Inhaltsverzeichnis

1. Die C++11 Erweiterungen

1.1. Ziele

☑ Die wichtigsten C++11 Erweiterungen anwenden können.

☑ Quellen

- ☐ http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/ (Danny Kalev: 1.7.2013)
- ☐ http://www.codeproject.com/Articles/570638/Ten-Cplusplus11-Features-Every-Cplusplus-Developer (Marius Bancila 1.7.2013)
- ☐ http://www.stroustrup.com/C++11FAQ.html (Stroustrup, 7.9.2013)
- ☐ http://en.wikipedia.org/wiki/C%2B%2B11
- ☐ http://en.cppreference.com/w/cpp
- ☐ http://dl.dropboxusercontent.com/u/13100941/C%2B%2B11.pdf (Alex Sinyakov, Software Engineer at AMC Bridge, 7.9.2013)

1.2. Zum Stand der Implementierung von C++11

C++ wurde 2003 überarbeitet und das Ergebnis wurde TR1 (Library Technical Library Report 1) genannt.
http://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1

C++ wurde auch 2005 überarbeitet und das Ergebnis wurde TR2 (Library Technical Libraray Report 2) genannt.

Die verschiedenen Kompilerhersteller orientierten sich an diesen beiden Reports.

Bei der C++11 Standardisierung wurden wesentliche Teile aus TR1 und TR2 übernommen.

Die derzeitigen Compiler und C++-Libraries unterstützen derzeit nur teilweise den C++11 Standard. Oft muss deshalb auf and. Bibliotheken (meist die Boost-Library) ausgewichen werden.

Siehe auch: http://www.stroustrup.com/C++11FAQ.html#when-libraries

Verwenden Sie zB:

g++ **-std=c++11** hallo.cpp -ohallo.exe

1.3. Type - Informationen

1.3.1. Standard-Datentypen

C++11 unterstützt nun:

☑ **int8_t**
uint8_t

☑ **int16_t**
uint16_t

☑ **int32_t**
uint32_t

☑ **int64_t**
uint64_t

String Literale C++11:

```
string test=R"(C:\A\B\C\D\file1.txt)";

cout << test << endl;
```

C:\A\B\C\D\file1.txt

```
string test;
test = R"(First Line.\nSecond line.\nThird Line.\n)";

cout << test << endl;
```

First Line.\nSecond line.\nThird Line.\n

```
string test =
R"(First Line.
Second line.
Third Line.)";
```

```
cout << test << endl;
```
First Line.
Second line.
Third Line.

1.3.2. Type traits – Typmerkmale

Typmerkmale zur Laufzeit abfragen.

C++11	Output

`//  type_traits.cpp`
`// g++ -std=c++1 type_traits.cpp -o  type_traits.exe`
 	
`#include <type_traits>`
`#include <iostream>`
`using namespace std;`
 	
`struct A { };`
`struct B { virtual void f(){} };`
`struct C : B {};`
 	
`int main(){`
`cout <<"int:"<<has_virtual_destructor<int>::value<<endl;`
`cout << "int:"<< is_polymorphic<int>::value << endl;`
 	
`cout << "A: " << is_polymorphic<A>::value << endl;`
`cout << "B: " << is_polymorphic<B>::value << endl;`
`cout << "C: " << is_polymorphic<C>::value << endl;`
 	
`typedef int mytype[][24][60];`
`cout << "(0 dim.): " << extent<mytype,0>::value << endl;`
`cout << "(1 dim.): " << extent<mytype,1>::value << endl;`
`cout << "(2 dim.): " << extent<mytype,2>::value << endl;`
`return 0;`
`}`
	int:0
	int:0
	
	A: 0
	B: 1
	C: 1
	
	(0st dim.): 0
	(1st dim.): 24
	(2st dim.): 60

1.3.3. Initialisierungen bei C++11

C++11 regelt die verschiedenen Initialisierungen durch die Verwendung von {}.

```
class Foo{
private:
     int a;
     int b;
public:
     Foo(int i, int j);
     ...
};


Foo aObj{0,0}; //C++11 only. Equivalent to: C c(0,0);
```

```
int* a = new int[3] { 1, 2, 0 }; //C++11 only
```

```
class Foo {
  int a[4];
public:
  Foo() : a{1,2,3,4} {} //C++11, member array initializer
};
```

