Objektorientiertes Programmieren im Wissenschaftlichen Rechnen

Olaf Ippisch
email: olaf.ippisch@iwr.uni-heidelberg.de

9. August 2013

Inhaltsverzeichnis

1 Einführung

1.1 Ziel der Vorlesung

Voraussetzungen

- Fortgeschrittene Beherrschung einer Programmiersprache
- Mindestens prozedurale Programmierung in C/C++
- Bereitschaft zu praktischer Programmierung

Ziele

- Verbesserung der Programmierkenntnisse
- Vorstellung von modernen Programmiermodellen
- Starker Bezug zu Themen mit Relevanz für das Wissenschaftliche Rechnen

Inhalt

- Eine kurze Wiederholung der Grundlagen objektorientierter Programmierung in C++ (Klassen, Vererbung, Methoden und Operatoren)
- konstante Objekte
- Fehlerbehandlung (Exceptions)
- Dynamischer Polymorphismus (Virtuelle Vererbung)
- Statischer Polymorphismus (Templates)
- Die C++ Standard-Template-Library (STL Container, Iteratoren und Algorithmen)
- Traits, Policies
- Design Pattern
- Template Metaprogramming
- C++-Threads

Während der ganzen Vorlesungen wird, soweit sinnvoll auf die Neuerungen durch den C++11-Standard eingegangen.

1.2 Vorteile objektorientierter Programmierung

Wie sollte ein gutes Programm sein?

- Korrekt/fehlerfrei
- Effizient
- Leicht zu benutzen
- Verständlich
- Erweiterbar
- Portierbar

Entwicklung der letzten Jahre

- Computer wurden schneller und billiger
- Der Umfang von Programmen stieg von mehreren hundert auf hunderttausende Zeilen
- Damit stieg auch die Komplexität von Programmen
- Programme werden heute in größeren Gruppen entwickelt, nicht von einzelnen Programmierern
- Paralleles Rechnen wird immer wichtiger, da heute nahezu alle verkauften Rechner über mehrere Prozessorkerne verfügen

Komplexität von Programmen

Zeit	Prozessor	Takt [MHz]	Cores	RAM [MB]	Platte [MB]	Linux Kernel [MB]
1982	Z80	6	1	0.064	0.8	0.006 (CPM)
1988	80286	10	1	1	20	0.020 (DOS)
1992	80486	25	1	20	160	0.140 (0.95)
1995	PII	100	1	128	2'000	2.4 (1.3.0)
1999	PII	400	1	512	10'000	13.2 (2.3.0)
2001	PIII	850	1	512	32'000	23.2 (2.4.0)
2007	Core2 Duo	2660	2	1'024	320'000	302 (2.6.20)
2010	Core i7-980X AMD 6174	3333 (3600) 2200	6 12	4'096	2'000'000	437 (2.6.33.2)
2013	Core i7-3970X AMD 6386 SE	3500 (4000) 2800 (3500)	6 16	8'192	4'000'000	482 (3.8.7)

Beispiel DUNE

- Framework für die Lösung Partieller Differentialgleichungen

- entwickelt von Arbeitsgruppen an den Universitäten Freiburg, Heidelberg, Münster, der Freien Universität Berlin und der RWTH Aachen
- 10 Core Developer, viele weitere Entwickler
- zur Zeit (16.04.2013) 365'768 Zeilen Programmcode (plus 74'025 Zeilen Kommentare)
- Anwender an vielen anderen Universitäten
- verwendet intensiv moderne C++-Konstrukte die in dieser Vorlesung vorgestellt werden

Programmierparadigmen

- Funktionale Programmierung (z.B. Haskel, Scheme)
  - Programm besteht nur aus Funktionen
  - Es gibt keine Schleifen, Wiederholungen werden durch Rekursion realisiert
- Imperative Programmierung
  - Programm besteht aus einer Abfolge von Anweisungen
  - Variablen können Zwischenwerte speichern
  - Es gibt spezielle Anweisungen die die Reihenfolge der Abarbeitung ändern, z.B. für Wiederholungen

Imperative Programmiermodelle

- Prozedurale Programmierung (z.B. C, Fortran, Pascal, Cobol, Algol)
  - Computerprogramm wird in kleine Teile (Prozeduren oder Funktionen) unterteilt
  - Diese können lokal nur temporäre Daten speichern, die beim Beenden der Prozedur gelöscht werden
  - Persistente Daten werden über Argumente und Rückgabewerte ausgetauscht oder als globale Variablen gespeichert
- Modulare Programmierung (z.B. Modula-2, Ada)
  - Funktionen und Daten werden zu Modulen zusammengefasst, die für die Erledigung bestimmter Aufgaben zuständig sind
  - Diese können weitgehend unabhängig voneinander programmiert und getestet werden

Lösungsansatz der objektorientierten Programmierung

In Analogie zum Maschinenbau:

- Zerlegung des Programms in eigenständige Komponenten
- Bestimmung der notwendigen Funktionalität die diese Komponente bereitstellen muss
- Alle dafür notwendigen Daten werden innerhalb der Komponente verwaltet

- Verschiedene Komponenten werden über Schnittstellen verbunden
- Verwendung der gleichen Schnittstelle für spezialisierte Komponenten die die gleiche Arbeit erledigen

Beispiel: Computer

Vorteile

- Die Komponenten können unabhängig voneinander entwickelt werden
- Wenn bessere Versionen einer Komponente verfügbar werden kann diese ohne größere Änderungen am Rest des Systems verwendet werden
- Es ist einfach mehrere Realisierungen der gleichen Komponente zu verwenden

Wie hilft C++ dabei?

C++ stellt einige Mechanismen zur Verfügung die diese Art ein Programm zu strukturieren unterstützen:

Klassen definieren Komponenten. Sie sind wie eine Beschreibung was eine Komponente tut und welche Eigenschaften sie hat (wie z.B. die Funktionen die eine bestimmte Grafikkartensorte zur Verfügung stellt)

Objekte sind Realisierungen der Klasse (wie eine Grafikkarte mit einer bestimmten Seriennummer)

Kapselung verhindert Seiteneffekte durch Verstecken der Daten vor anderen Programmteilen

Vererbung erleichtert eine einheitliche und gemeinsame Implementierung von spezialisierten Komponenten

Abstrakte Basisklassen definieren einheitliche Schnittstellen

Virtuelle Funktionen erlauben es zwischen verschiedenen Spezialisierungen einer Komponente zur Laufzeit auszuwählen

Templates erhöhen die Effizienz, wenn die Wahl der Spezialisierung bereits bei der Übersetzung bekannt ist

2 Klassen

Beispiel

```
#include<vector>

class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    double &Elem(int i, int j);
    void Print();
    int Rows();
    int Cols();

  private:
    std::vector<std::vector<double> > a_;
    int numRows_;
    int numCols_;
};
```

Klassendeklaration

```
class MatrixClass
{
// a list of the methods and attributes
};
```

Die Klassendeklaration definiert die Schnittstelle und die essentiellen Eigenschaften der Komponente

Eine Klasse hat *Attribute* (Variablen zur Speicherung von Daten) und *Methoden* (die Funktionen die eine Klasse zur Verfügung stellt). Die Definition von Attributen und die Deklaration von Methoden erfolgt zwischen geschweiften Klammern. Nach der schließenden Klammer muss ein Strichpunkt stehen.

Klassendeklarationen werden üblicherweise in einer Datei mit der Endung '.hh' oder '.h' gespeichert, sogenannten *Include*- oder Headerdateien.

Kapselung

1. One must provide the intended user with all the information needed to use the module correctly, and with nothing more.
2. One must provide the implementor with all the information needed to complete the module, and with nothing more.

David L. Parnas (1972)

. . . but much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies.

Brooks (1975)

```
class MatrixClass
{
  public:
    // a list of public methods
  private:
    // a list of private methods and attributes
};
```

Nach `public`: folgt die Beschreibung der Schnittstelle, d.h. der Methoden der Klasse die von außen aufgerufen werden können.

Nach `private`: steht die Definition von Attributen und von Methoden, die nur Objekten der gleichen Klasse zur Verfügung stehen. Dabei handelt es sich um die Daten und einige implementierungsspezifische Methoden die von der Komponente zur Bereitstellung der Funktionalität benötigt werden. Es sollte **nicht** möglich sein auf die in einer Komponente gespeicherten Daten von außen zuzugreifen um spätere Änderungen der Implementierung zu erleichtern.

```
struct MatrixClass
{
    // a list of public methods
  private:
    // a list of private methods and attributes
};
```

- Ist kein Schlüsselwort angegeben sind alle Methoden und Daten einer mit `class` definierten Klasse `private`. Wird eine Klasse als `struct` definiert z.B. `struct MatrixClass` dann sind alle Methoden per default `public`. Abgesehen davon sind `class` und `struct` identisch.

Definition von Attributen

```
class MatrixClass
{
  private:
    std::vector<std::vector<double> > a_;
    int numRows_;
    int numCols_;
    // further private methods and attributes
};
```

Die Definition eines Attributes in C++ besteht wie jede Definition einer Variable in C++ aus der Angabe von Typ und Variablennamen. Die Zeile wird mit einem Strichpunkt beendet. Mögliche Typen sind z.B.

- `float` und `double` für Fließkommazahlen mit einfacher und doppelter Genauigkeit
- `int` und `long` für ganze Zahlen
- `bool` für logische Zustände
- `std::string` für Zeichenketten

C++11: New Datatypes

- Die Länge und damit der Wertebereich von `short`, `int` and `long` (und ihrer vorzeichenlosen Varianten) ist in C und C++ nicht gut definiert. Es wird lediglich garantiert, dass

  ```
  sizeof(char)=1 <= sizeof(short) <= sizeof(int) <= sizeof(long)
  ```

- C++-11 führt neue Datentypen mit garantierten Längen und Wertebereichen ein:

  ```
  int8_t   [-128:127]         uint8_t   [0:255]
  int16_t  [-32768:32767]     uint16_t  [0:65535]
  int32_t  [-2^31:2^31-1]     uint32_t  [0:2^32-1]
  int64_t  [-2^63:2^63-1]     uint64_t  [0:2^64-1]
  ```

- Zusätzlich gibt es noch Varianten die mit `int_fast` oder `uint_fast` anfangen (z.B. `int_fast8_t`). Diese liefern den schnellsten Datentyp auf der jeweiligen Architektur, der mindestens die entsprechende Länge hat. Datentypen die mit `int_least` oder `uint_least` beginnen liefern die kürzesten Typen die den entsprechenden Wertebereich haben.
- `intptr_t` und `uintptr_t` liefern Datentypen die die richtige Länge haben um einen Pointer zu speichern.

C++11: Übersetzen

- Beim Übersetzen von Programmen mit C++11-Konstrukten muss beim `g++` ab Version 4.7 der Parameter `-std=c++11` angegeben werden, ebenso bei aktuellen Versionen von `clang` (ab 3.2). Bei `g++` bis Version 4.6 heißt der Parameter `-std=c++0x`.
- Informationen zur Unterstützung von C++11 durch verschiedene Versionen des `g++` finden sich auf `http://gcc.gnu.org/projects/cxx0x.html`

Methodendeklaration

```
class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    double &Elem(int i, int j);
};
```

Eine Methodendeklaration besteht immer aus vier Teilen:

- dem Type des Rückgabewertes
- dem Namen der Funktion
- einer Liste von Argumenten (mindestens dem Argumenttyp) getrennt durch Kommata und eingeschlossen in runde Klammern
- einem Strichpunkt

Wenn eine Methode keinen Wert zurück gibt, ist der Typ des Rückgabewertes `void`. Wenn eine Methode keine Argumente hat, bleiben die Klammern einfach leer.

Methodendefinition

```
class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    inline double &Elem(int i, int j)
    {
        return(a_[i][j]);
    }
};
```

Die Methodendefinition (d.h. die Angabe des eigentlichen Programmtextes) kann direkt in der Klasse erfolgen (sogenannte inline Funktionen). Der Compiler kann bei inline Funktionen den Funktionsaufruf weglassen und den Code direkt einsetzen. Mit dem Schlüsselwort `inline` vor dem Funktionsnamen kann man ihn explizit anweisen das zu tun.

```
void MatrixClass::Init(int numRows, int numCols)
{
    a_.resize(numRows);
    for (int i=0;i<a_.size();++i)
        a_[i].resize(numCols);
    numRows_=numRows;
    numCols_=numCols;
}
```

Wenn Methoden außerhalb der Klassendefinition definiert werden (dies erfolgt oft in einer Datei mit der Endung `.cpp`, `.cc` oder `.cxx`), muss vor dem Namen der Methode der Name der Klasse gefolgt von zwei Doppelpunkten stehen.

Überladen von Methoden

```
class MatrixClass
{
  public:
    void Init(int numRows, int numCols);
    void Init(int numRows, int numCols, double value);
    double &Elem(int i, int j);
};
```

Zwei Methoden (oder Funktionen) können in C++ den gleichen Namen haben, wenn sich ihre Argumente in Zahl oder Typ unterscheiden. Dies bezeichnet man als Überladen von Funktionen (overloading). Ein unterschiedlicher Typ des Rückgabewerts ist nicht ausreichend.

Konstruktoren

```
class MatrixClass
{
  public:
    MatrixClass();
    MatrixClass(int numRows, int numCols);
    MatrixClass(int numRows, int numCols, double value);
};
```

- Jede Klasse hat Methoden ohne Rückgabewert mit dem gleichen Namen wie die Klasse selbst: einen oder mehrere Konstruktoren und den Destruktor.
- Konstruktoren werden ausgeführt, wenn ein Objekt einer Klasse definiert wird bevor irgendeine andere Methode aufgerufen wird oder die Attribute verwendet werden können. Sie dienen zur Initialisierung.
- Es kann mehr als einen Konstruktor geben. Dabei gelten die gleichen Regeln wie bei überladenen Methoden.
- Gibt es keinen Konstruktor der `public` ist können keine Objekte der Klasse angelegt werden.

```
class MatrixClass
{
  public:
    MatrixClass()
    {
        // some code to execute at initialization
    };
};

MatrixClass::MatrixClass(int numRows, int numCols) :
    a_(numRows,std::vector<double> (numCols)),
    numRows_(numRows),
    numCols_(numCols)
{
    // some other code to execute at initialization
}
```

- Wie eine normale Methode können Konstruktoren innerhalb oder außerhalb der Klassendefinition definiert werden.
- Konstruktoren können auch dazu verwendet werden Attribute mit Werten zu initialisieren. Die Initialisierungsliste besteht aus dem Variablennamen gefolgt von dem Wert der zur Initialisierung verwendet werden soll (Konstante oder Variable) in Klammern getrennt durch Kommata. Sie steht getrennt durch einen Doppelpunkt nach der geschlossenen Klammer der Argumentliste.

C++11: In-Class Initialisierung, Delegierende Konstruktoren

```
class MatrixClass
{
  private:
    std::vector<std::vector<double> > a_;
    int numRows_ = 0;
    int numCols_ = 0;
  public:
    MatrixClass();
    MatrixClass(int numRows, int numCols, double value);
    MatrixClass(int numRows, int numCols) : MatrixClass(numRows,numCols,0.0)
    {}
};
```

- In C++11 können auch nicht-statische Mitglieder von Klassen gleich bei ihrer Definition initialisiert werden. Wird in einem Konstruktor eine Initialisierungsliste angegeben so hat diese Vorrang.
- Konstruktoren dürfen andere Konstruktoren aufrufen. Die Unterscheidung geht wie bei überladenen Funktionen nach den Typen in der Argumentliste.

Destruktor

```
class MatrixClass
{
  public:
    ~MatrixClass();
};
```

- Es gibt nur einen Destruktor pro Klasse. Er wird aufgerufen, wenn ein Objekt der Klasse gelöscht wird.
- Der Destruktor hat keine Argumente (die Klammern sind also immer leer).
- Das Schreiben eines eigenen Destruktors ist z.B. nötig, wenn die Klasse Speicher dynamisch alloziert.
- Der Destruktor sollte `public` sein.

Default Methoden

Für jede Klasse `class T` erzeugt der Compiler automatisch fünf Methoden, wenn diese nicht vom anderweitig definiert werden:

- Konstruktor ohne Argumente: `T();` (ruft rekursiv die Konstruktoren der Attribute auf). Der Default Konstruktor wird nur generiert, wenn keinerlei andere Konstruktoren definiert werden.
- Copy Konstruktor: `T(const T&);` (memberwise copy)
- Destruktor: `\~T();` (ruft rekursiv die Destruktoren der Attribute auf)
- Zuweisungsoperator: `T &operator= (const T&);` (memberwise copy)
- Adressoperator: `int operator& ();` (liefert Speicheradresse des Objekts zurück)

Copy Konstruktor und Zuweisungsoperator

```
class MatrixClass
{
  public:
    // Zuweisungsoperator
    MatrixClass &operator=(const MatrixClass &A);
    // copy konstruktor
    MatrixClass(const MatrixClass &A);
    MatrixClass(int i, int j, double value);
};

int main()
{
    MatrixClass A(4,5,0.0);
    MatrixClass B = A; // copy konstruktor
    A = B; // Zuweisungsoperator
}
```

- Der Copy Konstruktor wird aufgerufen, wenn ein neues Objekt als Kopie eines bestehenden Objektes angelegt wird. Das passiert oft auch implizit (z.B. beim Anlegen temporärer Objekte).
- Der Zuweisungsoperator wird aufgerufen, wenn einem bestehenden Objekt ein neuer Wert zugewiesen wird.

C++11: Management von Default-Methoden

```
class MatrixClass
{
  public:
    // Zuweisung und Kopieren verbieten
    MatrixClass &operator=(const MatrixClass &A) = delete;
    MatrixClass(const MatrixClass &A) = delete;
    // automatische Konvertierung von short verhindern
    MatrixClass(int i, int j, double value);
    MatrixClass(short i, short j, double value) = delete;
    virtual ~MatrixClass() = default;
};
```

- Manchmal möchte man verhindern, dass bestimmte Default-Methoden verfügbar sind, z.B. damit keine Objekte einer Klasse angelegt werden können, wenn man nur statische Attribute und Methoden verwendet.
- Bisher musste man dazu die Default-Methoden anlegen und `private` machen.
- Bei C++11 geht das mit dem keyword `delete`.
- Bei Klassen mit virtuellen Funktionen ist es ratsam einen virtuellen Destruktor anzulegen, auch wenn die aktuelle Klasse keinen Destruktor braucht. Das geht jetzt einfacher und klarer mit dem keyword `default`.

2.1 Operatoren

Überladen von Operatoren

- In C++ ist es möglich Operatoren wie + oder − für eigene Klassen neu zu definieren.
- Operatoren werden wie gewöhnliche Funktionen definiert. Der Funktionsname ist `operator` gefolgt vom Symbol des Operators z.B. `operator+`
- Wie für eine gewöhnliche Methode müssen auch für einen Operator der Typ des Rückgabewertes und die Argumentliste angegeben werden:

  ```
  MatrixClass operator+(MatrixClass &A);
  ```

- Operatoren können sowohl als Methoden eines Objektes als auch als gewöhnliche (non-member) Funktionen definiert werden.
- Die Anzahl der Argumente hängt vom Operator ab.

Unäre Operatoren

```
class MatrixClass
{
  public:
    MatrixClass operator-();
};

MatrixClass operator+(MatrixClass &A);
```

- Unäre Operatoren sind: `++ -- + - ! ~ & *`
- Ein unärer Operator kann entweder als Klassenfunktion ohne Argument oder as non-member Funktion mit einem Argument definiert werden.
- Der Programmierer muss sich für eine dieser zwei Möglichkeiten entscheiden, da es dem Compiler nicht möglich ist die beiden Varianten im Programmtext zu unterscheiden, z.B. `MatrixClass &operator++(MatrixClass A)` und `MatrixClass &MatrixClass::operator++()` würden beide aufgerufen über `++a`.

Binäre Operatoren

```
class MatrixClass
{
  public:
    MatrixClass operator+(MatrixClass &A);
};

MatrixClass operator+(MatrixClass &A, MatrixClass &B);
```

- Ein binärer Operator kann entweder als Klassenfunktion mit einem Argument oder als non-member Funktion mit zwei Argumenten definiert werden.
- Mögliche Operatoren sind: `* / % + - & ^ | < > <= >= == != && || >> <<`

- Operatoren die ein Element ändern wie `+= -= /= *= %= &= ^= |=` können nur als Klassenfunktion implementiert werden.
- Wenn ein Operator Argumente unterschiedlichen Typs hat, dann ist er auch nur für genau diese Reihenfolge von Argumenten zuständig, z.B. kann mit `MatrixClass operator*(MatrixClass &A, double b)` zwar der Ausdruck `A = A * 2.1` geschrieben werden, aber nicht `A = 2.1 * A`
- Es gibt einen einfachen Trick um beides effizient zu implementieren: man definiert den kombinierten Zuweisungsoperator z.B. `operator*=` für die Multiplikation innerhalb der Klasse und zwei non-member Funktionen außerhalb, die diesen Operator verwenden.

Inkrement und Dekrement

- Es gibt sowohl Präfix als auch Postfixversionen von Inkrement und Dekrement
- Die Postfixversion (`a++`) wird als `operator++(int)` definiert, während die Präfixversion als `operator++()` kein Argument erhält. Das `int` Argument der Postfixversion wird nicht verwendet und dient nur zur Unterscheidung der beiden Varianten.
- Beachte, dass der Postfix Operator keine Referenz zurück liefern kann, da er eine Kopie des unveränderten Ausgangszustandes zurückgeben soll.

```
class Ptr_to_T
{
    T *p;

  public:
    Ptr_to_T &operator++();     // Praefixversion
    Ptr_to_T operator++(int);   // Postfixversion
}

Ptr_to_T &operator++(T &);      // Praefixversion
Ptr_to_T operator++(T &,int);   // Postfixversion
```

Die Klammeroperatoren

```
class MatrixClass
{
  public:
    double &operator()(int i, int j);
    std::vector<double> &operator[](int i);
    MatrixClass (int);
};
```

- Die Operatoren für runde und eckige Klammern können auch überladen werden. Damit können Ausdrücke wie `A[i][j]=12` oder `A(i,j)=12` geschrieben werden.
- Der Operator für eckige Klammern erhält immer genau ein Argument.
- Der Operator für runde Klammern kann beliebig viele Argumente erhalten.
- Beide können mehrfach überladen werden.

Konvertierungsoperatoren

```
class Complex
{
  public:
    operator double() const;
};
```

- Konvertierungsoperatoren werden benutzt um benutzerdefinierte Variablen in einen der eingebauten Typen zu verwandeln.
- Der Name eines Konvertierungsoperators ist `operator` gefolgt von dem Namen des Variablentyps zu dem der Operator konvertiert (durch ein Leerzeichen getrennt)
- Konvertierungsoperatoren sind konstante Methoden.

```
#include<iostream>
#include<cmath>

class Complex
{
  public:
    operator double() const
    {
        return sqrt(re_*re_+im_*im_);
    }
    Complex(double real, double imag) : re_(real), im_(imag)
    {};
  private:
    double re_;
    double im_;
};

int main()
{
    Complex a(2.0,-1.0);
    double b = 2.0 * a;
    std::cout << b << std::endl;
}
```

Selbstreferenz

- Jede Funktion einer Klasse kennt das Objekt von dem sie aufgerufen wurde.
- Jede Funktion einer Klasse bekommt einen Zeiger/eine Referenz auf dieses Objekt
- Der Name des Zeiger ist `this`, der Name der Referenz entsprechend `*this`
- Die Selbstreferenz ist z.B. notwendig für Operatoren die ein Objekt verändern:

```
MatrixClass &MatrixClass::operator*=(double x)
{
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]*=x;
    return *this;
}
```

2.2 Beispiel Matrixklasse

Dieses Beispiel implementiert eine Klasse für Matrizen.

- `matrix.h`: enthält die Definition der `MatrixClass`
- `matrix.cc`: enthält die Implementierung der Methoden der `MatrixClass`
- `main.cc`: ist eine Beispielanwendung für die Verwendung der `MatrixClass`

Header der Matrixklasse

```
#include<vector>

class MatrixClass
{
  public:
    void Resize(int numRows, int numCols);
    void Resize(int numRows, int numCols, double value);
    // access elements
    double &operator()(int i, int j);
    double operator()(int i, int j) const;
    std::vector<double> &operator[](int i);
    const std::vector<double> &operator[](int i) const;
    // arithmetic functions
    MatrixClass &operator*=(double x);
    MatrixClass &operator+=(const MatrixClass &b);
    std::vector<double> Solve(std::vector<double> b) const;
    // output
    void Print() const;
    int Rows() const
    {
        return numRows_;
    }
    int Cols() const
    {
        return numCols_;
    }

    MatrixClass(int numRows, int numCols) :
            a_(numRows), numRows_(numRows), numCols_(numCols)
    {
        for (int i=0;i<numRows_;++i)
            a_[i].resize(numCols_);
    };

    MatrixClass(int dim) : MatrixClass(dim,dim)
    {};

    MatrixClass(int numRows, int numCols, double value)
    {
        Resize(numRows,numCols,value);
    };

    MatrixClass(std::vector<std::vector<double> > a)
    {
        a_=a;
```

```
        numRows_=a.size();
        if (numRows_>0)
            numCols_=a[0].size();
        else
            numCols_=0;
    }

    MatrixClass(const MatrixClass &b)
    {
        a_=b.a_;
        numRows_=b.numRows_;
        numCols_=b.numCols_;
    }

  private:
    std::vector<std::vector<double> > a_;
    int numRows_ = 0;
    int numCols_ = 0;
};

std::vector<double> operator*(const MatrixClass &a,
                              const std::vector<double> &x);
MatrixClass operator*(const MatrixClass &a,double x);
MatrixClass operator*(double x,const MatrixClass &a);
MatrixClass operator+(const MatrixClass &a,const MatrixClass &b);
```

Implementierung der Matrixklasse

```
#include "matrix.h"
#include<iomanip>
#include<iostream>
#include<cstdlib>

void MatrixClass::Resize(int numRows, int numCols)
{
    a_.resize(numRows);
    for (size_t i=0;i<a_.size();++i)
        a_[i].resize(numCols);
    numRows_=numRows;
    numCols_=numCols;
}

void MatrixClass::Resize(int numRows, int numCols, double value)
{
    a_.resize(numRows);
    for (size_t i=0;i<a_.size();++i)
    {
        a_[i].resize(numCols);
        for (size_t j=0;j<a_[i].size();++j)
            a_[i][j]=value;
    }
    numRows_=numRows;
    numCols_=numCols;
}

double &MatrixClass::operator()(int i,int j)
{
```

```
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is (0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal column index " << i;
        std::cerr << " valid range is (0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i][j];
}

double MatrixClass::operator()(int i,int j) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is (0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal column index " << i;
        std::cerr << " valid range is (0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i][j];
}

std::vector<double> &MatrixClass::operator[](int i)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is (0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}

const std::vector<double> &MatrixClass::operator[](int i) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is (0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
```

```
}

MatrixClass &MatrixClass::operator*=(double x)
{
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]*=x;
    return *this;
}

MatrixClass &MatrixClass::operator+=(const MatrixClass &x)
{
    if ((x.numRows_!=numRows_)||(x.numCols_!=numCols_))
    {
        std::cerr << "Dimensions␣of␣matrix␣a␣(" << numRows_
                  << "x" << numCols_ << ")␣and␣matrix␣x␣("
                  << numRows_ << "x" << numCols_ << ")␣do␣not␣match!";
        exit(EXIT_FAILURE);
    }
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<x.numCols_;++j)
            a_[i][j]+=x[i][j];
    return *this;
}

std::vector<double> MatrixClass::Solve(std::vector<double> b) const
{
    std::vector<std::vector<double> > a(a_);
    for (int m=0;m<numRows_-1;++m)
        for (int i=m+1;i<numRows_;++i)
        {
            double q = a[i][m]/a[m][m];
            a[i][m] = 0.0;
            for (int j=m+1;j<numRows_;++j)
                a[i][j] = a[i][j]-q*a[m][j];
            b[i] -= q*b[m];
        }
    std::vector<double> x(b);
    x.back()/=a[numRows_-1][numRows_-1];
    for (int i=numRows_-2;i>=0;--i)
    {
        for (int j=i+1;j<numRows_;++j)
            x[i] -= a[i][j]*x[j];
        x[i]/=a[i][i];
    }
    return(x);
}

void MatrixClass::Print() const
{
    std::cout << "(" << numRows_ << "x";
    std::cout << numCols_ << ")␣matrix:" << std::endl;
    for (int i=0;i<numRows_;++i)
    {
        std::cout << std::setprecision(3);
        for (int j=0;j<numCols_;++j)
            std::cout << std::setw(5) << a_[i][j] << "␣";
        std::cout << std::endl;
```

```
    }
    std::cout << std::endl;
}

MatrixClass operator*(const MatrixClass &a,double x)
{
    MatrixClass temp(a);
    temp *= x;
    return temp;
}

MatrixClass operator*(double x,const MatrixClass &a)
{
    MatrixClass temp(a);
    temp *= x;
    return temp;
}

std::vector<double> operator*(const MatrixClass &a,
                              const std::vector<double> &x)
{
    if (x.size()!=a.Cols())
    {
        std::cerr << "Dimensions␣of␣vector␣" << x.size();
        std::cerr << "␣and␣matrix␣" << a.Cols() << "␣do␣not␣match!";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    std::vector<double> y(a.Rows());
    for (int i=0;i<a.Rows();++i)
    {
        y[i]=0.0;
        for (int j=0;j<a.Cols();++j)
            y[i]+=a[i][j]*x[j];
    }
    return y;
}

MatrixClass operator+(const MatrixClass &a,const MatrixClass &b)
{
    MatrixClass temp(a);
    temp += b;
    return temp;
}
```

Anwendung der Matrixklasse

```
#include "matrix.h"
#include<iostream>

int main()
{   // define matrix
    MatrixClass A(4,6,0.0);
    for (int i=0;i<A.Rows();++i)
        A[i][i] = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
        A[i+1][i] = A[i][i+1] = -1.0;
```

```
    MatrixClass B(6,4,0.0);
    for (int i=0;i<B.Cols();++i)
        B[i][i] = 2.0;
    for (int i=0;i<B.Cols()-1;++i)
        B[i+1][i] = B[i][i+1] = -1.0;
    // print matrix
    A.Print();
    B.Print();
    MatrixClass C(A);
    A = 2*C;
    A.Print();
    A = C*2.;
    A.Print();
    A = C+A;
    A.Print();

    const MatrixClass D(A);
    std::cout << "Element␣1,1␣of␣D␣is␣" << D(1,1) << std::endl;
    std::cout << std::endl;
    A.Resize(5,5,0.0);
    for (int i=0;i<A.Rows();++i)
        A(i,i) = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
        A(i+1,i) = A(i,i+1) = -1.0;
    // define vector b
    std::vector<double> b(5);
    b[0] = b[4] = 5.0;
    b[1] = b[3] = -4.0;
    b[2] = 4.0;
    std::vector<double>x = A*b;
    std::cout << "A*b␣=␣(␣";
    for (size_t i=0;i<x.size();++i)
        std::cout << x[i] << "␣␣";
    std::cout << ")" << std::endl;
    std::cout << std::endl;
    // solve
    x = A.Solve(b);
    A.Print();
    std::cout << "The␣solution␣with␣the␣ordinary␣Gauss␣Elimination␣is:␣(␣";
    for (size_t i=0;i<x.size();++i)
        std::cout << x[i] << "␣␣";
    std::cout << ")" << std::endl;
}
```

Output der Anwendung

```
(4x6) matrix:
    2    -1     0     0     0     0
   -1     2    -1     0     0     0
    0    -1     2    -1     0     0
    0     0    -1     2     0     0

(6x4) matrix:
    2    -1     0     0
   -1     2    -1     0
    0    -1     2    -1
    0     0    -1     2
```

```
    0      0      0      0
    0      0      0      0

(4x6) matrix:
    4     -2      0      0      0      0
   -2      4     -2      0      0      0
    0     -2      4     -2      0      0
    0      0     -2      4      0      0

(4x6) matrix:
    4     -2      0      0      0      0
   -2      4     -2      0      0      0
    0     -2      4     -2      0      0
    0      0     -2      4      0      0

(4x6) matrix:
    6     -3      0      0      0      0
   -3      6     -3      0      0      0
    0     -3      6     -3      0      0
    0      0     -3      6      0      0

Element 1,1 of D is 6

A*b = ( 14   -17   16   -17   14   )

(5x5) matrix:
    2     -1      0      0      0
   -1      2     -1      0      0
    0     -1      2     -1      0
    0      0     -1      2     -1
    0      0      0     -1      2

The solution with the ordinary Gauss Elimination is: ( 3   1   3   1   3   )
```

3 Speicherverwaltung

3.1 Speicherorganisation

Statischer Speicher

- Dort werden globale (auch innerhalb eines Namensbereiches globale) und statische Variablen angelegt.
- Der Speicherplatz wird bei Programmstart einmal reserviert und bleibt bis Programmende unverändert erhalten.
- Die Adresse von Variablen im statischen Speicher ändert sich während des Programmablaufs nicht.

Stack (oder automatischer Speicher)

- Dort werden lokale und temporäre Variablen angelegt (die z.B. bei Funktionsaufrufen oder für Rückgabewerte benötigt werden).
- Der Speicherplatz wird automatisch freigegeben wenn die Variable ihren Gültigkeitsbereich verlässt (z.B. beim Verlassen der Funktion in der sie definiert wurde).

- Die Größe des Stacks ist begrenzt (z.B. in Ubuntu per default 8192kb).

Heap (oder Freispeicher)

- Kann vom Programm mit dem Befehl `new` angefordert werden.
- Muss mit dem Befehl `delete` wieder freigegeben werden.
- Ist in der Regel nur durch die Größe des Hauptspeichers beschränkt
- Kann verloren gehen.

3.2 Variablen, Referenzen und Pointer

Variable

- Eine Variable bezeichnet eine Speicherstelle an der Daten eines bestimmten Typs abgelegt werden können.
- Eine Variable hat einen Namen und einen Typ.
- Für die Variable wird eine vom Typ abhängende Menge Speicherplatz reserviert (je nachdem in einem der drei Speicherbereiche)
- Die für einen bestimmten Variablentyp benötigte Menge Speicherplatz kann mit der Funktion `sizeof(variablentyp)` abgefragt werden.
- Jede Variable hat eine Speicheradresse, die mit dem Adressoperator `\&` abgefragt werden kann. z.B. definiert `int blub` eine Variable mit dem Namen `blub` und dem Typ `int`. Die Adresse der Variablen erhält man mit `\&blub` und `sizeof(int)` gibt ihre Größe unter 32bit Linux mit 4 an.
- Die Adresse einer Variablen kann nicht geändert werden.

Referenzen

- Eine Referenz definiert nur einen anderen Namen für eine bereits existierende Variable
- Der Typ einer Referenz ist der Typ der Variable gefolgt von einem `\&`
- Eine Referenz wird bei ihrer Definition initialisiert und kann danach nicht mehr geändert werden, sie zeigt also immer auf dieselbe Variable, z.B. `int &blae=blub`
- Eine Referenz kann genauso verwendet werden wie die ursprüngliche Variable.
- Änderungen der Referenz ändern auch den Inhalt der ursprünglichen Variablen.
- Es kann mehrere Referenzen auf dieselbe Variable geben.
- Eine Referenz kann auch mit einer Referenz initialisiert werden.

Beispiel Referenzen

```
#include<iostream>

int main()
{
    int a = 12;
    int &b = a;    // definiert eine Referenz
    int &c = b;    // ist erlaubt
    float &d = a; // nicht erlaubt, da nicht der gleiche Typ
    int e = b;
    b = 2;
    c = a * b;
    std::cout << a << std::endl;
    std::cout << e << std::endl;
}
```

Pointer

- Pointer sind ein sehr hardwarenahes Konzept
- In einem Pointer oder Zeiger kann die Adresse einer Variablen eines bestimmten Typs oder die Adresse einer Funktion gespeichert werden.
- Der Typ eines Variablenpointers ist der Typ der Variablen auf die er zeigen kann gefolgt von einem Stern `*`
- Der Inhalt eines Pointers ist die Speicheradresse einer Variablen, ändert man den Pointer so greift man auf andere Speicherbereiche zu
- Möchte man auf den Wert an dieser Speicheradresse zugreifen, dann setzt man ein `*` vor den Namen des Pointers
- Zeigt ein Pointer auf ein Objekt und möchte man auf Attribute oder Methoden des Objekts zugreifen, dann kann man den Operator `->` verwenden. Dabei sind `*a.value` und `a->value` äquivalent.
- Ein Pointer muss bei seiner Definition *nicht* initialisiert werden. Er zeigt dann einfach irgendwo hin.
- Zeigt ein Pointer auf einen Speicherbereich, der dem Programm nicht vom Betriebssystem zugewiesen wurde und liest oder schreibt man den Wert an dieser Adresse, wird das Programm mit der Fehlermeldung `segmentation fault` beendet.
- Um klar zu machen, dass ein Pointer im Moment nicht auf eine Variable/Funktion zeigt, weißt man ihm den Wert `0` zu. in C++11 gibt es dafür das Schlüsselwort `nullptr`.
- Es lässt sich dann einfach testen, ob ein Pointer gültig ist.
- Es gibt auch Pointer auf Pointer, z.B.

```
int a = 2;
int *b = &a;
int **c = &b;
```

- Die Increment- und Decrementoperatoren `++/--` erhöhen einen Pointer nicht um ein Byte, sondern um die Größe des Variablentyps auf den der Pointer zeigt (der Pointer zeigt dann also auf "das nächste" Element).
- Wen eine Zahl i zu einem Pointer addiert/von einem Pointer abgezogen wird, dann ändert sich die Speicheradresse um i mal die Größe des Variablentyps auf den der Pointer zeigt.

Beispiel Pointer

```
#include<iostream>

int main()
{
    int a = 12;
    int *b = &a;  // definiert einen Pointer auf a
    float  *c;    // definiert einen float Pointer (zeigt nach
                  // irgendwo)
    double *d=nullptr;  // besser so
    float e;
    c = &e;
    *b = 3;       // aendert Variable a
    b = &e;       // nicht erlaubt, falscher typ
    e = 2**b;     // erlaubt, aequivalent zu d = 2* a
    std::cout << b << std::endl;
    b = b+a;      // ist erlaubt, aber gefaehrlich
                  // c zeigt nun auf eine andere Speicherzelle
    std::cout << a << std::endl;
    std::cout << d << std::endl;
    std::cout << b << std::endl;
}
```

C-Arrays

- Felder in C sind mit Pointern eng verwandt.
- Der Name eines Felds in C ist gleichzeitig ein Zeiger auf das erste Element des Feldes
- Die Verwendung des eckige Klammeroperators `a[i]` entspricht einer Pointeroperation `*(a+i)`

```
#include<iostream>

int main()
{
    int numbers[27];
    for (int i=0;i<27;++i)
        numbers[i]=i*i;
    int *end=numbers+26;
    for (int *current=numbers;current<=end;++current)
        std::cout << *current << std::endl;
}
```

Gefahr von Pointern

Im Umgang mit Pointern und Feldern in C/C++ gibt es zwei große Gefahren:

1. Ein Pointer (insbesondere auch bei der Verwendung von Feldern) wird so geändert (aus Versehen oder absichtlich), dass er auf Speicherbereiche zeigt, die nicht alloziert wurden. Im besten Fall führt das zu einem Programmende auf Grund eines `segmentation fault`. Im schlimmsten Fall kann es dazu verwendet werden sich Zugriffsrechte auf das System zu verschaffen.

2. Es wird über ein Feld hinaus geschrieben. Wenn der betroffene Speicher vom Programm reserviert wurde (weil dort andere Variablen gespeichert sind) führt dies oft zu sehr merkwürdigen Fehlern, weil diese anderen Variablen auf einmal falsche Werte enthalten. In umfangreichen Programmen ist die Stelle an der das Überschreiben erfolgt oft schwer zu finden.

3.3 Call by Value und Call by Reference

Call by Value

Wird ein Argument an eine Funktion übergeben, dann wird von diesem Argument bei jedem Funktionsaufruf eine lokale Kopie auf dem Stack erstellt.

- Steht eine normale Variable in der Argumentliste, dann wird eine Kopie dieser Variablen erzeugt.
- Dies bezeichnet man als *Call by Value*.
- Änderungen der Variablen innerhalb der Funktion wirken sich *nicht* auf die originale Variable im aufrufenden Programm aus.
- Werden große Objekte als Variable übergeben, dann kann das Erzeugen der Kopie sehr teuer werde (Laufzeit, Speicherplatz).

```
double SquareCopy(double x)
{
    x = x * x;
    return x;
}
```

Call by Reference

- Stehen eine Referenz oder ein Pointer in der Argumentliste, dann werden Kopien der Referenz oder des Pointers erzeugt. Diese zeigen immer noch auf dieselbe Variable.
- Dies bezeichnet man als *Call by Reference*.
- Änderungen des Inhalts der Referenz oder der Speicherzelle auf die der Pointer zeigt wirken sich sehr wohl auf die originale Variable im aufrufenden Programm aus.
- Dies ermöglicht das Schreiben von Funktionen, die mehr als einen Wert als Ergebnis liefern und von Funktionen mit Ergebnis aber ohne Rückgabewert (Prozeduren).

- Sollen große Objekte als Argumente übergeben werden, eine Veränderung aber ausgeschlossen sein, dann bietet sich die Verwendung einer konstanten Referenz an z.B. `double Square(const double &x)`

```
void Square(double &x)
{
    x = x * x;
}
```

3.4 Dynamische Speicherverwaltung

Große Objekte oder Felder deren Größe erst zur Laufzeit bekannt sind, können mit Hilfe von `new` auf dem Heap alloziert werden.

```
class X
{
  public:
    X();       // argumentloser Konstruktor
    X(int n);
    ...
};

X *p = new X;      // argumentloser Konstruktor
X *q = new X(17); // mit int Argument
...
```

Objekte die mit `new` erzeugt werden haben keinen Namen (unnamed objects), nur eine Adresse im Speicher. Das hat zwei Konsequenzen

1. Die Lebensdauer des Objektes ist nicht festgelegt. Es muss explizit mit dem Befehl `delete` durch den Programmierer zerstört werden:

   ```
   delete p;
   ```

   Dies darf nur einmal pro reserviertem Objekt erfolgen.

2. Dagegen hat der Zeiger über den auf das Objekt zugegriffen wird meist eine begrenzte Lebensdauer.

⇒ Objekt und Zeiger müssen konsistent verwaltet werden.

Mögliche Probleme:

1. Der Zeiger existiert nicht mehr, das Objekt existiert noch ⇒ Speicher ist verloren, Programm wird immer größer.

2. Das Objekt existiert nicht mehr, der Zeiger schon ⇒ bei Zugriff folgt ein `segmentation fault`. Besonders gefährlich wenn mehrere Zeiger auf dasselbe Objekt existieren.

Allozieren von Feldern

- Felder werden alloziert indem man die Anzahl der Elemente in eckigen Klammern hinter den Variablentyp schreibt.

- Um Felder zu allozieren braucht eine Klasse einen argumentlosen Konstruktor.
- Felder werden mit `delete []` gelöscht. Da `new []` und `delete []` implementierungsabhängig sein können, z.B. wird bei manchen Implementierungen die Länge des Arrays vor den Daten gespeichert und ein Pointer auf die eigentlichen Daten zurückgegeben.

```
int n;
std::cin >> n;     // lese einen Wert von der Tastatur
X *pa = new X[n];
...
delete [] pa;
```

  ⇒ *Man darf die beiden Formen von `new` und `delete` nicht mischen. Für einzelne Variablen `new` und `delete` und für Felder `new []` und `delete []`.*

Freigeben von dynamisch alloziertem Speicher

- Werden `delete` oder `delete []` auf einen Pointer angewandt, der auf einen schon freigegebenen oder nicht reservierten Speicherbereich zeigt, führt dies zum `segmentation fault`.
- `delete` und `delete []` können gefahrlos auf einen Nullpointer angewandt werden.
- `malloc` und `free` sollten in C++ Programmen nicht verwendet werden.

