Zusammenfassung

C++

Armend Lesi & Marco Endres

Hochschule für Technik Rapperswil

22. Februar 2020

Lizenz

Inhaltsverzeichnis

1 Trivia

1.1 Advance vs. Next

std::advance

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

std::next

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

1.2 API

Vector
- at, [ ], front, back, empty, size, clear, insert, erase, push_back, pop_back
Set
- empty, size, clear, insert, erase, count, find, contains
Map
- at, [ ], empty, size, clear, insert, erase, count, find, contains
Multimap
- empty, size, clear, insert, erase, count, find, contains

1.3 Iterators

```
auto it1 = set.begin(); // std::set::const_iterator
auto it2 = string.crend(); // std::string::const_reverse_iterator
auto it3 = string.end(); // std::string::iterator
auto it4 = set.end(); // std::set::const_iterator
```

```
vector<char> content{’S’,’t’,’a’,’c’,’k’,’o’,’v’,’e’,’r’,’f’,’l’,’o’,’w’};
auto it1 = begin(content);
cout << *(++it1); // it1 inkrementieren, dann dealozieren => t
cout << ++(*it1); // it1 dealozieren, dann Buchstaben inkrementieren => ++S => T
```

```
vector<char> content{’S’,’t’,’a’,’c’,’k’,’o’,’v’,’e’,’r’,’f’,’l’,’o’,’w’};
auto it1 = begin(content);
++it1; // it1 zeigt auf 2. Position im vector
sort(begin(content), end(content));
std::cout << *it1; // it1 immer noch auf 2 Pos. => ’a’ (sortiert)
```

```
vector<char> content{'S','t','a','c','k','o','v','e','r','f','l','o','w'};
// Sacefkloortvw
auto it2 = remove(begin(content), end(content), 'o'); // Sacefklrtvwoo, it2 zeigt auf
    1. 'o'
cout << distance(it2, end(content)); // 2
content.erase(it2); // loest nur 1. 'o', sonst muss von-bis angegeben werden
cout << content.size(); // 12
```

1.4 Output with Copy

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

1.5 Transform

```
std::transform(begin(counts), end(counts), begin(letters),
std::back_inserter(combined), [](int i, char c) {return std::string(i, c);});
```

```
//transform over set with inserter
void test() {
   using namespace std;
   string const input("Test");
   using out = std::ostream_iterator<char>;
   set<char> s{};

   std::transform(begin(input), end(input), inserter(s, s.begin()),::toupper);
   copy(begin(s), end(s), out(cout, "-"));
}
```

```
//transform over 2 iterators
using namespace std;
using out = ostream_iterator<string>;
transform(word.begin(), word.end(), values.begin(), out{cout, "\n"}, formatOutput);
```

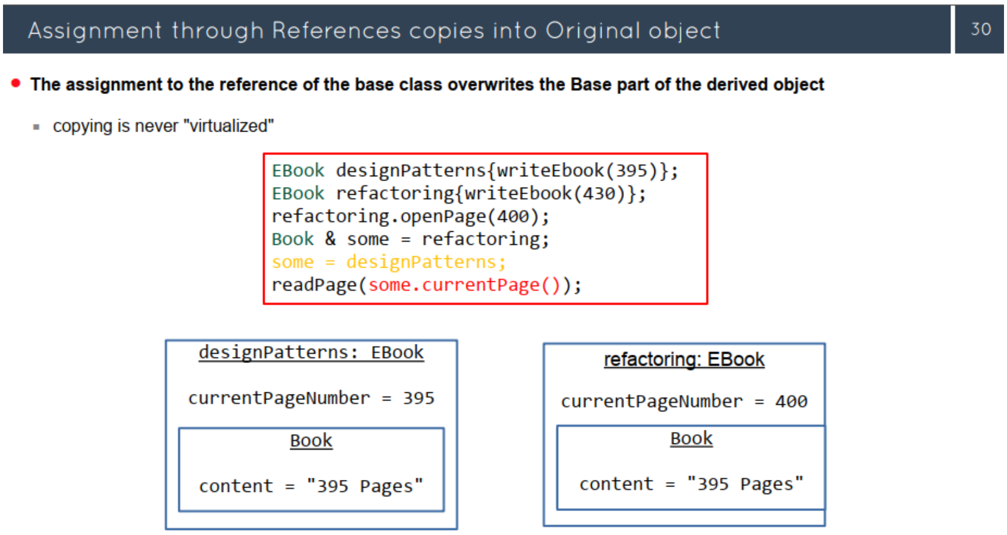
1.6 Accumulate

```
#include <algorithm>
#include <iterator>

transform(word.begin(), word.end(), back_inserter(values), toLetterValue);

using out = ostream_iterator<string>;

transform(word.begin(), word.end(), values.begin(), out{cout, "\n"}, formatOutput);

accumulate(values.begin(), values.end(), 0)
```

