SEMESTER HS2020

C++ Zusammenfassung

Joel Schaltegger, Philipp Emmenegger
29. Dezember 2020

Inhaltsverzeichnis

1 Introduction to C++

Why C++

- Works on almost all plattforms
- Multi-paradigm language with zero-cost abstraction
- High level abstraction facilities

In C++ gibt es keinen Garbage Collector, wie man es aus anderen Sprachen, wie Java oder C# kennt. Warnung: Wenn Code "falsch"geschrieben wurde, kann Undefined Behavior auftreten.

1.1 C++ Compilation Process

*.cpp Files

- Also called Implementation File
- For function implementations (can be in .h as well)
- Source of compilation

*.h File

- Also called Header File
- Declarations and definitions to be used in other implementation files

3 Phases of compilation

- Preprocessor Textual replacement of preprocessor directives (include)
- Compiler Translation of C++ code into machine code (source file to object file)
- Linker Combination of object files and libraries into libraries and executables

1.2 Declarations and Definitions

All things with a name that you use in a C++ program must be declared before you can do so.

One Definition Rule

While a program element can be declared several times without problem there can be only one definition of it. This is called the **One Definition Rule** (ODR)!

Include Guard

Include guards ensure that a header file is only included once. Multiple inclusions could violate the One Definition Rule when the header contains definitions.

```
1  #ifndef SAYHELLO_H_
2  #define SAYHELLO_H_
3
4  #include <iosfwd>
5  struct Greeter { /* Some Code */ };
6
7  #endif /* SAYHELLO_H_ */
```

2 Values and Streams

2.1 Variable Definitions

- Defining a variable consists of specifying its type, its variable name and its initial value. E.g. int x{42};
- Empty braces mean default initialization. E.g. double x{};
- Using = for initialization we can have the compiler determine its type. E.g. auto const i = 5;

Constants

- Adding the const keyword in front of the name makes the variable a single assignment variable, aka a constant. E.g. int const x{42};
  - Must be initialized and immutable
- Use the keyword constexpr if the variable is required to be fixed at compile time. E.g. double constexpr pi{3.14159};

Why shoud I use const?

- A lot of code needs names for values, but often does not intend to change it
- It helps to avoid reusing the same variable for different purposes (code smell)
- It creates safer code, because a const variable cannot be inadvertently changed
- It makes reasoning about code easier
- Constness is checked by the compiler
- It improves optimization and parallelization (shared mutable state is dangerous)

Important types for Variable

- short, int, long, long long - each also available as unsigned version
- bool, char, unsigned char, signed char
- float, double, long double
- void is special, it is the type with no values
- class defined: E.g. std::string, std::vector

2.2 Values and Expressions

Integer to boolean: 0 = False, every other value = True
if (a < b < c) → zuerst wird a < b ausgewertet (true oder false). Dann wird der Boolean mit einem int (c) verglichen. Der Bool wird dafür implizit in 0 oder 1 gecastet.

Literal Example	Type	Value
'a'	char	Letter a, value: 97
'\n'	char	\<NL\> character, value: 10
'\x0a'	char	\<NL\> character, value: 10
1	int	1
42L	long	42
5LL	long long	5
int{} (not really a literal)	int	0 (default value)
1u	unsigned int	1
42ul	unsigned long	42
5ull	unsigned long long	5
020	int	16 (octal 20)
0x1f	int	31 (hex 1F)
0XFULL	unsigned long long	15 (hex F)
0.f	float	0
.33	double	0.33
1e9	double	1000000000 (10^9)
42.E-12L	long double	0.00000000042 ($42*10^{-12}$)
.3l	long double	0.3
"hello"	char const [6]	Array of 6 chars: h e l l o \<NUL\>
"\012\n\\"	char const [4]	Array of 4 chars: \<NL\> \<NL\> \ \<NUL\>

2.3 Strings and Sequences

std::string is C++'s type for representing sequences of char (which is often only 8 bit) and are mutable. That means, we can modify the content. (Vergleich zu Java: Dort würde ein neues String Objekt erstellt werden)

Grundsätzlich werden Strings also als char const[ ] abgespeichert. Mit dem namespace std::literals hat man die Option hinter dem String eine 's' anzufügen, um das Objekt effektiv als String zu speichern. z.B. "ab"s

toUpper Iterator

```
1  void toUpper(std::string & value) {
2      transform(cbegin(value), cend(value), begin(value), ::toupper);
3  }
```

2.4 Input and Output Streams

Functions taking a stream object must take it as a reference, because they provide a side effect to the stream (i.e., output characters).

Reading from Input

- Reading into a std::string always works. Unless the stream is already !good() → Spaces werden übersprungen (neues String-Objekt)!
- Reading into other types (e.g. int) has no error recovery. A wrong input puts the stream into status fail and the characters remain in the input.
- Post-read check: if (in » age) { ... }
- Multiple subsequent reads are possible: if (in » symbol » count) { ... }
- Remove fail flag: in.clear()
- Ignore one char: in.ignore();
- Helpfull for reading: while (in.good()) um die Leseoperationen setzen.

Robust reading of an int value

```
1  // Use an std::istringstream as intermediate stream
2  int inputAge(std::istream & in)
3      std::string line{}
4      while (getline(in, line)) {
5          std::istringstream is{line};
6          int age{-1};
7          if (is >> age) {
8              return age;
9          }
10     }
11     return -1;
12 }
13 /* ## dealing with invalid input ##
14 in.clear(); //remove fail flag
15 in.ignore(); // ignore one char*/
```

Stream States

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

3 Sequences and Iterators

3.1 Std::array and std::vector

Array

- C++'s std::array<T, N> is a fixed size Container
    - T is a template type parameter. N is a positive integer parameter
- std::array can be initialized with a list of elements
    - The size of an array must be known at compile time and cannot be changed
    - Otherwise it contains N default constructed elements: std::array<int, 5> emptyArray
- The size is bound to the array object and can be queried using .size()
- Avoid plain C Array whenever possible: int arr [ ]{1, 2, 3, 4, 5};
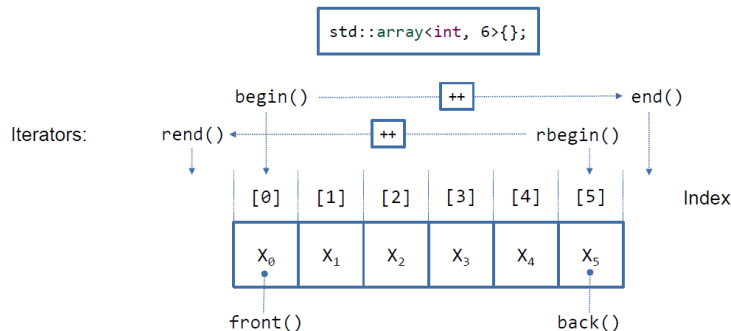
Vector

- C++'s std::vector<T> is a Container: contains its elements of type T (no need to allocate them)
    - java.util.ArrayList<T> is a collection = keeps references to T objects
    - T is a template parameter
- std::vector can be initialized with a list of elements
    - Otherwise it is empty: std::vector<double> vd{ };
    - Other construction means might need parentheses (legacy)
- When an initializer is given, the element type can be deduced! std::vector{1, 2, 3, 4, 5};
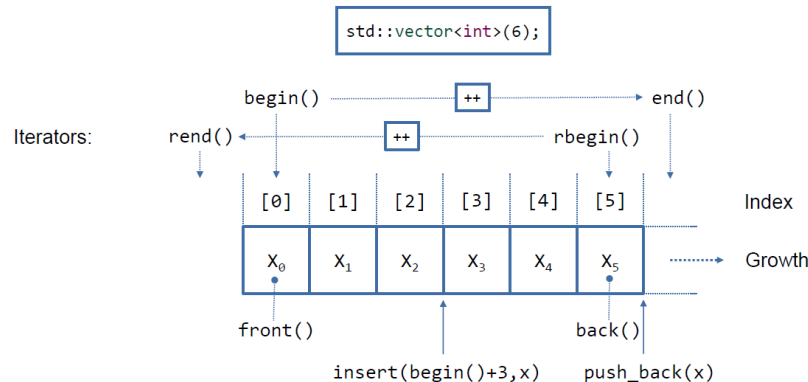
Parenthesis at definition allow providing initial size, when type of elements is a number:
std::vector<string> words{6}; → Um sicher zu gehen die Grösse mit runden Klammern angeben.