3.5 Klassen mit dynamisch allozierten Mitgliedern

- Kann die Details der Verwendung dynamischen Speichers vor den Nutzern verbergen
- Behebt (richtig programmiert) einige der wichtigsten Nachteile dynamischen allozierten Speichers in C:
  - Call by value möglich
  - Objekte kennen ihre Größe
  - Wird ein Objekt zerstört kann der Destruktor dynamisch allozierten Speicher automatisch freigeben

Beispiel: Matrixklasse mit dynamischen Speicher

- Daten werden in einem zweidimensionalen dynamisch allozierten Array gespeichert.
- Statt dem vector of vectors erhält die Matrixklasse einen Pointer to Pointer of `double` als private member.

```
double **a_;
int numRows_;
int numCols_;
```

- Zu implementierende Methoden: Konstruktor(en), Destruktor, Copy-Konstruktor, Zuweisungsoperator

Konstruktoren

```
    MatrixClass() : a_(0), numRows_(0), numCols_(0)
    {};

    MatrixClass(int dim) : a_(0)
    {
        Resize(dim,dim);
    };

    MatrixClass(int numRows, int numCols) :
            a_(0)
    {
        Resize(numRows,numCols);
    };

    MatrixClass(int numRows, int numCols, double value) : a_(0)
    {
        Resize(numRows,numCols,value);
    };
```

Resize Methoden

```
void MatrixClass::Resize(int numRows, int numCols)
{
    Deallocate();
    a_= new double*[numRows];
    a_[0] = new double[numRows*numCols];
    for (int i=1;i<numRows;++i)
        a_[i]=a_[i-1]+numCols;
    numCols_=numCols;
    numRows_=numRows;
}

void MatrixClass::Resize(int numRows, int numCols, double value)
{
    Resize(numRows,numCols);
    for (int i=0;i<numRows;++i)
        for (int j=0;j<numCols;++j)
        a_[i][j]=value;
}
```

Destruktor

```
    ~MatrixClass()
    {
        Deallocate();
    };

    private:
    inline void Deallocate()
    {
        if (a_!=0)
        {
            if (a_[0]!=0)
                delete [] a_[0];
            delete [] a_;
```

```
        }
    }
```

Copy-Konstruktor und Zuweisungsoperator

Die Default Versionen von Copy-Konstruktor und Zuweisungsoperator erstellen eine direkte Kopie aller Variablen. Dies würde dazu führen, dass jetzt zwei Pointer auf dieselben dynamisch allozierten Daten zeigen.

```
MatrixClass(const MatrixClass &b) :
          a_(0)
{
    Resize(b.numRows_,b.numCols_);
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]=b.a_[i][j];
}

MatrixClass &operator=(const MatrixClass &b)
{
    Resize(b.numRows_,b.numCols_);
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]=b.a_[i][j];
    return *this;
}
```

Weitere Anpassungen

Es müssen noch die eckige Klammer Operatoren angepasst werden (das betrifft eigentlich nur den Rückgabetyp). Bei den runde Klammer Operatoren ist keine Änderung nötig:

```
double *operator[](int i);
const double *operator[](int i) const;
```

Auf die Implementierung von Matrix-Vektorprodukt und Gaussalgorithmus für diese Variante der Matrixklasse wird verzichtet.

Header der Matrixklasse

```
class MatrixClass
{
  public:
    void Resize(int numRows, int numCols, double value);
    void Resize(int numRows, int numCols);
    // access elements
    double &operator()(int i, int j);
    double operator()(int i, int j) const;
    double *operator[](int i);
    const double *operator[](int i) const;
    // arithmetic functions
    MatrixClass &operator*=(double x);
    MatrixClass &operator+=(const MatrixClass &b);
    // output
    void Print() const;
    int Rows() const
```

```
{
    return numRows_;
}
int Cols() const
{
    return numCols_;
}

MatrixClass &operator=(const MatrixClass &b)
{
    Resize(b.numRows_,b.numCols_);
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]=b.a_[i][j];
    return *this;
}

MatrixClass() : a_(0), numRows_(0), numCols_(0)
{};

MatrixClass(int dim) : a_(0)
{
    Resize(dim,dim);
};

MatrixClass(int numRows, int numCols) :
        a_(0)
{
    Resize(numRows,numCols);
};

MatrixClass(int numRows, int numCols, double value) : a_(0)
{
    Resize(numRows,numCols,value);
};

MatrixClass(const MatrixClass &b) :
         a_(0)
{
    Resize(b.numRows_,b.numCols_);
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]=b.a_[i][j];
}

~MatrixClass()
{
    Deallocate();
};

private:
inline void Deallocate()
{
    if (a_!=0)
    {
        if (a_[0]!=0)
            delete [] a_[0];
```

```
            delete [] a_;
        }
    }
    double **a_;
    int numRows_;
    int numCols_;
};

MatrixClass operator*(const MatrixClass &a,double x);
MatrixClass operator*(double x,const MatrixClass &a);
MatrixClass operator+(const MatrixClass &a,const MatrixClass &b);
```

Implementierung der Matrixklasse

```
#include "matrix.h"
#include<iomanip>
#include<iostream>
#include<cstdlib>

void MatrixClass::Resize(int numRows, int numCols)
{
    Deallocate();
    a_= new double*[numRows];
    a_[0] = new double[numRows*numCols];
    for (int i=1;i<numRows;++i)
        a_[i]=a_[i-1]+numCols;
    numCols_=numCols;
    numRows_=numRows;
}

void MatrixClass::Resize(int numRows, int numCols, double value)
{
    Resize(numRows,numCols);
    for (int i=0;i<numRows;++i)
        for (int j=0;j<numCols;++j)
        a_[i][j]=value;
}

double &MatrixClass::operator()(int i,int j)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal row index " << i;
        std::cerr << " valid range is (0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal column index " << i;
        std::cerr << " valid range is (0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i][j];
}
```

```
double MatrixClass::operator()(int i,int j) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    if ((j<0)||(j>=numCols_))
    {
        std::cerr << "Illegal␣column␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numCols_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i][j];
}

double *MatrixClass::operator[](int i)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}

const double *MatrixClass::operator[](int i) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}

MatrixClass &MatrixClass::operator*=(double x)
{
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]*=x;
    return *this;
}

MatrixClass &MatrixClass::operator+=(const MatrixClass &x)
{
    if ((x.numRows_!=numRows_)||(x.numCols_!=numCols_))
    {
        std::cerr << "Dimensions␣of␣matrix␣a␣(" << numRows_
```

```
                    << "x" << numCols_ << ") and matrix x ("
                    << numRows_ << "x" << numCols_ << ") do not match!";
        exit(EXIT_FAILURE);
    }
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<x.numCols_;++j)
            a_[i][j]+=x[i][j];
    return *this;
}

void MatrixClass::Print() const
{
    std::cout << "(" << numRows_ << "x";
    std::cout << numCols_ << ") matrix:" << std::endl;
    for (int i=0;i<numRows_;++i)
    {
        std::cout << std::setprecision(3);
        for (int j=0;j<numCols_;++j)
            std::cout << std::setw(5) << a_[i][j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

MatrixClass operator*(const MatrixClass &a,double x)
{
    MatrixClass temp(a);
    temp *= x;
    return temp;
}

MatrixClass operator*(double x,const MatrixClass &a)
{
    MatrixClass temp(a);
    temp *= x;
    return temp;
}

MatrixClass operator+(const MatrixClass &a,const MatrixClass &b)
{
    MatrixClass temp(a);
    temp += b;
    return temp;
}
```

Anwendung der Matrixklasse

```
#include "matrix.h"
#include<iostream>

int main()
{   // define matrix
    MatrixClass A(4,6,0.0);
    for (int i=0;i<A.Rows();++i)
        A[i][i] = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
```

```
        A[i+1][i] = A[i][i+1] = -1.0;
    MatrixClass B(6,4,0.0);
    for (int i=0;i<B.Cols();++i)
        B[i][i] = 2.0;
    for (int i=0;i<B.Cols()-1;++i)
        B[i+1][i] = B[i][i+1] = -1.0;
    // print matrix
    A.Print();
    B.Print();
    MatrixClass C(A);
    A = 2*C;
    A.Print();
    A = C*2.;
    A.Print();
    A = C+A;
    A.Print();
    const MatrixClass D(A);
    std::cout << "Element 1,1 of D is " << D(1,1) << std::endl;
    std::cout << std::endl;
    A.Resize(5,5,0.0);
    for (int i=0;i<A.Rows();++i)
        A(i,i) = 2.0;
    for (int i=0;i<A.Rows()-1;++i)
        A(i+1,i) = A(i,i+1) = -1.0;
    A.Print();
    const MatrixClass E(5,5,1.0);
    for (int i=0;i<E.Rows();++i)
        std::cout << E[i][i] << std::endl;
}
```

3.6 Statische Variablen und Methoden

- Manchmal haben Klassen Mitglieder, die für alle Objekte der Klasse zusammen nur einmal vorhanden sind.
- Diese Variablen haben den Typ `static`.
- Es gibt in einem Programm nur genau eine Version eines statischen Elementes (nicht eine Version pro Objekt). Es wird also auch nur einmal Speicher belegt.
- Methoden die nicht mit den Daten eines bestimmten Objektes arbeiten (sondern höchstens statische Variablen verwenden) können auch als statische Elementfunktionen definiert werden.
- Auf statische Attribute und Methoden kann einfach durch Voranstellen des Klassennamens gefolgt von zwei Doppelpunkten zugegriffen werden ohne ein Objekt anzulegen.
- (Nicht konstante) statische Attribute müssen außerhalb der Klasse initialisiert werden.

```
#include<iostream>
class NumericalSolver
{
    static double tolerance;
  public:
    static double GetTolerance()
```

```
    {
        return tolerance;
    }
    static void SetTolerance(double tol)
    {
        tolerance=tol;
    }
};

double NumericalSolver::tolerance = 1e-8;

int main()
{
    std::cout << NumericalSolver::GetTolerance() << std::endl;
    NumericalSolver::SetTolerance(1e-12);
    std::cout << NumericalSolver::GetTolerance() << std::endl;
}
```

3.7 C++11 und dynamische Speicherverwaltung

3.7.1 Move-Konstruktor und -Zuweisungsoperator

Problem: Wird ein Wert z.B. von einer Funktion zurückgegeben, dann werden z.T. mehrfach temporäre Objekte erzeugt. Die folgende Funktion erzeugt beim Beenden bis zu zwei temporäre Objekte:

```
double SquareCopy(double x)
{
    return x*x;
}
```

- Ein temporäres Objekt speichert das Resultat von `x*x`.
- Da dieses Objekt in der Funktion erzeugt wurde und beim Verlassen der Funktion gelöscht wird, wird eine Kopie für den Rückgabewert erzeugt.

Das Kopieren größere Datenmengen kann dabei durchaus zeitaufwändig sein. C++-Compiler optimieren dies meist weg (return value optimisation, RVO).

Idee: Da die temporären Objekte anschließend wieder zerstört werden ist es gar nicht notwendig die Daten zu kopieren. Sie können einfach übernommen werden. (Es gibt auch noch andere Anwendungen). In C++11 gibt es dafür explizite Konstrukte:

- Bei Move-Konstruktoren und Move-Zuweisungsoperatoren wird der Inhalt eines anderen (meist temporären) Objekts übernommen. Die Werte des anderen Objekts werden dabei mit (billig zu erzeugenden) Default-Werten überschrieben.
- Dies ist insbesondere möglich bei der Initialisierung von Objekten, bei der Übergabe von Funktionsargumenten und bei Rückgabewerten von Funktionen.
- Dem Compiler muss explizit mitgeteilt werden, dass Ressourcen übernommen werden dürfen. Dies geschieht mit dem Schlüsselwort `std::move()`, z.B.

```
MatrixClass a(10,10,1.0);
MatrixClass b = std::move(a); // jetzt ist b eine 10x10 Matrix
std::vector<double> x(10,1.0);
x=b.Solve(std::move(x));     // Funktionsaufruf
```

- Move-Konstruktoren (und -Zuweisungsoperatoren) werden bei C++11 automatisch erzeugt wenn für eine Klasse vom Benutzer kein Konstruktor, Move-Konstruktor, Zuweisungsoperator oder Destruktor definiert wurde und wenn die Erzeugung eines Move-Konstruktors trivial ist.
- In anderen Fällen kann die Erzeugung eines Default-Move-Konstruktors oder Zuweisungsoperators mit wie bei normalen Konstruktoren mit dem Schlüsselwort `default` erzwungen werden, z.B.:

  ```
  MatrixClass(MatrixClass &&) = default;
  ```

  (`MatrixClass &&` ist eine sogenannte r-value Referenz, die nur auf temporäre oder durch `std::move` gekennzeichnete Objekte verweisen kann und erst in C++11 eingeführt wurde)
- Ein Move-Konstruktor ist trivial, wenn:
  - Die Klasse weder virtuelle Funktionen noch virtuelle Basisklassen hat.
  - Der Move-Konstruktor für jede direkte Basisklasse der Klasse trivial ist.
  - Der Move-Konstruktor aller nicht-statischen Attribute trivial ist.
- Alle Standard-Datentypen die mit C kompatibel sind sind trivial movable.
- Das Move-Konzept funktioniert nicht nur für Speicher sondern auch für andere Ressourcen, z.B. Dateien oder Kommunikatoren.

C++11 und dynamische Speicherverwaltung

Smart Pointer

C++11 bietet eine Reihe von sogenannten Smart Pointern an, die bei der Verwaltung dynamischen Speichers helfen und sich insbesondere um die Freigabe allozierten Speichers kümmern. Es gibt drei verschiedene Varianten von Smart Pointern:

- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::weak_ptr<T>`

Das Templateargument `T` gibt den Typ des Objekts an auf den der Smart Pointer zeigt. Die C++11 Smart Pointer werden in der Headerdatei `memory` definiert.

`unique_ptr`

- Bei einem `unique_ptr` gibt es genau einen Zeiger dem die allozierten Daten gehören. Wird dieser Zeiger zerstört (z.B. weil die Funktion in der er definiert wurde beendet oder weil der Destruktor des Objekts zu dem er gehört aufgerufen wurde), dann wird der virtuelle Speicher freigegeben.

- Smart Pointer und normale Pointer (Raw Pointer) sollten nicht gemischt werden um die Gefahr des Zugriffs auf schon freigegebenen Speicher oder der doppelten Freigabe von Speicher zu vermeiden. Die Allokation des Speichers muss deshalb direkt im Konstruktoraufruf des `unique_ptr` stattfinden.
- Eine Zuweisung eines normalen Pointers auf einen Smart Pointer ist nicht möglich (wohl aber eine Übergabe im Konstruktor).

Beispiel `unique_ptr`

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

struct blub
{
  void doSomething()
  {}
};

int main()
{
   std::unique_ptr<int> test(new int);
   test = new int;  // nicht erlaubt: Zuweisung von raw pointer
   int a;
   test = &a;       // nicht erlaubt: Zuweisung von raw pointer
   std::unique_ptr<int> test5(&a); // erlaubt aber boese => Absturz
   *test = 2;       // normaler Zugriff auf Speicher
   std::unique_ptr<int> test2(test.release()); // Uebertragung an anderen
       Pointer
   test = std::move(test2);   // Zuweisung an anderen Pointer nur mit move

   test.swap(test2);   // Austausch mit anderem Pointer
   if (test == nullptr)          // Vergleich
       std::cout << "test␣ist␣nullptr" << std::endl;
   if (!test2)                   // Test auf Existenz eines Objekts
       std::cout << "test2␣ist␣nullptr" << std::endl;
   std::unique_ptr<int[]> test3(new int[32]);  // Array
   test3[7] = 12;                              // Arrayzugriff
   if (test3)                    // Zugriff auf raw pointer
       std::cout << "test3␣ist␣" << test3.get() << std::endl;
   test3.reset();                // Freigabe des Speichers
   if (!test3)
       std::cout << "test3␣ist␣nullptr" << std::endl;
   std::unique_ptr<blub> test4(new blub);  // Objekt allozieren
   test4->doSomething();         // Methodenzugriff
   std::unique_ptr<FILE, int(*)(FILE*)> filePtr(
    fopen("blub.txt", "w"), fclose);       // Datei anlegen und schliessen
}
```

`shared_ptr`

- `shared_ptr` zeigen auf gemeinsam genutzten Speicher.
- Mehrere `shared_ptr` können auf den gleichen Speicherbereich zeigen. Die Anzahl der `shared_ptr` wird dabei per reference counting gezählt. Der allozierte Speicher wird freigegeben sobald der letzte `shared_ptr` verschwindet.

- Die Funktionalität von `shared_ptr` ist ansonsten die von `unique_ptr`.
- Beim ersten Erzeugen eines `shared_ptr` wird ein Managerobjekt angelegt, das zum einen die allozierten Ressourcen verwaltet und zum anderen einen Zähler wie viele Pointer im Moment darauf zeigen.
- Bei jedem Kopieren des `shared_ptr` wird der Zähler erhöht, bei jedem Löschen oder Umsetzen eines `shared_ptr` wird er erniedrigt. Ist der Zähler Null werden die Ressourcen freigegeben.

`weak_ptr`

- Haben mehrere Objekte `shared_ptr` aufeinander, dann können sie sich künstlich am Leben erhalten, weil immer noch jeweils in einem Zirkel ein Pointer darauf existiert.
- Um so einen Kreis zu brechen wurde die Klasse `weak_ptr` geschaffen.
- Ein `weak_ptr` ist kein vollständiger Pointer. Er kann nicht dereferenziert werden und über ihn können auch keine Methoden aufgerufen werden.
- Ein `weak_ptr` beobachtet eine dynamisch allozierte Ressource nur. Mit ihm kann geprüft werden ob sie noch existiert.
- Muss auf die Ressource zugegriffen werden, kann mit der Methode `lock()` des `weak_ptr` ein `shared_ptr` auf die Ressource erzeugt werden. Dieser sichert dann die Existenz der Ressource so lange solange sie verwendet wird.
- Das Managerobjekt eines `shared_ptr` hat einen weiteren Zähler den sogenannten Weak Counter, der analog die erzeugten `weak_ptr` zählt. Während die allozierte Ressource freigegeben wird, wenn kein `shared_ptr` mehr darauf zeigt, wird das Managerobjekt freigegeben wenn kein `weak_ptr` mehr darauf zeigt.

`shared_ptr` auf `this`

- Manchmal braucht man einen Pointer auf `this`. Da man Smart Pointer und Raw Pointer nicht mischen sollte, ist dann ein `shared_ptr` auf `this` notwendig.
- Erzeugt man diesen naiv mit `shared_ptr<T> blub(*this)` dann wird ein neues Managerobjekt erzeugt und der Speicher des Objekts wird entweder nicht oder zu früh freigegeben.
- Stattdessen leitet man die Klasse von der Templateklasse `enable_shared_from_this<T>` ab. Einen Pointer auf `this` erzeugt man dann mit der Methode `shared_from_this`

  ```
  shared_ptr<T> blub = shared_from_this();
  ```

- Beim Erzeugen eines so abgeleiteten Objekts im Konstruktoraufruf eines `shared_ptr` wird innerhalb der Klasse ein `weak_ptr` auf das Objekt selbst gespeichert, von dem dann mit `shared_from_this` ein `shared_ptr` erzeugt wird.

Beispiel shared_ptr

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

class Base : public std::enable_shared_from_this<Base>
{
    void doSomething()
    {
        std::shared_ptr<Base> myObj = shared_from_this();
    }
};

class Derived : public Base
{};

int main()
{
   std::shared_ptr<int> testPtr(new int), testPtr2;
   testPtr2 = testPtr; // erhoeht shared count
   std::cout << testPtr.use_count() << std::endl; // Anzahl shared_ptr auf den
       int
   testPtr.reset();    // erniedrigt shared count, testPtr ist nullptr

   // Beispiel weak pointer
   std::weak_ptr<int> weakPtr = testPtr2; // erhoeht weak count
   testPtr = weakPtr.lock();
   if (testPtr)
       std::cout << "Objekt existiert noch" << std::endl;
   if (weakPtr.expired())
       std::cout << "Objekt existiert nicht mehr" << std::endl;
   std::shared_ptr<int> testPtr3(weakPtr); // wirft exception wenn objekt
       nicht mehr existiert
   // Casting von shared pointern
   std::shared_ptr<Base> basePtr(new Derived);
   std::shared_ptr<Derived> derivedPtr;
   derivedPtr = std::static_pointer_cast<Derived>(basePtr); // create casted
       smart pointer sharing ownership with original pointer


}
```

4 Konstante Werte

Konstante Variable

- Bei konstanten Variablen stellt der Compiler sicher, dass der Inhalt während des Programmablaufs nicht verändert wird.
- Konstante Variablen müssen gleich bei ihrer Definition initialisiert werden.
- Danach dürfen sie nicht mehr geändert werden.

```
const int numElements=100;     // Initialisierung
numElements=200;               // nicht erlaubt, da const
```

- Im Vergleich zu den Makros bei C sind konstante Variablen zu bevorzugen, da sie die strenge Typprüfung des Compilers erlauben.

Konstante Referenzen

- Auch Referenzen können als konstant definiert werden. Der Wert auf den die Referenz verweist kann dann (mit Hilfe der Referenz) nicht geändert werden.
- Auf konstante Variablen sind nur konstante Referenzen möglich (da diese sonst mit Hilfe der Referenz geändert werden könnten).

```
int numNodes=100;           // Variable
const int &nn=numNodes;     // Variable kann ueber nn nicht
                            // geaendert werden ueber
                            // numElements schon
const int numElements=100; // Initialisierung
int &ne=numElements;        // nicht erlaubt, sonst Konstantheit
                            // nicht mehr garantiert
const int &numElem=numElements; // erlaubt
```

- Konstante Referenzen sind eine gute Möglichkeit eine Variable ohne Kopieren an eine Funktion zu übergeben

```
MatrixClass &operator+=(const MatrixClass &b);
```

Konstante Pointer

Bei Pointern gibt es zwei verschiedene Arten der Konstantheit. Bei einem Pointer kann es verboten sein

- den Inhalt der Variablen auf die er zeigt zu ändern. Dies wird ausgedrückt durch Schreiben von const vor den Typ des Pointers:

```
char s[17];
const char *pc=s;  // Zeiger auf Konstante
pc[3] = 'c';       // Fehler, Inhalt konstant
++pc;              // erlaubt.
```

- die Adresse die in dem Pointer gespeichert zu ändern (dies entspricht dann einer Referenz). Dies wird dadurch gekennzeichnet, das ein const zwischen den Typ des Pointers und den Namen des Pointers geschrieben wird:

```
char * const cp=s; // Konstanter Zeiger
cp[3] = 'c';       // erlaubt.
++cp;              // Fehler, Zeiger konstant
```

- Natürlich gibt es die Kombination aus beidem (das entspricht einer konstanten Referenz):

```
const char * const cpc=s; // Konstanter Zeiger auf Konstante
cpc[3] = 'c';             // Fehler, Inhalt konstant
++cpc;                    // Fehler, Zeiger konstant
```

Konstante Objekte

- Auch Objekte können als konstant definiert werden.
- Der Nutzer geht davon aus, dass sich der Inhalt eines konstanten Objektes nicht ändert. Dies muss von der Implementierung garantiert werden.
- Deshalb ist es nicht erlaubt Methoden aufzurufen, die das Objekt verändern könnten.
- Funktionen die die Konstantheit nicht verletzen werden durch das hinzufügen des Schlüsselwortes `const` nach der Argumentliste gekennzeichnet.
- Das Schlüsselwort ist Teil des Namens. Es kann eine `const` und eine nicht-`const` Variante mit gleicher Argumentliste geben.
- Wichtig: das `const` muss auch bei der Definition der Methodes außerhalb der Klasse angegeben werden.
- Nur `const` Methoden können für konstante Objekte aufgerufen werden.

```
#include<iostream>
class X
{
  public:
    int blub() const
    {
        return 3;
    }
    int blub()
    {
        return 2;
    }
};

int main()
{
    X a;
    const X &b = a;
    std::cout << a.blub() << "␣" << b.blub() << std::endl;
    // ergibt die Ausgabe "2 3"
}
```

Natürlich ist das hier verwendete Verhalten irreführend und sollte so nicht verwendet werden.

Beispiel Matrixklasse

```
double *MatrixClass::operator[](int i)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
```

```
    }
    return a_[i];
}

const double *MatrixClass::operator[](int i) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}
```

Damit können wir schreiben:

```
MatrixClass A(4,6,0.0);
for (int i=0;i<A.Rows();++i)
    A[i][i] = 2.0;
const MatrixClass E(5,5,1.0);
for (int i=0;i<E.Rows();++i)
    std::cout << E[i][i] << std::endl;
```

Durch die Rückgabe eines Pointers auf eine Konstante wird verhindert, dass das Objekt implizit durch den Rückgabewert geändert wird:

```
A[2][3] = -1.0;    // ok. keine Konstante
E[1][1] = 0.0;     // Compiler Fehler
```

Physikalische und logische Konstantheit

Wann ist eine Methode `const`?

1. Objekt bleibt bitweise unverändert. So sieht das der Compiler (das ist alles was er kann) und versucht es sicherzustellen indem z.B. alle Datenmitglieder eines `const` Objektes ebenfalls als Konstanten behandelt werden. Dies wird auch als physikalische Konstantheit bezeichnet.

2. Objekt bleibt konzeptionell für den Benutzer einer Klasse unverändert. Dies wird als logische Konstantheit bezeichnet. Die Semantik kann der Compiler aber nicht überprüfen.

Physikalische Konstantheit und Pointer

- Bei unserem Beispiel der Matrixklasse mit dynamischer Speicherverwaltung haben wir zum Speichern der Matrix einen Pointer vom Typ `double **` verwendet.

- Wird dieser konstant erhalten wir einen Pointer vom Typ `double ** const`. Damit ist es allerdings nur verboten die Speicheradresse die im Pointer gespeichert ist zu ändern aber nicht die Einträge in der Matrix.

- Der Compiler beschwert sich nicht über die Definition:

```
double &MatrixClass::operator()(int i, int j) const;
```

Damit ist dann auch ohne Probleme das Ändern eines konstante Objekts möglich:

```
const MatrixClass E(5,5,1.0);
E(1,1)=0.0;
```

- Es ist sogar erlaubt die Einträge innerhalb der Klasse selbst zu ändern:

```
double &MatrixClass::operator()(int i,int j) const
{
    a_[0][0]=1.0;
    return a_[i][j];
}
```

Alternativen

- Verwendung eines STL-Containers wie in der ersten Variante der Matrixklasse:

```
std::vector<std::vector<double> >
```

- In einem `const` Objekt wird daraus ein `const std::vector<std::vector<double> >`.
- Bei Definition der Zugriffsfunktion

```
double &MatrixClass::operator()(int i, int j) const;
```

gibt der Compiler die Fehlermeldung

```
matrix.cc: In member function 'double&␣MatrixClass::operator()(int,␣int)␣
    const':
matrix.cc:63: error: invalid initialization of reference of type
    'double&' from expression of type 'const␣double'
```

- Auch eine Rückgabe ganzer Vektoren mit

```
std::vector<double> &MatrixClass::operator[](int i) const;
```

scheitert:

```
matrix.cc: In member function 'std::vector<double,␣std::allocator<double>␣
    >&␣MatrixClass::operator[](int)␣const':
matrix.cc:87: error: invalid initialization of reference of type
    'std::vector<double,␣std::allocator<double>␣>&' from expression of
    type 'const␣std::vector<double,␣std::allocator<double>␣>'
```

Merke: Mit Pointern lässt sich die Compilerfunktionalität zur Überwachen der physikalischen Konstantheit leicht aushebeln. Bei der Definition von `const` Methoden für Objekte die dynamisch allozierten Speichers verwenden ist deshalb besondere Vorsicht angebracht

Logische Konstantheit und Caches

- Manchmal ist es sinnvoll aufwändig zu berechnende Werte aufzuheben um bei wiederholter Verwendung Rechenzeit zu sparen.
- Wir fügen der Matrixklasse die beiden privaten Variablen `double norm_` und `bool normIsValid_` und sorgen dafür dass `normIsValid_` in den Konstruktoren immer mit `false` initialisiert wird.
- Dann lässt sich eine Unendlichnorm wie folgt implementieren:

```
double MatrixClass::InfinityNorm()
{
  if (!normIsValid_)
  {
      norm_=0.;
      for (int j=0;j<numCols_;++j)
      {
          double sum=0.;
          for (int i=0;i<numRows_;++i)
              sum += fabs(a_[i][j]);
          if (sum>norm_)
              norm_=sum;
      }
      normIsValid_=true;
  }
  return norm_;
}
```

- Diese Funktion macht auch für eine konstante Matrix Sinn und verletzt semantisch nicht die Konstantheit.
- Der Compiler lässt es aber nicht zu.

Lösung

- Man definiert die beiden Variablen als `mutable`.

```
mutable bool normIsValid_;
mutable double norm_;
```

- `mutable` Variablen lassen sich auch in `const` Objekten ändern.
- Dies sollte nur angewendet werden, wenn es wirklich notwendig ist und es die logische Konstantheit des Objekts nicht verändert.

Friend

In einigen Fällen kann es notwendig werden, dass andere Klassen oder Funktionen Zugriff auf die geschützten Member einer Klasse benötigen.

Beispiel: Einfach verkettete List

- `Node` enthält die Daten.

- `List` soll Daten der `Node` ändern können.
- Daten der `Node` sollen privat sein.
- `List` ist `friend` zu `Node` und darf damit auf private Daten zugreifen.
- Klassen und freie Funktionen können `friend` zu einer anderen Klasse sein.
- `friend` darf auf private Daten der Klasse zugreifen.

Beispiel `friend` Klasse:

```
class List;

class Node {
private:
    Node * next;
public:
    int value;
    friend class List;
};
```

Beispiel `friend` Funktion:

```
class MatrixClass
{
    friend MatrixClass invert(const MatrixClass &);
    // ...
};

...
MatrixClass A(10);
...
MatrixClass inv = invert(A);
```

- Fast alles, was man als Klassenmethode schreiben kann, kann man auch als freie `friend` Funktion programmieren.
- Alle Klassen und Funktionen die `friend` sind, gehören logisch automatisch zur Klasse, da sie auf deren internen Struktur aufbauen.
- Vermeiden Sie `friend` Deklarationen. Diese brechen die Kapselung auf und erhöhen den Pflegeaufwand.

5 Build-Systeme

- Komplexe Projekte bestehen aus verschiedenen Programmen und Bibliotheken.
- Jedes Programm/Bibliothek besteht aus vielen Dateien (Header- und Source-Dateien).
- Build-Systeme sollen helfen die Arbeit beim Compilieren zu erleichtern.

Ziel:

- Ein Build-System weiß, wie man aus den Dateien die Programme und Bibliotheken erzeugt.
- Bei Änderungen an einer Datei soll das Projekt aktualisiert werden
- Nicht alle Dateien müssen neu compiliert werden, daher:
  - So viele Dateien wie nötigen...
  - und so wenige Dateien wie möglich neu übersetzen.

Auswahl an Build-Systemen

Es gibt viele verschiedene System mit unterschiedlichem Funktionsumfang:

- make
- mk
- SCons
- ant
- jam
- Rant
- eingebaut in die IDE
- ...

Darüber hinaus gibt es Metasysteme, welche Eingabedateien für andere Systeme erzeugen:

- automake/autoconf
- cmake
- qmake
- mkmf
- ...

Makefiles

- `make` ist ein Programm das ermöglicht nur die Dateien zu übersetzen die seit der letzten Übersetzung geändert wurden
- Ein `Makefile` beschreibt die Dateien die zu einem Projekt gehören und wie sie übersetzt und gelinkt werden compiled and linked
- `Makefile`-Regeln werden in einer Funktionalen Sprache beschrieben.
  - Man beschreibt *targets*, welche von *prerequisits* abhängen.
  - *targets* und *prerequisits* entsprechen in der Regel Dateien.
  - Zu einzelnen *targets* gibt man Regeln an, wie diese aus den *prerequisits* erzeugt werden.
- `Makefile`-Regeln habe die Form

  ```
  target-name: prerequisits-list
          build-rule
  ```

  wobei die eigentliche Regel mit einem `TAB` eingerückt wird.

einfaches Makefile Beispiel

```
# the compiler we want to use
CXX=g++
CC=$(CXX)
# some more variables
CPPSRC=$(wildcard *.cpp)
OBJS=$(CPPSRC:.cpp=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# how to compile apps
%: %.o
        $(CXX) $? -o $@
# how to compile object files
%.o: %.cc
        $(CXX) $(CXXFLAGS) -c -o $@ $<
# we use implicit compilation rules

### Dependencies
dep: .depends
# include .depends if file exists
-include .depends
# how to create .depends
.depends: $(SRC)
        $(CXX) -MM $? > .depends

### cleanup
clean:
        rm -f $(APPS) $(OBJS)
```

fortgeschrittenes Makefile Beispiel

```
# the compiler we want to use
CXX=g++

# some more variables
CPPSRC=$(wildcard *.cpp)
OBJS=$(CPPSRC:.cpp=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# how to compile apps
%: %.o
        $(CXX) $? -o $@
# how to compile object files
%.o: %.cc
        $(CXX) $(CXXFLAGS) -c -o $@ $<
```

```
# the compiler we want to use
CXX=g++
CC=$(CXX)
# some more variables
CPPSRC=$(wildcard *.cpp)
OBJS=$(CPPSRC:.cpp=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# we use implicit compilation rules
```

```
# the compiler we want to use
CXX=g++
CC=$(CXX)
```

```
# some more variables
CPPSRC=$(wildcard *.cpp)
OBJS=$(CPPSRC:.cpp=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# we use implicit compilation rules

### Dependencies
dep: .depends
# include .depends if file exists
-include .depends
# how to create .depends
.depends: $(SRC)
        $(CXX) -MM $? > .depends

### cleanup
clean:
        rm -f $(APPS) $(OBJS)
```

- `make` unterstützt Variablen.
- Regeln können generisch formuliert werden für verschiedene `targets`,
- ... dafür gibt es gibt eine Reihe automatischer Variablen
- `GNU make` erlaubt verschiedene Spezialerweiterung (z.B. wildcard).
- `GNU make` hat bereits verschiedene Regeln eingebaut.
- `make` kann auch gleich aufräumen
- und mit Hilfe des Compilers die Abhängigkeiten automatisch überprüfen.

Weiter Informationen zu Makefiles

`http://www.sethi.org/classes/cet375/lab_notes/ lab_04_makefile_and_compilation.html http://myweb.stedwards`
`http://mrbook.org/blog/tutorials/make/ http://www.metalshell.com/view/tutorial/120/ http://www.eng.hawaii.edu/Tu`

Alternative: IDE's

- Integrated Development Environments (IDE's) Kombinieren die Eigenschaften eines Editors, eines Build-Systems und eines Debuggers
- z.B.: Eclipse C/C++ Development Environment (`http://www.eclipse.org/cdt`)
- Eclipse ist mächtig und open-source, aber komplex in der Bedienung.

Tipp: Multiple Header Inclusion Prevention

- Man kann Makros verwenden, um ein mehrfaches einbinden des selben Headers zu verhindern.
- Der Inhalt der Headers Datei wird in einen bedingten Block gesetzt:

```
#ifndef _MYSPECIALHEADERFILE_
#define _MYSPECIALHEADERFILE_
// content of header file
#endif
```

- Beim ersten Einbinden wird der Inhalt der Header Datei gelesen und das Makro definiert.
- Bei jedem weiteren Einbinden wird der Inhalt übersprungen, da das Makro bereits definiert ist.

6 Namespaces

- Namespaces erlaube es Klassen, Funktionen und globale Variablen unter einem Namen zu gruppieren. Auf diese Weise kann der globale Namensraum in Unterräume zerteilt werden von denen jeder einen eigenen Namen hat.
- Ein Namespace wird definiert mit:

```
namespace Name
{
// classes, functions etc. belonging to the namespace
}
```

  Dabei ist `Name` ein beliebiger Namen, der den Regeln für Variablen- und Funktionsnamen genügt.
- Um ein Konstrukt aus einem Namespace zu verwenden muss der Name des Namespace gefolgt von zwei Doppelpunkten vor den Namen des Konstrukts geschrieben werden. Z.b. `std::max(a,b)`.
- Jede Klasse definiert ihren eigenen Namespace.
- Mit dem Schlüsselword `using` wird einer oder alle Namen aus einem Namensraum in den aktuelle Namensraum übernommen. Ein häufig verwendetes Beispiel ist die Zeile

```
using namespace std;
```

  Nach dieser Zeile können alle Konstrukte im Namensraum `std` ohne Präfix verwendet werden. Z.B. `max(a,b)`. Dabei darf es zu keinen Uneindeutigkeiten kommen.

Beispiel

Namespaces sind besonders nützlich, wenn die Möglichkeit besteht, dass es in unabhängig voneinander entwickeltem Code zwei Klassen, globale Variablen oder Funktionen mit gleichem Namen (und bei Funktionen gleicher Argumentliste) gibt. Dies führt zu Fehler mit der Fehlermeldung `... redefined`. Mit Namespaces lässt sich das verhindern:

```
// namespaces
#include <iostream>

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main ()
{
  std::cout << first::var << endl;
  std::cout << second::var << endl;
  return 0;
}
```

7 Nested Classes

- Eine Klasse benötigt oft andere "Hilfsklassen".
- Diesen können implementierungsspezifisch sein und sollten dann außerhalb nicht sichtbar sein.
- Beispiele:
  - Listenelemente
  - Iteratoren
  - Exceptions (Fehlermeldungsobjekte, nächste Vorlesung)
- Man kann diese als Klassen innerhalb der Klasse (sogenannte nested classes) realisieren.
- Vorteile:
  - globaler Namensraum wird nicht "verschmutzt".
  - Zugehörigkeit zu der Klasse wird verdeutlicht.

```
class Outer
{
  public:
    ...
    class Inner1
    {
        ...
    };
  private:
    ...
    class Inner2
    {
        void foo();
    };
};
```

```
void Outer::Inner2::foo()
{
    ...
}
```

Beispiel: Implementierung einer Menge mittels Liste

```
class Set
{
  public:
    Set();                  // leere Menge
    ~Set();                 // Menge loeschen
    void Insert(double);    // (nur einmal) einfuegen
    void Delete(double);    // falls in Menge loeschen
    bool Contains(double); // true wenn enthalten
  private:
    struct SetElem
    {
        double item;
        SetElem *next;
    };
    SetElem *first;
};
```

`SetElem` kann nur innerhalb des `Set` verwendet werden, deswegen können alle Attribute `public` sein (Wir erinnern uns: `struct` ist `class` mit `public` als default).

8 Vererbung

- Klassen ermöglichen die Definition von Komponenten, die bestimmte Konzepte der realen Welt oder des Programms repräsentieren
- Durch Vererbung lässt sich die Beziehung zwischen verschiedenen Klassen ausdrücken. Z.B. haben die Klassen `Kreis` und `Dreieck` gemeinsam, das sie eine geometrische Form darstellen. Dies soll auch im Programm zum Ausdruck kommen.
- In C++ ist es möglich zu schreiben:

  ```
  class Form {...};
  class Kreis : public Form {...};
  class Dreieck : public Form {...};
  ```

  Die Klassen Kreis und Dreieck sind von Form abgeleitet, sie erben die Eigenschaften von Form.
- Es ist so möglich gemeinsame Eigenschaften und Verhaltensweisen von `Kreis` und `Dreieck` in `Form` zusammenzufassen. Dies ist eine neue Stufe von Abstraktion.
- Eine abgeleitete Klasse ist eine
  - Erweiterung der Basisklasse. Sie hat alle Eigenschaften der Basisklasse und fügt diesen noch welche hinzu.
  - Spezialisierung der Basisklasse. Sie repräsentiert in der Regel eine bestimmte Realisierung eines generellen Konzepts.

- Das Wechselspiel aus Erweiterung und Einschränkung macht die Mächtigkeit (aber auch manchmal die Komplexität) dieser Technik aus.

Protected Members

- Neben `private` und `public` Klassenmitgliedern gibt es eine dritte Kategorie: `protected`
- Auf `protected` Methoden und Attribute kann nicht von außerhalb sondern nur aus der Klasse selbst zugegriffen werden, wie bei `private`
- Allerdings werden `protected` Methoden und Attribute bei öffentlicher Vererbung wieder `protected`, d.h. Es kann auch von allen abgeleiteten Klassen auf sie zugegriffen werden.
- Es gibt die weit verbreitete Meinung, dass man `protected` nicht braucht und dass die Verwendung dieses Typs ein Hinweis auf Designfehler ist (wie z.B. fehlende Zugriffsfunktionen ...).

Beispiel

```
class A
{
  protected:
    int c;
    void f();
};

class B : public A
{
  public:
    void g();
};

B::g()
{
    int d=c;  // erlaubt
    f();      // erlaubt
}

int main()
{
    A a;
    B b;
    a.f();    // verboten
    b.f();    // verboten
}
```

Protected Konstruktoren

Mit Hilfe von `protected` kann man verhindern, dass sich Objekte einer Basisklassen anlegen lassen:

```
class B
{
  protected:
    B();
};

class D : public B
{
```

```
  public:
    D();      // ruft B() auf
};

int main()
{
    B b;  // verboten
    D d;  // erlaubt
}
```

8.1 Klassenbeziehungen und Vererbungsarten

Klassenbeziehungen

Ist-ein Klasse Y hat dieselbe Funktionalität (evtl. in spezialisierter Form) wie Klasse X. Objekt y (der Klasse Y) kann für x (der Klasse x) eingesetzt werden. Beispiel: Ein VW Käfer ist ein Auto

Hat-ein (Aggregation): Klasse Z besteht aus Unterobjekten der Typen X und Y. x hat ein y und ein z. Beispiel: Ein Auto hat einen Motor, Reifen, ...

Kennt-ein (Assoziation): Klasse Y hat Verweis (Zeiger, Referenz) auf Objekte der Klasse X. x kennt ein y, benutzt ein y. Beispiel: Ein Auto ist auf einen Menschen zugelassen (Er hat es, besteht aber nicht daraus, es ist kein Teil von ihm).

Man kann hat-ein mittels kennt-ein implementieren.

Öffentliche Vererbung

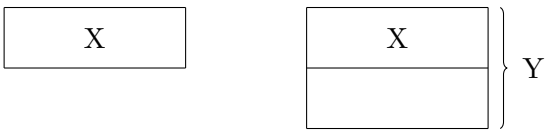
```
class X
{
  public:
    void a();
};

class Y : public X
{
  public:
    void b();
};
```

- Alle öffentlichen Mitglieder von X sind öffentliche Mitglieder von Y
- Implementierung wird übernommen, d.h.

  ```
  Y y;
  y.a();      // ruft Methode a von X auf
  ```

- Ist-ein-Beziehung



- Objekte der abgeleiteten Klasse können für Objekte der Basisklasse eingesetzt werden, dann ist aber nur der Basisklassenteil der Objekte zugänglich.

Slicing

```
class X
{
  public:
    void a();
};

class Y : public X
{
  public:
    void b();
};

int main()
{
    Y y;
    y.a();      // ruft Methode a des X-Teils von y auf
    X &x = y;
    x.a();      // ruft Methode a des X-Teils von y auf
    x.b();      // nicht erlaubt, nur Methoden von X
                // zugaenglich.
}
```

Wird ein Objekt der abgeleiteten Klassen call-by-value anstelle eines Objekts der Basisklasse übergeben, dann wird auch nur der Basisklassenteil kopiert.

Private Vererbung

```
class X
{
  public:
    void a();
};
```

```
class Y : private X
{
  public:
    void b();
};
```

- Alle öffentlichen Mitglieder von X sind private Mitglieder von Y
- Hat-ein Beziehung ist weitgehend gleichwertig zu:

  ```
  class Y
  {
    public:
      void b();
    private:
      X x;    // Aggregation
  }
  ```

  Deshalb ist private Vererbung auch nicht besonders essentiell.

- Benutzt man um eine Klasse mittels der anderen zu implementieren.