1.7 Destructors (non-virtual) with virtual members are a design error

```cpp
// Output:
// put into trash
struct Fuel {
	virtual void burn() = 0;
	/* virtual */ ~Fuel() { std::cout << "put into trash\n"; }
};
struct Plutonium : Fuel {
	void  burn() { std::cout << "split core\n"; }
	~Plutonium() { std::cout << "store many years\n"; }
};
int main() {
	std::unique_ptr<Fuel> surprise = std::make_unique<Plutonium>();
}
```

1.8 Assignment through References copies into Original object



Assignment through References copies into Original object 30

- **The assignment to the reference of the base class overwrites the Base part of the derived object**
  - copying is never "virtualized"

```cpp
EBook designPatterns{writeEbook(395)};
EBook refactoring{writeEbook(430)};
refactoring.openPage(400);
Book & some = refactoring;
some = designPatterns;
readPage(some.currentPage());
```

2 Histogram

```
#ifndef HISTOGRAM_H_
#define HISTOGRAM_H_
#include <map>

template<typename T>
struct Histogram {
   void insert (T const key) {
      ++m[key];
   }

   unsigned count (T const key) const {
      return m.find(key) != m.end() ? m[key] : 0u;
   }
   // oder
   unsigned count(T const key) const{
      auto result = myMap.find(key);
      if (result == myMap.end()) {
         return 0u;
      } else {
         return result->second;
      }
   }

private:
    std::map<T, unsigned> m{};
}
#endif
```

2.1 HistoramEntry

```
#ifndef HISTOGRAMENTRY_H_
#define HISTOGRAMENTRY_H_
#include "Word.h"
#include <algorithm>
#include <boost/operators.hpp>
struct HistogramEntry :boost::less_than_comparable<HistogramEntry>,
    boost::equality_comparable<HistogramEntry> {
   HistogramEntry (Word w, int amount);

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

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

   inline bool operator ==(HistogramEntry const & lhs, HistogramEntry const & rhs) {
      return lhs.amount == rhs.amount && lhs.word == rhs.word;
   }
   inline std::ostream& operator<<(std::ostream &out, HistogramEntry const &
       histogram) {
      histogram.word.print(out);
      out << ": " << histogram.amount;
      return out;
   }
private:
   Word word;
   int amount;
};
#endif /* HISTOGRAMENTRY_H_ */
```