Für beide Datentypen:
Element access using subscript operator [ ] or at():
at throws an exception and [ ] has undefined behavior on invalid access.

Speicherort:
Generell werden alle Elemente einer Klasse auf dem Stack abgelegt. So auch der Vector. Fügt man dem Vector ein Element hinzu, wird davon eine unabhängige Kopie auf dem Heap erstellt. Der Vector referenziert auf das neue Objekt.

3.2 Iteration

Element Iteration (Range-Based for-Loop)

	const: • element cannot be changed	**non-const:** • element can be changed

reference:
• element in vector is accessed
	`for (auto const & cref : v) {`
	  `std::cout << cref << '\n';`
	`}`
		`for (auto & ref : v) {`
		  `ref *= 2;`
		`}`

copy:
• loop has own copy of the element
	`for (auto const ccopy : v) {`
	  `std::cout << ccopy << '\n';`
	`}`
		`for (auto copy : v) {`
		  `copy *= 2;`
		  `std::cout << copy << '\n';`
		`}`

Iteration with Iterators

```
1  // Changing the element in a non-const container is possible in this way
2  for (auto it = std::begin(v); it != std::end(v); ++it) {
3      std::cout << (*it)++ << ", "; }
4  // Guarantee to just have read-only access with std::cbegin() and std::cend()
5  for (auto it = std::cbegin(v); it != std::cend(v); ++it) {
6      std::cout << *it << ", "; }
```

3.3 Iterators with Algrotithms

```
1  // Counting values: std::count
2  size_t count_blanks (std::string s) {
3      return std::count(s.begin(), s.end(), ' '); }
4
5  // Summing up all values in a vector: std::accumulate
6  std::vector<int > v{5, 4, 3, 2, 1};
7  std::cout << std::accumulate(std::begin(v), std::end(v), 0) << " = sum\n";
8
9  // Number of elements in range: std::distance
10 void printDistanceAndLength (std::string s) {
11     std::cout << "distance: " << std::distance(s.begin(), s.end()) << '\n';
12     std::cout << "in a string of length: " << s.size() << '\n';
13 }
14
15 // std::for_each
16 void printAll(std::vector<int> v) {
17     std::for_each(crbegin(v), std::crend(v), print); }
18
19 // std::for_each with Lambda
20 void printAll(std::vector<int> v, std::ostream & out) {
21     std::for_each(crbegin(v), std::crend(v), [&out](auto x) {
22         out << "print: " << x << '\n';
23     }); }
24
25 // std::copy (target needs to be an iterator too. target.end() would not work)
26 std::vector<int> source{1, 2, 3}, target{};
27 std::copy(source.begin(), source.end(), std::back_inserter target(target));
28
29 // Filling a vector with std::fill
30 std::vector<int> v(10);
31 std::fill(std::begin(v), std::end(v), 2);
32 // Or even easier:
33 std::vector v(10, 2);
34
35 // std::generate()
36 std::vector<double> powerOfTwos(5);
37 double x{1.0};
38 std::gegerate(powerOfTwos.begin(), powerOfTwos.end(), [&x] {return x *= 2.0; });
39 // std::generate_n
40 std::vector<double> powerOfTwos();
41 double x{1.0};
```

```
42  std::gegerate(std::back_inserter(powerOfTwos), 5, [&x] {return x *= 2.0; });
43
44  // fills a range with subsequent values (1,2,3,...): std::iota()
45  std::vector<int> v(100);
46  std::iota(std::begin(v), std::end(v), 1);
47
48  // std::find(), std::find_if() - If no match exists the end of the range is returned
49  auto zero_it = std::find(std::begin(v), std::end(v), 0);
50  if (zero_it == std::end(v)) { std::cout << "no zero found \n"; }
51  // std::count_if()
52  std::cout << std::count_if(begin(v), end(v), [](int x) { return isEven(x); }) << " even
        numbers\n";
```

3.4 Iterators for I/O

- std::ostream_iterator<T>
    - outputs values of type T to the given std::ostream
    - No end() marker needed for ouput, it ends when the input range ends
- std::istream_iterator<T>
    - reads values of type T from the given std::istream
    - End iterator is the default constructed std::istream_iterator<T>{}
    - It ends when the Stream is no longer good()

Shorter types with the keyword using.

```
1   // Copy Strings from standard input to standard output
2   // Skips white space !!
3   using input = std::istream_iterator<string>;
4   input eof{};
5   input in{std::cin};
6   std::ostream_iterator<string> out{std::cout, " "};
7   std::copy(in, eof, out);
8
9   // std::istreambuf_iterator<char> uses std::istream::get to get every character
10  // Only works with char-like types
11  using input = std::istreambuf_iterator<char>;
12  input eof{};
13  input in{std::cin};
14  std::ostream_iterator<char> out{std::cout, " "};
15  std::copy(in, eof, out);
16
17  // Fill a vector from a stream (copy with back_inserter)
18  using input = std::istream_iterator<int>;
19  input eof{};
20  std::vector<int> v{};
21  std::copy(input{std::cin}, eof, std::back_inserter(v));
22
23  // Fill a vector from a stream (directly rom two iterators)
24  using input = std::istream_iterator<int>;
25  input eof{};
26  std::vector<int> const v{input{std::cin}, eof};
```

4 Functions and Exceptions

4.1 Functions

	const: • Parameter cannot be changed	non-const: • Parameter can be changed
reference: • Argument on call-site is accessed	```void f(std::string const & s) {` ` //no modification` ` //efficient for large objects` `}```	```void f(std::string & s) {` ` //modification possible` ` //side-effect also at call-site` `}```
copy: • Function has its own copy of the parameter	```void f(std::string const s) {` ` //no modification` ` //used for maximum constness` `}```	```void f(std::string s) {` ` //modification possible` ` //side-effect only locally` `}```

When to use & and const Parameters:

- Value Parameter:
  - Default case
- Reference Parameter
  - When side-effect is required at call-site
- Const-Reference Parameter
  - Possible optimization, when type is large (costly to copy) and no side effects desired at call site
  - For non-copyable objects
- Const Value Parameter
  - The coding style guide of your project this might prefer this over non const value parameters
  - Could prevent changing the parameter in the function inadvertently

Function Overloading

The same function name can be used for different functions if parameter number or types differ
→ Functions cannot be overloaded just by their return type
→ If only the parameter type is different there might be ambiguities

Default Arguments

- A function declaration can provide default arguments for its parameters from the right.
  - E.g: void incr(int & var, unsigned delta = 1);
- Implicit overload of the function with fewer parameters
  - If n default arguments are provided, n+1 versions of the function are declared
- Default arguments can be omitted when calling the function

Functions as Parameters

Functions are "first class"objects in C++
→ You can pass them as argument, or keep them in reference variables.

```cpp
// As Argument (No Lambads/Captures before function allowed)
void applyAndPrint(double x, double f(double)) {
    std::cout << "f(" << x << ") = " << f(x) << '\n'; }

// As reference variable
double (&h)(double);

// std::function: template for Lambdas --> #include <functional>
void applyAndPrint(double x, std::function<double(double)> f) {
    std::cout << "f(" << x << ") = " << f(x) << '\n'; }
int main() {
    double factor{3.0};
    auto const multiply = [factor](double value() { // Lambda Function
        return factor * value; };
    applyAndPrint(1.5, multiply);
```

4.2 Failing Functions

What should you do, if a function cannot fulfill its purpose?
- Ignore the error and provide potentially undefined behavior
- Return a standard result to cover the error
- Return an error code or error value
- Provide an error status as a side-effect
- Throw an exception

Ignore the error:

- Relies on the caller to satisfy all preconditions
- Most efficient implementation (no unnecessary checks)
- Simple for implementer, harder for caller
- Should be done consciously and consistently!

