Zusammenfassung

C++

Michael Wieland

Hochschule für Technik Rapperswil

13. August 2017

Mitmachen
Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter `https://github.com/michiwieland/hsr-zusammenfassungen` forken.

Inhaltsverzeichnis

1. Konzepte: Der heilige Gral

- In C++ sind alles standardmässig Wertetypen und werden auf dem Stack alloziert
- Klassenmember sind implizit `inline` und `private`. Bei Structs sind die Member implizit `public`.
- Der Copy Konstruktor ist implizit verfügbar
- Parameter werden standardmässig **By Value** übergeben. Das beudetet, es wird eine Kopie des Parameters angelegt. Dies ist ein Problem bei grösseren Objekten. Man sollte diese deshalb mit dem `&` Operator explizit **By Reference** übergeben.
- Unveränderbare grosse Objekte sollten per `const &` (Const Referenz) übergeben werden
- Zur Funktionssignatur wird der Name, die Parameter und `const` dazugezählt.
- ";" hinter jede Klasse, Struct und Enum
- Konstruktoren mit Parameter sollten als `explicit` deklariert werden.
  `explicit MyClass(int a, int b): _a{a}, _b{b} {..validate..}`
- Variablen initialisieren mit geschweifte Klammern `{}` oder `=` bei `auto` Variablen (Theoretisch geht auch `={}`)
- Alles was nicht verändert wird oder etwas verändert `const` deklarieren.
- `const` Variablen müssen **direkt oder im Konstruktor** initialisiert werden.
- Nie Referenzen auf lokale Variablen einer Funktion zurückgeben, da der Stack nach dem Funktionsaufruf abgebaut wird.
- Es gibt zwei Variaten um auf das `this` Objekt zuzugreifen: `this->m();` und `(*this).m();`
- Eine Methode in der Basisklasse kann über eine Typendefinition aufgerufen werden.
  `using super = std::WHATEVER<T>; super::method();`
- Functoren überladen den call Operator: `void operator()(int value){..}`
- Bei Template Klassen sollte auf vererbte Member immer über das `this` Objekt zugegriffen werden.
- Memberfunktionen die nie benutzt werden, werden bei Template Klasse nie kompiliert.
- Mit dem `class` Keyword (Scoped) Enum ist der Enum nur innerhalb des Namespaces sichtbar. Ansonsten global.
- (std::string = `"mystring"s`) $\neq$ `"ab"` (char array)

1.1. Vererbung

- Beim **Aufbau** der Objekte bei der Vererbung: von Base nach Sub
- Beim **Abbau** der Objekte bei der Vererbung: von Sub nach Base.
- Bei Mehrfachvererbung wird von links nach rechts die Konstruktoren aufgerufen. Bei der Dekonstruktion genau umgekehrt
- Bei Mehrfachvererbung wird der Parent bei der Diamanten Form mehrere Male aufgerufen
- Vererbung ist bei Klassen standardmässig `private`. Möchten man die Membervariablen des Base Klasse nach aussen tragen, muss dies explit angegeben werden. (`class MyClass<T> : public Base<T> {..}`)
- Bei Structs ist die Vererbung implizit `public`
- Beim **Object Slicing** ist der statische und dynamische Typ gleich

Konstruktoren

- Der Copy Konstruktor ist ein impliziter, eigenständiger Konstruktur und führt keine anderen Konstruktoren aus.
- Beim Copy Konstruktor gibt es immer Object Slicing. Er ist implizit verfügbar.
- Die Konstruktoren werden nicht implizit vererbt

Methoden Overloading / Overriding

- Ist der statische Typ nicht mit dem dynamischen Typ identisch, werden die Methoden im Parent gesucht
- `const` wird zur Signatur gezählt
- Sobald eine Methode virtual ist, kommt Dynamic Dispatch zum Zug (Methode des dynamischen Typs wird genommen). Sonst die Methode des statischen Typs.
- Bei überdeckten Methoden wird die Methode im statischen Typ ausgegeben.
- Ist die Methode im Parent `virtual`, sind die Methoden in den Childs **implizit** `virtual`

Referenzen

- Bei einer Referenz wird kein neues Objekt erstellt:
  Statischer Typ: Neuer Typ und **Dynamischer Typ**: alter Typ. Es wird kein Konstruktor aufgerufen!
- Zuweisungen oder übermitteln von Parameter `by value` von abgeleiteten Klassen in Variablen vom Typ der Base Klasse resultieren in **Object Slicing**
  → Nur Base-Class Member Variablen werden behaldent. (`MyBase base = subVar;`)
- Bei Referenzen wird kein Destruktor aufgerufen

Häufige Fehler

- Kein virtueller Dekonstruktor → häufig zur Folge, dass die Objekte nicht sauber abgeräumt werden können. Dekonstrukoren sollten deshalb virtual sein, wenn eine Methode virtual ist.
- Function Hiding: Nicht virtuelle Methoden werden in der Subclass überschrieben.
- Member Funktionen, die nichts verändern, sollten const sein.
- Mehrfachvererbung

1.2. Iteratoren

- Der Stern-Operator(*) bindet mehr als "+1" (Präzedenz) → Char wird ausgelesen und dann +1 gezählt → nächster Buchstabe aus ASCII
- `(*it)++` = Wert wird gelesen und inkrementiert
- `*(it++)` = Inkrementiert den Iterator und greift dann auf das Objekt zu
- Geht man mit dem `std::advance` Algorithmus über den Bereich des Containers, resultiert **Undefined behaviour**
- Inklusive Leerzeichen: `using it = std::istreambuf_iterator<char>;`
- Exklusive Leerzeichen: `using it = std::istream_iterator<char>;`
- Bei `rend()` muss `--` genommen werden, um nach "vorne" zu iterieren
- `std::advance(it, 2)` verändert das `it` Argument. Gibt nichts zurück.
- `std::next(it, -2)` verändert das `it` Argument nicht. Gibt eine Kopie des Iterators zurück (verschoben um x)

```
1 // replace WHATEVER!
2 std::WHATEVER::iterator
3 std::WHATEVER::const_iterator
4 std::WHATEVER::reverse_iterator
5 std::WHATEVER::const_reverse_iterator
6
7 rbegin() = --end()
8 rend() = --begin()
```

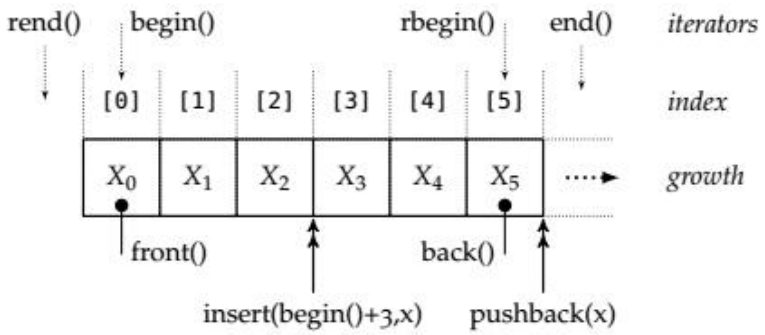

Abbildung 1: Iteratoren

1.3. ASCII Tabelle

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII
0	0x00	000	NUL	*32*	0x20	040	SP	*64*	0x40	100	@	*96*	0x60	140	`
1	0x01	001	SOH	*33*	0x21	041	!	*65*	0x41	101	A	*97*	0x61	141	a
2	0x02	002	STX	*34*	0x22	042	"	*66*	0x42	102	B	*98*	0x62	142	b
3	0x03	003	ETX	*35*	0x23	043	#	*67*	0x43	103	C	*99*	0x63	143	c
4	0x04	004	EOT	*36*	0x24	044	$	*68*	0x44	104	D	*100*	0x64	144	d
5	0x05	005	ENQ	*37*	0x25	045	%	*69*	0x45	105	E	*101*	0x65	145	e
6	0x06	006	ACK	*38*	0x26	046	&	*70*	0x46	106	F	*102*	0x66	146	f
7	0x07	007	BEL	*39*	0x27	047	'	*71*	0x47	107	G	*103*	0x67	147	g
8	0x08	010	BS	*40*	0x28	050	(	*72*	0x48	110	H	*104*	0x68	150	h
9	0x09	011	HT	*41*	0x29	051	)	*73*	0x49	111	I	*105*	0x69	151	i
10	0x0A	012	LF	*42*	0x2A	052	*	*74*	0x4A	112	J	*106*	0x6A	152	j
11	0x0B	013	VT	*43*	0x2B	053	+	*75*	0x4B	113	K	*107*	0x6B	153	k
12	0x0C	014	FF	*44*	0x2C	054	,	*76*	0x4C	114	L	*108*	0x6C	154	l
13	0x0D	015	CR	*45*	0x2D	055	-	*77*	0x4D	115	M	*109*	0x6D	155	m
14	0x0E	016	SO	*46*	0x2E	056	.	*78*	0x4E	116	N	*110*	0x6E	156	n
15	0x0F	017	SI	*47*	0x2F	057	/	*79*	0x4F	117	O	*111*	0x6F	157	o
16	0x10	020	DLE	*48*	0x30	060	0	*80*	0x50	120	P	*112*	0x70	160	p
17	0x11	021	DC1	*49*	0x31	061	1	*81*	0x51	121	Q	*113*	0x71	161	q
18	0x12	022	DC2	*50*	0x32	062	2	*82*	0x52	122	R	*114*	0x72	162	r
19	0x13	023	DC3	*51*	0x33	063	3	*83*	0x53	123	S	*115*	0x73	163	s
20	0x14	024	DC4	*52*	0x34	064	4	*84*	0x54	124	T	*116*	0x74	164	t
21	0x15	025	NAK	*53*	0x35	065	5	*85*	0x55	125	U	*117*	0x75	165	u
22	0x16	026	SYN	*54*	0x36	066	6	*86*	0x56	126	V	*118*	0x76	166	v
23	0x17	027	ETB	*55*	0x37	067	7	*87*	0x57	127	W	*119*	0x77	167	w
24	0x18	030	CAN	*56*	0x38	070	8	*88*	0x58	130	X	*120*	0x78	170	x
25	0x19	031	EM	*57*	0x39	071	9	*89*	0x59	131	Y	*121*	0x79	171	y
26	0x1A	032	SUB	*58*	0x3A	072	:	*90*	0x5A	132	Z	*122*	0x7A	172	z
27	0x1B	033	ESC	*59*	0x3B	073	;	*91*	0x5B	133	[	*123*	0x7B	173	{
28	0x1C	034	FS	*60*	0x3C	074	<	*92*	0x5C	134	\	*124*	0x7C	174	|
29	0x1D	035	GS	*61*	0x3D	075	=	*93*	0x5D	135	]	*125*	0x7D	175	}
30	0x1E	036	RS	*62*	0x3E	076	>	*94*	0x5E	136	^	*126*	0x7E	176	~
31	0x1F	037	US	*63*	0x3F	077	?	*95*	0x5F	137	_	*127*	0x7F	177	DEL

Abbildung 2: ASCII Tabelle

1.4. Constness

Generell

- Const Variablen müssen **direkt** initialisiert werden!
- `const int` und `int const` ist das Selbe
- Eine Variable die später nochmals verändet wird, kann nicht const sein
- Ein const Objekt kann nicht an eine nicht const Referenz gebunden werden
- Ist eine Collection const, müssen die Iteratoren darauf, auch const sein.
- Man kann ein Objekt **direkt nach seiner Deklaration** initialisieren. Die direkte Initialisierung kann `const` sein. (`class A {}const myvar{};`)
- Wenn auf einem Const Objekt eine Memberfunktion aufgerufen wird, die das `this` Objekt verändert, kann die Variable nicht `const` sein.

Klasse und Structs und Enums

- Klassen, Structs und Enum sind **nicht const**
- Instanzen von Klassen/Structs können const sein, sofern nachher auf ihnen keine Methode aufgerufen wird, die etwas verändern würde
- Const Konstruktoren gibt es nicht, da sie `this` Objekt "erstellen"
- Const Werte als Parameter bei Konstruktoren sind möglich
- Enum values können nicht const sein
- Enum Typangaben können const sein, macht aber wenig Sinn.

