Semester HS2020

C++ Zusammenfassung

Joel Schaltegger
25. Dezember 2020

Lizenz
"THE BEER-WARE LICENSE"(Revision 42): Joel Schaltegger wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy me a beer in return.

Inhaltsverzeichnis

1 Introduction to C++

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.

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_

#include <iosfwd>
struct Greeter { /* Some Code */ };

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

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

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

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

Stream States

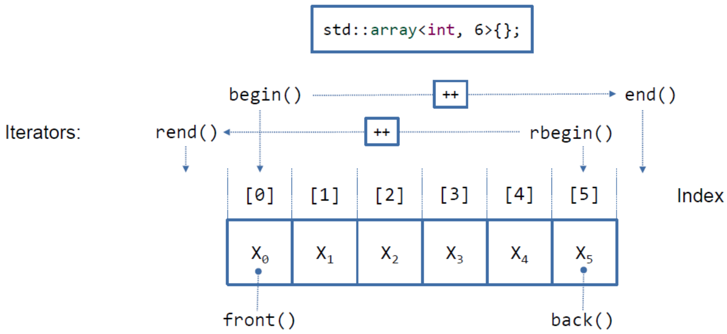
State Bit Set	Query	Entered
<none>	is.good()	initial is.clear()
failbit	is.fail()	formatted input failed
eofbit	is.eof()	trying to read at end of input
badbit	is.bad()	unrecoverable I/O error

3 Sequences and Iterators

3.1 Std::array and std::vector

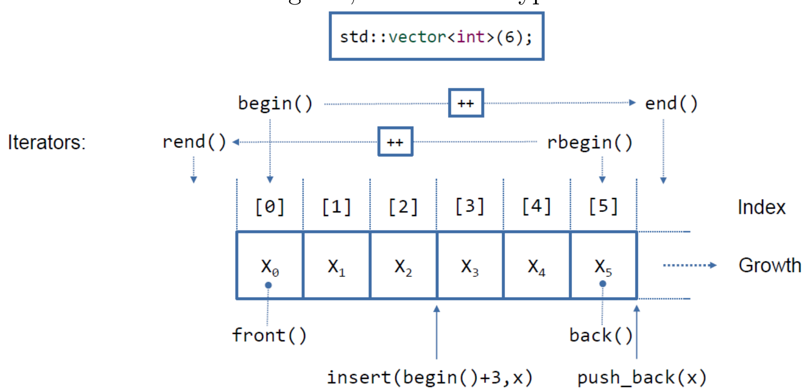
Array

- 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

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

3.3 Iterators with Algrotithms

```
// Counting values: std::count
size_t count_blanks (std::string s) {
    return std::count(s.begin(), s.end(), ' '); }

// Summing up all values in a vector: std::accumulate
std::vector<int > v{5, 4, 3, 2, 1};
std::cout << std::accumulate(std::begin(v), std::end(v), 0) << " = sum\n";

// Number of elements in range: std::distance
void printDistanceAndLength (std::string s) {
    std::cout << "distance: " << std::distance(s.begin(), s.end()) << '\n';
    std::cout << "in a string of length: " << s.size() << '\n';
}

// std::for_each
void printAll(std::vector<int> v) {
    std::for_each(crbegin(v), std::crend(v), print); }

// std::for_each with Lambda
void printAll(std::vector<int> v, std::ostream & out) {
    std::for_each(crbegin(v), std::crend(v), [&out](auto x) {
        out << "print: " << x << '\n';
    }); }

// std::copy (target needs to be an iterator too. target.end() would not work)
std::vector<int> source{1, 2, 3}, target{};
std::copy(source.begin(), source.end(), std::back_inserter target(target));

// Filling a vector with std::fill
std::vector<int> v(10);
std::fill(std::begin(v), std::end(v), 2);
// Or even easier:
std::vector v(10, 2);

// std::generate()
std::vector<double> powerOfTwos(5);
double x{1.0};
std::gegerate(powerOfTwos.begin(), powerOfTwos.end(), [&x] {return x *= 2.0; });
// std::generate_n
std::vector<double> powerOfTwos();
double x{1.0};
```

```
std::gegerate(std::back_inserter(powerOfTwos), 5, [&x] {return x *= 2.0; });

// fills a range with subsequent values (1,2,3,...): std::iota()
std::vector<int> v(100);
std::iota(std::begin(v), std::end(v), 1);

// std::find(), std::find_if() - If no match exists the end of the range is returned
auto zero_it = std::find(std::begin(v), std::end(v), 0);
if (zero_it == std::end(v)) { std::cout << "no zero found \n"; }
// std::count_if()
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.