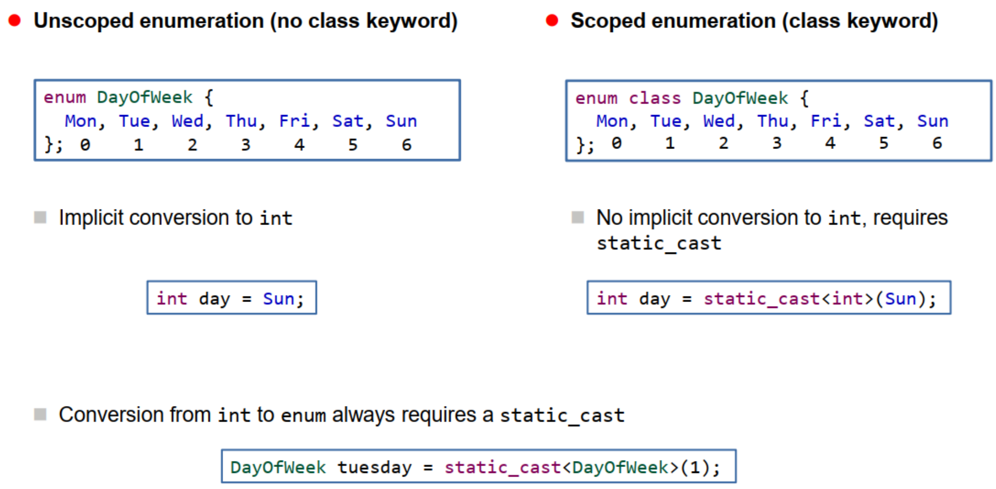
3 Word

```cpp
#ifndef WORD_H_
#define WORD_H_

#include <algorithm>
#include <cctype>
#include <iterator>
#include <string>
#include <ostream>

namespace text {

struct Word {
   Word();
   explicit Word(std::string const & value);
   void read(std::istream &is);
   void print(std::ostream & os) const;

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

   inline bool operator ==(Word const & rhs) const {
      return std::equal(
         std::begin(this->value), std::end(this->value),
         std::begin(rhs.value), std::end(rhs.value),
         [](const char l, const char r) {
            return std::tolower(l) == std::tolower(r);
         }
      );
   }

   bool operator>(Word const & other) const {
      return (other < *this);
   }

   bool operator<=(Word const & other) const {
      return !(other < *this);
   }

   bool operator>=(Word const & other) const {
      return !(*this < other);
   }

   bool operator!=(Word const & other) const {
      return !(*this == other);
   }
private:
   std::string value;
   bool isValid(std::string const & value);
};

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

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

#endif /* WORD_H_ */
```

```
#include "word.h"

#include <iterator>
#include <algorithm>
#include <cctype>
#include <stdexcept>
#include <string>

using text::Word;

Word::Word() : value{"default"} {
}

void Word::read(std::istream &is) {
   if(is.good()) {
      using iter = std::istreambuf_iterator<char>;
      iter input{is};
      iter eof{};
      auto firstChar = std::find_if(input, eof, ::isalpha);
      std::string readWord{};
      std::find_if(firstChar, eof, [&readWord](char c) {
         bool isWordFinished {!std::isalpha(c)};
         if (!isWordFinished) {
            readWord += c;
         }
         return isWordFinished;
      });
      if (isValid(readWord)) {
         value = readWord;
      } else {
         is.setstate(std::ios_base::failbit);
      }
   }
}

Word::Word(std::string const & value) : value { value } {
   if (!isValid(value)) {
      throw std::invalid_argument{"Word isn't valid"};
   }
}

bool Word::isValid(std::string const & value) {
   return !value.empty() && std::all_of(std::begin(value), std::end(value),
       ::isalpha);
}

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

4 ENUM

```cpp
namespace calendar {
   enum class DayOfWeek {
      Mon, Tue, Wed, Thu, Fri, Sat, Sun
   };
}
bool is_weekend(calendar::DayOfWeek day) {
   return day == calendar::DayOfWeek::Sat ||
      day == calendar::DayOfWeek::Sun;
}
```



- **Unscoped enumeration (no class keyword)**

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

- Implicit conversion to `int`

```cpp
int day = Sun;
```

- **Scoped enumeration (class keyword)**

```cpp
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`

```cpp
int day = static_cast<int>(Sun);
```

- Conversion from `int` to `enum` always requires a `static_cast`

```cpp
DayOfWeek tuesday = static_cast<DayOfWeek>(1);
```

