Prog3 Snippets

Emanuel Duss, Marcel Loop

2013-12-15

Contents

1 Inhalt Modul Prog3 (Klebzettelindex) 2

2 Algorithmen 3
- 2.1 all_of . . . 3
- 2.2 equal . . . 3
- 2.3 lower_bound, uppper_bound . . . 4
- 2.4 std::sort . . . 4
- 2.5 Distance . . . 4
- 2.6 Reverse . . . 4
- 2.7 swap . . . 4
- 2.8 find . . . 5
- 2.9 std::unique . . . 5
- 2.10 lexicographical_compare . . . 5

3 Templates 5
- 3.1 Template Klasse: `dynArray` . . . 5
- 3.2 lru_list . . . 7
- 3.3 Variadic Template . . . 7

4 Enumeratoren 8
- 4.1 Enum Type with Hidden Value (`Switch`) . . . 8

5 Iteratoren 9
- 5.1 Iteratoren nutzen . . . 9
- 5.2 Square-Iterator . . . 9
- 5.3 Potenz-Iterator (`potenz_iterator.h`) . . . 10
- 5.4 reciprocalsquare mit Boost . . . 10

6 Container 11
- 6.1 Vektoren und das Erase-Remove Idiom . . . 11
- 6.2 vector<double> in vector<int> füllen . . . 11
- 6.3 std::vector . . . 11
  - 6.3.1 `operator[size_type n]` vs `.at(size_type i)` . . . 11

7 Bind 11
- 7.1 Beispiel einer Subtraktion . . . 11
- 7.2 multiplikator_in_fünfen . . . 12

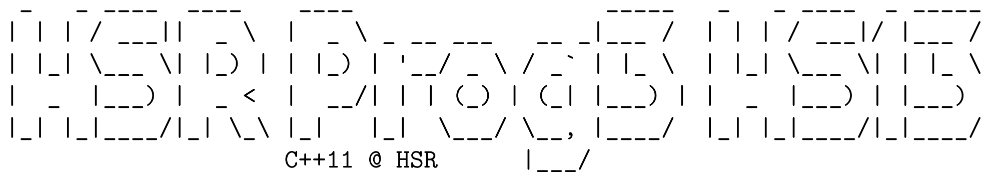
8 Const Correctness 12
- 8.1 Regeln . . . 12
- 8.2 Pointer . . . 12
- 8.3 References . . . 12
- 8.4 Funktionen . . . 12

9 Inheritance **13**
9.1 Quick Rules . . . 13
9.2 Beispiel . . . 13
9.3 Abstract Base Classes: pure virtual . . . 15

10 Diverses **15**
10.1 main(argc,argv) . . . 15
10.2 endl vs. \n . . . 15
10.3 Grossbuchstaben zählen und in Kleinbuchstaben umwandeln . . . 15
10.4 Ist es ein PalindrommordnilaP . . . 15
10.5 Wörter zählen . . . 16

11 PIMPL IDIOM **16**
11.1 Header . . . 16
11.2 Header file . . . 17

1 Inhalt Modul Prog3 (Klebzettelindex)



- Einführung in C++
- Rechnen mit Werten
  - Variabeln, const, Expressions, Typen, Referenzen, und I/O
- Einfache Sequenzen
  - std::vector, Iteratoren, algorithms, lambda
- Funktionen
  - Definieren, Lambdas, Namespaces, Using, Scopes, Rvalue References: Move (&&)
  - Pass-by(Value|Const Reference|Reference), default-Argumente
  - Parameter/Argumente, Sichtbarkeit, Lebenszeit, Garantien
  - Exceptions
- Types, Classes and Enums
  - Struct/Class, Members, Constructors, Operatoren überladen (z. B. <, <<, >>, explicit
  - Enumeratoren, Arithmetische Typen, User-defined Literals (UDL)
- Standard Container in der STL
  - Containertypen, STL Iteratoren Kategorien, Iteratoren (Looping)
  - vector, array, string, dequeue, stack, queue, (multi)set, (multi)map,
  - Prädikate, algorithms (fill, generate, iota, find, search)
- Algorithms, Functors and Parameterizing STL
  - Diverse Algorithmen, erase-remove idiom, sortieren, heap, kopieren
  - Funktoren, Lambdas, standard funktoren, Funktoren als Funktionsparameter, bind, eigene Iteratoren
- Parameterizing STL with Iterators
  - Eigene Iteratortypen, Boost Iterator Library
- Template Funktionen
  - Template Funktionen, Variadic Template Function