Protected Vererbung

```
class X
{
  public:
    void a();
};

class Y : protected X
{
  public:
    void b();
};
```

- Alle öffentlichen Mitglieder von X sind protected Mitglieder von Y
- Braucht man eigentlich nie.

Überblick Zugriffskontrolle bei Vererbung

Zugriffsrecht in der Basisklasse	Vererbungstyp öffentlich	protected	private
public	**public**	**protected**	**private**
protected	**protected**	**protected**	**private**
private	–	–	–

- Gibt es in der abgeleiteten Klasse eine Variable mit gleichen Namen oder eine Methode mit gleichem Namen und gleicher Argumentliste wie in der Basisklasse, dann überlagern diese die entsprechende Variable/Methode der Basisklasse.
- Ein Zugriff ist weiter möglich, wenn dem Variablen- oder Methodennamen der Name der Basisklasse als Namespace-Identifier vorangestellt wird und dies durch die Zugriffsrechte erlaubt ist.

8.2 Mehrfachvererbung

- Eine Klasse kann von mehr als einer Basisklasse abgeleitet werden.
- Gibt es Methoden oder Variablen mit dem gleichen Namen in beiden Basisklassen, dann müssen diese über den Namespace der entsprechenden Klasse identifiziert werden.
- Der Aufruf der Konstruktoren der Basisklassen richtet sich nach deren Reihenfolge in der Ableitung.
- Sollte nur in begründeten Ausnahmefällen verwendet werden. Oft lässt sich das gleiche Problem auch über eine hat-ein Beziehung lösen (also über ein entsprechendes Attribut).

```
#include<iostream>

class Zugmaschine
{
  public:
    float Gewicht()
```

```
    {
      return gewicht_;
    };
    Zugmaschine(float gewicht) : gewicht_(gewicht)
    {
      std::cout << "Zugmaschine initialisiert" << std::endl;
    };

  protected:
    float gewicht_;
};

class Auflieger
{
  public:
    float Gewicht()
    {
      return gewicht_;
    };
    Auflieger(float gewicht) : gewicht_(gewicht)
    {
      std::cout << "Auflieger initialisiert" << std::endl;
    };
  protected:
    float gewicht_;
};

class Sattelzug : public Zugmaschine, public Auflieger
{
  public:
    float Gewicht()
    {
        return Zugmaschine::gewicht_+Auflieger::gewicht_;
    }
    Sattelzug(float gewZug, float gewAuf) : Auflieger(gewAuf),
                                            Zugmaschine(gewZug)
    {
      std::cout << "Sattelzug initialisiert" << std::endl;
    };
};

int main()
{
  Sattelzug mikesLKW(10.0,25.0);
  std::cout << "Gewicht Sattelzug: " << mikesLKW.Gewicht() << std::endl;
  std::cout << "Gewicht Zugmaschine: " << mikesLKW.Zugmaschine::Gewicht() <<
      std::endl;
  std::cout << "Gewicht Auflieger: " << mikesLKW.Auflieger::Gewicht() <<
      std::endl;
}
```

Output:

```
Zugmaschine initialisiert
Auflieger initialisiert
Sattelzug initialisiert
Gewicht Sattelzug: 35
Gewicht Zugmaschine: 10
Gewicht Auflieger: 25
```

8.3 C++11: Final

C++11: Final

```
class X final
{
  public:
    void a();
};

class Y : public X  // Compilerfehler
{
  public:
    void b();
};
```

Bei C++11 lässt sich eine Klasse als `final` kennzeichnen. Danach ist eine weitere Ableitung von dieser Klasse nicht mehr erlaubt.

8.4 Vor- und Nachteile der Vererbung

Vorteile der Vererbung

Software reuse Gleiche Funktionen müssen nicht jedes mal neu programmiert werden. Spart Zeit und erhöht Sicherheit und Zuverlässigkeit.

Code sharing Code in der Basisklasse wird nicht in der abgeleiteten Klasse dupliziert. Fehler müssen nur einmal behoben werden.

Information Hiding Klasse kann ohne Kenntnis der Implementierungsdetails geändert werden.

Closed source Erweiterung Ist auch bei Klassen möglich, die nur als Binärcode zusammen mit einem Header mit Deklarationen verteilt werden.

Nachteile der Vererbung

Laufzeitgeschwindigkeit Aufruf aller Konstruktoren und Destruktoren beim Anlegen und Zerstören eines Objekts, evtl. höherer Speicherverbrauch, wenn abgeleitete Klasse nicht alle Eigenschaften der Basisklasse nutzt.

Programmgröße bei Verwendung allgemeiner Bibliotheken wird evtl. unnötiger Code eingebunden.

Programmkomplexität kann durch übertriebene Klassenhierarchien entstehen oder durch Mehrfachvererbung.

9 Exceptions

9.1 Fehlerbehandlung

Wenn in einem Programm in einer Funktion ein Fehler auftritt gibt es mehrere Möglichkeiten (auch Kombinationen sind möglich). Die Funktion

1. gibt eine Fehlermeldung aus.

2. versucht einfach weiterzumachen.
3. meldet den Fehler über einen Rückgabewert oder eine globale Variable
4. fragt den User um Hilfe.
5. beendet das Programm.

- Kombinationen aus den Varianten (1) bis (3) können zu einem unvorhersagbaren Programmablauf führen.
- Variante (4) ist nur in interaktiven Programmen möglich.
- Variante (5) ist unmöglich bei lebenswichtigen Systemen (z.B. Flugzeugsteuerung).

Problem

Eine Funktion kann oft nicht selbst entscheiden, was zu tun ist, wenn ein Fehler auftritt, da lokal nicht alle notwendigen Informationen zur Verfügung stehen um angemessen auf den Fehler zu reagieren.

Beispiel 1

- Simulationsprogramm fragt vom Nutzer die Anzahl der Gitterpunkte in x, y und z-Richtung ab.
- Das Hauptprogramm initialisiert ein Löserobjekt, das selbst wieder einen linearen Gleichungslöser anlegt, der eine Matrix benötigt. Dafür steht nicht genug Speicher zur Verfügung.
- Jetzt müsste der Nutzer vom Hauptprogramm aufgefordert werden eine kleinere Gittergröße zu wählen. Innerhalb des linearen Lösers ist dies nicht zu bewerkstelligen.

Beispiel 2

- Bei einer Transportsimulation konvergiert der lineare Löser innerhalb einer Newtoniteration nicht.
- Es gibt verschiedene Möglichkeiten damit umzugehen. Man könnte
  1. versuchen einen anderen (evtl. rechenaufwändigeren) linearen Löser zu verwenden.
  2. mit der momentan erreichten Konvergenz im Newtonverfahren weiter rechnen.
  3. die Zeitschrittweite verkleinern und den Zeitschritt neu rechnen.
  4. den Simulationslauf abbrechen.
- Diese Alternativen können nur auf unterschiedlichen Ebenen des Simulationsprogramms entschieden werden (z.B. Newtonverfahren, Zeitschrittsteuerung). Keine davon lässt sich lokal im linearen Löser anwenden.

9.2 Ausnahmen/Exceptions

- Ausnahmen/Exceptions können die Programmkontrolle über mehrere Aufrufebenen hinweg transferieren.
- Das aufrufende Programm entscheidet, ob es die Verantwortung für die Lösung eines Problems übernehmen will/kann.
- Dabei können Objekte eines beliebigen Typs übergeben werden (die z.B. nähere Informationen über das Problem enthalten).

Bei Exceptions wird die Fehlerbehandlung in zwei Teile zerlegt:

1. das Melden eines Fehlers, der sich lokal nicht beheben lässt.
2. die Behebung von Fehlern, die in Unterprogrammen aufgetreten sind.

Auslösen von Ausnahmen

- Tritt ein Fehler auf, wird eine Exception geworfen. Dazu wird mit der Anweisung `throw` ein Objekt eines beliebigen Typs erzeugt.
- Die Runtime-Umgebung geht dann nach und nach die aufrufenden Funktionen durch und sucht nach einem Programmteil, der die Verantwortung für Ausnahmen dieses Typs übernimmt.
- Alle lokalen Variablen in darunter liegenden Funktionen werden dabei zerstört. Für Objekte wird dabei der Destruktor aufgerufen.

```
MatrixClass &MatrixClass::operator+=(const MatrixClass &x)
{
    if ((x.numRows_!=numRows_)||(x.numCols_!=numCols_))
        throw std::string("Inkompatible␣Dimension␣der␣Matrizen");
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<x.numCols_;++j)
            a_[i][j]+=x[i][j];
    return *this;
}
```

Behandeln von Ausnahmen

- Ist eine Funktion bereit Ausnahmen in bestimmten Unterprogrammen zu behandeln, dann gibt sie das dadurch kund, dass der entsprechende Programmteil in einen `try` block eingeschlossen wird.
- Direkt darauf folgende `catch` blocks geben an, welche Ausnahmen behandelt werden können und wie darauf jeweils reagiert werden soll.

```
MatrixClass A(4,4,1.), B(4,4,2.);
try
{
    A += B;
}
catch (std::string error)
{
    if (error == "Inkompatible Dimension der Matrizen")
    {
        // irgendwas tun um Fehler zu beheben
    }
    else
        throw;  // Fehler weitergeben
}
```

Catch-Block

- Ein `catch` block wird ausgeführt, wenn
  - in einer der Anweisungen im `try` block `throw` ausgeführt wird.
  - `throw` ein Objekt des richtigen Typs wirft.
- Falls das Objekt im `catch` block nicht verwendet werden soll, braucht kein Name angegeben zu werden, der Typ ist ausreichend.
- Kann eine Ausnahme nicht oder nicht vollständig behandelt werden, kann sie mittels `throw;` weiter geworfen werden.

Throw

- `throw` erzeugt ein temporäres Objekt.
- sucht rückwärts auf dem call Stack das erste passende `catch`
- Findet sich keines wird das Programm durch Aufruf der Funktion `std::terminate()` beendet. Dabei wird eine Fehlermeldung ausgegeben, die den Objekttyp der Exception angibt, z.B. `terminate called after throwing an instance of 'std::string'`
- Wird das Programm so (also ohne Exception handler) beendet, dann ist nicht garantiert, dass die Destruktoren der Objekte aufgerufen werden (dies wird der Implementierung überlassen). Dies kann problematisch sein, wenn z.B. Dateien in den Destruktoren geschlossen werden sollen.

Deklaration von Ausnahmen

```
MatrixClass &operator+=(const MatrixClass &x) throw(std::string);
```

- Bei der Deklaration einer Funktion kann angegeben werden, welche Arten von Ausnahmen diese werfen kann.
- Dies erleichtert dem Programmierer dafür zu sorgen dass alle möglichen Ausnahmen auch behandelt werden.

- Wird innerhalb der Funktion eine andere Ausnahme geworfen, so wird `std::unexpected()` aufgerufen, das dann standardmäßig `std::terminate()` und dieses wiederum `std::abort()` aufruft. `std::unexpected` kann mit Hilfe der Funktion `set_unexpected` durch eine eigene Funktion ersetzt werden (Verwendung siehe unten bei `set_new_handler`).
- Die Ausnahmespezifikation muss bei allen Funktionsdeklarationen und der Funktionsdefinition wiederholt werden.
- Ist die Klammer hinter `throw` leer, so können von der Funktion keine Ausnahmen geworfen werden.

C++11: noexcept

- Die Deklaration von Ausnahmen mit Hilfe von `throw()` gilt in C++11 als veraltet (deprecated) und sollte laut Standard nicht mehr verwendet werden.
- Es gibt mehrere Gründe dafür:
  - Ausnahmespezifikationen werden erst zur Laufzeit überprüft, es gibt also keine Garantie, dass keine anderen Ausnahmen auftreten.
  - Ausnahmespezifikationen machen das Programm langsamer, da auf unerwartete Ausnahmen getestet werden muss.
  - Wenn die Funktion eine unerwartete Ausnahme wirft, wird das Programm auf eine suboptimale und unerwartete Weise beendet, die nicht wirklich abzufangen ist.
  - Ausnahmespezifikationen machen für Templatefunktionen keinen Sinn, da noch unbekannt ist, welche Ausnahmen z.B. der Konstruktor eines Typs werfen kann.
- In C++11 ist es nur noch möglich zu anzugeben ob eine Funktion *keine* Ausnahmen wirft. Dafür gibt es das keyword `noexcept`.
- Wird in einer Funktion die als `noexcept` definiert ist doch eine Exception geworfen wird anschließend *immer* `std::terminate()` aufgerufen. Es gibt also keinen Mehraufwand zur Laufzeit.
- Es gibt zwei Varianten von `noexcept` eine bedingte und eine bedingungslose.
- Bei der bedingungslosen Variante wird einfach dem Funktionskopf das Schlüsselwort `noexcept` nachgestellt.

```
MatrixClass &operator+=(const MatrixClass &x) noexcept;
```

- Der neue Operator `noexcept()` liefert `false` zurück, wenn der Ausdruck innerhalb der Klammern möglicherweise eine Exception werfen könnte, ansonsten `true`.
- Dies kann z.B. für Optimierungen verwendet werden, z.B. verwendet `std::vector` teilweise die move-Semantik nur, wenn der move-Konstruktor der Elemente `noexcept` ist und legt ansonsten Kopien an.

- Außerdem kann der Operator auch für die bedingte Variante von Funktionen ohne Exceptions verwendet werden, die insbesondere für die Verwendung mit Templates gedacht ist. Hier kommt hinter dem Schlüsselwort `noexcept` in Klammern eine Bedingung, bei der z.B. verlangt wird, dass bei bestimmten Operationen keine Ausnahmen geworfen werden können.

```
#include<iostream>

template<class T>
  T add(T a, T b) noexcept( noexcept(T(a+b)) )
{
  return a + b;
}

int main()
{
  int a,b;
  a = b = 1;
  if (noexcept(add(a,b)))
    std::cout << "exception␣safe,␣result␣is:␣" << add(a,b) << std::endl;
  else
    std::cout << "not␣exception␣safe" << std::endl;
  return 0;
}
```

Gruppieren von Ausnahmen

```
class Matherr {};
class Underflow : public Matherr {};
class Overflow : public Matherr {};
class DivisionByZero : public Matherr {};

void g()
{
    try
    {
        f();
    }
    catch (Overflow)
    {
        // alle Overflow-Fehler hier behandeln
    }
    catch (Matherr)
    {
        // alle anderen mathematischen Fehler hier
    }
}
```

Alle von der Standardbibliothek geworfenen Ausnahmen sind von der Klasse `std::exception`

Tritt ein Fehler auf, der mehrere Folgen hat, kann das durch Mehrfachvererbung zum Ausdruck gebracht werden.

```
class NetworkFileError : public NetworkError, public FileSystemError
{};
```

Dies beschreibt einen Fehler, der beim Zugriff auf eine über ein Netzwerk geöffnete Datei auftritt. Dies ist sowohl ein Netzwerkfehler, als auch ein Fehler beim Zugriff auf das Dateisystem.

Fangen aller Ausnahmen

`catch(...)` fängt alle Ausnahmen, egal welches Objekt geworfen wurde, erlaubt aber keinen Zugriff auf den Inhalt des Objektes. Dies kann z.B. verwendet werden um lokal aufzuräumen, bevor die Ausnahme weiter geworfen wird:

```
try
{
    f();
}
catch (...)
{
    // Aufraeumen
    throw;
}
```

Achtung: Folgen mehrere `catch` Blöcke aufeinander müssen sie vom Speziellen zum Allgemeinen geordnet sein.

9.3 Ausnahmen bei der Speicherverwaltung

- Ein häufiger Fall für das Auslösen von Ausnahmen ist, dass mehr Speicher alloziert werden soll, als verfügbar ist.
- Erhält `new` nicht genug Speicher vom Betriebssystem versucht es erst die Funktion `new_handler()` aufzurufen, die vom User definiert werden kann. Diese könnte z.B. versuchen bereits allozierten Speicher freizugeben.

```
#include<iostream>
#include<cstdlib>

void noMoreMemory()
{
    std::cerr << "unable to allocate enough memory" << std::endl;
    std::abort();
}  // dies ist keine gute Loesung! new handler wird von new
   // mehrmals aufgerufen und soll versuchen Speicher freizugeben

int main()
{
    std::set_new_handler(noMoreMemory);
    int *big = new int[1000000000];
}
```

- Wenn `new_handler()` nicht definiert ist wird die Ausnahme `std::bad_alloc` geworfen.

```
#include<new>

int main()
{
```

```
    int *values;
    try
    {
        values = new int[1000000000];
    }
    catch (std::bad_alloc)
    {
        // do something
    }
}
```

9.4 Multiple Resource Allocation

Oft (besonders in Konstruktoren) müssen mehrmals nacheinander Ressourcen alloziert werden (Öffnen von Dateien, Allozieren von Speicher, Betreten eines Locks beim Multithreading):

```
void acquire()
{
    // acquire resource r1
    ...
    // acquire resource r2
    ...
    // acquire resource rn
    ...
    use r1...rn
    //Freigabe in umgekehrter Reihenfolge
    // release resource rn
    ...
    // release resource r1
    ...
}
```

Problem

- wenn `acquire rk` fehlschlägt, dann müssen `r1,...rk-1` freigegeben werden bevor man abbrechen kann, sonst entsteht ein Ressourcenleck.
- was wenn allozieren der Ressource eine Exception auslöst, die weiter außen abgefangen wird? was passiert dann mit `r1,...rk-1`?
- Variante:

```
class X
{
  public:
    X();
  private:
    A *pointerA;
    B *pointerB;
    C *pointerC;
};
```

```
X::X()
{
    pointerA = new A;
    pointerB = new B;
    pointerC = new C;
}
```

Lösung

"Resource acquisition is initialization"

- ist eine Technik die das obige Problem löst
- Beruht auf
  - Eigenschaften von Konstruktoren und Destruktoren
  - Ihre Interaktion mit exception handling.

Regeln für Konstruktoren/Destruktoren

1. Erst wenn der Konstruktor beendet ist, ist ein Objekt vollständig konstruiert.
2. Ein ordentlicher Konstruktor hinterlässt das System möglichst so, wie es vor dem Aufruf war, falls es nicht erfolgreich beendet werden kann.
3. Besteht ein Objekt aus Unterobjekten, so ist es soweit konstruiert, wie seine Teile konstruiert sind.
4. Wenn ein Block verlassen wird, wird für alle erfolgreich konstruierten Objekte der Destruktor aufgerufen.
5. Werfen einer Exception bewirkt das Verlassen aller Blöcke bis zum Block in dem das korrespondierende `catch` gefunden wird.

```
class A_ptr
{
  public:
    A_ptr()
    {
      pointerA = new A;
    }
    ~A_ptr()
    {
      delete pointerA;
    }
    A *operator->()
    {
        return pointerA;
    }
  private:
    A *pointerA;
};

// entsprechende Klassen
// B_ptr und C_ptr
```

```
class X
{
  // kein Konstruktor und
  // Destruktor noetig, da
  // Defaultvarianten
  // ausreichend
  private:
    A_ptr pointerA;
    B_ptr pointerB;
    C_ptr pointerC;
};

int main()
{
    try
    {
        X x;
    }
    catch (std::bad_alloc)
    {
        ...
    }
}
```

- Konstruktor `X()` ruft Konstruktoren von `pointerA`, `pointerB` und `pointerC` auf.
- Wird eine Exception bei `pointerC` geworfen, so werden die Destruktoren von `pointerA` und `pointerB` aufgerufen und dann der Code im `catch` block ausgeführt
- Analog lässt sich das auch für die Allokation anderer Ressourcen (z.B. Öffnen von Dateien) implementieren.

```
#include<memory>

class A
{};

class B
{};

class C
{};

class X
{
  public:
    X() : pointerA(new A),
          pointerB(new B),
          pointerC(new C)
    {}
    // kein Destruktor noetig, da
        Defaultvariante ausreicht
  private:
    std::unique_ptr<A> pointerA;
    std::unique_ptr<B> pointerB;
    std::unique_ptr<C> pointerC;
};

int main()
{
    try
    {
        X x;
    }
    catch (std::bad_alloc)
    {
    }
}
```

9.5 Designprinzipien der Ausnahmebehandlung in C++

Grundannahmen für das Design von Ausnahmebehandlung in C++

1. Exceptions werden vorwiegend zur Fehlerbehandlung verwendet.
2. Es gibt wenige Exception Handler im Vergleich zu Funktionsdefinitionen.
3. Exceptions treten im Vergleich zu Funktionsaufrufen selten auf.
4. Exceptions sind ein Sprachmitteln nicht nur eine Konvention zur Fehlerbehandlung.

Konsequenzen

- Exceptions sind nicht nur eine Alternative zum return-Mechanismus, sondern ein Mechanismus zur Konstruktion fehlertoleranter Systeme.
- Nicht jede Funktion muss eine fehlertolerante Einheit sein. Stattdessen können ganze Subsysteme fehlertolerant sein, ohne dass jede Funktion diese Funktionalität implementieren muss.
- Exceptions sollen nicht der alleinige Mechanismus zur Fehlerbehandlung sein, sondern nur eine Erweiterung für Fälle, die sich nicht lokal lösen lassen.

Ideale für die Ausnahmebehandlung in C++

1. Typ-sichere Weitergabe von beliebigen Informationen vom throw-point zum Handler.
2. Verursache keine Kosten (zur Laufzeit oder im Speicher) wenn keine Exception geworfen wird.
3. Garantiere dass jede Exception von einem geeigneten Handler gefangen wird.
4. Erlaube das Gruppieren von Exceptions.
5. Der Mechanismus soll in Multi-threaded Programmen funktionieren.

6. Kooperation mit anderen Sprachen (C) soll möglich sein.
7. Einfache Benutzung.
8. Einfache Implementierung.

(3) und (8) wurden später als zu teuer bzw. zu einschränkend angesehen und sind nur ansatzweise erreicht.

Die Bezeichnung `throw` wurde gewählt, weil raise und signal schon an C library Funktionen vergeben waren.

Resumption oder Termination

Während des Entwurfs der Exceptions wurde diskutiert, ob die Semantik der Exceptions terminierend oder wiederaufnehmend sein sollte. Wiederaufnahme (Resumption) bedeutet: Eine Routine wird wegen Speichermangel gestartet, findet neuen Speicher und kehrt dann an die Stelle des Aufrufs zurück. Oder die Routine wird gestartet, weil das CD-ROM Laufwerk leer ist, bittet den Benutzer die CD einzulegen und kehrt zurück.

Hauptgründe für Wiederaufnahme:

- Wiederaufnahme ist ein allgemeinerer Mechanismus als Termination.
- Im Fall von blockierten Ressourcen (CD-ROM fehlt, ...) bietet Wiederaufnahme eine elegante Lösung.

Hauptgründe für Termination:

- Ist deutlich einfacher.
- Die Behandlung von knappen/fehlenden Ressourcen mit Wiederaufnahme führt zu fehleranfälligen und schwer zu verstehenden Programmen wegen der engen Verbindung von Bibliotheken und Benutzern.
- Große Softwaresysteme wurden ohne Wiederaufnahme geschrieben, sie ist also nicht unbedingt nötig, z.B. gibt es bei Xerox Cedar/Mesa einer fossilen Programmiersprache, die Wiederaufnahme unterstützen sollte, $\sim$ 500.000 Zeilen Programmcode, aber Resumption nur an einer (!) Stelle, alle anderen Verwendungen von Resumption mussten nach und nach durch Termination ersetzt werden.

$\Rightarrow$ **Der Standard in C++ ist deshalb Termination.**

9.6 C++11: Exception Pointer

- In C++11 wurde ein eigene Art von Pointer `std::exception_ptr` eingeführt, mit dem man Ausnahmen speichern und zur späteren Behandlung auch weitergeben kann. Er kann *alle* Typen von Ausnahmen aufnehmen.
- Ein Pointer auf die aktuelle geworfene Ausnahme kann in einem `catch` Block mit der Funktion `std::current_exception()` erhalten werden. Alternativ lässt sich aus einem Exception-Objekt in einem `catch` Block mit Hilfe der Funktion `std::make_exception_ptr` ein `std::exception_ptr` generieren.

- Mit der Funktion `std::rethrow_exception`, die als Argument einen `std::excpetion_ptr` erwartet, lässt sich die Ausnahme erneut werfen und dann entsprechend behandeln.
- Exception pointer sind insbesondere für Multithreading relevant (siehe dort).

Beispiel

```
#include <iostream>
#include <exception>

void verarbeiteEptr(std::exception_ptr exPtr) // Weitergabe des Eptr
{
    try {
        if (exPtr != nullptr) {
            std::rethrow_exception(exPtr);
        }
    } catch(const std::string& e) {       // Behandlung der Ausnahme
        std::cout << "Exception␣\"" << e << "\"␣gefangen" << std::endl;
    }
}

int main()
{
    std::exception_ptr exPtr;
    try {
        throw(std::string("blub"));
    } catch(...) {
        exPtr = std::current_exception(); // capture
    }
    verarbeiteEptr(exPtr);
} // erst hier wird der Destruktor fuer die gespeicherte Ausnahme aufgerufen
```

9.7 Exceptions und Assertions

- Schon in C gab es das Sprachmittel der Assertions. Dabei gibt es ein Macro `assert` dessen Argument ein Vergleich ist. Ist der Vergleich falsch, wird das Programm mit einer Fehlermeldung vom Typ

  ```
  Assertion failed: expression, file filename, line line number
  ```

  beendet.
- Das Macro wird im Headerfile `assert.h` definiert.
- Wird beim Übersetzen des Programms die Variable `NDEBUG` gesetzt (entweder über ein `#define NDEBUG` in einem Sourcfile oder über ein `-DNDEBUG` beim Compileraufruf), werden alle Assertions ignoriert.
- Assertions dienen also vor allem dazu Programmierfehler abzufangen. Sie werden in der Endversion eines Programms aus Performancegründen in der Regel deaktiviert.
- Dagegen dienen Exceptions dazu Fehler während eines normalen Programmlaufes zu behandeln, insbesondere solche, die sich automatisch beheben lassen.

Beispiel

Programm:

```
#include<assert.h>
#include<iostream>

int dividiere(int a, int b)
{
  assert(b!=0);
  return (a/b);
}

int main()
{
  int a,b;
  a = 1;
  b = 0;
  std::cout << "Der Quotient ist: " << dividiere(a,b) << std::endl;
}
```

Ausgabe:

```
Assertion failed: (b!=0), function dividiere, file assert.cc, line 6.
Abort trap: 6
```

Ausgabe übersetzt mit `-DNDEBUG`

```
Floating point exception: 8
```

10 Dynamischer Polymorphismus

10.1 Virtuelle Funktionen

Objekte einer abgeleiteten Klassen können (wie oben gezeigt) als Argument für Funktionen verwendet werden, die Objekte der Basisklasse erwarten, aber

- es wird nur der Basisklassenanteil kopiert, wenn das Argument als call-by-value übergeben wird (slicing).
- es ist nur der Basisklassenanteil zugreifbar, wenn das Argument per Referenz übergeben wird.

Insbesondere heißt dass, auch wenn eine Funktion in einer abgeleiteten Klasse redefiniert wurde, immer die Funktion der Basisklasse aufgerufen wird.

Probleme mit Vererbung

```
#include<iostream>

class A
{
    public:
        int doWork(int a)
        {
            return(a);
        }
};

class B : public A
```

```
{
    public:
        int doWork(int a)
        {
            return(a*a);
        }
};

int doSomeOtherWork(A &object)
{
    return(object.doWork(2));
}

int main()
{
    A objectA;
    B objectB;
    std::cout << objectA.doWork(2) << ",␣";
    std::cout << objectB.doWork(2) << ",␣";
    std::cout << doSomeOtherWork(objectA) << ",␣";
    std::cout << doSomeOtherWork(objectB) << std::endl;
}
```

Die Ausgabe dieses Programmes ist:

```
2, 4, 2, 2
```

Virtuelle Funktionen

```
#include<iostream>

class A
{
    public:
        virtual int doWork(int a)
        {
            return(a);
        }
};

class B : public A
{
    public:
        int doWork(int a)
        {
            return(a*a);
        }
};

int doSomeOtherWork(A &object)
{
    return(object.doWork(2));
}

int main()
{
    A objectA;
    B objectB;
```

```
    std::cout << objectA.doWork(2) << ", ";
    std::cout << objectB.doWork(2) << ", ";
    std::cout << doSomeOtherWork(objectA) << ", ";
    std::cout << doSomeOtherWork(objectB) << std::endl;
}
```

Die Ausgabe des Programms mit virtuellen Funktionen ist:

```
2, 4, 2, 4
```

- Wenn eine Funktion in der Basisklasse als `virtual` deklariert wird, dann wird die Funktion der abgeleiteten Klasse aufgerufen, auch wenn die Methode der abgeleiteten Klasse über eine Referenz oder einen Pointer mit dem Typ der Basisklasse aufgerufen wird.
- Die Form der Funktionsdefinition in der abgeleiteten Klasse muss der in der Basisklasse *exakt* entsprechen, sonst wird die Funktion normal überladen.
- Der Rückgabewert der Klasse darf sich unterscheiden, wenn es sich beim Rückgabetyp um eine von der Basisklasse abgeleitete Klasse handelt.
- Man bezeichnet dies als Polymorphismus (Vielgestaltigkeit).

C++11: Override

- Wenn die Form der Funktionsdefinition in der abgeleiteten Klasse der in der Basisklasse nicht *exakt* entspricht, wird die Funktion normal überladen.
- Das ist oft eher die Folge eines Schreibfehlers und nicht beabsichtigt.
- Schreibt man auch in der abgeleiteten Klasse ein `virtual` vor die Funktion, hat das nur die Folge, dass diese überladene Funktion in Zukunft auch virtuell ist.
- In C++11 gibt es das zusätzliche Schlüsselwort `override`. Wird dieses *nach* dem Funktionskopf in einer abgeleiteten Klasse geschrieben, dann gibt es einen Compilerfehler, wenn die Funktion keine virtuelle Funktion der Basisklasse redefiniert.
- Es ist ratsam dieses Schlüsselwort wo immer möglich zu verwenden.

```
class B : public A
{
    public:
        int doWork(int a) override
        {
            return(a*a);
        }
};
```

Wird eine Methode durch direkte Angabe des Namespaces (scoping) aufgerufen, dann wird direkt die entsprechende Variante aufgerufen.

```
int doSomeOtherWork(A &object)
{
    return(object.A::doWork(2));
}
```

erzeugt die Ausgabe

```
2, 4, 2, 2
```

Typische Implementierung

- Zur Implementierung fügt der Compiler jedem Objekt einen versteckten Pointer hinzu (den "virtual-pointer"oder "v-pointer"). Dieser "v-pointer"zeigt auf eine globale Tabelle (die "virtual-table"oder "v-table").
- Der Compiler erzeugt eine v-table für jede Klasse die mindestens eine virtuelle Funktion enthält. Die v-table selbst hat einen Pointer für jede virtuelle Funktion der Klasse.
- Während des Aufrufs einer virtuellen Funktion, greift das run-time System über den v-pointer des Objekts und dann über den Funktionspointer in der v-table auf den Code der Methode zu.
- Der Overhead bezüglich des Speicherverbrauchs ist damit ein Pointer pro Objekt das virtuelle Methoden enthält, plus ein Pointer pro virtueller Methode. Der Overhead bei der Ausführung sind zwei zusätzliche Speicherzugriffe (für den v-pointer und die Methodenadresse).
- Inlining ist mit virtuellen Methoden nicht möglich.

10.2 Schnittstellenbasisklassen

- Der Sinn einer abstrakten Basisklasse ist es, eine gemeinsame Schnittstelle für die abgeleiteten Klassen bereitzustellen.
- Schnittstellenbasisklassen haben normalerweise keine Attribute (sie enthalten also keine Daten).
- Die Funktionen der Schnittstellenbasisklasse sind meist rein virtuell, d.h. die Funktionalität ist nur in den abgeleiteten Klassen implementiert. Dies wird durch Hinzufügen von `= 0` nach der Funktionsdeklaration gekennzeichnet.
- Klassen die rein virtuelle Funktionen enthalten heißen auch abstrakte Basisklassen.
- Von einer abstrakten Basisklasse lassen sich keine Objekte anlegen, es kann aber Referenzen und Pointer dieses Typs geben (die dann auf Objekte einer abgeleiteten Klasse zeigen).
- Von einer Klasse lassen sich also erst dann Objekte anlegen, wenn *alle* rein virtuellen Funktionen der Basisklasse implementiert wurden. So kann sichergestellt werden, dass eine Klasse auch die komplette Schnittstelle erfüllt.

```
class BaseClass
{
    public:
        virtual int functionA(double x) = 0;
        virtual void functionB(int y) = 0;
        virtual ~BaseClass()
        {};
}
```

10.3 Funktoren

Definition: Ein *Funktionsobjekt* (Funktor) ist jedes Objekt, das wie eine Funktion aufgerufen werden kann[1]

Beispiel

- In C++ hat eine Funktion die Form `return_type foo(Type1 arg1, Type2 arg2);`
- Ein Objekt das den runde Klammer Operator `operator()` definiert kann wie eine Funktion benutzt werden, z.B.

```
class Foo
{
  public:
    return_type operator()(Type1 arg1, Type2 arg2);
};
```

Vorteile von Funktoren

- Funktoren sind "intelligente Funktionen". Sie können
  - neben dem `operator()` weitere Funktionen bereitstellen.
  - einen inneren Zustand haben.
  - vorinitialisiert sein.
- Jeder Funktor hat seinen eigenen Typ.
  - Die Funktionen (oder Funktionspointer auf) `bool less(int,int)` und `bool greater(int,int)` würden den gleichen Typ haben.
  - Die Funktoren `class less` und `class greater` haben verschiedenen Typ.
- Wenn Funktoren anstelle von Funktionspointern an Funktionen übergeben werden, ist das in der Regel schneller.

10.4 Beispiel: Numerische Integration

Als Beispiel dieses Konzepts wollen wir eine Klasse zur numerischen Integration beliebiger Funktionen mit der zusammengesetzten Mittelpunktsregel implementieren.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f((i+\frac{1}{2})\cdot h)\cdot h$$

mit $h = \frac{b-a}{n}$.

Das Beispielprogramm integriert $\cos(x-1)$ mit der zusammengesetzten Mittelpunktsregel. Die dafür verwendeten Dateien sind:

- `funktor.h`: enthält die Schnittstellenbasisklasse für einen Funktor.

[1] D. Vandevoorde, N. M. Josuttis: C++ Templates - The Complete Guide, p. 417

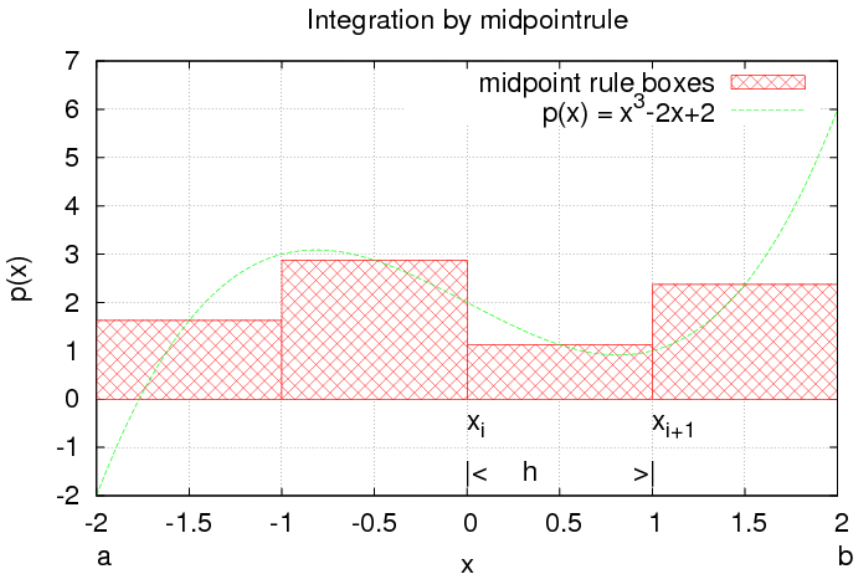


Abbildung 1: Anwendung der Mittelpunktsregel mit $n = 4$ für $p(x) = x^3 - 2x + 2$.

- `cosinus.h`: enthält die Definition eines speziellen Funktors: $\cos(ax + b)$
- `mittelpunkt.h`: enthält die Definition einer Funktion, die einen Funktor als Argument erhält und mit der zusammengesetzten Mittelpunktsregel integriert.
- `integration.cc`: enthält das Hauptprogramm, das den Integrator verwendet um $\cos(x-1)$ über den Bereich $[1 : \frac{\pi}{2} + 1]$ zu integrieren.

funktor.h

```
#ifndef FUNKTORCLASS_H
#define FUNKTORCLASS_H

// Base class for arbitrary functions with one double parameter

class Funktor
{
  public:
    virtual double operator()(double x) = 0;
};

#endif
```

cosinus.h

```
#ifndef COSINUSCLASS_H
#define COSINUSCLASS_H

#include <cmath>
#include "funktor.h"
typedef double blubber;
```

```
// realization of a function cos(a*x+b)
class Cosinus : public Funktor
{
  public:
    Cosinus(double a=1.0, double b=0.0) : a_(a), b_(b)
    {}
    double operator()(blubber x) override {
        return cos(a_*x+b_);
    }
  private:
    double a_,b_;
};

#endif
```

mittelpunkt.h

```
#include "funktor.h"

double MittelpunktsRegel(Funktor &f,double a=0.0, double b=1.0, size_t n=1000)
{
    double h = (b-a)/n; // lenght of a single interval

    // compute the integral boxes and sum them
    double result = 0.0;
    for (size_t i=0; i<n; ++i)
    {
        // evaluate function at midpoint and sum integral value
        result += f(a + (i+0.5)*h);
    }

    return h*result;
}
```

integration.cc

```
// include system headers
#include <iostream>
// own headers
#include "mittelpunkt.h"
#include "cosinus.h"

int main()
{
  // instanciate an object of class MidpointRule
  Cosinus cosinus(1.0,-1.0);
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣"
            << MittelpunktsRegel(cosinus,1.0,M_PI_2+1.0) << std::endl;

  return 0;
}
```

integrator.h

In einer alternativen Implementierung lässt sich auch der Integrator verallgemeinern:

```
#ifndef INTEGRATORCLASS_H
#define INTEGRATORCLASS_H

#include "funktor.h"

class Integrator
{
  public:
    virtual double operator()(Funktor &f) = 0;
};

#endif
```

mittelpunkt_class.h

```
#include "integrator.h"

class MittelpunktsRegel : public Integrator
{
    double a_,b_;
    size_t n_;
  public:
    MittelpunktsRegel(double a, double b, size_t n) : a_(a), b_(b), n_(n)
    {}
    double operator()(Funktor &f) override
    {
        double h = (b_-a_)/n_; // lenght of a single interval

        // compute the integral boxes and sum them
        double result = 0.0;
        for (size_t i=0; i<n_; ++i)
        {
            // evaluate poynomial at midpoint and sum integral value
            result += f(a_ + (i+0.5)*h);
        }

        return h*result;
    }
};
```

simpson_class.h

```
#include "integrator.h"

double SimpsonRegel(Funktor &f,double a=0.0, double b=1.0, size_t n=1000)
{
    double h = (b-a)/n; // lenght of a single interval

    double result = f(a)+f(b);
    for (size_t i=1; i<n; i+=2)
        result += 4. * f(a + i*h);
    for (size_t i=2; i<n; i+=2)
        result += 2. * f(a + i*h);

    return (h*result)/3.;
}
```

integration_class.cc

```
#include <iostream>
#include <memory>
#include "mittelpunkt_class.h"
#include "simpson_class.h"
#include "cosinus.h"

int main()
{
  Cosinus cosinus(1.0,-1.0);
  std::unique_ptr<Integrator> integrate(new
      MittelpunktsRegel(1.0,M_PI_2+1.0,10));
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣"
            << (*integrate)(cosinus) << std::endl;
  SimpsonRegel simpson(1.0,M_PI_2+1.0,10);
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣"
  << simpson(cosinus) << std::endl;
  return 0;
}
```

Arrays von Objekten

- Es ist oft nötig ein Array von Objekten einer gemeinsamen Schnittstellenbasisklasse anzulegen, z.B. die Parameterfunktionen für verschiedene Materialien, die in einer Simulation verwendet werden.
- Da Referenzen bereits beim Anlegen initialisiert werden müssen, kann hier nur ein Array von Basisklassenpointern verwendet werden, die dann auf die verschiedenen Objekte gesetzt werden.
- Die Pointer sollten mit 0 oder in C++11 mit `nullptr` initialisiert werden oder es sollten gleich `std::unique_ptr` oder `std::shared_ptr` verwendet werden.

```
std::vector<std::unique_ptr<Funktor> > Funktion(4);
Funktion[0] = std::unique_ptr<Funktor>(new Cosinus(1.0,-1.0));
// alternativ
Funktion[0].reset(new Cosinus(1.0,-1.0));
...
```

Virtuelle Destruktoren

- Wenn auf Objekte der abgeleiteten Klasse nur noch Basisklassenzeiger verfügbar sind, kann auch nur noch der Destruktor der Basisklasse aufgerufen werden.
- Da abgeleitete Klassen allozierte Ressourcen verwenden könnten, die im Destruktor freigegeben werden müssen, ist es sinnvoll der Basisklasse einen (meist leeren) virtuellen Destruktor zu geben.
- Damit wird dann für jedes Objekt der abgeleiteten Klasse auch über den Basisklassenzeiger der richtige Destruktor aufgerufen.
- Der Destruktor kann nicht rein virtuell sein.

```
class Funktor
{
  public:
    virtual double operator()(double x) = 0;
    virtual ~Funktor()
    {};
};
```

- In C++11 geht das wieder mit dem Schlüsselwort `default`:

  ```
  virtual ~Funktor() = default;
  ```

Dynamic Cast

- In einem Programm kann es wünschenswert sein herauszufinden, ob sich ein Zeiger auf ein Objekt einer Klasse in einen Zeiger auf eine andere Klasse konvertieren lässt (z.B. weil einer der beiden Zeiger, ein Zeiger auf eine Basisklasse des Objekts ist).
- Dies lässt sich mit einem `dynamic_cast` bewerkstelligen. `funk = dynamic_cast<Funktor*>(&f)` konvertiert wenn möglich den Pointer f in einen Pointer auf einen Funktor.
- Das funktioniert auch anders herum in der Ableitungshierarchie:

  ```
  Cosinus *cosin = dynamic_cast<Cosinus*>(funk);
  ```

- Ein `dynamic_cast` liefert entweder einen konvertierten Pointer zurück oder einen Nullpointer, wenn die Konvertierung nicht möglich ist.
- `dynamic_cast` funktioniert auch mit Referenzen:

  ```
  Cosinus &cosin = dynamic_cast<Cosinus&>(f);
  ```

  Wenn die Konvertierung hier nicht durchgeführt werden kann, wird eine Ausnahme vom Typ `std::bad_cast` geworfen.
- In C++11 übernimmt das für den `std::shared_ptr` die freie Funktion `std::dynamic_pointer_cast` .

```
#include<cstdlib>
#include "mittelpunkt.h"
#include "simpson.h"
#include "cosinus.h"

double Integrate(Funktor &f, double a, double b)
{
    Cosinus *cosin = dynamic_cast<Cosinus *>(&f);
    if (cosin==0)
        return MittelpunktsRegel(f,a,b);
    else
        return SimpsonRegel(f,a,b);
}
```

Virtuelle Konstruktoren

Wenn man nur einen Basisklassenpointer auf ein Objekt hat, dann ist es normalerweise nicht möglich ein Objekt des gleichen Typs (der abgeleiteten Klasse) zu erzeugen oder das komplette Objekt zu kopieren (sondern nur den Basisklassenteil). Mit sogenannten "virtuellen Konstruktoren"gelingt das trotzdem:

```
class Funktor
{
  public:
    ...
    virtual Funktor *create() = 0;
    virtual Funktor *clone() = 0;
}

class Cosinus : public Funktor
{
  public:
    ...
    Cosinus *create()
    {
        return new Cosinus();
    }
    Cosinus *clone()
    {
        return new Cosinus(*this);
    }
}
```

Virtuelle Basisklassen

- Manchmal ist es sinnvoll eine Klasse von mehreren anderen Klassen abzuleiten, die von der selben Basisklasse abgeleitet sind.
- Eine mögliche Anwendung ist es, die abgeleiteten Klassen verschiedene Teilaspekte eines Problems abarbeiten zu lassen, die jeweils auf verschiedene Art implementiert werden können, dabei jedoch auf die gleichen Daten der Basisklasse zugreifen müssen.
- Eine von diesen Klassen per Mehrfachvererbung abgeleitete Klasse fasst dann die Funktionalität zu einem bestimmten Gesamtprozess zusammen.

