein korrektes Datum zu erstellen.

```
#include <iostream>
Date.h
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;
    }
};
```

Mit std::cin >> std::noskipws werden die White Spaces nicht übersprungen.

In einem Statement darf man nur einen Seiteneffekt auf eine Variable haben, daher werden sep1 und sep2 verwendet.

## Konstruktor with std::istream &

Die Deklaration des Konstruktors nimmt einen std::istream & als Parameter. Dies gibt man mit explicit an, da man es genauso haben will.

Deklarationen mit nur einem Parameter sollen immer explizit erfolgen. Damit lässt sich eine automatische Konversion verhindern.

Es wird eine Exception geworfen, wenn der Input nicht ein gültiges Datum repräsentiert. Das Datum wird nicht erstellt. Alternativ kann auch ein default-Datum erstellt werden.

```
#ifndef DATE_H_
#define DATE_H_

class Date {
//...
    explicit Date(std::istream & in);
};

#endif /* DATE_H_ */
```

Date.h

```
#include "Date.h"

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

Date.cpp

## Factory Function for Creating Date Objects

Deklaration für einen Factory Funktion für das Datum

Diese Funktionen werden entweder mit make\_xxx() oder create\_xxx() angegeben. Plaziert sind Sie in der Klasse als statische Member Funktion oder innerhalb desselben Namespaces wie die Klasse.

Es gibt einen Default-Wert zurück, wenn das Lesen fehlschlägt.

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

## Default Value

### Default Constructor

Da wir einen Standardwert angegeben haben, sollte dies der Wert sein, welcher mit dem Standard Konstruktor (ohne Parameter) erstellt werden.

**Kommentar:** Dies ist gilt, wenn {} oder gar nichts angegeben wird. Zweiteres funktioniert aber nicht immer, daher sollten immer {} verwendet werden.

```
#ifndef DATE_H_
#define DATE_H_

class Date {
//...
    Date();
};

#endif /* DATE_H_ */
```

Date.h

```
#include "Date.h"

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

Date.cpp

Date date (2016.10.26) ==> geht so an sich auch, war die alte Definiton. Man kann nicht immer die runden Klammern verwenden. z.B. Date default() ist eine Funktionsdeklaration.

## Default Initialization for Members

Member Variablen können einen Defaultwert zugewiesen haben.

Diese Werte werden genutzt, wenn die Member nicht in der Initializerliste present sind des Konstruktors. Die Liste überschreibt immer noch diese Werte.

Dies ist nützlich wenn mehrere Konstruktoren das Datum etwa gleich initialisieren.

```
#ifndef DATE_H_
#define DATE_H_

class Date {
    int year{9999}, month{12}, day{31};
    //...
    Date();      ==> Initialisierung direkt im Header.
    Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.h

```
#include "Date.h"

Date::Date() {
}

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

Date.cpp

## Defaulted Constructors

Einige spezielle Memberfunktionen sind implizit verfügbaren in gewissen Fällen. Zum Beispiel ist der Default Konstruktor implizit verfügbar, wenn er nicht explizit definiert ist.

Die (Re-)Implementierung des Default Verhalten kann verhindert werden indem man folgendes angibt.

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

Dies ist möglich für den Default Constructor, Deconstructor, Copy/Move Constructor und Copy/Move Assignment Operator.

```
#ifndef DATE_H_
#define DATE_H_

class Date {
    int year{9999}, month{12}, day{31};
    //...
    Date() = default;
    Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.h

```
#include "Date.h"

Date::Date()=default;

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

Date.cpp

Teilweise sind implizite Konstruktoren nicht erwünscht. Entweder macht man diese explizit privat (können ihn von aussen nicht benutzen). Oder man löscht diese. Zweiteres ist vorzuziehen.

**<ctor-name>() = delete;** aktuell

```
#ifndef BANKNOTE_H_
#define BANKNOTE_H_

class Banknote {
    int value;
    //...
    Banknote(Banknote & const) = delete;
};

#endif /* BANKNOTE_H_ */
```

Banknote.h

```
#include "Banknote.h"           Forger.cpp

Banknote forge(Banknote const & note) {
    Banknote copy{note}; //!
    return copy;
}
```

Forger.cpp

## Date in Namespace

Die Klasse Date kann/sollte in einem Namespaces gruppiert werden mit seinen Operatoren.

```
#include <iostream>               Date.h

namespace date {
    class Date {
        int year, month, day;
    public:
        std::istream & read(std::istream & is);
        std::ostream & print(std::ostream & os) const;
    };

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

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

Date.h

Um Namen aus einem Namespace zu nutzen ist wie bei std:: auch eine Qualifikation nötig.

**date::Date d{};**

Member Implementationen müssen mit dem Namespace qualifiziert werden. Zum Beispiel date::Date::read.

```
namespace date { Date.h
class Date {
    int year, month, day;
public:
    std::istream & read(std::istream & is);
}; }
```

```
#include "Date.h" Date.cpp
std::istream & date::Date::read(std::istream & is) {
    //...
    return is;
}
```

Unqualifizierte Funktionen (Operatoren) - Aufrufe erfordern den Namespace nicht explizit im Aufruf.

**NOT:** std::cout date::<< birthday;

Funktionen und Operatoren werden im Namespaces nachgeschlagen aufgrund ihrer Argumentenliste.

```
~> diff Any.cpp Date.h
```

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

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

```
#include "Date.h" Any.cpp
void foo() {
    date::Date birthday{2011, 8, 2};
    //date::operator<< (std::cout, d);
    std::cout << birthday;
}
```

## ADL

Die Operatoren sind der Grund dafür, wieso ADL definiert wurde.

### ADL Examples

```
namespace one { adl.h
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) {/*...*/}
```

```
#include "adl.h" adl.cpp
int main() {
    one::type_one t1();
    f(t1);
    two::type_two t2();
    f(t2);
    h(t1);           ist nicht vorhanden und im globalen scope
    two::g(t1);      nicht definiert.
    g(t1);           Eigentlich global, aber Compilefehler
    g(t2);
}
```

## ADL Issues (Where Intuition Fails)

Generischer Code (Template) nimmt möglicherweise nicht den globalen << Operator in einem Algorithmusaufruf, bei welchem der ostream:iterator verwendet werden. Dies wenn der Ausgabewert ebenfalls im Namespace std ist. Ein Beispiel dafür ist std::vector.

Dieses Beispiel funktioniert nur wenn „operator<<(ostream &, vec const &)“ in den Namespace plaziert wird, weil beide Argumente in std sind. Dies ist aber gemäss Standard nicht erlaubt.

```
using std::vector;                                intuition.cpp
using std::ostream;
using vec = vector<int>;
using outv = std::ostream_iterator<vec>;
using out = std::ostream_iterator<int>;

namespace std {
ostream & operator<<(ostream & os, vec const & v) {
  copy(begin(v), end(v), out { os, "," });
  return os;
}
}

void work_only_with_shift_in_ns_std(ostream & os) {
  vector<vec> vv ({ { 1, 2, 3 }, { 4, 5, 6 } });
  copy(begin(vv), end(vv), outv { os, "\n" });
}
```

## Workaround for std:::Resolution Problem

Erstelle einen neuen Typ in dem vom std::vector<int> geerbt wird. Nur ein Alias wäre dafür nicht ausreichend. Es ist nicht empfohlen von Standard-Containeren im Allgemeinen abzuleiten.

```
using std::ostream;                                workaround.cpp
using std::vector;
using out = std::ostream_iterator<int>;

namespace X {
struct vec : vector<int> { // vec is a new type
  using vector<int>::vector; // inherit ctors
};
ostream & operator<<(ostream & os, vec const & v) {
  copy(begin(v), end(v), out{os, ","});
  return os;
}

void works_with_inheriting_ctors(ostream & os) {
  using outv = std::ostream_iterator<X::vec>;
  vector<X::vec> vv({{1,2,3},{4,5,6}});
  copy(begin(vv), end(vv), outv{os, "\n"});
}
```

## Enums

```
enum [class] <name> {
    <enumerators>
};
```

### Enumeration Types

Enumerationen sind nützlich um Typen zu repräsentieren, welche nur wenige Werte haben. Eine Enumeration kann einfach in einen integralen Typ umgewandelt werden. Der Weg zurück ist aber nicht direkt möglich. Die individuellen Werte sind im Typ spezifiziert. Wenn es nicht explizit angegeben werden, starten die Werte bei 0 und werden um 1 vergrössert.

```
enum class day_of_week {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
}; 0   1   2   3   4   5   6
```

### Scopes of Enumerators

#### Unscoped enumeration (no class keyword)

Die Enumeratoren leaken in den umgebenen Scope. Das heisst, sie sind direkt verfügbar. Am besten wird es als Member einer Klasse genutzt.

```
namespace date {

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

bool is_weekend(date::day_of_week day)
{
    return day == date::Sat || day == date::Sun;
}
```

#### Scoped enumeration (class keyword)

Die Enumeratoren leaken nicht in den umgebenen Scope. Das heisst, dass der Name des Enums noch angegeben werden muss.

```
namespace date {

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

bool is_weekend(date::day_of_week day)
{
    return day == date::day_of_week::Sat || day == date::day_of_week::Sun;
}
```

## Enumeration Conversion

Die Operatoren können auch für Enums überladen werden. Beispiele dafür sind folgende:

```
enum dayOfWeek {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

dayOfWeek operator++(dayOfWeek & aday) {
  int day = (aday + 1) % (Sun + 1);
  aday = static_cast<dayOfWeek>(day);
  return aday;
}

dayOfWeek operator++(dayOfWeek & aday, int) {
  dayOfWeek ret{aday};
  if (aday == Sun)
    aday = Mon;
  else
    aday = static_cast<dayOfWeek>(aday + 1);
  return ret;
}
```

### Prefix increment

```
dayOfWeek operator++(dayOfWeek &)
```

### Postfix increment

```
dayOfWeek operator++(dayOfWeek &, int)
```

### Implicit conversion from enum to int

#### Explicit conversion from int to enum

```
dayOfWeek tuesday =
static_cast<dayOfWeek>(1);

int day = Sun;
```

## Defining Values of Enumerators

Mit = können Werte für Enumeratoren angegeben werden. Nachfolgende Enumeratoren erhalten einen um jeweils eins grösseren Wert. Dabei können verschiedenen Enumeratoren dieselben Werte haben.

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

In einigen Fällen werden Enumerationen dazu verwendet um Bitmasken zu erstellen. Die Werte sind eine Potenz von 2.

```
enum file_permissions {
    readable = 1,
    writeable = 2,
    executable = 4
};
```

## Operator Overloading for Enumerations

Die Enumeratoren werden nicht automatisch auf ihren Originalnamen gemappt. Vielleicht kommt dies in einem späteren Standard. Man kann aber eine Lookup Table zur Verfügung stellen und den Outputoperator(<<) überladen.

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

## Specifying the Underlying Type

Die Enumerationsklassen können den zugrunden liegenden Typ durch Vererbung angeben. Der zugrundeliegende Typ kann ein Integraler Typ sein. Dies ermöglicht die Vorwärtsdeklaration von Enumerationen. Das Feature kann dazu verwendet werden um Implementationsdetails zu verstecken, wenn es als Klassenmember definiert ist.

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

## Example

```
#include "Statemachine.h"      statemachine.cpp
#include <cctype>
enum class Statemachine::State : unsigned short {
    begin, middle, end
};
Statemachine::Statemachine()
: theState {State::begin} {}

void Statemachine::processInput(char c) {
    switch (theState) {
        case State::begin:
            if (!isspace(c)) theState = State::middle;
            break;
        case State::middle:
            if (isspace(c)) theState = State::end;
            break;
        case State::end:
            break; // ignore input
    }
}

bool Statemachine::isDone() const {
    return theState == State::end;
}
```

```
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

struct Statemachine {
    Statemachine();
    void processInput(char c);
    bool isDone() const;
private:
    enum class State : unsigned short;
    State theState;
};

#endif /* STATEMACHINE_H_ */
```

# Arithmetic Types (Ring5)

Disclaimer: In der Regel wollen sie nicht ihre eigenen Arithmeticotypen implementieren. Zukünftige Standards werden zusätzliche arithmetische Typen zu Verfügung stellen (unbounded large integer und rational).

## Arithmetic Types

Hier die Basics, falls wir dennoch unsere eigenen Typen implementieren wollen.

### Ring5: Arithmetic Modulo 5

```
struct Ring5 {
    explicit Ring5(unsigned x = 0u)
        : val{x % 5} {}
    unsigned value() const {
        return val;
    }
private:
    unsigned val;
};
```

Die Membervariable ist im Range von 0 bis 4.

Es findet Zugriff auf den Wert statt.

Die Konstruktor ist explizit angegeben.

## Comparison of Ring5

Die Arithmetischen Typen müssen auf die Gleichheit hin vergleichbar sein. Also der Operator ==. CUTE benötigt diesen Operator in ASSERT\_EQUAL.

```
void testValueCtorWithLargeInput() {
    Ring5 four{19};
    ASSERT_EQUAL(Ring5{4}, four);
}
```

Boost kann verwendet werden um einfach == zu bekommen. Benutze dazu boost::equality\_comparable.

```
struct Ring5 :
    boost::equality_comparable<Ring5> {
    bool operator==(Ring5 const & r) const {
        return val == r.val;
    }
    //...
};
```

## Operators for Ring5

### Output Operator

Es könnte bequem sein, wenn man den Outputoperator (<< ) hat um ein Ring5 auszugeben. Wie immer darf dies nicht als Class Member implementiert sein.

CUTE braucht diesen Operator für schöne Fehlernachrichten.

```
std::ostream & operator<<(
    std::ostream & out,
    Ring5 const & r) {
    out << "Ring5{" << r.value() << '}';
    return out;
}
```

```
void testOutputOperator() {
    std::ostringstream out{};
    out << Ring5{4};
    ASSERT_EQUAL("Ring5{4}", out.str());
}
```

## Plus Operation

Das Resultat muss natürlich auch in einem Range von 0 bis 4 liegen. Zum Beispiel  $(4+4) = 8 \% 5 = 3$ . Dazu müssen die beiden Operatoren + und += selbst implementiert werden.

Alternativ kann auch boost verwendet werden und dann wird nur der Operator += benötigt.

```
struct Ring5 :  
    boost::equality_comparable<Ring5>,  
    boost::addable<Ring5> {  
    Ring5 operator+=(Ring5 const & r) {  
        val = (val + r.val) % 5;  
        return *this;  
    }  
    //...  
};
```

```
void testAdditionWrap() {  
    Ring5 four{4};  
    Ring5 three = four + four;  
    ASSERT_EQUAL(Ring5{3}, three);  
}
```

## Multiplication Operation

Ein Beispiel für die selbstimplementierte Multiplikation. Der Operator \*= als Memberfunktion und der Operator \* als free(inline) Funktion.

Modulo wird nur im \*= Operator gebraucht. Dies verhindert Code Duplication.

```
struct Ring5 :  
    boost::equality_comparable<Ring5>,  
    boost::addable<Ring5> {  
    Ring5 operator*=(Ring5 const & r) {  
        val = (val * r.val) % 5;  
        return *this;  
    }  
    //...  
};
```

```
inline Ring5 operator*(  
    Ring5 l, Ring5 const & r) {  
    l *= r;  
    return l;  
}
```

## Mixed Arithmetic

Was wenn wir Ring5 und int zusammenzählen möchten?

Entweder implementieren wir alle Kombinationen von Parameter für den Operator + (operator+(Ring5, int), operator+(int, Ring5)) oder wir machen den Konstruktor nicht mehr explizit.

Das Letztere kann aber zu einem Problem führen, wenn wir auch einen automatischen Konversion zu einem unsigned möchten.

Four sollte den Wert 4 haben und auch vom Typ Ring5 sein.

```
void test_AdditionWithInt_ValueIsFour() {  
    Ring5 two{2};  
    auto four = two + 2u;  
    ASSERT_EQUAL(Ring5{4}, four);  
}  
  
void test_AdditionWithInt_TypeIsRing5() {  
    Ring5 two{2};  
    auto four = two + 2u;  
    ASSERT_EQUAL(typeid(Ring5).name(),  
                typeid(decltype(four)).name());  
}
```

Overhead mit Code, der Duplizierung und der Gefahr der falschen Implementierung  
Testen hilft hier sehr viel. Es könnte sein, dass hier der Overload der Operatoren vergessen geht.

```
inline Ring5 operator+(Ring5 const & l, unsigned r) {
    return Ring5{l.value() + r};
}

inline Ring5 operator+(unsigned l, Ring5 const & r) {
    return Ring5{l + r.value()};
}
```

Der nicht-explizite Konstruktor bietet einen automatischen innere Conversion

- Type conversion operator
  - operator <type>() const member function
  - explicit preferred but requires static\_cast

```
struct Ring5 {
    Ring5(unsigned x) : val{x % 5} {}
    operator unsigned() const {
        return val;
    }
    //...
};
```

# Standard Containers STL

Vector, array, string, deque, stack, queue, set, multiset, map, multimap, unordered\_set, unordered\_map, algorithmus

## Categories of STL Containers

### Sequence Containers

Die Elemente sind in derselben Folge zugänglich wie Sie eingefügt bzw. erstellt wurden. Die Suche ist in linearer Zeit ( $O(n)$  Laufzeit) realisierbar. Im schlimmsten Fall müssen alle Elemente durchsucht werden.

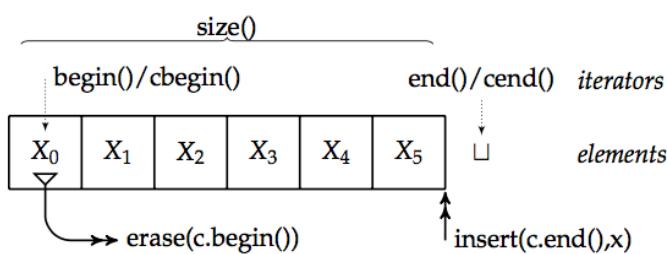
### Associative Containers

Die Elemente sind in sortiertet Ordnung zugänglich. Dadurch lassen sich Suchalgorithmen mit einer logarithmischen Laufzeit realisieren.

### Hashed Containers (unordered associative)

Die Elemente sind in irgendeiner Folge zugänglich (nicht definiert). Die Suche kann als Memberfunktion mit einer kostanten Zeit realisiert werden.

## Common Features of Containers

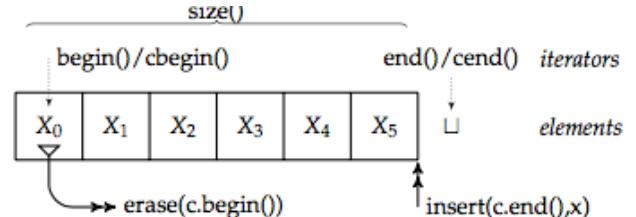


Alle Container haben das gleiche / ein ähnliches Interface.

Iteratoren für den Algorthmus und die Iteration, Erase(iter) für die Elementlöschung, Insert(iter, value) um Elemente einzufügen und Size() und empty() um die Grösse abzufragen.

## Common Container API

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



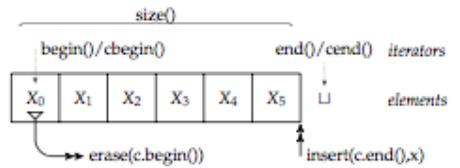
- Containers können mit Default konstruktet werden (Default-Konstruktor)
- Containers können kopiert werden von einem Container des selben Typs (Copy-Konstruktor)
- Containers vom selben Typ können auf Gleichheit verglichen werden
  - o Einige können sogar lexographisch verglichen werden mit relationalen Operatoren
- Containers können mit clear() leer gemacht werden.

## Common Container Constructors

```
std::vector<int> v{1,2,3,5,7,11};  

std::list<int> l(5,42);  

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



- Konstruktion mit einer Initializer Liste (gibt dafür einen eigenen Konstruktor)
- Konstruktion mit einer Anzahl von Elementen
  - o Ein Default Value kann mitgegeben werden
  - o Hier sollen immer () verwendet werden. Die {} haben sonst eine zweie Deutigkeit bei gewissen Datentypen.
- Konstruktion von einem Range mit einem Paar Operatoren
  - o Auch hier müssen teilweise () verwendet werden. Die im Falle, wenn es eine Liste von Iteratoren wäre.