- Template Klassen
  - Type aliases, Template Terminology
  - Construct from Iterators and std::initializer_list
  - Template Template Parameter: Templates as Parameters, variadic Templates
- Non-type Template Parameters
  - Checking from Prime Number, SFINAE
- Dynamic Heap Memory Namagement
  - new/delete
  - std::unique_ptr, std::shared_ptr, make_shared
  - Clas hierarchies with shared_ptr
  - PIMPL Idiom
- Inheritance and dynamic Polymorphism
  - Vererben, Konstruktoren, Sichtbarkeit
  - Object Slicing, Member Hiding Problem (solving with `using`)
  - Dynamic Polymorphism: `virtual` Member Function, pure virtual
  - Liskov Substitution Principle, Multiple Inheritance
- Arrays & Pointer
  - Passing Arguments to C++ Programs mit main(argc, argv)
  - Arrays
- User Defined Literals (UDL)
  - UDL: `operator"" _udl_name(...)`, Raw UDL
- Universal References
  - Type && is not always an rvalue-reference
- Async Future
- Exam Preparation

2 Algorithmen

2.1 all_of

Mit dem Algorithmus kann man bsp. alle Elemente aus einem String oder Vector prüfen. Hier als Beispiel wird überprüft ob der String keine Zahlen enthält

```
if (!std::all_of(word.begin(), word.end(), std::isalpha)) {
    throw std::invalid_argument { "The word can only consist of alpha values "};
}
```

2.2 equal

Mit dem `equal` Algorithmus kann man bsp. prüfen ob zwei Strigns identisch sind. Beispiel

```
bool Word::stringcaselessequal(std::string const &w, std::string const &other){
    size_t len=std::min(w.size(),other.size());
    return std::equal(w.begin(),w.begin()+len,other.begin(),[](char l, char r){
        return tolower(l) == tolower(r);
    });
}
```

2.3 lower_bound, uppper_bound

Braucht eine sortierte Reihenfolge. Mit den beiden kann beispielsweise alle zweier aus einer sortierten Liste gelöscht werden

2.4 std::sort

```
bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}

  std::vector<int> myvector{5,10,2,3,4,122};

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.end;

  myclass myobject;

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction);

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);
```

2.5 Distance

Mit Distance kann zum Beispiel die Grösse eines Inputs angegeben werden Beispiel, zählt alle wörter von einem Input:

```
while(std::getline(input, str)){
    std::stringstream stream{str};
    counter += distance(std::istream_iterator<Word>{stream},std::istream_iterator<Word>{});
}
```

2.6 Reverse

Mit Reverse kann man Strings umkehren. !*Beispiel*

Input: Marcel, Ausgabe: lecraM

```
std::string reversed { word };
std::reverse(reversed.begin(), reversed.end());
std::cout << reversed;
```

2.7 swap

Mit swap können einfach Werte ausgetauscht werden. Im folgenden Szenario werden die Argumente der Parameter gewechselt

```
#include <utility> // provides std::swap
template <typename T>
void rotate3arguments(T &a, T& b, T& c){
    std::swap(a,b); // using swap avoids unnecessary copying and temporaries
    std::swap(b,c);
}
```

2.8 find

Returns the first element in the range (first, last) that statisfes specific criteria. If not found, it returns the last element.

Folgender Code überprüft ob `value` im container `v` enthalten ist und gibt den entsprechenden `bool` Wert zurück.

```
return (find(v.begin, v.end, value) != v.end));
// Bsp:
std::cout << std::boolalpha; // true / false statt 1 / 0
std::cout << "Ist 5 enthalten? " << (find(v.begin(), v.end(), 5) != v.end()) << '\n';
std::cout << "Ist 523 enthalten? " << (find(v.begin(), v.end(), 523) != v.end()) << '\n';
```

2.9 std::unique

Entfernt, doppelt aufeinander folgende Werte.

```
std::vector<int> v2{10,20,20,20,30,30,20,20,10};
std::vector<int>::iterator it;

it = std::unique(v2.begin(), v2.end());
v2.resize(std::distance(v2.begin(),it));

std::copy(v2.begin(), v2.end(), std::ostream_iterator<int> (std::cout)); //Ausgabe 10,20,30,20,10
```

Bei einer einfachen `list` kann einfach `liste.unique` verwendet werden!