Beispiel: Newton-Verfahren aus DUNE-PDELab

- Beispiel: das Newtonverfahren soll verwendet werden um ein nicht-lineares Gleichungssystem zu lösen. Ein Newtonverfahren besteht aus
  - einem Grundalgorithmus
  - Schritten die zu Begin jeder Newtoniteration ausgeführt werden müssen (z.B. das Neuaufstellen der Jacobimatrix)
  - einem Test, ob das Verfahren konvergiert ist.
  - gegebenenfalls einem Line-Search um das Konvergenzgebiet zu vergrößern.

  Jeder dieser Zwischenschritte wird in eine eigene Klasse ausgelagert, damit man alle Komponenten unabhängig voneinander auswechseln kann. Die gemeinsamen Daten und die virtuellen Funktionen kommen in eine Basisklasse.

- Im Normalfall hat jede Klasse ihre eigene Basisklasse, es gäbe die Daten dann mehrfach. Um dies zu verhindern gibt es virtuelle Ableitungen.

```
class NewtonSolver : public virtual NewtonBase
{
...
};

class NewtonTerminate : public virtual NewtonBase
{
...
};

class NewtonLineSearch : public virtual NewtonBase
{
...
};

class NewtonPrepareStep : public virtual NewtonBase
{
...
};

class Newton : public NewtonSolver, public NewtonTerminate,
               public NewtonLineSearch, public NewtonPrepareStep
{
...
};
```

10.5 Zusammenfassung Dynamischer Polymorphismus

Wenn es verschiedene Objekte gibt, die ein grundlegendes Prinzip verkörpern (so wie Kreis, Dreieck, Rechteck ... spezielle Realisierungen eines geometrischen Objektes sind) oder eine bestimmte Funktionalität (so wie die Trapezregel und die Simpsonregel Integrationsverfahren sind), dann ist es guter Stil in C++ eine gemeinsame Schnittstelle zu definieren, die jede spezifische Realisierung auf ihre eigene Weise implementiert.

Dynamischer Polymorphismus

- verwendet dazu abstrakte Basisklassen und virtuelle Funktionen.
- es gibt spezielle Sprachkonstrukte die sicherstellen, dass jede Klasse alle Schnittstellenfunktionen auch wirklich implementiert (rein virtuelle Funktionen)
- erlaubt die Auswahl der zu verwendenden Variante zur Laufzeit.
- erfordert Mehraufwand (virtual function table)
- verhindert einige Optimierungen (inlining, loop-unrolling)

11 Statischer Polymorphismus

11.1 Generische Programmierung

- Die gleichen Algorithmen werden oft für unterschiedliche Datentypen benötigt.

- Ohne generische Programmierung muss die gleiche Funktion für jeden Datentyp neu geschrieben werden. Dies ist mühsam und fehlerträchtig. Beispiel:

```
int Square(int x)
{
    return(x*x);
}
long Square(long x)
{
    return(x*x);
}
```

```
float Square(float x);
{
    return(x*x);
}
double Square(double x);
{
    return(x*x);
}
```

- Generische Programmierung ermöglicht es, einen Algorithmus einmal zu schreiben und ihn mit dem Datentyp zu parametrisieren.
- Das Sprachmittel heißt in C++ Templates und kann sowohl für Funktionen als auch für Klassen verwendet werden.

11.2 Funktionstemplates

- Ein Funktionstemplate beginnt mit dem Schlüsselwort `template` und einer Liste mit einem oder mehreren durch Kommas getrennten Templateargumenten in spitzen Klammern:

```
template<typename T>
T Square(T x)
{
    return(x*x);
}
```

- `typename` bezeichnet einen Typen und wurde eingeführt, weil es in C++ ja auch built-in types gibt, die keine Klassen sind (z.B. `int`). Aus historischen Gründen sind `class` und `typename` in der Templateargumentliste äquivalent.

Template Instantiierung

- Der Compiler generiert beim ersten Verwenden der Funktion mit einem bestimmten Datentypen automatisch den Code für diesen Typen. Dies bezeichnet man als Template Instantiierung.
- Eine explizite Instantiierung ist nicht nötig.
- Die Templateparameter werden aus den Typen der Funktionsargumente bestimmt.
- Dabei ist keinerlei automatische Typumwandlung erlaubt (im Gegensatz zu normalen Funktionsaufrufen).
- Genau wie beim Überladen von Funktionen spielt der Typ des Rückgabewertes keine Rolle.
- Mehrdeutigkeiten können aufgehoben werden durch:
  - Explizite Typumwandlung der Argumente

– Explizite Angabe der Templateargumente in spitzen Klammern:

```
std::cout << Square<int>(4) << std::endl;
```

- Die Argumenttypen müssen zur Deklaration passen, die Klassen müssen alle benötigten Operationen bereit stellen (z.B. den `operator<`)

Beispiel Unäres Funktionstemplate

```
#include<cmath>
#include<iostream>

template<typename T>
T Square(T x)
{
    return(x*x);
}

int main()
{
    std::cout << Square<int>(4) << std::endl;
    std::cout << Square<double>(M_PI) << std::endl;
    std::cout << Square(3.14) << std::endl;
}
```

Beispiel Binäres Funktionstemplate

```
#include<cmath>
#include<iostream>

template<class U>
const U &max(const U &a, const U &b) {
    if (a>b)
        return(a);
    else
        return(b);
}

int main()
{
    std::cout << max(1,4) << std::endl;
    std::cout << max(3.14,7.) << std::endl;
    std::cout << max(6.1,4) << std::endl; // Compilerfehler
    std::cout << max<double>(6.1,4) << std::endl; // eindeutig
    std::cout << max(6.1,double(4)) << std::endl; // eindeutig
    std::cout << max<int>(6.1,4) << std::endl; // Compilerwarnung
}
```

Übersetzung von Templates

- Wenn Templates nicht verwendet und deshalb nicht instantiiert werden, wird der Templatecode nur auf grobe Syntaxfehler geprüft (z.B. fehlende Strichpunkte).
- Erst wenn ein Template instantiiert wird erfolgt die Überprüfung, ob auch alle Funktionsaufrufe gültig sind. Erst dann werden z.B. nicht unterstützte Funktionsaufrufe entdeckt. Die Fehlermeldungen können dabei recht merkwürdig ausfallen.

- Da der Code erst bei der Verwendung generiert wird, muss der Compiler zu diesem Zeitpunkt die vollständige Funktionsdefinition sehen, nicht nur ihre Deklaration wie bei normalen Funktionen.
- Damit ist die übliche Aufteilung in Header- und Sourcedatei für Templates nicht möglich. Die gesamte Definition muss in der Headerdatei stehen.

Überladen von Funktionen

- Funktionstemplates können genauso wie normale Funktionen überladen werden.
- Es darf auch Nichttemplate- und Templatefunktionen mit dem gleichen Namen geben.
- Wenn es eine passende (ohne Typkonvertierung) Nichttemplatefunktion gibt, wird immer diese verwendet.
- Wenn eine Templatefunktion erzeugt werden kann, die besser passt (ohne Typkonvertierung) wird diese genommen.
- Die Verwendung einer Templatefunktion kann durch Hinzufügen von leeren spitzen Klammern erzwungen werden.

Beispiel: Bestimmung des Maximums

```
inline const int &max(const int &a, const int &b){
  return a < b ? b : a;
}

template<typename T>
inline const T &max(const T &a, const T &b){
  return a < b ? b : a;
}

template<typename T>
inline const T &max(const T &a, const T &b, const T &c){
  return ::max(a, ::max(b, c));
}

int main(int argc, char** argv) {
  ::max(7, 42, 68);    // calls the template for three arguments, which calls
      twice the nontemplate for two ints
  ::max(7.0, 42.0);    // calls max<double> (argument deduction)
  ::max('a', 'b');     // calls max<char> (argument deduction)
  ::max(7,42);         // calls nontemplate for two ints
  ::max<>(7,42);       // calls max<int> (argument deduction)
  ::max<double>(7,42);// calls max<double> (no argument deduction)
  ::max('a', 42.7);    // calls nontemplate for two ints
}
```

Überladen von Funktionen kann hier z.B. verwendet werden, wenn es Typen gibt, bei denen ein Vergleich vom Standard abweicht:

```
#ifndef MAX_POINTER_REFERENCE_HH
#define MAX_POINTER_REFERENCE_HH
#include<cstring>
```

```
#include"max.hh"

// Vergleich von Pointern nach Inhalt
template<typename T>
inline const T* &max(const T* &a, const T* &b)
{
  return *a < *b ? b : a;
}

// Vergleich von C-Strings
inline const char* &max(const char* &a, const char* &b)
{
  return std::strcmp(a,b) < 0 ? b : a;
}

#endif

#include<string>
#include<iostream>
#include"max_pointer_reference.hh"

int main(int argc, char** argv)
{
  int a=47, b=9;
  std::cout << ::max(a,b) << "\n"; // max for two ints

  std::string s="hey", t="Leute";
  std::cout << ::max(s,t) << "\n"; // template max for two strings

  int *p1=&a, *p2=&b;
  std::cout << *::max(p1,p2) << "\n"; // template max for two pointers

  const char *s1="Anna", *s2="Marlene";
  std::cout << ::max(s1,s2) << "\n"; // max for two C strings
}
```

Spezialisierung von Funktionstemplates

Für bestimmte Parameterwerte lassen sich spezielle Templatefunktionen definieren. Dies wird Templatespezialisierung genannt. Es kann z.B. für Geschwindigkeitsoptimierungen verwendet werden:

```
template <size_t N>
double scalarProduct(const double *a, const double *b)
{
    double result = 0;
    for (size_t i=0;i<N;++i)
        result += a[i]*b[i];
    return result;
};

template<>
double scalarProduct<2>(const double *a, const double *b)
{
    return a[0]*b[0]+a[1]*b[1];
};
```

Nützliche Funktionstemplates

Die C++ Standardbibliothek stellt bereits einige nützliche Funktionstemplates zur Verfügung:

- `const T &std::min(const T &, const T &)` Minimum von a und b `int c = std::min(a,b);`
- `const T &std::max(const T &, const T &)` Maximum von a und b (in dem Beispiel oben stand immer `::max` um die Verwendung dieses Templates zu verhindern) `int c = std::max(a,b);`
- `void std::swap(T &, T &)` Vertauscht den Inhalt von a und b `std::swap(a,b);`

11.3 Klassentemplates

- Es ist oft hilfreich auch Klassen parametrisieren zu können.
- Klassentemplates werden genau wie Funktionstemplates definiert, z.B.

```
template<typename T1, typename T2>
class Blub
{
    T1 var1;
    T2 var2;
  public:
    T2 Multiply(T1 a, T2 b);
};
```

- Handelt es sich bei den Templateargumenten um Typen (z.B. `typename T1`) dann können diese innerhalb der Klasse verwendet werden um Attribute, Funktionsargumente und Rückgabewerte zu definieren.

Insbesondere Containerklassen können zum Speichern von Elementen ganz unterschiedlichen Typs verwendet werden. Hier als Beispiel ein Stack:

```
template<typename T>
class Stack
{
private:
  std::vector<T> elems;

public:
  void push(const T &);
  void pop();
  T top() const;
  bool empty() const
  {
    return elems.empty();
  }
};
```

- Es ist wichtig zwischen dem Typ der Klasse und ihrem Namen zu unterscheiden:
  - Der Typ der Klasse ist `Stack<T>`. Dieser muss verwendet werden, wenn Objekte dieser Klasse als Funktionsargumente oder Rückgabewert verwendet werden sollen (z.B. im Copy-Konstruktor).
  - Der Name der Klasse und damit der Name der Konstruktoren und des Destruktors ist `Stack`.

Implementierung von Methoden außerhalb der Klasse

- Die Methoden eines Klassentemplates können ganz normal als inline Funktionen definiert werden.
- Wird eine Methode außerhalb der Klasse definiert, dann muss dem Compiler mitgeteilt werden, dass die Methode zu einem Klassentemplate gehört.
- Dazu steht vor der Methodendefinition das Schlüsselwort `template` gefolgt von der Templateargumentliste der Klasse.
- Die Templateargumente werden nach dem Klassennamen in spitzen Klammern aufgeführt (der Namespace der Klasse besteht aus ihrem Namen und den Templateargumenten).

```
template<typename T>
void Stack<T>::push(const T &elem){
  elems.push_back(elem);
}

template<typename T>
void Stack<T>::pop(){
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  elems.pop_back();
}

template<typename T>
T Stack<T>::top() const{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  return elems.back();
}
```

Verwendung von Klassentemplates

- Um ein Objekt eines Klassentemplates zu definieren, muss dem Namen der Klasse eine Liste von passenden Argumenten in spitzen Klammern folgen (z.B. `Stack<int>`).
- Um Speicher und (Übersetzungs)zeit zu sparen wird nur für Methoden die auch tatsächlich aufgerufen werden Code generiert.
- Klassentemplates können also selbst für Typen instantiiert werden, die nicht alle erforderlichen Operationen bereitstellen, solange die Methoden in denen diese benötigt werden nie aufgerufen werden.
- Instantiierte Klassentemplates können wie normale Typen verwendet werden, also z.B. als `const` oder `volatile` deklariert oder in Feldern verwendet werden. Es können natürlich auch Pointer und Referenzen definiert werden
- Wenn Templates lange Argumentlisten haben, dann wird der Name der instantiierten Klasse sehr lang. Hier sind Typdefinitionen sehr hilfreich:

```
typedef Stack<int> IntStack;
void foo(const IntStack &s){
  IntStack is;
  ...}
```

- Natürlich können instantiierte Templates auch selbst als Templateargumente dienen.

```
Stack<Stack<int> > iss; // beachte das Leerzeichen zwischen
                        // schliessenden Klammern
```

```
#include<iostream>
#include<string>
#include<cstdlib>
#include"stack.hh"

int main(int argc, char** argv){
  try{
    Stack<int> intStack;
    Stack<std::string> stringStack;

    intStack.push(7);
    std::cout << intStack.top() << std::endl;

    stringStack.push("hello");
    std::cout << stringStack.top() << std::endl;
    stringStack.pop();
    stringStack.pop();
    }catch(const std::exception &e){
      std::cerr << "Exception␣" << e.what() << std::endl;
      return 1;
    }
}
```

Nützliches Klassentemplate: Pair

Ein nützliches Klassentemplate aus der C++ Standardbibliothek ist `pair`:

```
std::pair<int,double> a;
a.first=2;
a.second=5.;
std::cout << a.first << "␣" << a.second << std::endl;
```

`pair` erlaubt z.B. Funktionen mit zwei Rückgabewerten.

Spezialisierung von Klassentemplates

- Auch Klassentemplates lassen sich für bestimmte Argumentwerte spezialisieren, entweder weil für diese ein besonderes Verhalten notwendig ist oder zu Optimierungszwecken.
- Dies ähnelt in gewisser Weise dem Überladen von Funktionen.
- Es müssen alle Methoden spezialisiert werden:

```
template<>
class Stack<std::string>
{
private:
  std::vector<std::string> elems;

public:
  void push(const std::string &);
  void pop();
  std::string top() const;
  bool empty() const
  {
    return elems.empty();
  }
};

void Stack<std::string>::push(const std::string &elem){
  elems.push_back(elem);
}

void Stack<std::string>::pop(){
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  elems.pop_back();
}

std::string Stack<std::string>::top() const{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  return elems.back();
}
```

Teilweise Spezialisierung

Bei einer Klasse wie der Folgenden

```
template<typename T1, typename T2>
class MyClass { ... };
```

sind mehrere teilweise Spezialisierungen möglich:

```
// both template paramters are equal
template<typename T>
class MyClass<T,T>{ ... };

// second parameter has a specific type, e.g. an int
template<typename T>
class MyClass<T,int>{ ... };

// partial specialisation for pointers
template<typename T1, typename T2>
class MyClass<T1*,T2*>{ ... };
```

Dies führt zu folgenden Zuordnungen:

```
MyClass<int, float> mif;  // use MyClass<T1,T2>
MyClass<float,float> mff; // use MyClass<T,T>
MyClass<float,int> mfi;   // use MyClass<T,int>
MyClass<int*,float*> mpi; // use MyClass<T1*,T2*>
```

Es kann aber auch zu Mehrdeutigkeiten kommen:

```
MyClass<int,int> mii; // matches MyClass<T,T> and MyClass<T,int>
MyClass<int*,int*> m; // matches MyClass<T,T> and MyClass<T1*,T2*>
```

In diesen Fällen kommt es zu einem (schwer aufzulösenden) Compilerfehler.

Template Defaultargumente

- Auch für die Argumente von Klassentemplates lassen sich Defaultwerte definieren.
- Diese können auch von vorhergehenden Templateargumenten abhängen.
- Wie auch bei Funktionsargumenten, können jeweils nur die letzten Argumente Defaultwerte haben.
- Beispiel: Definiere einen Stack mit einem zusätzlich wählbaren Container:

```
template<typename T, typename C = std::vector<T> >
class Stack
{
public:
  typedef C Container;
private:
  Container elems;

public:
  void push(const T &);
  void pop();
  T top() const;
  bool empty() const
  {
    return elems.empty();
  }
};
```

- Die Angabe eines Defaultargumentes entbindet nicht von der Pflicht bei den Funktionsdefinitionen außerhalb der Klasse alle Templateargumente anzugeben

```
template<typename T, typename C>
void Stack<T,C>::push(const T &elem){
  elems.push_back(elem);
}

template<typename T, typename C>
void Stack<T,C>::pop(){
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop():␣empty␣stack");
  elems.pop_back();
}

template<typename T, typename C>
T Stack<T,C>::top() const{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop():␣empty␣stack");
  return elems.back();
}
```

- Der Stack kann genauso verwendet werden wie zuvor.
- Wird der zweite Templateparameter weggelassen, dann wird wie bisher ein `std::vector` verwendet um die Elemente zu speichern.
- Zusätzlich lässt sich jetzt auch ein anderer Containertyp verwenden, z.B. eine `std::deque`

```
int main(int argc, char** argv){
  try{
    Stack<int> intStack;
    Stack<std::string,std::deque<std::string> > stringStack;

    intStack.push(7);
    std::cout<<intStack.top()<<std::endl;

    stringStack.push("hello");
    std::cout<<stringStack.top()<<std::endl;
    stringStack.pop();
    stringStack.pop();
  }catch(const std::exception& e){
      std::cerr << "Exception␣"<<e.what()<< std::endl;
      return 1;
    }
}
```

11.4 Templateparameter die keine Typen sind

- Templateparameter müssen nicht notwendigerweise Typen sein.
- Es ist auch möglich konstante Werte zu verwenden.
- Diese müssen zur Übersetzungszeit bekannt sein.
- Sie können bei Klassen- und Funktionstemplates verwendet werden.

```
template<class T, int VAL>
T addValue(const T &x){
  return x + VAL;
}
```

- Nicht erlaubt sind Floatingpointzahlen, Nullpointerkonstanten, Stringliterale.
- Stringlaterale können durch Definition einer Variablen mit externer Bindung verwendet werden:

```
template<char const *name>
class MyClass { ... }

extern const char[] = "hello";

MyClass<s> x;
```

Erlaubte Templateparameter

```
template <typename T, T nontype_param> class C;

C<int,33> *c1;          // integer
int a;
C<int *,&a> c2;         // Adresse einer Variablen
void f();
void f(int);
C<void (*)(int),f> *c3; // Funktionspointer auf f(int)

class X {
  public:
    int n;
    static bool b;
};
C<bool &,X::b> *c4;     // statische Klassenmitglieder
C<int X::*,&X::n> *c5;  // Pointer auf Mitglieder
template<typename T>
void templ_func();
C<void (),&templ_func<double> > *c6; // Auch Funktions-
                        // templates sind Funktionen
```

Verbotene Templateparameter

```
template <typename T, T nontype_param> class C;

class Base {
  public:
    int i;
    static bool b;
} base;

class Derived : public Base {
} derived_obj;

C<Base *,&derived_obj> *err1; // Keine automatische Konvertierung
                              // zu Basisklasse
C<int &,base.i> *err2;        // Attribute von Objekten sind keine
                              // Variablen
int a[10];
C<int *,&a[0]> *err3;         // Adressen von einzelnen
                              // Feldelementen sind auch
                              // nicht erlaubt
```

11.5 Vererbung bei Klassentemplates

```
template<typename T>
class MyNumericalSolver : public NumericalSolver<T,3>
{
    T variable;
  public:
    MyNumericalSolver(T val) : NumericalSolver<T,3>(),
                               variable(val)
    {};
}
```

- Wenn eine Klasse von einem Klassentemplate abgeleitet wird, dann müssen die Templateargumente als Teil des Basisklassennamens angegeben werden.
- Dies gilt auch für den Aufruf der Basisklassenkonstruktoren.

11.6 Statischer Polymorphismus

Beispiel: Numerische Integration von $\cos(x-1)$

Das Beispiel realisiert die Integration von $\cos(x-1)$ mit der Mittelpunktsregel mit Hilfe von Templates anstelle von virtuellen Funktionen. Die Dateien sind

- `cosinustemp.h`: enthält die Definition und Implementierung des Funktors für $\cos(ax+b)$
- `mittelpunkttemp.h`: enthält die Definition eines Funktionstemplates, das einen Funktor als Templateargument erhält und diesen mit der Mittelpunktsregel integriert.
- `integrationtemp.cc`: enthält das Hauptprogramm das das Template für die Mittelpunktsregel verwendet um $\cos(x-1)$ über den Bereich $[1 : \frac{\pi}{2}+1]$ zu integrieren.

cosinetemp.h

```
#ifndef COSINECLASS_H
#define COSINECLASS_H

#include <cmath>

// realization of a function cos(a*x+b)
class Cosinus
{
  public:
    Cosinus(double a=1.0, double b=0.0) : a_(a), b_(b)
    {}
    double operator()(double x) const
    {
        return cos(a_*x+b_);
    }
  private:
    double a_,b_;
};

#endif
```

midpointtemp.h

```
template<typename T>
double MittelpunktsRegel(const T &f,double a=0.0, double b=1.0, size_t n=1000)
{
    double h = (b-a)/n; // lenght of a single interval

    // compute the integral boxes and sum them
    double result = 0.0;
    for (int i=0; i<n; ++i)
    {
        // evaluate poynomial at midpoint and sum integral value
        result += h * f(a + (i+0.5)*h);
```

```
    }

    return result;
}
```

integrationtemp.cc

```
// include system headers
#include <iostream>
// own headers
#include "mittelpunkttemp.h"
#include "cosinustemp.h"

int main()
{
  // instanciate an object of class MidpointRule
  Cosinus cosinus(1.0,-1.0);
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣"
            << MittelpunktsRegel(cosinus,1.0,M_PI_2+1.0) << std::endl;

  return 0;
}
```

Zusammenfassung Statischer Polymorphismus

Statischer Polymorphismus:

- verwendet Templates und das Überladen von Funktionen.
- wird (bisher) nicht mit eigenen C++-Sprachmitteln unterstützt.
- erlaubt die Auswahl der zu verwendenden Version zur Übersetzungszeit.
- erzeugt keinen Overhead
- erlaubt alle Optimierungen
- führt zu längeren Übersetzungszeiten

⇒ Statischer Polymorphismus ist besonders geeignet, wenn viele kurze Funktionsaufrufe benötigt werden (wie z.B. der Zugriff auf Matrixelemente . . .)

11.7 Dynamischer versus Statischer Polymorphismus

- Polymorphismus mit Vererbung ist begrenzt und dynamisch:
  - Begrenzt heißt, dass die Schnittstelle aller Realisierungen durch die Definition der gemeinsamen Basisklasse festgelegt ist.
  - Dynamisch heißt, dass die Festlegung welche Klasse die Schnittstelle realisiert erst zur Zeit der Ausführung erfolgt.
- Polymorphismus der mit Templates realisiert wird ist unbegrenzt und statisch:
  - Unbegrenzt heißt, dass die Schnittstellen aller am Polymorphismus teilnehmenden Klassen nicht festgelegt sind.

  - Statisch heißt, dass die Festlegung welche Klasse die Schnittstelle realisiert bereits zur Übersetzungszeit erfolgt.

- Dynamischer Polymorphismus:
  - Erlaubt eine elegante Verwaltung heterogener Mengen.
  - Kleinere Programmgröße.
  - Bibliotheken lassen sich als reiner Binärcode vertreiben. Es ist nicht notwendig den Sourcecode der Implementierung zu veröffentlichen.
- Statischer Polymorphismus:
  - Einfache Implementierung von (homogenen) Containerklassen.
  - Meist schnellere Programmausführung.
  - Klassen die nur Teile der Schnittstelle implementieren können verwendet werden, solange nur dieser Teil zur Ausführung kommt.

11.8 Template Besonderheiten

11.8.1 Schlüsselwort `typename`

```
template<typename T, int dimension = 3>
class NumericalSolver
{
    ...
  private:
    typename T::SubType value_type;
}
```

- Templateklassen definieren häufiger auch Typen (z.B. um den Typ von Rückgabewerten in Abhängigkeit von Templateparametern zurückzuliefern).
- Ein C++ Compiler kann nicht wissen worum es sich bei dem Konstrukt `T::Name` (wobei `T` ein typename Templateargument ist) handelt, da er die Klassendefinition von `T` ja noch nicht kennt. Er geht deshalb defaultmässig davon aus, dass es sich dabei um eine statische Variable handelt.
- Handelt es sich jedoch um einen in der Klasse definierten Typ, dann muss dies mit dem Schlüsselwort `typename` klargestellt werden.
- Dieses wird nur innerhalb eines Funktions- oder Klassentemplates benötigt (sonst ist ja klar was Name genau ist).
- Es wird nicht benötigt in einer Liste von Basisklassenspezifikationen oder einer Initialisierungsliste.

11.8.2 Member Templates

Auch Klassenmitglieder (Methoden oder nested Classes) können selbst Templates sein..

```
template<typename T>
class Stack
{
  private:
    std::deque<T> elems;
  public:
    void push(const T &);
    void pop();
    T top() const;
    bool empty() const
    {
      return elems.empty();
    }

    //assignment of stack of elements of type T2
    template<typename T2>
    Stack<T> &operator=(const Stack<T2> &);
};
```

In diesem Beispiel wird der Standardzuweisungsoperator überladen, nicht ersetzt (siehe die Regeln für das Überladen von Templatefunktionen).

```
template<typename T>
template<typename T2>
Stack<T> &Stack<T>::operator=(const Stack<T2> &other)
{
  if((void*)this==(void*)&other)
    return *this;

  Stack<T2> tmp(other);

  elems.clear();
  while(!tmp.empty())
  {
    elems.push_front(tmp.top());
    tmp.pop();
  }
  return *this;
}
```

- Wir brauchen jetzt zwei `template` Zeilen am Beginn der Methodendefinition.
- Da `Stack<T>` und `Stack<T2>` völlig unterschiedliche Typen sind, kann nur der öffentliche Teil der Schnittstelle verwendet werden. Um an die untersten Elemente des Stacks heranzukommen wird eine Kopie angelegt und dann nach und nach mit `pop` abgebaut.

Verwendung:

```
int main(int argc, char** argv)
{
  Stack<int> intStack;
  Stack<float> floatStack;

  intStack.push(100);
  floatStack.push(0.0);
  floatStack.push(10.0);
  floatStack=intStack; // OK, int konvertiert nach float
  intStack=floatStack; // hier geht information verloren
}
```

11.8.3 Schlüsselwort `.template`

```
class A
{
  public:
    template<class T> T doSomething() { };
};

template<class U> void doSomethingElse(U variable)
{
    char result = variable.template doSomething<char>();
}

template<class U,typename V> V doSomethingMore(U *variable)
{
    return variable->template doSomething<V>();
}
```

- Eine weitere Mehrdeutigkeit betrifft das < Zeichen. Ein C++ Compiler nimmt hier standardmässig an, dass es sich bei < um den Beginn eines Vergleiches handelt.
- Ist das < Teil eines Methodennamens, der explizit von einem Templateparameter abhängt, dann muss vor dem Methodennamen das Schlüsselwort `template` eingefügt werden.
- Wird benötigt mit den operatoren ".", "::" und "->"

11.8.4 Template Template Parameter

- Es kann nötig sein, dass ein Templateparameter selbst wieder ein Klassentemplate ist.
- Bei der Stackklasse mit austauschbarem Container musste der Anwender den im Container verwendeten Typ selbst angeben

  ```
  Stack<int,std::vector<int>> myStack;
  ```

  Dies ist Fehleranfällig, falls die beiden Typen nicht übereinstimmen.
- Mit einem template template Parameter lässt sich das besser schreiben:

  ```
  template<typename T, template<typename> class C=std::deque>
  class Stack{
  private:
    C<T> elems;
    ...
  }
  ```

- Verwendung:

  ```
  Stack<int,std::vector> myStack;
  ```

- Innerhalb der Klasse lassen sich template template Parameter mit jedem Typen instantiieren, nicht nur mit einem Templateparameter der Klasse.
- Das template template Argument muss genau zu dem template template Parameter passen, für den es eingesetzt wird. Dabei werden keine Defaultwerte eingesetzt.

Stack mit Template Template Parameter

```
template<typename T, template<typename U,
                              typename = std::allocator<U> >
                     class C=std::deque>
class Stack
{
  private:
    C<T> elems;
  public:
    void push(const T &);
    void pop();
    T top() const;
    bool empty() const
    {
      return elems.empty();
    }

    //assignment of stack of elements of type T2
    template<typename T2, template<typename, typename> class C2>
    Stack<T,C> &operator=(const Stack<T2,C2> &);
};

template<typename T, template<typename, typename> class C>
void Stack<T,C>::push(const T &elem)
{
  elems.push_back(elem);
}

template<typename T, template<typename, typename> class C>
void Stack<T,C>::pop()
{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop():␣empty␣stack");
  elems.pop_back();
}

template<typename T, template<typename, typename> class C>
T Stack<T,C>::top() const
{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop():␣empty␣stack");
  return elems.back();
}

template<typename T, template<typename, typename> class C>
template<typename T2, template<typename, typename> class C2>
Stack<T,C> &Stack<T,C>::operator=(const Stack<T2,C2> &other)
{
  if((void*)this==(void*)&other)
    return *this;

  Stack<T2,C2> tmp(other);
  elems.clear();
  while(!tmp.empty()){
    elems.push_front(tmp.top());
    tmp.pop();
  }
```

```
  return *this;
}
```

Verwendung:

```
int main(int argc, char** argv)
{
  Stack<int> intStack;
  Stack<float,std::deque> floatStack;

  intStack.push(100);
  floatStack.push(0.0);
  floatStack.push(10.0);
  floatStack=intStack; // OK, int konvertiert nach float
  intStack=floatStack; // Achtung, hier geht information verloren
}
```

11.8.5 Initialisierung mit Null

- In C++ werden die Variablen der built-in Typen (wie `int`, `double`, oder Pointer) aus Performancegründen nicht mit einem Defaultwert initialisiert.
- Jede uninitialisierte Variable hat einen undefinierten Inhalt (das was halt gerade an der Stelle im Speicher stand):

```
template<typename T>
void foo()
{
  T x; // x hat undefinierten Wert wenn T ein built-in type ist
}
```

- Es ist jedoch möglich für built-in Typen einen Defaultkonstruktor explizit aufzurufen, der die Variable auf Null setzt (oder auf `false` beim Typ `bool`)

```
template<typename T>
void foo()
{
  T x(); // x ist Null (oder false) wenn T ein built-in type ist
}
```

- Soll sichergestellt werden, dass alle Variablen in einem Klassentemplate initialisiert werden, so muss der argumentlose Konstruktor für alle Attribute explizit in einer Initialisierungsliste aufgerufen werden.

```
template<typename T>
class MyClass
{
  private:
    T x;
  public:
    MyClass() : x() //initialisiert x
    {
    }
    ...
};
```

C++11: Template Aliases

```
template <typename T, int U>
class GeneralType
{};

template <int U>    // fuer teilweise definierte Templates
using IntName = GeneralType<int,U>;

int main()
{
  using int32 = int;                    // fuer normale Typen
  using Function = void (*)(double);  // fuer Funktionen
  using SpecialType = GeneralType<int,36>;  // fuer vollstaendig definierte
      Templates
  IntName<7> blub;
}
```

- In C++11 gibt es eine alternative Methode zu Typedefs können um Abkürzungen für lange Typnamen zu definieren.
- Diese Alternative nennt sich “template aliasing”.
- Sie erlaubt auch das teilweise festlegen von Templateargumenten.

11.8.6 Abhängige und Unabhängige Basisklassen

Unabhängige Basisklassen

- Eine unabhängige Basisklasse ist auch ohne Kenntnis eines Templateparameters vollständig festgelegt.
- Unabhängige Basisklassen verhalten sich im wesentlichen wie Basisklassen in normalen (Nicht-Template) Klassen.
- Wenn ein Name in der Klasse auftaucht vor dem kein Namespace steht (ein unqualifizierter Typ) dann sucht der Compiler in der folgenden Reihenfolge nach einer Definition:
  1. Definitionen in der Klasse
  2. Definitionen in unabhängigen Basisklassen
  3. Templateargumente

```
template<typename X>
class Base
{
  public:
    int basefield;
    typedef int T;
};

class D1 : public Base<Base<void> >
{
  public:
    void f()
```

```
    {
      basefield = 3; // Zugriff auf geerbte Nummer
    }
};

template<class T>
class D2 : Base<double>
{ // unabhaengige Basisklasse
  public:
    void f()
    {
      basefield = 7; // Zugriff auf geerbte Nummer
    }
    T strange; // T hat den Typ Base<double>::T !!
};


int main(int argc, char** argv)
{
  D1 d1;
  d1.f();
  D2<double> d2;
  d2.f();
  d2.strange=1;
  d2.strange=1.1; // Vorsicht: d2.strange hat Typ int!
  std::cout << d2.strange << std::endl;
}
```

Abhängige Basisklassen

- Im letzten Beispiel war die Basisklasse vollständig festgelegt.
- Das gilt nicht für Basisklassen, die von einem Templateparameter abhängen.
- Der C++ Standard legt fest, dass unabhängige Namen, die in einem Template vorkommen, beim ersten Vorkommen aufgelöst werden.

```
template<typename T>
class DD : public Base<T>
{
  public:
    void f()
    {
      basefield = 0  // (1) wuerde zu Typaufloesung und Bindung an int fuehren
    }
};

template<>
class Base<bool>
{
  public:
    enum { basefield = 42 };  // (2) Templatespezialisierung will Variable anders definieren
};

void g(DD<bool>& d)
{
    d.f()  // (3) Konflikt
}
```

1. Der erste Zugriff auf `basefield` in `f()` bei der Klassedefinition von `DD` würde zur Bindung von `T` an `int` in der Funktion `f()` führen (wegen Definition in Klassentemplate).

2. anschliessend würde aber für den Typ `bool` der Typ von `basefield` in etwas unveränderbares geändert.
3. Bei der Instantiierung (3) käme es dann zu einem Konflikt.

- Damit dieses Problem nicht entsteht, legt C++ fest, dass unabhängige Namen in abhängigen Basisklassen nicht gesucht werden. Der C++ Compiler gibt deshalb schon bei (1) eine Fehlermeldung aus. (`error: 'basefield'was not declared in this scope`).
- Den Basisklassenattributen und -methoden muss deshalb entweder "`this->`" oder "`Base<T>::`" vorangestellt werden.
- Dies führt dazu dass der Name abhängig und damit erst bei der Instantiierung aufgelöst wird.
- Beispiele

```
template<typename T>
class DD : public Base<T>
{
  public:
    void f()
    {
      this->basefield = 0
    }
};
```

oder

```
template<typename T>
class DD : public Base<T>
{
  public:
    void f()
    {
      Base<T>::basefield = 0
    }
};
```

oder kurz:

```
template<typename T>
class DD : public Base<T>
{
    using Base<T>::basefield; // (1) ist jetzt abhaengig
                              // fuer ganze Klasse
  public:
    void f(){ basefield = 0 } // findet (1)
};
```

12 Unified Modeling Language (UML)

- Klassenhierarchien können ziemlich komplex werden.
- Es ist hilfreich, sie graphisch darstellen zu können.

- Als Standard hat sich hier die Unified Modeling Language (UML) durchgesetzt. Sie dient zur Visualisierung, Spezifikation, Konstruktion und Dokumentation von objektorientierter Software.
- Ist aus mehreren Vorgängern entstanden (z.B. Booch, OOSE und OMT).
- Version 1.0 erschien im September 1997.
- Es gibt auch Plugins für Entwicklungsumgebungen (z.B. Eclipse) mit denen sich aus UML-Diagrammen automatisch Code generieren lässt.

UML Diagrammarten

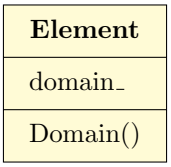
UML stellt 9 verschiedene Arten von Programmen bereit:

1. Class diagram
2. Object
3. Use case
4. Sequence
5. Collaboration
6. Statechart
7. Activity
8. Component
9. Deployment

Wir behandeln nur einen winzigen Ausschnitt aus diesem umfangreichen Werkzeugset.

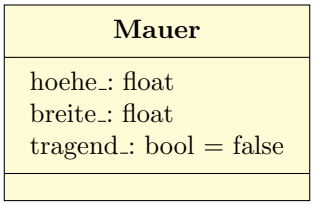
Klassen

- Klassen werden durch einen rechteckigen Kasten dargestellt, der durch horizontale Linien in verschiedene Abschnitte unterteilt ist.
- Zuerst kommt der Name der Klasse, dann ihre Attribute und dann ihre Methoden.



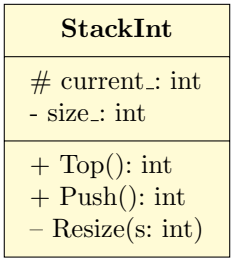
Attribute

- Attribute können einen Typ und gegebenenfalls einen default-Wert haben.



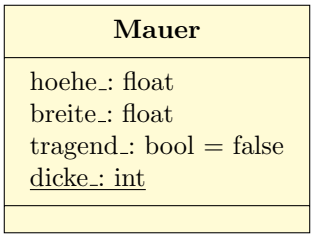
Zugriffskontrolle

- Die Zugriffsrechte auf Attribute werden durch die vorangestellten Zeichen `+` für `public`, `#` für `protected` und `-` für `private` gekennzeichnet.



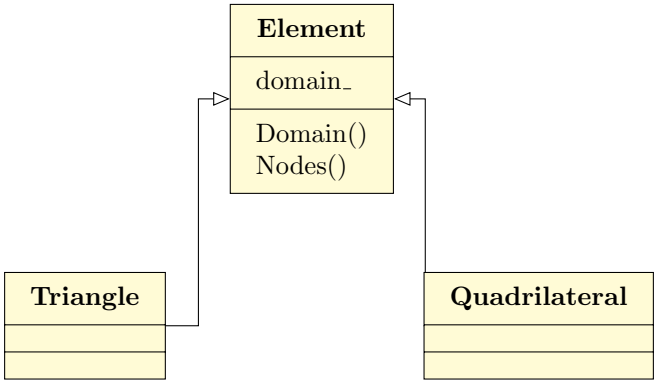
Statische Klassenelemente

- Statische Klassenelemente werden durch Unterstreichen gekennzeichnet.



Vererbung

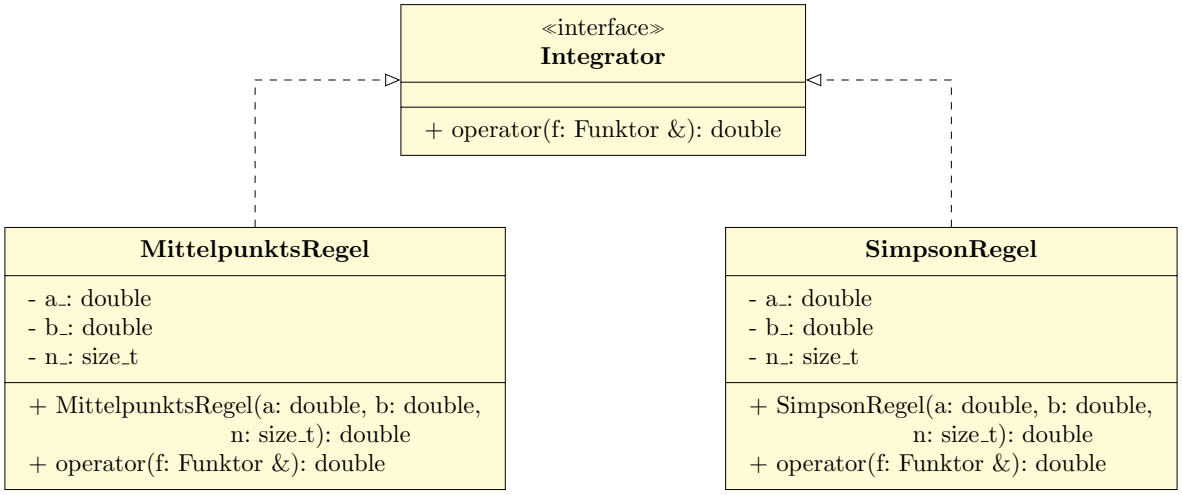
- Dass eine Klasse von einer anderen abgeleitet ist, wird dadurch gekennzeichnet, dass die Basisklasse mit der abgeleiteten Klasse durch eine Linie verbunden ist, die an ihrem Ausgangspunkt ein nicht ausgefülltes Dreieck hat, das sich von der Basisklasse zur abgeleiteten Klasse öffnet:



Schnittstellenbasisklassen, rein virtuelle Funktionen

- Schnittstellenbasisklassen werden durch die Zeile <<interface>> gekennzeichnet.

- Die Verbindung zu Klassen die das Interface implementieren ist wie eine Vererbung aber gestrichelt.

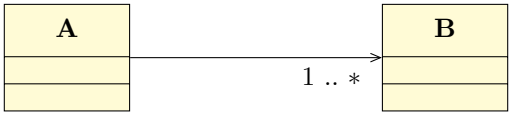


Assoziation

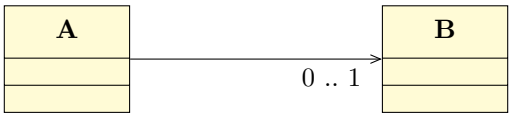
Die Assoziation zwischen zwei Elementen wird durch eine verbindende Linie dargestellt. Zahlen an der Linie können die Anzahl der Verbindungen und einen Namen für die Verbindung angeben. Eine Assoziation heißt, dass zwischen beiden Elementen eine Beziehung besteht, wie z.B. zwischen einem Bankkonto und einem Kunden.

A ist assoziiert mit einem B.

A ist assoziiert mit einem oder mehreren B.

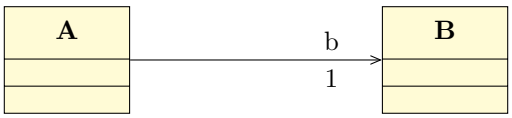


A ist assoziiert mit keinem oder einem B.



A ist assoziiert mit keinem, einem oder mehreren B.

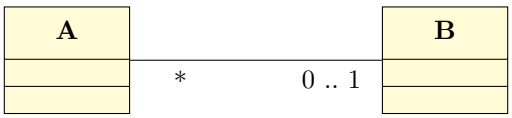
Eine Assoziation kann auch einen Namen haben:



z.B.