Methoden

- Non- Memberfunktionen können nicht `const` sein.
- Parameter sind beim Aufrufer nie `const`, sondern nur beim Empfänger.
- Methoden sollten wenn immer möglich als const deklariert werden
- Gibt eine Methode eine Refenz zurück, ist der Rückgabewert meistens const
- Verändert die Methoden keinen Wert ist die Methode const
- Bei kopierten Rückgabewerten macht const keinen Sinn
- Überladene Vergleichsoperatoren sind in der Regel `const` (Rückgabewert allerdings nicht, ausser es ist eine Referenz) → Ausnahme: `++,--`
- Returnwerte als const zurückzugeben macht oft keinen Sinn. `const void` ist zwar nicht falsch, macht aber keinen Sinn

1.5. Algorithmen

```cpp
// transform
std::string const input("teststring");
std::set<char> myset { };
std::transform(input.begin(),input.end(),inserter(myset, myset.begin()), [](char
    const c) {
   return tolower(c);
});

// copy
std::ostream_iterator<char> out{std::cout, "delmiter"};
std::copy(myset.begin(), myset.end(), out);

// erase / remove
vec.erase(remove_if(begin(vec), end(vec), [](auto const x) {
   return x == 2;
}), end(vec));

// find_if und insert
auto res = std::find_if(v.begin(), v.end(), [&item](T const curr){
   return item <= curr;
});
v.insert(res, item);

// for_each
std::for_each(begin(items), end(items), [&out](auto s){
   out << s;
});
```

1.6. Includes

```cpp
#include <string>       -> string
#include <algorithm>    -> transform, copy, find
#include <cctype>       -> lowercase, isalpha, isblank, toupper
#include <iterator>     -> begin, end, ostream_iterator
#include <iostream>     -> cout, cin
#include <iosfwd>       -> ostream, istream (header)
#include <stdexcept>    -> out_of_range, runtime_error, range_error
#include <sstream>      -> istringstream, ostringstream
#include <functional>   -> greater, less, logical_and
#include <numeric>      -> iota, accumulate
```

1.7. Templates

```
#ifndef INDEXABLESET_H_
#define INDEXABLESET_H_

#include <functional>
#include <set>
#include <stdexcept>

#include <string>

// or std::greater<T> for ascending
template<typename T, typename COMPARE = std::less<T>>
class indexableSet: public std::set<T, COMPARE> {
   using base=std::set<T, COMPARE>;
   using const_reference = typename base::const_reference;
   using size_type = typename parent::size_type;
   using iterator = typename base::iterator;

   const std::string ERROR_EMPTY{"indexableSet is empty"};
   const std::string ERROR_INVALID_INDEV{"index was to big"};

   // inherit constructors
   using base::set;



   // adapter
   base a{};
   indexableSet<T, COMPARE>() = default; // default constructor
   indexableSet<T, COMPARE>(std::initializer_list<T> list) : a{list} {..}
   template <typename IT>
   indexableSet<IT begin, IT end) : a {begin, end} {..} // it constructor

public:
   const_reference back() const;
   const_reference front() const;
   const_reference at(size_type const & index) const;
   T const & operator[](size_type const & index) const {
      return this->at(index);
   }

   // adapter
   size_type size() const {
      return a.size();
   }
   iterator begin() {
      return a.begin();
   }

   // Template Funktionen
   template <typename S, typename ...VARARGS>
   bool mytempfunc(S param, VARARGS ...args) { .. }
};

template<typename T, typename COMPARE>
inline T const & indexableSet<T, COMPARE>::back() const {
   if (this->empty()) {
      throw std::out_of_range { ERROR_EMPTY };
   }
   return *this->rbegin();
```

```
60    }
61
62    template<typename T, typename COMPARE>
63    inline T const & indexableSet<T, COMPARE>::front() const {
64       if (this->empty()) {
65          throw std::out_of_range { ERROR_EMPTY };
66       }
67       return *this->begin();
68    }
69
70    template<typename T, typename COMPARE>
71    inline T const & indexableSet<T, COMPARE>::at(const size_type & index) const {
72       typename std::set<T, COMPARE>::iterator it { };
73
74       if (index < 0) {
75          if (std::abs(index) > static_cast<size_type>(this->size())) {
76             throw std::out_of_range { ERROR_INVALID_INDEV };
77          }
78          it = this->end();
79       } else {
80          if (index >= static_cast<size_type>(this->size())) {
81             throw std::out_of_range { ERROR_INVALID_INDEV };
82          }
83          it = this->begin();
84       }
85
86       std::advance(it, index);
87       return *it;
88    }
89
90    #endif /* INDEXABLESET_H_ */
```

1.8. Testate

Listing 1: Basic Header File

```
#ifndef WORD_H_
#define WORD_H_

#include<string>
#include<iosfwd>

namespace word {

   // declarations
   class Word {

      std::string value;
      static bool isValidWord(std::string const & value);

      public:
      Word() = default;
      explicit Word(std::string const & value);

      void print(std::ostream & os) const;
      void read(std::istream & is);

      bool operator <(Word const & rhs) const;
   };

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

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

   inline bool operator !=(Word const & lhs, Word const & rhs) {
      return !(lhs == rhs);
   }

   inline bool operator >(Word const & lhs, Word const & rhs) {
      return rhs < lhs;
   }

}
#endif /* WORD_H_ */



#ifndef KWIC_H_
#define KWIC_H_
#include <iosfwd>
namespace kwic {
   void kwic(std::istream & in, std::ostream & out);
}
#endif /* KWIC_H_ */
```

Listing 2: Basic Cpp File

```cpp
#include"word.h"

#include<algorithm>
#include<iterator> // begin, end, istreambuf_iterator
#include<cctype> // isalpha
#include<stdexcept> // invalid argument
#include<istream>
#include<ostream>
#include<string>

Word::Word(std::string const & value) : value { value } {
   if (!isValidWord(value)) {
      throw std::invalid_argument{"msg"};
   }
}

bool Word::isValidWord(std::string const & value) {
   return !value.empty() && std::all_of(
   std::begin(value),
   std::end(value),
   static_cast<int(*)(int)>(std::isalpha));
}

void Word::print(std::ostream & os) const {
   os << value;
}

void Word::read(std::istream & is) {
   // skip no alpha characters first
   while (is.good() && !std::isalpha(is.peek())) {
      is.ignore();
   }

   // read alpha characters into separate buffer
   std::string buffer{};
   while (is.good() && std::isalpha(is.peek())) {
      buffer += is.get();
   }

   // set value
   if (isValidWord(buffer)) {
      value = buffer;
   } else {
      is.setstate(std::ios_base::failbit);
   }
}

bool Word::operator <(Word const & rhs) const {
   return std::lexicographical_compare(
   std::begin(value),
   std::end(value),
   std::begin(rhs.value),
   std::end(rhs.value), [](char l, char r) {
      return std::tolower(l) < std::tolower(r);
   });
}

bool Word::operator ==(Word const & rhs) const {
   return std::equal(
   std::begin(value),
```

```
61          std::end(value),
62          std::begin(rhs.value),
63          std::end(rhs.value),
64          [](char l, char r) {
65             return std::tolower(l) == std::tolower(r);
66          });
67       }
68
69       // KWIC
70       void kwic::kwic(std::istream & input, std::ostream & output) {
71          using word::Word;
72          using line = std::vector<Word>;
73          using sorted_lines = std::multiset<line>;
74
75          sorted_lines input_lines = readLines(input);
76          sorted_lines rotated_lines = rotate_lines(input_lines);
77          std::copy(
78             std::begin(rotated_lines),
79             std::end(rotated_lines),
80             std::ostream_iterator<line>{output, "\n"});
81       }
```

Listing 3: Basic Test File

```
1
2       void test_exercise_example() {
3          std::istringstream input{"compl33tely"};
4          Word w{};
5          input >> w;
6          ASSERT_EQUAL(Word{"compl"}, w);
7       }
```

Listing 4: Einener Comparator

```
1  // basic functor
2  #include <algorithm>
3  #include <iterator>
4  #include <cctype>
5  struct Caseless{
6     using string=std::string;
7     bool operator()(string const &l, string const &r){
8        return std::lexicographical_compare(
9           l.begin(),l.end(),r.begin(),r.end(),
10             [](char l,char r){
11                return std::tolower(l) < std::tolower(r);
12             });
13    }
14 };
15
16 // generic functor solution
17 #include <cctype>
18 template <typename T>
19 struct GenericCaseless {
20    bool operator()(T const& lhs, T const& rhs) {
21       return std::tolower(lhs) < std::tolower(rhs);
22    }
23 };
24
25 using caseless_set=std::set<string, GenericCaseless>;
```

```
#include "pocketcalculator.h"
#include "src/DivisionByZeroException.h"
#include "src/InvalidOperatorException.h"

#include <iostream>
#include <sstream>
#include <string>

void runCalculator(std::istream & in, std::ostream &out) {
   signed int lhs{};
   signed int rhs{};
   char op{};
   std::string errorMessage{"Error"};

   std::string line{};

   while(getline(in, line)) {

      std::istringstream lineStream{line};

      lineStream >> lhs >> op >> rhs;

      if (lineStream.bad() || lineStream.fail() || lhs < 0 || rhs < 0) {
         printLarge(errorMessage, out);
         continue;
      }

      try {
         int result = calc(lhs, rhs, op);
         if(result >= 100000000) {
            // result has more than 8 digits
            printLarge(errorMessage, out);
            continue;
         }
         printLarge(result, out);
      } catch(DivisionByZeroException const &e) {
         printLarge(errorMessage, out);
      } catch(InvalidOperatorException const &e) {
         printLarge(errorMessage, out);
      }
   }
}
```

2. Grundlagen

2.1. Merkmale

- C++ ist eine Multiparadigmen-Programmiersprache.
- C++ hat keine Methoden, sondern Funktionen, da eine Funktion nicht zwingend zu einem Objekt gehören muss. Gehört sie zu einem Objekt, handelt es sich um eine Memberfunktion.
- Das Schreiben von eigenen Loops und Containern sollte mit Hilfe der STL vermieden werden.
- C++ ist rückwärtskompatibel mit C.
- Es gibt keine Garbage Collection!
- Mit einer Library können Funktionalitäten für andere Programme zur Verfügung gestellt werden.
- Grundsätzlich gilt: **Declare before Use**

Listing 5: Hello World

```
1 #include <iostream>
2
3 int main() {
4    std::cout << "Hello World\n";
5    return 0; // obsolet, da es die Main Funktion ist
6 }
7
8 // or
9 void main(int argc, char *argv[]) { .. }
```

2.2. Terminologie

Value 42

Type `int`

Variable `int const i{42}`

Expression $(2 + 4) * 7$

Statement `while(true);`

Declaration `int foo();`

Definition `int j;`

Function `void bar(){}`

3. Kompilation

C++ kompiliert direkt in Maschiencode, was den Vorteil hat, dass der Overhead einer virtuellen Maschine wegfällt. Dafür läuft C++ Code immer nur auf dem System, für welches es kompiliert wurde.

3.1. Files

*.cpp Source File

*.h Header File

Deklarationen und Definition die in anderen Files verwendet werden können. Header Files können mit `#include "file.h"` included werden. Der Präprozessor kopiert dann das Header File in den Source Code.

Listing 6: Basic Header File

```
1  #ifndef SAYHELLO_H_
2  #define SAYHELLO_H_
3
4  #include <iosfwd>
5
6  namespace n {
7
8     void sayHello(std::ostream&);
9  }
10
11 #endif /* SAYHELLO_H_ */
```

3.2. Präprozessor

Der Präprozerssor sucht nach `include` Anweisungen und fügt den Inhalt der referenzierten Header File ein. Es resultiert ein *.i Datei.

3.3. Kompiler

Der Kompiler nimmt die *.i Datei und übersetzt diese in eine *.o Datei die aus Maschinencode besteht.

3.4. Linker

Der Linker fügt mehrere *.o Dateien zu einer ausführbaren *.exe Datei zusammen.

4. Grundlagen

4.1. Operantoren

Für die Operatoren muss immer die Präzedenz beachtet werden. Grundsätzlich gilt bei gleicher Präzedenz immer von links nach rechts.

Priorität	Operator	Beschreibung	Assoziativität
1	`::`	Bereichsauflösung	von links nach rechts
2	`++ --`	Suffix-/Postfix-Inkrement und -Dekrement	
	`()`	Funktionsaufruf	
	`[]`	Arrayindizierung	
	`.`	Elementselektion einer Referenz	
	`->`	Elementselektion eines Zeigers	
3	`++ --`	Präfix-Inkrement und -Dekrement	von rechts nach links
	`+ -`	unäres Plus und unäres Minus	
	`! ~`	logisches NOT und bitweises NOT	
	`(type)`	Typkonvertierung	
	`*`	Dereferenzierung	
	`&`	Adresse von	
	`sizeof`	Typ-/Objektgröße	
	`new, new[]`	Reservierung Dynamischen Speichers	
	`delete, delete[]`	Freigabe Dynamischen Speichers	
4	`.* ->*`	Zeiger-auf-Element	von links nach rechts
5	`* / %`	Multiplikation, Division und Rest	
6	`+ -`	Addition und Subtraktion	
7	`<< >>`	bitweise Rechts- und Linksverschiebung	
8	`< <=`	kleiner-als und kleiner-gleich	
	`> >=`	größer-als und größer-gleich	
9	`== !=`	gleich und ungleich	
10	`&`	bitweises AND	
11	`^`	bitweises XOR (entweder-oder)	
12	`|`	bitweises OR (ein oder beide)	
13	`&&`	logisches AND	
14	`||`	logisches OR	
15	`?:`	bedingte Zuweisung	von rechts nach links
	`=`	einfache Zuweisung (automatische Unterstützung ist in C++-Klassen Vorgabe)	
	`+= -=`	Zuweisung nach Addition/Subtraktion	
	`*= /= %=`	Zuweisung nach Multiplikation, Division, und Rest	
	`<<= >>=`	Zuweisung nach Links- bzw. Rechtsverschiebung	
	`&= ^= |=`	Zuweisung nach bitweisem AND, XOR, und OR	
16	`throw`	Ausnahme werfen	
17	`,`	Komma (Sequenzoperator)	von links nach rechts

Abbildung 3: Operatoren und deren Präzedenz

4.1.1. Pre- und Postinkrement

Postinkrement, Postdekrement `x++, x--` = Lesen, dann ändern

Preinkrement, Predekrement `++x, --x` = Ändern, dann lesen

4.2. Datentypen

- In C++ sind alles **Wertetypen**
- Im Gegensatz zu Java ist der Speicherverbrauch von Datentypen nicht fix, sondern Prozessorabhängig
- Short und Int sind minimal 16Bit gross, Long minimal 32 Bit und long long minimal 64 Bit.
- Es gibt keine unsigned Fliesskommatypen
- Es gilt 1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long).
- und float <= double <= long double
- Die konvertierung der Datentypen geschieht erst nach der Operation auf der rechten Seite der =.