2.10 lexicographical_compare

```
bool Word::operator< (Word const& r) const {
    return lexicographical_compare(word.begin(),word.end(),r.word.begin(),
        r.word.end(),[](char c1,char c2){
            return std::tolower(c1)<std::tolower(c2);
    });
}
```

3 Templates

3.1 Template Klasse: `dynArray`

```
#ifndef DYNARRAY_H_
#define DYNARRAY_H_

#include <initializer_list>
#include <vector>

template<typename T> class dynArray {
    using arrayType = std::vector<T>;
    arrayType theArray;

    using value_type = typename arrayType::value_type;
    using reference = typename arrayType::reference;
    using const_reference = typename arrayType::const_reference;
    using pointer = typename arrayType::pointer;
    using const_pointer = typename arrayType::const_pointer;
    using iterator = typename arrayType::iterator;
```

```
    using const_iterator = typename arrayType::const_iterator;
    using reverse_iterator = typename arrayType::reverse_iterator;
    using const_reverse_iterator = typename arrayType::const_reverse_iterator;
    using difference_type = typename arrayType::difference_type;
    using size_type = typename arrayType::size_type;

    size_type getRealAccessElement(int i){
        if (i < 0) return size() + i;
        else return i;
    }

public:
    explicit dynArray() = default;
    explicit dynArray (size_type n):theArray(n){};
    dynArray(std::initializer_list<T> il):theArray{il}{};
    dynArray (const dynArray& x):theArray{x.theArray}{ };
    dynArray (dynArray&& x):theArray{std::move(x.theArray)}{};

    dynArray& operator= (const dynArray& x){theArray=x.theArray;};
    dynArray& operator= (dynArray&& x){theArray=x.theArray;};
    dynArray& operator= (std::initializer_list<value_type> il){theArray = il;};

    iterator begin() noexcept{ return theArray.begin(); }
    const_iterator begin() const noexcept{ return theArray.begin(); }
    iterator end() noexcept{ return theArray.end(); }
    const_iterator end() const noexcept{ return theArray.end(); }
    reverse_iterator rbegin() noexcept{ return theArray.rbegin(); }
    const_reverse_iterator rbegin() const noexcept{ return theArray.crbegin(); }
    reverse_iterator rend() noexcept{ return theArray.rend(); }
    const_reverse_iterator rend() const noexcept{ return theArray.rend(); }
    const_iterator cbegin() const noexcept{ return theArray.cbegin(); }
    const_iterator cend() const noexcept{ return theArray.cend(); }
    const_reverse_iterator crbegin() const noexcept{ return theArray.rbegin(); }
    const_reverse_iterator crend() const noexcept{ return theArray.crend(); }

    size_type size() const { return theArray.size();}
    void resize(size_type n){ theArray.resize(n); };
    void resize(size_type n, const value_type& val){ theArray.resize(n,val); }
    void clear() { theArray.clear();}
    size_type capacity() const noexcept{ return theArray.capacity();}
    bool empty() const { return theArray.empty();}

    // Element Access
    reference operator[] (size_type n) { return theArray[getRealAccessElement(n)];}
    const_reference operator[] (size_type n) const { return theArray[getRealAccessElement(n)];}
    reference at (size_type n) { return theArray.at(getRealAccessElement(n));}
    const_reference at (size_type n) const { return theArray.at(getRealAccessElement(n));}
    reference front() { return theArray.front();}
    const_reference front() const { return theArray.front();}
    reference back() { return theArray.back();}
    const_reference back() const { return theArray.back();}
    const value_type* data() const noexcept { return theArray.data();}
};
#endif
```