## STL Iterator Categories

```
struct input_iterator_tag { };  

struct output_iterator_tag { };  

struct forward_iterator_tag : public input_iterator_tag { };  

struct bidirectional_iterator_tag : public forward_iterator_tag { };  

struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

Die verschiedenen Container unterstützen Iteratoren mit verschiedenen Möglichkeiten.

### Input Iterator

```
struct input_iterator_tag { };  

T const operator *()  

operator++() Prefix  

operator++(int) Postfix  

operator==(myiter) // and != Iteratoren
```

Unterstützt das lesen vom aktuellen Elementm aber nur einmal (\*it). Anschliessend muss der Iterator mit ++ incrementiert werden.

Dieser Iterator erlaubt Algorithmen mit einem One-Pass Eingang.

Es modelliert std::istream\_iterator und std::istreambuf\_iterator. Ein Vergleich von Iteratoren mit == und != ist möglich. Die meisten anderen Iteratoren sind auch Input Iteratoren.

### Forward Iterator

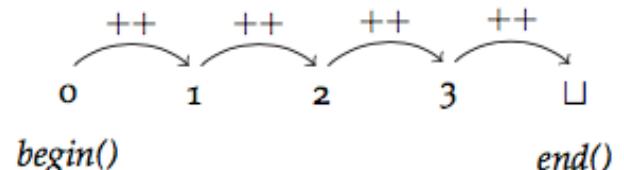
```
struct forward_iterator_tag { };  

T & operator *() const  

operator++()  

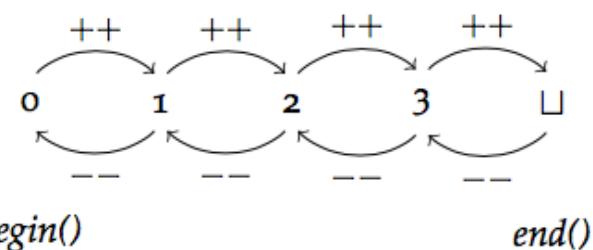
operator++(int)  

operator==(myiter) // and !=
```



Grundsätzlich, sicher mal alles was der Input Iterator auch unterstützt. Es unterstützt das Lesen und das ändern des aktuellen Elementes (\*it). Es sei denn, der Container oder die Elemente sind const. Er erlaubt One-Pass Algorithmen (++it). Schritte rückwärts sind nicht möglich. Es ist aber möglich eine Iterator Kopie zu halten für den späteren Gebrauch. Er modelliert die std::forward\_list Iteratoren. Viele andere Iterator sind auch Forward Iteratoren.

```
struct forward_iterator_tag { };
T & operator *() const
operator++()
operator++(int)
operator--()
operator--(int)
operator==(myiter)
operator!=(myiter)
```



Mehr oder weniger, was der Forward Iterator macht, zusätzlich sit aber rückwärts auch noch möglich. Das Lesen und das ändern vom aktuellen Wert ist möglich. Es unterstützt zwei Weg Algorithmen. Zum Beispiel modelliert er die std::set Iteratoren. Random Access Iteratoren sind ebenfalls bidirektionale Iteratoren, unterstützt aber auch indexing [].

### Special Case: Output Iterator

```
struct output_iterator_tag { };
T& operator *()
operator++()
operator++(int)
```

Kann einen Wert in das aktuelle Element schreiben, aber nur einmal (\*it=value). Anschliessend ist ein Increment notwendig. Modelliert für den ostream\_iterator. Die meisten anderen Iteratoren können ebenfalls als Output Iteratoren agieren, wenn der zugrunde liegende Container nicht const ist.

### Why these iterator categories?

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);

template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

Einige Algorithmen funktionieren mit mächtigen Iteratoren wie zum Beispiel sort, welcher ein Paar von Random Access Iteratoren voraussetzt. Einige Algorithmen funktionieren besser, wenn mir mächtige Iteratoren einsetzen (std::advance).

### <iterator> functions

```
std::distance(beg, end);
std::advance(it, n);
```

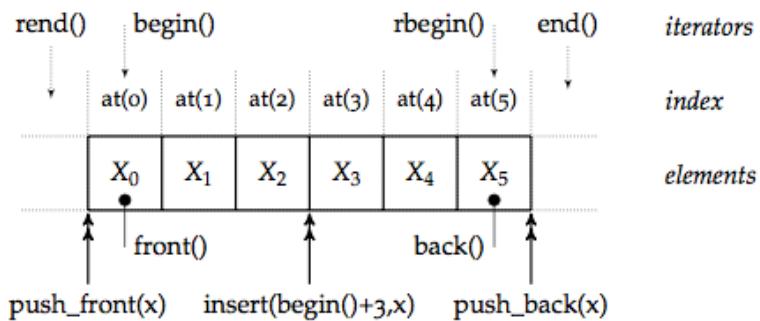
distance() zählt die Anzahl von „Hops“ betteln muss bis er das Ende erreicht. Effizient ist es für Random Access Iteratoren, andernfalls ist es einfach ein Loop.

Advance() lässt n mal den Iterator hoppen. Dies ist ebenfalls effizient für Random Access Iteratoren. Andernfalls ist es ein Loop. Es ist auch negativ möglich bei bidirektionalen Iteratoren.

## Sequence Containers

**std::vector<int>, std::deque<int>,  
std::list<int>, std::forward\_list<int>**

Vergleichbar: **LinkedList**



Definiert eine Ordnung von Elementen wie sie eingefügt wurden.

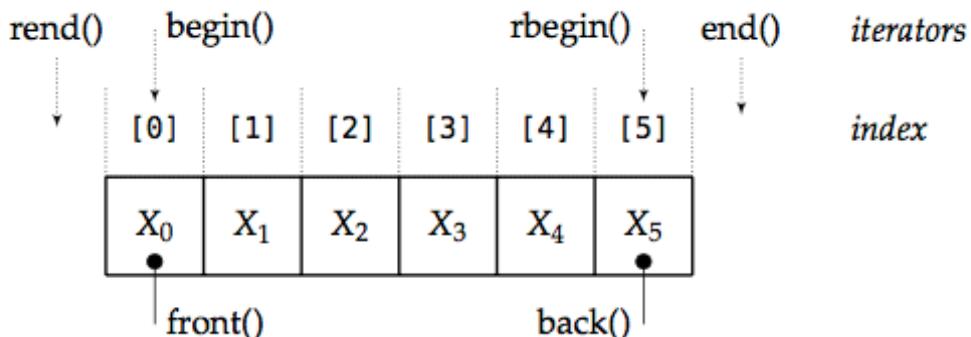
Die Listen sind gut für Splicing und für das mittlere Einfügen.

Vector/Deque sind effizient, ausser bei schlechter Verwendung.

## Array

**std::array<int, 6> a{{1,1,2,3,5,8}};** double braces{{...}}

Der Array zerfällt zu einem Pointer-,

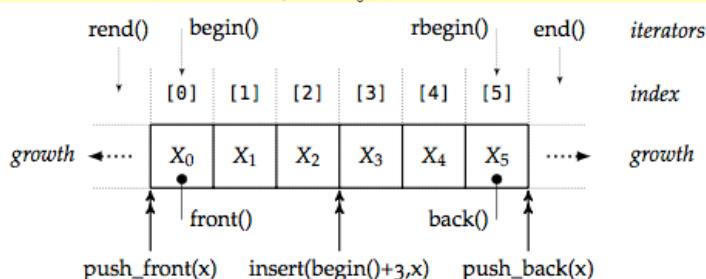


Es ist ein Container mit einer fixen Grösse. Für die Initialization ist eine spezielle Syntax vorhanden. Die Initialization kann nur Compilezeit geschehen. Nutzen Sie `std::arrays` anstatt C-Style arrays, wenn Sie arrays definieren.

## Double-ended Queue – `std::deque`

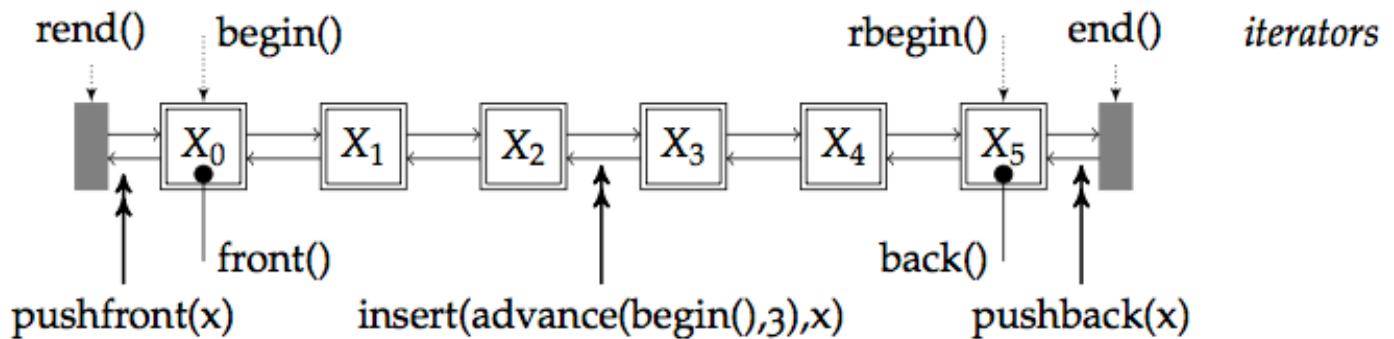
**std::deque<int> q{begin(v), end(v)};  
q.push\_front(42);q.pop\_back();**

Zusätzliche Funktionen hier, welche verfügbar sind.



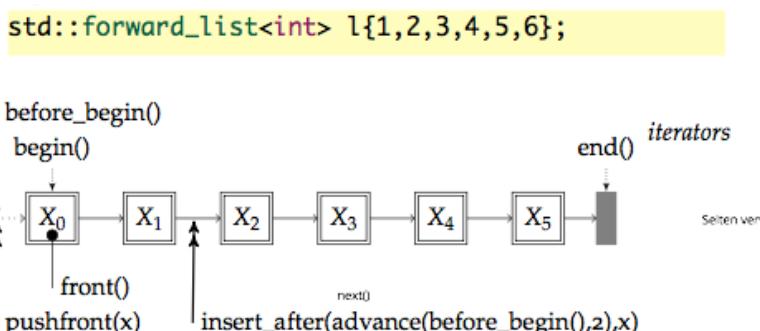
Deque ist wie `std::vector` aber mit einer effizienten Front Einfügung sowie Löschalgorithmen. (`push_front()`/`pop_front()`). Zudem ist ebenfalls `push_back()` und `pop_back()` vorhanden.

```
std::list<int> l(5,1);
```



Ist effizient beim Einfügen an jeder Position. Weniger effizient ist es aber in der Ausführung von Bluk Operationen. Es benötigt Memberfunktionen für sort, etc. Es sind nur bidirektionale Iteratoren vorhanden, aber kein indexed Zugriff.

### Singly-linked List: std::forward\_list



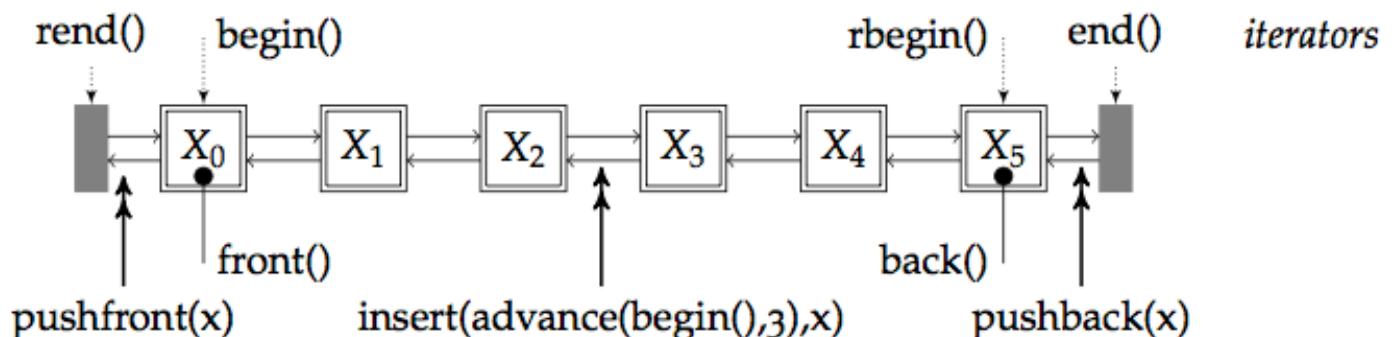
Effiziente Einfügung nach jeder Position, aber ungeschickt mit Iterator bei before.

Nur Forward-Iteratoren, ungeschickt für die Suche sowie für das Löschen.

Es ist selten verwendet. Wenn immer möglich lieben den Vector oder eine Liste verwenden.

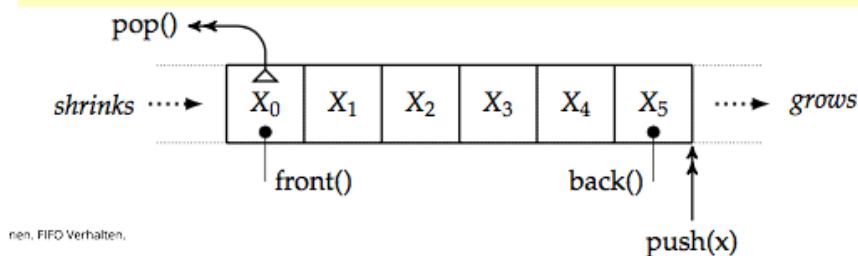
### LIFO Adapter - std::stack

```
std::list<int> l(5,1);
```



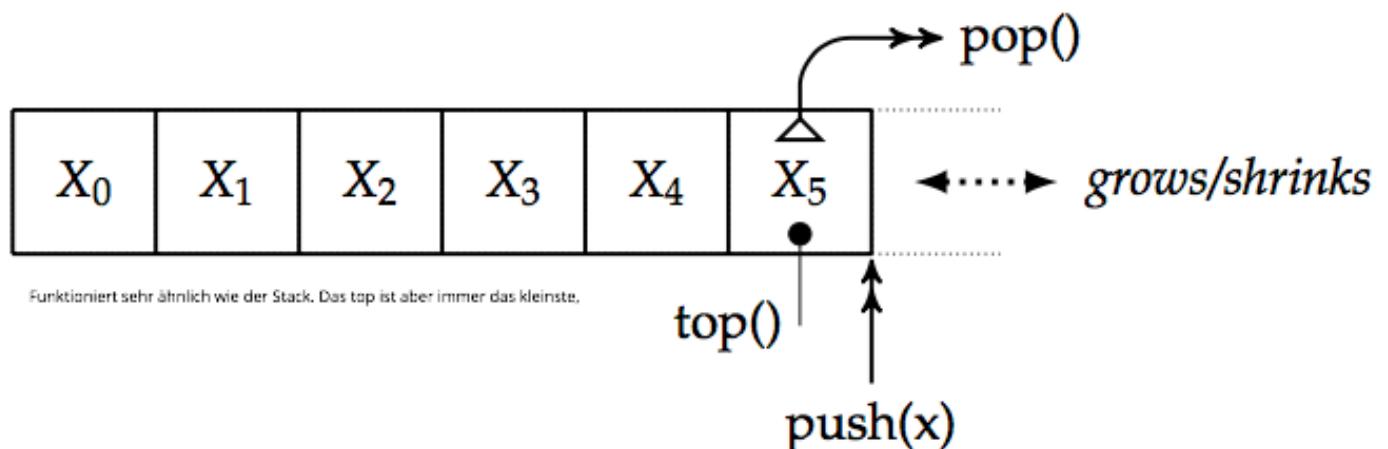
Nutzt std::deque (oder vector oder list) und limitieren die Funktionalität auf die Stack Operationen. Es sind keine Iteratoren möglich. Pop ist void und es wird also hier nichts zurückgegeben. Top ist für die Ausgabe zuständig. Es nutzt im Hintergrund push\_back(), back() und pop\_back().

```
std::queue<int> q{};
q.push(42); std::cout << q.front(); q.pop();
```



Nutzt std::deque (oder vector oder list) und limitiert die Funktionalität auf die Warteschlangen Operationen. Es sind keine Iteratoren möglich. Pop ist void und es wird also hier nichts zurückgegeben. Top ist für die Ausgabe zuständig. Es nutzt im Hintergrund push\_back() und front().

### Adapter - std::priority\_queue



Funktioniert ähnlich wie der Stack, aber top ist immer das kleinste Element. Nutzt std::deque oder vector und limitiert die Funktionalität, aber es hält die Elemente sortiert als binären Heap.

### Example Stack and queue

Es zeigt den Unterschied zwischen den beiden Typen sehr gut.

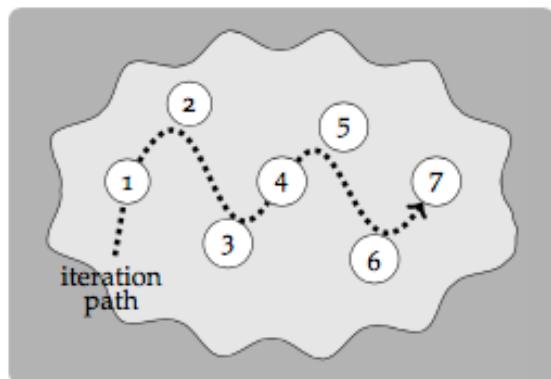
```
#include <stack>
#include <queue>
#include <iostream>
int main(){
    std::stack<std::string> lifo;
    std::queue<std::string> fifo;
    for(std::string s
        : {"Fall", "leaves", "after", "leaves", "fall"}){
        lifo.push(s);
        fifo.push(s);
    }
    while (!lifo.empty()){
        std::cout << lifo.top() << " ";
        lifo.pop();
    } // fall leaves after leaves Fall
    std::cout << "\n";
    while (!fifo.empty()){
        std::cout << fifo.front() << " ";
        fifo.pop();
    } // Fall leaves after leaves fall
}
```

## Associative Containers aka: Sorted (Tree-) Containers

Erlaubt die Suche nach dem Inhalt und nicht nach der Sequenz. Also eine Suche beim Schlüssel und der Zugriff über das Schlüssel-Werte Paar. Die Schlüssel können einzigartig sein (Set, map) oder mit mehreren gleichen Schlüssel (multiset, multimap).

Set of element – std::set

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



Haltet die Elemente in sortierterer Reihenfolge in einem Baum. Die Sortierung kann mit einem zweiten Template typename parameter redefiniert werden.

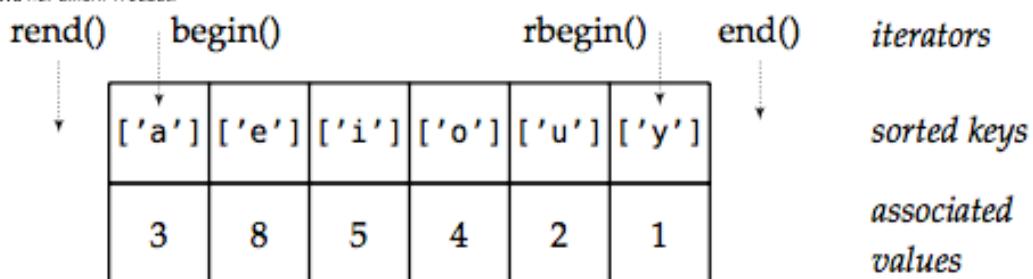
Es nutzt Memberfunktionen für find() und count(). Die Suche ist nicht sequentiell basiert sondern Tree basiert. Count kann nur 1 oder 0 zurückgeben, da die Elemente ja nicht mehrfach vorkommen.

