C++ Einleitung

- Mitte der sechziger Jahr wurde bei AT&T an Mehrbenutzer-Betriebssytemen gearbeitet.
- Nach dem Rückzug von AT&T aus dem Großprojekt Multics entwickelten Thompsen und Ritchie Ende der sechziger Jahre eine erste in Assembler geschriebene Version, UNIX, auf einer PDP 7.
- Die Implementierung von UNIX in der von Ritchie entwickelten prozeduralen Programmiersprache C stellt einen Schlüsselbaustein unserer heutigen Computing Technologie dar.
- C++ wurde von Bjarne Stroustrup (Bell Laboratories) ab 1979 entwickelt, um objektorientiertes Programmieren in einer C-artigen Sprache zu erleichtern.
- C++ ist eine Erweiterung der Sprache C und abwärtskompatibel
- 1985 wurde aus „C with classes“ C++ mit dem release 2.0 im Jahr 1989
- GNU Compiler collection (gcc), implementiert die 6 ISO C++ standards der letzten Jahre (C++98, C++03, C++11, C++14, C++17, C++20)
- Der letzte offizielle Standard wurde 2020 festgelegt, ISO/IEC 14882:2020(E) see http://www.open-std.org/jtc1/sc22/wg21/

C++ Einleitung

- Computerprogramm
  Computer prozessieren Daten, führen Berechnungen durch, fällen Entscheidungen mit Hilfe eines Satzes von Anweisungen.
  Eine Aktion wird mit Hilfe von Schlüsselwörtern beschrieben, die durch eine Programmiersprache, z. B. C++, festgelegt sind.
  Die Schlüsselwörter werden im Klartext in einer Datei gespeichert und von einem Übersetzungsprogramm (Compiler) in Maschinensprache umgewandelt
- Wir benutzen als C++ Compiler die öffentlich verfügbare GNU Compiler Collection (gcc), die seit version 11 den letzten C++ Standard unterstützt.
- Im Rahmen dieses Kurses wird keine Entwicklungsumgebung verwendet. Wir arbeiten auf der shell in der Linux Umgebung des CIP pools.
- Arbeiten Sie in Gruppen, dadurch werden Fehler beim Implementieren der Programme reduziert.
  Es ist ok Programme oder Programmteile zu kopieren, aber
  - Verwenden Sie nur code, den Sie auch verstehen und getestet haben
  - Urheberrechte beachten (GNU General Public License, Version)