5 Vectorset

```cpp
#ifndef VECTORSET_H_
#define VECTORSET_H_
#include <vector>
#include <set>
#include <functional>
#include <algorithm>

template <typename T, typename COMPARE=std::less<T>>
struct vectorset : public std::vector<T> {

  using vectorType = std::vector<T>;
  using vectorType::vectorType;

// Aliases
  using size_type = typename vectorType::size_type;
  using reference = typename vectorType::reference;
  using const_reference = typename vectorType::const_reference;
  using iterator = typename vectorType::iterator;
  using const_iterator = typename vectorType::const_iterator;

   vectorset() = default;

   explicit vectorset(std::initializer_list<T> li) : vectorType{li} {
      std::sort(this->begin(), this->end(), COMPARE());
   }

   template <typename ITER>
   vectorset(ITER b, ITER e) : vectorType(b, e) {
     std::sort(this->begin(), this->end(), COMPARE());
   }

   template <typename Elt>
   explicit operator std::multiset<Elt>() const{
      return std::multiset<Elt>(this->begin(), this->end());
   }

// Functions
  const_iterator find(T const key) const {
      return std::find_if(this->cbegin(), this->cend(), [&key](const T &entry) {
            COMPARE comp{};
            return !comp(key, entry) && !comp(entry, key);
         });
      // return std::find_if(this->cbegin(), this->cend(), [](const T &e) { return e == key; });
      // is equivalent to: return std::find(this->cbegin(), this->cend(), key)
  }

  size_type count(T const key) const {
      return std::count_if(this->cbegin(), this->cend(), [&key](const T &entry) {
            COMPARE comp{};
            return !comp(key, entry) && !comp(entry, key);
      });
  }

  std::multiset<T, COMPARE> asMultiset() {
      return std::multiset<T, COMPARE> (this->cbegin(), this->cend());
  }
};
#endif
```

6 Indexable Set

```cpp
#ifndef INDEXABLESET_H_
#define INDEXABLESET_H_

#include <set>
#include <stdexcept>
#include <algorithm>

template<typename T, typename COMPARE=std::less<T>>
struct indexableSet : std::set<T, COMPARE> {
   using container = std::set<T, COMPARE>;
   using container::container;
   using difference_type = typename container::difference_type;
   using const_reference = typename container::const_reference;

   const_reference at(difference_type index) const {
      if (index < 0) {
         const long long unsigned int absIndex = abs(index);
         if (absIndex > this->size()) {
            throw std::out_of_range(absIndex + " is out of range");
         }
         return *std::prev(this->cend(), absIndex);
      } else {
         if (static_cast<long long unsigned int>(index) >= this->size()) {
            throw std::out_of_range(index + " is out of range");
         }
         return *std::next(this->cbegin(), index);
      }
   }

   const_reference operator[](difference_type index) const {
      return this->at(index);
   }

   const_reference front() const {
      return this->at(0);
   }

   const_reference back() const {
      return this->at(-1);
   }
};

#endif /* INDEXABLESET_H_ */
```

7 Deck

```cpp
#ifndef DECK_H
#define DECK_H
#include <deque>
#include <algorithm>
#include <stdexcept>
#include <random>
#include <iterator>

template <typename T>
class Deck {
   using container = std::deque<T>;
   container c;
   using size_type = typename container::size_type;
   using const_iterator = typename container::const_iterator;
   using const_reverse_iterator = typename container::const_reverse_iterator;
   using const_reference = typename container::const_reference;
public:
   Deck() = default;
   explicit Deck(std::initializer_list<T> li) : c{li} {
      this->shuffle();
   }
   template <typename ITER>
   Deck(ITER b, ITER e) : c(b, e) {
      this->shuffle();
   }

   size_type size() const { return c.size(); }
   bool empty() const { return c.empty(); }
   const_reference front() const {
      checkContainer();
      return c.front();
   }

   const_reference back() const {
      checkContainer();
      return c.back();
   }

   void push_back(T const elem) {
      c.push_back(elem);
      shuffle();
   }

   void pop_front() {
      checkContainer();
      c.pop_front();
   }

   void shuffle() {
      std::random_device rd;
      std::mt19937 g(rd());
      std::shuffle(c.begin(), c.end(), g);
   }

   void checkContainer() const {
      if (c.empty()) {
         throw std::out_of_range{"Out of range"};
      }
   }
```

```

    // Iteratoren
    const_iterator begin() const { return c.begin(); }
    const_iterator cbegin() const { return c.cbegin(); }

    const_iterator end() const { return c.end(); }
    const_iterator cend() const { return c.cend(); }

    const_reverse_iterator rbegin() const { return c.rbegin(); }
    const_reverse_iterator crbegin() const { return c.crbegin(); }

    const_reverse_iterator rend() const { return c.rend(); }
    const_reverse_iterator crend() const { return c.crend(); }
};
#endif
```