```
#include <set>
#include <iostream>
void filtervowels(std::istream &in, std::ostream &out){
    std::set<char> const vowels{'a','e','o','u','i','y'};
    char c{};
    while (in>>c)
        if (!vowels.count(c))
            out << c;
}
int main(){
    filtervowels(std::cin, std::cout);
}
```

Map – std::map<key, value>

```
std::map<char, size_t> vowels
{{'a',0}, {'e',0}, {'i',0}, {'o',0}, {'u',0}, {'y',0}};
```

Entspricht etwa von Java her einem TreeSet.



Haltet Schlüssel-Werte Paare in sortierter Reihenfolge. Das Ordering kann mit einem 3ten Parameter definiert werden. Iteratoren grefien auf std::pair<key,value> zu. Mit first kann man auf den Key zugreifen, mit second auf den Wert.

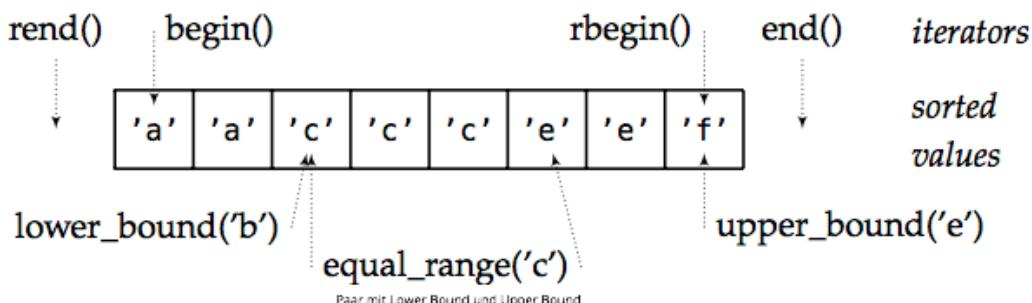
```
#include <map>
#include <iostream>
#include <iterator>
int main(){
    std::map<char,size_t> vowels{{'a',0},{'e',0},{'i',0},{'o',0},{'u',0},{'y',0}};
    char c{};
    while (std::cin >> c)
        if (vowels.count(c)) // only count those chars that are already in the map
            ++vowels[c]; Auf den Wert in den Pairs zugreifen.
    for(auto const &p:vowels) // p is a pair<char,size_t>
        std::cout << p.first << " = " << p.second << '\n';
}
```

```
#include <map>
#include <iostream>
#include <iterator>
int main(){
    std::map<std::string,size_t> words;
    std::string s;
    while (std::cin >> s)
        ++words[s]; --> wenn es nicht vorhanden ist in der Map, wird ein neues hinzugefügt zur Map mit Default Werten.
    for(auto const &p:words)
        std::cout << p.first << " = " << p.second << '\n';
}
```

Wenn Schlüssel nicht vorhanden ist, wird automatisch ein neues Element in die Map hinzugefügt. Map ist natürlich automatisch auch sortiert. Umgangssprachlich entspricht es dem Dictionary.

## Multiset / Multimap

```
std::multiset<char> letters{};
```



Mehrere gleiche Keys sind erlaubt. Nutzen Sie equal\_range or lower\_bound/upper\_bound Funktionen um die gleiche Keys zu finden. Kann ein bisschen langweilig sein damit zu arbeiten, anstatt mit std::set.

```
#include <set>
#include <iostream>
#include <iterator>
int main(){
    using in=std::istream_iterator<std::string>;
    using out=std::ostream_iterator<std::string>;
    std::multiset<std::string> words{in{std::cin},in{}};
    copy(cbegin(words),cend(words),out(std::cout,"\\n"));
    auto bw=cbegin(words);
    while(bw!=cend(words)){
        auto ew=words.upper_bound(*bw); // end of range
        copy(bw,ew,out{std::cout,", "});
        std::cout << '\\n'; // next range on new line
        bw=ew;
    }
}
```

## Hashed Containers – `unordered_set`

Diese sind effizienter im Lookup, keine Suche. Für alle eigenen Implementationen muss die eigenen Hash-Funktion implementiert werden. Für jene im Standard sind Sie bereits definiert. Eigene Hashfunktionen ist schwer auf Fehler zu prüfen. So produzieren Sie vielleicht zu viele Kollisionen.

### Example – `unordered_set`

```
#include <unordered_set>
#include <iostream>
#include <iterator>
#include <algorithm>
int main(){
    std::unordered_set<char> const vwl{'a','e','i','o','u'};
    using in=std::istreambuf_iterator<char>;
    using out=std::ostreambuf_iterator<char>;
    remove_copy_if(in{std::cin},in{},out{std::cout},
                  [&](char c){return vwl.count(c);});
}
```

Fast äquivalent mit `std::set`, aus das Fehlen von der Sortierung. `Unordered_set` sollte bei den eigenen Typen nicht gebraucht werden, auch wenn sie Experten sind und es Speedvorteile geben würde.

### Example – `unordered_map`

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

Die Arbeitsweise entspricht etwa der `std::map`, aber ohne die Sortierung. `Unordered_map` sollte bei den eigenen Typen nicht gebraucht werden, auch wenn sie Experten sind und es Speedvorteile geben würde.

# STL Algorithms

Why should we use algorithms?

## Correctness

```
void reverse(std::vector<int> & values) {
    for (int i = 0; i <= values.size(); i++) {
        auto otherIndex = values.size() - i;
        auto tmp = values[0];
        values[i] = values[otherIndex];
        values[otherIndex] = tmp;
    }
}
```

```
void reverse(std::vector<int> & values) {
    reverse(begin(values), end(values));
}
```

Es ist viel einfacher einen Algorithmus korrekt zu benutzen, als einen Loop korrekt zu implementieren.

## Readability

```
int ???????(std::vector<int> values) {
    int var = std::numeric_limits<int>::min();
    for (auto v : values) {
        if (v > var) {
            var = v;
        }
    }
    return var;
}
```

```
int ???????(std::vector<int> values) {
    if (values.empty()) {
        return std::numeric_limits<int>::min();
    }
    return *std::max_element(begin(values), end(values));
}
```

Die Anwendung des richtigen Algorithmus drückt ihre Absicht viel besser aus als eine Schleife. Jemand anderes (oder sogar Sie) wird es schätzen, wenn der Code lesbar und leicht verständlich ist.

## Performance

```
bool contains(std::vector<int> values, int sought) {
    for (auto it = begin(values); it != end(values); it++) {
        if (*it == sought) return true;
    }
    return false;
}
```

```
bool contains(std::vector<int> values, int sought) {
    return find(begin(values), end(values), sought) != end(values);
}
```

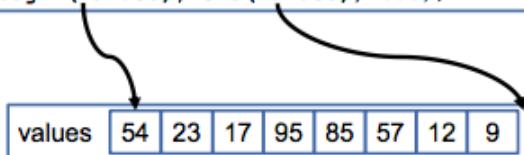
Algorithmen performen meist besser als hangeschriebene Lopps.

## Algorithm Basics

### Iterators for Ranges

Iteratoren geben Ranges an, welche für die Algorithmen verwendet werden. Der First-Iterator zeigt auf das erste Element im Range, während der Last-Iterator auf das letzte Element im Range zeigt. Wenn First == Last, dann ist der Range leer.

```
std::vector<int> values{54, 23, 17, 95, 85, 57, 12, 9};
std::xxx(begin(values), end(values), ...);
```



## Iterators as Output of Ranges

Streams brachen einen Wrapper um mit Algorithmen verwendet werden zu können.

```
std::ostream -> std::ostream_iterator<T>
std::istream -> std::istream_iterator<T>

Default-constructed std::istream_iterator<T> marks EOF
```

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

## Algorithm- for\_each

Ausführen von einer Operation für jedes Element in einem Range. Die Operation ist eine Funktion, ein Lambda oder ein Functor, welches für jedes Element aufgerufen wird. Für den Range werden Input Iteratoren verwendet.

```
std::vector<unsigned> values{3, 0, 1, 4, 0, 2};
auto f = [](unsigned v) {};
std::for_each(begin(values), end(values), f);
```

## Reading Algorithm Signatures

Signature of std::for\_each from [cppreference.com](http://cppreference.com):

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

Jedes Algorithmus hat einen Namen, Parameter und eine Return Type, welcher auch void sein kann. Die Beschreibung spezifiziert die Anforderungen. Parameter und Return Typen sind normalerweise Templateparameter. Algorithmen funktionieren mit folgenden Iteratorenkategorien:

- Input Iterator
- Forward Iterator
- Bidirectional Iterator
- Random access Iterator
- Output Iterator

```
struct Accumulator {
    int count{0};
    int accumulated_value{0};
    void operator()(int value) {
        count++;
        accumulated_value += value;
    }
    int average() const;
    int sum() const;
};

int average(std::vector<int> values) {
    Accumulator acc{};
    for(auto v : values) { acc(v); }
    return acc.average();
}
```

Ein Functor ist ein Typ(Klasse) welche den Calloperator zur Verfügung stellt. (operator()). Ein Objekt/Instanz von diesem Typ kann wie eine Funktion aufgerufen werden. Es können mehrere Overloads von diesem Parameter angegeben werden. Es können unendliche Parameter entgegengenommen werden.

```
int average(std::vector<int> values) {
    Accumulator acc{};
    return std::for_each(begin(values), end(values), acc).average();
}
```

## Predicates

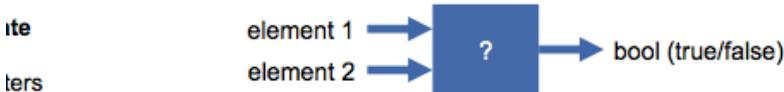
Eine Funktion oder Lambda, welche bool zurückgibt (oder eine Typen, der in einen Bool umgewandelt werden kann). Sind dazu da um Bedingungen zu prüfen.

### Unary Predicate (Ein Parameter)



```
auto is_odd = [](int i) {return i % 2 == 0;};
```

### Binary Predicate (Zwei Parameter)

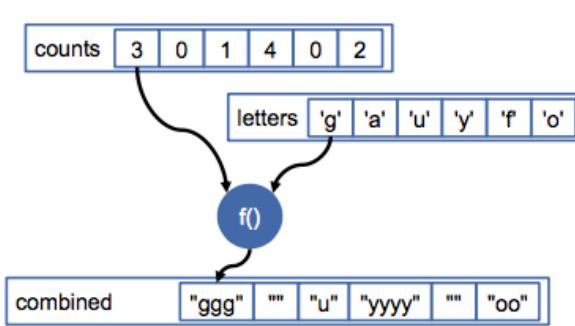


```
auto divides = [](int i, int j) {return !(i % j);};
```

## Algorithm Examples

### Transform

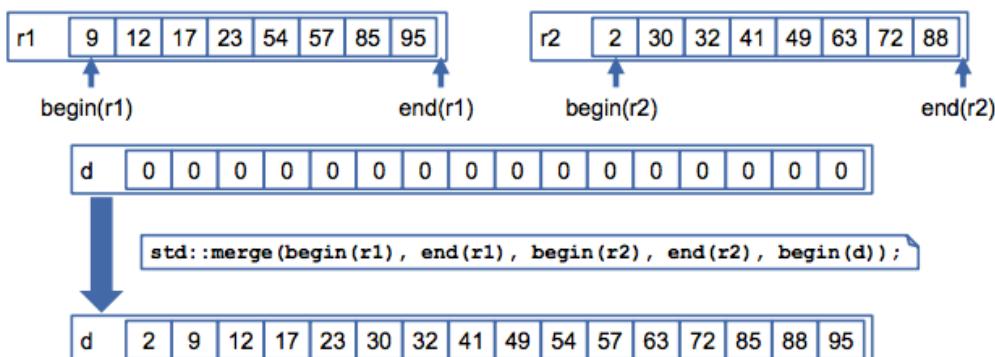
Mappen von einem Range auf neue Werte oder zwei Ranges mit der gleichen Grösse. Dabei wird eine Lambda/function/functor für die Map Operation verwendet. Die Input und Outputtypen können unterschiedlich sein, solang die Operation die korrekten Typen hat.



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

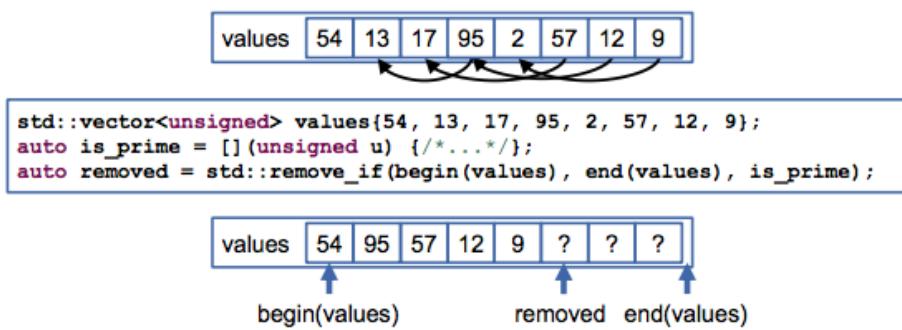
Zusammenführen von zwei sortierten!!! Ranges.

```
std::vector<int> r1{9, 12, 17, 23, 54, 57, 85, 95};
std::vector<int> r2{2, 30, 32, 41, 49, 63, 72, 88};
std::vector<int> d(r1.size() + r2.size(), 0);
```



### Erase-Remove-Idiom

`Std::remove` löscht nicht wirklich die Elemente. Es verschiebt die „nicht-gelöschten“ Elemente nach Vorne und gibt einen Iterator über das Ende des „neuen“ Ranges zurück.

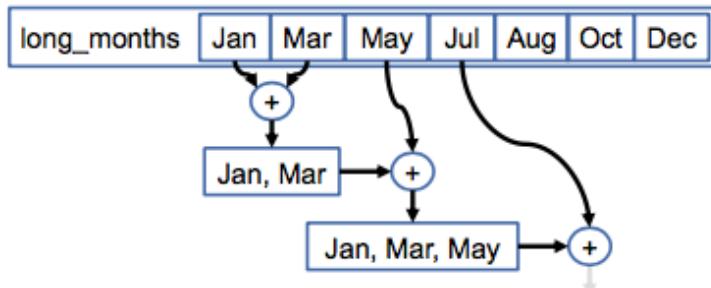


Um wirklich Elemente zu entfernen sollte die „`erase`“ Member Funktion aufgerufen werden.

```
values.erase(remove_if(values.begin(), values.end(), is_prime), values.end());
```

## Numveric Algorithm – accumulate

Einige numerische Algorithmen können auch im nicht-numerischen Kontext verwendet werden. Ein Beispiel dafür ist std::accumulate. Es summiert Elemente, welche addierbar sind (+operator). Oder macht es basierend auf einer eigenen binären Funktion. Zurückgegeben wird die Summe.



```

std::vector<std::string> long_months{"Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"};
std::string accumulated_string = std::accumulate(
    begin(long_months) + 1, //Second element
    end(long_months),       //End
    long_months.at(0),      //First element, usually the neutral element
    [] (std::string const & acc, std::string const & element) {
        return acc + ", " + element;
}); //Jan, Mar, May, Jul, Aug, Oct, Dec
  
```

### \_if Versions

Einige Algorithmen haben eine Variation mit dem „\_if“ Suffix. Diese nehmen ein Prädikat (anstatt eines Wertes) um eine Bedingung zur Verfügung zu stellen.

```

std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto is_prime = [] (unsigned u) { /*...*/ };
auto nOfPrimes = std::count_if(begin(numbers), end(numbers), is_prime);
  
```

## Algorithmen mit dem Suffix

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

### \_n Versions

Einige Algorithmen haben eine Variation mit dem „\_n“ Suffix. Diese Varianten sind in Verbindung mit einer Zahl (meist anstatt des Last Iterators).

```

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

## Algorithmen mit dem „\_n“ Suffix

- |            |              |                      |
|------------|--------------|----------------------|
| ■ search_n | ■ fill_n     | ■ for_each_n (C++17) |
| ■ copy_n   | ■ generate_n |                      |

## Heap Algorithms

Implementiert einen binären Heap auf sequenzed Containern. Dafür ist ein Random Access Iterator notwendig. Der Algorithmus garantiert, dass das oberste Element das grösste Element ist und Adding und Removing haben Leistungsgarantien. Der Algorithmus wird gebraucht um Priority Queues zu implementieren.

### Operationen

`make_heap (3 * N comparisons)`

`pop_heap (2 * log(N) comparisons)`

`push_heap (log(N) comparisons)`

`sort_heap (N*log(N) comparisons)`

### Heap Example

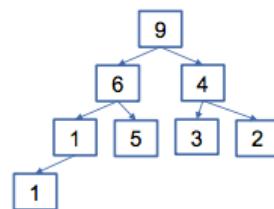
v	3	1	4	1	5	9	2	6
---	---	---	---	---	---	---	---	---

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

### Make\_heap

v	3	1	4	1	5	9	2	6
v	9	6	4	1	5	3	2	1

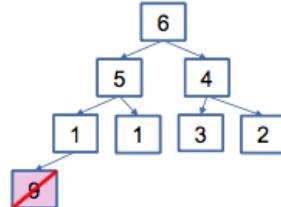
```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```



**Pop\_heap**

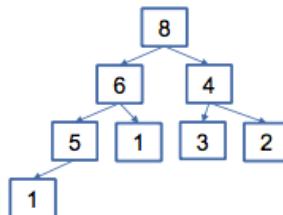
v	3	1	4	1	5	9	2	6
v	9	6	4	1	5	3	2	1
v	6	5	4	1	1	3	2	9
v	6	5	4	1	1	3	2	9

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

**Push\_heap**

v	3	1	4	1	5	9	2	6
v	9	6	4	1	5	3	2	1
v	6	5	4	1	1	3	2	9
v	6	5	4	1	1	3	2	
v	6	5	4	1	1	3	2	8
v	8	6	4	5	1	3	2	1

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

**Sort\_heap**

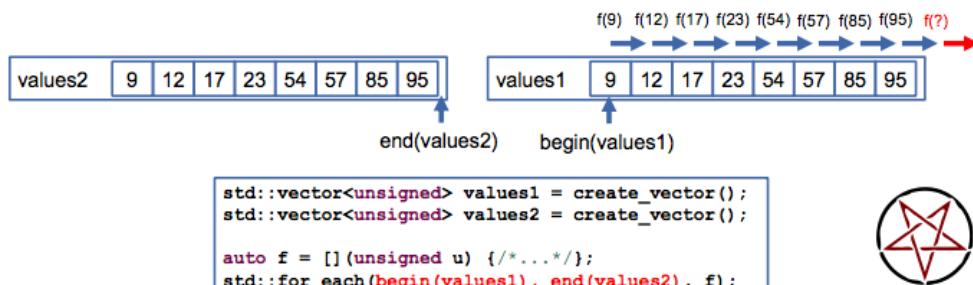
v	3	1	4	1	5	9	2	6
v	9	6	4	1	5	3	2	1
v	6	5	4	1	1	3	2	9
v	6	5	4	1	1	3	2	
v	6	5	4	1	1	3	2	8
v	8	6	4	5	1	3	2	1
v	1	1	2	3	4	5	6	8

```
std::vector<int> v{3,1,4,1,5,9,2,6};
make_heap(v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(8);
push_heap(v.begin(),v.end());
sort_heap(v.begin(),v.end());
```

## Fallstricke

### Mismatching Iterator Pairs

Es ist zwingend erforderlich, dass die Iteratoren, die einen Bereich angeben, zu demselben Bereich gehören müssen. Der Fortschritt des "frist" Iterators muss schließlich den "last" Iterator erreichen (ohne den Bereich zu verlassen). Andernfalls kann der Zugriff auf den Wert zu undefiniertem Verhalten führen.

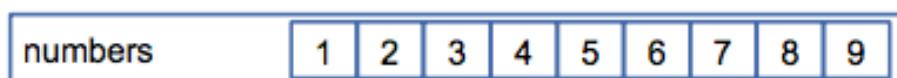


### Reserving Enough Space

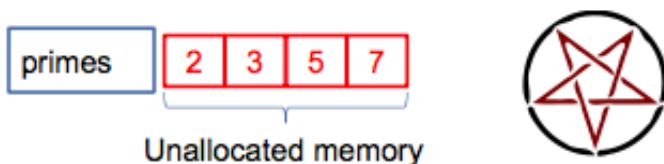
Wenn Sie einen Iterator zur Spezifizierung der Ausgabe für einen Algorithmus verwenden, müssen Sie sicherstellen, dass genügend Speicherplatz zugewiesen wird.