Man braucht nun keine langen push_back() Aufrufe zur Initialisierung.

```
// C++11 container initializer
vector<string> vs={ "first", "second", "third"};
```

```
map<string,string> singers ={
     {"Lady Gaga", "+1 (212) 555-7890"},
     {"Hansi Hinterwald", "+1 (212) 555-0987"}
};
```

Es werden auch **In-Class Initialisierungen** von Membern erlaubt.

```
class Foo{
 int a=7;  //C++11 only
public:
 Foo();
};
```

Hier noch ein kleines Beispiel:

```
// init.cpp
// g++ -std=c++11 init.cpp -o init.exe

#include <iostream>
using namespace std;

int main(){
     size_t count{0};
     char c{};

     cout << "Zeicheneingabe bis EOT (^Z or ^D):"<< endl;
     while (std::cin >> c)
          ++count;

     std::cout << count << "\n";
     return 0;
}
```

1.3.4. std::string

Typkonvertierungen bei string Objekten.

☑ **vorzeichenbehaftetet**:

int **stoi**(const std::string& str, size_t *pos = 0, int base = 10);

long **stol**(const std::string& str, size_t *pos = 0, int base = 10);

long long **stoll**(const std::string& str, size_t *pos = 0, int base = 10);

☑ **vorzeichenlos**:

unsigned long **stoul**(const std::string& str, size_t *pos = 0, int base = 10);

unsigned long long **stoull**(const std::string& str, size_t *pos = 0, int base = 10);

☑ **Fließkommazahlen**:

float **stof**(const std::string& str, size_t *pos = 0);

double **stod**(const std::string& str, size_t *pos = 0);

long double **stold**(const std::string& str, size_t *pos = 0);

1.3.5. auto und decltype

In C++ müssen alle Variablen vor ihrer Verwendung deklariert werden, d.h. Es muss der Datentyp angeben werden.

C++11 erlaubt im Falle einer Initialisierung der Variablen den Datentyp nicht explizit angeben zu müssen. Man verwendet das Schlüsselwort auto. Der Datentyp wird von der Initialisierung her bestimmt.

```
auto i = 42;        // i is an int
auto l = 42LL;      // l is a long long
auto p = new foo(); // p is a foo*
```

Der eigentliche Vorteil dieser Methode ergibt sich bei der Verwendung von **Iteratoren** aus der STL.

```
std::map<std::string, std::vector<int>> map;

for(auto it = map.begin(); it != map.end(); ++it){
     ...
}
```

Hier noch ein weiteres Beispiel:

```
// alte Methode
void func(const vector<int> &vi){
```

```
     vector<int>::const_iterator ci=vi.begin();
}

unter Verwendung von auto kann man folg. Schreiben:

// neue Methode
void func(const vector<int> &vi){
     auto ci=vi.begin();
}
```

C++11 bietet auch die Möglichkeit

den **Typ eines Objektes** oder ganzen Ausdruckes **mit dem neuen Operator decltype zu erfragen**.

decltype übernimmt einen Ausdruck und liefert seinen Typ:

```
const vector<int> vi;

typedef decltype (vi.begin()) CONST_ITER;

CONST_ITER another_const_iterator;
```

Bei Templates ist dies etwas schwieriger:

Um den Return-Typ zu ermitteln verwendet man decltype:

```
template<class T, class U>
??? add(T x, U y)
//return type???
{
     return x+y;
}
```

C++11:

```
template<class T, class U>
auto add(T x, U y) -> decltype(x+y)
{
     return x+y;
}
```

1.3.6. Strongly typed enums

Enumerations können nun an einen bestimmten Scope/Gültigkeitsbereich mit der Angabe **enum class** gebunden werden.

```
enum class Options {None, One, All};
Options o = Options::All;
```

1.3.7. nullptr

Man sollte statt des bisherigen NULL bzw. 0 für den sog. NULL-Pointer nun das Schlüsselwort nullptr verwenden.

Es wird nun also bei einer irrtümlichen Verwendung des nullptr in Verbindung mit integer-Größen **ein Fehler generiert**.

```
void f(int);    //#1
void f(char*);  //#2


// bisher besteht die Frage: Welche Funktion wird aufgerufen?
f(0);

// ab: C++11
f(nullptr) // durch die Verwendung von nullptr ist klar definiert,
           // welche Funktion aufgerufen wird: →  #2
```

nullptr kann für alle Arten v. Pointern verwendet werden: (… inkl. function-pointer und pointers to members)

```
const char* pc= str.c_str(); //data pointers
if (pc != nullptr)
  cout<<pc<<endl;


void (*pmf)()=nullptr; //pointer to function

int (PERSON::*pmf)()=nullptr; //pointer to member function
```

nullptr kann zwar mit bool verwendet werden, nicht aber mit integer-Größen:

```
bool f = nullptr;
int i = nullptr;     // error: A native nullptr can only be
                     //converted to bool
```

siehe weiter unten:

nullptr kann auch mit shared_ptr verwendet werden.