Wertetype	Beispiele und Literale	Beschreibung
void		der leere Typ
bool	false = 0 und true= n $\neq$ 0	Wird als Integer interpretiert (default = 1)
char / unsigned char	`'a'`, `'\n'`, `'\x0a'`	Wird als Integer interpretiert
char array	`"ab"`	
short		
int	`1, 052(octal), 0x2A(hex)`	
long	`42L`	
long long	`5LL`	
unsigned short, unsigned int, unsigned long, unsigned long long	`1u, 42ul, 0xFULL`	
float	`0.f`	
double	`.33`	default
long double	`0.33L`	

Tabelle 1: Wertetypen in C++

4.2.1. Signed und Unsigned

Der Wertebereich von Ganzzahlen und Gleitkommazahlen ist implementationsabhängig und kann somit nicht fix angegeben werden. Bei den folgenden angaben ist n = Anzahl Bits im Speicher

signed (Standard)
Mit Vorzeichen: Wertebereicht $(-\frac{2^n}{2})$ bis $(\frac{2^n}{2} - 1)$ (Wie in Java implizit signed)

unsigned Ohne Vorzeichen: Wertebereich 0 bis $2^n - 1$

4.2.2. Ganzzahlen

- C++ übernimmt eine automatische Typ Conversion.
- Bei Ganzzahldivisionen wird der Kommateil einfach abgeschnitten
- Lokale Integer Variablen werden standardmässig mit 0 initialisiert
- Teilen von Ganzzahlen durch 0, resultiert in `undefined behavior` (Ausnahme Modulo)

4.2.3. Gleitkommazahlen

- Es sollte immer Double verwendet werden
- NaN und (+/-)Inf sind gültige Werte für eine Gleitkommazahl
- Double Division durch 0 ergibt immer (+/-) **inf**
- Double Variablen können nicht in Integer gespeichert werden, da sie mehr Speicher benötigen
- Lokale double Variablen werden standardmässig mit 0 initialisiert
- 0.0 / 0 = **NaN**

4.2.4. NaN

- NaN Vergleiche geben immer false
- sqrt(-1) = NaN (Komplexes Resultat = NaN)

4.3. Scopes

1. Global scope
2. Named namespaces (können verschachtelt werden)
3. Anonymous namespace (Vesteckt name vor dem Linker)
4. Class scope (Members)
5. Function scope (Parameter)
6. Block scope (lokale Variablen)
7. Temporaries (Resultate von Ausdrücken)

4.3.1. Shadowing

Es ist möglich Parametervariablen zu überdecken. Dies sollte aber wenn möglich unterlassen werden, da es zu unübersichtlichem Programmcode führt.

4.4. Namespaces

- Namespaces erstellen einen eigenen Scope mit welchem Nameskollision verhindert werden können.
- Namespaces erlaubes es, den gleichne Namen in verschiedenen Namespaces zu verwenden.
- Der Namespace muss mit dem zweifachen Doppelpunkt Prefix verwendet werden
- Globaler namespace wird mit Doppelpunkten gekennzeichnet `::separator, ::read()`
- **`using`** Keyword niemals in Header File verwenden, sondern möchlichst lokal
- Mit **`using`** können der Namespace weggelassen werden.
- Überladene Funktionen sollten immer im gleichen Namespace sein. Bekannte Funktionen wie z.B `min()` sollten in ein eigenen Namespace gepackt werden, da sie ansonsten die `min()` Funktion des Standard Namespace überladen.
- Funktionen werden immer im Namespace der Parameter gesucht. Möchte man das die eigenen Funktion aufrufen, muss der Namespace angegeben werden.
- **`using namespace`** sollte nicht verwendet werden

```
1  namespace demo {
2     void foo(); //1
3     namespace subdemo {
4        void foo(); // 2
5     } // subdemo
6  } // demo
7
8  namespace demo { // extends other namespace 'demo'
9     void bar() {
10       foo(); // 1
11       subdemo::foo(); // 2
12    }
13 }
14
15 void demo::foo() { // definition of 1
16 }
17
18 {
19 // anonymous namespace
20 // functions declared here, can not be called outside this file
21 // for helper functions, helper types, constants: used to hide members
22 }
23
24 using demo::subdemo::foo // single class or function
25 using namespace demo::subdemo // complete namespace
26
27 // example for using
28 using std::string;
29 string s{"no std::"};
```

4.5. Unqualifizierte Namensauflösung / Argument Dependent Lookup

Wenn der Typ des Funktionsargument im gleichen Namespace wie die Funktion ist, kann der Namespace weggelassen werden. Dies ist inbesondere bei den Operatoren wichtig. Das Lookup greift nur bei unqualifizierten Namen (ohne Angabe des Namspaces)

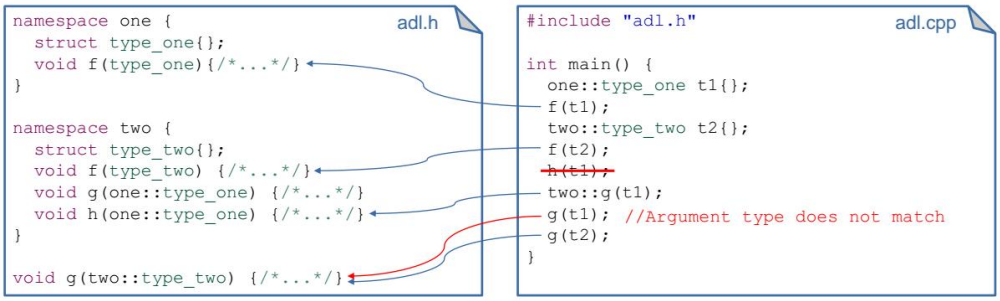

Abbildung 4: Argument Dependent Lookup

4.6. Alias / Using

Mit dem Keyword **using** kann ein Alias gesetzt werden.

```
1 using name=type;
2
3 // example
4 using input=std::istream_iterator<std::string>;
5 input eof{}
```

4.7. Casting

Casting sollte wenn möglich vermieden werden. In wenigen Fällen ist es aber unumgänglich und dann sollte man den **static_cast** der C Variante mit den runden Klammern vorziehen.

```
1 // bad
2 std::abs(index) > (int) this->size())
3
4 // good
5 std::abs(index) > static_cast<int>(this->size())
```

4.8. Kontrollstrukturen

4.8.1. Switch Case

Switch Case funktioniert nur für Ganzzahlen

```
void testSwitchCase(int weekDay) {
   switch (weekDay) {
      case 1:
      break;
      case 2:
      break;
      default:
      break;
   }
}
```

4.8.2. For-Each

```
#include <string>
#include <vector>
#include <algorithm>

#include <iterator>
#include <iostream>

void testForEach(std::vector<std::string> const & items, std::ostream & out) {
   std::for_each(begin(items), end(items), [&out](auto s){
      out << s;
   });
}
int main() {
   std::vector<std::string> items{};
   items = {"abc","de","efg","hijk"};

   testForEach(items, std::cout);

   // implicit
   return 0;
}
```

5. Variablen

- Variablen beginnen immer mit einem Kleinbuchstaben
- Lokale Variablen müssen wie in Java immer mit einem Default Wert deklariert werden! (Geschweifte Klammern oder = bei `auto` Variablen)
- Globale Variablen sollten nie verwendet werden.
- Es sollten keine veränderbaren globale Variablen deklariert werden
- Eine Variable sollte so nahe wie möglich bei dem Ort wo sie benötigt wird, deklariert werden.
- Variablen sind standardmässig Value Types und und werden somit bei der Definition auf dem Stack alloziert.
- Globale, veränderbare Variablen sollten vermieden werden, da sie schwer zu testen sind und beim Multithreading Probleme bereiten.

```
1  <type> <name> { <default-value> }
2
3  int i{42}
4  int &ri{i} // must initialize ref
5  int const &cri{i}; // const alias
6  int const &cr{6*7} // extend lifetime of 6*7
7  ri = 43 // changes i, ri only an lias
8  // --cri; // does not work, because of const
9  int &&rv{3*14}; // extends lifetime of 3*14
10 // int &&rvri{i} impossible
11 int &&rvri{std::move(i)}; // steal i's content
```

5.1. Reference / Value

5.1.1. By Value

- Bei Zuweisungen bei value von vererbten Werten gibt es ein Object Slicing (`x=a`) (Siehe Vererbung)

5.1.2. By Reference

- Variablenzuweisungen `by reference` machen nur bedingt Sinn.

5.2. Const

- Const sollte so oft wie möglich verwendet werden.
- Const ist ähnlich wie das `final` in Java, jedoch mit höherer Garantie dass eine Variable oder Memberfunktion nicht verändert wird
- Konstanten müssen initialisiert werden
- Das Const bei Memberfunktionen bezieht sich auf das this Objekt. Const Funktionen dürfen Inhalt von dem `this` Objekt nicht verändern.
- Um Konstanten zur Compilezeit zu setzten muss man das Keyword `constexpr` verwendet werden.

Merke 5.1: Const

Alle Variablen die nicht verändert werden, müssen const sein!

5.3. Auto

Das Auto Keyword versucht automatisch den Typ zu bestimmen und zwar gemäss der Zuweisung bei der Deklaration.

```
1 auto const m=4; //int
2 auto const s="Hello"s; // std:string
```

5.4. Inline

Wenn ein Member in Header definiert wird, muss die Definition inline sein. Dies sollte nur gemacht werden, wenn der Funktionsbody sehr kurz ist. Eine Inline Function erlaubt dem Compiler den Function Call Overhead weg zu optimieren.

```
1 inline double square(double val) {
2    return val * val;
3 }
```

6. Funktionen

- Funktionen werden im lower Camel Case geschrieben
- Eine Funktion muss immer zuerst im Header File deklariert werden, bevor man sie verwenden kann
- Eine Funktion hat einen guten Namen, maximal 5 Parameter (besser 3) und erledigt genau eine Sache.
- Bei den Funktionsparameter ist die Aufrufreihenfolge nicht definiert (z.B Wenn der Funktionsparameter der Rückgabewert einer Funktion ist)
- Die Main Funktion gibt implizit 0 zurück
- Auto sollte nicht als Return Typ angegeben werden. Ausnahme sind Inline, Template oder Constexpr Funktionen in Header Files.
- Void sollte nicht als Funktionsparameter verwendet werden.

```
1 // result type can be auto, but should not
2 result_type function_name(parameter_type parameter) {
3    return return_variable;
4 }
```

6.1. Default Arguments

Default werde sollte erst bei der Deklaration gesetzt werden. Die Parameter mit einem Default werde, müssen nicht übergeben werden.

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

6.2. Reference / Value

6.2.1. Call by Value

- Parameter werden standardmässig als Wert (value) übergeben
- Es wird dabei eine Kopie des Parameters erstellt (Achtung: Grosse Objekte) → besser Refenz verwenden.
- `ret_type f(type param){ .. }`

6.2.2. Call by Reference

- Referenzen machen nur als Parameterübergabe Sinn, nicht aber bei Zuweisungen von lokalen Variablen.
- Referenz Parameter werden mit dem & Prefix markiert (lvalue)
- RValue Referenzen werden mit dem && Prefix markiert. RValue Referenzen verschieben den Inhalt
- Referenzen können `const` sein. Speziell für grosse Objekte die nicht verändert werden.

- Niemals eine Referenz auf eine lokale Variable zurückgeben, da das Stackframe nach dem Methodenaufruf abgebaut wird! (Dangling Reference). `int & my_func(){ int n{42} return n;}`
- Das Orginal muss mindestens so lange leben wie die Referenz darauf.
- Referenzen könnnen zurückgegeben werden. `int my_func(int &n){ return n; }`

```
1 void askForName(std::ostream &out)
```

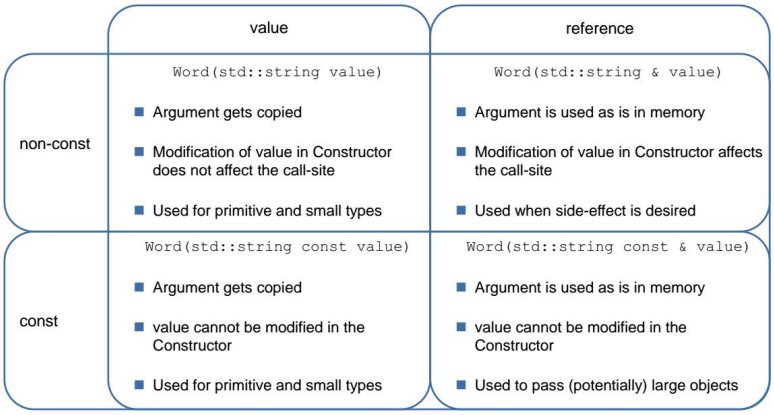

Abbildung 5: Constructor und Function Parameter

6.3. Function Overloading

Unter Function Overloading versteht man, dass mehrere Funktionen den selben Namen aber unterschiedliche Parameter haben. (Unterschiedliche Signatur)

```
1 void incr(int& var);
2 int incr(int& var); // does not compile
3 void incr(int& var, unsigned delta);
```

6.4. Template Function

- Funktions Templates werden normalerweise direkt im Header deklariert und implementiert
- Template Funktionen sind implizit inline
- Mit Templates können generische Funktionen geschrieben werden, damit es keine Probleme mit überladenen Funktionen gibt (mehrdeutigkeiten)
- Bei Mehrdeutigkeiten muss der Typ in den Spitzen-Klammern übergeben werden.
- Das `typename` Keyword kann mit durch das Keyword `class` ersetzt werden. (deprecated)

```cpp
template <typename T, typenmae U, etc.>

namespace CustomMin {
   // could be ambiguous
   inline int min(int a, int b){
      return (a < b)? a : b ;
   }
   inline double min(double a, double b){
      return (a < b)? a : b ;
   }

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

// call
min(1, 1);
min(2, 2)
min<doubl>(1, 2.0); // ambiguos without template

// char array mit unterschiedlichen laengen muessen explizit als string verglichen
    werden
MyMin::min<std::string>("Test", "Test2"); // std namespace already has a min function
MyMin::min("Test"s, Test2"s); // alternative variante
```

6.4.1. Variable Parameter

Wie in Java werden die drei Punkte für beliebig viele Parameter benutzt.

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

// example
template<typename Head, typename... Tail>
   void println(std::ostream &out, Head const& head, Tail const& ...tail) {
   out << head;
   if (sizeof...(tail)) {
      out << ", ";
   }
   println(out,tail...); //recurse on tail
}
// base case
void println(std::ostream &out) {
   out << "\n";
}
```

6.5. Funktionsparameter

Alle Funktionen und Lamdas können als Parameter eine weiteren Funktion übergeben werden. Funktionen, Funktoren und Lamdas können in einem `std::function<x(y)>` Objekt gespeichert werden.

```

// std::function definiert den Rueckgabe und Parameter Typ
std::function<bool(int)> apredicate{};

// member function pointer, wobei calc eine Methode einer Klasse ist
std::function<int (X const &,int)> const f{&X::calc};


void printfunc(double x, double f(double)){
   std::cout<< f(x);
}

double square(double d) {
   return d*d;
}

printfunct(1.0, std::atan);
printfunct(1.0, square);
printfunct(2.0, [](double x){return x*x;});
```

6.6. Lamdas

- Lamdas können auch in Variablen geschrieben werden `auto l = []{}; l();`
- Das kleinste Lamda wäre `[]{}()`, wobei die ersten beiden Klammern das Funktionsobjekt sind und die runde Klammer der Aufruf.
- Der `return_type` kann auch weggelassen wenn dieser **`void`** oder konsistent ist.
- Das Lamda Capture kann verwendet werden, wenn das Lamda als Funktionsparameter übergeben wird und man trotzdem noch Variablen aus dem lokalen Context verwendetn möchte.
- Ein Predicate nimmt ein oder mehrere Parameter entgegen und gibt **`true`** oder **`false`** zurück.