Error Value

- Only feasible if result domain is smaller than return type
- E.g: Error Value for strings: std::string::npos
- Optional as return type can contain no value.
  → #include<optional>
  E.g: std::optional<std::string>
  - caller side: has to check with: var.has_value()

Return standard result:

- Reliefs the caller from the need to care if it can continue with the default value
- Can hide underlying problems
- Often better if caller can specify its own default value

Error Status

- Requires reference parameter
- Alternative: Global Variable (BAD decision)
- E.g: std::istreams's states (good(), fail()) is changed as a side-effect of input

Exceptions

- Prevent execution of invalid logic by throwing an exception

4.3 Exceptions

Principle: Throw by value, catch by const reference.
Functions can be declared to explicitly not throw an exception with the keyword noexcept
#include <stdexcept>
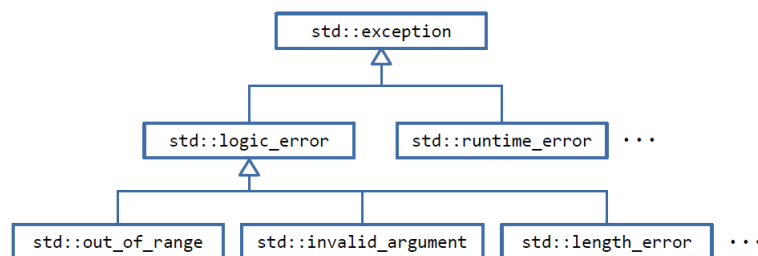

```cpp
// Throw an Exception
if (x < 0) {
    throw std::invalid_argument{"square_root imaginary"};
}

// Catch an Exception
try {
    throwingCall();
} catch (type const & e) { /* Handle type exception */
} catch (type2 const & e) { /* Handle type2 exception */
} catch (...) { /* Handle other exception types */
} // Caught exceptions can be rethrown with throw;

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

5 Classes and Operators

5.1 Classes

- A class defines a new type
- A class is usually defined in a header file
- At the end of a class definition a semicolon is required
- Include guard in header file
- Keyword: calss or struct
    - Default visibility: Class: private, struct: public
- Access specifiers:
    - private: visible only inside the class; for hidden data members
    - protected: also visible nin subclasses
    - public: visible everywhere; for the interface of the class
- Constructor: Initializer list for member initialization

```
 1  #ifndef DATE_H_                          #include "Date.h"
 2  #define DATE_H_
 3  class Date {                             Date::Date(int year, int month, int day)
 4      int year, month, day;                  : year {year}, month {month}, day {day} {
 5  public:                                    /* ... */
 6      Date(int year, int month, int day);  }
 7
 8      static bool isLeapYear(int year);    bool Date::isLeapYear(int year) {
 9                                             /* ... */
10  private:                                 }
11      bool isValidDate() const;            bool Date::isValidDate() const {
12  };                                         /* ... */ }
13  #endif /* DATE_H_ */                     }
```

Special Constructors

- Default Constructor
    - No parameters. Implicitly available if there are no other explicit constructors. Has to initialize member variables with default values.
- Copy Constructor
    - Has one <own-type> const & parameter. Implicitly available (unless there is an explicit move constructor or assignment operator). Copies all member variables.
- Move Constructor
    - Has one <own-type> && parameter. Implicitly available (unless there is an explicit copy constructor or assignment operator). Moves all memebers
- Typeconversation Constructor
    - Has one <other-type> const & parameter. Converts the input type if possible. Declare explicit to avoid unexpected conversions.
- Initializer List Constructor
    - Has one std::initializer_list parameter. Does not need to be explicit, implicit conversion is usually desired. Initializer List constructors are preferred if a variable is initialized with { }
- Destructor
    - Named like the default constructor but with a ∼. Must release all resources. Implicitly available. Must not throw an exception. Called automatically at the end of the block for local instances.

```
 1  class Date {
 2  public:
 3      Date(int year, int month, int day);
 4      Date(); // Default-Constructor
 5      Date() = default; // explizit Default-Constructor
 6      Date(Date const &); // Copy-Constructor
 7      Date(Date &&); // Move-Constructor
 8      explicit Date(std::string const &); // Typeconversation-Constructor
 9      Date(std::initializer_list<Element> elements); // Initializer List-Constructor
10      ~Date(); // Destructor
11      Date(Date const &) = delete; // delete implicit Copy-Constructor
12  };
```

5.2 Operator Overloading

- Custom operators can be overloaded for user-defined types.
- Declared like a function, with a special name: <returntype> operator op(<parameters>);
- Non-Overloadable Operators: :: , .* , . , ?:
- Keyword inline when defined in header file. But: Problem with private varaibles.
  → Define operator in class

std::tie creates a tuple and binds the argument with lvalue references. std::tuple provides comparison operators: == , != , < , <= , > , >=

```cpp
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)));
    }
};

// With std::tie
#include <tuple>
bool operator<(Date const & rhs) const {
        return std::tie(year, month, day) < std::tie(rhs.year, rhs.month, rhs.day);
}

// Sending Date to std::ostream
class Date {
    int year, month, day;
public:
    std::ostream & print(std::ostream & os) const {
        os << year << "/" << month << "/" << day;
        return os;
    }
};
inline std::ostream & operator<<(std::ostream & os ,Date const & date) {
    return date.print(os);
}

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

/* Keyword friend, um Kapselung zwischen private/public zu brechen. --> Dann ist es mö
    glich die read/print Operatoren ohne inline Hilfsfunktionen zu schreiben. */
// Header File:
class Date {
    int year, month, day;
public:
    friend std::istream & operator>>(std::istream & is, Date & date);
    friend std::ostream & operator<<(std::ostream & os, Date const & date);
};
// .cpp File:
std::istream & operator<<(std::istream & is, Date & date) {
    // read logic
    return is;
}
std::ostream & operator<<(std::ostream & os, Date const & date) {
    // print logic
    return os;
}
```

6 Namespaces and Enums

6.1 Namespaces

- Namespaces are scopes for grouping and preventing name clashes
- Global namespace has the :: prefix
- Nesting of namespaces is possible
- Nesting of scopes allows hiding names
- Anonymous namespaces (without a name) are only accessible in the current file
- Keyword using to use a defined namespace in your file.
  E.g: after typing **using std::string;** you can write: **string s{"my string"};**

Argument Dependent Lookup

When the compiler encounters an unqualified function or operator call with an argument of a user defined type it looks into the namespace in which that type is defined to resolve the function/operator.

```
namespace one {                                 #include "adl.h"
    struct type_one{};
    void f(type_one) { /* ... */ } /* 1 */      int main() {
}                                                   one::type_one t1{};
                                                    f(t1); // Function 1
namespace two {                                     two::type_two t2{};
struct type_two{};                                  f(t2); // Function 2
    void f(type_two) { /* ... */ } /* 2 */          h(t1); // No Function found
    void g(one::type_one) { /* ... */ } /* 3 */     two::g(t1); // Function 3
    void h(one::type_one) { /* ... */ } /* 4 */     g(t1); // No Function found
}                                                   //(5 gefunden, aber arg passt nicht)
void g(two::type_two) { /* ... */ } /* 5 */         g(t2) // Function 5
```

6.2 Enums

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

```
// Unscoped enumeration, leaks into surrounding scope
enum DayOfWeek {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
};   0   1   2   3   4   5   6
// Implicit conversion:
int day = Sun;

// Scoped enumeration (class keyword), requires DayOfWeek::
enum class DayOfWeek {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
};   0   1   2   3   4   5   6
// No implicit conversion to int, requires static_cast:
int day = static_cast<int>(DayOfWeek::Sun);