3.2 lru_list

```
#ifndef QUICK_ACCESS_LIST_H_
#define QUICK_ACCESS_LIST_H_

#include <vector>

template <typename T>
class quick_access_list {
    using vector_type = std::vector<T>;
    using size_type = typename vector_type::size_type;
    vector_type v;
public:
    using iterator = typename vector_type::iterator; // MUSS PUBLIC SEIN!!!
    using const_iterator = typename vector_type::const_iterator;

    quick_access_list() = default;
    template <typename input_iterator>
    explicit quick_access_list(input_iterator begin, input_iterator end){insert(begin, end); }

    iterator begin() { return v.begin(); }
    const_iterator begin() const { return v.begin(); }
    iterator end() { return v.end(); }
    const_iterator end() const { return v.end(); }

    bool empty() const { return v.empty(); }
    size_type size() const { return v.size(); }

    T front() const { return v.at(0); }

    bool contains(T e) const { return(find(v.begin(), v.end(), e) != v.end()); }

    void insert(T e) {
        v.erase(std::remove(v.begin(), v.end(), e), v.end());
        v.insert(v.begin(), e);
    }
    template <typename input_iterator>
    void insert(input_iterator begin, input_iterator end) {
        for_each(begin, end, [&](T x){ insert(x); });
    }

    template<typename U>
    std::vector<U> as_vector(){
        return std::vector<U>(v.begin(), v.end());
    }
};
#endif
```

3.3 Variadic Template

```
template <typename Arg>
void readln(std::istream &in, Arg& arg){
    getline(in,arg);
}
template<typename Head, typename ...Tail>
void readln(std::istream &in, Head& head, Tail& ...tail){
    in >> head;
```

```
    readln(in, tail...);
}
```

4 Enumeratoren

4.1 Enum Type with Hidden Value (`Switch`)

switch.h

```
#ifndef SWITCH_H_
#define SWITCH_H_

#include <iostream>
#include <vector>

struct Switch {
    Switch();
    void pressButton();
    std::string status();
private:
    enum class State : unsigned short;
    State state;
};

std::ostream& operator <<(std::ostream& out, Switch s);

#endif
```

switch.cpp

```
#include "switch.h"

enum class Switch::State : unsigned short { off, on, blinking };

Switch::Switch() : state{State::off}{}

void Switch::pressButton(){
    switch(state){ // Uff! break; nicht vergessen!!
    case State::off:
        state=State::on; break;
    case State::on:
        state=State::blinking; break;
    case State::blinking:
        state=State::off; break;
    }
}

std::string Switch::status(){
    switch(state) {
    case State::off:
        return "off"; break;
    case State::on:
        return "on"; break;
    case State::blinking:
        return "blinking"; break;
    }
}
```

```
std::ostream& operator <<(std::ostream& out, Switch s){
    out << s.status();
    return out;
}
```

main.cpp

```
#include "switch.h"
int main(){
    Switch myswitch{};
    myswitch.pressButton();
    // myswitch.status();
    myswitch.pressButton();
    // std::cout << myswitch.status();
    std::cout << myswitch;
}
```

5 Iteratoren

5.1 Iteratoren nutzen

- `set<char>::iterator iterator = s.begin(); ++it; std::cout << *it;`
- Ab C++11: `auto iterator = s.begin();`
- `std::list<foo>`: nach `.end()` folgt wieder `.begin()`

5.2 Square-Iterator

```
#ifndef SQUARES_ITERATOR_H_
#define SQUARES_ITERATOR_H_
#include <iterator>

struct squares
: std::iterator<std::input_iterator_tag,int>
{
  explicit squares(value_type start=0):value{start}{}
  bool operator==(squares const &r) const {
    return value == r.value;
  }
  bool operator!=(squares const &r) const {
    return !(*this == r);
  }
  value_type operator*() const { return value*value; }
  squares &operator ++() {
    ++value;
    return *this;
  }
  squares operator ++(int) {
    auto old= *this;
    ++(*this);
    return old;
  }
private:
  value_type value;
};

#endif
```