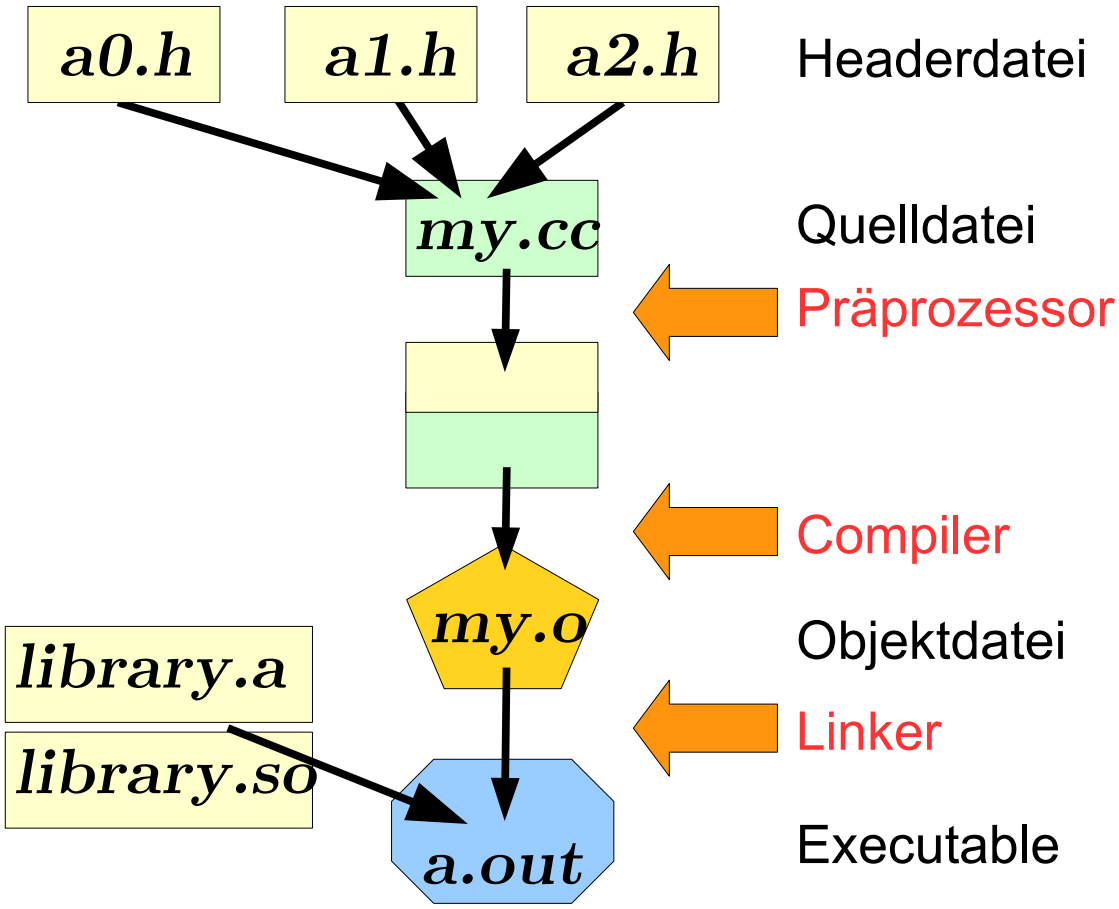
Ein Beispiel

myFirst.cc

```
// Add 2 Integer typed in by the user via keyboard

#include <iostream>
using namespace std;

int main()

{

  int a,b,sum;

  cout << "Enter integers to be added:" << endl;

  cin >>  a  >> b ;

  sum = a + b ;

  cout << "The sum is " << sum << endl ;

  return 0 ;

}
```

Ein ausführbares C++ Programm

Die Erzeugung eines ausführbaren C++ Programmes erfolgt in 3 Schritten:



Daten zur Struktur und Definitionen des Programms

Enthält Anweisungen zum Ablauf und Verhalten des Programms

Aus dem C++ Quellcode übersetzter Maschinencode

Alle vom Linker zusammen-gebundenen Objektdateien bilden das ausführbare Programm

Einleitung - GNU Compiler Collection

- Übernimmt generell folgende Aufgaben: preprocessing, compilation, assembly and linking
- Beinhaltet Compiler für die Programmiersprachen C, C++, Objective-C, Fortran, Ada, Go und Java
- Freie Software unter GNU General Public License
- Unterstützung vieler Hardware und Betriebssystem Plattformen (die am häufigsten portierte Compiler Suite, > 60 Plattformen)
- Standard Compiler der linuxartigen Betriebssysteme
- Unterstützt auch mitgelieferte Softwaresammlungen (libraries, libstdc++,..)
- Dokumentation https://gcc.gnu.org/onlinedocs/
- Verhalten wird durch Optionen in Form von Flags und File-Namen gesteuert
  - Optionen haben „multi letter names“, daher nicht wie in Unix Befehlen: -dv $\neq$ -d -v
  - File-Name Endungen bestimmen den aktivierten compiler

C++ Files: .C, .cc, .cpp, .cxx, .c++, .h, Extension: .hh, .hpp, .hxx, .h++

  - Details auf der shell mit dem Befehl `man gcc` oder ausührlicher `info gcc` (falls die Hilfen installiert sind, fehlen im CIP pool, https://www.gnu.org/software/gcc/)

GNU Compiler Collection gcc

- Sprachoptionen:
  - c c-header cpp-output
  - c++ c++-header c++-cpp-output → g++
  - objective-c objective-c-header objective-c-cpp-output
  - objective-c++ objective-c++-header objective-c++-cpp-output
  - assembler assembler-with-cpp
  - ada
  - f77 f77-cpp-input f95 f95-cpp-input
  - go
  - java

- Verwendung von g++ setzt Standard Library Pfade für den Linker

GNU Compiler Collection gcc

- Übersetzen eines C++ Programmes

```
g++ myFirst.cc        →    a.out
```

Ausführen des Programms in der shell:

```
./a.out
```

- Nützliche Optionen von g++
  - Programm Namen vergeben

```
g++ myFirst.cc -o MyProgramm        →    MyProgramm
```

  - Compiler Warnungen einschalten

```
g++ -Wall -Wextra myFirst.cc -o MyProgramm
```