8 Sack

8.1 Iterator constructors

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

template <typename T>
class Sack {
//...
public:
      template <typename Iter>
      Sack(Iter begin, Iter end) :  theSack(begin, end) {}
      //...
};
```

```
// Retain default constructor
Sack() = default;
```

8.2 Initializer list constructors

```
Sack(std::initializer_list<T> values) : theSack(values) {}
```

8.3 Extracting a std::vector

8.3.1 Usage

```
//explicit conversion operator
Sack<unsigned> aSack{1, 2, 3};
auto  values = static_cast<std::vector<unsigned>>(aSack);
```

```
//member function
Sack<unsigned> aSack{1, 2, 3};
auto  values = aSack.asVector();
auto  doubleValues = aSack.asVector<double>();
```

8.3.2 Implementation

```
//explicit conversion operator
template <typename Elt>
explicit operator std::vector<Elt>() const {
      return std::vector<Elt>(begin(theSack), end(theSack));
}
```

```
//member function
template <typename Elt = T>
auto  asVector() const {
      return std::vector<Elt>(begin(theSack), end(theSack));
}
```

8.4 Deduction Guide

```
template <typename Iter>
Sack(Iter begin, Iter end) -> Sack<typename std::iterator_traits<Iter>::value_type>;
```

8.5 Template Specialization

```
//Partial Specialization
template <typename T>
struct Sack<T *>;
// Explicit Specialization
template <>
struct Sack<char const *>;
```

9 Espresso

9.1 espresso.h

```
#ifndef ESPRESSO_H_
#define ESPRESSO_H_

#include <iosfwd>
namespace Coffee {

enum class Aroma {
   Cosi, Dharkan, Fortissio, Kazaar, Livanto, Water
};

struct Espresso {
   Aroma aroma{Aroma::Water};

   Espresso() = default;
   explicit Espresso (Aroma aroma);

   bool operator==(Espresso const &other) const;
   bool operator!=(Espresso const &other) const;
   friend std::istream & operator>>(std::istream & in, Espresso & e);


};
}
#endif /* ESPRESSO_H_ */
```

9.2 espresso.cpp

```cpp
#include <string>
#include <istream>
#include <map>
#include "espresso.h"

namespace Coffee {

using namespace std::string_literals;

std::map<std::string, Aroma> const aromaNames {
   {"Cosi"s, Aroma::Cosi},{"Dharkan"s, Aroma::Dharkan},
   {"Fortissio"s, Aroma::Fortissio},{"Kazaar"s, Aroma::Kazaar},
   {"Livanto"s, Aroma::Livanto},{"Water"s, Aroma::Water},
};

Espresso::Espresso(Aroma aroma) : aroma{aroma}{};

bool Espresso::operator ==(Espresso const &other) const {
   return this->aroma == other.aroma;
}

bool Espresso::operator !=(Espresso const &other) const {
   return !(*this == other);
}

std::istream & operator>>(std::istream & in, Espresso & e){
   std::string stringAroma{};
   if (in >> stringAroma) {
      auto const aroma = aromaNames.find(stringAroma);
      if (aroma == aromaNames.end()){
         in.setstate(std::ios_base::failbit);
         return in;
      } else {
         e = Espresso{aroma->second}; // otherwise: e.aroma = aroma->second;
         return in;
      }
   } else {
      return in;
   }
}
}
```

10 Functor

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

struct caselessless {
   bool operator()(char const c1, char const c2) const {
      return std::tolower(c1) < std::tolower(c2);
   }
};

void teilB(){
   using namespace std;
   string kasten("OachkatzlSchwoaf");

   set<char, caselessless> s{};

   for (char c : kasten) s.insert(c);

   cout << s.size() << '\n';

   using out = std::ostream_iterator<char>;

   copy(s.begin(), s.end(), out(cout, "."));
}
```