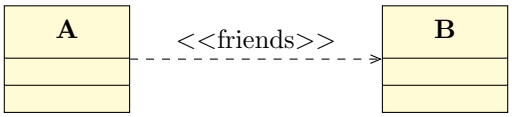
```
class B;
class A {
    B *b;
};
```

und es gibt auch Assoziationen in beide Richtungen:



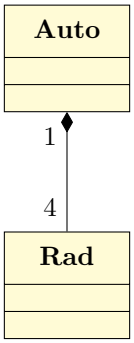
Abhängigkeiten

Eine Klasse kann von einer anderen Abhängen, z.B. weil sie ein `friend` ist:



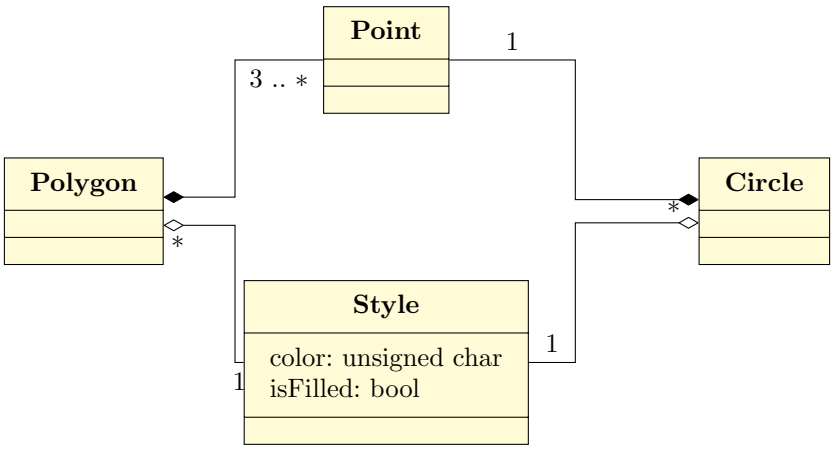
Komposition

Komposition wird durch verbindende Linie mit einer gefüllten Raute auf der Seite der zusammengesetzten Klasse dargestellt. Eine Komposition besteht zwischen Teilen, die zu einem ganzen gehören und für die das ganze auch die Verantwortung hat. Es handelt sich in der Regel um n zu eins Beziehungen.



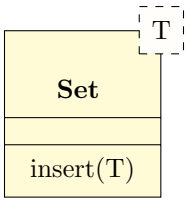
Aggregation

Aggregation wird durch verbindende Linie mit einer leeren Raute auf der Seite der aggregierten Klasse dargestellt. Die Aggregation ist eine etwas stärkere Beziehung als die Assoziation. Im Gegensatz zur Komposition werden die abhängigen Objekte nicht automatisch mit dem Hauptobjekt zerstört. Eine Universität kann z.B. als Komposition von Fakultäten dargestellt werden, diese sind aber nur Aggregationen von Professoren.

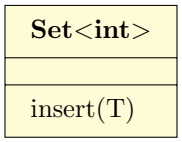


12.1 Templates in UML

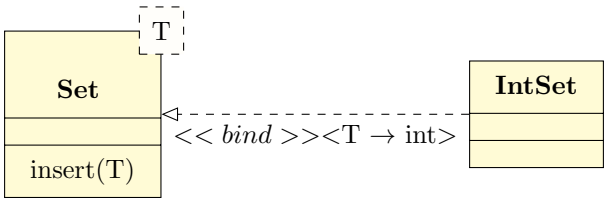
```
template<typename T>
class Set
{
  void insert (T element);
};
```



Realisierung von Set mit `T=int`.



oder



13 Die Standard Template Library (STL)

- Die Standard Template Library (STL)
  - ist eine Klassenbibliothek für unterschiedlichste Bedürfnisse
  - stellt Algorithmen zur Verfügung um mit diesen Klassen zu arbeiten.

- Außerdem formuliert sie Schnittstellen, die andere Sammlungen von Klassen zur Verfügung stellen müssen um wie STL-Klassen verwendet werden zu können oder Algorithmen zu schreiben die mit allen STL-artigen Containerklassen funktionieren.
- Die STL stellt eine neue Stufe der Abstraktion dar, die den Programmierer von der Notwendigkeit befreit oft benötigte Konstrukte wie dynamische Felder, Listen, binäre Bäume, Suchalgorithmen usw. selbst schreiben zu müssen.
- STL-Algorithmen werden so optimal wie möglich programmiert, d.h. wenn es einen STL-Algorithmus für ein Problem gibt, dann sollte man einen sehr guten Grund haben ihn nicht zu verwenden.
- Leider ist die STL nicht selbsterklärend.

STL Bestandteile

- Die Hauptbestandteile der STL sind:

  Container werden verwendet um Objekte eines bestimmten Typs zu verwalten. Die verschiedenen Container haben unterschiedliche Eigenschaften und damit zusammenhängende Vor- und Nachteile. Es sollte der jeweils am besten passende Container verwendet werden.

  Iteratoren ermöglichen es über den Inhalt eines Containers zu iterieren. Sie bieten eine einheitliche Schnittstelle für jeden STL-konformen Container unabhängig von seinem inneren Aufbau.

  Algorithmen arbeiten mit den Elementen eins Containers. Sie verwenden Iteratoren und müssen deshalb für eine beliebige Anzahl STL-konformer Container nur einmal geschrieben werden.

- Teilweise widerspricht auf den ersten Blick der Aufbau der STL der ursprünglichen Idee objektorientierter Programmierung, dass Algorithmen und Daten zusammengehören.

13.1 Container

STL-Containerklassen oder kurz Container verwalten eine Menge von Elementen des gleichen Typs. Je nach Containertyp gibt die STL Zusicherungen über die Ausführungsgeschwindigkeit bestimmter Operationen.

Es gibt zwei grundsätzlich verschiedene Arten von Containern:

Sequenzen sind geordnete Mengen von Elementen mit frei wählbarer Anordnung. Jedes Element hat seinen Platz, der vom Programmablauf und nicht vom Wert des Elements abhängt.

Assoziative Container sind nach einem bestimmten Sortierkriterium geordnete Mengen von Elementen bei denen die Position ausschließlich vom Wert des Elements abhängt.

Abbildung 2: Struktur eines `vector`

13.1.1 Sequenzen

STL-Sequenzcontainer sind Klassentemplates. Sie haben zwei Templateargumente, den Typ der zu speichernden Objekte und einen sogenannten Allokator mit dem sich die Speicherverwaltung ändern lässt (dies macht z.B. Sinn wenn man oft kleine Objekte anlegen und freigeben muss und dafür nicht jedes mal den Betriebsystem-Overhead bezahlen will). Für letzteren gibt es einen Defaultwert, der `new()` und `delete()` verwendet.

Vector ist ein Feld variabler Größe.

- das Hinzufügen und Entfernen von Elementen am Ende eines `vector` ist schnell, d.h. $O(1)$.
- der Elementzugriff kann direkt über einen Index erfolgen (wahlfreier Zugriff).

Amortisierte Komplexität

- In der Regel erfolgt das Hinzufügen von Elementen am Ende eines `std::vector` in $O(1)$.
- Im Einzelfall kann es jedoch deutlich länger dauern, insbesondere wenn der allozierte Speicher nicht mehr ausreicht. Dann muss neuer Speicher alloziert und im Regelfall umkopiert werden. Dies ist ein $O(N)$ Prozess.
- Allerdings reserviert die Standardbibliothek für einen wachsenden Vektor immer größere Speicherblöcke. Der Overhead ist dabei abhängig von der Länge des Vektors. Damit wird die Geschwindigkeit auf Kosten des Speicherverbrauchs optimiert.
- Der $O(N)$-Fall tritt also nur sehr selten auf.
- Dies nennt man “amortisierte Komplexität”.
- Ist bereits bekannt, dass insgesamt eine bestimmte Menge Elemente benötigt werden, dann kann man das mit der Methode `reserve(size_t size)` angeben. Es wird dadurch nicht die aktuelle Größe eines Vektors geändert, sondern nur die entsprechende Menge Speicher reserviert.
- Ähnliches gilt auch für die `deque`.

Beispiel STL-Vector

```
#include <iostream>
#include <vector>
#include <string>

int main(){
    std::vector<double> a(7);
    std::cout << a.size() << std::endl;
    for (int i=0;i<7;++i)
```

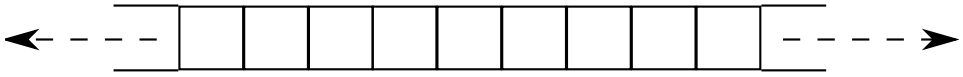


Abbildung 3: Struktur einer `deque`

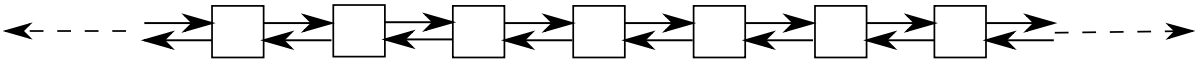


Abbildung 4: Struktur einer `list`

```
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::vector<double> c(a);
    std::cout << a.back() << "␣" << c.back() << std::endl;
    std::vector<std::string> b;
    b.resize(3);
    for (int i=2;i>=0;--i)
        std::cin >> b[i];
    b.resize(4);
    b[3] = "blub";
    b.push_back("blob");
    for (int i=0;i<b.size();++i)
        std::cout << b[i] << std::endl;
}
```

Deque , ist eine "double-ended" Queue, also ebenfalls ein Feld dynamischer Größe allerdings

- ist das Hinzufügen und Entfernen von Elementen auch am Anfang einer `deque` schnell, d.h. $O(1)$.
- kann der Elementzugriff direkt über einen Index erfolgen. Der Index eines bestimmten Elementes kann sich jedoch ändern, wenn Elemente am Anfang des Containers eingefügt werden.

List ist eine doppelt-verkettete Liste von Elementen.

- es gibt keinen direkten Zugriff auf ein bestimmtes Element.
- um auf das zehnte Element zuzugreifen ist es nötig am Anfang der `list` zu beginnen und die ersten neun Elemente zu durchlaufen, der Zugriff auf ein bestimmtes Element ist also $O(N)$.
- das Einfügen und Entfernen von Elementen ist an jeder Stelle der `list` schnell, d.h. $O(1)$.

Beispiel STL-List

```
#include <iostream>
#include <list>
#include <string>

int main()
{
    std::list<double> vals;
    for (int i=0;i<7;++i)
      vals.push_back(i*0.1);
```

Abbildung 5: Struktur eines `array`

```
    vals.push_front(-1);
    std::list<double> copy(vals);
    std::cout << vals.back() << "␣" << copy.back() << std::endl;
    std::cout << vals.front() << "␣" << copy.front() << std::endl;
    for (int i=0;i<vals.size();++i)
    {
      std::cout << i << ":␣" << vals.front() << "␣" << vals.size() << std::endl;
      vals.pop_front();
    }
    std::cout << std::endl;
    for (int i=0;i<copy.size();++i)
    {
      std::cout << i << ":␣" << copy.back() << "␣" << copy.size() << std::endl;
      copy.pop_back();
    }
}
```

Array , ist ein C++11-Ersatz für die klassischen C-Arrays, also für ein Feld mit fester Größe

- das `std::array` hat zwei Template Parameter. Den Typ der zu speichernden Daten und die Anzahl der Elemente.
- das Hinzufügen und Entfernen von Elementen ist nicht möglich.
- kann der Elementzugriff direkt über einen Index erfolgen.
- im Gegensatz zu C-Arrays kennt das Array seine Größe und kann wie ein Standardcontainer verwendet werden.

Beispiel STL-Array

```
#include <iostream>
#include <array>
#include <string>

int main(){
    std::array<double,7> a;
    std::cout << a.size() << std::endl;
    for (int i=0;i<7;++i)
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::array<double,7> c(a);
    std::cout << a.back() << "␣" << c.back() << std::endl;
    std::array<std::string,4> b;
    for (int i=2;i>=0;--i)
        std::cin >> b[i];
    b[3] = "blub";
    for (int i=0;i<b.size();++i)
        std::cout << b[i] << std::endl;
}
```

13.1.2 Assoziative Container

Set/Multiset

- `set` und `multiset` sind sortierte Mengen von Elementen.
- Während beim `set` jedes Element nur einmal vorkommen kann, kann es beim `multiset` mehrfach vorhanden sein.
- Bei einem Set ist es insbesondere wichtig schnell feststellen zu können, ob (und beim Multiset wie oft) sich ein Element in der Menge befindet.
- Das Suchen eines Elementes ist von optimaler Komplexität $O(\log(N))$.
- `set` und `multiset` haben drei Templateparameter: den Typ der Objekte, einen Vergleichsoperator und einen Allokator. Für die letzten gibt es Defaultwerte (`less` und den Standardallokator).

Map/Multimap

- `map` und `multimap` sind sortierte Paare aus zwei Variablen, einem Schlüssel und einem Wert. Die Wertepaare in der Map sind nach dem Schlüssel sortiert.
- Während bei der `map` jeder Schlüssel nur einmal vorkommen kann, kann er bei der `multimap` mehrfach vorhanden sein (unabhängig vom zugehörigen Wert).
- Eine `map` lässt sich schnell nach einem Schlüssel durchsuchen um dann auf den zugehörigen Wert zuzugreifen.
- Das Suchen eines Schlüssels ist von optimaler Komplexität $O(\log(N))$.
- `map` und `multimap` haben vier Templateparameter: den Typ der Schlüssel, den Typ der Werte, einen Vergleichsoperator und einen Allokator. Für die letzten gibt es Defaultwerte (less und new/delete).

13.1.3 Container Konzepte

- Die Eigenschaften von STL-Containern sind in bestimmte Kategorien unterteilt.
- Sie sind z.B. `Assignable`, `EqualityComparable`, `Comparable`, `DefaultConstructible`...
- Die Objekte einer Klasse, die in einem Container gespeichert werden sollen müssen `Assignable` (es gibt einen Zuweisungsoperator), `Copyable` (es gibt einen Copy-Konstruktor) , `Destroyable` (es gibt einen öffentlichen Destruktor), `EqualityComparable` (es gibt den `operator==`) und `Comparable` (es gibt den `operator<`) sein.

`Container`

- Ein `Container` selbst ist `Assignable` (es gibt einen Zuweisungsoperator), `EqualityComparable` (es gibt den `operator==`) und `Comparable` (es gibt den `operator<`).
- Assoziierte Typen:

`value_type`	Der Typ des gespeicherten Objektes. Muss `Assignable` sein, aber nicht `DefaultConstructible`.
`iterator`	Der Typ des Iterators. Muss ein `InputIterator` sein. Eine Konvertierung zum `const_iterator` muss existieren.
`const_iterator`	Ein Iterator über den die Werte der Objekte im Container abgefragt aber nicht geändert werden können.
`reference`	Der Typ einer Referenz auf den `value_type` des Containers.
`const_reference`	Der Typ einer konstanten Referenz auf den `value_type` des Containers.
`pointer`	Der Typ eines Pointers auf den `value_type` des Containers.
`const_pointer`	Ditto aber `const`.
`difference_type`	Ein Typ der sich eignet um die Differenz zwischen zwei Iteratoren auf den Container zu speichern.
`size_type`	Ein vorzeichenloser ganzzahliger Datentyp der jeden nicht-negativen Wert der Entfernung zwischen zwei Elementen des Containers speichern kann.

Zusätzlich zu den Methoden von `Assignable`, `EqualityComparable` und `Comparable` hat ein Container immer die Methoden:

`begin()`	Liefert einen Iterator auf das erste Element. Wenn der Container nicht verändert werden darf einen `const_iterator`
`end()`	wie `begin()` zeigt auf ein Element hinter das Letzte.
`size()`	liefert die Größe eines Containers, also die Anzahl der Elemente mit Rückgabetyp `size_type`
`max_size()`	Liefert die momentan maximale Größe. (`size_type`) die der Container haben kann.
`empty()`	Wahr wenn der Container leer ist (kann nach `bool` konvertiert werden)
`swap(b)`	Vertauscht Elemente mit Container `b`.

Spezialisierungen des `Container` Konzepts

`ForwardContainer`

- spezialisiert das `Container` Konzept.
- Es gibt einen `iterator` mit dem man vorwärts durch den Container laufen kann (`ForwardIterator`).

`ReversibleContainer`

- Es gibt einen Iterator mit dem man vorwärts und rückwärts durch Container laufen kann (`BidirectionalIterator`).
- Zusätzliche assoziierte Typen:

`reverse_iterator`	Iterator bei dem der `operator++` zum vorhergehenden Element wechselt.
`const_reverse_iterator`	`const` Version

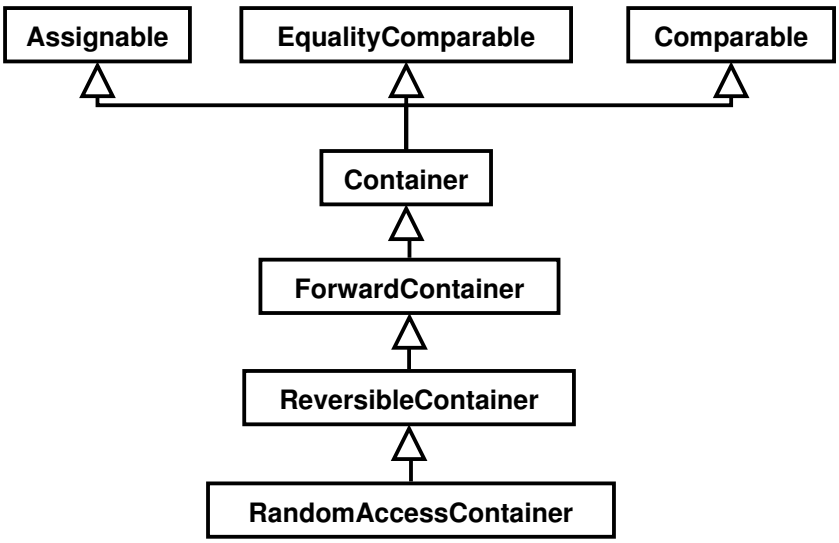


Abbildung 6: Container Konzepte

- Zusätzliche Methoden:

`rbegin()`	Liefert einen Iterator auf das erste Element eines umgekehrten Durchlaufs (letztes Element des Containers).
`rend()`	wie `rbegin()` zeigt auf ein Element hinter das Letzte eines umgekehrten Durchlaufs.

Implementierung

- `std::list`
- `std::set`
- `std::map`

`RandomAccessContainer`

- Spezialisiert `ReversibleContainer`.
- Es gibt einen `iterator` mit dem man wahlfrei über einen Index auf ein Element des Containers zugreifen kann (`RandomAccessIterator`).
- Zusätzliche Methoden `operator[](size_type)` (und `const` Version) Zugriffsoperator für wahlfreien Zugriff.

Implementierungen

- `std::vector`
- `std::deque`

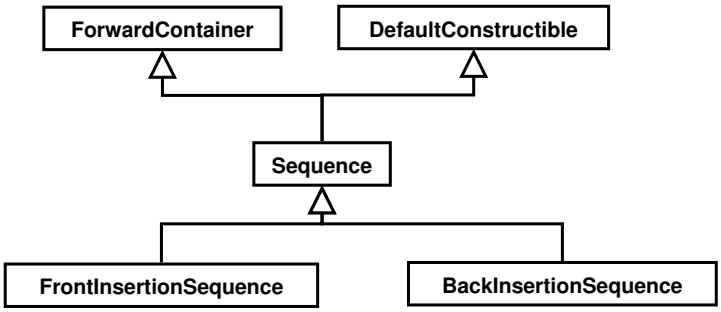


Abbildung 7: Sequenz Konzepte

`Sequence`

`Sequence` Methoden

Eine `Sequence` spezialisiert das Konzept des `ForwardContainer` (man kann also mindestens vorwärts über den Container iterieren) und ist `DefaultConstructible` (es gibt einen Konstruktor ohne Argumente/einen leeren Container).

`X(n,t)`	Erzeugt eine Sequenz mit `n>=0` Elementen initialisiert mit `t`.
`X(n)`	Erzeugt eine Sequenz mit `n>=0` initialisiert mit dem Defaultkonstruktor.
`X(i,j)`	Erzeugt eine Sequenz, die eine Kopie des Bereichs [`i`,`j`) ist. `i` und `j` sind `InputIterator`.
`insert(p, t)`	Fügt das Element `t` vor dem Element ein auf das der Iterator `p` zeigt und liefert einen Iterator zurück der auf das eingefügte Element zeigt.
`insert(p, i, j)`	Dito für den Bereich `InputIterator` [`i`,`j`).
`insert(p, n, t)`	Fügt `n` Kopien des Elements `t` vor dem Element ein auf das der Iterator `p` zeigt und liefert einen Iterator zurück der auf das letzte eingefügte Element zeigt.
`erase(p)`	Ruft den Destruktor für das Element auf, auf den der Iterator `p` zeigt und entfernt es aus dem Container.
`erase(p,q)`	Dito für den Bereich [`p`,`q`).
`erase()`	Löscht alle Elemente.
`resize(n,t)`	Verkleinert oder vergrößert auf `n` und initialisiert neue Elemente mit `t`
`resize(n)`	`resize(n, T())`

Komplexitätsgarantien für Sequenzen

- Die Konstruktoren `X(n,t)` `X(n)` `X(i,j)` haben lineare Komplexität.
- Das Einfügen von Elementen mit `insert(p, t)`, `insert(p, i, j)` und das Löschen mit `erase(p,q)` haben lineare Komplexität.
- Die Komplexität des Einfügens und Löschens einzelner Elemente ist von der jeweiligen Sequenzimplementierung abhängig.

`BackInsertionSequence`

Zusätzliche Methoden zum `Sequence` Konzept:

`back()`	Liefert eine Referenz auf das letzte Element.
`push_back(t)`	Fügt eine Kopie von `t` nach dem letzten Element ein.
`pop_back()`	Löscht das letzte Element der Sequenz.

Komplexitätsgarantien

`back`, `push_back`, und `pop_back` haben eine amortisiert konstante Komplexität, d.h. im Einzelfall kann es länger dauern aber im Mittel ist die Zeit unabhängig von der Anzahl Elemente.

Implementierungen

- `std::vector`
- `std::list`
- `std::deque`

`FrontInsertionSequence`

Zusätzliche Methoden zum `Sequence` Konzept:

`front()`	Liefert eine Referenz auf das erste Element.
`push_front(t)`	Fügt eine Kopie von `t` vor dem ersten Element ein.
`pop_front()`	Löscht das erste Element der Sequenz.

Komplexitätsgarantien

`front()`, `push_front()`, und `pop_front()` haben eine amortisiert konstante Komplexität.

Implementierungen

- `std::list`
- `std::deque`

STL-Sequenzcontainer

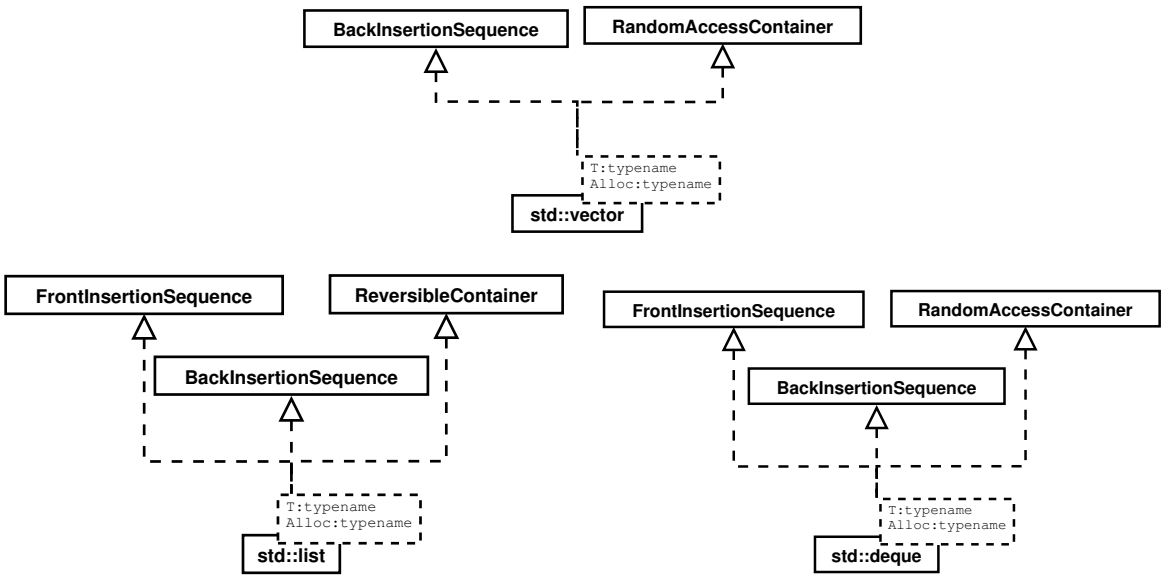


Abbildung 8: STL Sequenzcontainer

Assoziative Container

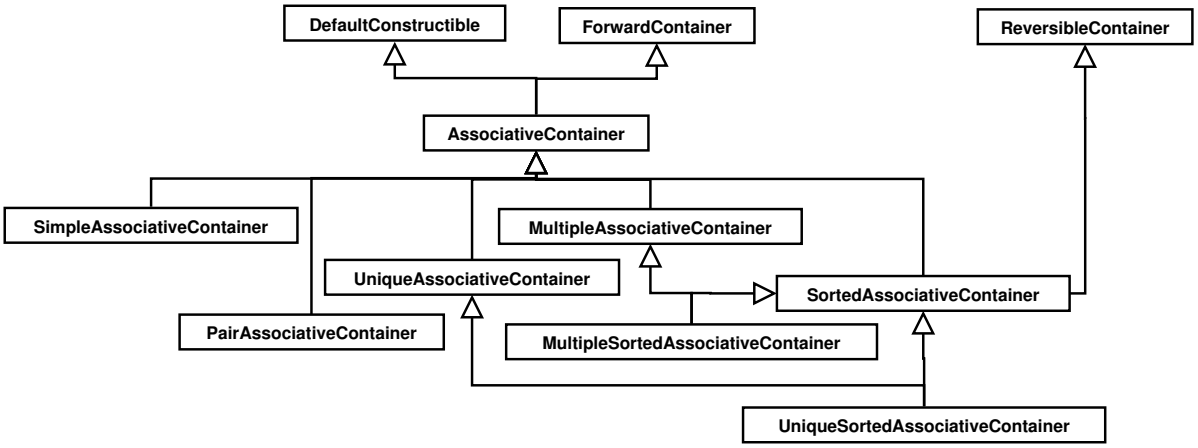


Abbildung 9: Assoziative Container Konzepte

`AssociativeContainer`

- Spezialisiert `ForwardContainer` und `DefaultConstructible`.
- Zusätzlicher assoziierter Typ: `key_type` ist der Typ eine Schlüssels.
- Zusätzliche Methoden:

`erase(k)`	Lösche alle Methoden deren Schlüssel gleich `k` ist.
`erase(p)`	Löscht das Element auf das der Iterator `p` zeigt.
`erase(p,q)`	Dito für den Bereich [`p`,`q`).
`clear()`	Löscht alle Elemente.
`find(k)`	Liefert einen Iterator zurück der auf das Element mit dem Schlüssel `k` zeigt oder `end()` wenn der Schlüssel nicht existiert.
`count(k)`	Liefert die Anzahl der Elemente zurück deren Schlüssel gleich `k` ist.
`equal_range(k)`	Liefert ein `pair p` von Iteratoren zurück so dass [`p.first`,`p.second`) alle Elemente enthält deren Schlüssel gleich `k` ist.

- Zusicherungen:

  Kontinuierlicher Speicher : alle Elemente mit dem gleichen Schlüssel folgen unmittelbar aufeinander.

  Unveränderbarkeit der Schlüssel : Der Schlüssel jedes Elementes eines assoziativen Containers ist unveränderbar.

Komplexitätsgarantien

`erase(k)`	Durchschnittliche Komplexität höchstens $O(\log(size() + count(k))$
`erase(p)`	Durchschnittliche Komplexität konstant
`erase(p,q)`	Durchschnittliche Komplexität höchstens $O(\log(size() + N))$
`count(k)`	Durchschnittliche Komplexität höchstens $O(\log(size() + count(k))$
`find(k)`	Durchschnittliche Komplexität höchstens logarithmisch
`equal_range(k)`	Durchschnittliche Komplexität höchstens logarithmisch

Das sind nur durchschnittliche Komplexitäten!

Im schlimmsten Fall können die Komplexitäten wesentlich schlechter sein!

`SimpleAssociativeContainer` **und** `PairAssociativeContainer`
spezialisieren den `AssociativeContainer`.

`SimpleAssociativeContainer`
Hat die folgenden Einschränkungen:

- `key_type` und `value_type` müssen gleich sein.
- `iterator` und `const_iterator` müssen den gleichen Typ haben.

`PairAssociativeContainer`

- Fügt den assoziierten Datentyp `mapped_type` hinzu. Der Container bildet `key_type` auf `mapped_type` ab.
- Der `value_type` ist `std::pair<key_type,mapped_type>`.

`SortedAssociativeContainer`
verwenden ein Ordnungskriterium zum Sortieren der Schlüssel. Zwei Schlüssel sind äquivalent wenn keiner kleiner ist als der andere.

Zusätzliche assoziierte Typen

`key_compare`	Der Typ der `StrictWeakOrdering` implementiert um zwei Schlüssel zu vergleichen.
`value_compare`	Der Typ der `StrictWeakOrdering` implementiert um zwei Values zu vergleichen. Vergleicht zwei Objekte vom Typ `value_type` indem er deren Schlüssel an `key_compare` weiterreicht.

Zusätzliche Methoden

`key_compare()`	Liefert das Schlüsselvergleichsobjekt zurück.
`value_compare()`	Liefert das Valuevergleichsobjekt zurück.
`lower_bound(k)`	Liefert einen iterator der auf das erste Element zeigt dessen Schlüssel nicht kleiner ist als `k`, oder `end()` wenn es kein solches Element gibt.
`upper_bound(k)`	Liefert einen iterator der auf das erste Element zeigt dessen Schlüssel größer ist als `k`, oder `end()` wenn es kein solches Element gibt.

`SortedAssociativeContainer`

Komplexitätsgarantien

- `key_comp`, `value_comp` und `erase(p)` haben konstante Komplexität
- `erase(k)` ist $O(\log(size()) + count(k))$
- `erase(p,q)` ist $O(\log(size() + N))$
- `find` ist logarithmisch.
- `count(k)` ist $O(\log(size() + count(k))$
- `lower_bound`, `upper_bound`, und `equal_range` sind logarithmisch

Zusicherungen

value_compare: wenn `t1`und `t2` die assoziierten Schlüssel `k1` und `k2` haben, dann liefert

`value_compare()(t1,t2)==key_compare(k1,k2)` garantiert `true`

Aufsteigende Reihenfolge der Elemente wird garantiert.

`UniqueAssociativeContainer` **und** `MultipleAssociativeContainer`

Ein `UniqueAssociativeContainer` ist ein `AssociativeContainer` mit der zusätzlichen Eigenschaft, dass jeder Schlüssel nur einmal vorkommt. Ein `MultipleAssociativeContainer` ist ein `AssociativeContainer` in dem jeder Schlüssel mehrfach vorkommen kann.

Zusätzliche Methoden:

`X(i,j)`	Erzeugt einen assoziativen Container der die Elemente im `InputIterator` Bereich [`i`,`j`) enthält.
`insert(t)`	Füge den `value_type` t ein und liefere ein `std:pair` zurück aus einem iterator der auf die Kopie von `t` zeigt und einem `bool` (`true` wenn `t` eingefügt wurde)
`insert(i,j)`	Fügt alle Elemente im `InputIterator` Bereich [`i`,`j`) ein.

Komplexitätsgarantien

- Die durchschnittliche Komplexität von `insert(t)` ist höchstens logarithmisch.
- Die durchschnittliche Komplexität von `insert(i,j)` ist höchstens $O(N * \log(size()) + N)$, wobei `N=j-i`

Assoziative Containerklassen

Eigenschaften der verschiedenen Containerklassen

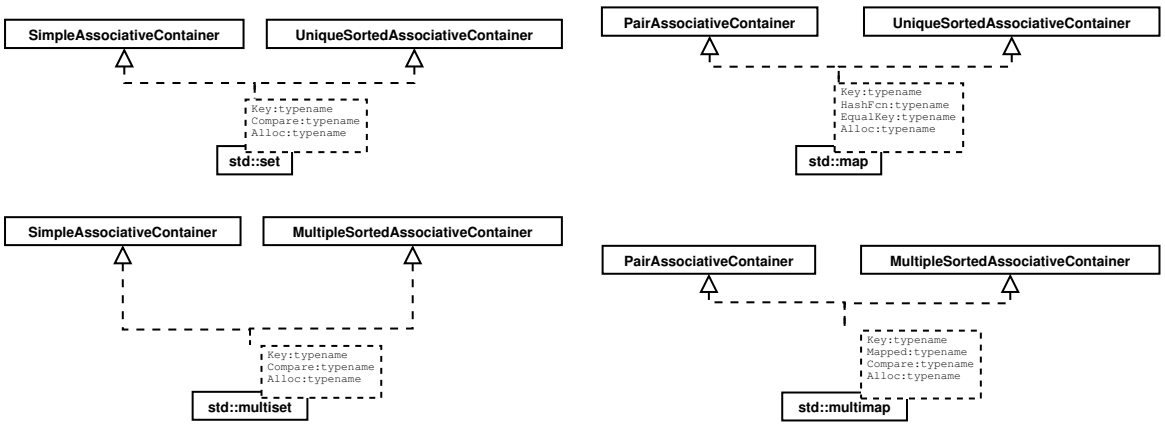


Abbildung 10: Assoziative Containerklassen

	`vector`	`deque`	`list`	`set`	`map`
Typische innere Datenstruktur	Dynamisches Feld	Feld von Feldern	Doppelt verkettete Liste	Binärer Baum	Binärer Baum
Elemente	Wert	Wert	Wert	Wert	Schlüssel/ Wert
Suchen	Langsam	Langsam	Sehr Langsam	Schnell	Schnell nach Schlüssel
Einfügen / Entfernen schnell	Am Ende	An Anfang und Ende	Überall	–	–
Gibt Speicher entfernter Elemente frei	Nie	Manchmal	Immer	Immer	Immer
Erlaubt Reservierung von Speicher	Ja	Nein	–	–	–

Tabelle 1: Eigenschaften der verschiedenen Containerklassen

Welchen Container sollte man verwenden?

- Wenn es keinen anderen Grund gibt, dann `vector`, da es die einfachste Datenstruktur ist und wahlfreien Zugriff erlaubt.
- Wenn Elemente oft auch am Anfang oder Ende eingefügt/entfernt werden müssen, verwendet man eine `deque`. Dieser Container wird auch wieder kleiner wenn Elemente entfernt werden.
- Müssen Elemente überall eingefügt/entfernt/verschoben werden müssen, ist eine `list` der Container der Wahl. Auch das Verschieben von einer `list` in eine andere kann in konstanter Zeit durchgeführt werden. Es gibt keinen wahlfreien Zugriff.
- Wenn es möglich sein soll schnell wiederholt nach Elementen zu suchen, verwendet man ein `set` oder `multiset`.

- Ist es notwendig Paare von Schlüsseln und Werten zu verwalten (wie in einem Wörter- oder Telefonbuch) dann verwendet man eine `map` oder `multimap`.

13.2 Iteratoren

Motivation

- Wie greift man auf die Einträge eines Assoziativen Containers zu, z.B. ein `set`?
- Wie schreibt man einen Algorithmus, der für alle Arten von STL-Containern funktioniert?
- Erforderlich ist ein allgemeines Verfahren um über die Elemente eines Containers zu iterieren.
- Am schönsten wäre es, wenn das auch für traditionelle C-Arrays funktioniert.
- Dabei sollte es möglich sein besondere Fähigkeiten eines Containers (wie den wahlfreien Zugriff eines `vector`) immer noch nutzen zu können.

Ein Iterator

- ist ein Objekt einer Klasse mit dem man über Elemente in einem Container iterieren kann (Container und Iterator müssen nicht zur selben Klasse gehören).
- ist `Assignable`, `DefaultConstructible` und `EqualityComparable`.
- zeigt auf eine bestimmte Position in einem Containerobjekt.
- Zum nächsten Element des Containerobjektes kommt man über den `operator++` des Iterators.

Beispiel für Iteratoren

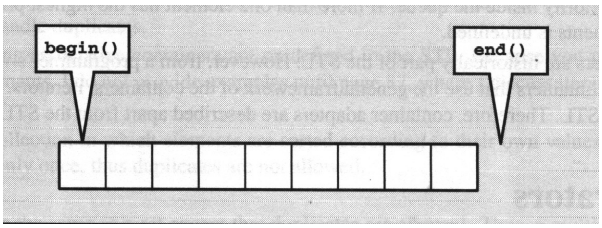


Abbildung 11: Iterator über einen Container

Iteratoren für Container

- Jeder Container gibt den Typ der Iteratorobjekte für diesen Container durch ein `typedef` an:
  - `Container::iterator` Ein Iterator mit Schreib- und Leserechten.
  - `Container::const_iterator` Ein read-only Iterator

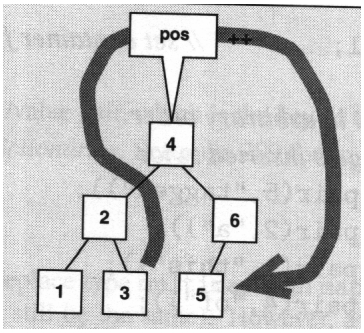


Abbildung 12: Iterator über ein Set

- Zusätzlich verfügt jeder Container über die folgenden Methoden:
  - `begin()` liefert einen Iterator zurück der auf das erste Element des Containerobjektes zeigt.
  - `end()` liefert einen Iterator, der auf das Ende des Containers zeigt, d.h. auf ein Element nach dem letzten Element des Containerobjektes.
- Bei leeren Containern ist `begin()==end()`.