1.4. Kontrollsteuerungen

1.4.1. Range for-loops, std::begin, std::end

Um über alle Elemente einer beliebigen Collection zu iterieren verwendet man nun das foreach-Paradigma. Man braucht sich also nicht um Iteratoren, Collection-Größe oder Indices kümmern.

Beispiel1:

```
int arr[] = {1,2,3,4,5};

for(int& e : arr){
  e = e*e;
}
```

Beispiel2:
Was gibt das Programm aus?

```
std::map<std::string, std::vector<int>> map;
std::vector<int> v;

v.push_back(1);
v.push_back(2);
v.push_back(3);

map["one"] = v;
...

for(const auto& kvp : map) {
  std::cout << kvp.first << std::endl;

  for(auto v& : kvp.second){
     std::cout << v << std::endl;
  }
}
```

```
vector<int> v;
sort( std::begin(v), std::end(v) );

int a[] = {1,2,3,4,5};
sort( std::begin(a), std::end(a) );
```

1.4.2. Lambda Expressions

Zur Einleitung siehe:

http://goparallel.sourceforge.net/exploring-new-lambda-features-c11/

Lamdas sind sogenannte anonyme Funktionen – ein Konzept aus den funktionalen Programmiersprachen. Lamdas können überall dort stehen, wo ein Funktionsaufruf stehen kann.

Hier die allg. Syntax:

```
[capture](parameters)->return-type {body}
```

Hier nun ein erstes Beispiel:

```
/* lambda.cpp
 * g++ -std=c++11 lambda.cpp -o  lambda.exe
 */

#include <iostream>
#include <vector>
#include <algorithm> // for_each, find_if

using namespace std;

int main(){
     std::vector<int> v;
     v.push_back(1);
     v.push_back(2);
     v.push_back(3);

     std::for_each(  std::begin(v),
                          std::end(v),
                          [](int n) -> void {cout << n << endl;});

     auto is_odd = [](int n)-> bool {return n%2==1;};

     auto pos = std::find_if(std::begin(v), std::end(v), is_odd);

     if(pos != std::end(v))
          std::cout << *pos << std::endl;

     return 0;
}
```

Frage: Was gibt das obige Programm aus?

1
2
3
1

Es stellt sich nun die Frage, wie kann ein Lambda auf den G**ültigkeitsbereich seiner Umgebung** zugreifen?

Hier kommt der [capture] Ausdruck zum Einsatz. Darin kann man den Zugriff regeln. Ein Beispiel:

```
/* lambda2.cpp
 * g++ -std=c++11 lambda2.cpp -o  lambda2.exe
 */

#include <iostream>
#include <algorithm> // for_each
#include <string>
using namespace std;

int main(){
     string s="Hello World!";
     int iUppercase = 0; //modified by the lambda

     std::for_each(  std::begin(s),
                     std::end(s),
                     [&iUppercase] (char c) -> void {
                          if (isupper(c)) iUppercase++;}
     );

     cout<< iUppercase<<" uppercase letters in: "<< s<<endl;
     return 0;
}
```

Das & bei [**&iUppercase**] gibt an, dass per Referenz auf die Variable zugegriffen wird. Ohne dem & würde per Value zugegriffen.

Frage: Was gibt das obige Programm aus?

2 uppercase letters in: Hello World!

1.5. Klassen-Informationen

1.5.1. Override and final

Override und final werden zur genaueren Spezifizierung der Vererbung verwendet.

☑ **Override**
gibt an, dass eine geerbte Methode überschrieben werden soll.

☑ **Final**
gibt an, dass eine Methode von den Unterklassen nicht überschrieben werden kann.