  - Nur Compiler verwenden ohne Linker

```
g++ -c myFirst.cc -o mFirst.o       →  myFirst.o
```

  - Abschalten der nicht standard-conformen Compiler Optionen von g++

```
g++ -ansi myFirst.cc
```

  - Warnungen bei nicht standard-conformen Erweiterungen von g++

```
g++ -pedantic myFirst.cc
```

  - Automatische Optimierungen des Programms

```
g++ -Ox myFirst.cc     x [1,2,3]
```

C++ Wiederholung

- Linux
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/LinuxCommands.pdf
- Zugang zum CIP Pool
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/cipKursInformation.pdf
- Zeiger Konzept
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/Pointer.pdf
- Funktionen
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/Funktionen.pdf

Wir wollen zwei 3er-Vektoren über die Tastatur einlesen. Es soll der Normalenvektor der Ebene, die durch die beiden Vektoren aufgespannt wird, bestimmt und ausgegeben werden.

Arbeitsvorschlag:

normalenVektor.cc

- Schreiben Sie eine Funktion, die den Normalenvektor zurück gibt.
  Wie werden die beiden Vektoren übergeben und das Resultat zurückgegeben?

- Brauchen wir weitere Hilfsfunktionen?

- Welche Rolle spielen Funktionsheader?
  Lagern Sie die header in eine eigene Datei aus.
  Welche Änderungen sind notwendig, wenn die header in ein directory `include` oberhalb ihres directories mit dem Quellcode, e.g. `src`, ausgelagert werden sollen.

- Erzeugen Sie eine Bibliothek mit `ar` die dann statisch gelinkt werden soll.
  Welche commands auf der shell müssen wir ausführen?
  Die Bibliothek soll in einem directory mit dem Namen lib untergebracht werden.

solution.txt

Funktionszeiger

Zeiger können nicht nur auf Variablen gesetzt werden, sondern auch auf Funktionsspeicherbereiche. Dadurch läßt sich das Programm Verhalten zur Laufzeit dynamischer gestalten. Ein modernes Konzept wird mit Lambda Funktionen oder mit Funktionsobjekten (Funktoren) implementiert. Funktionszeiger werden meistens mit Type Definitionen `typedef` verwendet.

Beispiel:

```
#include <iostream>
int multiplication(int a, int b){ return a*b;}
int difference (int a, int b){return a-b;}
typedef int  (*pointerType)(int, int);
int main() {
    pointerType calcOperation = nullptr;
    calcOperation = &multiplication;
    std::cout << (*calcOperation)(40, 8) << std::endl;
    calcOperation = &difference;
    std::cout << (*calcOperation)(40, 8) << std::endl;
}
```

function pointer auf Funktion, die int zurück gibt und 2 int Argumente hat

Definition der Funktion

Neudefinition der Funktion

Funktionszeiger

Function Pointer erlauben an Funktionen mit numerischen Methoden, wie `integrate` oder `differentiate` , beliebige Funktionen zu übergeben.

Beispiel: Ausgabe Funktion

funcPointer.cc

```
#include <iostream>
#include <cmath>
typedef double (*FuncPtr)(double);
void printVal (FuncPtr, double);
int main() {
   FuncPtr f = nullptr;
   f = & sin;
   printVal(f,1.5);
   f = & exp;
   printVal(f,2.);
   return 0;
}
void printVal (double (*FuncPtr)(double), double x){
   cout << "x = " << x << " f=" << FuncPtr(x) << endl;
}
```

Print beliebiger Funktionswerte

Funktionszeiger

Function Pointer erlauben an Funktionen mit numerischen Methoden, wie `integrate` oder `differentiate` , beliebige Funktionen zu übergeben.

Beispiel: Ausgabe Funktion

funcPointer.cc

```
#include <iostream>
#include <cmath>
typedef double (*FuncPtr)(double);
void printVal (FuncPtr, double);
int main() {
    FuncPtr f;
    f = & sin;
```

Arbeitsvorschlag:

- Schreiben Sie eine Funktion integrate(….), die für eine beliebige gegebene Funktion eine numerische Integration durchführt. Die Funktion soll mit Hilfe von Funktionszeigern übergeben werden.

numIntegrate.cc

Type Inference - auto and decltype

- Type Inference in C++ - automatische Bestimmung des zu verwendenden Datentyps zur compile Zeit.
  Mit C++11 wurden keywords eingeführt, die eine automatische Festlegung des Datentyps durch den Compiler erlauben. Dadurch ist im Prinzip eine weniger strikte Spezifikation des Datentyps notwendig.
- `auto` erlaubt eine automatische Festlegung des Datentyps aus dem Programm Kontext. Bei der Einführung von Variablen legt die Anfangszuweisung den Datentyp fest. Bei Funktionen mit `auto` als return type wird der Datentyp zur Laufzeit ermittelt.