Erstes Iterator Beispiel: Headerfile

```
#include<iostream>

template<class T>
void print(const T &container)
{
    for(typename T::const_iterator i=container.begin();
                                   i!=container.end(); ++i)
        std::cout << *i << "␣";
     std::cout << std::endl;
}

template<class T>
void push_back_a_to_z(T &container)
{
    for(char c='a'; c <='z'; ++c)
        container.push_back(c);
}
```

Erstes Iterator Beispiel: Sourcefile

```
#include"iterator1.hh"
#include<list>
#include<vector>

int main(int argc, char** argv)
{
    std::list<char> listContainer;
    push_back_a_to_z(listContainer);
    print(listContainer);
```

```
    std::vector<int> vectorContainer;
    push_back_a_to_z(vectorContainer);
    print(vectorContainer);
}
```

Ausgabe:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
    117 118 119 120 121 122
```

Iterator Konzepte

- Iteratoren können zusätzliche Eigenschaften haben.
- Dies hängen von den spezifischen Eigenschaften des Containers ab.
- Damit lassen sich effizientere Algorithmen für Container schreiben, die über zusätzliche Fähigkeiten verfügen.
- Iteratoren lassen sich nach ihren Fähigkeiten gruppieren.

Typen von Iteratoren	Fähigkeit
Input iterator	Vorwärts lesen
Output iterator	Vorwärts schreiben
Forward iterator	Vorwärts lesen und schreiben
Bidirectional iterator	Vorwärts und rückwärts lesen und schreiben
Random access iterator	Lesen und schreiben mit wahlfreiem Zugriff

Tabelle 2: Vordefinierte Typen von Iterator

`Trivial Iterator`

- Ein `TrivialIterator` ist ein Objekt das auf ein anderes Objekt zeigt und sich wie ein Pointer dereferenzieren lässt. Es gibt keine Garantie, dass arithmetische Operationen möglich sind.
- Assoziierte Typen:: `value_type` ist der Typ des Objekts auf das der Iterator zeigt.
- Methoden:

`ITERTYPE()`	Defaultkonstruktor.
`operator*()`	Dereferenzierung.
`*i=t`	Wenn der Iterator `x` veränderlich ist, dann ist eine Zuweisung möglich.
`operator->()`	Zugriff auf Methoden und Attribute des referenzierten Objekts.

- Komplexitätsgarantien: Alle Operationen haben amortisiert konstante Komplexität.

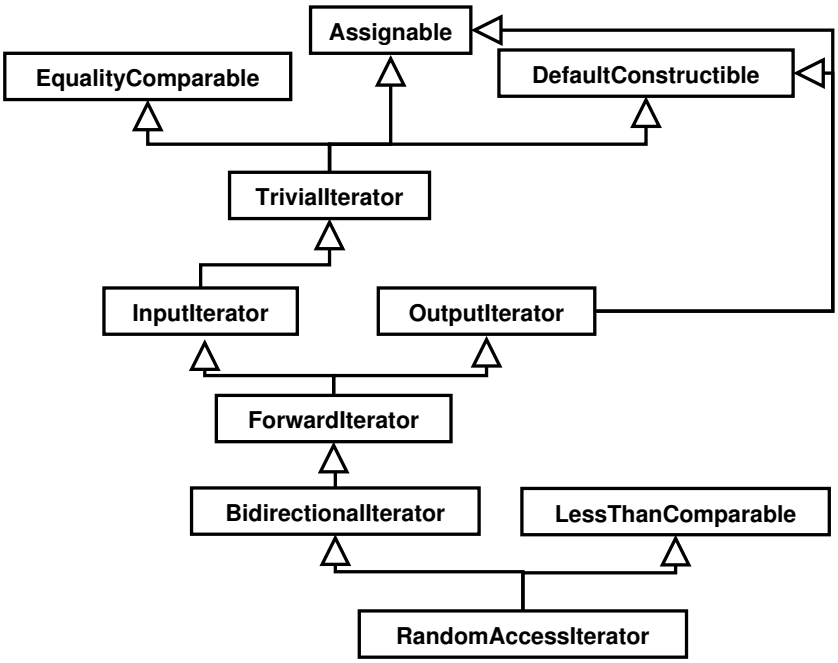


Abbildung 13: Iterator Konzepte

`Input Iterator`

- Ein `Input Iterator` ist ein Objekt das auf ein anderes Objekt zeigt, das sich wie ein Pointer dereferenzieren lässt und das sich inkrementieren lässt um einen Iterator auf das nächste Objekt im Container zu erhalten.
- Assoziierte Typen: `difference_type`: Eine vorzeichenbehaftete Ganzzahl um die Entfernung zwischen zwei Iteratoren (bzw. die Anzahl Elemente in dem Bereich dazwischen) zu speichern.
- Methoden:

Ausdruck	Wirkung
`x=*i`	ist dereferenzierbar
`++i`	Macht einen Schritt vorwärts
`(void)i++`	Macht einen Schritt vorwärts, identisch zu `++i`.
`*i++`	Identisch zu `T t=*i;++i;return t;`.

- Komplexitätsgarantien: Alle Operationen haben amortisiert konstante Komplexität.

`Output Iterator`

- Ein `Output Iterator` ist ein Objekt auf das sich schreiben und das sich inkrementieren lässt.
- `Output Iterator` sind nicht vergleichbar und müssen keinen `value_type` und `difference_type` definieren.
- Vergleichbar einem Endlospapierdrucker.

- Inkrementieren und zuweisen muss sich abwechseln. Vor dem ersten Inkrement muss eine Zuweisung erfolgen, vor jeder weiteren Zuweisung ein Inkrement.
- Methoden:

Ausdruck	Wirkung
`ITERTYPE(i)`	Copy-Konstruktor.
`*i=value`	Schreibt einen Wert an die Stelle auf die der Iterator zeigt.
`++i`	Macht einen Schritt vorwärts.
`i++`	Macht einen Schritt vorwärts, identisch zu `++i`.

- Komplexitätsgarantien: Alle Operationen haben amortisiert konstante Komplexität.

`Forward Iterator`

- Ein `Forward Iterator` entspricht der gängigen Vorstellung einer linearen Folge von Werten. Mit einem `Forward Iterator` sind (im Gegensatz zu einem `Output Iterator`) mehrere Durchgänge über einen Container möglich.
- Definiert keine zusätzlichen Methoden im Vergleich zum `Input Iterator`.
- Inkrementieren macht frühere Kopien des Iterators nicht ungültig.
- Ein `forward iterator` ist kein `output iterator` da `++i` nicht immer auf eine beschreibbare Stelle zeigt, z.B. wenn `i==end()`.
- Komplexitätsgarantien: Alle Operationen haben amortisiert konstante Komplexität.
- Zusicherungen: Für zwei Iteratoren `i` und `j` gilt falls `i == j` dann `++i == ++j`

`Bidirectional Iterator`

- Kann vorwärts und rückwärts verwendet werden.
- Iteration von `list`, `set`, `multiset`, `map` und `multimap`
- Zusätzliche Methoden:

`--i`	Macht einen Schritt rückwärts.
`i--`	Macht einen Schritt rückwärts

- Komplexitätsgarantien: Alle Operationen haben amortisiert konstante Komplexität.
- Zusicherungen: Wenn `i` auf ein Element im Container zeigt, dann sind `++i;--i;` und `--i;++i;` Nulloperationen.

`Random Access Iterator`

- Ein `Random Access Iterator` ist ein `Bidirectional Iterator`, der zusätzlich Methoden zur Verfügung stellt um in konstanter Zeit Schritte beliebiger Größe vorwärts und rückwärts zu machen. Er erlaubt im wesentlichen alle Operationen, die mit Pointern möglich sind.
- Wird bereitgestellt von `vector`, `deque`, `string` und C-Arrays.

- Zusätzliche Methoden:

`i+n`	Liefert einen Iterator auf das `n`te Element.
`i-n`	Liefert einen Iterator auf das `n`te vorhergehende Element.
`i+=n`	Geht `n` Elemente vorwärts.
`i-=n`	Geht `n` Elemente rückwärts.
`i[n]`	Entspricht *(i+n).
`i-j`	Liefert die Entfernung zwischen `i` und `j`, bzw. die Anzahl der Elemente d

- Komplexitätsgarantien: Alle Operationen haben amortisiert konstante Komplexität.

`Random Access Iterator`

- Zusicherungen:
  - Wenn `i+n` definiert ist, dann ist `i+=n;i-=n;` eine Nulloperationen entsprechend für `i-n`.
  - Wenn `i-j` definiert ist, dann gilt `i == j + (i-j)`.
  - Wenn `i` von `j` durch eine Reihe von Inkrement- oder Dekrementoperationen erreichbar ist, dann ist `i-j >= 0`.
  - Zwei Operatoren sind `Comparable`.

Beispiel Vector

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<double> a(7);
    std::cout << a.size() << std::endl;
    for (int i=0;i<a.size();++i)
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::vector<double> c(a);
    std::cout << a.back() << " " << c.back() << std::endl;
    std::vector<std::string> b;
    b.resize(3);
    typedef std::vector<std::string>::reverse_iterator VectorRevIt;
    for (VectorRevIt i=b.rbegin();i!=b.rend();++i)
        std::cin >> *i;
    b.resize(4);
    b[3] = "blub";
    b.push_back("blob");
    typedef std::vector<std::string>::iterator VectorIt;
    for (VectorIt i=b.begin();i<b.end();++i)
        std::cout << *i << std::endl;
}
```

Beispiel List

```
#include <iostream>
#include <list>

int main()
{
    std::list<double> vals;
    for (int i=0;i<7;++i)
```

```
        vals.push_back(i*0.1);
    vals.push_front(-1);
    std::list<double> copy(vals);
    typedef std::list<double>::iterator ListIt;
    for (ListIt i=vals.begin();i!=vals.end();++i,++i)
        i=vals.insert(i,*i+0.05);
    std::cout << "vals size: " << vals.size() << std::endl;
    for (ListIt i=vals.begin();i!=vals.end();i=vals.erase(i))
        std::cout << *i << " ";
    std::cout << std::endl << "vals size: " << vals.size() << std::endl;
    typedef std::list<double>::reverse_iterator ListRevIt;
    for (ListRevIt i=copy.rbegin();i!=copy.rend();++i)
        std::cout << *i << " ";
    copy.clear();
    std::cout << std::endl << "copy size: " << copy.size() << std::endl;
}
```

Beispiel Set: Speicher für globale Optimierung

```
#include<vector>
#include<set>

class Result
{
    double residuum_;
    std::vector<double> parameter_;
  public:
    bool operator<(const Result &other) const
    {
        if (other.residuum_<=residuum_)
            return false;
        else
            return true;
    }
    double Residuum() const
    {
        return residuum_;
    }
    Result(double res) : residuum_(res)
    {};
};

#include<iostream>
#include<set>
#include"result.h"

int main()
{
    std::multiset<Result> valsMSet;
    for (int i=0;i<7;++i)
        valsMSet.insert(Result(i*0.1));
    for (int i=0;i<7;++i)
        valsMSet.insert(Result(i*0.2));
    typedef std::multiset<Result>::iterator MultiSetIt;
    for (MultiSetIt i=valsMSet.begin();i!=valsMSet.end();++i)
        std::cout << i->Residuum() << " ";
    std::cout << std::endl << "valsMSet size: " << valsMSet.size() << std::endl;
    std::set<Result> vals(valsMSet.begin(),valsMSet.end());
    typedef std::set<Result>::iterator SetIt;
    for (SetIt i=vals.begin();i!=vals.end();++i)
        std::cout << i->Residuum() << " ";
    std::cout << std::endl << "vals size: " << vals.size() << std::endl;
}
```

Output:

```
0 0 0.1 0.2 0.2 0.3 0.4 0.4 0.5 0.6 0.6 0.8 1 1.2
valsMSet size: 14
0 0.1 0.2 0.3 0.4 0.5 0.6 0.8 1 1.2
vals size: 10
```

Beispiel Map: Parameterverwaltung

```
#include <iostream>
#include <map>

template<typename T>
bool GetValue(const std::map<std::string,T> &container,std::string key, T
    &value)
{
    typename std::map<std::string,T>::const_iterator
        element=container.find(key);
    if (element!=container.end())
    {
        value=element->second;
        return(true);
    }
    else
        return(false);
}

template<typename T>
T GetValue(const std::map<std::string,T> &container,std::string key,bool
    abort=true,T defValue=T())
{
    typename std::map<std::string,T>::const_iterator
        element=container.find(key);
    if (element!=container.end())
        return(element->second);
    else
    {
        if (abort)
        {
            std::cerr << "GetValue: key \"" << key << "\" not found";
            std::cerr << std::endl << std::endl << "Available keys: " <<
                std::endl;
            for(element=container.begin();element!=container.end();++element)
                std::cerr << element->first << std::endl;
            throw "No Value found";
        }
    }
    return(defValue);
}
```

13.3 STL Algorithmen

- Die STL definiert viele Algorithmen die sinnvoll auf die Objekte von Containern angewendet werden können, z.B. zum Suchen, Sortieren, Kopieren ...
- Es handelt sich um globale Funktionen nicht um Methoden der Container.
- Für Input und Output werden Iteratoren verwendet.

- Der Header `algorithm` muss eingebunden werden.
- Im Header `numeric` befinden sich Algorithmen die Berechnungen durchführen.

Beispiel

```
#include<vector>
#include<iostream>
#include<algorithm>

template<typename T>
void print (const T &elem)
{
  std::cout << elem << "␣";
}

int add(int &elem)
{
  elem+=5;
}

int main()
{
  std::vector<int> coll(7,3);
  std::for_each(coll.begin(), coll.end(), print<int>);
  std::cout << std::endl;
  std::for_each(coll.begin(), coll.end(), add);
  std::for_each(coll.begin(), coll.end(), print<int>);
  std::cout << std::endl;
  std::cout << std::endl;
}
```

Iteratorbereiche

- Alle Algorithmen arbeiten auf einer (oder mehreren) Menge von Elementen die durch Iteratoren begrenzt wird. Dies bezeichnet man auch als einen Range.
- Ein Range ist eingegrenzt von den Iteratoren: `[begin,end)` . `begin` zeigt auf das erste Element und `end` auf das erste Element nach dem letzten.
- Es muss kann sich auch um Teilmengen eines Containers handeln.
- Der Benutzer ist dafür verantwortlich, dass es sich um eine gültige/sinnvolle Menge handelt, d.h. dass man von `begin` aus zu `end` gelangt, wenn man über die Elemente iteriert.
- Bei Algorithmen, die mehr als einen Iteratorbereich erwarten, wird das Ende nur für den ersten Bereich angegeben. Für alle anderen wird angenommen, dass sie genauso viele Elemente enthalten (können):

  ```
  std::copy(coll1.begin(), coll1.end(), coll2.begin()))
  ```

Algorithmen mit Suffix

Manchmal gibt es zusätzliche Versionen eines Algorithmus, die durch ein Suffix gekennzeichnet werden. Dies dient dazu die Unterscheidung verschiedener Varianten für Compiler und Programmierer zu erleichtern.

_if Suffix

- Das Suffix _if wird inzugefügt, wenn zwei Varianten eines Algorithmus existieren, die sich nicht durch die Zahl der Argumente unterscheiden, sondern nur durch deren Bedeutung.
- Bei der Version ohne Suffix ist das letzte Argument ein Wert, mit dem die Elemente verglichen werden.
- Die Version mit Suffix `_if` erwartet ein Prädikat, d.h. eine Funktion die `bool` zurückliefert (s.u.) als Parameter. Diese wird für alle Elemente ausgewertet.
- Es gibt nicht von allen Algorithmen eine Version mit `_if` Suffix, z.B. wenn sich die Anzahl der Argumente zwischen der Wert- und Prädikatversion unterscheidet..
- Beispiel: `find` und `find_if`.

_copy Suffix

- ohne Suffix wird der Inhalt des jeweiligen Elements geändert, mit Suffix werden die Elemente kopiert und dabei verändert.
- Diese Version des Algorithmus hat jeweils ein zusätzliches Argument (einen Iterator auf den Platz an den kopiert werden soll).
- Beispiel: `reverse` und `reverse_copy`.

Es gibt auch den kombinierten Suffix `_copy_if`

For-each

Der einfachste und allgemeinste Algorithmus dürfte `for_each(b,e,f)` sein. Dabei wird der Funktor `f(x)` für jedes Element im Range `[b:e)` aufgerufen. Da hierbei Referenzen an f übergeben werden können ist auch eine Veränderung der Wert im Range möglich.

Nicht verändernde Algorithmen

`count(b,e,v)`	Zählt die Anzahl Elemente im Range `[b:e)` die gleich `v` sind	integer
`count_if(b,e,v,f)`	Zählt die Anzahl Elemente im Range `[b:e)` für die `f(*p)` true ist	integer
`all_of(b,e,f)`	Bedingung `f(*p)` ist wahr für alle Elemente in Range `[b:e)`	bool
`any_of(b,e,f)`	Bedingung `f(*p)` ist wahr für ein Element im Range `[b:e)`	bool
`none_of(b,e,f)`	Bedingung `f(*p)` ist wahr für kein Element im Range	bool
`min_element(b,e)`	Kleinstes Element im Range `[b:e)`	iterator
`min_element(b,e,f)`	Kleinstes Element im Range `[b:e)`	iterator
`max_element(b,e)`	Größtes Element im Range `[b:e)`	iterator
`max_element(b,e,f)`	Größtes Element im Range `[b:e)`	iterator
`minmax_element(b,e)`	Iteratoren auf kleinstes und größtes Element im Range `[b:e)`	pair(min,max)
`minmax_element(b,e,f)`	Iteratoren auf kleinstes und größtes Element im Range `[b:e)`	pair(min,max)

Suchalgorithmen

`find(b,e,v)`	Finde erstes Element im Range `[b:e)` mit Wert `v` oder `e`	iterator	`adjacent_find(b,e)`
`find_if(b,e,f)`	Finde erstes Element im Range `[b:e)` für das `f(*p)` wahr ist oder `e`	iterator	`adjacent_find(b,e,f`
`find_if_not(b,e,f)`	Finde erstes Element im Range `[b:e)` für das `f(*p)` falsch ist oder `e`	iterator	`search(b,e,b2,e2)`
`find_first_of(b,e,b2,e2)`	Finde erstes Element im Range `[b:e)` das einem Element des Ranges `[b2:e2)` gleich ist oder `e`	iterator	
`find_first_of(b,e,b2,e2,f)`	Finde erstes Element`p` im Range `[b:e)` für das `f(*p,*q)` für ein Element `q` des Ranges `[b2:e2)` wahr ist oder `e`	iterator	`search(b,e,b2,e2,f)`
`find_end(b,e,b2,e2)`	Finde letztes Element im Range `[b:e)` das einem Element des Ranges `[b2:e2)` gleich ist oder `e`	iterator	`search_n(b,e,n,v)`
`find_end_of(b,e,b2,e2,f)`	Finde letztes Element`p` im Range `[b:e)` für das `f(*p,*q)` für ein Element `q` des Ranges `[b2:e2)` wahr ist oder `e`	iterator	`search_n(b,e,n,v,f)`

Vergleichsgorithmen

`equal(b,e,b2)`	Wahr wenn alle Elemente der zwei Ranges `[b:e]` und `[b2:b2+(b-e)` gleich sind	bool
`equal(b,e,b2,f)`	Wahr wenn `f(*p,*q)` für alle Elemente der zwei Ranges wahr ist	bool
`mismatch(b,e,b2)`	Liefert erstes Element in `[b:e)` und in `[b2:b2+(b-e)` zurück die ungleich sind, oder zweimal `e`	`pair<iterator>`
`mismatch(b,e,b2,f)`	Liefert erstes Element in `[b:e)` und in `[b2:b2+(b-e)` zurück für die `f(*p,*q)` falsch ist, oder zweimal `e`	`pair<iterator>`
`lexicographical_compare(b,e,b2,e2)`	Lexikographischer Vergleich zweier Ranges	bool
`lexicographical_compare(b,e,b2,e2,f)`	Lexikographischer Vergleich zweier anhand des Kriteriums `f`	bool

Kopieren und Verschieben

`copy(b,e,out)`	Kopiere Range `[b:e)` nach `[out:out+(e-b))`
`copy_backward(b,e,out)`	Kopiere Range `[b:e)` in umgekehrter Reihenfolge nach `out:out+(e-b)`
`copy_n(b,n,out)`	Kopiere alle Elemente im Range `[b:(b+n)]` nach `[out:out+(e-b))`
`copy_if(b,e,out,f)`	Kopiere alle Elemente im Range `[b:e)` für die `f(*p)` wahr ist nach `[out:out+(e-b))`
`move(b,e,out)`	Verschiebe Range `[b:e)` nach `[out:out+(e-b))` (C++11)
`move_backward(b,e,out)`	Verschiebe Range `[b:e)` in umgekehrter Reihenfolge nach `out:out+(e-b)` (C++11)
`swap_ranges(b,e,b2)`	Vertausche Elemente im Range `[b:e)` mit denen im Range `[b2:b2+(b-e))`

Setzen und Ersetzen von Werten

`fill(b,e,v)`	Setze alle Elemente im Range `[b:e)` gleich `v`
`fill_n(b,n,v)`	Setze die ersten `n` Elemente ab `b` gleich `v`
`generate(b,e,f)`	Setze alle Elemente im Range `[b:e)` gleich `f()`
`generate_n(b,n,f)`	Setze die ersten `n` Elemente ab `b` gleich `f()`
`replace(b,e,v,v2)`	Setze alle Elemente im Range `[b:e)` die gleich `v` gleich `v2`
`replace_if(b,e,f,v2)`	Setze alle Elemente im Range `[b:e)` für die `f(*p)` wahr ist gleich `v2`
`replace_copy(b,e,out,v,v2)`	Erstelle Kopie aller Elemente im Range `[b:e)` bei der Elemente, die gleich `v` sind gleich `v2` gesetzt werden. Liefert ierator auf Ende der Kopie zurück.
`replace_copy_if(b,e,out,f,v2)`	Erstelle Kopie aller Elemente im Range `[b:e)` bei der Elemente, für die `f(*p)` wahr ist gleich `v2` gesetzt werden. Liefert iterator auf Ende der Kopie zurück.

Verändern von Werten

`transform(b,e,out,f)`	Wende die Operation `*q=f(*p)` auf jedes Element `p` im Range `[b:e)` an und schreibe die Ergebnisse `q` in den Range `[out:out+(e-b))`
`transform(b,e,b2,out,f)`	Wende die Operation `*q=f(*p1,*p2)` auf alle Element `p` im Range `[b:e)` und `p1` im Range `[b2:b2+(e-b))` an und schreibe die Ergebnisse `q` in den Range `[out:out+(e-b))`

`transform` **vs.** `for_each`

```
#include<algorithm>
#include<iterator>
#include<list>
#include<iostream>

int myrand()
{
  return 1 + (int) (10.0 * (rand() / (RAND_MAX + 1.0)));
}

template<typename T>
```

```
void print(std::string prefix, const T& coll)
{
  std::cout << prefix;
  std::copy(coll.begin(), coll.end(),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
}

template<typename T>
T mult(const T &elem)
{
  return elem*10;
}

template<typename T>
void multAssign(T &elem)
{
  elem = elem*10;
}

int main()
{
  std::list<int> coll;
  std::generate_n(std::back_inserter(coll), 9, myrand);
  print("initial: ", coll);
  std::for_each(coll.begin(), coll.end(), multAssign<int>);
  print("for_each: ", coll);
  std::transform(coll.begin(), coll.end(), coll.begin(),
  mult<int>);
  print("transform: ", coll);
}
```

Output:

```
initial: 9 4 8 8 10 2 4 8 3
for_each: 90 40 80 80 100 20 40 80 30
transform: 900 400 800 800 1000 200 400 800 300
```

Löschalgorithmen

`remove(b,e,v)`	Entferne alle Elemente im Range `[b:e)`, die gleich `v` sind.
`remove_if(b,e,f)`	Entferne alle Elemente im Range `[b:e)`, für die `f(*p)` wahr ist.
`remove_copy(b,e,out,v)`	Erzeuge Kopie mit Entfernen aller Elemente im Range `[b:e)`, die gleich `v` sind.
`remove_copy_if(b,e,f)`	Erzeuge Kopie mit Entfernen alle Elemente im Range `[b:e)`, für die `f(*p)` wahr ist.
`unique(b,e)`	Entferne alle aufeinanderfolgende Duplikate
`unique(b,e,f)`	Entferne alle aufeinanderfolgende Elemente für die `f(*p,*(p+1))` wahr ist
`unique_copy(b,e,out)`	Erzeuge Duplikat-freie Kopie
`unique_copy(b,e,out,f)`	Erzeuge Duplikat-freie Kopie

- Elemente werden mit den nachfolgenden Elementen überschrieben, die nicht entfernt werden.

- Die Funktionen liefern einen Iterator zurück, der auf das Ende des Bereichs zeigt in dem nicht entfernte Elemente stehen. Damit können diese dann auch physikalisch entfernt werden.

Beispiel für Löschen

```
#include<list>
#include<algorithm>
#include<iostream>
#include<iterator>

template<typename T>
void print(T coll)
{
  typedef typename T::value_type value_type;
  std::copy(coll.begin(), coll.end(),
            std::ostream_iterator<value_type>(std::cout, " "));
  std::cout << std::endl;
}

int main()
{
  std::list<int> coll;
  for(int i=0; i<6; ++i)
  {
    coll.push_front(i);
    coll.push_back(i);
  }

  std::cout << "pre:     ";
  print(coll);
  std::list<int>::iterator newEnd = remove(coll.begin(),
                                           coll.end(), 3);
  std::cout << "post:    ";
  print(coll);
  coll.erase(newEnd, coll.end());
  std::cout << "removed: ";
  print(coll);
}
```

Output of sample program:

```
pre:     5 4 3 2 1 0 0 1 2 3 4 5
post:    5 4 2 1 0 0 1 2 4 5 4 5
removed: 5 4 2 1 0 0 1 2 4 5
```

Vertauschende Algorithmen

`reverse(b,e)`	Kehre Reihenfolge der Elemente im Range `[b:e)` um
`reverse_copy(b,e,out)`	Erstelle Kopie derElemente im Range `[b:e)` mit umgekehrter Reihenfolge
`rotate(b,m,e)`	Verschiebe alle Elemente zyklisch um `m` Elemente nach links
`rotate_copy(b,m,e,out)`	Erstelle Kopie bei der alle Elemente zyklisch um `m` Elemente nach links verschoben sind
`random_shuffle(b,e)`	Bringt alle Elemente in zufällige Reihenfolge
`random_shuffle(b,e,f)`	Bringt alle Elemente in zufällige Reihenfolge mit Zufallsgenerator `f`
`partition(b,e,f)`	Bringe alle Elemente für die `f(*p)` wahr ist nach vorne.
`stable_partition(b,e,f)`	Wie `partition`, aber erhalte Reihenfolge innerhalb der Partitionen

Verändernde Algorithmen und Assoziative Container

- Mit Iteratoren von assoziativen Containers lassen sich keine Zuweisungen machen, da der unveränderbare key Teil des `value_type` ist.
- Sie können daher nicht als Ziel eines verändernden Algorithmus verwendet werden.
- Ihre Verwendung führt zu einen Compilerfehler.
- Statt der Löschalgorithmen kann die Containermethode `erase` verwendet werden.
- Ergebnisse können mit Hilfe eines Insert Iterator Adapter in solchen Containern gespeichert werden (s.u.).

Algorithmen versus Containermethoden

- Während die STL Algorithmen sich allgemein auf beliebige Container anwenden lassen, haben sie oft nicht die optimale Komplexität für einen bestimmten Container.
- Wenn es auf Geschwindigkeit ankommt sollten lieber Containermethoden verwendet werden.
- Um z.B. alle Elemente mit dem Wert 4 aus einer `list` zu entfernen ist es besser `coll.remove(4)` aufzurufen als

```
coll.erase(remove(coll.begin(),coll.end(), 4)), coll.end);
```

Sortieralgorithmen

`sort(b,e)`	Sortiere alle Elemente im Range `[b:e)` (basiert auf Quicksort)
`stable_sort(b,e)`	Sortiere alle Elemente im Range `[b:e)` unter Beibehaltung der Ordnung gleicher Elemente (basiert auf Mergesort)
`partial_sort(b,m,e)`	Sortiere bis die ersten `m` Elemente die richtige Reihenfolge haben (basiert auf heapsort)
`partial_sort_copy(b,e,b2,e2)`	Sortiere bis die ersten `e2-b2` Elemente die richtige Reihenfolge haben (basiert auf heapsort)

Alle diese Algorithmen gibt es auch als Version mit Vergleichsoperator `f`.

Heaps

`make_heap(b,e)`	Konvertiere die Elemente im Range `[b:e)` in einen Heap
`push_heap(b,e)`	Füge Element `*(e-1)` zu Heap `[b:e-1)` hinzu, so dass anschließend `[b:e)` ein Heap ist
`pop_heap(b,e)`	Entferne Element `*(e-1)` vom Heap, so dass anschließend `[b:e-1)` ein Heap ist
`sort_heap(b,e)`	Sortiere den Heap (ist anschließend kein Heap mehr)

Alle diese Algorithmen gibt es auch als Version mit Vergleichsoperator `f`.

Suchen in vorsortierten Bereiche

`binary_search(b,e,v)`	Enthält der Range `[b:e)` das Element `v`?	bool
`lower_bound(b,e,v)`	Erstes Element größer oder gleich dem Wert `v` im Range `[b:e)` oder `e`	iterator
`upper_bound(b,e,v)`	Erstes Element größer dem Wert `v` im Range `[b:e)` oder `e`	iterator
`equal_range(b,e,v)`	Iteratoren auf den Bereich mit Werten gleich `v` (oder zweimal `e`)	`pair<iterator>`

Alle diese Algorithmen gibt es auch als Version mit Vergleichsoperator `f`.

Mergen vorsortierter Bereiche

`merge(b,e,b2,e2,out)`	Merges two ranges
`merge(b,e,b2,e2,out,f)`	Merges two ranges
`inplace_merge(b,m,e)`	Merges two consecutive sorted ranges `[b:m)` and `[m:e)` into `[b:e)`
`inplace_merge(b,m,e,f)`	Merges two consecutive sorted ranges `[b:m)` and `[m:e)` into `[b:e)`

Algorithmen für vorsortierte Bereiche: Sets

`includes(b,e,b2,e2)`	Sind alle Elemente aus `[b:e)` auch in `[b2:e2)` enthalten?	bool
`set_union(b,e,b2,e2,out)`	Liefert die sortierte Vereinigungsmenge der Ranges `[b:e)` und `[b2:e2)`	Iterator auf letztes Element
`set_intersection(b,e,b2,e2,out)`	Liefert die sortierte Schnittmenge der Ranges `[b:e)` und `[b2:e2)`	Iterator auf letztes Element
`set_difference(b,e,b2,e2,out)`	Liefert die sortierte Menge von Werten die in `[b:e)` sind, aber nicht in `[b2:e2)`	Iterator auf letztes Element
`set_symmetric_difference(b,e,b2,e2,out)`	Liefert die sortierte Menge von Werten die in `[b:e)` oder in `[b2:e2)` aber nicht in beiden	Iterator auf letztes Element

Alle diese Algorithmen gibt es auch als Version mit Vergleichsoperator `f`.

Numerische Algorithmen

`accumulate(b,e,i)`	Verknüpft alle Elemente im Range `[b:e)` mit dem Operator `plus`, verwendet Initialwert `i`
`accumulate(b,e,i,f)`	Verknüpft alle Elemente im Range `[b:e)` mit dem binären Operator `f`, verwendet Initialwert `i`
`inner_product(b,e,b2,i)`	Verknüpft die Werte der Ranges `[b:e)` und `[b2:b2+(e-b))` zu einem Skalarprodukt, verwendet Initialwert `i`
`inner_product(b,e,b2,i,f,f2)`	Verknüpft die Werte `(*p)` und `(*q)` aus den Ranges `[b:e)` und `[b2:b2+(e-b))` mit dem Operator `f` und die Ergebnisse mit dem Operator `f2`, verwendet Initialwert `i`
`adjacent_difference(b,e,out)`	Erstes Element von `out` ist `*b`, die nächsten Elemente sind jeweils `*p-*(p-1)`
`adjacent_difference(b,e,out,f)`	Erstes Element von `out` ist `*b`, die nächsten Elemente sind jeweils `f(*p,*(p-1))`
`partial_sum(b,e,out)`	Erstes Element `q` von `out` ist `*b`, die nächsten Elemente `q` von `out` sind `*(q-1)+p`
`partial_sum(b,e,out)`	Erstes Element `q` von `out` ist `*b`, die nächsten Elemente `q` von `out` sind `f(*(q-1),p)`
`iota(b,e,v))`	Weise jedem Element im Range `[b:e)` den Wert `++v` zu

13.4 Iterator Adapter

Iterator Adapter sind iteratorähnliche Klassen mit spezieller Funktionalität.

- `Insert iterator adapter`
- `Stream iterators adapter`

`Insert Iterator Adapter`

- sind `Output Iterator`, bei denen die zugewiesenen Objekte in einem angegebenen Container gespeichert werden.
- Sie dienen unter anderem zum Speichern von Ergebnissen, die STL Algorithmen liefern , die Operationen mit jedem Element eines Containers durchführen.
- Die STL definiert drei verschiedene `Insert Iterator Adapter`:

Klasse	Aufgerufene Fu des Containers
`back_insert_iterator`	`push_back(value)`
`front_insert_iterator`	`push_front(value`
`insert_iterator`	`insert(pos,value`

Beispiel Insert Iterator Adapter

```
#include<iostream>

template<class T>
void print(const T &container)
{
    for(typename T::const_iterator i=container.begin();
                                   i!=container.end(); ++i)
        std::cout << *i;
     std::cout << std::endl;
}

template<class Iterator>
void push_back_a_to_z(Iterator i)
{

    for(char c='a'; c <='z'; ++c, ++i)
        *i=c;
}

#include"insert.hh"
#include<list>
#include<set>
#include<iterator>

int main(int argc, char** argv)
{
    typedef std::list<char> clist;
    clist coll;

    std::back_insert_iterator<clist> bins(coll);
    push_back_a_to_z(bins);
    push_back_a_to_z(std::front_inserter(coll));
    print(coll);

    std::string s="A-Z␣Sequence␣is:␣␣.";
    push_back_a_to_z(std::inserter(s, s.begin()+18));
    print(s);

    std::set<char> sc;
    push_back_a_to_z(std::inserter(sc, sc.begin()));
    print(sc);
}
```

```
zyxwvutsrqponmlkjihgfedcbaabcdefghijklmnopqrstuvwxyz
A-Z Sequence is:  abcdefghijklmnopqrstuvwxyz.
abcdefghijklmnopqrstuvwxyz
```

Stream Iterator Adapter

Ein `stream iterator` erlaubt es Werte auf einen Stream zu schreiben oder von ihm zu lesen. Es handelt sich also entweder um einen `output operator` oder einen `input operator`.

```
namespace std{
template<typename T, typename charT=char, typename
traits=char_traits<charT> >
```

```
class istream_iterator;

template<typename T, typename charT=char,
typename traits=char_traits<charT>,
typename Distance = ptrdiff_t>
class ostream_iterator;
}
```

- Mit einem `istream_iterator` lassen sich Werte eines bestimmten Typs von dem Stream lesen.
- Mit einem `ostream_iterator` lassen sich Werte eines bestimmten Typs auf einen Stream schreiben.

`Stream Iterator Adapter` Funktionalität

- Ein `ostream_iterator` ist in seiner Funktionalität sehr ähnlich einem `Insert Iterator Adaptor` nur dass die Werte auf einen Stream statt in einen Container geschrieben werden.
- Ein `istream_iterator` hat die Funktionalität eines `Input Operator`.
- Zwei `istream_iterator` sind gleich, wenn sie beide auf das Ende eines Streams zeigen oder wenn sie den gleichen Stream verwenden.

`Stream Iterator Adapter` Beispiel

```
#include<iostream>
#include<iterator>
int main()
{
  std::istream_iterator<int> intReader(std::cin);
  std::istream_iterator<int> intReaderEOF;
  while(intReader != intReaderEOF)
  {
    std::cout<<*intReader<<"␣"<<*intReader<<std::endl;
    ++intReader;
  }
}
```

Die Eingabe `1 2 3 4 f 5` liefert die Ausgabe

```
1 1
2 2
3 3
4 4
```

Die Eingabe von `f` beendet das Programm, da sie zu einem Lesefehler auf dem Stream führt und damit gleichwertig mit `end-of-stream` ist.

```
#include<iostream>
#include<iterator>
int main()
{
  using std::string;
```

```
  std::istream_iterator<string> inPos(std::cin);
  std::ostream_iterator<string> outPos(std::cout, "␣");
  while(inPos != std::istream_iterator<string>())
  {
    std::advance(inPos, 2);
    if(inPos != std::istream_iterator<string>())
      *outPos = *inPos++;
  }
}
```

Schreibt jedes dritte Wort der Eingabe auf die Ausgabe. Die Eingabe

```
No one objects if you are doing a good programming job
for someone whom you respect
```

führt zur Ausgabe

```
objects are good for you
```

13.5 STL Funktoren

Funktoren

Funktoren sind ein zentrales Element von STL Algorithmen, sie werden für Vergleichs- und Suchfunktionen ebenso benötigt wie zur Manipulation von Containerelementen. Vorteil von Funktoren

- Funktoren sind "smart functions". Sie können
  - zusätzliche Funktionalität zum `operator()` haben.
  - einen inneren Zustand haben.
  - vorinitialisiert sein.
- Jeder Funktor hat seinen eigenen Typ.
  - `void less(int,int)` und `void greater(int,int)` haben den gleichen Typ (den eines Funktionspointers auf ein Funktion mit zwei `int` Argumenten).
  - Der Typ der Funktoren `less<int>` und `greater<int>` ist verschieden.
  - Deshalb ist auch der Typ von `set<int,less<int> >` und `set<int,greater<int> >` verschieden (was z.B. in Bezug auf Copy-Konstruktor und Zweisungsoperator sehr sinnvoll ist).
- Funktoren sind oft schneller als normale Funktionen.

Generator

- ist ein Funktor ohne Argumente, der Werte eines bestimmten Typs zurückgibt.
- Assoziierter Typ: `result_type`.
- Zwei nacheinanderfolgende Aufrufe können unterschiedliche Resultate liefern.
- Ein `Generator` kann sich auf einen lokalen Zustand beziehen, I/O Operationen durchführen ....
- Ihr innerer Zustand kann sich beim Aufruf ändern (z.B. bei einem Zufallsgenerator)

Beispiel Generator

```
#include<iostream>
#include<list>
#include<iterator>

class IntSequence{
public:
  typedef int return_type;

  IntSequence(int initial)
    : value(initial){}

  return_type operator()(){
    return value++;
  }
private:
  return_type value;
};

int main(){
  std::list<int> coll;

  std::generate_n(std::back_inserter(coll), 9,
                  IntSequence(1));
  std::copy(coll.begin(), coll.end(),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout<<std::endl;
  std::generate(++coll.begin(), --coll.end(), IntSequence(42));
  std::copy(coll.begin(), coll.end(), std::ostream_iterator<int>(std::cout, "
      "));
  std::cout<<std::endl;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
1 42 43 44 45 46 47 48 9
```

`UnaryFunction`

- Ein Funktor mit nur einem Argument.
- Assoziierte Typen: `argument_type` und `result_type`.
- Die STL stellt eine Basisklasse zur Verfügung, die es ermöglicht anschließend auch Funktoradapter zu nutzen (s.u.):

```
namespace std{
   template<typename Arg, typename Res>
   struct unary_function;
 }
```

`BinaryFunction`

- Ein Funktor mit zwei Argumenten.

- Assoziierte Typen: `first_argument_type`, `second_argument_type` und `result_type`.
- Die STL-Basisklasse ist:

```
namespace std{
  template<typename Arg1, typename Arg2, typename Res>
  struct binary_function;
}
```

`Predicate`

- Ein `Predicate` ist ein Funktor der einen `bool` zurückliefert `bool`.
- Das Verhalten eines `Predicate` sollte nicht davon abhängen ob/wie oft es kopiert/aufgerufen wird!

Beispiel Prädikat

```
#include<string>
class Person
{
public:
  Person(const std::string& first, const std::string& last);

  std::string firstname() const;

  std::string lastname() const;

private:
  std::string firstname_, lastname_;
};

struct PersonSortCriterion
{
  bool operator()(const Person& p1, const Person& p2){
    return p1.lastname()<p2.lastname()||
      (!(p2.lastname()<p1.lastname()) && p1.firstname()<p2.firstname());
  }
};

#include<set>

int main()
{
  typedef std::set<Person,PersonSortCriterion> PersonSet;

  PersonSet coll;

  coll.insert(Person("Max", "Muster"));
  coll.insert(Person("Eva", "Muster"));
}
```

Vordefinierte Funktoren

Unäre Funktoren	
`negate<type>`	`- param`
Binäre Funktoren	
`plus<type>`	`param1 + param2`
`minus<type>`	`param1 - param2`
`multiplies<type>`	`param1 * param2`
`divides<type>`	`param1 / param2`
`modulus<type>`	`param1 % param2`
Prädikate	
`equal_to<type>`	`param1 == param2`
`not_equal_to<type>`	`param1 != param2`
`less<type>`	`param1 < param2`
`greater<type>`	`param1 > param2`
`greater_equal<type>`	`param1 <= param2`
`logical_not<type>`	`!param`
`logical_and<type>`	`param1 && param2`
`logical_or<type>`	`param1 || param2`

Funktoradapter

Funktoradapter können verwendet werden um Funktoren anzupassen. So kann aus einem binären Funktor ein unärer gemacht werden, indem das eine Argument auf einen konstanten Wert gesetzt wird, oder ein Prädikat kann negiert werden.

Ausdruck	Effekt
`bind1st(op, val)`	`op(val, param)`
`bind2nd(op, val)`	`op(param,val)`
`not1(op)`	`!op(param)`
`not2(op)`	`!op(param1,param2)`

Beispiel:

```
#include<vector>
#include<iostream>
#include<iterator>
#include<algorithm>

int main()
{
  std::vector<int> coll(7,3);
  std::transform(coll.begin(), coll.end(),
      coll.begin(),std::bind2nd(std::plus<int>(),3));
}
```

Wenn ein Container Objekte einer Klasse oder Pointer darauf enthält, ist es möglich für jedes Element eine Methode aufrufen zu lassen.

Ausdruck	Effekt
`mem_fun_ref(op)`	Ruft die Methode `op()` der Klasse für jedes Objekt auf.
`mem_fun(op)`	Ruft die Methode `op()` der Klasse für jeden Pointer auf.

Beispiel:

```
std::vector<Personen> coll(5,Personen("Max","Mustermann"));
for_each(coll.begin(), coll.end(), mem_fun_ref(&Person::print);
std::vector<Personen *> pointColl;
for_each(pointColl.begin(), pointColl.end(), mem_fun(&Person::print));
```

Existiert eine unnäre oder binäre C-Funktion `op`, so lässt auch diese sich in einen Funktor verwandeln:

Ausdruck	Effekt
`ptr_fun(op)`	`*op(param)` oder `*op(param1, param2)`

Beispiel:

```
bool check(int elem);
std::vector<int> coll(7,3);
std::vector<int>::iterator pos = find_if(coll.begin(), coll.end(),
    not1(ptr_fun(check)));
```

14 Traits

Problem

- Die vielfältige Konfigurierbarkeit von Algorithmen mit Templates verführt dazu mehr und mehr Templateparameter einzuführen.
- Es gibt unterschiedliche Arten von Templateparametern:
  - Unverzichtbare Templateparameter.
  - Templateparameter die sich mit Hilfe von anderen Templateparametern bestimmen lassen.
  - Templateparameter die Defaultwerte haben und nur in sehr seltenen Fällen angegeben werden müssen.

Definition: Traits

- Laut Oxford Dictionary:

  Trait a distinctive feature characterising a thing

- Eine Definition aus dem Bereich der C++ Programmierung[2]:

  Traits represent natural additional properties of a template parameter.

Beispiel

Summe über eine Sequenz
Die Summe über eine Reihe von Werten die in einem C-Array gespeichert sind, lässt sich wie folgt schreiben:

[2] D. Vandevoorde, N. M. Josuttis: C++ Templates - The Complete Guide, Addison Wesley 2003

```
template<typename T>
T accum(T const *begin, T const *end)
{
  T result=T();
  for(; begin!=end; ++begin)
    result += *begin;
  return result;
}
```

Problem

- Hier tritt ein Problem auf, wenn der Wertebereich der zu summierenden Elemente nicht groß genug ist um auch die Summe ohne Überlauf speichern zu können.

```
#include<iostream>
#include"accum1.h"

int main()
{
    char name[] = "templates";
    int length = sizeof(name)-1;
    std::cout << accum(&name[0], &name[length])/length << std::endl;
}
```

- Wenn `accum` verwendet wird um die Summe der `char`-Variablen im Wort “templates” zu berechnen, dann erhält man -5 (was kein ASCII code ist).
- Deshalb brauchen wir eine Möglichkeit den richtigen Rückgabetyp der Funktion `accum` anzugeben.

Die Einführung eines zusätzlichen Templateparameters für diesen Spezialfall führt zu schwer lesbarem Code:

```
template<class V, class T>
V accum(T const *begin, T const *end)
{
  V result=V();
  for(; begin!=end; ++begin)
    result += *begin;
  return result;
}

int main()
{
    char name[] = "templates";
    int length = sizeof(name)-1;
    std::cout << accum<int>(&name[0], &name[length])/length << std::endl;
}
```

14.1 Type Traits

Listing 1: Type Traits Beispiel

```
template<typename T>
struct AccumTraits {
  typedef T AccumType;
};

template<>
struct AccumTraits<char> {
  typedef int AccumType;
};

template<>
struct AccumTraits<short> {
  typedef int AccumType;
};

template<>
struct AccumTraits<int> {
  typedef long AccumType;
};

template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const *begin, T const *end)
{
  // short cut for the return type
  typedef typename AccumTraits<T>::AccumType AccumType;

  AccumType result=AccumType(); // intialize to zero

  for(; begin!=end; ++begin)
    result += *begin;

  return result;
}
```

14.2 Value Traits

- Bisher verlassen wir uns darauf, dass der Default-Konstruktor unseres Rückgabetyps die Variable mit Null initialisiert:

```
AccumType result=AccumType();
for(; begin!=end; ++begin)
  result += *begin;
return result;
```

- Leider gibt es keinerlei Garantie dafür, dass dies auch der Fall ist.
- Eine Lösung hierfür ist das Hinzufügen von sogenannten Value Traits zu der Traitsklasse.

Listing 2: Value Traits Beispiel

```
template<typename T>
struct AccumTraits{
```

```
  typedef T AccumType;
  static AccumType zero(){
    return AccumType();
  }

};

template<>
struct AccumTraits<char>{
  typedef int AccumType;
  static AccumType zero(){
    return 0;
  }
};

template<>
struct AccumTraits<short>{
  typedef int AccumType;
  static AccumType zero(){
    return 0;
  }
};

template<>
struct AccumTraits<int>{
  typedef long AccumType;
  static AccumType zero(){
    return 0;
  }
};

template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const *begin, T const *end)
{
  // short cut for the return type
  typedef typename AccumTraits<T>::AccumType AccumType;

  // intialize to zero
  AccumType result=AccumTraits<T>::zero();

  for(; begin!=end; ++begin)
    result += *begin;

  return result;
}
```

14.3 Promotion Traits

Type Promotion

- Nehmen wir an es sollen zwei Vektoren mit Objekten eines Zahlentyps addiert werden:

```
template<typename T>
std::vector<T> operator+(const std::vector<T> &a, const std::vector<T>
    &b);
```

- Was sollte der Rückgabetyp sein, wenn die Typen der Variablen in den beiden Vektoren unterschiedlich sind?

```
template<typename T1, typename T2>
std::vector<???> operator+(const std::vector<T1> &a, const
    std::vector<T2> &b);
```

z.B.

```
std::vector<float> a;
std::vector <complex<float> > b;
std::vector<???> c = a+b;
```

Promotion Traits

- Die Auswahl des Rückgabetyps muss passend zu zwei verschiedenen Typen gewählt werden.
- Auch dies kann mit Hilfe von Traitsklassen bewerkstelligt werden:

```
template<typename T1, typename T2>
std::vector<typename Promotion<T1, T2>::promoted_type>
            operator+ (const std::vector<T1> &,
                       const std::vector<T2> &);
```

- Die Promotion Traits werden wieder mit Hilfe von Templatespezialisierung definiert:

```
template<typename T1, typename T2>
struct Promotion {};
```

- Es ist einfach eine teilweise Spezialisierung für zwei identische Typen zu machen:

```
template<typename T>
struct Promotion<T,T> {
  public:
    typedef T promoted_type;
};
```

- Andere Promotion Traits werden mit voller Templatespezialisierung definiert:

```
template<>
struct Promotion<float, complex<float> > {
  public:
    typedef complex<float> promoted_type;
};

template<>
struct Promotion<complex<float>, float> {
  public:
    typedef complex<float> promoted_type;
};
```

- Da Promotion Traits oft für viele verschiedene Kombinationen von Variablentypen definiert werden müssen kann folgendes Makro hilfreich sein:

```
#define DECLARE_PROMOTE(A,B,C) \
  template<> struct Promotion<A,B> { \
    typedef C promoted_type; \
  }; \
  template<> struct Promotion<B,A> { \
    typedef C promoted_type; \
  };

DECLARE_PROMOTE(int, char, int);
DECLARE_PROMOTE(double, float, double);
DECLARE_PROMOTE(complex<float>, float, complex<float>);
// and so on...

#undef DECLARE_PROMOTE
```

- Die Funktion zur Addition von zwei Vektoren lässt sich dann wie folgt schreiben:

```
template<typename T1, typename T2>
std::vector<typename Promotion<T1,T2>::promoted_type>
operator+(const std::vector<T1>& a, const std::vector<T2>& b)
{
  typedef typename Promotion<T1,T2>::promoted_type T3;
  typedef typename std::vector<T3>::iterator Iterc;
  typedef typename std::vector<T1>::const_iterator Iter1;
  typedef typename std::vector<T2>::const_iterator Iter2;

  std::vector<T3> c(a.size());
  Iterc ic=c.begin();
  Iter2 i2=b.begin();
  for(Iter1 i1=a.begin();i1 != a.end(); ++i1, ++i2, ++ic)
    *ic = *i1 + *i2;
  return c;
}
```

- Oder in der ganz allgemeinen Form mit generischem Container:

```
template<typename T1, typename T2,
         template<typename U,typename =std::allocator<U> > class Cont>
Cont<typename Promotion<T1,T2>::promoted_type>
operator+(const Cont<T1> &a, const Cont<T2> &b)
{
  typedef typename Promotion<T1,T2>::promoted_type T3;
  typedef typename Cont<T3>::iterator Iterc;
  typedef typename Cont<T1>::const_iterator Iter1;
  typedef typename Cont<T2>::const_iterator Iter2;

  Cont<T3> c(a.size());
  Iterc ic=c.begin();
  Iter2 i2=b.begin();
  for(Iter1 i1=a.begin();i1 != a.end(); ++i1, ++i2, ++ic)
    *ic = *i1 + *i2;
  return c;
}

int main()
{
```

```
  std::vector<double> a(5,2);
  std::vector<float> b(5,3);
  a = a + b;
  for (size_t i=0;i<a.size();++i)
    std::cout << a[i] << std::endl;
  std::list<double> c;
  std::list<float> d;
  for (int i=0;i<5;++i)
  {
    c.push_back(i);
    d.push_back(i);
  }
  c = d + c;
  for (std::list<double>::iterator i=c.begin();i!=c.end();++i)
    std::cout << *i << std::endl;

}
```

14.4 Iterator Traits

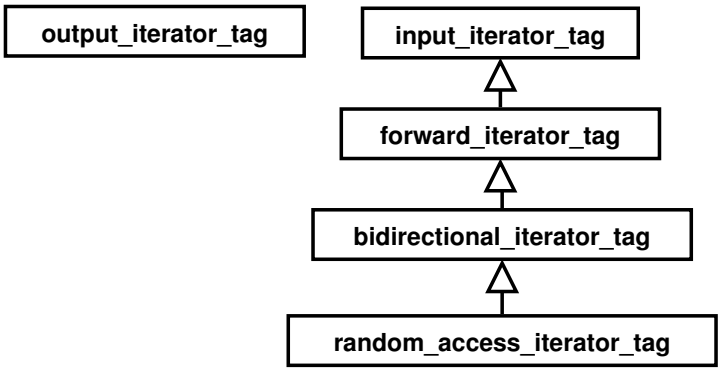
- Auch STL Iteratoren exportieren viele ihrer Eigenschaften über Traits:
- Laut C++ Standard werden folgende Informationen über Iteratoren bereitgestellt:

```
namespace std{
  template<class T>
  struct iterator_traits{
    typedef typename T::value_type value_type;
    typedef typename T::difference_type
      difference_type;
    typedef typename T::iterator_category
      iterator_category;
    typedef typename T::pointer pointer;
    typedef typename T::reference reference;
  };
}
```

- Es existiert auch eine Spezialisierung für Pointer. Pointer sind deshalb eine spezielle Form von Iterator.