```
// return_type is optional
auto const l = [lamda_capture](parameters) -> return_type {
   statements;
};
// call
l(param);

// lamda campture
#include <functional>
void f(std::function<char(char)> function) {
   std::cout << function('a');
}
void main() {
   unsigned i{1};
   auto const g = [i](char c) -> {
      return std::toupper(c) + i;
   }
}
```

6.6.1. Captures

Es ist guter Stil explizit zu capturen. Es sind auch Kombination von Caputures möglich

- [=] - 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
- $[=, \&out]$ - capture all by copy, but out by reference
- $[\&, = x]$ - capture all by reference, but x by copy/value

Gecapturete Variablen sind standardmässig nicht veränderbar. Man muss sie also entweder **mutable** definieren und per Referenz übergeben:

```
1
2  std::vector<int> v;
3  int x{}; // memory for lambda below
4
5  // mutable
6  generate_n(std::back_inserter(v),10,[x=0]() mutable {
7
8  // reference
9  generate_n(std::back_inserter(v),10,[&x]{
10 ++x; return x*x;
11 });
12
13 class make_squares{
14 int x{};
15 public:
16 int operator()() { ++x; return x*x; }
17 };
18
19 //...
20 generate(v.begin(),v.end(),make_squares{});
```

6.6.2. Parameter

- Die Parameter können auch als **auto** deklariert, wenn klar ist, welcher Typ übergeben wird.

6.7. Functor

Functoren sind Typen die eine bestimmte Operation zur Verfügung stellen. Funktoren haben den call Operator überladen haben. Lamdas arbeiten intern mit Functoren. Die `operator()` Funktion kann theoretisch beliebig oft überladen werden.

```
struct Accumulator {
   int count{0};
   int accumulated_value{0};

   // two parentheses required
   // Wenn die Funktion keine Memeber veraendert, sollte sie const sein!
   void operator()(int value) {
      count++;
      accumulated_value += value;
   }
   int average() const;
   int sum() const;
};

// Use of Accumulator. Return Average of accumalted Sum
int average(std::vector<int> values) {
   Accumulator acc{};
   for(auto v : values) { acc(v); }
   return acc.average();

   // rsp. mit einem Algorithmus
   return std::for_each(begin(values), end(values), acc).average();
}
```

6.8. Standard Funktoren

Häufig gebrauchte Funktoren werden von der Functional Libary zur Verfügung gestellt. Associative Container Typen können die Standardfunktion zur Initialisierung übergeben werden.

```
#include <functional>
// arithmetic
plus<>{}
minus<>{}
divides<>{}
multiplies<>{}
modules<>{}
logical_and<>{}
logical_or<>{}
// unary
negate<>{}
logical_not<>{}
// binary
less<>{}
less_equal<>{}
equal_to<>{}
greater_equal<>{}
not_equal_to<>{}

// example
transform(v.begin(), v.end(), v.begin(), v.begin(), std::multiplies<>{})

// sort set in reverse order
std::set<int, std::greater<>> reverse_int_set{};
```

7. Klassen

Klassen und Structs werden immer in Header Files definiert. Die Implementierung kann dann in einem beliebigen File stattfinden.

- Bei einer class sind alle Member implizit private
- Klassenmember sind implizit **inline**!
- Nach der Klassen Definition muss ein Semikolon stehen!
- Membervariablen sollten nie für die Kommunikation zwischen Memberfunktionen missbraucht werden.
- Zugriff auf die aktuelle Instanz ist via **this**->member rsp. **this**->memberfunction()
- Das **this** Objekt ist ein Pointer und muss nur verwendet werden, wenn der Name des Membervariable gleich einer lokalen Variable ist.
- Wenn man das wirkliche **this** Objekt übergeben will, muss der Pointer mittels ∗**this** dereferenziert werden.
- Memberfuntionen sollten wenn möglich **const** sein, solange sie das **this** Objekt nicht verändern.

Listing 7: A good Class

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

Listing 8: Klassentyp im Header File

```
#ifndef DATE_H_
#define DATE_H_
class Date {

   int year, month, day;
   bool isValid();

public:
   Date() = default;
   explicit 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_ */
```

Listing 9: Implementierung des Klasse

```
#include "Date.h"
Date::Date(int year, int month, int day)
   : year{year}, month{month}, day{day} {

      if (!isValidDate()) {
         throw std::out-of_range{"invalid date"};
      }
   }

Date::Date() : Date{1,1,1980} {} // default constructor

Date(Date const & other) : Date{other.day, other.month, other.year} {} // copy
    constructor

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

bool Date::isValidDate() const
{
   // acess member
   ()*this).member
   this -> member
}
```

Listing 10: Verwendung des Klasse

```
#include "Date.h"

void foo() {
   Date today{19.10.2016};

   auto thursday{today.tomorrow()};

   Date::isLeapYear(2016);
}
```

7.1. Struct

- Bei einer Struct sind alle Member implizit public
- Nach dem Kompilieren ist die Struct aber genau gleich wie die Klasse

7.2. Sichtbarkeiten

Die Visibility gilt für einen bestimmten Bereich

private Nur innerhalb der Klasse gültig (and friends)

protected Auch sichtbar in Subklassen

public Sichtbar für alle

7.3. Konstruktoren

- Anstatt der Default Initialisierung können die Member Variablen auch schon im Header File initialisiert werden (NSDMI: Non Static Data Member Initializers)

Listing 11: Konstruktoren

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

// default constructor
<class name> ()
Date::Date() : year{2016}, month{10}, day{26} {}
Date d{}; // call

// copy constructor
<class name> ( <class name> const & )
Date(Date const &)
Date d2{d}; // call

// move constructor (C++ Advanced)
<class name> ( <class name> &&)
Date(Date &&)
Date d2{std::move(d)};

// typeconversion constructor
// explicit avoids implicit conversion!
// const & avoids unnecessary copy
explicit <class name> ( <other type> const & )
explicit Date(std::string const &);
Date d{"19/10/2016"s};

// Examples
Date{16,10,2016};
Date(int day, int month, int day) :
localDay{day}, localMonth{day}, localYear{Year} { /* .. */}

// inherit constructor from base class
using std::set<T, COMPARE>::set;
```

7.4. Defaulted Constructors

Damit der Default Konstruktor im *.cpp File nicht mehr angegeben werden muss, kann er direkt im Header File als Default deklariert werden. Dies ist auch für den Copy und Move Konstruktor möglich. Mit dem **default** Keyword wird ebenfalls der Default Konstrutor redefiniert, wenn nur ein Konstruktor mit Parameter angegeben ist.

Listing 12: Defaulted Constructors

```
#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_ */
```

7.5. Factory Functions

Eine Factory Methode beginnt meist mit `make` oder `create`

Listing 13: Factory Functions

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

7.6. Destruktor

Im Destruktor darf nie eine Exception geworfen werden.

Listing 14: Destruktor

```
   ~Date(); // implizit
```

7.7. Vererbung

Die Art der vererbung kann wieder mit den Sichtbarkeits Modifier gesteuert werden. (private, protected, public)

Listing 15: Konstruktoren

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

class Sub : public Base {

};
```

7.8. Statische Member

- Die **static** Definition ist nur im Header File
- Statische Member verfügen über kein **this** Objekt. D.h sie gehören zu der Klasse und nicht zu deren Instanzen.
- Statische Member (in der Implementierung) können nicht **const** sein. Im Headerfile jedoch schon: **static const int** `zero{0};`
- Können direkt via `<classname>::<member>();` aufgerufen werden

7.9. Klassen Templates

Hinweis 7.1: Prüfung

Klassen Template kommen bestimmt an der Prüfung und geben viel Punkte.

- Template Klassen werden komplett im Header File definiert
- Klassentemplates liefern Typen mit **Compile-time** Parameter
- Man muss `typename` verwenden, wenn das Element direkt oder indirekt vom Template Parameter abhängt.
- Als Template Parameter können Typen (typename), Kostanten oder weitere Templates (template) übergeben werden.
- T als Übergabeparameter darf nicht void sein
- T als Rückgabewert muss kopierbar (oder movebar) sein (da der Vector Wertetypen enthält, keine Referenzen)
- Memberfunktionen die nie benutzt werden, werden bei Template Klasse nie kompiliert.
- Auf vererbten Member sollte immer mit `this->member` oder `[class-name]::member` zugegriffen werden.

7.10. Terminologie

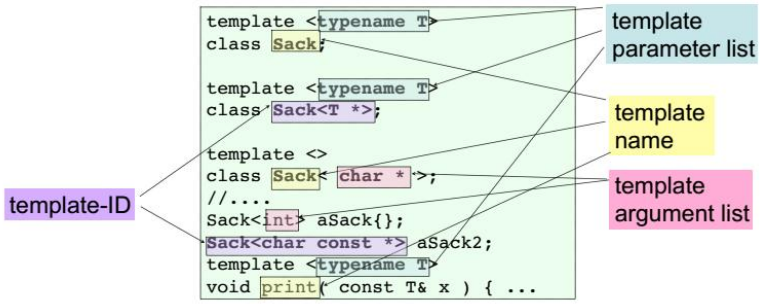

Abbildung 6: Klassen Template Initialisierung

Listing 16: Klassen Templates

```
template <typename T> class [classname];
template <typename T> class [classname]<T *>;
template <> class [classname]<char *>;


template <typename T> class Sack {
   using SackType = std::vector<T>;
   // inherit constructor from base class
   using std::set<T, COMPARE>::set;
   using size_type = typename SackType::size_type;

   SackType theSack{};

public:
   bool emtpy() const {return theSack.empty(); }
   size_type size() const { return theSack.size(); }
   void putInto(T const & item) { theSack.push_back(item); }

   // forward declaration
   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;
}


// specialization
template <typename T> class Sack;

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

7.10.1. Pointer

Pointer sollten nicht verwendet werden.

Listing 17: Klassen Templates Pointer

```
// may the only exception
Sack<char const *> shouldkeepStrings;

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

7.10.2. Konstruktor

Listing 18: Klassen Templates Pointer

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

public:
   Sack()=default;

   template <typename ITER>
      Sack(ITER b, ITER e):theSack(b,e){}
}
```

```
struct B1 {
   B1(int);
};
struct D1 : B1 {
   using B1::B1;
   // The set of candidate inherited constructors is
   // 1. B1(const B1&)
   // 2. B1(B1&&)
   // 3. B1(int)

   // D1 has the following constructors:
   // 1. D1()
   // 2. D1(const D1&)
   // 3. D1(D1&&)
   // 4. D1(int) <- inherited
};

struct B2 {
   B2(int = 13, int = 42);
};
struct D2 : B2 {
   using B2::B2;
   // The set of candidate inherited constructors is
   // 1. B2(const B2&)
   // 2. B2(B2&&)
   // 3. B2(int = 13, int = 42)
   // 4. B2(int = 13)
   // 5. B2()

   // D2 has the following constructors:
```

```
31     // 1. D2()
32     // 2. D2(const D2&)
33     // 3. D2(D2&&)
34     // 4. D2(int, int) <- inherited
35     // 5. D2(int) <- inherited
36 };
37 
38 \subsection{Template Template Parameter}
39 \begin{lstlisting}[language=C++, caption=Klassen Templates Pointer]
40 template <typename T, template<typename...> class container=std::vector> class Sack {
41 
42 }
43 
44 Sack<int,std::list> listsack{1,2,3,4,5};
```

8. Enums

- Enumerationen werden verwendet für Typen die nur wenige Werte halten.
- Jedes Enum Feld kann leicht in einen int konvertiert werden, beginnend bei 0. Die int Werte können auch manuell mit = beliebig zugewiesen werden.
- Mit dem **class** Keyword (Scoped Enum) ist der Enum Typ nicht ausserhalb des Namespaces sichtbar. Beim normalen unscoped Enum jedoch schon
- Die Namen des Enums können standardmässig nicht ausgegeben werden. Dazu muss man eine Lookuptable mit einem String Array anlegen.