```
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned> primes{};
auto is_prime = [] (unsigned u) { /*...*/};
std::copy_if(begin(numbers), end(numbers), begin(primes), is_prime);
```

Der Vektor „primes“ ist leer, was nicht durch den Iterator verändert wird, welcher als Kopie mitgegeben wird.



Der Output wird dann vielleicht in unallocated Memory kopiert.



## Insert Iterators

Was ist, wenn wir Elemente in einen Container einfügen wollen, ohne den benötigten Speicher vorher zuzuweisen? Es gibt drei Einfügefunktionen für die Verpackung von Behältern, die sich um diesen Fall kümmern:

### Back\_inserter

Erzeugt einen std :: back\_insert\_iterator, der die push\_back-Memberfunktion des Containers verwendet.

### Front\_inserter

Erzeugt einen std :: front\_insert\_iterator, der die push\_front-Memberfunktion des Containers verwendet

### Inserter

Erzeugt einen std :: insert\_iterator, der die insert-Member Funktion des Containers verwendet

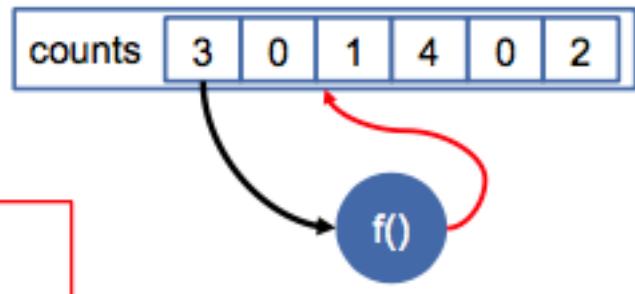
```
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned> primes{};
auto is_prime = [](unsigned u) {/*...*/};
std::copy_if(begin(numbers), end(numbers), back_inserter(primes), is_prime);
```

## Input Validation

Einige Operationen auf Containern machen die Iteratoren ungültig. Am Beispiel von Std::vector<T>::push\_back.

If the new `size()` is greater than `capacity()` then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated.<sup>1</sup>

```
std::vector<unsigned> values{3, 0, 1, 4, 0, 2};
auto f = [&values](unsigned v) {
    values.push_back(v);
};
std::for_each(begin(values), end(values), f);
```



## Algorithm Headers

In the Standard Library

### Algorithms Header

Non-modifying sequence operations

Mutating sequence operations

Sorting and related operations

C library algorithms

### Numerics Header

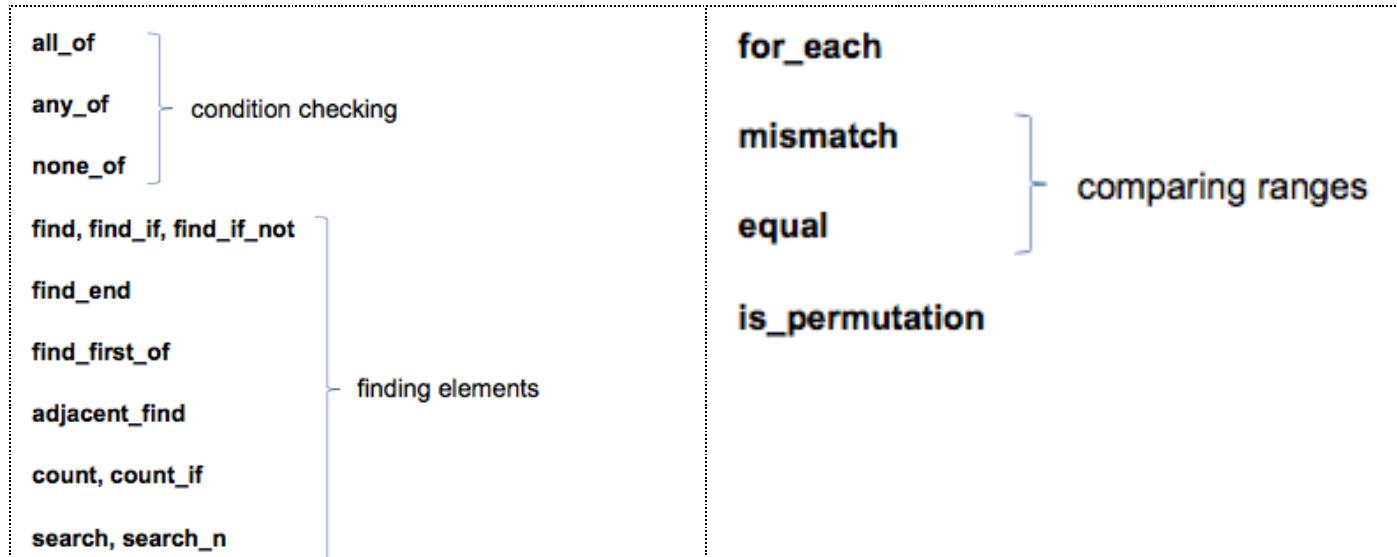
accumulate

inner\_product

partial\_sum

adjacent\_difference

iota



## Mutating sequence operations

<code>copy, copy_n, copy_if, copy_backward</code>	<code>unique, unique_copy</code>
<code>move, move_backward</code>	<code>reverse, reverse_copy</code>
<code>swap_ranges</code>	<code>rotate, rotate_copy</code>
<code>transform</code>	<code>shuffle</code>
<code>replace, replace_if, replace_copy,</code> <code>replace_copy_if</code>	<code>is_partitioned, partition, stable_partition,</code> <code>partition_copy, partition_point</code>
<code>fill, fill_n</code>	
<code>generate, generate_n</code>	
<code>remove, remove_if, remove_copy,</code> <code>remove_copy_if</code>	

## Sorting and Related Operations

<code>sort, stable_sort, partial_sort,</code> <code>partial_sort_copy</code>	<code>push_heap, pop_heap, make_heap,</code> <code>sort_heap</code>
<code>is_sorted</code>	<code>is_heap_until, is_heap</code>
<code>nth_element</code>	<code>min, max, minmax</code>
<code>lower_bound, upper_bound, equal_range</code>	<code>min_element, max_element,</code> <code>minmax_element</code>
<code>binary_search</code>	<code>next_permutation, prev_permutation</code>
<code>merge, inplace_merge</code>	
<code>includes, set_union, set_intersection,</code> <code>set_difference, set_symmetric_difference</code>	

# Functors and Parameterizing STL

Functors, Lambdas, Standard Functors. `Std::function<>`

## Function Objects

```
struct donothingfunctor{
    void operator()()const{}
};
```

Wir können den Funktionsaufrufparameter als Member einer Klasse überladen. → Functor Klasse als Instanz ein Functor Object. Dabei sind ein zwei Paare von Klammern nötig. Normalerweise const-member, es sei denn, die Membervariablen des Funktionsobjekts ändern. Member-Variablen können (im) Speicher behalten (werden)!

```
ret operator()(params) { ... }
```

## Example Functor with memory

```
#include <iostream>
#include <algorithm>
#include <vector>
struct averager {
    void operator()(double d) {
        accumulator += d;
        ++counter;
    }
    double sum() const { return accumulator; }
    double count() const { return counter; }
    double average() const { return sum()/count(); }
private:
    double accumulator{};
    unsigned counter{};
};
int main(){
    std::vector<double> v{7,4,1,3,5,3,4,7};
    auto const res=for_each(v.begin(),v.end(),averager{}); res is resulting averager object
    std::cout << "sum = " << res.sum() << '\n'
           << "count = " << res.count() << '\n'
           << "average = " << res.average() << '\n';
}
```

## Function/Functor Terminology

Funktionen

Dabei gibt es unary functions und binary functions.

Operator Funktionen

Hier gibt es Unary Operatoren und Binary Operatoren.

Functor

In diesem Fall unary Functor und Binary Functor.

Predicate

Funktion/Funktör mit einem bool Resultat.

- **Unary Predicate:** Eine Eigenschaft des Arguments
- **Binary Predicate:** Vergleichen zweier Argumente

```
std::vector<int> v;
int x{}; // memory for lambda below
generate_n(std::back_inserter(v), 10, [&x]{
    ++x; return x*x;
});
```

```
class make_squares{
    int x{};
public:
    int operator() { ++x; return x*x; }
};
//...
generate(v.begin(), v.end(), make_squares());
```

Der generate()-Algorithmus benötigt eine Funktion mit einem „Speicher“, wenn nicht alle Werte die gleichen sein sollen (nullary function, no parameters). Das Lambda erfordert eine Variable erfasst mit & oder ein mutable Lambda.

## Lambda Function Syntax

### Defining Inline Functions

definition

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

call

auto const für die Funktionsvariable von Lambda. Dies wird gebraucht um die Funktion aufzurufen. [] stellt die Lambdafunktion vor. Es kann „captures“ beinhalten um auf die Variablen zugreifen zu können. (parameters) gleich wie bei anderen Funktionen, sie können aber auch auto sein. Schlussendlich gibt es ein Body Block mit den Statements.

## Recap Lambda Rules

[ capture ] (parameters) mutable<sub>opt</sub>->return\_type{body}

**Capture** [=], [&], [var=value]

**Parameters** auto p1, int p2, auto ....

**Mutable** Erlauben es die Werte der Capture Variablen zu verändern.

**Return Type** Typischerweise automatisch abgeleitet

**Body** Die Return Statements stellen den Return Type implizit zur Verfügung.

## Lambda Captures

**Capture:**

- [=] - default implicit capture variables used in body by value
- [&] - default capture variable used in body by reference
- [var=value] - introduce new capture variable with value
- combinations of specific captures with a default
- [=,&out] - capture all by copy, but out by reference
- [&,=x] - capture all by reference, but x by copy/value

**Guideline: always capture variables explicitly**

```
generate_n(std::back_inserter(v),10,[=0]() mutable{
    return ++x , x*x; // mutable allows change
});
```

allow changing x  
introduce new capture variable x

Variablen gefangen von Copy (=) sind const innerhalb des lambdas, es sei denn:

- Das Lambda ist als mutable markiert und
  - o das Lambda bekommt eine eigene Kopie der Variable oder
  - o das Lambda definiert sein Capture mit einem Initialisator.

Lambdas werden intern auf Funktoren gemappt.

### Lambdas are functors

Der C++ Complier mappt intern die Lambdas zu Funktorobjekten mit einem nichtzugreifbaren/versteckten Klassenname.

```
[&x]{
    ++x; return x*x;
}
```

```
struct __unknown_generator_name_ {
    __unknown_generator_name_(int &x):x{x}{}
    auto operator()() const { ++x; return x*x; }
private:
    int &x;
};
```

```
[(char c)->char{
    return std::toupper(c);
}]

struct __unknown_converter_name_ {
    auto operator()(char c) const ->char{
        return std::toupper(c);
};
};
```

### Lambdas in Member Functions

```
struct DemoLambdaMemberVariables {
    int x{};
    std::vector<int> demoAccessingMemberFromLambda() {
        std::vector<int> v;
        generate_n(back_inserter(v),10,[=] {
            return ++x, x*x; // member x can be changed
        });
        return v;
    }
};
```

capture this by copy

Captured-Membervariablen werden immer als Referenz erfasst, auch wenn der Standardwert [=] ist. Der Grund ist, dass this ein Pointer (Referenz) ist und wenn sie kopiert werden, sind die Membervariablen auf Sie verwiesen. This kann nicht als Referenz erfasst werden.

### Standard Functor template classes

```
transform(v.begin(),v.end(),v.begin(),[int x]{return -x;});
transform(v.begin(),v.end(),v.begin(),std::negate<int>{});
```

Lambdas machen das Verwenden von transform, etc. sehr einfach. Auch wenn nur ein einfacher Ausdruck gebraucht wird, besteht die Möglichkeit einer Wiederverwendung von Standard-Library Functor Klassen oder auch von Relationalen Operatoren.

```
sort(v.begin(),v.end(),std::greater<>{});
```

parameter type deduced

## #include <functional>

- binary arithmetic and logical

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

```
transform(v.begin(), v.end(), v.begin(),
         v.begin(), std::multiplies<>{});
```

- unary

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

- binary comparison

- `less<> (<)`
- `less_equal<> (≤)`
- `equal_to<> (==)`
- `greater_equal<> (≥)`
- `greater<> (>)`
- `not_equal_to<> (!=)`

## Parametrize Associative Containers

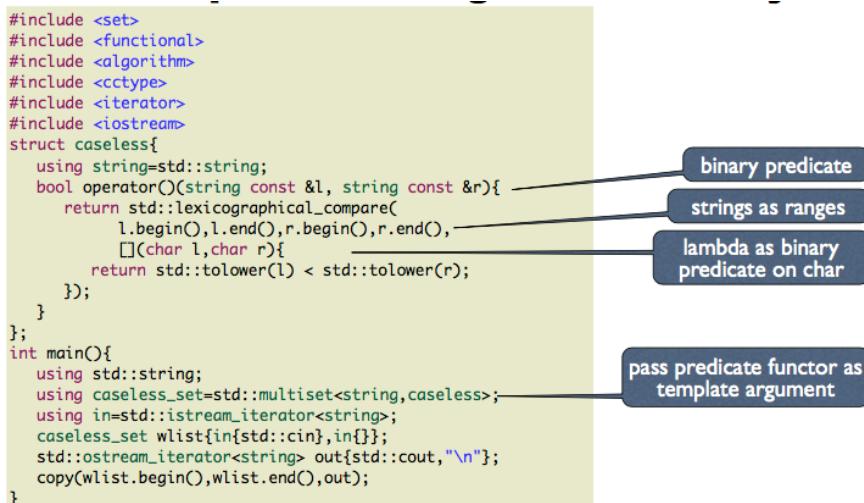
```
std::set<int, std::greater> reverse_int_set{};
```

Die associativen Container erlauben es ein zusätzliches Template Argument für die Vergleichsoperation zu verwenden. Es muss eine Functorklasse mit einem gegebenen Binären Predicate sein. Das Predicate muss irreflexive und trasistive sein. Eigene Funktoren können spezielle Sortorders zur Verfügung stellen. Achtung `>=` darf dafür nicht verwendet werden, da es reflexive ist.

### Example `set<string> dictionary`

```
#include <set>
#include <functional>
#include <algorithm>
#include <cctype>
#include <iostream>
struct caseless{
    using string=std::string;
    bool operator()(string const &l, string const &r){
        return std::lexicographical_compare(
            l.begin(), l.end(), r.begin(), r.end(),
            [] (char l, char r){
                return std::tolower(l) < std::tolower(r);
            });
    }
};

int main(){
    using std::string;
    using caseless_set=std::multiset<string,caseless>;
    using in=std::istream_iterator<string>;
    caseless_set wlist{in{std::cin},in{}};
    std::ostream_iterator<string> out{std::cout, "\n"};
    copy(wlist.begin(),wlist.end(),out);
}
```



## How to keep functors around?

In Situationen, in denen Sie nicht nur einen Funktor, sondern Sie wollen eine Funktion oder functor als Parameter, z. B. in einem Konstruktor und später anwenden, was sollte der Typ der Variable (Member) und der Parameter sein?

### Foo(double f(double))

Funktioniert nicht mit Funktoren. Auch nicht spezifiziert es Funktorentypen für andere.

### Es gibt dafür 2 Optionen:

- 1. Template Typename Parameter (später)
- 2. Std::function<double(double)>

Std::function<SIGNATURE>

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

Es ist ein genereller Funktionshalter. Er kann Funktionen, Funktoren und Lambdas abspeichern. Die Signatur muss mit dem Signature Funktions-Template-Argument kompatibel sein. Kann erneut zugewiesen werden oder geleert werden. Er kann auch leer sein, dies sollte aber geprüft werden.

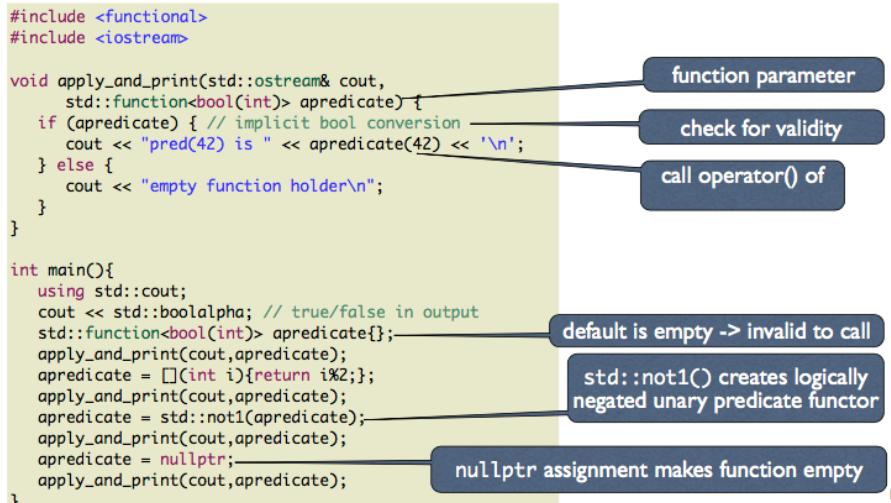
```
if (apredicate) { // implicit bool conversion: is valid?
    cout << "pred(42) is " << apredicate(42) << '\n';
} else {
    cout << "empty function\n";
}
```

### Example using std::function

```
#include <functional>
#include <iostream>

void apply_and_print(std::ostream& cout,
                     std::function<bool(int)> apredicate) {
    if (apredicate) { // implicit bool conversion
        cout << "pred(42) is " << apredicate(42) << '\n';
    } else {
        cout << "empty function holder\n";
    }
}

int main(){
    using std::cout;
    cout << std::boolalpha; // true/false in output
    std::function<bool(int)> apredicate{};
    apply_and_print(cout,apredicate);
    apredicate = [](int i){return i%2;};
    apply_and_print(cout,apredicate);
    apredicate = std::not1(apredicate);
    apply_and_print(cout,apredicate);
    apredicate = nullptr;
    apply_and_print(cout,apredicate);
}
```



The diagram shows several annotations pointing to specific parts of the code:

- A callout box labeled "function parameter" points to the parameter declaration `std::function<bool(int)> apredicate`.
- A callout box labeled "check for validity" points to the condition `if (apredicate)`.
- A callout box labeled "call operator() of" points to the call `apredicate(42)`.
- A callout box labeled "default is empty -> invalid to call" points to the assignment `apredicate{}`.
- A callout box labeled "std::not1() creates logically negated unary predicate functor" points to the assignment `apredicate = std::not1(apredicate);`.
- A callout box labeled "nullptr assignment makes function empty" points to the assignment `apredicate = nullptr;`.