5.3 Potenz-Iterator (potenz_iterator.h)

```
#ifndef POTENZ_ITERATOR_H_
#define POTENZ_ITERATOR_H_

#include <stdexcept>
#include <boost/operators.hpp>

struct potenz_iterator : boost::input_iterator_helper<potenz_iterator, double> {
    explicit potenz_iterator(value_type base = 1.0, int n = 1):base{base}, n{n}{
        if (n < 0 || base <= 0.0)
            throw std::logic_error("zero");
        else
            value = pow(base, n);
    }

    bool operator==(potenz_iterator const &r) const {
        return base == r.base && n == r.n;
    }
    value_type operator*() const { return value; }

    potenz_iterator &operator ++() {
        value *= base;
        ++n;
        return *this;
    }

private:
    value_type value;
    double const base;
    int n{};

    double pow(double b, unsigned int n){
        if (n > 0)
            return (b*pow(b, n-1));
        else
            return 1;
    }
};

#endif
```

5.4 reciprocalsquare mit Boost

```
#include <boost/iterator/counting_iterator.hpp>
#include <boost/iterator/transform_iterator.hpp>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

int main(){
  using std::placeholders::_1;
  auto reciprocalsquare=bind(std::divides<double>{},1,bind(std::multiplies<double>{},_1,_1));
  std::vector<double> v{boost::make_transform_iterator(boost::make_counting_iterator(1),
      reciprocalsquare), boost::make_transform_iterator(boost::make_counting_iterator(11),
          reciprocalsquare)};
```

```
  copy(v.begin(),v.end(),std::ostream_iterator<double>{std::cout,"\n"});
  auto b=boost::make_transform_iterator(boost::make_counting_iterator(11),reciprocalsquare);
  auto e=boost::make_transform_iterator(boost::make_counting_iterator(21),reciprocalsquare);
  copy(b,e,std::ostream_iterator<double>{std::cout,"\n"});
}
```

6 Container

6.1 Vektoren und das Erase-Remove Idiom

```
std::vector<int> v{1,2,3,4,5};
v.erase(remove(v.begin(), v.end(), 3),v.end()); // Element "3" löschen
v.insert(v.begin(), 3); // Element "3" am Anfang einfügen
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, "\n"));
```

6.2 vector<double> in vector<int> füllen

```
std::vector<double> v1{5.23, 2.3, 5.0};
std::vector<int> v2(v1.size());
std::copy(v.begin(), v.end(), v2.begin()); //  Castet selber
```

Und noch einfacher (am Beispiel einer Template Funktion):

```
template<typename U>
std::vector<U> as_vector(){
    return std::vector<U>(v.begin(), v.end());
}
```

6.3 std::vector

6.3.1 `operator[size_type n]` vs `.at(size_type i)`

```
int value1{};
int value2{};

std::vector<int> v3{10, 20};

value1 = v3[3];
value2 = v3.at(3)
```

Der Vorteil von `at` ist, dass er bei einem Zugriff auf ein ungültiges Element eine Excepetion ausgibt.

7 Bind

7.1 Beispiel einer Subtraktion

```
#include <functional>   // std::bind

using std::placeholders::_1;
using std::placeholders::_2;

// Bind für (_1 * 2 ) - (_2/3) = 17
auto bindSubtraction=std::bind(std::minus<double>{},
    std::bind(std::multiplies<double>{}, 2, _1),
```

```
    std::bind(std::divides<double>{}, _2, 3)
);

//Beispiel der Rechnung (10 * 2 ) - (9/3) = 17
ASSERT_EQUAL(17, bindSubtraction(10,9));
```

7.2 multiplikator_in_fünfen

```
using std::placeholders::_1;
auto multi_5 = std::bind(std::multiplies<double>{}, _1, 5.0);
transform(start, end, std::ostream_iterator<double>(std::cout, "\n"), multi_3);
```

8 Const Correctness [1]

8.1 Regeln

- `const` als Rückgabewert einer Funktion ist sinnfrei.
- `const` als Rückgabewert eines Konstruktors ist falsch, da diese niemals einen Rückgabewert besitzen.
- `const struct Foo` ist falsch (Klassen sind nicht `const`)
- `struct Bar : const Foo` ist falsch (zugriff `const` gibt es nicht).