Beispiel:

Automatisches Festlegen der Datentypen

autoVar.cc

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main() {
 auto i=4 ;  auto f=3.37;  auto ptr = &f;
 cout << typeid(i).name()<<" "<<typeid(f).name()
      <<" "<<typeid(ptr).name()<<endl;
 return 0; }
```

Compiler Fehler mit
auto k;
k=10;

Datentyp Abfrage

Ausgabe → i d Pd

int double *double

Type Inference - auto and decltype

- `declype` extrahiert den Datentyp aus einem Ausdruck im Programm Kontext. Das keyword erscheint wie ein Operator, der den Datentyp einer Anweisung auswertet.

Beispiel:

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

char retChar () { return 'X';}
int retInt() { return 99;}

int main() {
 decltype(retChar()) myChar;
 decltype(retInt()) myInt;
 cout << typeid(myChar).name() << " " <<
                  typeid(myInt).name() << endl;
 return 0 ;
}
```

Automatisches Festlegen des Datentypen

Ausgabe → c i

char int

Lambda Funktionen in C++

Ein Lambda Ausdruck ist eine namenlose Funktion (Funktionsobjekt) für kurze, nicht wiederverwendbare Programmteile, die in den laufenden Quellcode geschrieben werden und an die Variablen in der „Umgebung“ binden.

- Syntax Lambda Funktionen (C++11)

```
[ ]          (    )          →          {…..}
```

Bindung | Parameter (optional) | Rückgabe (optional) | Funktionskörper

Details: https://en.cppreference.com/w/cpp/language/lambda

- Beispiele

```
…..
auto func=[](int x,int y){cout << x*y << endl; return x*y ;};
func(10,5);
```

Ausgabe → 50

Lambda Ausdrücke sind Funktionsobjekte, Rückgabewerte sind einfach mit `auto` zu verwenden.

```
…...
const int mySize = 5; double all;
for (int j = 0 ; j < mySize ; j++)
    [&all] (double x) { all += x; } ((double) j);
cout << all << endl ;
```

Ausgabe → 10.

Lambda Funktionen in C++

Ein Lambda Ausdruck ist eine namenlose Funktion (Funktionsobjekt) für kurze, nicht wiederverwendbare Programmteile, die in den laufenden Quellcode geschrieben werden und an die Variablen in der „Umgebung“ binden.

- Syntax Lambda Funktionen (C++11)

```
[ ]          (    )          →          {…..}
```

Bindung | Parameter (optional) | Rückgabe (optional) | Funktionskörper

Details: https://en.cppreference.com/w/cpp/language/lambda

- Beispiele

```
…..
auto func=[](int x,int y){cout << x*y << endl; return x*y ;};
func(10,5);
```

Ausgabe → 50

Lambda Ausdrücke sind Funktionsobjekte, Rückgabewerte sind einfach mit `auto` zu verwenden.

Lambda Ausdrücke können Variable als Referenz binden

```
…...
const int mySize = 5; double all;
for (int j = 0 ; j < mySize ; j++)
    [&all] (double x) { all += x; } ((double) j);
cout << all << endl ;
```

Ausgabe → 10

Lambda Funktionen in C++

- Optionen für die Bindung [ ] (capture)

```
[ ]       //no variables defined. Attempting to use any external variables in the lambda
            expression is an error.
[x, &y]   //x is captured by value, y is captured by reference
[&]       //any external variable is implicitly captured by reference if used
[=]       //any external variable is implicitly captured by value if used
[&, x]    //x is explicitly captured by value. Other variables will be captured by reference
[=, &z]   //z is explicitly captured by reference. Other variables will be captured by value
```

https://en.wikipedia.org/wiki/Anonymous_function#C.2B.2B_.28since_C.2B.2B11.29

- Beispiel Nutzung mit Standard Template Library Funktionen

```
#include <vector>
#include <algorithm> // for_each
…..
vector<int> myList{ 1, 3, 5, 7, 9 };
int sum = 0;
for_each(begin(myList),end(myList),[&sum](int x) {sum += x;} );
cout << sum << endl;
…..
```

function pointer
function object

Ausgabe → 25

lambda.cc

C++ Wiederholung

- Input/Output
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/FileIO.pdf

- Strukturen
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/Strukturen.pdf