```
#include <functional>
#include <iostream>

struct X {
    int calc(int i) const { return x*i; }
private:
    int x{7};
};

int main(){
    std::function<int (X const &,int)> const f{&X::calc};
    X const anX{};
    std::cout << f(anX,6);
}

```

member function to wrap

make variable const to avoid invalid function objects (general rule!)

must pass object as this argument explicitly

member function pointer must specify & address-of operator to member

## What is a member-(function) pointer?

Es ist möglich auf individuelle Class Member zu referenzieren, indem man „Point-to-Members“ benutzt. Zum Beispiel um mit verschiedenen Members vom selben Typen bei der selben Funktion umgehen zu können. Syntax des Typs ist: type Class::\* varname. Initialisiert wird dieser Wert durch &Class::member.

### Example Pointer to Members.\*

```
struct X {
    void foo() const { std::cout << a << " foo\n"; }
    void bar() const { std::cout << b << " bar\n"; }
    int a;
    int b;
};

void doit(void (X::*mfunc)() const, X const &x){
    (x.*mfunc)();
}

void change(int X::*memvar, X& x, int val){
    x.*memvar = val;
}

int main(){
    X x{1,2};
    doit(&X::foo,x);
    doit(&X::bar,x);
    change(&X::a,x,42);
    change(&X::b,x,43);
    doit(&X::foo,x);
    doit(&X::bar,x);
}
```

must match m-f signature including const and parameters

use operator .\* for member function pointer () required - precedence

use operator .\* for pointer to member access

1 foo  
2 bar  
42 foo  
43 bar

22

## Std::function calling member function

```
auto g=std::function<void(X const&, int)>{&X::foo};
X x{42,43};
g(x,1); // calls -> x.foo(1)
x.bar(2);

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

std::function für eine Memberfunktion macht es möglich das this Objekt implizit weiterzugeben. Dies kann am Beispiel gesehen werden. Der linke Teil des .Operator ist aktuell an die Member Funktion als Argument weitergegeben.

## #include <iterator>

Algorithmen werden durch Funktoren, aber auch durch Iteratoren parametrisiert. Sie können eigenen Iteratoren erstellen, indem Sie boost :: iterators library verwenden.

```
#include <boost/iterator/counting_iterator.hpp>
#include <boost/iterator/filter_iterator.hpp>
#include <boost/iterator/transform_iterator.hpp>
```

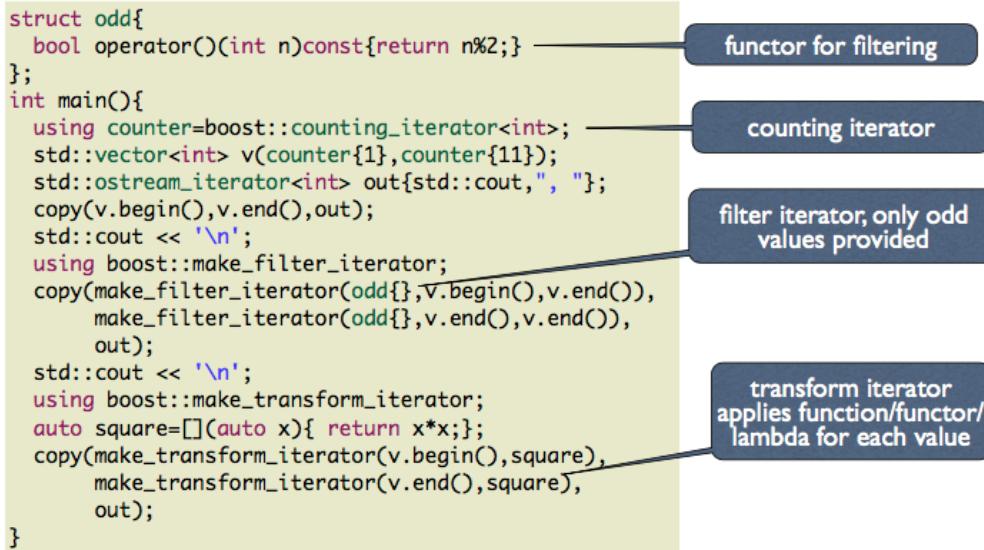
Mehrere vordefinierte Adapter mit werkseiten Funktionen sind vorhanden. Zum Beispiel:

- counting
- filtering
- transforming

### Example using Boost Iterator

```
struct odd{
    bool operator()(int n) const {return n%2==1;}
};

int main(){
    using counter=boost::counting_iterator<int>;
    std::vector<int> v(counter{1},counter{11});
    std::ostream_iterator<int> out{std::cout," "};
    copy(v.begin(),v.end(),out);
    std::cout << '\n';
    using boost::make_filter_iterator;
    copy(make_filter_iterator(odd{},v.begin(),v.end()),
        make_filter_iterator(odd{},v.end(),v.end()),out);
    std::cout << '\n';
    using boost::make_transform_iterator;
    auto square=[](auto x){ return x*x;};
    copy(make_transform_iterator(v.begin(),square),
        make_transform_iterator(v.end(),square),out);
}
```



# Function

Defining your own flexible function templates

What you already know

```
[x=1](auto y){return x*y;}
```

Lambdas erlauben auto als Parametertyp (Aufruf). Der Typ des Lambda Capture muss nicht angegeben werden. Der Returnwert ebenfalls nicht. Alle konkreten Typen werden aus dem Kontext, der Definition und der Aufrufseite des Lambdas abgeleitet. Ähnliche Flexibilität gibt es auch für (Inline)-Funktionen, aber mit einer anderen Syntax

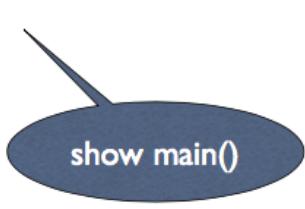
## Motivation for templates

### Template Myths

- Die Spaltenklammern sind schwer zu lesen „>>“
- Compiler-Nachrichten sind verwirrend (ja gcc war so)
  - o Fehlermeldungen beziehen sich auf Definition, aber die Ursache ist die Nutzung
  - o Nesting erzeugt lange Instantiierungsketten
- Compiler behandeln Vorlagen(Template) falsch
- Nutzungsbeschränkungen bestehen aus historischen Gründen
  - o Viel weniger mit modernem C++, aber Sie sind nicht ganz weg.

## Why function templates

```
namespace MyMin{
inline int min(int a, int b){
    return (a < b)? a : b ;
}
inline double min(double a, double b){
    return (a < b)? a : b ;
}
} // MyMin
```



show main()

Überladen einer Funktion für verschiedene Typen. Dann sind die Funktionsbodies genau gleich. Ein Anpassen des Codes bei allen ist im Nachhinein sehr mühsam.

```
#include <string>
namespace MyMin {
inline std::string const &min(std::string const & a, std::string const & b){
    return (a < b)? a : b ;
}
} // MyMin
```

## Specify a function template

Why can a reference be used as return type?

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

Eine Definition, viele Schabloneninstanzen. Dafür muss das Keyword template mit den <> Spitzenklammern angegeben werden. Typennamen für den Typparameter.

Ersatz für T wird automatisch bei Funktionsaufruf aus seinen Argumenten (Template-Argument Abzug) wie ein Lambda-Auto-Parameter ermittelt

### Using our MyMin::min() template

Template-Funktionen müssen in Header-Dateien definiert werden. Es ist eine vollständige Definition für die Verwendung erforderlich

```
#include "MyMin.h"
#include <iostream>
#include <string>

int main(){
    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";
    cout << "min(Hallo,Hallihallo)= "
    << MyMin::min(s1,s2)<<'\n';
//min(2,pi); // compile error
min(static_cast<double>(2),pi);
cout << "min(Pete,Toni) = "
    << min("Pete","Toni") << '\n';
}

min("Peter","Toni") doesn't compile!
```

min<int>()

min<double>()

Why is MyMin:: needed?

Delivers "Toni" as result, why?

### Template Argument's Concepts

template <typename T> what is the concept of type T in min()?

```
T const& min(T const& a, T const& b){
    return (a < b)? a : b ;
}
```

Anforderungen an den Parameter typename einer Vorlage werden durch ihre Verwendung implizit gegeben. Im Gegensatz zu Java, wo man Schnittstellen von generischen Argumenten spezifiziert. Solche impliziten Anforderungen werden auch als das "Konzept" des Template-Parameters bezeichnet. Die Erfüllung nur geprüft, wenn das Template verwendet wird, d.h. min () mit einem Typ aufgerufen wird.

```
template <typename T>
T const& min(T const& a, T const& b){
  return (a < b)? a : b ;
}
```

Der Operator <(T const&, T const&) muss definiert werden oder der Operator <(T,T). Zudem muss es einen Rückgabewert mitgeben, welcher sich auf einen Boolean konvertieren lässt. (wegen des ?:). Der void Operator<(T,T) wäre unmöglich.

What is the concept for max()'s T?

```
template <typename T>
T max(T a, T b){
  return a < b ? b : a;
}
```

Der Operator <(T const&, T const&) muss definiert werden oder der Operator <(T,T). Zudem muss es einen Rückgabewert mitgeben, welcher sich auf einen Boolean konvertieren lässt. (wegen des ?:).

### Function Template Argument Deduction

Der Compiler leitet automatisch den zu verwendenden Typ ab. Wie auto bei Variablen. Wie nach Design, stehen alle möglichen Überladungen zur Verfügung. Wenn es mehrdeutig ist, müssen Sie den gewünschten Typ angeben.

```
//min(2,pi); // compile error
min(static_cast<double>(2),pi);
min<double>(2,pi);
```

Surprising type deduction

```
//cout << "min(Peter,Toni) = " << min("Peter","Toni") << '\n';
cout << "min(Pete,Toni) = " << min("Pete","Toni") << '\n';
cout << "max(Peter,Toni) = " << max("Peter","Toni") << '\n';
```

String Literale sind kein String es wird durch Pass by Value dennoch der Pointer übergeben. Ein Vergleich von Pointers ist aber kein Vergleich von Strings. Wir müssen das Template Argument nutzen, um zu angeben, dass wir einen String übergeben möchten.

```
cout << "max<string>(Peter,Toni) = "
  << max<std::string>("Peter","Toni") << '\n';
```

## Function template overloads

```
template <typename T>
T const& min(T const& a, T const& b){
    return (a < b)? a : b ;
}

template <typename T>
T const * min(T const * a, T const* b){
    return (*a < *b)? a : b ;
}
```

Problem, wenn mit einem Zeiger, vergleicht min die Adressen und nicht die angegebenen Werte. Wir können einen Overload zur Verfügung stellen, wenn sie mit Pointer aufgerufen wird, um die Pointer zu dereferenzieren.

## Function overloads and templates

```
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); declaration of overload in header

#include "MyMin.h"
#include <string>
namespace MyMin {
    char const* min(char const* a, char const* b) { implementation in .cpp to avoid header dependency on <string>
        return std::string(a)<std::string(b)?a:b;
    }
}
```

**Problem:** String Literale funktionieren immer noch nicht.

Wir stellen daher auch einen Regular Function Overload zur Verfügung. Die spezialisierte Überladung gewinnt!

### Caution: Overloading function templates

- Verwenden Sie es nicht übereigrig
- Mehrdeutigkeiten ergeben sich schnell, besonders bei mehreren Parametern.
- Sichtbarkeit aller Überladungen ist wichtig (Bug in C++14)
  - o Sonst könnte der identische Suchcode in verschiedenen Kompilierungseinheiten unterschiedlich kompiliert werden
- Setzen Sie alle Overload Deklarationen in gemeinsamen Header
  - o Oder mit dem Header des verwendeten Typs

## Problem – Varying number of arguments

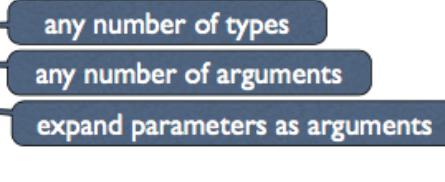
```
println(out,1,2,3,"hello",' ','world');
```

Einige Sprachen bieten eine Möglichkeit, eine variable Anzahl von Argumenten an eine Funktion zu übergeben. Die C-Lösung verwendet void func (...), ist aber inhärent nicht typsicher, z. B. printf (). In Java you specify one type for all arguments (or Object) an. In C ++ 11 verwenden Sie eine typsichere "Variadic-Vorlagen"

## Definitg a variadic template function

```
template <typename... ARGS>
void variadic(ARGS... args){
    println(std::cout, args...);
}
```

also a variadic template function



Die Syntax benutzt den ...operator an verschiedenen Stellen

- **Vor einem Namen** -> definiert den Namen als Platzhalter für einen Variable Anzahl von Elementen
- **Nach einem Namen** -> erweitern den Namen zu einer Liste aller Elemente, die es darstellt
- **Zwischen zwei Namen**, als zweiter Name welche eine Liste von Parametern definiert, gegeben vom ersten.

## Implementing a variadic function

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

Der **Schlüssel** ist die Rekursion in den Definitonen.

- Basisfall mit keinem Argument
- Rekursiven Fall mit 1 explizitem Argument und einem Ende (Tail), der aus einer varadischen Liste von Argumenten besteht

## Template Classes

In Ergänzung zu den Funktionen können auch Klassen Template Parameters haben. Im Gegensatz zu den Funktionsvorlagen müssen Sie bei der Verwendung vom Klassenvorlagen die Template Argumente immer angeben. Es gibt eine auto-deduction, aber eine Funktionsschablone kann helfen.

### Template Class – Class Template

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

**Typen** typename

**Konstanten** int typ oder auto

**Weitere Templates** template

Die Terminology ist gemischt. Es sind aber beide Versionen OK. Der Typen wird mit Compile-Time Parameters angegeben. Datenmembers können auf Template Parameters basieren. Die Funktions-Members sind Template-Funktionen mit den Template-Parametern der Klasse.

### Template Class Sack<T>

```
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();
};

template <typename T>
inline T Sack<T>::getOut()
{
    if (!size()) 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;
}
```

template class with one typename parameter

dependent name: size\_type

needs typename, because of dependent name

member function forward declaration

shows how member functions are implemented outside the template class

pick random element

### Type aliases

```
using SackType=std::vector<T>;
```

Es ist üblich, dass Template-Definitionen Typ-Aliase definieren, um ihre Verwendung zu erleichtern (weniger zum Schreiben und Lesen). Es geschieht über using newtype=existingtype. Die alte Konstruktionsweise dafür ist typedef. Diese sollte aber nicht mehr verwendet werden.

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

typename tells this is a type

Innerhalb der Vorlagendefinition können Sie Namen verwenden, die direkt oder indirekt vom Vorlagenparameter abhängig sind. Zum Beispiel alles mit SackType::. Sie müssen aber dem Compiler sage, ob es sich um einen Typ handelt.

### What Concept for Sack's T?

```
using SackType=std::vector<T>;
void putInto(T const &item) { theSack.push_back(item);}
T getOut() ....
template <typename T>
inline T Sack<T>::getOut()...
    T retval{theSack.at(index)}....
    return retval;
}
```

T hat einige Anforderungen/Konzepte. Das T muss in vector<T> passen. Zudem darf T nicht void sein. Zuletzt muss T kopier(oder movebar) sein.

### Members outside of class template

```
inline to avoid ODR violation
template <typename T> repeat template intro
inline T Sack<T>:: template ID of class Sack
member signature
getOut(){
    if (!size()) 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;
}
```

Members können aus der Templateklasse definiert werden. Der Syntax dafür ist aber sehr unschön. Sie müssen aber immer noch inline sein. Die Zeilenumbrüche oben sind nur zur Hervorhebung von den verschiedenen Aspekten hier.

### Template Class Rules

- Definieren Sie Template Klassen komplett in Header Dateien
  - o Entweder Member Funktionen direkt in der Klasse
  - o Oder als Inline Funktionen im Header File
- Bei der Verwendung von Sprachelementen, die sich direkt oder indirekt auf einen Schablonenparameter beziehen, müssen Sie bei der Namensgebung einen Typnamen angeben.
- Es können statische Membervariablen einer Template-Klasse im Header definiert werden, ohne ODR zu verletzen, auch wenn sie in mehreren Kompilierungseinheiten enthalten sind.

## Demo static member variables in template class

```
#ifndef TEMPLATEWITHSTATICMEMBER_H_
#define TEMPLATEWITHSTATICMEMBER_H_
template <typename T>
struct staticmember{
    static int dummy;           declaration in class
};

template <typename T>
int staticmember<T>::dummy{sizeof(T)}; definition outside
#endif /* TEMPLATEWITHSTATICMEMBER_H_ */
don't forget type of variable
```

```
#include "templatewithstaticmember.h"

int foo(){
    using dummytype=staticmember<int>;
    dummytype::dummy=42;
    return dummytype::dummy;
}
```

separate compilation unit = .cpp file

- both compilation units share staticmember<int>::dummy

```
#include "templatewithstaticmember.h"
#include <iostream>
int main(){
    int foo(); // declaration of foo() function
    std::cout << staticmember<double>::dummy << '\n';
    std::cout << staticmember<int>::dummy << '\n';
    std::cout << foo() << '\n';
    std::cout << staticmember<int>::dummy << '\n';
}
```

8  
4  
42  
42

C++17 will provide  
inline variables with  
definition in class

## Class template gotchas

Bei der Verwendung von Vererbung in einer Vorlageklasse, die von einer Vorlagenklasse übernommen wird, kann der Name-Lookup überraschen! Es sollte daher this=> verwendet werden.

```
template <typename T>
struct parent {
    int foo() const { return 42; }
    static int const bar{43};
};

int foo() { return 1; }
double const bar{ 3.14};
```

```
template <typename T>
struct demogotchas : parent<T> {
    std::string demo()const {
        std::ostringstream result{};
        result << bar << " bar \n";
        result << this->bar << " this->bar \n";
        result << demogotchas::bar << " demogotchas::bar\n";
        result << foo() << " foo() \n";
        result << this->foo() << " this->foo() \n";
        return result.str();
    }
};
```

3.14 bar  
43 this->bar  
43 demogotchas::bar  
1 foo()  
42 this->foo()

## Class Templates that inherit

Nutzen Sie immer `this=>` oder die Klasse `name::` um auf vererbte Member in einer Template Klasse zu referenzieren. Wenn der Name ein abhängiger Name sein könnte, wird der Compiler es nicht beim Kompilieren der Template-Definition suchen. Checks können nur für abhängige Namen bei der Verwendung von Templates vorgenommen werden. Das ist der Grund für manchmal langwierige Fehlermeldungen vom Schablonengebrauch.

What about a Sack with pointers?

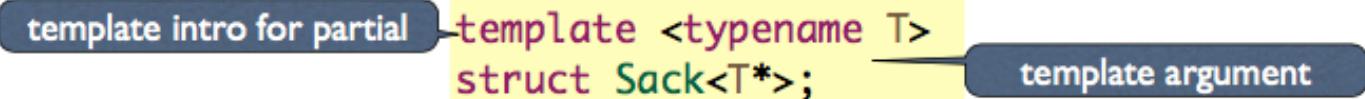
```
Sack<int *> shouldNotCompile;
```

Das Halten von den Zeigern in einem Sack würde verlange, dass alle Variablen oder Objekte den Stack überleben. In den meisten Fällen ist dies schwer zu erreichen. Und jemand muss die Objekte trotzdem irgendwann aufräumen.