Listing 19: Enum

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

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

   //operator overload
   day_of_week operator++(day_of_week & aDay) {
      int day = (aDay + 1) % (Sun + 1); // convert to int
      aDay = static_cast<day_of_week>(day);
      return aDay;
   }
};

// ex2: specify type with inheritance
enum class launch_policy : unsigned char {
   sync=1, async=2, gpu=4, process=8, none=0
};

// v3
enum Color { red, green, blue };
Color r = red;
switch(r) {
   case red : std::cout << "red\n"; break;
   case green: std::cout << "green\n"; break;
   case blue : std::cout << "blue\n"; break;
}
```

9. Operator Overloading

- Operatoren können für Klassen, Structs und Enums überladen werden

Listing 20: Mögliche Operatoren zum Überladen

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

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

// non overloadable operators
:: .* . ?
```

Listing 21: Operator Overloading

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

#include <iostream>
class Date {
   int year, month, day; // private
   public:
      // Memberfunktionen
      std::ostream & print(std::ostream & os) const {
         os << year << "/" << month << "/" << day;

         // return stream obj, to allow output to chain values
         return os;
      }

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

```
};

// Inline Funktionen die die Member aufrufen
inline std::ostream & operator<<(std::ostream & os, Date const & date){
   return date.print(os);
}

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

// simplyfied solution with tuples
#include "Date.h"
#include <tuple>
bool Date::operator<(Date const & rhs) const {
   return std::tie(year, month, day) < std::tie(rhs.year, rhs.month, rhs.day);
}
```

9.1. Pre- Postfix Incrementation Overload

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

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

// Postfix (Int is obsolete, just for signature diversity)
dayOfWeek operator++(dayOfWeek & aday, int) {
   dayOfWeek ret{aday};
   if (aday == Sun) {
      aday = Mon;
   } else {
      aday = static_cast<dayOfWeek>(aday + 1);
      return ret;
   }
}
```

9.1.1. Grösser / Kleiner als / gleich Overload

Wörter vergleichen

```
// word.h
class Word {
private:
   std::string word;

public:
   Word() = default;
   Word(std::string word);

   std::ostream & print(std::ostream & os) const;
   std::istream & read(std::istream & is);

   bool operator<(Word const & rhs) const;
};

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

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

inline bool operator>(Word const & lhs, Word const & rhs) {
   return rhs < lhs;
}

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

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

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

inline bool operator!=(Word const & lhs, Word const & rhs) {
   return !(lhs == rhs);
}

// word.cpp
bool Word::operator<(Word const & rhs) const {
   return std::lexicographical_compare(word.begin(), word.end(), rhs.word.begin()
       ,rhs.word.end(), [](char lhs, char rhs) {
      return std::tolower(lhs)<std::tolower(rhs);
   });
}
```

10. Fehlerbehandlung

Es gibt fünf Möglichkeiten der Fehlerbehandlung

1. Fehler ignorieren wobei undefined behaviour möglich ist (z.B vector[0])
2. Default Wert retournieren (z.B Beim Einlesen des Hostnamen ”localhost”)
3. Fehlercode oder Fehlerwert zurückgeben (`bool` oder -1 (falls Wert nur positiv))
4. Statusbit setzen (Streams good(), fail(), bad())
5. Exception werfen

10.1. Exceptions

- `std::exception` ist die Basisklasse aller Exceptions
- In der Memberfunktion `what()` steht die Fehlerbeschreibung
- Throw by value, catch by const reference!
- Es könnten beliebige Werte Typen (**value** types) geworfen werden
- Das Keyword `noexcept` garantiert, dass eine Memberfunktion nie eine Exception wirft. Tut sie dies doch, terminiert das Programm.
- Schreibt man eigene Exceptions, kann der Ausdruck ”Exception” im Namen der Exception weggelassen werden

```
#include <stdexcept> // contains some subclass of std::exception (std::logic_error or
    std::invalid_argument)

throw std::logic_error("message");

try {
..
} catch ( std::logic_error const & e) {
e.what();
throw; // re-throw
} catch ( ... ) {
// catchs all exceptions, because there is no specific super type
}

int minimum (int a, int b) noexcept {
   return a<b?a:b;
}
```

11. Streams IO

- In den Header Files sollte die Vorwärtsdeklaration `#include <iosfwd>` verwendet werden. In den Source Files für `std::cin` und `std::cout` `#include <iostream>` und falls nur einer der beiden Stream Objekte benötigt wird, `#include <istream>` rsp. `#include <ostream>`. Die letzten beiden Includes beinhalten kein `std::cin` und `std::cout`.
- Innerhalb von Main verwendet man std::cin und std::cout und die jeweiligen Shift Operatoren >> und <<
- std::istream Objekte geben false zurück, falls sie sich in einem invaliden Status befinden.
- std::endl flushed gepufferten Out Stream. Man sollten deshalb besser das einfach `'\n'` verwenden.

11.1. Bits

Auf der linken Seite der Tabelle sind alle möglichen Bit zustände aufgelistet. Auf der rechten Seite stehen die Rückgabewerte der einzelnen Abfragefunktionen.

eofbit	failbit	badbit	good()	fail()	bad()	eof()	operator bool	operator!
false	false	false	true	false	false	false	true	false
false	false	true	false	true	true	false	false	true
false	true	false	false	true	false	false	false	true
false	true	true	false	true	true	false	false	true
true	false	false	false	false	false	true	true	false
true	false	true	false	true	true	true	false	true
true	true	false	false	true	false	true	false	true
true	true	true	false	true	true	true	false	true

Abbildung 7: Stream Bits

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

Abbildung 8: Input Stream State Bits

11.2. Input

Listing 22: Simple Input Stream

```
#include <istream>

int readInteger(std::istream& in) {
   while(in) {
      int mynumber{-1};
      if (in >> mynumber) {
         return mynumber;
      }
      in.clear(); // renive fail flag
      in.ignore(); // ignore one char
   }
   return -1;
}

int readString(std::istream & in) {
   while(in) {
      std::string line{};
      std::getline(in, line);
      std::istringstream is{line};
   }

   while (is.peek() != EOF) {
      if (std::isalpha(is.peek())) {
         word.push_back(is.get());
      } else {
         break;
      }
   }
}
```

11.3. Output

Listing 23: Formatting Output

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

std::cout << std::[oct|hex|dec] << 24 << '\t'
std::cout << std::setw(10) << 24 // set width of next output line

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

11.4. Manipulatoren

Für die Formatierung der Ausgabe kann man einer breiten Auswahl an Manipulatoren gebrauch machen.

Manipulator	Input / Output	Beschreibung
`std::dec`	beides	sets integer format to decimal
`std::oct`	beides	sets integer format to octal
`std::hex`	beides	sets integer format to hexadecimal
`std::showpos`	beides	show plus sign for positive numbers
`std::noshowpos`	beides	do not show plus sign
`std::showbase`	beides	sets integer format to show base
`std::noshowbase`	beides	sets integer format to not show base
`std::uppercase`	beides	use upper case for hex digits, exponent etc
`std::nouppercase`	beides	use lower case for hex digits, exponent etc
`std::boolalpha`	beides	use ”true” and ”false” for bool
`std::noboolalpha`	beides	use 1 and 0 for bool i/o
`std::setw(n)`	beides	sets next field width to n, non sticky
`std::setfill(c)`	output	sets output field fill character
`std::left`	output	output towards the left in a wider field, fill character on the right
`std::right`	output	output towards the right in a wider field, fill character on the left
`std::internal`	output	fill character between sign/base and number
`std::scientific`	beides	sets floating point format with exponent
`std::fixed`	beides	sets fixed floating point format without exponent
`std::defaultfloat`	beides	sets default floating point format
`std::setprecision(n)`	beides	sets significant or fractional number of digits
`std::showpoint`	output	include floating point always
`std::noshowpoint`	output	omit floating point if possible
`std::flush`	output	flush buffered output
`std::endl`	output	output newline and flush buffered output
`std::ends`	output	output `'0'` and flush buffered output

Tabelle 2: Wertetypen in C++

12. Container und Collections

Container Hält die Objekte bei Value (vector, string, set, map)

Collection Hält die Objekte by Reference

- Container können einfach kopiert werden, in dem man den Ursprungscontainer dem Konstruktor übergibt (deep Copy) `std::vector<in> vv{v};`
- Container unterstützen die `clear()` Funktion, welche den Container als `empty()` markiert.
- Es gibt Sequence Container (String, Vector, Array, Deque, List, Stack), Associative Container (Set, Map) und Hashed Container (Unordered Set, Unordered Map)
- Bei Container sollten die Memberfunktion den Standardalgorithmen vorgezogen werden.
- Zwei Container vom selben Typ könne mit `c1 == c2` auf ihren Inhalt verglichen werden.

12.1. Strings

- (std::string = `"mystring"s`) ≠ `"ab"` (char array)
- std::string ist ein Subtyp von std:basic_string<char>
- Raw Strings: `R"(a raw string)"` (zwischen dem Gänsefüsschen und (bzw.) kann eine beliebiger Trennsstring eingefügt werden).
- Strings bauen auf Chars aus, und sind daher ASCII (und nicht wie im Java Unicode).

```
1 std::string my_string{"5"};
2 int i = stoi(my_string)
```

12.2. Array

Beim Array muss immer noch die Länge mitgegeben werden. Arrays sind Container mit einer fixen grösse und sollten den C-style Arrays vorgezogen werden.

```
1 std::array<int,6> a{{1,1,2,3,5,8}};
```

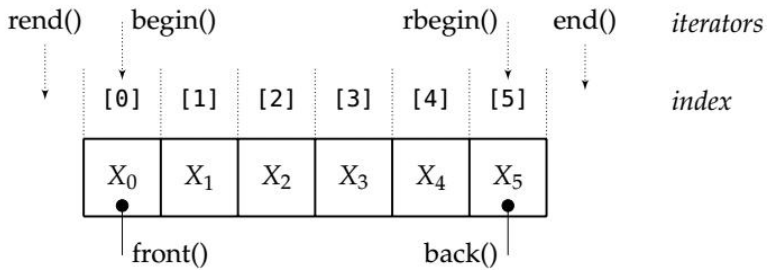

Abbildung 9: Array

12.3. Vector

- std:vector<T> ist Sequenz von Wert des Types T (Vergleichbar mit einer ArrayList in Java)
- Die Elemente des Vectors werden auf dem Heap alloziert (bei Elementen auf dem Stack wird eine Kopie auf den Heap angelegt).
- Wird ein Vector einer Funktion übergeben, kann man mit dem Keyword **const** verhindern, dass eine Kopie eines sehr grossen Vectors angelegt wird. Wird vom Vector nur gelesen, muss dies in jedem Fall gemacht werden!
- Wird der Vector mit runden Klammern initialisiert, alloziert der Compiler die übergeben Anzahl an Elemente. Wenn es geschweifte Klammern sind, wird einfach ein Element mit dem übergebenen Wert übergeben.

```
1  template<class T, class Allocator = std::allocator<T>> class vector;
2
3  #include <vector>
4  #include <algorithm>
5  #include <iostream>
6  #include <iterator>
7
8  // declaration
9  std::vector<int> v{1,2,3,4,5} // initialize with 5 elements
10 std::vector<int> v(6); // alocate 6 elements
11 std::vector<int> v(6, -1); // allocate with default value
12 std::vector<std::string> v(6, "hello"); // allocate with default value
13
14 // 2d matrix
15 std::vector<std::vector<std::string>> const matrix { {}, {}, {} };
16 matrix[i][j];
17
18 // add element
19 v.push_back(val); // add at the tail
20 v.insert(end(v), val); // add at the tail
21 v.insert(begin(v), val); // add at the front
22
23 // remove elements (1,2,3,4,5)
24 v.pop_back(); // 1,2,3,4
25 v.erase(begin(v)); // 2,3,4
26 v.erase(end(v) + 1, v.end()); // 2
27 v.clear(); //
28
29 // removes the element, but keeps space in vector. (undefined value) erase does also
       decrease vector size.
30 remove(begin(v), end(v), element);
31
32 // Find
33 auto found = find(begin(v), end(v), 42);
34 if (found != end(v)) {
35    out << *found;
36 }
37
38 // Loop
39 void outputIndex(std::ostream & out, std::vector<int> const & v) {
40    for (auto const i: v) {
41       out << i << ", ";
42    }
```

```
43      out << "\n";
44  }
45
46  for(auto &e:v) {
47      e *= 2; // change element
48  }
49
50  // create vector from stream
51  using input = std::istream_iterator<int>;
52  input eof{};
53  std::vector<int> const v{input{std::cin}, eof};
54
55  // fill with input
56  std::vector<int> v2{};
57  std::copy(input{std::cin}, eof, back_inserter{v}); // uses push_back
58
59  // fill vector
60  std::vector<int> v(size,default_value);
61  // or
62  std::vector<int> v(10); fill(begin(v), end(v), default_value);
63
64  // fill with computed values (1,2,3,4,5...100)
65  std::vector<int> v(100);
66  iota(begin(v),end(v), 1);
```

12.3.1. Memberfunktionen

front() Zugriff auf das erste Element

back() Zugriff auf das letzte Element

insert(index, element) Einfügen an einer bestimmten Stelle

at(index) Element beim Index (Out of Bound Check im Gegensatz zu v[index])

size() Länge des Vectors (Variable vom Typ `size_t`)

12.4. Double-linked List

Die Double-linked List erlaubt effizientes einfügen Es gibt auch noch eine Singly-linked List. Diesen sollte man aber nicht verwenden.

```
1  std::list<> l(<x-times>, <value>)
```

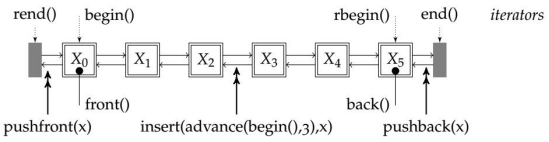

Abbildung 10: Double-linked List

12.5. Double-ended Queue, Deque

Ist wie ein Vector, Objekte können aber zusätzlich effizient am Anfang der Queue eingefügt und entfernt werden.

```
1 std::deque<int> q{begin(v),end(v)};
2 q.pop_front();
3 q.push_front(42);
4 q.push_back(42);
5 q.pop_back();
```

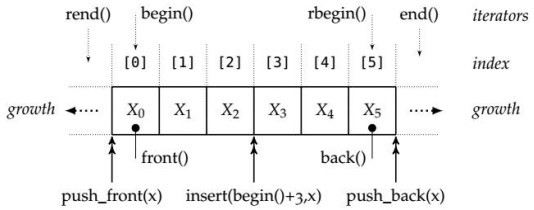

Abbildung 11: Double-ended Queue

12.6. Queue, FIFO Adapter

Im Gegensatz zum Stack nimmt `pop()` die Element am Anfang der Queue raus.

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

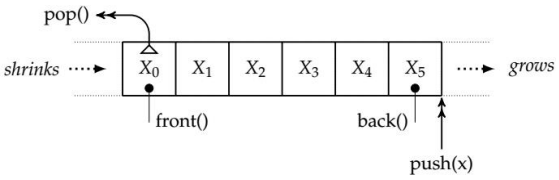

Abbildung 12: FIFO queue

12.7. Stack, LIFO Adapter

```
std::stack<int> s{};
s.push(42); std::cout << s.top(); s.pop();
```

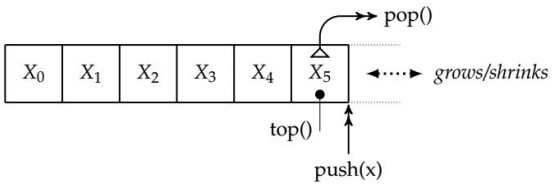

Abbildung 13: Stack

12.8. Set

Ein Set speichert die Elemente immer sortiert in einem Baum. Es gibt keine Duplikate!

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

#include <set>
#include <string>
#include <algorithm> -> transform
#include <iostream>  -> cout
#include <iterator>  -> ostream_iterator
#include <cctype>    -> lowercase

// insert
std::string const input("teststring");
std::set<char> myset { };
std::transform(input.begin(), input.end(), inserter(myset, myset.begin()),
   [](char c) {
      return tolower(c);
   });

// print
std::ostream_iterator<char> out{std::cout};
std::copy(myset.begin(), myset.end(), out);
```

12.9. Multiset

Ein Multiset erlaubt Duplikate

```
#include <iostream>
#include <iterator>
#include <string>
#include <set>

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"));
```

12.10. Map

Wie beim Set werden die Key-Value paare sortiert gespeichert.