- Klassenkonzept
  https://www.physi.uni-heidelberg.de/~marks/c++_einfuehrung/Folien/Klassen.pdf

Functor / Function Object

Ein Funktionsobjekt (functor) ist eine Klasse/Struktur mit einem überladenen Operator (). Es können Objekte instanziert werden, die sich wie Funktionsaufrufe verhalten.

Beispiel:

```
#include <iostream>
struct squareVal{
      double operator()(double a) { return a*a;}
};
using namespace std;
int main( ) {
  double x = -7.;
  squareVal myObject;
  double square_x = myObject(x);
  cout << "x=" << x << " square_x=" << square_x << endl;
  return 0;}
```

`myObject` ist ein Objekt und keine Funktion, trotzdem können wir das Objekt durch das Überladen des () Operators wie eine Funktion aufrufen. Der Funktionsaufruf Operator ist als member Funktion deklariert.

Functor / Function Object

Das Funktionsobjekt hat alle Eigenschaften eines Objektes, es besitzt also einen Zustand und speichert Daten.

Beispiel: mit einer Klasse

```
.........
class MyAddFunctor {
      public:
      MyAddFunctor(int inp) { x = inp; }
      int operator()(int y){ return x+=y; }
      private:
      int x;
};
.........
MyAddFunctor func(5);
int v = func(10);
int u = func(25);

MyAddFunctor(5)(10) ;
```

Definition des Konstruktors

Der Operator () wird mit der gewünschten Funktionalität überladen.

Klasse wird instanziert

v = 15

u = 40

= 15

In der Standard Template Library werden häufig Funktoren benutzt.

functorTest.cc

Functor / Function Object

Das Funktionsobjekt hat alle Eigenschaften eines Objektes, es besitzt also einen Zustand und speichert Daten.

Beispiel: mit einer Klasse

.....

```
class MyAddFunctor {
      public:
      MyAddFunctor(int inp) { x = inp; }
      int operator()(int y){ return x+=y; }
      private:
      int x;
```

Definition des Konstruktors

Der Operator () wird mit der gewünschten Funktionalität

Arbeitsvorschlag:

- Schreiben Sie eine Klasse, die eine Gerade über slope und Achsenabschnitt implementiert. Überladen Sie nun den () Operator so, dass er für jeden als Argument übergebenen Wert den Funktionswert der Geradeninstanz zurückgibt.

functorSlope.cc

Standard Template Library (STL)

Die STL geht auf Entwicklung einer Standard Bibliothek für generische Algorithmen bei HP zurück. Sie wurde als Teil des C++ ISO Standards in die C++ Standard Bibliothek aufgenommen und dort weiterentwickelt. Sie enthält Datenstrukturen (Klassentemplates) in Form von Containern und Iteratoren für den Zugriff und Algorithmen, die auf den Datenstrukturen operieren.

- Container http://www.cplusplus.com/reference/stl/
  Ist eine Struktur die beliebige Daten aufnimmt und deren Verwaltung organisiert. Der Zugriff wird über member Funktionen und Iteratoren geregelt. Sie sind als generische Klassentemplates implementiert
  Container Typen:
  - Sequentielle Containerklassen
    array, vector, list, forward_list, deque
  - Geordnete und ungeordnete assoziative Containerklassen
    unordered_map, unordered_multimap, unordered_multiset, unordered_set, map, multimap, multiset, set
  - Containeradapterklassen
    priority_queue, queue, stack

Standard Template Library (STL)

- Iteratoren

Iteratoren bilden das Interface für den Datenzugriff auf Container. Sie weisen zwei grundsätzliche Funktionen auf:

- Zeige auf ein bestimmtes Element in einer Menge von Objekten
- Zeige durch selbst-Modifizierung auf das nächste Element in der Menge.