// Conversion from int to enum always requires a static_cast:
DayOfWeek tuesday = static_cast<DayOfWeek>(1);
```

→ **Beispiel im Anhang unter Woche06**

6.3 Arithmetic Types

- Arithmetic types must be equality comparable
- Boost can be used to get != operator → boost::equality_comparable
- It might be convenient to have the output operator
- Result must be in a specific range (Modulo)

→ **Beispiel im Anhang unter Woche06**

7 Standard Container & Iterators

7.1 STL Containers: General API

- Sequence Containers
    - Elements are accessible in order as they were inserted created
    - Find in linear time through the algorithm find
- Associative Containers
    - Elements are accessible in sorted order
    - find as member function in logarithmic time
- Hashed Containers
    - Elements are accessible in unspecified order
    - find as member function in constant time

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

7.2 Sequence Containers

std::vector & std::array

Siehe Kapitel3: Sequences and Iterators (Seite 6)

std::deque<T> → #include<deque>

std::deque is like std::vector but with additional, efficient front insertion/removal

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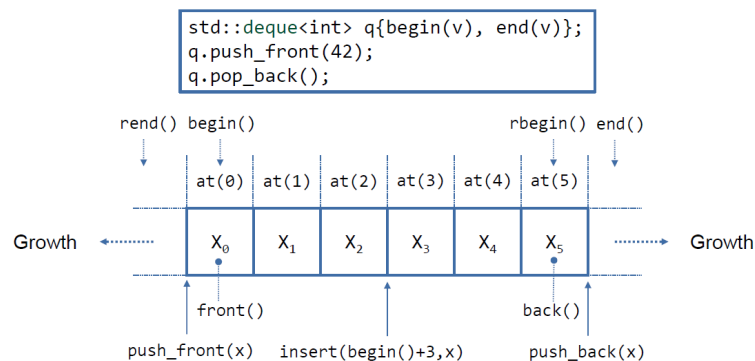

std::list<T> → #include<list>

Efficient insertion in any position (Double-Linked List). Lower efficiency in bulk operations. Requires member function call for sort etc. Only bi directional iterators no index access

```
std::list<int> l(5, 1);
```
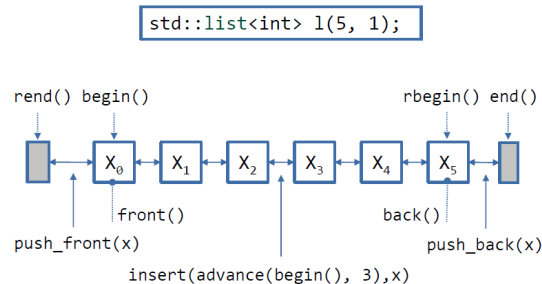
std::forward_list<T> → #include<forward_list>

Efficient insertion AFTER any position, but clumsy with iterator to get "before"position (Singly-Linked List). Only forward iterators, clumsy to search and remove, use member functions not algorithms.

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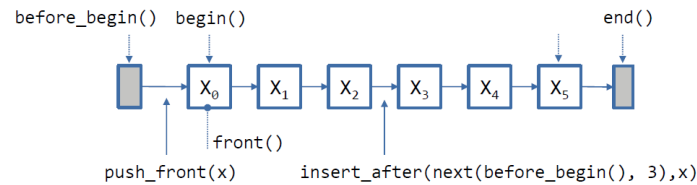

std::stack → #include<stack>

Uses std::deque (or std::vector, std::list) and limits its functionality to stack operations. Iteration not possible.

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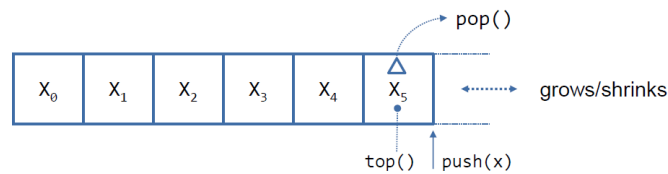

std::queue → #include<queue>

Uses std::deque (or std::list) and limits its functionality to queue operations. Iteration not possible.

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


std::priority_queue → #include<stack>

Uses std::deque (or std::vector) and limits its functionality to stack operations. top() element is always the smallest.

7.3 Associative Containers

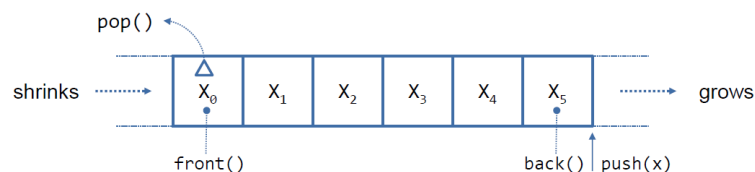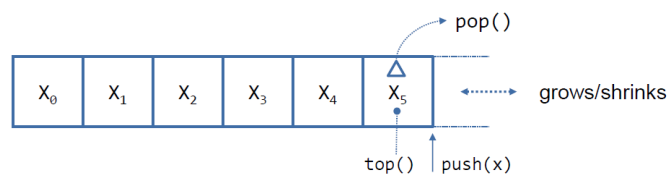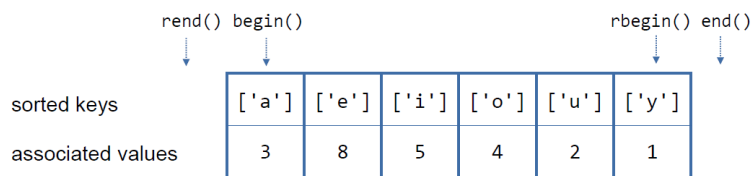|	Key Only	Key-Value Pair
Key Unique	std::set<T>	std::map<K, V>
Multiple Equivalent Keys	std::multiset<T>	std::multimap<K, V>

std::set → #include <set>

- std::set<int> values{7, 1, 4, 3, 2, 5, 6}
- Stores elements in sorted order (ascending by default, overwritten by 2nd template parameter)
- Iteration walks over the elements in order
- Member functions to check if element exists: .find(element) or .count(element)

std::map → #include <map>

Stores key-value pairs in sorted order. Default: By key ascending (Overwritten by 3rd template param). Use .first() for key and .second() for value.

```
std::map<char, size_t> vowels{{'a', 3}, {'e', 8}, {'i', 5}, {'o', 4}, {'u', 2}, {'y', 1}};
```

```
            rend() begin()                              rbegin() end()

sorted keys       ['a']  ['e']  ['i']  ['o']  ['u']  ['y']
associated values   3      8      5      4      2      1
```

std::multiset → #include <set>, std::multimap →#include <map>

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

```
std::multiset<char> letters{'a', 'a', 'c', 'c', 'c', 'e', 'e', 'f'};
```

```
              rend() begin()                     rbegin() end()

sorted values    'a'  'a'  'c'  'c'  'c'  'e'  'e'  'f'

         lower_bound('b')                     upper_bound('e')
                        equal_range('c')
```
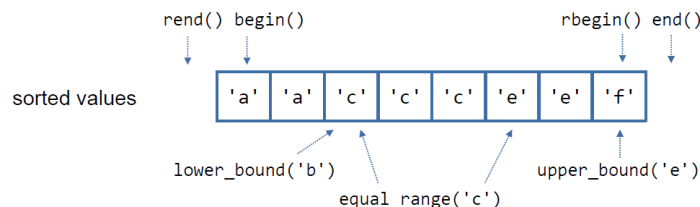
7.4 Hashed Containers

std::unordered_set → #include <unordered_set>, std::unordered_map → #include <unordered_map>

Usage is almost equivalent to std::set/std::map. Except for the lack of ordering.

7.5 Iterators

What are these Iterator Categories for?
- Some algorithms only work with powerful iterators, e.g: std::sort() requires a pair of random access iterators (it needs to jump forward and backward)
- Some algorithms can be implemented better with more powerful iterators, e.g: std::advance() or std::distance()

Input Iterator

- Supports reading the current element (of type Element)
- Allows for one-pass input algorithms (cannot step backwards)
- Models the std::istream_iterator and std::istream
- Can be compared with == and != to other iterator objects of the same type: It
- Can be copied (after increment all other copies are invalid!)

Forward Iterator

- Can do whatever an input iterator can, plus: Supports changing the current element
- Allows only for one-pass input algorithms
- Models the std::forward_list iterators

Bidirectional Iterator

- Can do whatever a forward iterator can, plus: Can go backwards
- Allows for forward-backward-pass algorithms
- Models the std::set iterators

Random Access Iterator

- Can do whatever a bidrectional iterator can, plus:
    - Directly access element at index
    - Go n steps forward or backward
    - Subtact two iterators to get the distance
    - Compare with relational operators ($<, <=, >, >=$)
- Allows random access in algorithms
- Models the std::vector iterators

Output Iterator

- Can write value to current element, but only once
- Modeled after std::ostream_iterator

Iterator Functions → #include <iterator>

```
std::distance(start, goal);
std::advance(itr, n);
```

- std::distance() counts the number of hops iterator start must make until it reaches goal
- std::advance() lets itr hop n times

std::advance vs. std::next

```cpp
int main() {
    std::vector<int> primes{2, 3, 5, 7, 11, 13};

    auto current = std::begin(primes);
    auto afterNext = std::next(current);
    std::cout << "current: " << *current << " afterNext: " << *afterNext << '\n';

    std::advance(current, 1);
    std::cout << "current: "<< *current << " afterNext: " << *afterNext << '\n';
}
```

8 STL Algorithms

Why should we use Algorithms?
- Correctness: It is much easier to use an algorithm correctly than implementing loops correctly
- Readability: Applying the correct algorithm expresses your intention much better than a loop
- Performance: Algorithms might perform better than handwritten loops

8.1 Functor

A functor is a type (class) that provides a call operator: operator().