Iteratorkategorien

- Die Kategorie eines Iterators kann über ein Tag in den Iteratortraits abgefragt werden.



Beispiel: Verwendung der Iteratorkategorie in generischem Code

- Vorrücken des Iterators um eine bestimmte Anzahl Elemente.
- Falls vorhanden werden optimierte Iteratorfunktionen verwendet.

Listing 3: Using Iterator Category in Generic Code

```
#include<iterator>

template<class Iter, class IterTag>
struct AdvanceHelper{
  static void advance(Iter& pos, std::size_t dist){
    for(std::size_t i=0; i<dist; ++dist)
      ++pos;
  }
};

template<class Iter>
struct AdvanceHelper<Iter,
                     std::random_access_iterator_tag>{
  static void advance(Iter& pos, std::size_t dist){
    pos+=dist;
  }
};

template<class Iter>
void advance(Iter& pos, std::size_t dist)
{
  AdvanceHelper<Iter,
    typename std::iterator_traits<Iter>::iterator_category>
  ::advance(pos, dist);
}
```

14.5 Beispiel: Schreiben von HDF5-Files

```
#include <hdf5.h>

template<typename T>
struct H5ValueType
{
```

```
  static hid_t fileType()
  {
        return 0;
  };
  static hid_t memType()
  {
        return 0;
  };
};

template<>
struct H5ValueType<float>
{
  static hid_t fileType()
  {
    return H5T_IEEE_F32BE;
  }
  static hid_t memType()
  {
    return H5T_NATIVE_FLOAT;
  }
};

template<>
struct H5ValueType<double>
{
  static hid_t fileType()
  {
    return H5T_IEEE_F64BE;
  }
  static hid_t memType()
  {
    return H5T_NATIVE_DOUBLE;
  }
};


template<typename T>
void SaveSolutionHDF5(std::string filename, std::string fieldname, const
    std::vector<T> &outValues)
{
    ...
    hid_t dataset_id = H5Dcreate1(file_id, fieldname.c_str(),
        H5ValueType<T>::fileType(), dataspace_id, H5P_DEFAULT);

    status = H5Dwrite(dataset_id, H5ValueType<T>::memType(), memspace_id,
        dataspace_id, xferPropList, &outValues[0]);
    ...
}
```

15 Policies

Definition: Policies

- Aus dem Oxford Dictionary:

  Policy any course of action adopted as advantageous or expedient

- Eine Definition aus dem Bereich der C++ Programmierung[3]:

  Policies represent configurable behaviour for generic functions and types.

Erweiterung auf andere Arten von Akkumulation

- Wir haben bei der Akkumulation der Sequenz eine Summe gebildet.
- Dies ist nicht die einzige Möglichkeit. Wir könnten die Werte auch multiplizieren, aneinanderhängen oder das Maximum suchen.
- Beobachtung: Der Code von `accum` bleibt bis auf folgende Zeile unverändert:

  ```
  result += *begin
  ```

- Wir nennen diese Zeile die Policy unseres Algorithmus.
- Wir können diese Zeile in eine sogenannte Policyklasse packen und diese als Templateparameter an unseren Algorithmus übergeben.
- Eine Policyklasse ist eine Klasse die eine Schnittstelle bereitstellt um eine oder mehrere Policies in einem Algorithmus anzuwenden.

Listing 4: Akkumulationsalgorithmus mit Policyklasse

```
template<typename T>
struct AccumTraits{
  typedef T AccumType;

  static AccumType zero(){
    return AccumType();
  }

  static AccumType one(){
    return ++AccumType();
  }

};

template<>
struct AccumTraits<char>{
  typedef int AccumType;

   static AccumType zero(){
     return 0;
   }

  static AccumType one(){
    return 1;
  }

};
```

[3]D. Vandevoorde, N. M. Josuttis: C++ Templates - The Complete Guide, Addison Wesley 2003

```
template<>
struct AccumTraits<short>{
  typedef int AccumType;

   static AccumType zero(){
     return 0;
   }

  static AccumType one(){
    return 1;
  }
};

template<>
struct AccumTraits<int>{
  typedef long AccumType;

   static AccumType zero(){
     return 0;
   }

  static AccumType one(){
    return 1;
  }
};

class SumPolicy{
public:
  template<typename T>
  static T init(){
    return AccumTraits<T>::zero();
  }

  template<typename T1, typename T2>
  static void accumulate(T1& total, T2 const& value){
    total +=value;
  }
};

class MultPolicy{
public:
  template<typename T>
  static T init(){
    return AccumTraits<T>::one();
  }

  template<typename T1, typename T2>
  static void accumulate(T1& total, T2 const& value){
    total *=value;
  }
};



template<typename T, class Policy>
typename AccumTraits<T>::AccumType
accum(T const *begin, T const *end){
```

```
  // short cut for the return type
  typedef typename AccumTraits<T>::AccumType AccumType;

  // intialize depending on policy (mult: 1, sum: 0)
  AccumType result=Policy::template init<T>();

  for(; begin!=end; ++begin)
    Policy::accumulate(result, *begin);

  return result;
}
```

Kombination von Policies

- Typischerweise wird eine hochkonfigurierbare Klasse mehrere verschiedene Policies kombinieren.
- Nehmen wir z.B. einen `SmartPointer`:

  Checking: `CheckingPolicy<T>` muss eine `check` methode bereitstellen, die gegebenenfalls überprüft, ob der Pointer gültig (i.d.R. ungleich Null) ist.

  Threading: Die Templateklasse `ThreadingModel<T>` muss einen Typen `Lock` definieren dessen Konstruktor eine Referenz `T &` als Argument bekommt.

  Storage: `StorageModel<T>` muss die Typen `pointer_type` und `reference_type` exportieren und die Methoden `setPointer`, `getPointer` und `getReference` definieren die den Pointer vom Typ `pointer_type` setzen, den Pointer zurückliefern oder eine Referenz vom Typ `reference_type` auf die Speicherstelle zurückliefern.

- Der Anwender kann dann das Verhalten der SmartPointerklasse durch Auswahl geeigneter Templateparameter beeinflussen:

```
typedef SmartPointer<Object,NoChecking,
                     SingleThreaded,DefaultStorage> ObjectPtr;
```

Checking Policies

```
#include<exception>

template<typename T>
struct NoChecking
{
  static void check(T)
  {}
};

template<typename T>
struct EnsureNotNull
{
  class NullPointerException : public std::exception
  {};

  static void check(const T ptr)
  {
```

```
    if(!ptr) throw NullPointerException();
  }
};
```

Default Threading Policy

```
template<typename T>
struct SingleThreaded
{
  struct Lock
  {
    Lock(const T &o)
    {}
  };
};
```

Default Storage Policy

```
template<typename T>
class DefaultStorage
{
public:
  typedef T * pointer_type;
  typedef T & reference_type;
  typedef const T * const_pointer_type;
  typedef const T & const_reference_type;
protected:
  reference_type getReference()
  {
    return *ptr_;
  }

  pointer_type getPointer()
  {
    return ptr_;
  }
  const_reference_type getReference() const
  {
    return *ptr_;
  }

  const_pointer_type getPointer() const
  {
    return ptr_;
  }

  void setPointer(pointer_type ptr)
  {
    ptr_=ptr;
  }

  DefaultStorage() : ptr_(0)
  {}

  void releasePointer()
  {
```

```
    if (ptr_ != 0)
      delete ptr_;
  }
private:
  pointer_type ptr_;
};
```

Smart Pointer

```
template<typename T,
         template<typename> class CheckingPolicy = EnsureNotNull,
         template<typename> class ThreadingModel = SingleThreaded,
         template<typename> class StorageModel = DefaultStorage>
class SmartPointer
  : public CheckingPolicy<typename StorageModel<T>
                          ::const_pointer_type>,
    public StorageModel<T>
{
public:
  typedef typename StorageModel<T>::pointer_type pointer_type;
  typedef typename StorageModel<T>::reference_type reference_type;
  typedef typename StorageModel<T>::const_pointer_type const_pointer_type;
  typedef typename StorageModel<T>::const_reference_type const_reference_type;

  SmartPointer(pointer_type ptr){
    this->setPointer(ptr);
  }

  ~SmartPointer(){
    StorageModel<T>::releasePointer();
  }

  pointer_type operator->(){
    typename ThreadingModel<SmartPointer>::Lock guard(*this);
    CheckingPolicy<pointer_type>::check(this->getPointer());
    return this->getPointer();
  }

  reference_type operator*(){
    typename ThreadingModel<SmartPointer>::Lock guard(*this);
    CheckingPolicy<pointer_type>::check(this->getPointer());
    return this->getReference();
  }

  const_pointer_type operator->() const{
    typename ThreadingModel<SmartPointer>::Lock guard(*this);
    CheckingPolicy<pointer_type>::check(this->getPointer());
    return this->getPointer();
  }

  const_reference_type operator*() const{
    typename ThreadingModel<SmartPointer>::Lock guard(*this);
    CheckingPolicy<pointer_type>::check(this->getPointer());
    return this->getReference();
  }
};
```

```
#endif
```

Kompatible und Inkompatible Policies

- Angenommen wir instantiieren zwei verschiedene `SmartPointer`:

  FastPointer: Ohne irgendwelche Überprüfungen

  SafePointer: Eine Variante die prüft ob der Pointer Null ist

- Sollte es möglich sein einem `SafePointer` einen `FastPointer` zuzuweisen, oder anders herum?
- Es liegt in unserer Hand ob wir explizite Konvertierungen erlauben.
- Der beste und skalierbarste Weg ist es `SmartPointer` Objekte *policy by policy* zu initialisieren und zu kopieren.

```
template<typename T,
         template<typename> class CheckingPolicy = EnsureNotNull,
         template<typename> class ThreadingModel = SingleThreaded,
         template<typename> class StorageModel = DefaultStorage>
class SmartPointer
  : public CheckingPolicy<typename StorageModel<T>
                          ::const_pointer_type>,
    public StorageModel<T>
{
public:
  template<typename T1,
           template<typename>  class CP1,
           template<typename>  class TM1,
           template<typename>  class SM1>
  SmartPointer(const SmartPointer<T1,CP1,TM1,SM1> &other) :
                       CheckingPolicy<T>(other),
                       StorageModel<T>(other)
  {}
```

- Angenommen wir wollen ein Objekt vom Typ `SmartPointer<ExtendedObject,NoChecking,...>` in ein Objekt vom Typ `SmartPointer<Object,NoChecking,...>` kopieren wobei `ExtendedObject` von `Object` abgeleitet ist,
- dann versucht der Compiler `Object *` mit einem `ExtendedObject *` zu initialisieren (was geht) und `NoChecking` mit `SmartPointer<ExtendedObject,NoChecking,...>` (was auch funktioniert da letztere Klasse von `NoChecking` abgeleitet ist).
- Weiteres Beispiel: Wir wollen `SmartPointer<Object,NoChecking,...>` nach `SmartPointer<Object, EnforceNotNull,...>` kopieren:
- Hier versucht der Compiler `SmartPointer<Object,NoChecking,...>` an den Konstruktor von `EnforceNotNull` zu übergeben.
- Dies funktioniert nur, wenn `EnforceNotNull` einen Copykonstruktor bereitstellt, der `NoChecking` als Argument akzeptiert.

Static Assertion

```
template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const *begin, T const *end)
{
  // short cut for the return type
  typedef typename AccumTraits<T>::AccumType AccumType;
  static_assert(sizeof(typename AccumTraits<T>::AccumType)>=sizeof(T),"The␣
      size␣of␣the␣accumulation␣type␣is␣smaller␣than␣the␣accumulated␣type");
  AccumType result=AccumType(); // intialize to zero
```

- Klassen die als Templateargumente eingesetzt werden, können verschiedene Eigenschaften haben.
- Sind für das Funktionieren des Templates bestimmte Voraussetzungen notwendig, können diese zur Übersetzungszeit durch `static_assert` überprüft werden.
- Das erste Argument von `static_assert` ist ein Ausdruck, dessen Ergebnis vom Typ `bool` ist und zur Übersetzungszeit feststeht.
- Das zweite Argument ist eine vom Compiler auszugebende Fehlermeldung.

Type Traits

```
#include <type_traits>

template <class T>
void swap( T& a, T& b)
{
    static_assert(std::is_copy_constructible<T>::value,
                  "Type␣needs␣to␣have␣a␣copy␣constructor␣for␣swap");
    T c = b;
    b = a;
    a = c;
}
```

- Mit den Type Traits stellt die Standardbibliothek Informationen über Typ-Eigenschaften zur Verfügung.
- Bei den eigentlichen Type Traits handelt es sich um Prädikate (also Funktionen die einen `bool` zurückliefern.

Beispiele für Type Traits

`is_void<T>`	ist `T` `void`?
`is_integral<T>`	ist `T` ganzzahlig?
`is_floating_point<T>`	ist `T` Fließkommazahl?
`is_array<T>`	ist `T` ein C-array?
`is_pointer<T>`	ist `T` ein Pointer (ohne Pointer of Methoden)?
`is_class<T>`	ist `T` eine Klasse (inkl. `struct`)?
`is_function<T>`	ist `T` eine Funktion?
`is_member_object_pointer<T>`	ist `T` ein Pointer auf ein nicht-statisches Attribut?
`is_reference<T>`	ist `T` eine Referenz?
`is_arithmetic<T>`	ist `T` eine Zahl?
`is_object<T>`	ist `T` ein Objekt?
`is_const<T>`	ist `T` konstant?
`is_abstract<T>`	hat `T` rein virtuelle Methoden?
`is_signed<T>`	ist `T` vorzeichen-behaftet?
`is_default_constructible<T>`	hat `T` einen argumentlosen Konstruktor?
`is_copy_constructible<T>`	hat `T` einen Copy-Konstruktor?

`is_assignable<T,X>`	kann `X` an `T` zugewiesen werden?
`is_same<T,X>`	sind `X` und `T` identisch?
`is_base_of<T,X>`	ist `X` von `T` abgeleitet?
`is_convertible<X,T>`	kann `X` implizit nach `T` konvertiert werden?

Typ Umwandlungen

`remove_const<T>`	liefert äquivalenten Typ zu `T` aber ohne `const`
`remove_volatile<T>`	liefert äquivalenten Typ zu `T` aber ohne `volatile`
`remove_cv<T>`	entfernt `const` und `volatile`
`add_const<T>`	fügt `const` zu Typ hinzu, wenn noch nicht vorhanden
`add_volatile<T>`	fügt `volatile` zu Typ hinzu, wenn noch nicht vorhanden
`add_volatile<T>`	fügt `const` und `volatile` zu Typ hinzu, wenn noch nicht vorhanden
`remove_reference<T>`	liefert Typ ohne Referenz wenn `T` Referenz, sonst einfach `T`
`make_signed<T>`	macht äquivalenten vorzeichenbehafteten Typ
`make_unsigned<T>`	macht äquivalenten vorzeichenlosen Typ

Automatische Typerkennung

```
#include<iostream>
#include<vector>
#include<type_traits>

int f() {
  return 1;
}

int main()
{
  auto var1 = 5678;    // var1 ist int
  auto var2 = 'x';     // var2 ist char
  auto var3 = f();     // var3 ist Rueckgabetyp von f() also int
  std::vector<double> x(5);
  typedef std::vector<double>::iterator vdIter;
  int i=0;
  for (vdIter y=x.begin();y!=x.end();++y,++i)
```

```
    *y = i*1.2;
  for (auto y=x.begin();y!=x.end();++y)
    std::cout << *y << std::endl;
}
```

- In C++11 kann für den Typ einer Variable `auto` eingesetzt werden, wenn sich der Typ aus der dahinter stehenden Initialisierung ableiten lässt.

```
int main()
{  // alle suffixe gehen auch als Grossbuchstaben
  auto var1 = 5.0;   // double
  auto var2 = 5.0f;  // float
  auto var3 = 5.0l;  // long double
  auto var4 = 5;     // int
  auto var5 = 5l;    // long
  auto var6 = 5u;    // unsigned int
  auto var7 = 5ul;   // unsigned long int
  auto var8 = 'a';   // char
  auto var9 = "a";   // const char *
  auto varA = new int;  // pointer to int
}
```

- Für die Typerkennung bei `auto` gelten die üblichen Regeln für C++ Literale.

Range For-Loops

```
#include<iostream>
#include<vector>

int main()
{
  std::vector<double> x(5);
  int i=0;
  for (auto &y : x)          // mit Referenz
  {
    y = i*1.2;
    ++i;
  }
  for (auto y : x)           // mit Kopie, laesst x unveraendert
  {
    y*=y;
    std::cout << y << std::endl;
  }
  for (const auto &y : x)    // mit konstanter Referenz
    std::cout << y << std::endl;
}
```

- `auto` lässt sich in `for`-loops verwenden um über ganz Container zu iterieren.
- Die Variable zeigt dabei standardmäßig auf eine Kopie des jeweiligen Containerelementes.
- Es lassen sich auch konstante Variablen oder Referenzen erzeugen.

DeclType

- Mit `decltype(expr)` lässt sich der Typ des Ergebnisses des Ausdrucks `expr` bestimmen.
- Im einfachsten Fall ist das der Typ einer Variable

```
int i;
decltype(i) j;  // macht zweite Integervariable
```

- Aber auch kompliziertere Ausdrücke sind möglich, z.B.

```
int main()
{
  std::vector<double> x(5,4.0);
  std::vector<float> y(5,2.0);
  decltype(x+y) z;  // macht passende Variable fuer die Summe
  z = x + y;
}
```

Suffix Return Syntax

```
template <class X, class Y>
auto add(const X &x, const Y &y) -> decltype(x + y)
{
  return x+y;
}
```

- Der Rückgabewert einer Funktion lässt sich mit `decltype` auch automatisch bestimmen.
- Manchmal möchte man dazu die Variablen in der Argumentliste verwenden. Allerdings sind diese noch nicht definiert, wenn der Rückgabetyp bestimmt werden muss.
- Es ist jedoch möglich die Angabe nach hinten zu verschieben.

Promotion-Type Example mit C++11

```
#include<vector>
#include<list>
#include<iostream>

template<typename T1, typename T2,
         template<typename U,typename =std::allocator<U> > class Cont>
auto
operator+(const Cont<T1> &a, const Cont<T2> &b)-> Cont<decltype(T1()+T2())>
{
  Cont<decltype(T1()+T2())> c(a.size());
  auto ic=c.begin();
  auto i2=b.begin();
  for(auto i1=a.begin();i1 != a.end(); ++i1, ++i2, ++ic)
    *ic = *i1 + *i2;
  return c;
}

int main()
{
  std::vector<double> a(5,2);
  std::vector<float> b(5,3);
  decltype(a+b) z;
  static_assert(std::is_same<decltype(z),std::vector<double>>(),"Type␣of␣z␣is␣not␣
      vector<double>");
  z = a + b;
  for (auto x : z)
    std::cout << x << std::endl;
  std::list<char> c;
  std::list<char> d;
  for (int i=0;i<5;++i)
  {
```

```
    c.push_back(i);
    d.push_back(i);
  }
  decltype(d+c) e;
static_assert(std::is_same<decltype(e),std::list<int>>(),"Type␣of␣e␣is␣not␣list<int>");
  e = d + c;
  for (auto x : e)
    std::cout << x << std::endl;

}
```

Lambda Ausdrücke

- In C++11 gibt es eine stark verkürzte Möglichkeit um (meist temporäre) Funktionen zu definieren, sogenannte Lambda Ausdrücke.
- Ein Lambda Ausdruck wird häufig an Stellen verwendet an denen Funktoren erwartet werden.
- Ein Lambda Ausdruck besteht aus einer capture list, einer Argumentliste und einem Funktionsrumpf.
- Manchmal kommen noch optional `mutable` und `noexcept` Angaben und ein Rückgabetyp hinzu.

Lambda Example

```
#include<iostream>
#include<vector>
#include<algorithm>

int main()
{
  std::vector<int> a(5,4);
  for_each(a.begin(),a.end(),[](int x){ std::cout << "␣" << x;});
  std::cout << std::endl;
}
```

- Braucht ein Lambda keinen Zugriff auf lokale Variablen der aufrufenden Umgebung beginnt es mit `[]`.
- Als nächstes folgt die Argumentliste wie bei einer normalen Funktion. Werden keine Argumente übergeben können die (leeren) Klammern entfallen.
- Anschließend kommt im einfachsten Fall direkt der Funktionsrumpf, wie üblich in geschweiften Klammern.

Lambda: Capture List

```
#include<iostream>
#include<vector>
#include<algorithm>

int main()
{
  std::vector<int> a(5,4);
  int i=1;
```

```
  for_each(a.begin(),a.end(),[&i](int &x){x=i*i;++i;});
  std::cout << i << std::endl; // i ist jetzt 6
  for(auto x : a)
    std::cout << "␣" << x;
  std::cout << std::endl;
}
```

- Ein Lambda kann direkten Zugriff auf Variablen der lokalen Umgebung bekommen. Dies erfolgt in der sogenannten Capture List, den eckigen Klammern am Beginn des Lambdas.
- Die Variablen können als Kopie oder als Referenz übergeben werden.
- Besteht die Capture List nur aus `[=]` , werden alle lokalen Variablen als Kopie übergeben.
- Besteht die Capture List nur aus `[&]` , werden alle lokalen Variablen als Referenz übergeben.
- Es ist auch möglich nach `&` oder `=` Variablen anzugeben, die von der generellen Regel abweichend als Kopie respektive Referenz übergeben werden sollen.
- Sollen nur einzelne Variablen als Kopie oder Referenz übergeben werden, dann gibt man diese einfach in der Capture List an.
- Referenzen werden immer durch ein dem Variablennamen vorangestelltes `&` gekennzeichnet.
- Bei einem Lambda in einem Objekt erhält man Zugriff auf die Attribute des Objekts durch Aufnahme von `this` in die Capture List. Der Zugriff erfolgt immer als Referenz, dafür ist kein `&this` nötig. `this` darf nicht in einer Capture List nach einem generellen `=` stehen.
- Globale Variablen sind immer zugreifbar und müssen nicht in der Capture List aufgeführt werden.
- Bei der Übergabe von Referenzen ist darauf zu achten, dass die Lebensdauer des Lambdas die der aufrufenden Umgebung nicht übersteigt (z.B. durch Übergabe an einen Thread).

```
#include<iostream>
#include<vector>
#include<algorithm>

int main()
{
  std::vector<int> a(5,4);
  int i=1;
  // Compilerfehler:
  for_each(a.begin(),a.end(),[i](int &x){x=i*i;++i;});
  // Korrekt:
  for_each(a.begin(),a.end(),[i](int &x)mutable{x=i*i;++i;});
  std::cout << i << std::endl; // i ist immer noch 1
  // Lambda lassen sich auch mit einem Namen versehen
  auto Print = [](int x){ std::cout << "␣" << x;};
  for(auto x : a)
    Print(x);
  std::cout << std::endl;
}
```

- Soll es einem Lambda möglich sein, Variablen aus der Capture List, die als Kopie übergeben wurden, zu ändern, dann muss das Wort Mutable nach der Argumentliste stehen.

Lambda: Rückgabetyp

```
int main()
{
  double y=3.0;
  // return type wird automatisch bestimmt
  auto Add = [=](int x){ return x+y; };
  auto Bool = [=]{ if (y) return true; else return false; };
  auto Bool2 = [y](){ return y ? true : false; };
  // return type wird explizit bestimmt
  auto Bool3 = [y]()mutable->bool{ if (y) return true; else return false;};
  // verwende Lambda
  bool x = Bool();
}
```

- Gibt es in einem Lambda keine `return` Anweisung, ist der Rückgabetyp `void`.
- Gibt es nur eine einzige Rückgabeanweisung oder ist das Lambda einfach genug, wird der Rückgabetyp automatisch bestimmt.
- Geht das nicht, wird er nach der Argumentliste und einem eventuellen `mutable` nach `->` explizit angegeben.

Einheitliche Initialisierung

```
struct Date
{ unsigned int day, month, year;
  Date(int d, int m, int y) : day(d), month(m), year(y)
  {}
};

int main()
{
  int x {2};
  int u[] = { 1, 2, 3, 4, 5 };
  std::vector<double> y {2.3, 2.4, 2.5, 2.6, 2.7, 3.0, 5.0};
  std::vector<int> val1(7); // Ein Vektor mit sieben Elementen
  std::vector<int> val2{7}; // Ein Vektor mit einem Element mit Wert 7
  std::map<std::string,int> konto { {"Mueller", 100},
     {"Merkel", 1000000}, {"Ippisch", -200} };
  Date birthday {01, 01, 2013};
  for (int i : {1, 3, 9})
    std::cout << i << std::endl;
}
```

- C++11 führt eine neue Art der Initialisierung von Variablen und Objekten mit Hilfe von Initialisierungslisten ein.
- Diese hat den Vorteil, dass sie für unterschiedlichste Typen und Objekte funktioniert.

Typsichere Initialisierung

```
int main()
{
  char a1 = 1024;   // Overflow Warnung
  int z1 = y[0];    // Funktioniert
  char a2 {1024};   // Compilerfehler Narrowing
  int z2 {y[0]};    // Compilerfehler Narrowing
}
```

- Initialisierungslisten erlauben eine Typsichere Initialisierung.
- Würde bei einer Initialisierung Information verloren gehen (Narrowing) dann bricht der Compiler mit einer Fehlermeldung ab (sollte er jedenfalls).

Konstruktoren mit Initialisierungsliste

```
MyList(std::initializer_list<T> vals)
```

- Funktionen und Konstruktoren können Initialisierungslisten als Argumente erhalten. Der Typ ist `std::initializer_list<T>`.
- Über diese lässt sich wie über einen normalen Container iterieren. Die Werte lassen sich aber nicht ändern.
- Gibt es keinen solchen Konstruktor aber einen Konstruktor mit passenden Argumenten für eine gegebene Initialisierungsliste, wird dieser aufgerufen.
- Wird eine leere Initialisierungsliste übergeben hat der Default-Konstruktor Vorrang vor dem Konstruktor mit Initialisierungsliste.
- Gibt es einen normalen Konstruktor mit passend vielen Argumenten und einen Konstruktor mit Initialisierungliste, wird letzterer aufgerufen.

```
#include<list>

template<typename T>
class MyList
{
    std::list<T> mylist;
  public:
    MyList(std::initializer_list<T> vals)
    {
      for (auto x : vals)
        mylist.push_back(x);
    }
    MyList(unsigned int s)
    {
      for (int i=0;i<s;++i)
        mylist.push_back(i);
    }
};

int main()
{
  MyList<int> m { 1, 2, 3, 4, 5 };
  MyList<int> k {1}; // ruft initializer_list Konstruktor auf
  MyList<int> l(1);  // ruft Konstruktor mit einem Argument auf
}
```

17 Templatebasierte Design Patterns

- In großen Softwareprojekte wie z.B. in DUNE wollen wir verschiedene Realisierungen (Modelle) von bestimmten Konzepten (z.B. strukturierte und unstrukturierte Gitter ...) einfach verwenden und austauschen zu können.

- Wird statischer Polymorphismus zur Implementierung verwendet, gibt es keine einfache Möglichkeit den Compiler überprüfen zu lassen, ob ein Konzept korrekt implementiert ist (im Gegensatz zu dynamischem Polymorphismus wo wir dazu abstrakte Basisklassen verwenden können).
- Dies kann zu seltsamen Fehlermeldungen führen.
- Außerdem ist es nicht möglich die Funktionalität von virtuellen Funktionen zu bekommen (Basisklassenalgorithmen verwenden die Funktionalität der abgeleiteten Klasse).
- Zwei Ansätze die Situation zu verbessern sind:
  - das "Engine Konzept"
  - Vererbung unter Benutzung des "Curious Recurring Template Pattern" (manchmal auch fälschlich als Barton-Nackman-Trick bezeichnet).

17.1 Engine Konzept

- Es wird ein Klassentemplate definiert, dass einen Wrapper für das Modell darstellt, welches ein Templateparameter der Wrapperklasse ist.
- Alle Methodenaufrufe werden an das Modell weitergeleitet.
- Alle eingebetteten Typen des Modells werden in der Wrapperklasse gespiegelt.
- Generische Algorithmen werden mit Hilfe der Wrapperklasse geschrieben.

⇒ Das Modell ist wie ein Motor, der die Engineklasse antreibt.

Beispiel Engine Konzept

```
template<class EngineImp>
class Wrapper
{
  protected:
    EngineImp engine;
  public:
    typedef EngineImp ImplementationType;
    typedef typename EngineImp::someType someType;
    double doSomething(complex<float> blub)
    {
      return engine.doSomething(blub);
    }
};
```

17.2 Das Curious Recurring Template Pattern

Das *curious recurring template pattern* (CRTP) bezeichnet eine Gruppe von Templateprogrammen bei denen eine abgeleitete Klasse als Templateparameter an die Basisklasse weitergeleitet wird.

```
template<typename Derived>
class CuriousBase{ ... };

class Curious : public CuriousBase<Curious> { ... };
```

Das lässt sich noch weiter treiben:

```
template<template<typename> class Derived>
class MoreCuriousBase{ ...};

template<typename T>
class MoreCurious : public CuriousBase<MoreCurious<T> >{ ... };
```

Ein Anwendungsbeispiel ist der `curious counter`, der mitzählt, wie viele Objekte einer abgeleiteten Klasse gerade existieren.

curious_counter.hh

```
#ifndef CURIOUS_COUNTER_HH
#define CURIOUS_COUNTER_HH

#include<cstdlib>

template<typename CountedType>
class Counter
{
private:
  static std::size_t count;

protected:
  Counter(){
    ++count;
  }

  Counter(const Counter&){
    ++count;
  }

  ~Counter(){
    --count;
  }
public:
  static size_t live(){
    return count;
  }
};

template<typename T>
size_t Counter<T>::count=0;

#endif
```

curious_counter.cc

```
#include"curious_counter.hh"
#include<iostream>

template<typename CharT>
class MyString : public Counter<MyString<CharT> >
{};

int main()
{
  MyString<char> s1, s2;
```

```
  MyString<wchar_t> ws;
  std::cout<<"MyString<char>: "<<MyString<char>::live()
           <<std::endl;
  std::cout<<"MyString<wchar_t>: "<<MyString<wchar_t>::live()
           <<std::endl;
}
```

Statischer Polymorphismus mit dem CRTP

CRTP kann auch als Erweiterung des Engine Konzepts verwendet werden um Abstrakte Basisklassen und die damit definierten Schnittstellen zu emulieren:

```
template<typename Derived>
class AbstractBase{
private:
  Derived& getImplementation(){
    return *static_cast<Derived*>(this);
  }
  const Derived& getImplementation() const{
    return *static_cast<Derived*>(this);
  }
public:
   double interfaceMethod(){
     return getImplementation().interfaceMethod();
   }
   double constInterfaceMethod(int i) const{
     return getImplementation().constInterfaceMethod(i);
   }
};
```

Vor- und Nachteile von CRTP

- Dieser Ansatz lässt sich gut mit Traits kombinieren um die Eigenschaften der abgeleiteten Klasse zu charakterisieren.
- Der Vorteil ist, dass durch das Aufrufen von Basisklassenmethoden automatisch redefinierte Methoden der abgeleiteten Klasse verwendet werden.
- Wenn in der abgeleiteten Klasse Methoden fehlen, wird das Programm mit einem `segfault` enden.
- Wenn die abgeleitete Klasse von zusätzlichen Templateparametern abhängt oder wenn gleich mehrere Klassen mit diesem Trick verknüpft werden, kann die Sache ziemlich kompliziert werden.

18 Template Metaprogramming

18.1 Grundlagen des Template Metaprogramming

Was ist ein Metaprogramm?

Ein Metaprogramm ist ein Programm das andere Programme (oder sich selbst) schreibt und verändert oder das einen Teil der Berechnungen zur Übersetzungszeit durchführt die sonst

zur Laufzeit ausgeführt werden müssten. Metaprogramme konvertieren die Ausdrücke aus der *domänenspezifischen Sprache* in die *Wirts-Programmiersprache*.

In vielen Fällen erlaubt dies einem Programmierer in der selben Zeit mehr erledigt zu bekommen als wenn er den Code manuell schreiben würde.

Beispiele

- C++ Compiler
- Parser Generatoren (YACC, etc.)

Metaprogrammierung in C++

- Domänenspezifische Sprache und Wirtssprache sind identisch.
- Die Übersetzung zwischen beiden erfolgt durch den Compiler.
- Durch Zufall entdeckt [Unr94, Vel95]
- Die ersten C++ Metaprogramme führten Berechnungen mit ganzzahligen Datentypen zur Übersetzungszeit durch (Der Entdecker Unruh berechnete Primzahlen).
- Wichtigster Anwendungsfall ist die Fähigkeit von C++ mit Typen zu rechnen.

Warum Metaprogrammierung?

- Schneller als reine Laufzeitberechnungen.
- Enge Interaktion zwischen Meta- und Zielsprache, z.B. kann die Größe von Datentypen am besten direkt bei der Übersetzung festgestellt werden.
- Bequemer als die Durchführung der Berechnung per Hand.
- Verständlicherer Code.
- Entstehende Programme sind mit höherer Wahrscheinlichkeit korrekt und wartbar.

Erstes Template Metaprogramm

```
template<std::size_t N>
struct Factorial{
  enum{ value = N * Factorial<N-1>::value };
};

template<>
struct Factorial<1>{
  enum { value = 1 };
};

int main()
{
  std::cout<<Factorial<10>::value<<std::endl;
}
```

- `Factorial<N>::value` wird rekursiv durch Aufruf von `Factorial<N-1>::value` berechnet.
- Das Ende der Rekursion wird durch die Templatespezialisierung `Factorial<0>` erreicht.
- Die Berechnung erfolgt zur Übersetzungszeit!

Enum versus static Konstanten

- Konstanten lassen sich entweder als Enums oder als statische Konstanten realisieren:

```
struct Constants{
  enum{three=3};
  static const int four=4;
}
```

- Statische Konstanten sind lvalues (das heißt rein prinzipiell dürften sie auch auf der linken Seite einer Zuweisung stehen): Wird eine Funktion definiert als

  ```
  void foo(const int&)
  ```

  und übergibt man ihr eine statische Konstante wie z.B.

  ```
  foo(Constants::four)
  ```

  dann muss der Compiler der Funktion tatsächlich eine konstante Variable `Constants::four` übergeben und diese dafür auch erzeugen und dafür Speicher reservieren.

- Enums sind keine lvalues. Sie werden vom Compiler direkt eingesetzt (und benötigen deshalb keinen zusätzlichen Speicher).

Berechnung der Quadratwurzel

Wir können die (ganzzahlige) Quadratwurzel wie folgt durch Intervallschachtelung berechnen:

```
#include<iostream>

template<std::size_t N, std::size_t L=1, std::size_t H=N>
struct Sqrt
{
public:
  enum{ mid = (L+H+1)/2 };
  enum{ value = (N<mid*mid)? (std::size_t)Sqrt<N,L,mid-1>::value :
                (std::size_t)Sqrt<N,mid,H>::value };
};

template<std::size_t N, std::size_t M>
struct Sqrt<N,M,M>
{
  enum{ value = M };
};

int main()
{
  std::cout<<Sqrt<9>::value<<"␣"<<Sqrt<42>::value<<std::endl;
}
```

Template Instantiierungen

- Berechnung der `Sqrt<9>` führt zuerst mal zu der Ausführung von:

  ```
  Sqrt<9,1,9>::value=(9<25)? Sqrt<9,1,4>::value : Sqrt<9,5,9>::value =
      Sqrt<9,1,4>::value
  ```

  Daraus ergibt sich dass als nächstes `Sqrt<9,1,4>` berechnet werden muss:

  ```
  Sqrt<9,1,4>::value= (9<9) ? Sqrt<9,1,2>::value : Sqrt<9,3,4>::value =
      Sqrt<9,3,4>::value
  ```

  Der nächste Rekursionsschritt ist dann:

  ```
  Sqrt<9,3,4>::value = (9<16) ? Sqrt<9,3,3>::value : Sqrt<9,4,3>::value =
      Sqrt<9,3,3>::value = 3
  ```

- Es gibt allerdings ein Problem mit dem ternären Operator `<condition>?<true-path>:<false-path>`. Der Compiler generiert hier nicht nur den zutreffenden Teil, sondern auch den anderen. Dazu muss er aber auf der (für die Berechnung völlig uninteressanten Seite) weitere Rekursionslevel expandieren, so dass sich am Ende ein vollständiger Baum ergibt.

- Für die Quadratwurzel führt dies zu `N` Template Instantiierungen. Dies führt zu einer großen Beanspruchung der Ressourcen des Compilers (sowohl Laufzeit als auch Speicher) und schränkt den Anwendungsbereich ein.

Typauswahl zur Übersetzungszeit

Wir können die überflüssigen Template Instantiierungen loswerden indem wir einfach den richtigen Typ auswählen und nur diesen auswerten.

Dies lässt sich mit einem kleinen Metaprogramm ausführen, dass einer `if` Anweisung entspricht (auch “compile time type selection” genannt).

```
// Definition inklusive Spezialisierung fuer den true Fall
template<bool B, typename T1, typename T2>
struct IfThenElse
{
  typedef T1 ResultT;
};

// partielle Spezialisierung fuer den false Fall
template<typename T1, typename T2>
struct IfThenElse<false ,T1,T2>
{
  typedef T2 ResultT;
};
#endif
```

Verbesserte Berechnung der Quadratwurzel

Mit Hilfe unserer meta-`if` Anweisung können wir die Quadratwurzel wie folgt implementieren:

```
template<std::size_t N, std::size_t L=1, std::size_t H=N>
struct Sqrt
{
public:
  enum{ mid = (L+H+1)/2 };

  typedef typename IfThenElse<(N<mid*mid),Sqrt<N,L,mid-1>,
                              Sqrt<N,mid,H> >::ResultT ResultType;

  enum{ value = ResultType::value };
};

template<std::size_t N, std::size_t M>
struct Sqrt<N,M,M>
{
  enum{ value = M };
};
```

Diese kommt mit etwa $log_2(N)$ Template Instantiierungen aus!

Turing-Vollständigkeit von Template Metaprogrammierung

Template Metaprogramme können enthalten:

- Zustandsgrößen: die Templateparameter.
- Schleifen: durch Rekursion.
- Bedingte Abarbeitung: durch den Fragezeichen-Doppelpunkt-Operator oder Templatespezialisierung (z.B. das meta-`if`)
- Ganzzahlenberechnungen.

Dies ist ausreichend um jede mögliche Berechnung durchzuführen, solange es keine Beschränkung der Anzahl rekursiver Instantiierungen und der Anzahl von Zustandsvariablen gibt (was noch nicht heißt, dass es auch sinnvoll ist alles mit Template Metaprogrammierung zu berechnen).

Loop Unrolling

Bei der Berechnung eines Skalarprodukts wie in

```
template<typename T>
inline T dot_product (int dim, T* a, T* b)
{
  T result = T();
  for (int i=0;i<dim;++i)
  {
    result += a[i]*b[i];
  }
  return result;
}
```

optimiert der Compiler die Berechnung oft für große Arrays. Werden jedoch sehr häufig kleine Skalarprodukte des Types

```
dp = dot_product(3,a,b);
```

gebraucht, dann lässt sich das ganze effizienter schreiben.

```
// Primaeres Template
template <int DIM, typename T>
class DotProduct
{
  public:
    static T result (T *a, T *b)
    {
        return *a * *b  +  DotProduct<DIM-1,T>::result(a+1,b+1);
    }
};

// teilweise Spezialisierung als Abbruchkriterium
template <typename T>
class DotProduct<1,T>
{
  public:
    static T result (T *a, T *b)
    {
        return *a * *b;
    }
};

// zur Vereinfachung
template <int DIM, typename T>
inline T dot_product (T *a, T *b)
{
    return DotProduct<DIM,T>::result(a,b);
}
```

Anwendung Loop Unrolling

```
#include <iostream>
#include "loop_unrolling.h"

int main()
{
    int a[3] = { 1, 2, 3};
    int b[3] = { 5, 6, 7};

    std::cout << "dot_product<3>(a,b) = "
              << dot_product<3>(a,b) << '\n';
    std::cout << "dot_product<3>(a,a) = "
              << dot_product<3>(a,a) << '\n';
}
```

Loop Unrolling für Random Access Container

```
// Primaeres Template
template <int DIM, typename T, template<typename U,typename=std::allocator<U>
    > class vect>
struct DotProduct
{
    static T result(const vect<T> &a, const vect<T> &b)
    {
        return a[DIM-1]*b[DIM-1]  +  DotProduct<DIM-1,T,vect>::result(a,b);
```

```
    }
};

// teilweise Spezialisierung als Abbruchkriterium
template <typename T, template<typename U,typename=std::allocator<U> > class
    vect>
struct DotProduct<1,T,vect>
{
    static T result(const vect<T> &a, const vect<T> &b)
    {
        return a[0] * b[0];
    }
};

// zur Vereinfachung
template <int DIM, typename T, template<typename U,typename=std::allocator<U>
    > class vect>
inline T dot_product(const vect<T> &a, const vect<T> &b)
{
    return DotProduct<DIM,T,vect>::result(a,b);
}
```

Anwendung Loop Unrolling für Random Access Container

```
#include <iostream>
#include <vector>
#include "loop_unrolling2.h"

int main()
{
    std::vector<double> a(3,3.0);
    std::vector<double> b(3,5.0);

    std::cout << "dot_product<3>(a,b) = "
              << dot_product<3>(a,b) << '\n';
    std::cout << "dot_product<3>(a,a) = "
              << dot_product<3>(a,a) << '\n';
}
```

constexpr

```
#include<iostream>

int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1; // Fehler, x1 ist keine constexpr
constexpr int x4 = x2;

constexpr int Fac(int n)
{
  return n<2 ? 1 : n * Fac(n-1);
}

int main()
{
```

```
  std::cout << Fac(10) << std::endl;
}
```

- C++11 führt eine einfache Alternative zu Template Metaprogramming ein: Ausdrücke, die schon zur Übersetzungszeit ausgewertet werden.
- In `constexpr` dürfen nur Variablen oder Funktionen verwendet werden, die selbst wieder `constexpr` sind.