Es könnte daher besser sein, die Zeiger zu verbieten. Mit Ausnahme von Zeichenzeigern, die String-Literale darstellen, können wir `std :: string` verwenden, um sie im Sack zu speichern.

```
Sack<char const *> shouldkeepStrings;
```

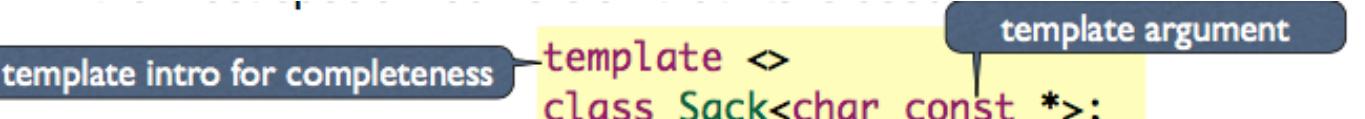
Template (partial) Specialization)



template intro for partial      template <typename T>  
                                      struct Sack<T\*>;

Wie beim Überladen von Template-Funktionen können wir „Template-Spezialisierungen“ für die Klassen-Templates anbieten. Diese können teilweise immer noch einen Template-Parameter haben, aber bieten (einige) Argumente. Oder vollständige Spezialisierungen, die alle Argumente mit konkreten Typen haben. Zunächst muss eine nicht-spezialisierte Vorlage deklariert werden.

Die am meisten spezialisierte Version, die passt, wird verwendet.



template intro for completeness      template <>  
    class Sack<char const \*>;

How to avoid creating a `Sack<int*>`

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

Eine Klassen Template Spezialisierung kann jeden Inhalt haben. Auch einen leeren Inhalt. Es kann völlig unabhängig von der ursprünglichen Vorlage sein. Wirkliche keine Beziehung vorhanden.

Eine Möglichkeit, das Instanziieren einer Klasse zu verbieten, besteht darin, die Fähigkeit ihrer Zerstörung zu verbieten, indem sie ihren Destruktor als = delete deklariert. Wenn man es nicht zerstören kann, kann man auch nie ein Objekt schaffen.

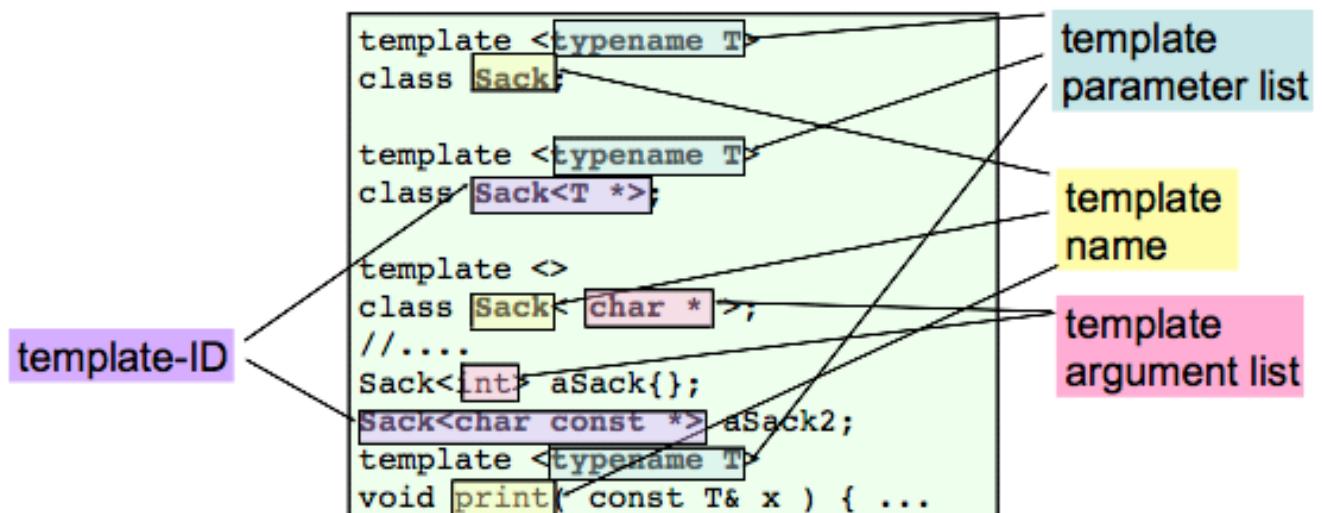
```
template <typename T> class Sack; -- forward declaration of template

template <>
class Sack<char const *> { -- class template specialization
    typedef std::vector<std::string> SackType;
    typedef SackType::size_type size_type;
    SackType theSack;
public:
    // no explicit ctor/dtor required
    bool empty() { return theSack.empty(); }
    size_type size() { return theSack.size(); }
    void putInto(char const *item) { theSack.push_back(item); }
    std::string getOut() {
        if (!size()) throw std::logic_error{"empty Sack"};
        std::string result=theSack.back();
        theSack.pop_back();
        return result;
    }
};
```

no typename, because no dependent name

implementation of specialization can behave completely different (no randomness)

# Template Terminology



- 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) { .... }
```

## Extending our Stack<T>

- Erstellen eines Sack<T> mit einem Iterator befüllen
- Besorgen einer Kopie der Elemente in einem std::vector
  - o Für Iteration und Inspektion
- Ein Sack<T> mit einer Initializerliste erstellen.
- o Nutzen von std::initializer\_list<T>
- Auto-deducing T für einen Sack <T> aus einer Initialisierungsliste
- Variieren des Typs des zu verwendenden Behälters

## Construct a Sack from Iterators

Std::vector<T> kann von einem Paar Iteratoren erstellen werden. Der Sack könnte dies auch.

```
void createSackFromIterators() {
    std::vector<int> v{3,1,4,1,5,9,2,6};
    Sack<int> asack{v.begin(),v.end()};
    ASSERT_EQUAL(v.size(),asack.size());
}
```

```
void defaultConstructorStillWorks() {
    Sack<char> defaultctor{};
    ASSERT_EQUAL(0, defaultctor.size());
}
```

```
template <typename T>
class Sack {
    using SackType=std::vector<T>;
    using size_type=typename SackType::size_type;
    SackType theSack{};
public:
    Sack():default{}; —————— re-establish default ctor
    when other ctor is defined
    template <typename ITER> —————— can use additional template
    Sack(ITER b, ITER e):theSack(b,e){} —————— parenthesis to avoid triggering
                                              initializer_list overload
```

21

Surprise: Creation allows two ints

```
void creationAlsoAllowsTwoInts() {
    Sack<unsigned> asack { 10, 3 };
    ASSERT_EQUAL(10, asack.size());
    ASSERT_EQUAL(3, asack.getOut());
}
```

Durch die Definition des Templatized Konstruktors von Sack können wir auch andere Vektor-Konstruktoren mit 2 (identischen) Parametern "wiederverwenden". Dies wird jedoch nicht mehr funktionieren, wenn wir unseren eigenen Initialisierungslisten-Konstruktor definieren.

```
void creationAlsoAllowsTwoInts2() {
    Sack<unsigned> asack ( 10, 3 );
    ASSERT_EQUAL(10, asack.size());
    ASSERT_EQUAL(3, asack.getOut());
}
```

will continue to work

22

```
auto v=static_cast<std::vector<unsigned>>(asack);
```

other compatible type can be used as template argument

```
template <typename Elt>
explicit operator std::vector<Elt>() const {
    return std::vector<Elt>(theSack.begin(),theSack.end());
}
```

Mit einem expliziten Umwandlungsoperator können wir einen std :: vector aus dem Sack durch Kopieren extrahieren. Als Alternative kann man auch ein Member Function Template verwenden.

```
template <typename Elt=T>
std::vector<Elt> asVector() const {
    return std::vector<Elt>(theSack.begin(),theSack.end());
}
Sack<int> asack = sackForTest();
auto v=asack.asVector();
auto vd=asack.asVector<double>();
```

default template argument can be omitted

other compatible type can be used as template argument

### Filling a Stack with std::initializer\_list<T>

```
void createSackFromInitializerList(){
    Sack<char> csack{'a','b','c'};
    ASSERT_EQUAL(3,csack.size());
}
```

(Homogene) Initialisierungslisten {1,2,3} werden als Typ std :: initializer\_list <T> übergeben. Das Definieren eines Konstruktors, der initializer\_list <T> nimmt, erlaubt es uns, einen Sack <T> vorzufüllen.

**Sack(std::initializer\_list<T> il):theSack(il){}**

### Factory for Sack from an Initializer

Es ist ein bisschen ärgerlich, <int> anzugeben, wenn Sie eine Liste mit integralen Konstanten zur Erstellung eines Sacks angeben. Ein verbreiterter Trick kombiniert die Funktion template argument deduction, um das Template-Argument einer Template-Klasse zu wählen.

```
#include <initializer_list>
template <typename T>
Sack<T> makeSack(std::initializer_list<T> list){
    return Sack<T>{list};
```

template argument deduction works also when parameter is nested

## More on using std::initializer\_list<T>

```
template <typename T>
Sack<T> makeSack1(std::initializer_list<T> list){
    Sack<T> sack;
    for (auto it=begin(list); it != end(list); ++ it)
        sack.putInto(*it);
    return sack;
}

template <typename T>
Sack<T> makeSack3(std::initializer_list<T> list){
    return Sack<T>(begin(list),end(list));
}
```

Anstatt einfach ein initializer\_list-Argument an std :: vector zu übergeben, hätten wir makeSack auch ohne einen initializer\_list <T> Konstruktor von Sack implementieren können. Initializer\_list bietet begin () und end () Iteratoren.

```
template <typename T>
Sack<T> makeSack2(std::initializer_list<T> list){
    Sack<T> sack;
    for (auto const &elt:list) sack.putInto(elt);
    return sack;
}
```

## Varying Sack's Container

Ein std :: vector ist möglicherweise nicht die optimale Datenstruktur für einen Sack, ganz abhängig von seiner Verwendung. Das Entfernen aus der "Mitte" erfordert zum Beispiel Verschiebungswerte im Vektor. Was wäre, wenn wir sowohl den Containertyp als auch einen Template-Parameter angeben könnten. Dies benötigt einige allgemeine APIs.

## Template Template Parameters

```
template <typename T, template<typename> class container>
class Sack1;
```

must be class!

Ein Template kann Template als Parameter aufnehmen (Template Template Parameter). Der Template-Template Parameter muss die Anzahl der typename Parameter angeben. **Problem:** std :: Container nehmen in der Regel mehr als nur den Elementtyp als Template-Parameter.

```
Sack1<int,std::vector> doesntwork;
```

std::vector<> is defined with 2 template parameters

**Lösung:** eine beliebige Anzahl von typename Parametern verwenden: typename ... (variadic Schablone). Dadurch können wir auch eine Standard-Container-Vorlage als std :: vector angeben. Aber auch andere Container für die Sack Mitglieder.

```
template <typename T,
         template<typename...> class container=std::vector>
class Sack
{
```

```
Sack<int,std::list> listsack{1,2,3,4,5};
```

## Template-Template Argument factory

```
auto setsack=makeOtherSack<std::set>({'a','b','c','c'});
ASSERT_EQUAL(3,setsack.size());
```

Das Template-Template-Argument können wir in unserer Factory-Funktion nicht bestimmen, sondern nur den Elementtyp. Die Template Parameter Sequenz muss umgeschaltet werden.

```
template <template<typename...> class container,typename T>
Sack<T,container> makeOtherSack(std::initializer_list<T> list)
{
    return Sack<T,container>{list};
}
```

## Adapting standard containers

Wie können wir einen Standardcontainer durch Hinzufügen von Invarianten oder durch Erweiterung seine Funktionalität anpassen?

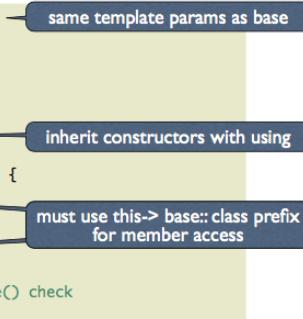
- SafeVector → Kein unentdeckter Out-of-bounds-Zugriff
- IndexableSet → []-Operator zur Verfügung stellen
- SortedVector → Garantieren sortiert Reihenfolge der Elemente

Durch eine Template-Klasse, die von der Template-Basisklasse übernommen wird. Erbt die Konstruktoren des Standard Containers.

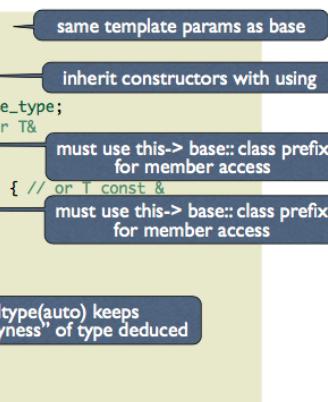
**Vorsicht:** keine sichere Umwandlung in Basisklasse, keine Polymorphie

### Example SafeVector<T>

```
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
T const & operator[](size_type index) const {
return this->at(index);
}
T & operator[](size_type index) {
return Base::at(index);
}
// should also provide front/back with size() check
};
```



```
template<typename T>
struct safeVector
:std::vector<T> {
using std::vector<T>::vector;
using size_type=typename std::vector<T>::size_type;
decltype(auto) operator[](size_type i){ // or T&
return this->at(i);
}
decltype(auto) operator[](size_type i) const { // or T const &
return this->at(i);
}
};
```



## Main (argc, argv)

Übergabe von Argumenten an C++ - Programme und ältere Datenstrukturen.

### Übergeben von Programmargumenten

```
int main(int argc, char *argv[])
```

array of pointers to char representing strings

Wie in vielen anderen Sprachen, kann man auch in C++ Kommandozeilenargumente mitgeben. Der Mechanismus wurde von C geerbt. Es werden Points und ein Array Argument genutzt. Dazu sind separate Elementzähler (Counters) nötig. Char\* repräsentiert in C ein String. Eine Char Sequenz wird mit einem «\0» abgeschlossen.

### «Nackte» Arrays als Parameter

```
int main(int argc, char *argv[])
```

Das Definieren eines Funktionsparameters als Array bedeutet, dass nur ein nackter Zeiger auf das erste Element übergeben wird. Dabei gehen die Größen verloren und müssen daher explizit übergeben werden. Am besten nutzt man einfach std::array oder vector.

Die Grösse kann dabei als Parameter (argc) übergeben werden oder mittels einer bekannten Endsequenz ermittelt werden.

Pointer zu Arrays sind Random-Access Iteratoren

```
int main(int argc, char *argv[]){
    copy(argv+1, argv+argc,
        std::ostream_iterator<std::string>(std::cout, "-"));
}
```

Die Arrayparameter sind wie bereits gesagt "Nackte" Pointer. P bzw. argv ist also das erste Element. Mit p+dimension(size) erreicht man dann das letzte Element. Sie funktionieren dabei wie Random Access Iteratoren. Also mit ++p, \*p, p+int, p[int] oder \*(p+int).

### Initialisieren eines Arrays

int five[5]{};	5 zeros
int four[4]={1,2,3,4};	
double d[4]{1.0, 2.0,};	d[2]==d[3]==0.0
double m[2][3]{{1,2,3},{4,5,6}};	

Alle Arrays brauchen eine Grösse. Also die Anzahl Elemente. Arrays ohne eine Initialisierung können nicht initialisiert werden. Wird der Initialisierungswert vergessen, wird ein Default Wert genommen.

Mehrdimensionale Arrays sind auch möglich. Ein Array int x[5][6] hat 5 Elemente in welchen je ein Array mit 6 int Werten drin ist. X[1,2] ist auch möglich, es ist aber falsch. Der Komma-Operator ist da die Schlüssel stellen.

char s[6>{"hello"};	5 chars + '\0'
---------------------	----------------

what does that mean?

```
double m[2][3]{{1,2,3},{4,5,6}};
m[0,1];
```

Grundsätzlich findet der Zugriff auf ein Element mit `[]()` statt. Dabei ist es wichtig, dass alle Brackets zugreifbar sind. Ein Komma zu verwenden ist vom Syntax her ebenfalls korrekt. Dies ist aber nur nützlich, wenn der erste Teil einen Seiteneffekt hat

### Erschliessen der Array Grössen

```
template <typename T, unsigned N>
void printArray(std::ostream &out, T const (&x)[N]){
    copy(x, x+N, std::ostream_iterator<T>{out, ", "});
}
```

Das Übergeben eines Array-Typs durch Verweis in einer Templatefunktion ermöglicht Dimensionsabzug durch Template-argumentabzug. Also `T(&x)[N]` wobei `N` dann die Dimension ist.

Die Größe eines Arrays wird bei der Erstellung automatisch aus der Anzahl der Elemente abgeleitet.

```
int a[]={1,2,3};           int a[3]
printArray(std::cout,a);
char str[]="world";        char str[6]
```

### Zusammenfassung «Nackte» Arrays

- Benutzen Sie keine «nackten» Arrays → Refakturieren Sie diese zu `std::arrays`.
  - o Die einzige Ausnahme sind die Programmargumente im `main()`.
- Degenerieren des Arrays zu einem «nackten» Zeiger, wenn Sie als Funktionsargumente übergeben werden
  - o Man muss auch immer die Länge mitgeben.
- Zeiger zu Arrays funktionieren wie Random Access-Iteratoren
  - o `Ptr+dimension = end iterator`

# Dynamic Heap Memory Management

## Heap Memory

Das Ganze basiert immer auf Bibliotheksklassen, welche für die Verwaltung da sind. Heap Memory kann für die Erstellung von Objektstrukturen erforderlich sein. Unteranderem für die Polymorphismus Factory Funktionen. Wenn am beispielsweise einen Basisklassenpointer an einen der Subklasse weitergibt.

## C++ Heap Memory Basics

```
auto ptr=new int{5};  
std::cout << *ptr << '\n';  
delete ptr;
```

**NEVER DO THIS**

C++ lässt es zu Objekte auf dem Heap zu allozieren. Wenn es jedoch manuell getan wird, ist man selbst für die Deallocation sowie das Risiko des undefined Behavior verantwortlich. Zu den Probleme gehören dann Memory Leaks, Dangling Pointers oder doppelte Deletes.

In C++ findet keine Garbage Collection statt, daher muss man es selber deleten oder es gibt ein Leak.

## C++ Richtige Heap Memory Nutzung

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

Nutzen Sie nicht «nackte» Pointer mit Heap Allocation im modernen C++. (Vorheriges Kapitel). Es sollen std::unique\_ptr<T> und std::shared\_ptr<T> genutzt werden.

Also kommen Sie nie zu einer Situation, bei welcher Sie explizit delete ptr aufrufen müssen. Es sei denn, Sie ein Experte in der Implementation von Libarys.

### Std::unique<ptr>

Wird für unshared Heap Memory verwendet. Also zum Beispiel auf für lokale Sachen, welche auf dem Heap sein müssen. Dieser Fall ist aber sehr sehr selten nötig. Nur mit sehr grossen Instanzen.

Der Pointer kann bei einer Factory Funktion zurückgegeben werden und hat nur einen Besitzer. Zudem kann der Unique-Pointer befreite Zeiger von C-Funktionen umwickeln.

Für Klassenhierarchien sind es aber nicht die besten. Dort sollten die shared\_ptr verwendet werden. Die Unique\_Ptr brachen einen virtuellen «virtual» Destruktor, was wir eigentlich nicht wollen.

## Resourcenmanagement mit unique\_ptr<T>

Dabei ist es gantiert ein «Hochländer» zu sein. Es kann nur eine Referenz geben und keine zweite, oder ... Referenz.