```cpp
struct Accumulator {
    int count{0};
    int accumulatedValue{0}
    void operator()(int value) {
        count++;
        accumulatedValue += value;
    }
    int average() const; // To implement
    int sum() const;
};

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

Standard Functor Template Classes → #include <functional>

```cpp
// Lambda to transform
transform(v.begin(), v.end(), v.begin(), [](int x){ return -x;} );
// Standard Library's Functor: std::negate
transform(v.begin(), v.end(), v.begin(), std::negate<int>{} );
// Sorting with Standard Library Funcor: std::greater
sort(v.begin(), v.end(), std::greater<>{} );
```

Standard Functor Classes → #include <functional>

```cpp
// Binary arithmetic and logical        Unary
plus<> (+)                              negate<> (-)
minus<> (-)                             logical_not<> (!)
divides<> (/)
multiplies<> (*)                        //Binary comparison
modulus<> (%)                           less<> (<)
logical_and<> (&&)                      less_equal<> (<=)
logical_or<> (||)                       equal_to<> (==)
                                        greater_equal<> (>=)
                                        greater<> (>)
                                        not_equal_to<> (!=)
```

8.2 Algorithm Examples

- transform: Lambda, function or functor for map
- merge: ranges must to be sorted
- remove/erase: Remove does not actually delete element, it moves the not-removed elements to the front. To get rid of *removed* elements, usually *erase* is called
- accumulate: Sums elements that are addable (+ operator)
- _if versions: Take a predicate (instead of value) to provide a condition
- _n versions: algorithm is related to a number (e.g: instead of last iterator)
  - search_n, copy_n, fill_n, generate_n, for_each_n

```cpp
1  // transform
2  // combined contains: | "ggg" | "" | "u" | "yyyy" | "" | "oo" |
3  std::vector<int> counts{3, 0, 1, 4, 0, 2};
4  std::vector<char> letters{'g', 'a', 'u', 'y', 'f', 'o'};
5  std::vector<std::string> combined{};
6  auto times = [](int i, char c) { return std::string(i, c); };
7  std::transform(begin(counts), end(counts), begin(letters), std::back_inserter(combined),
       times);
8
9  // merge
10 std::vector<int> r1{9, 12, 17, 23, 54, 57, 85, 95};
11 std::vector<int> r2{2, 30, 32, 41, 49, 63, 72, 88};
12 std::vector<int> d(r1.size() + r2.size(), 0);
13 std::merge(begin(r1), end(r1), begin(r2), end(r2), begin(d));
14
15 // remove
16 std::vector<unsigned> values{54, 13, 17, 95, 2, 57, 12, 9};
17 auto is_prime = []unsigned(u) { /* ... */ };
18 auto removed = std::remove_if(begin(values), end(values), is_prime);
19
20 // accumulate
21 std::vector<std::string> longMonths {"Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"};
22 std::string accumulatedString = std::accumulate(
23     next(begin(longMonths)), // Second element
24     end(longMonths), // End
25     longMonths.at(0), //First element, usually the neutral element
26     [](std::string const & acc, std::string const & element) {
27         return acc + ", " + element;
28     }); //Jan, Mar, May, Jul, Aug, Oct, Dec
29
30 // count_if
31 std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
32 auto isPrime = [](unsigned u) { /* ... */ };
33 auto nOfPrimes = std::count_if(begin(numbers), end(numbers), isPrime);
34
35 // copy_n
36 std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
37 std::vector<unsigned> top5(5);
38 std::copy_n(rbegin(numbers), 5, begin(top5));
```

Heap Algorithms

- Implementing a binary heap on a sequenced container
- Requires random access iterator
- Guarantees: Top element is the largest (max), Adding and removing performance guarantees
- Used for implementing priority queues
- Heap operations: make_heap, pop_heap, push_heap, sort_heap

```cpp
1  // create heap
2  std::vector<int> v{3, 1, 4, 1, 5, 9, 2, 6}
3  make_heap(v.begin(), v.end()); // create tree order
4  // remove largest element
5  pop_heap(v.begin(), v.end()); // reorder
6  v.pop_back();
7  // add element
8  v.push_back(8);
9  push_heap(v.begin(), v.end()); // reorder
10 sort_heap(v.begin(), v.end()); // smallest element first in vector
```

9 Function Templates

9.1 Templates Syntax

- Keyword template for declaring a template
- Template parameter list: Contains one or more template parameters
    - A template parameter is a placeholder for a type, which can be used within the template as a type
    - A type template parameter is introduced with the typename (or class) keyword

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

// smaller.cpp
#include "min.h"
#include <iostream>
int main() {
    int first;
    int second;
    if (std::cin >> first >> second) {
        auto const smaller = min(first, second);
        std::cout<< "Smaller of "<< first<< " and "<< second<< " is: "<< smaller<< '\n';
    }
}
```

Template Definition

- Templates are usually defined in a header file. They are implicitly inline
- The definition in a source file is possible, but then it can only be used in that translation unit
- Type checking happens twice: When the template is defined, when template is instantiated (used)

9.2 Variadic Templates

- For function templates with an arbitrary number of parameters
- Needs at least one pack parameter
- Pack Expansion: For each argument in that pack an instance of the pattern is created
- In an instance of the pattern the parameter pack name is replaced by an argument of the pack
- Needs a base case for the recursion (after the last parameter is done, it would call the function without a parameter, which is invalid) → Base case must be written before the template function.

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

9.3 Overloading

Problem: Wenn Pointer als Argument übergeben werden, werden die Pointer-Adressen miteinander verglichen und nicht die Objekte selbst.
Lösung: Mehrere function templates mit dem Selben Namen können existieren. Spezifischere/Normale Funktionen werden den function templates aber bevorzugt.

```cpp
// other template function in min.h
template <typename T>
    T * min(T * left, T * right) {
        return *left < *right ? left : right;
}
```

10 Class Templates

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

Rules

- Define class templates completely in header files
- Member functions of class templates: Either in class template directly, or as inline function template in the same header file
- When using language elements depending directly or indirectly on a template parameter, you must specify typename when it is naming a type
- static member variables of a template class can be defined in header without violating ODR, even if included in several compilation units

Type Aliases & Dependent Names

- It is common for template definitions to define type aliases in order to ease their use
- Within the template definition you might use names that are directly or indirectly depending on the template parameter.
- Dependent Name: Compiler geht standardmässig davon aus, dass es sich im eine Variable, oder eine Funktion hanelt. Wenn es ein Typ ist (wie size_type), muss das keyword typename verwendet werden.

Members Outside of Class Template

Must have: template declaration, keyword inline, member signature, template ID as namescope.
$\rightarrow$ Example im Anhang

Class Template that Inherit

Rule: Always use this->variable (or className::) to refer to inherited members in a template class.

Template (Partial) Specialization

partial: still using a template parameter, but provide (some) arguments
Explicit: providing all arguments with concrete types.
$\rightarrow$ Must declare non-specialized template first. Most specialized version that fits is used.