```
// Copy Strings from standard input to standard output
// Skips white space !!
using input = std::istream_iterator<string>;
input eof{};
input in{std::cin};
std::ostream_iterator<string> out{std::cout, " "};
std::copy(in, eof, out);

// std::istreambuf_iterator<char> uses std::istream::get to get every character
// Only works with char-like types
using input = std::istreambuf_iterator<char>;
input eof{};
input in{std::cin};
std::ostream_iterator<char> out{std::cout, " "};
std::copy(in, eof, out);

// Fill a vector from a stream (copy with back_inserter)
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> v{};
std::copy(input{std::cin}, eof, std::back_inserter(v));

// Fill a vector from a stream (directly rom two iterators)
using input = std::istream_iterator<int>;
input eof{};
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.

```
// 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

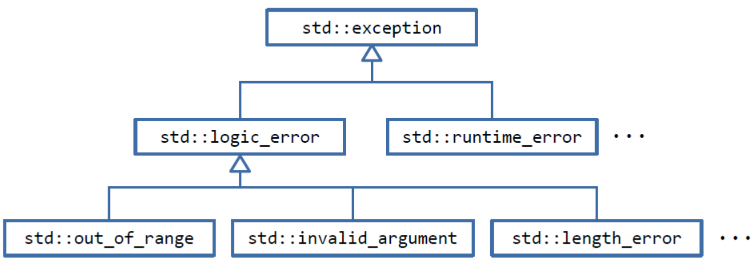
- Requires reference parameter
- Alternative: Global Variable (BAD decision)
- E.g: std::istreams's states (good(), fail()) is changed as a side-effect of input

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

```cpp
#ifndef DATE_H_
#define DATE_H_
class Date {
    int year, month, day;
public:
    Date(int year, int month, int day);

    static bool isLeapYear(int year);

private:
    bool isValidDate() const;
};
#endif /* DATE_H_ */
```

```cpp
#include "Date.h"

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

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

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.

```cpp
class Date {
public:
    Date(int year, int month, int day);
    Date(); // Default-Constructor
    Date(Date const &); // Copy-Constructor
    Date(Date &&); // Move-Constructor
    explicit Date(std::string const &); // Typeconversation-Constructor
    Date(std::initializer_list<Element> elements); // Initializer List-Constructor
    ~Date(); // Destructor
};
```

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: == , != , < , <= , > , >=

```
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.

```cpp
namespace one {
    struct type_one{};
    void f(type_one) { /* ... */ } /* 1 */
}

namespace two {
struct type_two{};
    void f(type_two) { /* ... */ } /* 2 */
    void g(one::type_one) { /* ... */ } /* 3 */
    void h(one::type_one) { /* ... */ } /* 4 */
}
void g(two::type_two) { /* ... */ } /* 5 */
```

```cpp
#include "adl.h"

int main() {
    one::type_one t1{};
    f(t1); // Function 1
    two::type_two t2{};
    f(t2); // Function 2
    h(t1); // No Function found
    two::g(t1); // Function 3
    g(t1); // No Function found
    //(5 gefunden, aber arg passt nicht)
    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

```cpp
// Unscoped enumeration
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)
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

8 Woche08

9 Woche09

10 Woche10

11 Woche11

12 Woche12

13 Woche13

14 Anhang

14.1 Übungen Woche XX

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

14.2 Übungen Woche 06

Enumeration Example

```
// 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;
}
```

14.3 Includes

```
// Only the declaration for input and output streams
#include <iosfwd>

// Implementation of input stream
#include <istream>

// Implementation of output stream
#include <ostream>

```

```cpp
// Declaration of both streams and additionally std::cout, std::cin, std::cerr
#include <iostream>

// Functions: std::tolower(c), std::isupper(c)
#include <cctype>

// Strings
#include <string>

// Arrays
#include <array>

// Vectors (ArrayList)
#include <vector>

// Iterators: std::count, std::accumulate, std::distance, std::for_each
#include <iterator>

// std::iota
#include <numeric>

// function template, which allows passing lambdas (with capture)
#include <functional>

// std::tie (creates tupel)
#include <tupel>
```