```
#include <memory>
#include <iostream>
std::unique_ptr<int> afactory(int i){
    return std::make_unique<int>(i);
}
int main(){
    auto pi=afactory(42);

    std::cout << "*pi =" << *pi << '\n';
    std::cout << "pi.valid? "<< std::boolalpha
        << static_cast<bool>(pi) << '\n';
    auto pj=std::move(pi);— transfer of ownership from lvalue
    std::cout << "*pj =" << *pj << '\n';
    std::cout << "pj.valid still? "
        << static_cast<bool>(pj) << '\n';— false
}
```

transfer of ownership through return by value

true

transfer of ownership from lvalue

false

## Für C Pointer

Es gibt einige C Funktionen welche Pointers zurückgeben, welche mit der Funktion ::free(ptr) dealloziert werden müssen. Wir können einen unique\_ptr verwenden um dies zu erreichen. \_\_cxa\_demangle() ist zum Beispiel eine solche Funktion.

```
std::string demangle(char const *name){
    std::unique_ptr<char, decltype(&::free)>
        toBeFreed{ __cxxabiv1::__cxa_demangle(name,0,0,0),&::free};
    std::string result(toBeFreed.get());
    return result;
}
```

Selbst wenn es eine Ausnahme geben würde, wird «free» auf den zurückgegebenen Zeiger aufgerufen werden, also kein Speicherleck.

## Richtlinien für unique\_ptr

### Als Membervariable

Um eine polymorphe Referenz zu behalten, die von der Klasse instanziert oder als unique\_ptr übergeben wurde und deren Besitz nun übertragen wird.

### Als Lokale Variable

um RAI (Resource Acquisition Is Initialization) zu implementieren.

### std::unique\_ptr<T> const p{new T{}}; //local

Hier kann der Ownership nicht übertragen werden und daher gibt es auch keine Memory Lecks.

### Shared\_ptr<T> + make\_shared<T>()

Unique Pointers lassen nur einen Besitzer zu und können nicht kopiert werden, also nur als Wert zurückgegeben werden. Die Shared Pointers (shared\_ptr) funktionieren wie die Java Referenzen. Sie können kopiert werden, weitergegeben werden und der letzte Zugriff löscht das Element dann.

Solche Pointer können mit std::make\_shared<T>(...) erstellt werden, in dem der Typ T verwendet wird. Make\_shared lässt alle T «public Constructors» zu.

- Wenn Sie wirklich Heap-zugeordnete Objekte benötigen, weil Sie eigene Objektnetzwerke erstellen, können Sie `shared_ptr <T>` verwenden.
- Wenn Sie zur Laufzeit polymorphe Containerinhalte oder Klassenelemente unterstützen müssen, die nicht als Referenz übergeben werden können, z. B. wegen Laufzeit-Problemen.
- Factory-Funktionen, die `shared_ptrs` für heapzugeordnete Objekte zurückgeben.

Prüfen Sie aber in allen Fällen ob die Alternativen nicht besser sind:

- (Const) Referenzen als Parametertypen oder Klassenmitglieder (zu überlebenden Objekten!)
- Plain-Member-Objekte oder Container mit einfachen Klasse-Instanzen

### Resourcenmanagement mit `shared_ptr<T>`

```
#include <memory> // or <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');
}
```

Wenn man wirklich etwas explizit im Heap haben möchte, sollte man eine Factory, wie diese, benutzen.

Eine Verwendung würde dann wie folgt aussehen.

```
int main(){
    auto an_a=A_factory();
    auto b=an_a; // second pointer to same object
    A c{*b}; // copy ctor.
    auto another = std::make_shared<A>(c); // copy ctor on heap
}
```

### Klassenhierarchien mit `shared_ptr<T>`

Benutzen wir `std::ostream` als ein Beispiel für eine Basisklasse und eine sehr primitive Factory Funktion, welche den konkreten Typ in `make_shared` nutzt.

```
std::shared_ptr<std::ostream> os_factory(bool file){
    if (file)
        return std::make_shared<std::ofstream>("hello.txt");
    else
        return std::make_shared<std::ostringstream>();
}
```

Ein einfaches Anwendungsszenario würde dann wie folgt aussehen.

```
int main(){
    auto out = os_factory(false);
    if (out) (*out) << "hello world\n";
    auto fileout = os_factory(true);
    if (fileout) (*fileout) << "Hello, world!\n";
    fileout.reset(); // clears shared_ptr, deallocates stream object
}
```

not really needed

## Interessante Seiteneffekte

- Wenn der letzte shared\_ptr Handle zerstört wird, wird das allozierte Objekt zerstört bzw. gelöscht.
- Wenn Instanzen einer Klassenhierarchie immer als ein shared\_ptr<Base> repräsentiert werden, aber durch make\_shared<concrete>() erstellen werden, muss der Destruktor nicht weiter «virtual» sein.
- Der shared\_ptr kann aufgrund der Kreisabhängigkeit dazu führen, dass die Objektzykeln nicht weiter gelöscht werden können.
  - o Es ist ein weak\_ptr nötig um solche Zyklen zu verhindern.

### Beispiel, wo weak\_ptr nötig sind

Wir schreiben eine Klasse Person, welche eine Person repräsentiert. Jede Person kennt seine Eltern (Mutter und Vater) und weiss, ob Sie noch leben. Jede Person kann verheiratet sein und kennt ihre Kinder (beide, Mutter und Vater, kennen ihre Kinder).

Wenn wir hier nicht direkt die Members (Personen) nutzen können, würde das dazu führen, dass wir die Personen kopieren müssen.

### Zusammenfassung von shared und weak Pointers

Sei vorsichtig, wenn Sie Objektstrukturen mit shared Pointers (shared\_ptr) erzeugen. Verhindern Sie zyklische Abhängigkeiten.

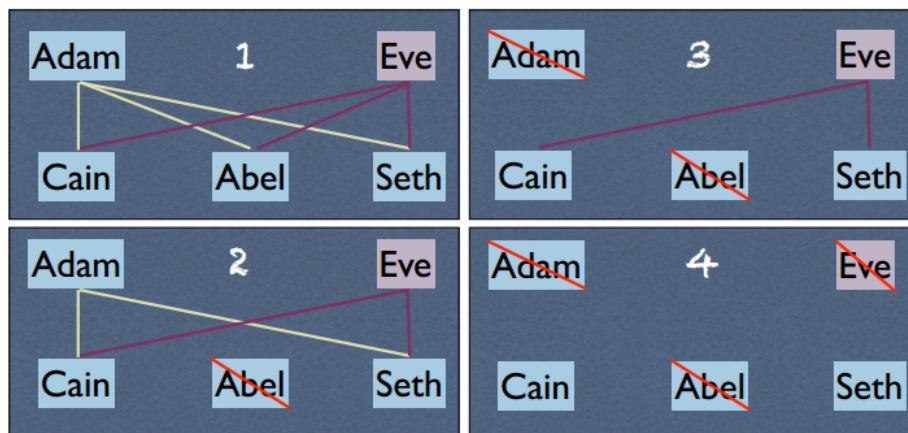
Nutzen Sie daher konsequent Weak Pointers (weak\_ptr). Hält alle «lebende» Objekte in einer separaten Datenstruktur wie ein shared\_ptr und die Modellabhängigkeiten in einem weak\_ptr. Ein Löschen von der «Live» - Liste zerstört das Objekt und der Speicher wird freigegeben wenn der letzte weak\_ptr abläuft.

### Beispiel zu Pointers (Zeigern) – Teile der ersten biblischen Familie

Dabei gibt es Adam und Eva (Eve) sowie die Söhne Abel, Cain und Seth. Gezeigt wird der Tod von Abel durch den Mörder Cain, der Tod von Adam sowie der Tod von Eva. Zudem zeige eine Person mit dem Namen und den Namen der Eltern und deren Kindern an.

Es ist hier nur der Code ohne weitere Erklärungen.

### Schritte und Effekte



```
#include <memory>
#include <string>
#include <vector>
#include <iostream>
using PersonPtr=std::shared_ptr<class Person>

class Person {
    std::string name;           store shared_ptr in vector
    std::vector<PersonPtr> children;
public:
    Person(std::string name):name{name}{}  

    void addChild(PersonPtr child){  

        children.push_back(child);
    }
    void print(std::ostream &) const;
    static PersonPtr makePerson(std::string name){
        return std::make_shared<Person>(name);
    }
};
```

can define shared\_ptr with incomplete type

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

void Person::print(std::ostream& out)
const {
    out << "Person: " << name;
    out << "\n" ;
    for(auto child:children){
        out << child->name << ", ";
    }
    out << '\n';
}
```

### Das erste Main()

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

void addson(std::string name,PersonPtr adam, PersonPtr eva) {
    auto son = Person::makePerson(name);
    eva->addChild(son);           must add shared_ptr not object!
    adam->addChild(son);
}

int main() {
    auto adam=Person::makePerson("Adam");
    adam->print(std::cout);
    auto eva=Person::makePerson("Eva");
    eva->print(std::cout);
    addson("Cain",adam, eva);
    addson("Abel",adam, eva);
    addson("Seth",adam, eva);
    adam->print(std::cout);
    eva->print(std::cout);
    // how to have Cain kill Abel?
}
```

Person: Adam  
Person: Eva  
Person: Adam  
Cain, Abel, Seth,  
Person: Eva  
Cain, Abel, Seth,

### Zweiter Schritt, töten von Abel

```
class Person {
    std::string name;
    PersonPtr father;
    PersonPtr mother;
    std::vector<PersonPtr> children;
public:
    Person(std::string name,
    PersonPtr father,PersonPtr mother)
    :name{name},father{father},mother{mother}{}  

    void addChild(PersonPtr child){  

        children.push_back(child);
    }
    std::string getName() const { return name; }
    PersonPtr findChild(std::string name) const;
    void killChild(PersonPtr child);
    void print(std::ostream &) const;
static PersonPtr makePerson(std::string name,
                           PersonPtr father={},
                           PersonPtr mother={}){
    auto res = std::make_shared<Person>(name,father,mother);
    if (father) father->addChild(res);
    if (mother) mother->addChild(res);
    return res;
};
```

know your parents

search and get rid of children

```

void Person::print(std::ostream& out) const {
    out << "Person: " << name ;
    out << "  " << (father?father->getName()."orphan");
    out << "  " << (mother?mother->getName()."orphan");
    out << "\n  ";
    for(auto const &child:children){
        out << child->name << ", ";
    }
    out << '\n';
}

PersonPtr Person::findChild(std::string theName) const {
    using namespace std::placeholders;
    auto finder=[theName](PersonPtr const &person){
        return person->getName() == theName;
    };
    auto it=find_if(children.begin(),children.end(),finder);
    if (it != children.end()) return *it;
    return nullptr;
}

void Person::killChild(PersonPtr child) {
    if (child){
        children.erase(find(children.begin(),children.end(),child));
        //if (child->father == ) ?
    }
}

```

know your parents

search children, predicate as lambda

nullptr means empty shared\_ptr

erase found object, shared\_ptr == used

## Das Main dazu

```

#include "Person.h"
#include <iostream>

void addson(std::string name,PersonPtr adam, PersonPtr eva) {
    auto son = Person::makePerson(name);
    eva->addChild(son);
    adam->addChild(son);
}

int main() {
    auto adam=Person::makePerson("Adam");
    adam->print(std::cout);
    auto eva=Person::makePerson("Eva");
    eva->print(std::cout);
    addson("Cain",adam, eva);
    addson("Abel",adam, eva);
    addson("Seth",adam, eva);
    adam->print(std::cout);
    eva->print(std::cout);

    // how to have Cain kill Abel?
    {
        auto abel=eva->findChild("Abel");
        eve->killChild(abel);
        adam->killChild(abel);
        abel->print(std::cout);
    }
    eve->print(std::cout);
    // how can we kill Adam?
}

```

last shared\_ptr after killChild  
killing means forgetting all references

Person: Adam	orphan	orphan
Person: Eve	orphan	orphan
Person: Adam	orphan	orphan
Cain, Abel, Seth,		
Person: Eve	orphan	orphan
Cain, Abel, Seth,		
Person: Abel	Adam	Eve
Person: Eve	orphan	orphan
Cain, Seth,		

3:

Das Problem, dass der shared\_ptr nicht zugreifbar ist.

```

class Person
: public std::enable_shared_from_this<Person> {

```

CRTP

Die Lösung ist ein Erben von enable\_shared\_from\_this. Dabei wird das CRTP Pattern eingesetzt (pass own class as template argument).

Der Zugriff findet dann durch die shared\_from\_this() Funktion statt.

```

auto me=shared_from_this();
// or PersonPtr me=shared_from_this();

```

```

class Person
: public std::enable_shared_from_this<Person> {
    std::string name;
    PersonPtr father;
    PersonPtr mother;
    std::vector<PersonPtr> children;
public:
    Person(std::string name, PersonPtr father, PersonPtr mother)
    :name{name}, father{father}, mother{mother} {
        // can not do shared_from_this here!
        // if(father) father->addChild(shared_from_this());
    }
    void addChild(PersonPtr child){
        children.push_back(child);
    }
    std::string getName() const { return name; }
    PersonPtr findChild(std::string name) const;
    void killChild(PersonPtr child);
    void killMe();
    void print(std::ostream &) const;
    static PersonPtr makePerson(std::string name,
                                PersonPtr father={},
                                PersonPtr mother={});
    auto res = std::make_shared<Person>(name, father, mother);
    if (father) father->addChild(res);
    if (mother) mother->addChild(res);
    return res;
};

```

inherit from enable\_shared\_from\_this

disentangling circular object dependency by hand!

```

void Person::killMe() {
    auto me=shared_from_this();
    if (father) father->killChild(me);
    if (mother) mother->killChild(me);
    for(PersonPtr son:children){
        if (me == son->father)
            son->father.reset();
        if (me == son->mother)
            son->mother.reset();
    }
    children.clear();
}

```

must still inform parents in factory function!

## Das Main dazu

```

must disentangle object circular reference before destroying
...
// how can we kill Adam and Eve?
{
    adam->killMe();
    adam->print(std::cout);
    adam.reset();           last reference to Adam forgotten
}
eve->print(std::cout);
auto cain=eve->findChild("Cain");
if (cain) cain->print(std::cout);
eve->killMe(); // avoid memory leak
eve->print(std::cout);
eve.reset();           last reference to Eve forgotten
                        Cain doesn't have children,
                        so no need for killMe()

```

Person: Adam orphan orphan

Person: Eve orphan orphan

Person: Adam orphan orphan  
Cain, Abel, Seth,

Person: Eve orphan orphan  
Cain, Abel, Seth,

Person: Abel Adam Eve

Person: Eve orphan orphan  
Cain, Seth,

Person: Adam orphan orphan

Person: Eve orphan orphan  
Cain, Seth,

Person: Cain orphan Eve

Person: Eve orphan orphan

## Das Problem der zyklischen Objektreferenzen

```

using WeakPersonPtr=std::weak_ptr<class Person>;
WeakPersonPtr father; // don't lock parent objects
WeakPersonPtr mother;

```

Die Lösung ist, eine Richtung basierend auf dem weak\_ptr zu machen. Intern wird das Observer Pattern angewendet. Es merkt, wenn der shared\_ptr nicht existiert. Es muss lock() aufrufen um auf den shared\_ptr zugreifen.

```

auto realfather=father.lock();
out <<(realfather?realfather->getName():"orphan");

```

```

PersonPtr Person::myLock() {
    try {
        auto me=shared_from_this(); ——————
        return me;
    }catch(std::bad_weak_ptr const &ex){}
    std::cout << "====already dead? " << name<< '\n';
    return PersonPtr{}; // already dead
}

void Person::killMe() {
    // here shared_from_this is possible
    auto me=myLock();
    if (!me) return; // already dead
    auto realfather=father.lock();
    if (realmother) realfather->killChild(me);
    auto realmother=mother.lock();
    if (realmother) realmother->killChild(me);
    children.clear();
}

Person::~Person() {
    std::cout << "killing me: "<< name << '\n';
    //killMe(); // can not call shared_from_this() in dtor!
}

```

throws when called from destructor

still need to inform parent,  
because it keeps a shared\_ptr

no more need to inform children,  
because they keep only weak\_ptr

just to show what happens, not needed!

## Das Main dazu

```

int main() {
    auto adam=Person::makePerson("Adam");
    adam->print(std::cout);
    auto eve=Person::makePerson("Eve");
    eve->print(std::cout);
    addson("Cain",adam, eve);
    addson("Abel",adam, eve);
    addson("Seth",adam, eve);
    adam->print(std::cout);
    eve->print(std::cout);
    // Cain kills Abel: need to remove from parents
    {
        auto abel=eve->findChild("Abel");
        eve->killChild(abel);
        adam->killChild(abel);
        abel->print(std::cout);
    }
    eve->print(std::cout);
    // kill Adam by forgetting last reference
    {
        std::cout << "killing Adam:\n";
        adam->print(std::cout);
        adam.reset(); ——————
        last reference to Adam forgotten
    }
    eve->print(std::cout);
    auto cain=eve->findChild("Cain");
    if (cain) cain->print(std::cout);
    eve.reset(); ——————
    last reference to Eve forgotten
    if (cain) cain->print(std::cout);
}

```

no more need to disentangle

last reference to Adam forgotten

last reference to Eve forgotten

Person: Eve	orphan	orphan
Person: Adam	orphan	orphan
Cain, Abel, Seth,		
Person: Eve	orphan	orphan
Cain, Abel, Seth,		
Person: Abel	Adam	Eve
killing me: Abel		
Person: Eve	orphan	orphan
Cain, Seth,		
killing Adam:		
Person: Adam	orphan	orphan
Cain, Seth,		
killing me: Adam		
Person: Eve	orphan	orphan
Cain, Seth,		
Person: Cain	orphan	Eve
killing me: Eve		
killing me: Seth		
Person: Cain	orphan	orphan
killing me: Cain		

# Inheritance and dynamic Polymorphism

## Gründe für Vererbung

Dabei gibt es einige. Hier aufgeführt die wichtigsten, welche im Verlaufe der Vorlesung auch angetroffen worden sind.

- Mischen der Funktionalität einer leeren Basisklasse
  - o Meist geschieht dies mit eigenen Klassen als Template Argument
  - o Zum Beispiel `boost::equality_comparable<T>`.
- Adaptierung von konkreten Klassen (Vorheriges Kapitel)
  - o Somit keinen eigenen Daten - Members
  - o Es ist oft besser den Adapter als Member zu wickeln
  - o Zum Beispiel `IndexableAdapter<T>` oder `IndexableSet<T>`.

## Vererbung für dynamische Bindung

Eine Implementierung eines Design Patterns mit einer dynamischen Aufgabe. Dabei bietet die Vererbung eine gemeinsame Schnittstelle für eine Vielzahl von sich dynamisch verändernden oder unterschiedlichen Implementierungen. Die Austausch findet zur Laufzeit statt. Somit werden eine oder mehrere Klassen auf dieselbe Art und Weise behandelt. Der Schlüssel dabei ist, dass die Basisklasse bzw. die Interfaceklasse eine gemeinsame Abstraktion bietet, die von den anderen Klassen verwendet werden kann.

## Vererbungssyntax

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

Dies geschieht in der Klassendefinition. Nachdem Klassennamen und einem Doppelpunkt. Angegeben wird eine Liste von Basisklassen. Die Reihenfolge ist dabei wichtig, wenn mehrere Klassen angegeben werden.

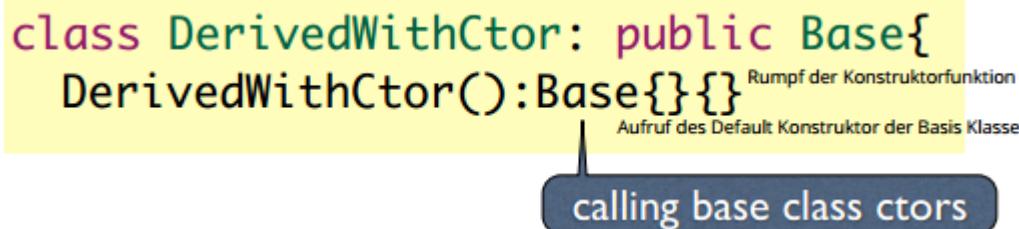
Bei der Interface Vererbung muss die Basisklasse «public» sein. Private Vererbung ist auch möglich, aber nur Sinn wenn man verschiedene Klassen mixen tut, welche eine ähnliche Funktion haben.