```
1  // Partial Specialization              Explicit Specialization
2  template <typename T>                  template <>
3  struct Sack<T *>;                      struct Sack<char const *>;
4
5  // Prevent Creation of Sack<int *> --> Delete Destructor
6  template <typename T>
7  struct Sack<T *> {
8      ~Sack() = delete;
9  }
```

Extending Sack<T>

Implementation im Anhang Kapitel Übungen Woche 10

- Create a Sack<T> using iterators to fill it
- Create a Sack<T> of multiple default values
- Create a Sack<T> from initializer list
- Obtain copy of contents in a std::vector (for iteration and inspection)
- Auto-deducting T for a Sack<T> from initializer list $\rightarrow$ User Provided Deduction Guides
- Allow to vary the type of the container to be used

11 Heap Memory Management

Don't do it yourselfe: Always rely on library classes for managing it (if possible).

11.1 When is Heap Memory used

- Stack memory is scarce
- Might be needed for creating object structures
- Polymorphic factory functions to class hierarchies

11.2 Heap Memory Legacy

Don't do that. C++ allows allocating objects on the heap directly.
If done manually, you are responsible for deallocation and risk undefined behaviour:

- Memory leaks
- Dangling pointers
- Double deletes

No garbage collection happens, it is your responsibility.

```
1   // Don't use new / delete
2   auto ptr = new int{};
3   std::cout << *ptr << '\n';
4   delete ptr;
```

11.3 Modern Heap Memory Management

With the following smart pointers you don't have to call *delete ptr;* yourself.
Still: Prefer storing a value locally as value-type variable

std::unique_ptr<T>

- Used for unshared heap memory
- For local stuff that must be on the heap (rare)
- Can be returned from a factory function
- Only a single owner exists
- Not best for class hierarchies (use *std::shared_ptr*)
- Can not be copied

Use-Cases:
- As member variable
  - To keep a polymorphic reference instantiated by the class or passed in as *std::unique_ptr* and transferring ownership
- As local variable
  - To implement RAII (Resource Acquisition Is Initialization)
  - Can provide custom deleter function as second template argument
- *std::unique_ptr<T>const p{new{}}*
  - Cannot transfer ownership
  - Cannot leak

```
1   // Factory
2   std::unique_ptr<X> factory(int i) {
3       return std::make_unique<X>(i);
4   }
```

std::shared_ptr<T>

- Works more like Java's references
- It can be copied and passed around
- The last one ceasing to exist deletes the object
- *std::make_shared<T>* allows T's public constructor's parameter to be used

Use-Cases:
- If you need heap-allocated objects, because you create your own object networks
- To support run-time polymorphic container contents or class members that can not be passed as reference
- Factory functions returning *std::shared_ptr* for heap allocated objects
- First check if alternatives are available
  - (const) references as parameter types or class members
  - Plain member objects or containers with plain class instances

```
1   struct A { /* constructor of A */ }
2   // Factory
3   auto createA() {
4       return std::make_shared<A>(5, "hi", 'a');
5   }
6   int main() {
7       auto anA = createA();
8       auto sameA = anA; // second pointer to same obj
9       A copyA{*sameA}; // copy ctr
10      auto another = std::make_shared<A>(copyA); // copy ctr on heap
11  }
```

Problem: Cyclic shared-Pointer Structures

Usually, the last shared-Pointer handle destroyed will delete the allocated object.
Cyclic shared-Pointer structures keep themselfe alive, if they have circular dependencies to each other.
Even if initial shared-Pointer handles are destroyed. → *std::weak_ptr* breaks such cycles

std::weak_ptr <T>
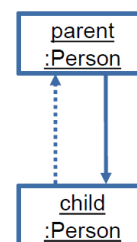
- The *shared_ptr* cycles need to be broken
- *weak_ptr* does not allow direct access to the object
- A *weak_ptr* does not know weather the pointee is still alive
-
- with *lock()* a *shared_ptr* to the object can be acquired if alive

12 Dynamic Polymorphism

Reasons for using Inheritance

- Mix-in functionality from empty base class
  - Often with own class as template argument
  - No inherited data members, only added functionality

```
1  struct Date : boost::equality_comparable<Date> {
2      // ...
3  }
```

- Adapting concrete classes
  - No additional own data members
  - Convenient for inheriting member functions and constructors

```
1  template<typename T, typename Compare>
2  struct indexableSet : std::set<T, Compare> {
3      // ...
4  }
```

Inheritance for dynamic binding

- Implementing a design pattern with dynamic dispatch
  - Provide common interface for a variety of dynamically changing or different implementations
  - Exchange functionality at run-time
- Base class/interface provides a common abstraction that is used by clients
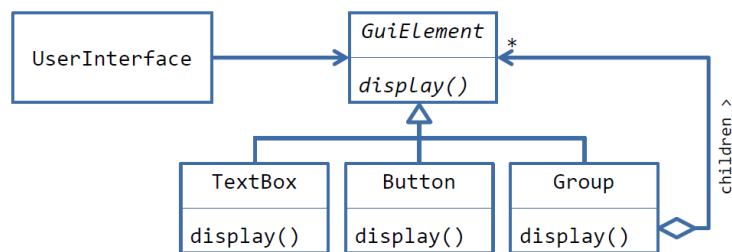
12.1 Inheritance Syntax

- Inherit class: default private
- Inherit struct: default public
- *public*-Keyword: public Base class
- *private*-Keyword: private Base struct
- Multiple Bases: Sequence is important (Reihenfolge)
- With interface inheritance, base class must be public

```
1  class Base {};
2  struct Base2 {};
3  class DerivedPrivateBase : Base {}; // private -> class
4  class DerivedPublicBase : public Base {}; // public -> public keywork
5  struct DerivedPublicBase : Base {}; // public -> struct
6  struct MultipleBases : public Base, private Base2 {}; // Multiple Bases
```

Initializing multiple base classes

- Base constructors can be explicitly called in the member initializer list
- You should put base class constructor before the initialization of members

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

12.2 Dynamic polymorphism

- Operator and function overloading and templates allow polymorphic behaviour at compile time
- Dynamic polymorphism needs object references or (smart) pointers to work
  - Syntax overhead
  - The base class must be a good abstraction
  - Copying carries the danger of slicing (partially copying)

Shadowing Member Functions

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

```
struct Base {
    // shadowed function
    void sayHello() const {
        std::cout << "Hi, I'm Base\n";
    }
};

struct Derived : Base {
    // shadowing function
    void sayHello() const {
        std::cout << "Hi, I'm Derived\n";
    }
};

void greet(Base const & base) {
    base.sayHello();
}

int main() {
    Derived derived{};
    greet(derived); // Hi, I'm Base (static call)
}
```

Virtual Member Functions

- Dynamic polymorphism requires base classes with *virtual* member functions
- *virtual* Member functions are bound dynamically

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

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

void greet(Base const & base) {
    base.sayHello();
}

int main() {
    Derived derived{};
    greet(derived); // Hi, I'm Derived (dynamic call)
}
```

Overriding virtual Member Functions

- *virtual* is inherited and can be omitted in the derived calss
- It is possible to mark an overriding function with *override*

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

Signatures when Overriding

- To override a virtual function in the base class the signature must be the same
- constness of the member function belongs to the signature

Calling virtual Member functions

- Value Object
  – Class type determines function, regarless of virtual
- Reference
  – Virtual member of derived class called through base class reference
- Smart Pointer
  – Virtual member of derived class called through smart pointer to base class
- Dumb Pointer (rarely used)
  – Virtual member of derived class called through base class pointer

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

// Reference
void greet(Base const & base) {
    // calls sayHello() of the actual type
    base.sayHello();
}

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

```
16        base->sayHello();
17    }
18
19    // Dump Pointer
20    void greet(Base const * base) {
21        // calls sayHello() of the actual type
22        base->sayHello();
23    }
```

Abstract Base Classes: Pure Virtual

- There are no Interfaces in C++
- A pure virtual member function makes a class abstract
- To mark a virtual member function as pure virtual it has zero assigned after its signature
- Abstract classes cannot be instantiated (like in Java)

```
1    struct AbstractBase {
2        virtual void doitnow() = 0;
3    };
```