Hier ein Beispiel für folg. Klassenhierarchie:

B → D → F

```
/* override.cpp
 * g++ -std=c++11 override.cpp -o override.exe
 */
```

```
#include <iostream>
using namespace std;

class B {
public:
   virtual void f(int) {cout << "B::f" << endl;}
   virtual void g(int) {cout << "B::g" << endl;}
};



class D : public B{
public:
   virtual void f(int) override final {cout << "D::f" << endl;}
   virtual void g(int) override {cout << "D::g" << endl;}
};

class F : public D{
public:
   virtual void g(int) override {cout << "F::g" << endl;}
};


int main(){
     B* base;
     B b;
     D d;
     F f;

     cout << "B* base= &b:........." << endl;
     base= &b;
     base->f(1);
     base->g(1);

     cout << "B* base= &d:........." << endl;
     base= &d;
     base->f(1);
     base->g(1);

     cout << "B* base= &f:........." << endl;
     base= &f;
     base->f(1);
     base->g(1);

     cout << endl;
     return 0;
}
```

Frage: Was gibt das obige Programm aus?

```
B* base= &b:.........
B::f
B::g
```

```
B* base= &d:.........
D::f
D::g
B* base= &f:.........
D::f
F::g
```

Frage:

Warum wird hier der Compiler einen Fehler melden?

B → D

```
class B {
public:
   virtual void f(short) {std::cout << "B::f" << std::endl;}
};

class D : public B
{
public:
   virtual void f(int) override {std::cout << "D::f" << std::endl;}
};
```

Antwort:
Die zu überschreibende Funktion f in der Klasse D hat **keine entsprechende Funktion in ihrer Oberklasse** B. Beachte, dass beide Funktionen zwar den gleichen Namen f haben, aber sie sind wegen des unterschiedlichen Parameters (short bzw. int) im Sinne von c++ unterschiedliche Funktionen.

1.5.2. Delete und default Funktionen

```
class Foo {
     Foo()=default;
     virtual ~Foo()=default;
};
```

☑ **default** bedeutet,
dass für diese Funktionen der Compiler seine **interne Standard-Implementierung** der Funktion verwenden soll. Dies ist effizienter als die manuelle Implementierung.

☑ **Delete** bedeutet,
Wir wissen, dass in C++ automatisch der
interne Kopier-Konstruktor und der
interne Kopier-Zuweisungsoperator erstellt werden.

Dies kann mit der =delete Angabe **verhindert** werden.

Dadurch kann das Kopieren von Objekten auf einfache Weise verhindert werden.

```
int func()=delete;
```

„Deleted functions are useful for **preventing object copying**, among the rest. Recall that C++ automatically

declares a copy constructor and an assignment operator for classes. To disable copying, **declare these two special member functions =delete“**

```
class NoCopy {
     NoCopy & operator =( const NoCopy & ) = delete;
     NoCopy ( const NoCopy & ) = delete;
};

NoCopy a;
NoCopy b(a); //compilation error, copy ctor is deleted
```

1.5.3. Move Semantik – eine Optimierung der Copy Semantik

C++11 hat das Konzept der **Rvalue-Referenz** durch den Operator: **&&** eingeführt.

Der Grund dafür war, dass man eine Referenz auf einen Lvalue und eine Referenz auf einen Rvalue unterscheiden kann.

Warum dies wichtig sein kann, soll weiter unten besprochen werden. Hier wollen wir zunächst einige Grundlagen besprechen:

☑ **Rvalue und Lvalue**

- ☐ Ein **Lvalue** ist ein Objekt, das **einen Namen hat**.
- ☐ Ein **Rvalue** hat keinen Namen und ist ein **temporäres** Objekt.

☑ **Implizite Member-Funktionen** einer C++-Klassse sind:

- ☐ Default-Konstruktor (nur wenn kein anderer Konstruktor definiert wurde)
- ☐ Destruktor
- ☐ Kopier-Konstruktor
- ☐ Kopier-Zuweisungsoperator

☑ **Shallow-Copy:**
Der Kopier-Konstruktor und der Kopier-Zuweisungsoperator kopieren die Member **bitweise**. D.h. Wenn ein Pointer-Member verwendet wird, werden **nur die Pointer** kopiert, **nicht aber die Objekte selbst**. Mann nennt dies 'Shallow-Copy'.

☑ **Deep-Copy:**
In vielen Fällen ist aber 'Deep-Copy' gefordert. Dazu muss allerdings

1. der Kopier-**Zuweisungoperator** und
2. der **Kopierkonstruktor** und
3. der **Destruktor**
   explizit programmiert werden.

Frage: Warum ist eine Referenz auf einen Rvalue (temporäres Objekt) sinnvoll?