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

// Increment Value of Key
++vowels['a'];

// Beim Iterieren ist jedes Element ein pair<char, size>
for(auto const &p:vowels) {
   std::cout << p.first << " = "<< p.second << '\n';
}
```

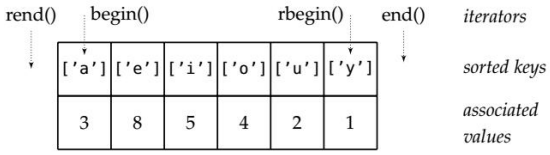

Abbildung 14: Map

12.11. Multimap

Die Multimap erlaubt mehrere Keys

12.12. Hash Container

In C++ muss man sich selber um die Hashfunktion `std::hash<>` kümmern. Die Hashcontainer sind nie sortiert. Es gibt zwei Hash Container:

- Unordered Set
- Unordered Map

13. Iteratoren

Man hat immer zwei Iteratoren (begin() und end()). Man kann auch die Liste von hinten nach vorne durchlaufen (rbegin() und rend()). Werden die Member nur gelesen werden müssen, können auch das **const** Pendant cbegin(), cend(), crbegin() und crend() verwendet werden.

```
// Read only use: use cbegin() und cend()
for (auto it=begin(v); it!=end(v); ++it) {
   // access current element with (*element)
   std::cout << (*it)++ << ", ";
}
```

<table>
<tr><th colspan="3">Iterator category</th><th>Defined operations</th></tr>
<tr><td rowspan="5">RandomAccessIterator</td><td rowspan="4">BidirectionalIterator</td><td rowspan="3">ForwardIterator</td><td>InputIterator</td><td>• lesen
• Inkrement (ohne mehrere Durchgänge)</td></tr>
<tr><td>OutputIterator</td><td>• schreiben
• Inkrement (ohne mehrere Durchgänge)</td></tr>
<tr><td></td><td>• Inkrement (mit mehreren Durchgängen)</td></tr>
<tr><td colspan="2"></td><td>• Dekrement</td></tr>
<tr><td colspan="3"></td><td>• Direktzugriffsspeicher</td></tr>
</table>

Abbildung 15: Iterator Übersicht

13.1. Algorithmen

- `std::advance(iter, n)` überspringt n Iterationen in einem Sprung
- `std::next(iter, n)` macht das Selbe wie advance(), verändert aber den Iter Parameter nicht. (`*next(begin(c), 2)`)
- `std::distance(from, to)` Zählt die Anzahl Element in dem Bereich

13.2. Typen

Es gibt 5 verschiedene Iterator Typen:

```
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 { };
```

13.3. Input Iterator

- Aktuelles Element kann genau einmal gelesen werden, danach muss der Iterator inkrementiert werden.(Ganze Sequenz kann also genau einmal durchlaufen werden)
- Verwendet für `std::istream_iterator` and `std::istreambuf_iterator`

```
struct input_iterator_tag { };
T const operator *()
operator++()
operator++(int)
operator==(myiter) // and !=
```

13.4. Forward Iterator

- Aktuelles Element kann gelesen und verändert werden (ausser Element oder Container ist const)
- Erlaubt nur vorwärts Iteration
- Sequenz kann mehrfach durchlaufen werden

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

13.5. Bidirectional Iterator

- Aktuelles Element kann gelesen und verändert werden (ausser Element oder Container ist const)
- Erlaubt Iteration vorwärts und rückwärts
- Sequenz kann mehrfach durchlaufen werden
- Der Random Acccess Iterator verhält sich gleich wie der Bidirectional Iterator, mit dem Unterschied, dass er den Zugriff mit Index [] erlaubt.

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

13.6. Output Iterator

- Aktuelles Element kann einmal verändert werden, danach muss der Iterator inkrementiert werden
- Es gibt kein definiertes Ende für die Sequenz (Beispiel: Konsole)
- Verwendet für `std::ostream_iterator`
- Schreibt das Resultat ohne über das Ende bescheid zu wissen

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

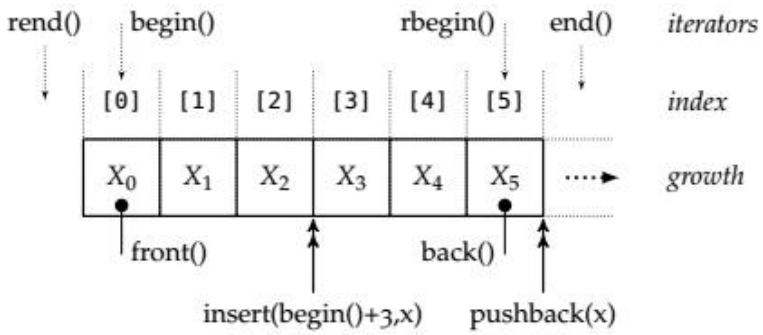

Abbildung 16: Iteratoren

13.6.1. `back_insert_iterator`

`std::back_insert_iterator` ist ein `OutputIterator` welches Elemente an den übergebenen Container mit `push_back()` anhängt.

Oft wird für den `back_insert_iterator` die Hilfsfunktion `std::back_inserter` zur Hilfe genommen:

```
using iiw=std::istream_iterator<Word>;

iiw it { lineIs };
std::vector<Word> words { };
std::copy(iiw(it), iiw(), back_inserter(words));
```

13.7. Loops

Index basierte for-Loops sollte wenn möglich nicht verwendet werden. Erlaubt sind jedoch die folgenden Range Based for-Loops. Diese arbeiten mit den begin() und end() Iteratoren.

```
1  for (auto const e : v) {
2     out << e;
3  }
4
5  for(auto &e:v) {
6     e *= 2; // change element
7  }
8
9  for (auto it=v.begin(); it != v.end(); ++it) {
10    // same as for(auto e:v)
11 }
```

13.7.1. Algorithmen

for_each() Loope über alle Element

```
1     for_each(rbegin(v), rend(v), [&out](auto x){out << x});
```

count(begin, end, value) Zähle das vorkommen des Zeichens `value`

```
1     int linecount(std::istream &in, char c){
2        using it=std::istreambuf_iterator<char>;
3        return count(it{in}, it{}, c);
4     }
```

find(begin, end, value) Such `value` in Container

copy(begin, end, begin_target) Kopiere Inhalt eines Containers in einen anderen

```
// fill vector with content
std::vector<int> v{};
using intiter = std::istream_iterator<int>;
copy(intiter{in}, intiter{}, back_inserter(v));
return v;

// is equal to the directly method
std::vector<int> fill(std::istream &in) {
   using intiter = std::istream_iterator<int>;
   return std::vector<int>{intiter{in}, intiter{}};
}
```

accumulate(begin, end, startValue) Summere auf, beginnend bei `startValue`

distance(begin, end) Anzahl Elemente im Range

```
int charCount(std::istream &in) {
   using it = std::istream_iterator<char>; //Excl whitespaces
   using it = std::istreambuf_iterator<char>; // Incl whitespaces
   // begin() = input iterator, end()= empty iterator, not bound to any stream
   return distance(it { in }, it { });
}
```

advance(it, n) Verschiebt einen Iterator um n Stellen

13.8. Advance

- modifies its argument
- returns nothing
- works on input iterators or better (or bi-directional iterators if a negative distance is given)

13.9. Next

- leaves its argument unmodified
- returns a copy of the argument, advanced by the specified amount
- works on forward iterators or better (or bi-directional iterators if a negative distance is given)

13.10. Spezielle Iteratoren

```
std::ostream_iterator<T>
std::istream_iterator<T>

// Alles kommagetrennt ausgeben (std::cout nur in main benutzen! sonst untestbar)
copy(begin(v), end(v), std::ostream_iterator<int>{std::cout, ", "});

// Whitespace werden nicht uebersprungen
std::istreambuf_iterator<char>;
std::ostreambuf_iterator<char>;

// does not skip whitespaces (use get())
using input=std::istreambuf_iterator<char>;
input eof{};
input in{std::cin};
std::ostream_iterator<char> out{std::cout};
copy(in,eof,out);
```

14. STL Algorithmen

Algorithmen bieten drei klare Vorteile gegenüber handgeschriebene loops:

1. Weniger Fehleranfällig
2. Lesbarer
3. Je nach Implementierung schneller

Listing 24: Algorithmen Headers

```
// filling, finding, checks, transformation
#include <algorithm>

// accumulate, inner_product, partial_sum, adjacent_difference, iota
#include <numeric>
```

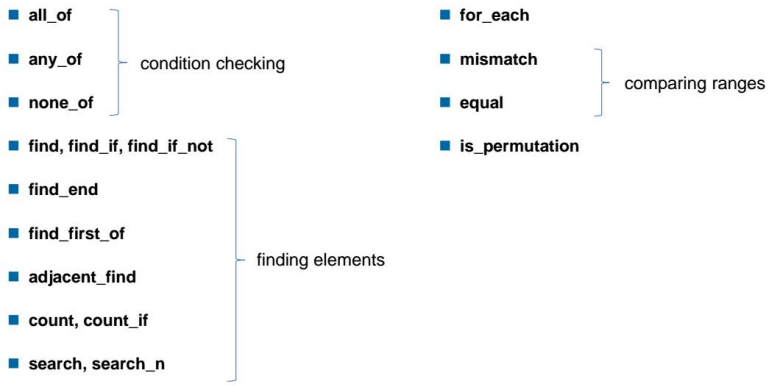

Abbildung 17: Nicht verändernde Algorithmen

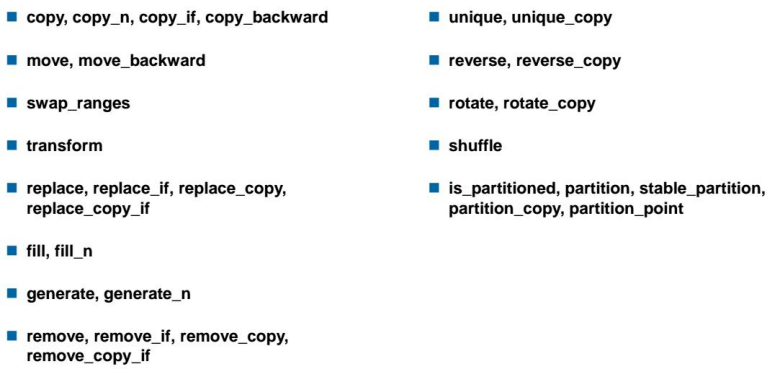

Abbildung 18: Verändernde Algorihtmen

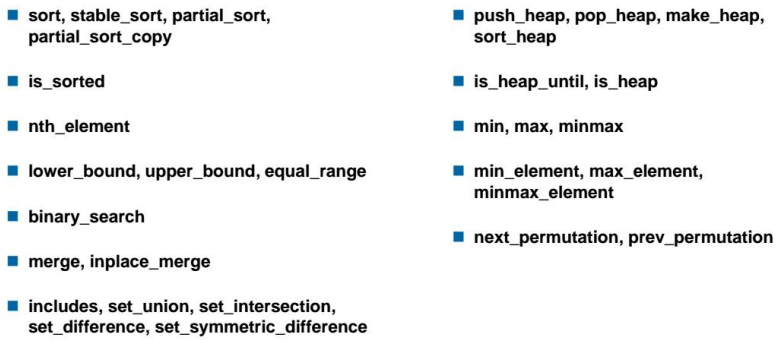

Abbildung 19: Sortierende Algorithmen

14.1. Min und Max Element Algorithm

Listing 25: Max Element Algorithm

```
1  // expected = 1;
2  auto res = std::min({9, 6, 5, 1, 2, 10, 3, 8});
3
4  // expected = 10
5  auto res = std::max({9, 6, 5, 1, 2, 10, 3, 8});
6
7  // expected pair[1,10]
8  auto res = std::minmax({9, 6, 5, 1, 2, 10, 3, 8});
9
10 // expected begin(in1) + 3
11 std::vector<int> in1{9, 6, 5, 1, 2, 10, 3, 8};
12 auto res = std::min_element(std::begin(in1), std::end(in1));
13
14 // expected begin(in1) + 5
15 std::vector<int> in1{9, 6, 5, 1, 2, 10, 3, 8};
16 auto res = std::max_element(std::begin(in1), std::end(in1));
17
18 // expected begin(in1) + 3 , begin(in1) + 5
19 std::vector<int> in1{9, 6, 5, 1, 2, 10, 3, 8};
20 auto res = std::minmax_element(std::begin(in1), std::end(in1));
```

14.2. Checking Algorithm

Listing 26: Property Checking Algorithm

```
1  // Checks that a predicate is false for at least an element of a sequence
2  std::vector<unsigned> in1{2, 3, 5, 6, 7};
3  bool res = std::any_of(
4     std::begin(in1),
5     std::end(in1),
6     is_prime);
7
8  // expected true: Checks whether a permutation of the second sequence is equal to the
       first sequence
9  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
10 std::vector<int> in2{1, 5, 7, 4, 2, 6, 3, 8};
11 auto res = std::is_permutation(
```

```
12     std::begin(in1),
13     std::end(in1),
14     std::begin(in2));
15
16  // expected = std::make_pair(std::begin(in1) + 4, std::begin(in2) + 4);
17  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
18  std::vector<int> in2{1, 2, 3, 4, 0, 6, 7, 8};
19  auto res = std::mismatch(
20     std::begin(in1),
21     std::end(in1),
22     std::begin(in2));
23
24  // Checks that a predicate is true for all the elements of a sequence
25  std::vector<unsigned> in1{2, 3, 5, 6, 7};
26  bool res = std::all_of(
27     std::begin(in1),
28     std::end(in1),
29     is_prime);
30
31  // auto expected = false;
32  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
33  std::vector<int> in2{1, 2, 3, 4, 0, 6, 7, 8};
34  auto res = std::equal(
35     std::begin(in1),
36     std::end(in1),
37     std::begin(in2));
38
39  // expected true
40  std::vector<unsigned> in1{1, 4, 6, 8, 9};
41  bool res = std::none_of(
42     std::begin(in1),
43     std::end(in1),
44     is_prime);
```

14.3. Find Algorithm

Listing 27: Find Algorithm