```
class DerivedPrivateBase : Base {};
struct DerivedPublicBase : Base {};
```

## Syntax für Mehrfachvererbung (Nicht Prüfungsstoff)

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

Wie bereits vorher angedeutet ist die Mehrfachvererbung möglich. Dabei wird nicht nur eine Klasse sondern, eine Liste von Basisklassen angegeben. Die Reihenfolge spielt dabei eine Rolle. Dabei können public und private Basisklassen gemischt werden. Die private Vererbung kann dazu gebraucht werden um Basisklassen zu mischen, welche nur «Friend Functions» hinzufügen. Ein Beispiel sind die `boost/operators.hpp`. In den meisten Fällen sind aber private Klassen ein falsches Design.



Setzen Sie dabei den Basisklassenaufruf bevor die Member Initialisierung mit einer Konstruktorliste stattfindet. Ein «super()» Aufruf im Body, wie aus Java ist nicht möglich. Es gibt dabei kein «Kurzbefehl» um auf die Basisklasse zu referenzieren.

Wenn die Basisklasse keinen Default Konstruktor hat, kann auch das Objekt nicht erzeugt werden.

Basisklasse «ctor» bevor der Member Initialisierung

```
class DerivedWithCtor: public Base{
  DerivedWithCtor(int i, int j)
  : Base{i}, mvar{j} {}
```

Wenn die eigenen Member initialisiert werden, werden die Basisklassen zuvor initialisiert. Der Kompiler erzwingt diese Regeln, obwohl Sie die Liste der Initialisierungswerte in einer falschen Reihenfolge angeben können. Es ist zudem auch möglich die Erzeug zu einem anderen Konstruktor in der eigenen Klasse zu delegieren.

### Wann Vererbung schlecht ist

- Die Vererbung führt eine sehr strenge Koppelung zwischen der Basisklassen und deren Subklasse ein. Dies erschwert es sehr die Basisklasse zu ändern.
- Die API der Basisklasse must für alle Subklassen passen. Es ist sehr schwer richtig zu machen.
- Konzeptionelle Hierarchien werden meist als Beispiel gebraucht, sind aber sehr schlechtes Software Design. Zum Beispiel Animal → Bird → Duck.
- Nur eine Standard Library (die älteste) nutzt Vererbung mit dynamischen Polymorphismus. Ist ist iostremas. Eine vereinfachte Zeichung ist im weiteren Verlauf des Dokumentes zu finden.

## Vererbung und Sichtbarkeit

### Public

Die Members der Basisklasse sind sichtbar und nutzt in den abgeleiteten Klassenobjekten. Es sei denn, die abgeleitete Klasse definiert Members mit demselben Namen (auch mit unterschiedlichen Parametern).

### Protected

Members der Basisklasse können in der abgeleiteten Klasse genutzt werden. Meist ist protected überbewertet, da es die Kapselung aufbricht.

### Private

Auf die Members der Basisklasse kann nicht zugegriffen werden.

**Public**

Für Members welche das Interface einer Klasse formen. Meist member functions, Typen (alias) und teilweise auch Konstante.

**Protected**

Die Members, welche von der abgeleiteten Klasse verwendet werden sollen, wenn die Klasse als Basisklasse konzipiert ist. Meistens «overused» oder anstatt «private» genutzt. ➔ Achtung bricht die Kapselung auf.

**Private**

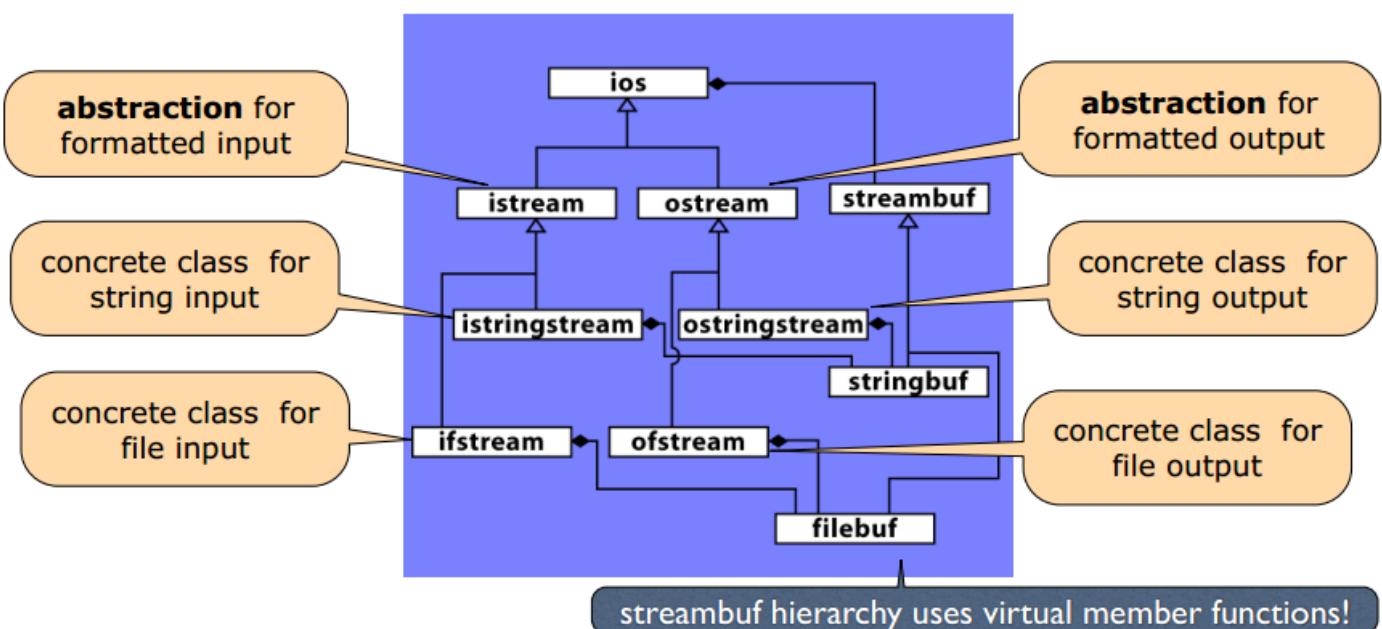
Members welche für die Implementation nötig sind, aber nicht durch die Klassenbenutzer ungreifbar sein sollen. So werden Variablen gekapselt.

**Dynamischer Polymorphismus****Wichtig**

Wenn man auto &x = a macht findet ein Copy statt. Evtl. neues wird abgeschnitten (Splicing). Das heisst alles was darunter liegt. Richtig funktionieren würde decltype(auto)x a

**Wann wird es in C++ gebraucht**

- Bei der Implementierung von Design Patterns mit Laufzeitflexibilität: Zum Beispiel das Strategy, das Composite oder das Decorator Pattern.
  - o In diesen Fällen nutzt der «Client» ein abstraktes Interfaces und wird parametrisiert aufgerufen mit einer Referenz auf die konkrete Instanz.
- Aber auch wenn die Laufzeitflexibilität nicht erforderlich ist, können Templates viele Patterns mit einer Laufzeitflexibilität implementieren.
- Meist ist es (miss)braucht für konzeptionelle Hierarchien mit einer is-a Verbindung.
  - o Auch wenn fast alle Einführungsbeispiele Sie verwenden.

**Hierarchie von iostremas (Vereinfacht)**

```
#include <iostream>
struct Bird {
    bird = hummingbird;
```

Beim Zuweisen oder Übergeben (By Value) eines Wertes der abgeleiteten Klasse in einen Basisklassenparameter findet Object Slicing statt. Das heisst, dass nur Basisklassenmembervariablen transferiert werden.

Ein weiteres potentielles Problem ist das Hiding Problem. Member Funktionen der abgeleiteten Klasse verstecken die Basisklassenmember mit dem sieben Namen. (auch wenn die Parameter nicht gleich sind). Dies kann bei const/non-const problematisch sein.

### Member Hiding Problem

```
#include <iostream>
struct Base {
    void foo(int i) const {
        std::cout << "foo(i):" << i << '\n';
    }
};

struct Derived:Base {
    void foo() {
        std::cout << "Derived::foo()\n";
    }
};
int main(){
    Base b{};
    b.foo(42);
    Derived d{};
    //d.foo(42);
    d.foo();
}
```

hides Base::foo(int) member function

can not call Base::foo(int) on Derived objects

Die überladenen Member Funktionen in den Subklassen verstecken alle Member Class Funktionen der Basisklasse mit demselben Namen.

Alternativ kann man das zu warnende Überladungsschlüsselwort angeben (override).

Dabei ist zu beachten, dass dies mit Typen wie float und int nicht so

offensichtlich ist. In diesem Fall kommt es dann nicht zu einem Fehler, sondern der Typ wird automatisch umgewandelt.

### Lösung

```
#include <iostream>
struct Base {
    void foo(int i) const {
        std::cout << "foo(i):" << i << '\n';
    }
};

struct Derived:Base {
    using Base::foo; // provide all Base::foo members as class members
    void foo() {
        std::cout << "Derived::foo()\n";
    }
};
int main(){
    Base b{};
    b.foo(42);
    Derived d{}; // can now call Base::foo(int)
    d.foo(42);
    d.foo();
}
```

Mit dem Wort using können die Implementationen der Basisklasse «importiert» werden.

Somit stehen dem Programm dann beide foo()-Funktionen zur Verfügung.

## Virtuelle Member Funktionen

Der dynamische Polymorphismus benötigt Basisklasse mit «virtual» Member Funktionen.

Nicht-«virtual» Memberfunktionen sind statisch gebunden und somit nicht dynamisch. Das heisst, dass immer die Implementation der Basisklasse genommen wird. (also immer die eins höhere in der Hierarchie).

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

class Implementor: public PolymorphicBase {
public:
    void doit() {
        /* something else */
    }
};
```

client API of abstraction with virtual member

virtual can be omitted in derived class

can be marked with 'override', that will produce a compile error when signature differs from base class member function.  
IDE will also show that, IMHO 'override' is superfluous.

## Const nicht vergessen

Member Funktionen mit oder ohne const sind unterschiedlich. Es können beide Overloads definiert werden.

Eine abgeleitete Klasse welche Member Funktionen mit einer anderen «const-ness» zur Verfügung als die Basisklasse, wird die Basisklassenfunktionen nicht überschreiben. Der selbe Name wird aber die Basisklasse verstecken.

## Aufruf von virtuellen Member Funktionen

- Werte Objekt (direkt)
  - o Es wird die Funktion des Klassentyps verwendet, unabhängig von virtuell
- Referenzen oder Pointer (indirekt)
  - o Dabei wird das virtuelle Member der abgeleiteten Klasse aufgerufen, das über die Basisklassenreferenz aufgerufen wird.
  - o Dies aber nicht für Member-Funktionen, die nicht virtuell sind.

```
#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";}
};
```

```
#include "AnimalBirdHummingbird.h"
// bad code for demonstration purpose only!
int main(){
    cout << "(a)-----\n";
    1 Hummingbird hummingbird; 3 Ausgabe von born()
    2 Bird         bird = hummingbird;
    3 Animal &     animal = hummingbird;
    cout << "(b)-----\n";
    4 hummingbird.makeSound(); "peep".
    5 bird.makeSound(); "chirp"
    6 animal.makeSound(); "--, da nicht virtual immer zu oberst.
    cout << "(c)-----\n";
    7 hummingbird.move(); "hum"
    8 bird.move(); "fly"
    9 animal.move(); "hum" => oberstes Virtual gilt.
    cout << "(d)-----\n";
}

} Des Konsrtor Anima ==> keine Destruktion da Referenz ==> keine Ausgabe
Des Brid ==> brid crashed ==> animal died (auch Methoden von Unten nach Oben)
```

## **Beschreibung der verschiedenen Ausgaben**

1	<p><i>animal born</i>  <i>bird hatched</i>  <i>hummingbird hatched</i></p> <p>Bei der Erstellung eines abgeleiteten Objektes werden jeweils zuerst Objekte jeder Basisklasse von oben nach unten erstellt. Daher 3 Ausgaben.</p>
2	Eine Ausgabe findet nicht statt. Es kommt aber zum Objekt Slicing und daher ist alles von Humming Brid nicht mehr vorhanden.
3	Hier wird nur die Referenz abgespeichert. Es ist alles von Humming Brid weiterhin vorhanden und es wird auf dem gleichen Objekt gearbeitet.
4	<i>Peep</i>  Die entsprechende Funktion wird aufgerufen. Dabei wird direkt auf der jeweiligen Klasse begonnen, da der Typ Hummingbrid auch hummingbrid ist.
5	<i>Chirp</i>  Die entsprechende Funktion wird aufgerufen. Dabei wird direkt auf der jeweiligen Klasse begonnen. Das Objekt wurde ja geslicet und daher ist alles von Hummingbrid nicht mehr vorhanden und die Methoden der nächst höheren Klassen werden aufgerufen.
6	---  Es wird beim Typ begonnen. Dies ist in diesem Fall Animal. Die Methode makeSound() verfügt auf dieser Höhe nicht über virtual und wird daher nicht weitergegeben, obwohl eigentlich ein Hummingbrid darin steckt.
7	<i>Hum</i>  Die Methode ist direkt in Hummingbrid vorhanden und wird dort aufgerufen.
8	<i>Fly</i>  Wie vorher ist durch das Slicing alles von Hummingbrid nicht mehr verfügbar und daher wird alles von Brid genommen. Der Aufruf findet auch direkt statt.
9	<i>Hum</i>  Es wird beim Typ Animal begonnen. Da es virtual ist wird es an die entsprechende Klasse weitergegeben. Die Klassen welche dazwischen liegen sind egal.

<b>10 Destrukt- ion Animal</b>	Es findet keine Destruktion statt, da es sich dabei ja um eine Referenz handelt. Somit wird auch nichts auf der Konsole ausgeben.
<b>11 Destrukt- ion Brid</b>	<i>bird crashed</i> <i>animal died</i> Der Abbau findet nun über die Destruktoren statt. Es werden alle Destruktoren der gesamten Hierarchie aufgerufen. Von unten nach oben natürlich.
<b>12 Destrukt- ion Hum</b>	<i>hummingbird died</i> <i>bird crashed</i> <i>animal died</i> Wie der Aufbau auch der Abbau. Es werden alle Destruktoren der gesamten Hierarchie aufgerufen. Von unten nach oben natürlich.

Bevor die Objekte selbst abgebaut werden können, werden zuvor alle Referenzen abgebaut. Daher werden die Destruktoren in umgekehrter Reihenfolge aufgerufen.

Wenn ein Parameter als Objekt mitgegeben wird, wird ein Objekt auch erstellt und es findet eine entsprechene Ausgabe des Konstruktors statt. Zudem sollten Parameter grundsätzlich bei Referenz sein, da sonst immer Objekt Slicing stattfindet.

### Was ist schlecht an diesem Code?

Alle Memberfunktionen sollten const sein, da das Objekt mit keiner Funktion verändert wird. Die Konstruktoren sollten virtual sein, da sonst Klassen ev. nicht sauber aufgeräumt werden (bei der Verwendung von (Smart-)Pointern/Referenzen).

Nicht virtuelle Methoden werden in der Unterkasse überschrieben → Funktions Hiding., mit dem Typ der Unterkasse kann dann die Funktion der Oberklasse nicht mehr aufgerufen werden (ausser Sie werden in der Unterkasse in den Namespace geladen, was hier nicht der Fall ist.). Des weiteren ist die Initialisierung nicht offensichtlich, da die Klammern fehlen.

### Abstrakte Basis Klassen – Pure Virtual

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

pure virtual member function

Die «virtual» Member Funktionen der Basisklasse können «Pure virtual» oder «Abstrakt» sein. Damit wird eine Implementation gemacht und dies explizit mit = 0 angeben. Jede abgeleitete Klasse muss dann diese Methoden implementieren, ansonsten können Sie nicht kreiert werden.

Eine Basisklasse mit virtuellen Members brauchen einen virtual Destruktor, wenn der Heap nicht mit einem shared\_ptr alloziert werden. Brauchen Sie sowieso immer shared\_ptr<Base> für Objekte auf dem Heap. Somit müssen virtuellen Destruktoren nicht definiert werden.

## Gebrauch von Polymorphen Klassen

- Geben Sie die Basisklassenreferenz als Funktionsparameter an
  - o Zum Beispiel `print(std::ostream &)`
  - o Const-Referenzen funktionieren auch, wenn nur const Member Funktionen aufgerufen werden
- Keine Heap-Zuweisung für die Subklassen-Objekte
  - o Übergeben Sie diese einfach als Argumente
- Wenn `heap allocated` nicht verhindert werden kann, dann sollte man immer `std::shared_ptr<Base>` brauchen und diese mit einem `make_shared<ConcreteDerived>` initialisieren.

## Wieso brauchen wir das «virtual»-Schlagwort?

Die Philosophy von C++ ist, dass du nicht einen Preis «zahlen» musst, für etwas was du nicht verwendest. Member-Funktionsaufrufe abhängig vom dynamischen Typ eines Objekts erfordern Overhead.

Die Adressen der Virtual Member Funktionen werden in einer Tabelle pro Klasse abgelegt und jedes Objekt hält eine Referenz zu dieser Tabelle. Diese Tabelle wird meist «vtable» genannt. Das Resultat dieser Tabelle ist ein Overhead in der Objektgrösse.

## Design Guidelines

### Niedrige Koppelung

Versuchen, den Zweck einer Funktion / Klasse " mit einem minimalen Satz von Abhängigkeiten zu erreichen. Die Vererbung ist dabei eine sehr strenge Kopplung und sollte wenn immer möglich vermieden werden.

### Hohe Kohäsion

Eine Funktion/Klasse sollte eine Sache gut machen. Kombinieren Sie nicht unabhängige Funktionalität. Teilen Sie Klassen/Funktionen auf, welche zu viel machen.

Ich merke, ob ich dies erreicht habe, wenn ich gut und einfach Testcases machen kann.

### Vererbung und Dynamischer Polymorphismus

Sie sollten nur Vererbung und virtuelle Member-Funktionen anwenden, wenn Sie wissen, was Sie tun. Nicht wie die IDE welche Klasse mit virtuellen Membern erstellt.

Wenn Sie Basisklassen mit polymorphen Verhalten zu entwerfen, verstehen Sie die gemeinsame Abstraktion, die sie präsentieren. Stellen Sie dabei nicht zu wenige oder zu viele Members zur Verfügung. Extrahieren Sie die Basisklasse von existierenden Klassen.

### Regeln zum Überschreiben von «virtual» Members

- Folge dem Liskov Substitution Principle
  - o Basisklassenstatus muss für Unterklasse gültig sein
  - o Brechen Sie nicht die Invarianten der Basisklasse
  - o Member-Funktionen in der Unterklasse müssen mindestens die Argumente der Basisklasse akzeptieren und keine Werte außerhalb des Bereichs der Basisklassen-Memberfunktion zurückgeben (co- und contra-variance).
  - o Ändern Sie die Semantik nicht unerwartet.