Betrachten Sie folgenden Code:

```
MyString s, s2="Hello, world!";
```

```
s= s2.clone();
```

wobei folg. gilt:

```
class MyString{
private:
     char* _s;
     size_t _size;
public:
     …
     MySgring MyString::clone(void) const;
     …
};
```

Frage: Was passiert in der Zeile?

```
s= s2.clone();
```

1. **Die Methode clone() liefert ein MyString-Objekt (=durch den Kopierkonstruktor-Aufruf), das ein Rvalue (also ein temporäres Objekt ist), denn es**

2. **wird anschliessend durch den Kopier-Zuweisungsoperator dem Objekt s zugewiesen.**

3. **Anschliessend wird das temporäre Objekt nicht mehr benötigt und der Destruktor für dieses temporäre Objekt aufgerufen.**

Es wird also 2 mal Speicher alloziiert und 2 mal die Daten kopiert. (vgl. Deep-Copy).

1. **Return in clone() ruft den Kopier-Konstruktor auf**
2. **Der Kopier-Zuweisungsoperator**

Merke:

Dies kann man sich **sparen**, wenn man die **move-Semantik** (vlg.: **&&**) verwendet.

Es zahlt sich also aus, einer **Klasse, die Member mit Zeigern hat**, folgendes hinzuzufügen:
☑ den **Move-Konstruktor** und den
☑ **Move-Zuweisungsoperator**

Diese beiden Funktionen erhalten ein **T&& Argument**. Also eine Referenz auf ein Rvalue-Objekt. Es wird nicht alloziiert und Speicher kopiert, sondern **nur die Zeiger werden kopiert**.

Das folgende Beispiel zeigt eine einfache Buffer-Implementierung, die einen Zeiger auf ein Array von Elementen vom Type T hat. Zusätzlich hat die Klasse einen Member zur Speicherung der Anzahl der Elemente und zur Speicherung eines Namens.

```
/*
 * buffer.cpp
 * g++ -std=c++11 buffer.cpp -o buffer.exe
```

```
 */


#include <cassert>

#include <iostream>
#include <memory>

using namespace std;

template <typename T>
class Buffer {
     string               _name;
     size_t               _size;
     std::unique_ptr<T[]> _buffer;

public:
     // default constructor
     Buffer():
          _size(16),
          _buffer(new T[16])
     {}

     // constructor
     Buffer(const string& name, size_t size):
          _name(name),
          _size(size),
          _buffer(new T[size])
     {}

     // copy constructor
     Buffer(const Buffer& other):
          _name(other._name),
          _size(other._size),
          _buffer(new T[other._size])
     {
          T* source = other._buffer.get();   // unique_ptr
          T* dest = _buffer.get();
          std::copy(source, source + other._size, dest);
     }

     // copy assignment operator
     Buffer& operator=(const Buffer& other){
          if(this != &other){
               _name = other._name;

               if(_size != other._size) {
                    _buffer = nullptr;
                    _size = other._size;
                    _buffer = _size > 0 ? new T[_size] : nullptr;
               }

               T* source = other._buffer.get();
               T* dest = _buffer.get();
               std::copy(source, source + other._size, dest);
```

```
            }

            return *this;
      }

      // move constructor
      Buffer(Buffer&& temp):
            _name(std::move(temp._name)),
            _size(temp._size),
            _buffer(std::move(temp._buffer))
      {
            temp._buffer = nullptr;
            temp._size = 0;
      }

      // move assignment operator
      Buffer& operator=(Buffer&& temp){
            assert(this != &temp); // assert if this is not a temporary

            _buffer = nullptr;

            _name = std::move(temp._name);
            _size = temp._size;
            _buffer = std::move(temp._buffer);


            temp._buffer = nullptr;
            temp._size = 0;

            return *this;
      }
};


template <typename T>
Buffer<T> getBuffer(const string& name) {
      Buffer<T> b(name, 128);

      return b;  //calls move constructor
}

int main(){
      Buffer<int> b1;
      Buffer<int> b2("buf2", 64);

      Buffer<int> b3 = b2;

      Buffer<int> b4 = getBuffer<int>("buf4");  // move cons

      b1 = getBuffer<int>("buf5");  // move cons and move assign

      return 0;
}
```

Frage:

b4 wird durch den Move-Konstruktor erzeugt.

b1 erhält seinen neuen Wert durch den Move-Zuweisungsoperator.

Was ist der Grund dafür?

Antwort:

Der Grund für beide Fälle ist, dass getBuffer eine **temporäre Kopie (einen sog. Rvalue)** liefert.