```
1   find(begin(values), end(values), sought) != end(values));
2
3   // Find the first occurrence of a value in a sequence
4   // expected = std::begin(in1) + 4
5   std::vector<int> in1{1, 2, 3, 4, 5, 6, 7};
6   auto res = std::find(
7      std::begin(in1),
8      std::end(in1),
9      5);
10
11  // std::find_if
12  auto res = std::find_if(
13     std::begin(in1),
14     std::end(in1),
15     is_prime);
16
17  // Find two adjacent values in a sequence that are equal.
18  std::vector<int> in1{5, 6, 4, 7, 7, 2, 2};
19  auto res = std::adjacent_find(
20     std::begin(in1),
21     std::end(in1));
22
```

```
// Find element from a set in a sequence
// expected = std::begin(in1) + 5;
std::vector<int> in1{5, 6, 4, 7, 6, 2, 1};
std::vector<int> in2{1, 2, 3};
auto res = std::find_first_of(
   std::begin(in1),
   std::end(in1),
   std::begin(in2),
   std::end(in2));

// Find the first element in a sequence for which a predicate is true
// expected = std::begin(in1) + 1;
std::vector<int> in1{1, 2, 3, 4, 5, 6, 7};
auto res = std::find_if(
   std::begin(in1),
   std::end(in1),
   is_prime);

// Find two adjacent values in a sequence that are equal
// expected = std::begin(in1) + 3;
std::vector<int> in1{5, 6, 4, 7, 7, 2, 2};
auto res = std::adjacent_find(
   std::begin(in1),
   std::end(in1));

// Find the first element in a sequence for which a predicate is false
// expected = std::end(in1);
std::vector<int> in1{2, 3, 5, 7, 11, 13, 17};
auto res = std::find_if_not(
   std::begin(in1),
   std::end(in1),
   is_prime);

// Find last matching subsequence in a sequence
// expected = std::begin(in1) + 3;
std::vector<int> in1{1, 2, 3, 1, 2, 3, 1};
std::vector<int> in2{1, 2, 3};
auto res = std::find_end (
   std::begin(in1),
   std::end(in1),
   std::begin(in2),
   std::end(in2));
```

14.4. Search Algorithm

Listing 28: Seach Algorithm

```
// Search a sequence for a matching sub-sequence
// expected = std::begin(in1) + 4;
std::vector<int> in1{1, 2, 1, 2, 1, 2, 3, 1, 2, 3};
std::vector<int> in2{1, 2, 3};
auto res = std::search(
   std::begin(in1),
   std::end(in1),
   std::begin(in2),
   std::end(in2));

// Search a sequence for a number of consecutive values
// expected = std::begin(in1) + 5;
std::vector<int> in1{1, 1, 2, 2, 2, 1, 1, 1, 3, 3};
```

```
14    auto res = std::search_n(
15    std::begin(in1),
16    std::end(in1),
17    3,
18    1);
```

14.5. Sort Algorithm

```
1  // Sort the elements of a sequence
2  // expected{1, 2, 3, 4, 5, 6, 7, 8, 9}
3  std::vector<int> in_out1{2, 3, 5, 7, 1, 4, 6, 8, 9};
4  std::sort(
5     std::begin(in_out1),
6     std::end(in_out1));
7
8
9  // Copy the smallest elements of a sequence
10 // expected{1, 2, 3, 4, 5}
11 std::vector<int> in1{2, 5, 3, 7, 1, 4, 6, 8, 9};
12 std::vector<int> out1{0, 0, 0, 0, 0};
13 std::partial_sort_copy(
14    std::begin(in1),
15    std::end(in1),
16    std::begin(out1),
17    std::end(out1));
18
19 // Determines whether the elements of a sequence are sorted
20 std::vector<unsigned> in1{2, 3, 5, 6, 7};
21 bool res = std::is_sorted(
22    std::begin(in1),
23    std::end(in1));
24
25 // Sort the smallest elements of a sequence
26 // expected{1, 2, 3, 4}
27 std::vector<int> in_out1{2, 5, 3, 7, 1, 4, 6, 8, 9};
28 std::partial_sort(
29    std::begin(in_out1),
30    std::begin(in_out1) + 4,
31    std::end(in_out1));
32
33 // Sort the elements of a sequence using a predicate for comparison, preserving the
       relative order of equivalent elements
34 // std::vector<std::pair<int, int>> expected{{1, 0}, {1, 2}, {1, 4}, {2, 1}, {2, 3}};
35 std::vector<std::pair<int, int>> in_out1{{2, 1}, {1, 0}, {1, 2}, {1, 4}, {2, 3}};
36 std::stable_sort(
37    std::begin(in_out1),
38    std::end(in_out1),
39    [](std::pair<int, int> l, std::pair<int, int> r) {return l.first < r.first;});
40
41 // Sort a sequence just enough to find a particular position
42 std::vector<unsigned> in_out1{45, 27, 73, 15, 95, 64, 44, 0, 99};
43 std::nth_element(
44    std::begin(in_out1),
45    std::begin(in_out1) + 3,
46    std::end(in_out1));
```

14.6. Sorted Sequence Algorithms

```
// Merges two sorted ranges in place
// expected{1, 2, 3, 3, 7, 8, 9, 10, 13, 15, 16}
std::vector<int> in_out1{2, 3, 8, 9, 10, 16, 1, 3, 7, 13, 15};
std::vector<int> ;
   std::inplace_merge(
   std::begin(in_out1),
   std::begin(in_out1) + 6,
   std::end(in_out1));

// Determines whether an element exists in a range
std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto res = std::binary_search(
   std::begin(in1),
   std::end(in1),
   7);

// Merges two sorted ranges
// expected{1, 2, 3, 3, 7, 8, 9, 10, 13, 15, 16};
std::vector<int> in1{1, 3, 7, 13, 15};
std::vector<int> in2{2, 3, 8, 9, 10, 16};
std::vector<int> out{};
std::merge(
   std::begin(in1),
   std::end(in1),
   std::begin(in2),
   std::end(in2),
   std::back_inserter(out));


// Finds the largest subrange in which val could be inserted at any place in it
    without changing the ordering
// auto expected = std::make_pair(std::begin(in1) + 3, std::begin(in1) + 6);
std::vector<unsigned> in1{1, 1, 1, 2, 2, 2, 3, 4, 4};
auto res = std::equal_range(
   std::begin(in1),
   std::end(in1),
   2);

// Finds the first position in which @a val could be inserted without changing the
    ordering
std::vector<unsigned> in1{1, 1, 1, 2, 2, 2, 3, 4, 4};
auto expected = std::begin(in1) + 3;
   auto res = std::lower_bound(
   std::begin(in1),
   std::end(in1),
   2);
```

14.7. Count Algorithm

Listing 29: Count Algorithm

```
// Count the number of copies of a value in a sequence
// expected 4 matches
std::vector<int> in1{1, 2, 3, 2, 1, 2, 3, 4, 3, 2};
int res = std::count(
   std::begin(in1),
   std::end(in1),
```

```
7     2);
8
9  // Count the elements of a sequence for which a predicate is true
10 // int expected = 4;
11 std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
12 int res = std::count_if(
13    std::begin(in1),
14    std::end(in1),
15    is_prime);
```

14.8. Set Algorithm

Listing 30: Set Algorithm

```
1  // Return the intersection of two sorted ranges
2  // expected{4, 5, 6, 9}
3  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
4  std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
5  std::vector<int> out{};
6
7  std::set_intersection(
8     std::begin(in1),
9     std::end(in1),
10    std::begin(in2),
11    std::end(in2),
12    std::back_inserter(out));
13
14 // Return the difference of two sorted ranges
15 // expected{1, 2, 3, 7, 8}
16 std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
17 std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
18 std::vector<int> out{};
19
20 std::set_difference(
21    std::begin(in1),
22    std::end(in1),
23    std::begin(in2),
24    std::end(in2),
25    std::back_inserter(out));
26
27 // Return the union of two sorted ranges
28 // expected{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
29 std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
30 std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
31 std::vector<int> out{};
32 std::vector<int>
33
34 std::set_union(
35    std::begin(in1),
36    std::end(in1),
37    std::begin(in2),
38    std::end(in2),
39    std::back_inserter(out));
40
41 // Return the symmetric difference of two sorted ranges
42 // expected{1, 2, 3, 7, 8, 10, 11, 12}
43 std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
44 std::vector<int> in2{4, 5, 6, 9, 10, 11, 12};
45 std::vector<int> out{};
46 std::set_symmetric_difference(
```

```
47     std::begin(in1),
48     std::end(in1),
49     std::begin(in2),
50     std::end(in2),
51     std::back_inserter(out));
52
53  // Determines whether all elements of a sequence exists in a range
54  std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
55  std::vector<int> in2{2, 3, 4, 7, 8, 9};
56  auto res = std::includes(
57     std::begin(in1),
58     std::end(in1),
59     std::begin(in2),
60     std::end(in2));
```

14.9. Copy Algorithm

Listing 31: Copy Algorithm

```
1   // Leite den kompletten Input in den Output
2   using in_iter = std::istream_iterator<char>;
3   using out_iter = std::ostream_iterator<char>;
4   std::copy(in_iter{in}, in_iter{}, out_iter{out});
5
6   // expected {5, 6, 3, 5, 6, 3, 7, 4};
7   std::vector<int> in_out1{5, 6, 3, 7, 4, 0, 0, 0};
8   std::copy_backward(
9      std::begin(in_out1),
10    std::begin(in_out1) + 5,
11    std::end(in_out1));
12
13  // expected {5, 6, 3, 7, 9, 1, 5};
14  std::vector<int> in1{5, 6, 3, 7, 9, 1, 5};
15  std::vector<int> out1{};
16  std::copy(
17     std::begin(in1),
18     std::end(in1),
19     std::back_inserter(out1));
20
21  // expected {5, 3, 7, 5};
22  std::vector<int> in1{5, 6, 3, 7, 10, 10, 5};
23  std::vector<int> out1{};
24  std::copy_if(
25     std::begin(in1),
26     std::end(in1),
27     std::back_inserter(out1),
28     [](int const & i) {return i % 2;});
29
30  // expected {5, 6, 3, 7, 9, 1};
31  std::vector<int> in1{5, 6, 3, 7, 9, 1, 5};
32  std::vector<int> out1{};
33  std::copy_n(
34     std::begin(in1),
35     6,
36     std::back_inserter(out1));
```

14.10. Replace Algorithm

Listing 32: Replace Algorithm

```
// 1, 4, 3, 4, 1, 4, 3, 4
std::vector<int> in_out1{1, 2, 3, 2, 1, 2, 3, 2};
std::replace(
   std::begin(in_out1),
   std::end(in_out1),
   2,
   4);

// std::replace_if: expected{1, 0, 0, 4, 0, 6, 0, 8};
std::replace_if(
   std::begin(in_out1),
   std::end(in_out1),
   is_prime,
   0);

// std::replace_copy_if: expected{5, 6, 3, 7, 9, 1};
std::vector<int> out1{};
std::replace_copy_if(
   std::begin(in1),
   std::end(in1),
   std::back_inserter(out1),
   is_prime,
   0);
```

14.11. Transform Algorithm

Listing 33: Transform Algorithm

```
// Transform: Print a letter x Times, according to counts
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);

// 2^x: expected 32, 64, 128, 256, 1, 1024
std::vector<int> in1{5, 6, 7, 8, 0, 10};
std::vector<int> out1{};

std::transform(
      std::begin(in1),
      std::end(in1),
      std::back_inserter(out1),
      [](int i){return std::pow(2, i);});
```

14.12. Swap Algorithm

Listing 34: Transform Algorithm

```
// Swapt die beiden Vector in den jeweilig anderen
std::vector<int> in_out1{1, 2, 3, 4};
std::vector<int> in_out2{5, 6, 7, 8};

std::swap_ranges(
   std::begin(in_out1),
   std::end(in_out1),
   std::begin(in_out2));
```

14.13. Merge Algorithm

Listing 35: Merge Algorithm

```
// Merge two sorted 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);
std::merge(begin(r1), end(r1), begin(r2), end(r2), begin(d));
```

14.14. Erase-Remove-Idiom

Das `remove_if` verschiebt die validen Element einfach nach vorne und überschreibt damit die invaliden Elemente. Am Ende des Vectors bleiben dann Elemente mit unbekanntem Status übrig. Diese können mit dem Algorithmus `remove` entfernt werden.

Listing 36: Erase-Remove-Idiom

```
// Transforms the range [first,last) into a range with all the elements for which
    pred returns true removed, and returns an iterator to the new end of that range.
std::vector<unsigned> values{54, 13, 17, 95, 2, 57, 12, 9};
auto is_prime = [](unsigned u) {/*...*/};
auto removed = std::remove_if(begin(values), end(values), is_prime);
values.erase(removed, values.end()); // actually remove items


// expected{1, 3, 1, 3}: Remove elements from a sequence
std::vector<int> in_out1{1, 2, 3, 2, 1, 2, 3, 2};
auto new_end = std::remove(
   std::begin(in_out1),
   std::end(in_out1),
   2);

// expected{1, 4, 6, 8};
std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
auto new_end = std::remove_if(
   std::begin(in_out1),
   std::end(in_out1),
   is_prime);

// expected{1, 4, 6, 8}
std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8};
std::vector<int> out1{};
std::remove_copy_if(
   std::begin(in1),
```

```
27     std::end(in1),
28     std::back_inserter(out1),
29     is_prime);
30
31  // expected{1, 3, 1, 3}
32  std::vector<int> in1{1, 2, 3, 2, 1, 2, 3, 2};
33  std::vector<int> out1{};
34  std::remove_copy(
35     std::begin(in1),
36     std::end(in1),
37     std::back_inserter(out1),
38     2);
```

14.15. Reverse Algorithm

Listing 37: Reverse Algorithm