Destructors (virtual)

- Classes with virtual members require a virtual Destructor
- Otherwise when allocated on the heap with *make_unique* and assigned to a *unique_ptr* only the destructor of Base is called

```
1    struct Fuel {
2        virtual void burn () = 0;
3        virtual ~Fuel() { std::cout << "put into trash\n"; }
4    };
5
6    struct Plutonium : Fuel {
7        void burn () { std::cout << "split core\n"; }
8        ~Plutonium() { std::cout << "store many years\n"; }
9    }
10
11   int main() {
12       // Both destructors called
13       std::unique_ptr<Fuel> surprise = std::make_unique<Plutonium>();
14   }
```

Why Inheritance can be bad

- Very strong coupling between subclass and base class
- You can hardly change the base class
- API of base class must fit for all subclasses (hard)

Problem with Inheritance and Pass-by-Value

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

```
1    // Pass-by-Value (slicing of derived object)
2    void modifyAndPrint(Base base) {
3        base.modify();
4        base.print(std::cout);
5    }
6
7    int main() {
8        Derived derived{25};
9        // reduces the derived object to its base (sliced)
10       modifyAndPrint(derived);
11   }
```

Problems with Member Hiding

- Member functions in derived classes hide base class member with the same name, even if different parameters are used
- By *using [member]*, the hidden member becomes visible

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

struct Derived : Base {
    using Base::Base;
    // using Base::modify; <-- resolves hiding issue

    // hides base function
    void modify(int value) {
        member += value;
    }
};

int main() {
    Derived derived{25};
    derived.modify(); // Error, base function hidden
}
```

Prevent Object Slicing in Base Class

- Declare the copy operations as deleted

```cpp
struct Base {
    Base & operator=(Base const & other) = delete;
    Book(Book const & other) = delete;
}
```

Guidelines

- Only apply inheritance and virtual members if you know what you do
- Do not create classes with virtual members by default
- Follow the Liskov Substitute Principle
  - Base class states must be valid for subclasses
  - Do not break invariants of the base class
  - Don't change semantics unexpectedly
  - *If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.¨*

13 Initialization and Aggregates

13.1 Kinds of Initialization

Default Initialization

- Simplest form of initialization
  - Simply dont provide an initializer
  - Effect depends on the kind of entity we declare
  - Does not work for references
- Danger: when using default initialized entities
- Does not necessarily work with *const*
- **Static values:**
  - Zero initialized first
  - then their type's default constructor is called
- **Non-Static integral and floating point variables:**
  - Unitialized
- **Objects of class types:**
  - Constructed using their default constructor
- Danger: Reading an unitialized value incurs undefined behaviour

```cpp
int global_variable; // implicitly static = 0, kein default ctor

void di_function() {
    static long local_static;
    long local_variable; // unitialized
    std::string local_text; // default ctor
}

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

Value Initialization

- Initialization performed with empty () or {}
  - {} is prefferable, works in more cases
- Invoked the default constructor for class types

```cpp
void vi_function() {
    int number{}; // 0
    std::vector<int> data {}; // default ctor
    std::string actually_a_function(); // DANGER! may be a function declaration, use {}
}
```

Direct Initialization

- Similar to Value Initialization
- Uses non-empty () or {}
  - prefer {}
- When using {} only applies if not a class type
- Danger: most vexing parse lurks with ()

```cpp
void diri_function() {
    int number{32};
    std::string text { "CPl" }; // passender ctor
    // zweideutig, direct initialization / function declaration, use {}
    word vexing (std::string());
}
```

Most Vexing Parse:
- Two interpretations:
  - Initialization with a value-initialized string
  - Declaration of a function returning a word and taking an unnamed pointer to a function returning a string
- The compiler interprets the second option (function declaration)
- Therefore prefer {}

Copy Initialization

- Initialization using =
- If the object has class type and the right hand side has the same type
  - If right hand side is a temporary, the object is constructed in-place, not copied
  - otherwise the copy ctor is invoked
- Otherwise, a suitable conversion sequence is searched for
- Also applies to return statements and throw/catch

```
1  std::string string_factory() { retzrb ""; }
2
3  void ci_function() {
4      std::string in_place = string_factory();
5      std::string copy = in_place;
6      std::string converted = "CPl";
7  }
```

List Initialization

- Uses non-empty {}
- Direct List Initialization
- Copy List Initialization
- Constructors are selected in two phases
  - If there is a suitable ctor taking *std::initializer_list*, it is selected
  - Otherwise, a suitable ctor is searched

```
1  // Direct List Initialization
2  std::string direct { "CPl" };
3  // Copy List Initialization
4  std::string copy = { "CPlA" };
```

List Initialization Pitfall

- Since *std::initializer_list* ctor is preffered, you might run into trouble

```
1  // vector(size_type count, const T& value, const Allocator& alloc = Allocator());
2
3  int ouch() {
4      std::vector<int> data{ 10, 42 }; // list initialization, size() = 2
5      std::vector<ing> data2( 10, 42 ); // other ctor, size() = 10
6
7      return data[5]; // DANGER, undefined behaviour
8  }
```

13.2 Aggregates

- Simple class types
  - Can have other types as public base classes
  - Can have member variables and functions
  - Must not have user-provided, inherited or explicit constructors
  - Must not have protected or private direct members
- Mostly used for *simple* types
  - No invariant that has to be established
  - Example: Data Transfer Objects
- Arrays are also Aggregates

Aggregate Initialization

- Special case of List Initialization
- If more elements than members are given, the program is ill-formed
- Can also provide less initializers than there are bases and members
  - If the uninitialized members habe a member initializer, it is used
  - otherwise they are initialized from empty lists

Example

```cpp
struct person {
    std::string name;
    int age{42};

    bool operator<(person const & other) const {
        return age < other.age;
    }

    void write(std::ostream & out) const {
        out << name << ": " << age << "\n";
    }
};

int main() {
    person rudolf{"Rudolf", 32};
    person mike{"Mike Schmid"}; // Age will be set to 42
    rudolf.write(std::cout);
}
```

14 Anhang

14.1 Übungen Woche 06

Enumeration Example

```cpp
// statemachine.h
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

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

#endif /* STATEMACHINE_H_ */

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

14.2 Übungen Woche 07

Example: Stack and Queue

```cpp
#include <stack>
#include <queue>
#include <iostream>
#include <string>

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

Example Code using std::set

```
1  #include <set>
2  #include <iostream>
3
4  void filterVowels(std::istream & in, ostream & out) {
5      std::set const vowels{'a', 'e', 'o', 'u', 'i', 'y'};
6      char c{};
7      while (in >> c) {
8          if (!vowels.count(c)) { // returns 0 (false) or 1 (true)
9              out << c;
10         }
11     }
12 }
13 int main () {
14     filterVowels(std::cin, std::cout);
15 }
```

Example Code using std::map

```
1  // Example 1
2  void countVowels(std::istream &in, ostream &out) {
3      std::map<char, size_t> vowels{{'a', 0}, {'e', 0}, 'i', 0}, {'o', 0}, {'u', 0}, {'y',
       0}};
4      char c{};
5      while (in >> c) {
6          if (vowels.count(c)) { // only count those chars that are already in the map
7          ++vowels[c];
8              for_each(cbegin(vowels), cend(vowels), [&out](auto const & entry) {
9                  // entry is a pair<char, size_t>
10                 out << entry.first << " = " << entry.second << '\n';
11             });
12         }
13     }
14 }
15
16 // Example 2
17 void countStrings(std::istream & in, std::ostream & out) {
18     std::map<std::string, size_t> occurrences{};
19     std::istream_iterator<std::string> inputBegin{in};
20     std::istream_iterator<std::string> inputEnd{};
21     for_each(inputBegin, inputEnd, [&occurrences](auto const & str) {
22         ++occurrences[str]; // inserts new entry with value 1, if not existst, else:
       counts ++
23     });
24     for(auto const & occurrence : occurrences) {
25         out << occurrence.first << " = " << occurrence.second << '\n'
26     }
27 }
```