Frage:

Warum wurde im obigen Beispiel die std::move() Funktion verwendet, obwohl string und unique_ptr die Move-Semantik implementiert haben?

```
...
            _name(std::move(temp._name)),
…
            _buffer(std::move(temp._buffer))
...
```

Antwort:

Weil **temp ein Lvalue** ist, denn es hat einen Namen. In diesem Fall würde automatisch der Kopier-Konstruktor verwendet werden.

Die Funktion std::move() macht aus einem Lvalue ein Rvalue Objekt. Damit wird erreicht, dass der Move-Konstruktor aufgerufen wird.

1.6. Smart Pointers

Smart Pointer bieten eine Art von automatischer 'Garbage Collection' für alle dynamisch angelegten Objekte.

☑ **unique_ptr**
werden verwendet, wenn der Pointer **nicht** an andere Pointer **kopiert** werden soll. unique_ptr bieten keinen Kopierkonstruktor.
Man kann aber mit den sogenannten move-Methode die Verwendung des unique_ptr an einen anderen Pointer übertragen.

☑ **shared_ptr**
werden verwendet, wenn auch **andere** Pointer **auf das gleiche Objekt** verweisen müssen. Hier wird ein interner Referenz-Zähler verwendet, der mitzählt, wie viele Pointer auf das Objekt verweisen.

☑ auto_ptr
sollten nicht mehr verwendet werden.

Hier ein Beispiel für unique_ptr

```
/* unique_ptr.cpp
 * g++ -std=c++11 unique_ptr.cpp -o  unique_ptr.exe
 */

#include <iostream>
#include <memory>
using namespace std;


void foo(int* p){
     cout << *p << endl;
}

int main(){
     std::unique_ptr<int> p1(new int(42));

     std::unique_ptr<int> p2 = std::move(p1); // transfer ownership

     if(p1)
          foo(p1.get());

     (*p2)++;

     if(p2)
          foo(p2.get());

     // kein delete p1 notwendig

     return 0;
}
```

Die Methode get() liefert den eigentlichen Zeiger.

☑ **Frage**:
Was gibt das folgende Programm aus?
43

☑ **Frage**:
Warum wird foo(p1.get()) **nicht** aufgerufen?

Weil zuvor mit p2= std::move(p1) die Ownership des unique_ptr an den Pointer p2 übertragen wurde.
Dabei wird p1 auf null gesetzt.

Nun ein Beispiel für shared_ptr:

```
/* shared_ptr.cpp
 * g++ -std=c++11 shared_ptr.cpp -o  shared_ptr.exe
 */
```

```
#include <iostream>
#include <memory>

using namespace std;

void foo(int* p){
     cout << *p << std::endl;
}

void bar(std::shared_ptr<int> p){
     ++(*p);
}

int main(){
     std::shared_ptr<int> p1(new int(42));
     std::shared_ptr<int> p2 = p1;

     bar(p1);
     foo(p2.get());

     return 0;
}
```

☑ Frage: Was gibt das folgende Programm aus?
43

Weitere Informationen zu Smart-Pointer finden Sie in

CPP/07-smart-pointer/01-lernen/smart_ptr.odt

1.7. C++11 STL

Wir wollen in der Folge einige Beispiele kennen lernen:

1.7.1. std::regex

```
bool equals = regex_match("subject", regex("(sub)(.*)") );
```

Der g++ Compiler unterstützt derzeit REGEX nicht ausreichend, sodass auf die BOOST-Library ausgewichen werden sollte.

Siehe: CPP/10-boost/01-lernen

1.7.2. std::chrono

```
// chrono.cpp
// g++ -std=c++11 chrono.cpp -o chrono.exe

#include <chrono>
#include <iostream>

using namespace std;
using namespace std::chrono;

int main(){
     auto start = high_resolution_clock::now();

//   some_long_computations();

     auto end = high_resolution_clock::now();

     cout<<duration_cast<milliseconds>(end-start).count();

     return 0;
}
```

1.7.3. +STL-Algorithmen

std::all_of, std::none_of, std::any_of, std::find_if_not,

std::copy_if, std::copy_n, std::move, std::move_n, std::move_backward,

std::shuffle, std::random_shuffle,

std::is_partitioned, std::partition_copy, std::partition_point,

std::is_sorted, std::is_sorted_until, std::is_heap_until,

std::min_max, std::minmax_element,

std::is_permutation, std::iota

siehe: http://en.cppreference.com/w/cpp

1.8. Threading

Siehe
CPP/99-thread/01-lernen/thread.odt