```
1   // Dreht einen std::vector<int> & values
2   reverse(begin(values), end(values));
3
4   // Copy a sequence, reversing its elements
5   // expected{10, 0, 8, 7, 6, 5}
6   std::vector<int> in1{5, 6, 7, 8, 0, 10};
7   std::vector<int> out1{};
8   std::reverse_copy(
9      std::begin(in1),
10     std::end(in1),
11     std::back_inserter(out1));
12
13  // expected{10, 0, 8, 7, 6, 5}: Reverse a sequence
14  std::vector<int> in_out1{5, 6, 7, 8, 0, 10};
15  std::reverse(
16     std::begin(in_out1),
17     std::end(in_out1));
```

14.16. Unique Algorithmen

Listing 38: Unique Algorithmen

```
1   // Remove consecutive duplicate values from a sequence
2   // expected{1, 3, 4, 2}
3   std::vector<int> in_out1{1, 1, 3, 3, 4, 2, 2, 2};
4
5   auto new_end = std::unique(
6      std::begin(in_out1),
7      std::end(in_out1));
8
9   // Copy a sequence, removing consecutive duplicate values
10  // expected{1, 3, 4, 2};
11  std::vector<int> in1{1, 1, 3, 3, 4, 2, 2, 2};
12  std::vector<int> out1{};
13
14  std::unique_copy(
15     std::begin(in1),
16     std::end(in1),
17     std::back_inserter(out1));
```

14.17. Rotate Algorithm

```
// Rotate the elements of a sequence
// expected{5, 6, 7, 8, 9, 1, 2, 3, 4}
std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::rotate(
   std::begin(in_out1),
   std::begin(in_out1) + 4,
   std::end(in_out1));

// Copy a sequence, rotating its elements
// expected{5, 6, 7, 8, 9, 1, 2, 3, 4}
std::vector<int> in1{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> out1{};
std::rotate_copy(
   std::begin(in1),
   std::begin(in1) + 4,
   std::end(in1),
   std::back_inserter(out1));
```

14.18. Numerische Algorithmen

```
// Create a range of sequentially increasing values
// expceted 0, 1, 2, 3, 4, 5, 6, 7
std::vector<int> out1(8);
std::iota(std::begin(out1), std::end(out1), 0);

// Return differences between adjacent values
// expected 1, 1, 2, -1, 6, -4, 2
std::vector<int> in1{1, 2, 4, 3, 9, 5, 7};
std::vector<int> out1(in1.size());
std::adjacent_difference(std::begin(in1), std::end(in1),  std::begin(out1));


// expected 121: Accumulate values in a range
std::vector<int> in1{1, 2, 3, 4, 5, 6};
int res = std::accumulate(std::begin(in1), std::end(in1), 100);

// 1, 21,321,4321, 54321: Return list of partial sums
std::vector<int> in1{1, 20, 300, 4000, 50000};
std::vector<int> out1(in1.size());
std::partial_sum(std::begin(in1), std::end(in1), std::begin(out1));

// Compute inner product of two ranges
// "begin, 1a, 2b, 3c, 2d, 1e"
std::vector<int> in1{1, 2, 3, 2, 1};
std::vector<char> in2{'a', 'b', 'c', 'd', 'e'};

std::string res = std::inner_product (
      std::begin(in1),
      std::end(in1),
      std::begin(in2),
      std::string{"begin"},
      [](std::string l, std::string r) {return l + ", " + r;},
      [](int i, char c) {return std::to_string(i) + c;});
```

14.19. Partition Algorithmen

```
// move elements for which a predicate is true to the beginning of a sequence,
// expected{7, 2, 3, 5, 4, 6, 1, 8, 9};
std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::partition(
   std::begin(in_out1),
   std::end(in_out1),
   is_prime);

// same as partition, but preserving relative ordering.
// expected {2, 3, 5, 7, 1, 4, 6, 8, 9};
std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::stable_partition(
   std::begin(in_out1),
   std::end(in_out1),
   is_prime);

// Find the partition point of a partitioned range
// expected = std::begin(in1) + 4;
std::vector<int> in1{2, 3, 5, 7, 1, 4, 6, 8, 9};
auto res = std::partition_point(
   std::begin(in1),
   std::end(in1),
   is_prime);

// expected true: Checks whether the sequence is partitioned
std::vector<int> in1{2, 3, 5, 7, 1, 4, 6, 8, 9};
bool res = std::is_partitioned(
   std::begin(in1),
   std::end(in1),
   is_prime);
```

14.20. Accumulate

Accumulate wird standardmässig fürs aufsummieren verwendet. Man kann ihn aber auch verwenden, wenn z.B ein Komma nur zwischen den Elementen eingefügt werden soll (und nicht am Ende).

Listing 39: Accumulate

```
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
```

14.21. If Algorithmen

Listing 40: If Algorithmen

```
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);

count_if
find_if
find_if_not
copy_if
remove_if
remove_copy_if
replace_if
replace_copy_if
```

14.22. Fill und Generate Algorithmen

Listing 41: Fill und Generate Algorithmen

```
// Fills the range [first,last) with copies of value.
// expected{42, 42, 42, 42, 42, 42, 42, 42};
std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
std::fill(
   std::begin(in_out1),
   std::end(in_out1),
   42);

// Fills a range with copies of the same value.
// expected{42, 42, 42, 42, 5, 6, 7, 8};
std::vector<int> in_out1{1, 2, 3, 4, 5, 6, 7, 8};
std::fill_n(
   std::begin(in_out1),
   4,
   42);

// Assign the result of a function object to each value in a sequence
// expected{100, 101, 102, 103, 104};
std::vector<int> out1(5);
int start = 100;
std::generate(
   std::begin(out1),
   std::end(out1),
   [start]() mutable {return start++;});

// Assign the result of a function object to each value in a sequence
// expected{100, 101, 102, 103, 104};
std::vector<int> out1{};
int start = 100;
std::generate_n(
   std::back_inserter(out1),
   5,
   [start]() mutable {return start++;});
```

14.23. Heap Algorithmen

Listing 42: Heap Algorithmen

```
std::vector<int> v{3,1,4,1,5,9,2,6};

// Construct a heap over a range
// Erstelle Heap (Baum von links her aufgefuellt) aus Vector
make_heap{v.begin(), v.end()};

// Switched Root mit letztem Item und stellt Heap Eigenschaft mit Downheap wieder her
pop_heap(v.begin(), v.end());
v.pop_back(); // entfernt das Element vom Heap

// Fuege Element dem Heap hinzu
v.push_back(8);
push_heap(v.begin(), v.end());

// Sortiere den Heap. Dies ist extrem schnell
sort_heap(v.begin(), v.end());
```

14.24. Distance Algorithm

Listing 43: Distance ALgorithm

```
#include "wcount.h"
#include "src/word.h"
#include <istream>
#include <iterator>

unsigned wcount(std::istream& in) {
   return std::distance(std::istream_iterator<Word>(in),
       std::istream_iterator<Word>());
}

// count only different words
unsigned wdiffcount(std::istream& in) {
   std::vector<Word> wordVector{std::istream_iterator<Word>(in),
       std::istream_iterator<Word>()};
   std::sort(wordVector.begin(), wordVector.end());
   return std::distance(wordVector.begin(), std::unique(wordVector.begin(),
       wordVector.end()));
}
```

15. Legacy C Structures

15.1. C Arrays

Wenn ein C Array einer Funktion übergeben wird, wird einzig ein Pointer auf das erste Element des Arrays übergeben. Dies Grösseninformation geht dabei verloren.

```
int a[5];
sum(a);

int sum(int a[]) { .. }

// Loesung

// oder std::vector<int> v(5);
// oder std::array<int, 5> sa;
// oder template <size_t n> int sum(int (&a)[n]);

// example
template<typename T, unsigned N> void printArray(std::ostream &out, T const (&x)[N]) {
   copy(x, x+N, std::ostream_iterator<T>{out, ", "});
}

int a[] = {1,2,3}M
printArray(std::cout, a);
```

15.2. Array Pointer

Array Pointer sind Random Access Iteratoren und können demnach inkrementiert/dekrementiert werden. Diese sollte aber nie so verwendet werden!

```
++p, *p, p+int, p[int], *(p+int)
```

15.3. Array Initialisierung

Arrays die direkt Initialisiert werden, nehmen implizit die Dimension anhand der Anzahl Elemente. Wird das Array ohne Elemente Initialisiert, wird das Default Value des Types genommen. Wird es nicht initialisiert, kann undefined behaviour entstehen.

Listing 44: Initializing an Array

```
int five[5] // might be unitialized
int five[5]{}; // 5 zeros
double m[2][3]{{1,2,3},{4,5,6}};
```

16. Dynamic Heap Memory Managemnt

Das Heap Memory sollte nur in Spezialfällen selbständig verwaltet werden! Deshalb sollte kein `new` verwendet werden. (Alloziert Objekte auf dem Heap). Wenn man es trotzdem macht, ist man auch für das Aufräumen verantworlich. Ansonsten drohen Memory Leaks, DanglingPointers und Double Deletes. (es gibt keine Garbage Collection. Die Garbage Collection kommt bei der schliessenden geschweiften Klammer sofern man die Pointer mit einer der folgenden Funktionen erzeugt wurde.). Möchte man Heap Pointer verwenden sollte man dies immer mit `std::unique_ptr<T>` oder `std::shared_ptr<T>` verwenden. Das PIMPL (Pointer to Implementation) Idiom versteckt die Implementierung der eigentlichen Klassenmember einer Klasse in der Impl-Klasse.

- Delete Pointer muss grundsätzlich nie aufgerufen werden
- Unique Pointer: Für unshared Heap Memory (Kann nicht kopiert werden)
- Shared Pointer: Für geteiltes Heap Memory (Arbeiten ähnlich wie Java Referenzen. Können kopiert und verschoben werden)
- Wenn das letzte shared_ptr Handle zerstört wird, wird das allozierte Objekt gelöscht.
- Shared Pointer haben das Problem von Zyklen. Deshalb gibt es den weak_ptr um die Zyklen aufzubrechen.

```
#include <memory>
#include <string>

// real usage of unique heap pointers
std::unique_ptr<T> factory(int i) {
   return std::make_uniqe<T>(i);
}

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

// real usage of shared heap pointers
struct A {
   A(int a, std::string b, char c) {}
};

int main() {
   auto an_a = A_factory();
   auto b=an_a; // second pointer to same object
   A c{*b};
   auto another = std::make_shared<A>(c);
}
```

```
// person.h
#include <memory>
#include <string>
#include <vector>
#include <iosfwd>

// type forward declaration!
using PersonPtr=std::shared_ptr<class Person>;

class Person {
   std::string name;
   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);
}
};


// usage
#include "person.h"
std::string name{"name"};
auto pers = Person::makePerson(name);
```

17. Vererbung

Vererbung wird immer dann verwendet, wenn man Verhalten wiederverwenden und erweitern möchte. (besser: Adapter verwenden) Vererbung ist deshalb schlecht, weil es eine starke Kopplung zwischen den Klassen erzeugt.

- Vererbung ist standardmässig `public` (bei Klassen!). Bei Structs ist die Vererbung implizit `public`
- Die Konstruktoren werden nicht implizit vererbt
- Es wird immer zuerst das Parent Objekt konstruiert und danach die Subklassen
- `const` wird zur Signatur gezählt
- Zuweisungen oder übermitteln von Parameter `by value` von abgeleiteten Klassen in Variablen vom Typ der Base Klasse resultieren in **Object Slicing**
  → Nur Base Class Member Variablen werden behalden. (`MyBase base = subVar;`)

```
class MyClass : Base {}; // implicit private
struct MyStruct : Base {}; // implicit public
class MyClass : public Base {
   public:
      // inherit constructor
      using Base::Base;
};

// Initializing base classes (super call)
class DerivedWithCtor : public Base {
   DerivedWithCtor():Base{}{} // default constrcutor
};

// always call base class befor member init
class DerivedWithCtro : public Base {
   DerivedWithCtor(int i, int j) : Base{i}, myLocalVar{j} {}
};

// abstract function (zero)
virtual void foo() const = 0;
```

17.1. Sichtbarkeiten

- public: Member sind sichbar und nutzbar in den abgeleiteten Klassen (solange in der abgeleiteten Klasse **kein** Member mit dem gleichen Namen definiert ist)
- protected: Sind nur in abgeleiten Klassen sichtbar
- private: Sind nur in der Base Klasse sichtbar

17.2. Dynamic Binding

Fürs Dynamic Dispatch ist das Keyword **virtual** nötig. (Overhead von VTable) Damit die korrekte Funktion aufgerufen wird, wird in der VTable nachgeschaut.

```
class PolymorphicBase {
   public:
   virtual void doit() { /* something*/ }
};
class Implementor: public PolymorphicBase {
   public:
   void doit() {
      /* something else */
   }
};
```

17.3. Abstrakte Klassen

Abstrakte Methoden werden auch als ”pure virtual” bezeichnet.

```
struct AbstractBase {
   // base classes with virtual members require a virtual
   // destructor if heap allocated without shared_ptr (not recommended)
   virtual ~AbstractBase(){}

   // abstract member function (zero -> pure virtual)
   virtual void doitnow() = 0;
};
```

18. CUTE

Listing 45: Basic CUTE Test File

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

void testSayHelloSaysHelloWorld() {
   std:ostringstream out{}; // use default constructor
   sayHello(out);
   ASSERT_EQUAL("Hello, world\nLeftarrow", 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);
}
```

18.1. Streams

Listing 46: Test IO Stream

```
void wcount(std::istream &in, std::ostream &out) {
   using it=std::istream_iterator<std::string>;
   int count = distance(it{in}, it{});
   out << count;
}

void test() {
   std::istringstream in{"this is a test"};
   std::ostringstream out{};
   wcount(in, out);
   ASSERT_EQUAL("4", out.str());
}
```

18.2. Asserts

Gleichheit `ASSERT_EQUAL()`

Exceptions `ASSERT_THROWS(empty_vector.at(0), std::out_of_range)`

Gleitkommazahlen `ASSERT_EQUAL_DELTA()`

A. Listings

B. Abbildungsverzeichnis

C. Tabellenverzeichnis