8.2 Pointer

```
void Foo( int * ptr,
          int const * ptrToConst,
          int * const constPtr,
          int const * const constPtrToConst ){
    *ptr = 0; // OK: modifies the "pointee" data
    ptr  = NULL; // OK: modifies the pointer

    *ptrToConst = 0; // Error! Cannot modify the "pointee" data
    ptrToConst  = NULL; // OK: modifies the pointer

    *constPtr = 0; // OK: modifies the "pointee" data
    constPtr  = NULL; // Error! Cannot modify the pointer

    *constPtrToConst = 0; // Error! Cannot modify the "pointee" data
    constPtrToConst  = NULL; // Error! Cannot modify the pointer
}
```

8.3 References

```
int i = 22;
int const & refToConst = i; // OK
int & const constRef = i; // Error the "const" is redundant
```

8.4 Funktionen

```
class C{
    int i;public:
    int Get() const // Note the "const" tag
```

[1] Aus Wikipedia https://en.wikipedia.org/wiki/Const-correctness

```
      { return i; }
    void Set(int j) // Note the lack of "const"
      { i = j; }
};
 void Foo(C& nonConstC, const C& constC){
    int y = nonConstC.Get(); // Ok
    int x = constC.Get();    // Ok: Get() is const
    nonConstC.Set(10); // Ok: nonConstC is modifiable
    constC.Set(10); // Error! Set() is a non-const method and constC is a const-qualified object
}
```

9 Inheritance

9.1 Quick Rules

- Virtuelle Methoden sollten einen viruelen Destruktor haben.
- Nur virtuelle Methoden sollten in abgeleiteter Klasse überschrieben werden

```
Dog dog{};
Animal animal = dog;
animal.move(); // Es wird immer die Funktion move() von Animal ausgeführt!
Animal &animal2 = dog; // Jetzt wird animal2 wie ein Dog behandelt. Jetzt auf virtual achten!
```

- Falls eine Funktion virtual ist, wird die spezialisiertere Funktion aufgerufen.
- Eine Funktion die in der Basisklasse `virtual` ist, bleibt auch in den abgeleiteten Klassen virtual.
- Mehrfache Ableitung: Der Reihe nach; (De|Kon)struktor wird von der "Ober"-Basisklasse mehrfach aufgerufen.
- Konflikte bei Mehrfachvererbung falls Basisklasse Membervariabeln oder Virtuelle Memberfunktionen hat (Namenskonflikte). Sollte vermieden werden.
- Zu tiefe Hierarchien sind nicht empfohlen.

9.2 Beispiel [2]

animal.h

```
#ifndef ANIMALS_H_
#define ANIMALS_H_

#include <iostream>

struct Animal {
    Animal() { std::cout << "Animal: Constructor\n"; }
    ~Animal() { std::cout << "Animal: Deconstructor\n"; }
    void makeSound() { std::cout << "Animal: makeSound()\n"; }
    virtual void move() { std::cout << "Animal: move()\n"; }
};

struct Bird : Animal {
    Bird() { std::cout << "Bird: Constructor\n"; }
    ~Bird() { std::cout << "Bird: Deconstructor\n"; }
    virtual void makeSound() { std::cout << "Bird: makeSound()\n"; }
    void move() { std::cout << "Bird: move()\n"; }
};

struct Duck : Bird {
```

[2] Vorlesungsfolien

```
    Duck() { std::cout << "Duck: Constructor\n"; }
    ~Duck() { std::cout << "Duck: Deconstructor\n"; }
    void makeSound() { std::cout << "Bird: makeSound()\n"; }
    virtual void move() { std::cout << "Bird: move()\n"; }
};

#endif
```

main.cpp

```
#include "animal.h"
int main() {
    std::cout << "[*] Constructors\n";
    std::cout << "[+] Duck duck;\n";
    Duck duck;
    std::cout << "[+] Bird bird = duck;\n";
    Bird bird = duck;
    std::cout << "[+]  Animal & animal = duck;\n";
    Animal & animal = duck;

    std::cout << "\n[*] makeSound()\n";
    std::cout << "[+] duck.makeSound();\n";
    duck.makeSound();
    std::cout << "[+] bird.makeSound();\n";
    bird.makeSound();
    std::cout << "[+] animal.makeSound();\n";
    animal.makeSound();

    std::cout << "\n[*] move()\n";
    std::cout << "[+] duck.move();\n";
    duck.move();
    std::cout << "[+] bird.move();\n";
    bird.move();
    std::cout << "[+] animal.move();\n";
    animal.move();

    std::cout << "\n[*] Deconstructors\n";
}
```