Example Code using std::multiset

```
1  void sortedStringList(std::istream & in, std::ostream & out) {
2      using inIter = std::istream_iterator<std::string>;
3      using outIter = std::ostream_iterator<std::string>;
4      std::multiset<std::string> words{inIter{in}, inIter{}};
5      copy(cbegin(words), cend(words), outIter(out, "\n"));
6      auto current = cbegin(words);
7      while (current != cend (words)) {
8          auto endOfRange = words.upper_bound(*current);
9          copy(current, endOfRange, outIter{out, ", "});
10         out << '\n'; // next range on new line
11         current = endOfRange;
12     }
13 }
```

Example Code using std::unordered_set

```
1  #include <algorithm>
```

```cpp
2  #include <iostream>
3  #include <iterator>
4  #include <unordered_set>
5
6  int main () {
7      std::unordered_set<char> const vowels{'a', 'e', 'i', 'o', 'u'};
8      using in = std::istreambuf_iterator<char>;
9      using out = std::ostreambuf_iterator<char>;
10     remove_copy_if(in{std::cin}, in{}, out{std::cout}, [&](char c) {
11         return vowels.count(c);
12     });
13 }
```

Example Code using std::unordered_map

```cpp
1  #include <unordered_map>
2  #include <iostream>
3  #include <string>
4  int main() {
5      std::unordered_map<std::string, int> words{};
6      std::string s{};
7      while (std::cin >> s) { ++words[s]; }
8      for (auto const & p : words) {
9          std::cout << p.first << " = " << p.second << '\n';
10     }
11 }
```

14.3 Übungen Woche 10

Adapting Standard Containers

```cpp
1  template<typename T>
2  struct safeVector : std::vector<T> {
3      using container = std::vector<T>;
4      using container::container; // or using std::vector<T>::vector; --> Inherits
           constructors with using
5      using size_type = typename container::size_type; // Type Alias
6      using reference = typename container::reference; // Type Alias
7      using const_reference = typename container::const_reference; // Type Alias
8
9      reference operator[](size_type index) {
10         return this->at(index); // this-> for member access
11     }
12
13     const_reference operator[](size_type index) const {
14         return this->at(index);
15     }
16     // should also provide front/back with empty() check
17 };
```

Template Sack

```cpp
1  // sack.h
2  template<typename T> // Class template with one typename parameter
3  class Sack {
4      using SackType = std::vector<T>; // Alias type: SackType
5      using size_type = typename SackType::size_type; // Dependent name: size_type
6      SackType theSack{};
7
8  public:
9      bool empty() const {
10         return theSack.empty{};
11     }
12     size_type size() const {
13         return theSack.size();
14     }
15     void putInto(T const & item) {
16         theSack.push_back(item);
17     }
18     T getOut(); // Member function forward declaration
```

```
19  };
20
21  template <typename T> // implementing member function outside of a class
22  inline T Sack<T>::getOut() {
23      if (empty()) {
24          throw std::logic_error{"Empty Sack"};
25      }
26      auto index = static_cast<size_type>(rand() % size()); // Pick random element
27      T retval{theSack.at(index)};
28      theSack.erase(theSack.begin() + index);
29      return retval;
30  }
```

Extending Sack<T>

```
1   template <typename T>
2   class Sack {
3       //...
4   public:
5       Sack() = default; // Retain default constructor
6
7       // Add constructor template for iterators
8       template <typename Iter>
9       Sack(Iter begin, Iter end) : theSack(begin, end) {}
10
11      // Add constructor for initializer_list
12      Sack(std::initializer_list<T> values) : theSack(values) {}
13
14      // Add constructor for Deduction Guide
15      Sack(size_type n, T const & value) : theSack(n, value) {}
16
17      // extracting a std::vector<T> from a Sack<T>
18      template <typename Elt>
19          explicit operator std::vector<Elt>() const {
20                  return std::vector<Elt>(theSack.begin(),theSack.end());
21      }
22
23  };
24
25  // User Provided Deduction Guide for Sack<T> (after template definition)
26  template <typename Iter>
27  Sack(Iter begin, Iter end) -> Sack<typename std::iterator_traits<Iter>::value_type>;
```

Template/Default/Non-Type Template Parameter

```
1   // A template can take templates as parameters
2   template<typename T, template<typename...> typename Container> // ... for unspecified
        amount of parameters
3   class Sack;
4
5   // with default argument:
6   template<typename T, template<typename...> typename Container = std::vector>
7   class Sack;
8
9   // Templates can have non type parameters
10  template <typename T, auto n>
11  auto average(std::array<T, n> const & values) {
12      auto sumOfValues = accumulate(begin(values), end(values), 0);
13      return sumOfValues / n;
14  }
```

14.4 Übungen Woche 11

```
1   struct monster{
2           monster(){ cout << "a monster is bread\n"; }
3           ~monster(){ cout << "monster killed\n"; }
4           void health(){ cout << "immortal?\n";  }
5           virtual void attack(){ cout << "roar\n";}
6   };
7
8   struct troll: monster {
```

```
 9              troll(){ cout << "a troll grows\n";}
10              ~troll() { cout << "troll petrified\n";}
11              void attack(){ swing_club();}
12              virtual void swing_club(){
13                      cout << "clubbing kills me\n";
14                      myhealth--;
15              }
16              void health(){cout << "troll-health:"<< myhealth<<'\n';}
17 protected:
18              int myhealth{10};
19 };
20
21 struct forum_troll: troll {
22              forum_troll():troll{}{ cout << "not quite a monster\n";}
23              ~forum_troll(){ cout << "troll banned\n";}
24              virtual void swing_club(){
25                      cout << "swinging is healthy\n";
26                      myhealth++;
27              }
28              void attack(){ cout << "write stupid things\n";}
29 };
30
31 int main(){
32              cout << "a ------\n";
33              forum_troll ft{};
34              troll t{ft} ;
35              monster &m{ft};
36              cout << "b ------\n";
37              ft.attack();
38              t.attack();
39              m.attack();
40              cout << "c ------\n";
41              ft.swing_club();
42              t.swing_club();
43              cout << "d ------\n";
44              ft.health();
45              t.health();
46              m.health();
47              cout << "end ------\n";
48 }
49
50 // Output
51 /*
52 a ------
53 a monster is bread
54 a troll grows
55 not quite a monster
56 b ------
57 write stupid things
58 clubbing kills me
59 write stupid things
60 c ------
61 swinging is healthy
62 clubbing kills me
63 d ------
64 troll-health:11
65 troll-health:8
66 immortal?
67 end ------
68 troll petrified
69 monster killed
70 troll banned
71 troll petrified
72 monster killed
73 */
```

14.5 Includes

```
1 // Only the declaration for input and output streams
2 #include <iosfwd>
3
4 // Implementation of input stream
5 #include <istream>
6
7 // Implementation of output stream
8 #include <ostream>
```

```cpp
 9
10  // Declaration of both streams and additionally std::cout, std::cin, std::cerr
11  #include <iostream>
12
13  // Functions: std::tolower(c), std::isupper(c)
14  #include <cctype>
15
16  // Strings
17  #include <string>
18
19  // Arrays
20  #include <array>
21
22  // Vectors (ArrayList)
23  #include <vector>
24
25  // Iterators: std::count, std::accumulate, std::distance, std::for_each
26  #include <iterator>
27
28  // std::iota
29  #include <numeric>
30
31  // function template, which allows passing lambdas (with capture)
32  #include <functional>
33
34  // std::tie (creates tupel)
35  #include <tupel>
36
37  // std::deque<T>
38  #include <deque>
39
40  // std::list<T>
41  #include <list>
42
43  // std::forward_list<T>
44  #include <forward_list>
45
46  // std::stack
47  #include <stack>
48
49  // std::queue
50  #include <queue>
51
52  // std::priority_queue
53  #include <stack>
54
55  // std::set
56  #include <set>
57
58  // std::map
59  #include <map>
60
61  // std::unordered_set --> Hashed
62  #include <unordered_set>
63
64  // std::unordered_map --> Hashed
65  #include <unordered_map>
66
67  // std::unique_ptr / std::shared_ptr (smart pointer)
68  #include <memory>
```