Der Zugriff auf Container Elemente kann ohne Kenntnis der Objektmenge erfolgen
Es gibt folgende Klassen von Iteratoren, die sich in ihren Zugriffseigenschaften unterscheiden:

- Eingabe- und Ausgabe-Iteratoren
  Durchlauf nur in eine Richtung, einmalige Dereferenzierung, Lesen oder Schreiben
- Forward-Iteratoren
  Durchlauf nur in eine Richtung, beliebig häufige Dereferenzierung
- Bidirektionale Iteratoren
  Durchlauf in Vorwärts- und Rückwärts-Richtung möglich
- Random-Access-Iteratoren
  Wahlfreier Zugriff auf die Elemente (nicht nur Vorgänger und Nachfolger), d.h. es kann vom Start-Iterator eine Integer > 1 addiert werden.

```
.........
list< int > A = { 5, 88, 1, 111, 12 };
for(auto i = begin(A); i != end(A); ++i) {
  cout << *i << ", " ; }
```

Instanziere Container Klasse

Iterator erlaubt Zugriff auf die Elemente (durch Dereferenzieren)

Standard Template Library (STL)

- Iteratoren - Zu beachtende Eigenschaften
  - Ausgabeiteratoren unterstützen keinen Vergleich, Test auf das Ende der Sequenz über Iteratoren nicht möglich!
  - Iterator Schleife

```
list< int > A = { 5, 88, 1, 111, 12 };
for(auto i = begin(A); i != end(A); ++i) { ….};
for(auto const& i: A){ ….};
```

Vergleich auf Ende

`auto` ermittelt den Datentyp als `int`

`const &` ist eine Referenz auf `const int`

→ schneller Zugriff auch wenn `int` durch komplexe Struktur ersetzt wird

  - Bei Eingabe- und Ausgabeiteratoren sollten Elementzugriff und Inkrementieren immer abwechselnd erfolgen

```
*ziel = 1; ++ziel; *ziel = 2; ++ziel;
```
erlaubt

```
*ziel = 1; ++ziel; ++ziel; ziel = 2;
```
verboten

`*iter++ = a` (Ausgabeiterator) und `a = *iter++` (Eingabeiterator) erlaubt

  - Bei bidirectionalen Iteratoren

```
iter++;
```
setzt Iterator `iter` eine Position weiter

```
iter--;
```
setzt Iterator `iter` auf vorherige Position zurück

  - Bei random access Iteratoren

```
iter += 3;
```
bedeutet `++iter; ++iter; ++iter;`

```
*(iter - 1) = 8;
```
bedeutet `tmp=iter; --tmp; *tmp = 8;`

Standard Template Library (STL)

- Iteratoren - Zu beachtende Eigenschaften
  - Iterator Adapter: die C++ standard library stellt einige spezialisierte Iteratoren zur Verfügung. Sie werden als Iterator Adapter bezeichnet, z.B. insert iterators, die Elemente einfügen ohne andere zu löschen,

```
back_inserter (container); front_inserter (container)
inserter (container , position)
```

stream iterators, die in einen input oder output strom schreiben

```
istream_iterator<string>(cin)
```

und reverse iterators, die die Iterator Richtung umkehren. Alle Container können reverse iterators über die Methoden `rbegin()` and `rend()` realisieren

```
vector< int > A = { 5, 88, 1, 111, 12 };
```

gibt `vector A` in umgekehrter aus

```
copy (A.rbegin(), A.rend(),ostream_iterator<int> (cout," "));
```

STLiterator.cc

Standard Template Library (STL)

- Algorithmen http://www.cplusplus.com/reference/algorithm/

sind Funktionstemplates mit Iteratoren und Operationen als Argumente.
Die Operationen können mit Lambda Funktionen oder Funktionsobjekten realisiert werden. Die Daten werden mit Iteratoren, die auf Container zeigen, zur Verfügung gestellt.
In der STL sind ca 80 Algorithmen definiert, häufig benutzte sind

`for_each`: wendet eine Operation auf alle Elemente eines Containers an
`transform`: transformiert Daten eines Containers mit einer Funktion
`copy`: kopiert Daten eines Containers in einen anderen
`sort`: sortiert innerhalb eines Containers
`find`: sucht nach einem bestimmten Element in einem Container
`search`: sucht nach einer Elementreihe in einem Container

Algorithmen können insbesondere feststellen, ob sie das Ende einer Sequenz erreicht haben.

```
// functor for printing
struct myPrint {
void operator() (double x) {cout << ' ' << x;}
void operator() (int i)  {cout << ' ' << i;}  };
.....
myPrint pr;
for_each (aVec.begin(), aVec.end(), pr);
```

STLalgorithm.cc