Output

```
[*] Constructors
[+] Duck duck;
Animal: Constructor
Bird: Constructor
Duck: Constructor
[+] Bird bird = duck;
[+]  Animal & animal = duck;

[*] makeSound()
[+] duck.makeSound();
Bird: makeSound()
[+] bird.makeSound();
Bird: makeSound()
[+] animal.makeSound();
Animal: makeSound()

[*] move()
[+] duck.move();
Bird: move()
```

```
[+] bird.move();
Bird: move()
[+] animal.move();
Bird: move()

[*] Deconstructors
Bird: Deconstructor
Animal: Deconstructor
Duck: Deconstructor
Bird: Deconstructor
Animal: Deconstructor
```

9.3 Abstract Base Classes: pure virtual

```
struct AbstractBase {
    // Wichtig: Basisklassen mit virtual Members, brauchen einen Virtuellen Destruktor!
    virtual ~AbstractBase(){}
    // Pure Virtual Member Function = No Implementation provided
    virtual void function()=0;
};
```

10 Diverses

10.1 main(argc,argv)

Programm mit Parameter aufrufen:

```
int main(int argc, char *argv[]){
     copy(argv+1, argv+argc,
       std::ostream_iterator<std::string>{std::cout,"-"});
}
```

10.2 endl vs. \n

- Stackoverflow meint: “endl is good, because if the program crashes (flösch böffer), your stuff gets printed. BUT, there is a performance hit with endl.”

10.3 Grossbuchstaben zählen und in Kleinbuchstaben umwandeln

```
int countingToLower(std::string& s){
    size_t counter{};
    for (size_t i{}; i < s.size(); ++i){
        if(isupper(s[i])){
            ++counter;
            s[i] = tolower(s[i]);
        }
    }
    return counter;
}
```

10.4 Ist es ein PalindrommordnilaP

```
bool is_palindrome(std::string str){
    return std::equal(str.rbegin(), str.rend(), str.begin(), [](char s1, char s2){
```

```
        return std::tolower(s1) == std::tolower(s2);
    });
}
void find_palindrome(std::istream& input, std::ostream& output){
    std::ostream_iterator<std::string> out{output, "\n"};
    std::istream_iterator<std::string> in{input};
    std::istream_iterator<std::string> eof{};

    std::copy_if(in, eof, out, is_palindrome);
}
```

10.5 Wörter zählen

```
/*
 * wcount
 * output how many words exist in standard input
 */
using input_word_iterator = std::istream_iterator<Word>;
int wcount(std::istream& in){
    return std::distance(input_word_iterator{in}, input_word_iterator{});
}

/*
 * wdiffcount
 * output how many different words exist in standard input
 */
int wdiffcount(std::istream& in){
    std::set<Word> words{};
    std::copy(input_word_iterator{in}, input_word_iterator{}, std::inserter(words, words.begin()));
    return words.size();
}
```

11 PIMPL IDIOM

11.1 Header

```
#include "Book.h"
//PIMPL-Klasse
//diese versteckt die Implementierung vor der eigentlichen Klasse
//auf der der Programmierer zugriff hat.

class BookImpl{
public:
    BookImpl() {
}
void addContent(unsigned int number, std::string text){
    auto f = cont.find(number);
    if(f != cont.end()){
        cont[number] += text;
    }else {cont.insert(std::pair<unsigned int, std::string>(number, text));
}

}
std::string getChapter (unsigned int number){
auto f = cont.find(number);
if(f != cont.end()){
```

```
    return cont[number];
} else {
return "invalid chapter number";
}

}
private:
std::map<unsigned int, std::string> cont;
};

Book::Book() : bookImpl{std::make_shared<BookImpl>()}{
}
void Book::addContent(unsigned int number, std::string text){
    bookImpl->addContent(number, text);
}
std::string Book::getChapter(unsigned int number){
    return bookImpl->getChapter(number);
}
Book::~Book() {
}
```

11.2 Header file

```
#ifndef BOOK_H_
#define BOOK_H_
#include <map>
#include <string>
#include <memory>

class Book {
public:
    Book();
virtual ~Book();
void addContent(unsigned int number, std::string text);
        std::string getChapter(unsigned int number);

private:
    std::shared_ptr<class BookImpl> bookImpl;
};
#endif /* BOOK_H_ */
```