Eine `constexpr` muss zur Übersetzungszeit evaluierbar sein:

```
void f(int n)
{
    constexpr int x = Fac(n); // Fehler, n ist zur Uebersetzungszeit nicht
        bekannt
    int f10 = Fac(10);        // In Ordnung
}

int main()
{
    const int ten = 10;
    int f10 = Fac(10);        // Auch in Ordnung
}
```

Dies funktioniert sogar für Objekte von Klassen deren Konstruktor einfach genug ist um als `constexpr` definiert zu werden:

```
struct Point
{
  int x,y;
  constexpr Point(int xx, int yy) : x(xx), y(yy)
  {}
};

constexpr Point origo(0,0);
constexpr int z = origo.x;

constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2)};
constexpr x = a[1].x;   // x wird 1
```

- `constexpr` Funktionen dürfen nicht den Rückgabetype `void` haben und es dürfen in ihnen keine Variablen oder Funktionen definiert werden (das gilt auch für `constexpr` Konstruktoren).
- Der Funktionsrumpf darf nur Deklarationen und eine einzige `return`-Anweisung enthalten.

18.2 Beispiel: Zahlen mit Einheiten

- Bei Rechnungen mit einheitenbehafteten Größen, kann es leicht zu Fehlern kommen.
- Im schlimmsten Fall werden Äpfel mit Birnen verwechselt.
- Hier soll nun eine Klasse entwickelt werden, die das Rechnen mit Einheiten erlaubt.
- Die Realisierung erfolgt mit Template-Metaprogramming. Alle Berechnungen (außer der Konvertierung bei ein und Ausgabe) erfolgen genauso schnell wie ohne Einheiten.

- Die entsprechenden Tests werden zur Übersetzungszeit ausgeführt und weg optimiert.
- Das Beispiel orientiert sich an der entsprechenden Klasse aus [Str13]

Units: Einheitenklasse

Wir führen zunächst eine Template-Klasse für die Einheit ein:

```
template<int M, int K, int S>
struct Unit {
  enum { m=M, kg=K, s=S };
};

using M = Unit<1,0,0>;      // Meter
using Kg = Unit<0,1,0>;     // Kilogramm
using S = Unit<0,0,1>;      // Sekunde
using MpS = Unit<1,0,-1>;   // Meter pro Sekunde (m/s)
using MpS2 = Unit<1,0,-2>; // Meter pro Sekundequadrat (m/(s*s))
```

Enum-Klassen

```
enum {RED, GREEN, BLUE}; // C-Enum
enum class Color {RED, GREEN, BLUE}; // C++11-Enum
```

- Enums in C sind einfach Integer-Werte mit einem Namen.
- Der gleiche Namen darf nur einmal verwendet werden.
- Es kann an der gleichen Stelle an der ein Enum eingesetzt wird auch einfach irgendein anderer Integer-Wert eingesetzt werden.
- C++11 führt Enum-Klassen ein.
- Damit ist jede Enum-Klasse ein eigener Typ und hat ihren eigenen Namen und damit auch Namespace. Beide oben beschriebenen Probleme sind damit behoben.
- Enums können nur noch explizit nach Integer gecastet werden.
- Für Template-Metaprogramming sind die C-Enums deshalb praktischer

```
#include<iostream>

enum class TimeIntegration : unsigned char {EE = 2, IE = 4, CN = 8, BDF2 = 16};

// verwendet int als internen Datentyp
enum class SpatialIntegration {CCFV,FCFV,FE,DG};

template<TimeIntegration T, SpatialIntegration S>
void DoTimeStep()
{
  // explizite Typumwandlung nach int moeglich
  std::cout << (unsigned int)T << "␣" << (int) S << std::endl;
}

int main()
```

```
{
  // Scope muss vorangestellt werden
  DoTimeStep<TimeIntegration::CN,SpatialIntegration::FE>();
  TimeIntegration ti = TimeIntegration::IE;
  ti = 1;  // geht nicht, keine Konversion von int->TimeIntegration
  SpatialIntegration si = ti; // geht nicht, falscher Typ
}
```

Units: Hilfsklassen

Für das Rechnen mit Einheiten brauchen wir noch ein paar Hilfsklassen. Mit Hilfe der `using` Deklarationen bauen wir uns Template-Funktionen.

```
template<typename U1, typename U2>
struct Uplus {
  using type = Unit<U1::m+U2::m, U1::kg+U2::kg, U1::s+U2::s>;
};

template<typename U1, typename U2>
using Unit_plus = typename Uplus<U1,U2>::type;

template<typename U1, typename U2>
struct Uminus {
  using type = Unit<U1::m-U2::m, U1::kg-U2::kg, U1::s-U2::s>;
};

template<typename U1, typename U2>
using Unit_minus = typename Uminus<U1,U2>::type;

template<typename U>
using Unit_negate = typename Uminus<Unit<0,0,0>,U>::type;
```

Quantities

Jetzt können wir eine Klasse einführen, in der die Werte mit ihren Einheiten abgespeichert werden. Da die Einheiten nur als Templateparameter verwendet werden, wird durch sie nur der Klassentyp beeinflusst, aber kein Speicher verbraucht. Um möglichst flexibel zu bleiben ist auch der Datentyp ein Templateparameter.

```
template<typename U, typename V=double>
struct Quantity {
  V val;
  explicit constexpr Quantity(V d) : val{d}
  {}
  template<typename V2>
  constexpr Quantity(Quantity<U,V2> d) : val{static_cast<V>(d.val)}
  {}
};
```

Explizite Konversion

- Konstruktoren mit einem Argument werden von C++ für die automatische Typkonvertierung verwendet.
- Dies ist nicht immer gewünscht. So soll es z.B. in diesem Beispiel nicht möglich sein eine einheitenlose mit einer einheitenbehafteten Zahl zu addieren oder subtrahieren.

- Dazu kann man in C++11 Konstruktoren `explicit` machen.
- Der Konstruktor wird dann nur verwendet, wenn er explizit aufgerufen wird, z.B. mit `Quantity<M>(2.73)`

Quantities: Addition und Subtraktion

Wir können jetzt mit den Quantities rechnen. Für Addition und Subtraktion muss die Einheit von beiden Operanden und für das Ergebnis gleich sein. Als Datentyp kann alles verwendet werden für das es einen passenden `operator+` bzw. `operator-` gibt.

```
template<typename U, typename V1, typename V2>
auto operator+(Quantity<U,V1> x, Quantity<U,V2>
    y)->Quantity<U,decltype(x.val+y.val)>
{
  return Quantity<U,decltype(x.val+y.val)>(x.val+y.val);
}

template<typename U, typename V1, typename V2>
auto operator-(Quantity<U,V1> x, Quantity<U,V2>
    y)->Quantity<U,decltype(x.val-y.val)>
{
  return Quantity<U,decltype(x.val-y.val)>(x.val-y.val);
}
```

Quantities: Multiplikation und Division

Bei der Multiplikation addieren sich die Einheiten komponentenweise, bei der Subtraktion wir die Differenz gebildet.

```
template<typename U1, typename U2, typename V1, typename V2>
auto operator*(Quantity<U1,V1> x, Quantity<U2,V2>
    y)->Quantity<Unit_plus<U1,U2>,decltype(x.val*y.val)>
{
  return Quantity<Unit_plus<U1,U2>,decltype(x.val*y.val)>(x.val*y.val);
}

template<typename U1, typename U2, typename V1, typename V2>
auto operator/(Quantity<U1,V1> x, Quantity<U2,V2>
    y)->Quantity<Unit_minus<U1,U2>,decltype(x.val/y.val)>
{
  return Quantity<Unit_minus<U1,U2>,decltype(x.val/y.val)>(x.val/y.val);
}
```

Quantities: Multiplikation mit einheitenlosen Werten

Bei der Multiplikation mit einem einheitenlosen Wert bleibt die Einheit gleich.

```
template<typename U,typename V1, typename V2>
auto operator*(Quantity<U,V1> x, V2 y)->Quantity<U,decltype(x.val*y)>
{
  return Quantity<U,decltype(x.val*y)>(x.val*y);
}

template<typename U,typename V1, typename V2>
auto operator*(V1 y, Quantity<U,V2> x)->Quantity<U,decltype(x.val*y)>
{
```

```
  return Quantity<U,decltype(x.val*y)>(x.val*y);
}
```

Quantities: Division mit einheitenlosen Werten

Wird eine Quantity durch eine einheitenlose Zahl geteilt, bleibt die Einheit gleich. Im umgekehrten Fall drehen sich die Vorzeichen aller Komponenten der Einheit um.

```
template<typename U,typename V1, typename V2>
auto operator/(Quantity<U,V1> x, V2 y)->Quantity<U,decltype(x.val/y)>
{
  return Quantity<U,decltype(x.val/y)>(x.val/y);
}

template<typename U,typename V1, typename V2>
auto operator/(V1 y, Quantity<U,V2>
    x)->Quantity<Unit_negate<U>,decltype(y/x.val)>
{
  return Quantity<Unit_negate<U>,decltype(y/x.val)>(y/x.val);
}
```

Benutzerdefinierte Literale

Wir haben in der letzten Vorlesung gesehen, wie der Typ von Variablen mit Hilfe von Literalen festgelegt werden kann. Es wäre schön, wenn sich hier weitere Literale definieren ließen, z.B.

```
"Hi!"s            // string, nicht ‘‘zero-terminated array of char’’
1.2i              // imaginaere Zahl
123.4567891234df  // decimal floating point (IBM)
101010111000101b  // binaere Zahl
123s              // Sekunden
123.56km          // Kilometer
12345678901234567890123456789012345678901234567890x   // extended-precision
```

In C++11 ist dies möglich, z.B. für komplexe Zahlen und Strings (es sind allerdings nur Literale, die mit einem '_' beginnen erlaubt.:

```
constexpr complex<double> operator"" _i(long double d)  // imaginaeres literal
{
    return {0,d};   // liefert entsprechenden komplexe Zahl zurueck
}

std::string operator"" _s (const char* p, size_t n)   // std::string literal
{
    return string(p,n); // requires free store allocation
}
```

Dies lässt sich wie folgt verwenden:

```
template<class T> void f(const T&);
f("Hello");       // gibt const char * an Funktion
f("Hello"_s);     // gibt string an Funktion
f("Hello"_s+"Dolly");  // geht, weil erster Operand string ist

auto z = 2+1_i;  // complex(2,1)
```

Es lassen sich auch direkt C-Strings übergeben:

```
Bignum operator"" _x(const char *p)
{
    return Bignum(p);
}

void f(Bignum);
f(1234567890123456789012345678901234567890_x);
```

aber nicht immer:

```
std::string operator"" _S(const char *p);  // funktioniert nicht

std::string blub = "one␣two"_S; // Fehler: no applicable literal operator
```

Raw String Literale

Will man in einem String z.B. einen Backslash verwenden, dann muss der Backslash als \\ geschrieben werden muss. Das macht das ganze schwer lesbar, insbesondere bei den neu eingeführten regulären Ausdrücken:

```
    string s = "\\w\\\\\\w";    // Hoffentlich stimmt das...
```

Bei einem raw String Literal wird jedes Zeichen einfach direkt als solches geschrieben:

```
std::string s = R"(\w\\w)"; // I'm pretty sure I got that right
std::string path=R"(c:\Programme\blub\blob.exe)";
```

Der erste Vorschlag für die Einführung von raw String Literalen wurde mit dem folgenden Beispiel motiviert:

```
"('(?:[^\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
// Are the five backslashes correct or not?
// Even experts become easily confused.
```

Ein raw String Literal wird durch `R"(` angefangen und durch `)"` beendet.

```
R"("quoted string")"    // the string is "quoted string"
```

Sollen in dem String selbst die Kombination `"(` oder `)"` vorkommen, dann lassen sich zwischen Klammer und Anführungszeichen beliebige Zeichenkombinationen einschieben, die den Begrenzer eindeutig machen:

```
R"***("quoted string containing the usual terminator ("])")***"
//␣the␣string␣is␣"quoted string containing the usual terminator ("])"
```

Reguläre Ausdrücke

Wie in vielen anderen Programmiersprachen lassen sich auch in C++11 reguläre Ausdrücke verwenden:

```
#include<regex>
#include<iostream>
#include<string>

int main()
{
  std::regex name_re(R"--(([a-zA-Z]+)\s+([a-zA-Z]+))--");
  std::string name="Torsten␣Will";
  if(regex_match(name.begin(),name.end(),name_re))
    std::cout << "Hallo" << name << std::endl;
```

```
  else
    std::cout << "Wer␣sind␣Sie?" << std::endl;
}
```

String nach Zahl und Zahl nach String Umwandlung

```
#include<string>
#include"print.h"

int main()
{
  std::string sVal = "-2.47␣3.1415␣1e300␣42␣376857689403␣0xFF";
  size_t next=0;
  float fVal=std::stof(sVal,&next);
  sVal = sVal.substr(next);
  double dVal=std::stod(sVal,&next);
  sVal = sVal.substr(next);
  long double ldVal=std::stold(sVal,&next);
  sVal = sVal.substr(next);
  int iVal=std::stoi(sVal,&next);
  sVal = sVal.substr(next);
  long lVal= std::stol(sVal,&next);
  sVal = sVal.substr(next);
  unsigned long ulVal= std::stoul(sVal,&next,16);
  Printf("%d,␣%d,␣%d,␣%s,␣%g,␣%s\n",iVal,lVal,ulVal,
      std::to_string(fVal),dVal,std::to_string(ldVal));
}
```

Literale für Quantities mit Fließkommazahlen

Jetzt können wir jede Menge eigene Literale definieren. Erst für Quantities, die als Datentyp Fließkommazahlen verwenden:

```
constexpr Quantity<M,long double> operator"" _m(long double value)
{
  return Quantity<M,long double> {value};
}

constexpr Quantity<Kg,long double> operator"" _kg(long double value)
{
  return Quantity<Kg,long double> {value};
}

constexpr Quantity<S,long double> operator"" _s(long double value)
{
  return Quantity<S,long double> {value};
}

constexpr Quantity<M,long double> operator"" _km(long double value)
{
  return Quantity<M,long double> {1000*value};
}

constexpr Quantity<Kg,long double> operator"" _g(long double value)
{
  return Quantity<Kg,long double> {value/1000};
}
```

```
constexpr Quantity<Kg,long double> operator"" _mg(long double value)
{
  return Quantity<Kg,long double> {value/1000000};
}

constexpr Quantity<S,long double> operator"" _min(long double value)
{
  return Quantity<S,long double> {60*value};
}

constexpr Quantity<S,long double> operator"" _h(long double value)
{
  return Quantity<S,long double> {3600*value};
}

constexpr Quantity<S,long double> operator"" _d(long double value)
{
  return Quantity<S,long double> {86400*value};
}

constexpr Quantity<S,long double> operator"" _ms(long double value)
{
  return Quantity<S,long double> {value/1000};
}

constexpr Quantity<S,long double> operator"" _us(long double value)
{
  return Quantity<S,long double> {value/1000000};
}

constexpr Quantity<S,long double> operator"" _ns(long double value)
{
  return Quantity<S,long double> {value/1000000000};
}
```

Literale für Quantities mit Integern

Dann für Quantities, die als Datentyp Ganzzahlen verwenden:

```
constexpr Quantity<M,unsigned long long> operator"" _m(unsigned long long
    value)
{
  return Quantity<M,unsigned long long> {value};
}

constexpr Quantity<Kg,unsigned long long> operator"" _kg(unsigned long long
    value)
{
  return Quantity<Kg,unsigned long long> {value};
}

constexpr Quantity<S,unsigned long long> operator"" _s(unsigned long long
    value)
{
  return Quantity<S,unsigned long long> {value};
}
```

```
constexpr Quantity<M,unsigned long long> operator"" _km(unsigned long long
    value)
{
  return Quantity<M,unsigned long long> {1000*value};
}

constexpr Quantity<S,unsigned long long> operator"" _min(unsigned long long
    value)
{
  return Quantity<S,unsigned long long> {60*value};
}

constexpr Quantity<S,unsigned long long> operator"" _h(unsigned long long
    value)
{
  return Quantity<S,unsigned long long> {3600*value};
}

constexpr Quantity<S,unsigned long long> operator"" _d(unsigned long long
    value)
{
  return Quantity<S,unsigned long long> {86400*value};
}
```

Quantities: Quadratur und Vergleichsfunktion

Wir möchten Quantities auch Quadrarieren und vergleichen können.

```
template<typename U, typename V>
Quantity<Unit_plus<U,U>,V> square(Quantity<U,V> x)
{
  return Quantity<Unit_plus<U,U>,V>(x.val*x.val);
}

template<typename U,typename V>
bool operator==(Quantity<U,V> x, Quantity<U,V> y)
{
  return x.val==y.val;
}

template<typename U,typename V>
bool operator!=(Quantity<U,V> x, Quantity<U,V> y)
{
  return x.val!=y.val;
}
```

Quantities: Größer und Kleiner

Ungleichheit sollte sogar für unterschiedliche Datentypen bei gleicher Einheit gehen:

```
}

template<typename U,typename V1, typename V2>
bool operator<(Quantity<U,V1> x, Quantity<U,V2> y)
{
  return x.val<y.val;
}
```

```
template<typename U,typename V1, typename V2>
bool operator>(Quantity<U,V1> x, Quantity<U,V2> y)
{
  return x.val>y.val;
}
```

Quantities: Ausgabe

Wir möchten Quantities auch mit den richtigen Einheiten ausgeben können. Das geht über eine einfache Funktion und einen überladenen Ausgabeoperator:

```
std::string suffix(int u, const char *x)
{
  std::string suf;
  if (u) {
    suf += x;
    if (1<u) suf += '0' + u;
    if (u<0) {
      suf += '-';
      suf += '0'-u;
    }
  }
  return suf;
}

template<typename U, typename V>
std::ostream &operator<<(std::ostream &os, Quantity<U,V> v)
{
  return os << v.val << suffix(U::m,"m") << suffix(U::kg,"kg") <<
      suffix(U::s,"s");
}
```

Quantities: Anwendungsbeispiel

Jetzt können wir mit unseren Quantities rechnen:

```
#include"units.h"

int main()
{
  Quantity<M,double> x {10.5};
  Quantity<S,int> y {2};
  Quantity<MpS,double> v = x/y;
  v = 2*v;
  auto distance = 10_m;
  Quantity<S,double> time = 20_s;
  auto speed = distance/time;
  Quantity<MpS2,double> acceleration = distance/square(time);

  std::cout << "Geschwindigkeit = " << speed << " Beschleunigung = " <<
      acceleration << std::endl;
}
```

Output: `Geschwindigkeit = 0.5ms-1 Beschleunigung = 0.025ms-2`

18.3 C++ Printf

Die Funktion `printf()` ist in C++ bisher immer noch eine C-Funktion. In C++11 lässt sich dies ändern. Die Anwendung sieht wie folgt aus:

```
#include"print.h"

int main()
{
  const char *pi = "pi";
  Printf("The␣value␣of␣%s␣is␣about␣%g␣(unless␣you␣live␣in␣%s).\n",  pi,
      3.14159, "Indiana");
  const std::string name="Stefan";
  int age=24;
  float grade=1.3;
  Printf(
  R"(
␣␣The␣student␣%s,␣%d␣years␣old␣has␣received␣the␣grade␣%g.
␣␣He␣is␣among␣the␣top␣1␣%%.
)",name,age,grade);
}
```

Output:

```
The value of pi is about 3.14159 (unless you live in Indiana).
The student Stefan, 24 years old has received the grade 1.3.
He is among the top 1 %.
```

Variadische Templates

Der einfachste Fall von `printf()` ist der in dem es außer dem Formatstring keine Argumente gibt (siehe [Str13]):

```
#include<iostream>
#include<type_traits>
#include<stdexcept>

void Printf(const char *s)
{
  if (s==nullptr)
    return;
  while (*s)
  {
    if (*s=='%' && *++s!='%') // stelle sicher, dass es nicht doch
                              // Argumente gibt. %% ist ein
                              // normales % in einem Formatstring
          throw std::runtime_error("invalid␣format:␣missing␣arguments");
    std::cout << *s++;
  }
}
```

Nun müssen wir `printf()` mit mehreren Argumenten behandeln. Notwendig dafür ist eine variable Anzahl von Templateargumenten:

```
template<typename T, typename... Args> // beachte die "..."
void Printf(const char *s, T value, Args... args)
{
  while (s && *s)
  {
```

```
    if (*s=='%'){
      switch (*++s){
        case '%':
          break;
        case 's':
          if (!Is_C_style_string<T>() && !Is_string<T>())
            throw std::runtime_error("Bad Printf() format");
          break;
        case 'd':
          if (!std::is_integral<T>())
            throw std::runtime_error("Bad Printf() format");
          break;

        case 'g':
          if (!std::is_floating_point<T>())
            throw std::runtime_error("Bad Printf() format");
          break;
        default:
            throw std::runtime_error("Unknown Printf() format");
        }
        std::cout << value;
        return(Printf(++s,args...)); // hier sind wieder die "..."
      }
      std::cout << *s++;
  }
  throw std::runtime_error("extra arguments provided to Printf");
}
```

- Mit variadischen Templates bekommt man also in jedem Aufruf nur den vordersten Teil der Argumentliste zu sehen. `T` kann dabei bei jeden Aufruf ein anderer Typ sein. Dann kann man den Rest erneut weiter geben.
- Wichtig sind dabei die drei Punkte nach `typename`, Templatetype in der Argumentliste und nach dem entsprechenden Variablennamen im erneuten Funktionsaufruf.

Während die Prädikate `is_integral<T>()` und `is_floating_point<T>` schon vordefiniert sind, müssen `Is_C_style_string<T>()` und `Is_string<T>()` noch definiert werden:

```
template<typename T>
bool Is_C_style_string()
{
  return std::is_same<T,const char*>() || std::is_same<T,char *>();
}

template<typename T>
bool Is_string()
{
  return std::is_same<T,const std::string>() || std::is_same<T,std::string>();
}
```

Tuples

Eine weitere Anwendung von variadic Templates sind Tuple, eine Verallgemeinerung von Pairs auf beliebig viele Komponenten.

- Deren Typ kann mit Hilfe von `auto` und `std::make_tuple` auch automatisch generiert werden.
- Die Hilfsfunktion `std::get<i>` liefert den i-ten Bestandteil des Tuples zurück.

```
#include<tuple>
#include<string>

int main()
{
  std::tuple<std::string,int> t2("Mueller",123);
  auto t = std::make_tuple(std::string("Mayer"),10, 1.23);
  // der Typ von t ist tuple<string,int,double>
  std::string s = std::get<0>(t);
  int x = std::get<1>(t);
  double d = std::get<2>(t);
}
```

Externe Templates

Bei Projekten, die aus vielen Einzeldateien bestehen, werden Templates oft an mehreren Stellen instanziiert. Durch die Deklaration von Templates unter Verwendung des Schlüsselwortes `extern` erfolgt das nicht. Hier z.B. im Headerfile `extern.h`

```
#include<vector>

extern template class std::vector<int>;

int blub(std::vector<int> &a)
{
  return a[0];
}
```

und ebenfalls nicht im Hauptprogramm `extern.cc`

```
#include"extern.h"

extern template class std::vector<int>;

int main()
{
  std::vector<int> x(5,5.);
  int y = blub(x);
}
```

Die eigentliche Instantiierung erfolgt an einer wohldefinierten Stelle explizit, hier in der Datei `extern2.cc`:

```
#include<vector>

template class std::vector<int>;
```

Es darf dann aber nicht vergessen werden alle Dateien auch zusammen zu linken, im einfachsten Fall durch: `g++ extern2.cc extern.cc`

C++11: Zufallszahlen

C++11 stellt mächtige Zufallszahlengeneratoren bereit, die mit verschiedenen Verteilungen arbeiten können, z.B. gleichverteilte Zufallszahlen:

```
#include<iostream>
#include<iomanip>
#include<string>
#include<map>
#include<random>
#include<cmath>

int main()
{
  // Implementierungsabhaengiger vordefinierter Zufallszahlengenerator
  std::default_random_engine e1;
  // produziere gleichverteilte Integer Zahlen zwischen 1 und 6
  std::uniform_int_distribution<int> uniformDist(1,6);
  // ziehe eine Zahl
  int mean = uniformDist(e1);
  std::cout << "Zufaellig␣gewaehlter␣Mittelwert:␣" << mean << '\n';
```

Normalverteilte Zufallszahlen

```
  // Generator fuer nicht-deterministische gleichverteilte Zufallszahlen
  std::random_device rd;
  // Alternativer Zufallszahlengenerator basierend auf dem
  // Mersenne Twister Algorithmus
  std::mt19937 e2(rd());
  // Erzeuge Normalverteilung um Mittelwert
  std::normal_distribution<> normalDist(mean, 2);

  std::map<int, int> hist;
  // Erzeuge Zufallszahlen und zaehle Haeufigkeit
  for (int n = 0; n < 10000; ++n)
      ++hist[std::round(normalDist(e2))];
  // Gebe Histogramm am Bildschirm aus
  std::cout << "Normalverteilung␣um␣den␣Mittelwert␣" << mean << ":\n";
  for (auto p : hist)
      std::cout << std::fixed << std::setprecision(1) << std::setw(2)
                << p.first << "␣" << std::string(p.second/200, '*')
                << std::endl;
}
```

Eigener gleichverteilter Integer-Zufallsgenerator

```
class RandomInt
{
  public:
    RandomInt(int low, int high, unsigned int seed=0) :
      rand_{std::bind(std::uniform_int_distribution<>{low,high},
          std::default_random_engine{seed})}
    {}
    int operator()() const
    {
```

```
      return rand_();
    }
  private:
    std::function<int()> rand_;
};
```

Eigener normalverteilter Double-Zufallsgenerator

nicht nur für Integer-Werte:

```
class RandomDouble
{
  public:
    RandomDouble(double mean, double stddev, unsigned int seed=0 ) :
      rand_{std::bind(std::normal_distribution<>{mean,stddev},
          std::mt19937{seed})}
    {}
    double operator()() const
    {
      return rand_();
    }
  private:
    std::function<double()> rand_;
};
```

Anwendung

```
int main()
{
  // Generator fuer nicht-deterministische gleichverteilte Zufallszahlen
  std::random_device rd;
  RandomInt randInt{1,6,rd()};
  int mean = randInt();
  std::cout << "Zufaellig␣gewaehlter␣Mittelwert:␣" << mean << '\n';

  RandomDouble randDouble{(double)mean,2.,rd()};

  std::map<int, int> hist;
  // Erzeuge Zufallszahlen und zaehle Haeufigkeit
  for (int n = 0; n < 10000; ++n)
      ++hist[std::round(randDouble())];
  // Gebe Histogramm am Bildschirm aus
  std::cout << "Normalverteilung␣um␣den␣Mittelwert␣" << mean << ":\n";
  for (auto p : hist)
      std::cout << std::fixed << std::setprecision(1) << std::setw(2)
                << p.first << "␣" << std::string(p.second/200, '*')
                << std::endl;
}
```

20 Threads

20.1 Introduction

Prozesse und Threads in Unix

- Mit einem Prozess in UNIX sind folgende Informationen verbunden:

  - IDs (prozess,user,group)
  - Umgebungsvariablen
  - Verzeichnis
  - Code
  - Register, Stack, Heap.
  - File descriptor, Signale
  - Message Queues, Pipes, Shared-memory Segmente
  - Shared libraries.
- Jeder Prozess hat seinen eigenen Adressraum.
- Threads (abgespeckte Prozesse) existieren innerhalb eines Prozesses.
- Threads teilen den Adressraum des Hauptprozesses.
- Ein Thread besteht aus:
  - ID
  - Stack pointer
  - Register
  - Scheduling Eigenschaften
  - Signalen
- Das Umschalten zwischen Threads ist schneller als das Umschalten zwischen Prozessen.

C++-11 Threads

- POSIX Threads (oder pthreads) wurden 1995 eingeführt und sind seit langem das Standardmodell für Threads auf UNIX-Computern. Sie werden über ein C Interface erzeugt und gemanagt.
- C++-11 definiert ein eigenes threading Konzept, das im Grunde ein Wrapper für POSIX-Threads ist. Es erlaubt jedoch in vielen Fällen multi-threadding Programme viel einfacher zu programmieren. Dabei werden die folgenden Konzepte unterstützt:

  Threads Im Header `threads` werden Thread-Klassen und Funktionen für das wiederzusammenführen von Threads definiert. Außerdem Funktionen um eine Thread-ID zu erhalten und für das die Loslösung von Threads, die dann als eigenes Programm unabhängig weiter laufen können.

  Mutual Exclusion Im Header `mutex` werden Klassen für mutexes und locks definiert (auch rekursive Varianten und solche mit Timeouts). Außerdem Funktionen die prüfen ob ein lock verfügbar ist.

Condition Variables Im Header `condition_variable` wird eine Klasse definiert, die die Koordination von mehreren Threads mit Condition Variables erlaubt.

Futures Der Header `futures` definiert Klassen und Funktionen deren Operationen nicht sofort ausgeführt werden müssen sondern asynchron zum Hauptteil des Programms ausgeführt werden können. Dies erfolgt wenn möglich parallel, wenn nicht sequentiell an der Stelle an der das Ergebnis gebraucht wird.

Atomare Operationen Die Klassen aus dem Header `atomic` erlauben es bestimmte Operationen mit Integer oder Bool Variablen sowie Pointern in einer atomaren Operation durchzuführen.

20.2 C++-11 Thread Erzeugung

```
#include <array>
#include <iostream>
#include <thread>

void Hello(size_t number)
{
    std::cout << "Hello␣from␣thread␣" << number << std::endl;
}

int main()
{
    const int numThreads = 5;
    std::array<std::thread,numThreads> threads;

    // starte threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i] = std::thread(Hello, i);

    std::cout << "Hello␣from␣main\n";

    // Wiedervereinigung mit den Threads, implizite Barriere
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();

    return 0;
}
```

Ausgabe

```
Hello from thread Hello from thread Hello from thread Hello from thread Hello from main
0Hello from thread 213
4
```

20.3 Beispiel: Berechnung der Vektornorm

```
#include <array>
#include <vector>
#include <iostream>
#include <thread>

void Norm(const std::vector<double> &x, double &norm, const size_t i,
    const size_t p)
{
     size_t numElem = x.size()/p;
     size_t first = numElem * i + std::min(i,x.size()%p);
     numElem += (i<(x.size()%p)?1:0);
     double localNorm = 0.0;
```

```
    for (size_t j=0;j<numElem;++j)
        localNorm+=x[first+j]*x[first+j];

    norm += localNorm;  // gefaehrlich...
}

int main()
{
    const size_t numThreads = 5;
    const size_t numValues = 1000000;
    std::array<std::thread,numThreads> threads;
    std::vector<double> x(numValues,2.0);

    double norm = 0.0;
    // Create threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i] =
           std::thread(Norm,std::cref(x),std::ref(norm),i,numThreads);

    // Rejoin threads with main thread, barrier
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();

    std::cout << "Norm␣is:␣" << norm << std::endl;
    return 0;
}
```

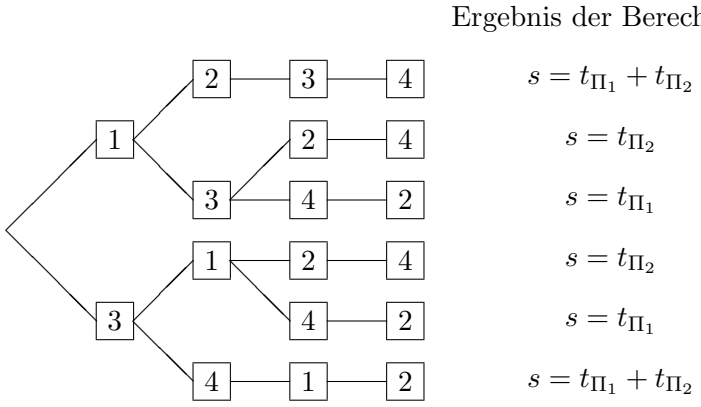
Critical Sections

- Eine Anweisung vom Typ $s{=}s{+}t$ ist nicht atomic, d.h. sie wird nicht in einem Schritt durchgeführt:

	Prozess Π_1		Prozess Π_2
1	lade s in R1	3	lade s in R1
	lade t in R2		lade t in R2
	addiere R1, R2, speichere in R3		addiere R1, R2, speichere in R3
2	speichere R3 in s	4	speichere R3 in s

- Die Reihenfolge der Ausführung dieser Anweisungen relativ zueinander ist nicht festgelegt.
- Dadurch ergibt sich eine exponentiell anwachsende Folge von möglichen Ausführungsreihenfolgen.

Mögliche Ausführungsreihenfolgen



Nur einige der Reihenfolgen führen zum richtigen Ergebnis!

20.4 Mutual Exclusion/Locks

- Es ist eine zusätzliche *Synchronisierung* notwendig um Ausführungsreihenfolgen zu verhindern, die nicht das richtige Ergebnis liefern.
- Critical sections müssen unter gegenseitigem Ausschluss (*mutual exclusion*) ausgeführt werden.
- Mutual exclusion erfordert:
  - Höchstens ein Prozess betritt die critical section auf einmal.
  - Es gibt keine deadlocks (also dass zwei Prozesse gegenseitig aufeinander warten).
  - Kein Prozess wartet auf eine freie critical section.
  - Wenn ein Prozess die critical section betritt, wird er sie am Ende auch erfolgreich verlassen.

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>

std::mutex m;

void e(int &sh)
{
  m.lock();
  sh+=1; // veraendere gemeinsame Daten
  m.unlock(); // darf nie vergessen werden
}
```

- In C++11 gibt es dafür das Konstrukt des `mutex`.
- Ein `mutex` kann einen Berech sperren, so dass er nur von einem Prozess betreten werden kann.
- Alle anderen Prozesse müssen dann vor dem kritischen Abschnitt warten. Dieser sollte deshalb so klein wie möglich sein.
- Der gleiche Mutex muss in allen Threads verfügbar sein.

Lock Guard

```
void f(int &sh)
{
  std::lock_guard<std::mutex> locked{m};
  sh+=1; // veraendere gemeinsame Daten
}
```

- Damit nicht vergessen wird ein `lock` auch wieder freizugeben (auch wenn z.B. eine Exception geworfen wird) gibt es Hilfsklassen.
- Die einfachste ist ein `lock_guard`.
- Dieser bekommt bei der Initialisierung einen `mutex`, sperrt diesen und gibt ihn wieder frei, wenn der Destruktor des `lock_guard` aufgerufen wird.
- Ein `lock_guard` sollte deshalb entweder am Ende einer Funktion oder in einem eigenen Block definiert werden.

Unique Lock

Ein `unique_lock` verfügt über mehr Funktionalitäten als ein `lock_guard`. Es hat z.B. Funktionen um einen `mutex` zu sperren und wieder freizugeben oder um zu testen ob ein kritischer Abschnitt betreten werden kann, so dass andernfalls etwas anderes getan werden kann. Ein `unique_lock` kann auch einen bereits gesperrten `mutex` übernehmen oder das sperren erst einmal aufschieben.

```
void g(int &sh)
{
  std::unique_lock<std::mutex> locked{m,std::defer_lock};  // verwalte mutex, aber
      sperre ihn nicht
  bool successful=false;
  while (!successful)
  {
    if (locked.try_lock()) // sperre mutex wenn moeglich
    {
      sh+=1; // veraendere gemeinsame Daten
      successful=true;
      locked.unlock();     // in diesem Beispiel eigentlich nicht noetig
    }
    else
    {
      // mache etwas anderes
    }
  }
}

int main()
{
    const size_t numThreads=std::thread::hardware_concurrency();
    std::vector<std::thread> threads{numThreads};

    int result=0;
    // starte Threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i] = std::thread{e, std::ref(result)};

    // Wiedervereinigung mit dem Threads, implizite Barriere
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();

    std::cout << "Ihre␣Hardware␣unterstuetzt␣" << result << "␣Threads" << std::endl;
```

```
    return 0;
}
```

20.5 Berechnung der Vektornorm mit einem Mutex

```
#include<array>
#include<vector>
#include<iostream>
#include<thread>
#include<mutex>
#include<cmath>

static std::mutex nLock;

void Norm(const std::vector<double> &x, double &norm, const size_t i,
    const size_t p)
{
     size_t numElem = x.size()/p;
     size_t first = numElem * i + std::min(i,x.size()%p);
     numElem += (i<(x.size()%p)?1:0);
     double localNorm = 0.0;
     for (size_t j=0;j<numElem;++j)
         localNorm+=x[first+j]*x[first+j];

     std::lock_guard<std::mutex>
        block_threads_until_finish_this_job(nLock);
     norm += localNorm;
}

int main()
{
     const size_t numThreads = 5;
     const size_t numValues = 10000000;
     std::array<std::thread,numThreads> threads;
     std::vector<double> x(numValues,2.0);

     double norm = 0.0;
     // starte Threads
     for (size_t i = 0; i < threads.size(); ++i)
         threads[i] =
            std::thread(Norm,std::cref(x),std::ref(norm),i,numThreads);

     // Wiedervereinigung mit den Threads
     for (size_t i = 0; i < threads.size(); ++i)
         threads[i].join();

     std::cout << "Norm␣is:␣" << sqrt(norm) << std::endl;
     return 0;
}
```

20.6 Berechnung der Vektornorm mit Tree Combine

Parallelisierung der Summe

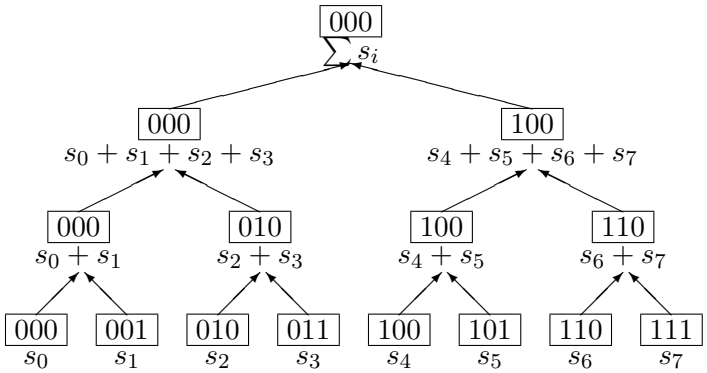
- Die Berechnung der globalen Summe der Norm ist bei Verwendung eines `mutex` für die Berechnung von $s = s + t$ nicht parallel.
- Sie lässt sich wie folgt parallelisieren ($P = 8$):

$$s = \underbrace{\underbrace{\underbrace{s_0}_{s_{01}} + \underbrace{s_1}_{s_{01}}}_{s_{0123}} + \underbrace{\underbrace{s_2}_{s_{23}} + \underbrace{s_3}_{s_{23}}}_{s_{0123}}}_{s} + \underbrace{\underbrace{\underbrace{s_4}_{s_{45}} + \underbrace{s_5}_{s_{45}}}_{s_{4567}} + \underbrace{\underbrace{s_6}_{s_{67}} + \underbrace{s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

- Damit reduziert sich die Komplexität von $O(P)$ auf $O(\log_2 P)$.

Tree Combine

Wenn wir eine Binärdarstellung der Prozessnummer verwenden, ergibt die Kommunikationsstruktur einen binären Baum:




```
#include <array>
#include <vector>
#include <iostream>
#include <thread>
#include <cmath>

void Norm(const std::vector<double> &x, std::vector<double> &norm,
    std::vector<bool> &flag,const size_t i, const size_t d)
{
    size_t p = pow(2,d);
    size_t numElem = x.size()/p;
    size_t first = numElem * i + std::min(i,x.size()%p);
    numElem += (i<(x.size()%p)?1:0);
    for (size_t j=0;j<numElem;++j)
        norm[i]+=x[first+j]*x[first+j];

    // tree combine
    for (size_t j=0;j<d;++j)
    {
        size_t m = pow(2,j);
        if (i&m)
        {
            flag[i]=true;
```

```
            break;
        }
        while (!flag[i|m]);
        norm[i] += norm[i|m];
    }
}

int main()
{
    const size_t logThreads = 1;
    const size_t numThreads = pow(2,logThreads);
    const size_t numValues = 10000000;
    std::array<std::thread,numThreads> threads;
    std::vector<double> x(numValues,2.0);
    std::vector<bool> flag(numThreads,false);

    std::vector<double> norm(numThreads,0.0);
    // starte Threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i] = std::thread(Norm,std::cref(x),std::ref(norm),
                                 std::ref(flag),i,logThreads);

    // Wiedervereinigung mit den Threads
    for (size_t i = 0; i < threads.size(); ++i)
        threads[i].join();

    std::cout << "Norm is: " << sqrt(norm[0]) << std::endl;
    return 0;
}
```

Condition Synchronisation

- Tree combine ist eine Form der Condition Synchronisation
- Ein Prozess wartet bis eine Bedingung (boolescher Ausdruck) wahr wird. Die Bedingung wird durch einen anderen Prozess wahr gemacht.
- Hier warten mehrere Prozesse bis ihre Flags wahr werden.
- Die richtige Initialisierung der Flags ist wichtig.
- Wir implementieren die Synchronisierung mit *busy wait*.
- Das ist wahrscheinlich keine besonders gute Idee bei multi-threading.
- Die Flags werden auch *condition variables* genannt.
- Wenn condition variables wiederholt verwendet werden (z.B. wenn mehrere Summen nacheinander berechnet werden müssen) sollten die folgenden Regeln befolgt werden:
  - Ein Prozess der auf eine condition variable wartet, setzt sie auch wieder zurück.
  - Eine condition variable darf nur dann erneut wieder auf true gesetzt werden, wenn es sichergestellt ist, dass sie vorher zurückgesetzt wurde.

20.7 Atomics

Diskrete Fourier Transformation

- Nehme Zahlenpaare (z_i, a_i) wobei a_i ein Messwert an der Stelle z_i ist.
- Berechne Interpolationspolynom

$$A(z) = \frac{1}{N}\left(f_0 + f_1 z + \cdots + f_{N-1} z^{N-1}\right)$$

- Dabei werden die N-ten Einheitswurzeln $z_i = e^{\frac{2\pi i}{N}k}$ als Stützstellen verwendet
- Die Koeffizienten dies Interpolationspolynoms ergeben sich als

$$f_k = \sum_{j=0}^{N-1} a_j \cdot e^{-2\pi i \frac{jk}{N}} \quad \text{für} \quad k = 0, \ldots, N-1$$

Fast Fourier Transformation

- Für ganzzahlige Werte von $N = 2n$ lässt sich die Summe

$$f_k = \sum_{j=0}^{2n-1} a_j \cdot e^{-2\pi i \frac{jk}{2n}} \quad \text{für} \quad k = 0, \ldots, 2n-1$$

  umsortieren in

$$f_k = \sum_{j=0}^{n-1} a_{2j} \cdot e^{-2\pi i \frac{2jk}{2n}} + \sum_{j=0}^{n-1} a_{2j+1} \cdot e^{-2\pi i \frac{k(2j+1)}{2n}}$$

Fast Fourier Transformation

- mit $a'_0 = a_0, a'_1 = a_2, a'_2 = a_4, \ldots,\ f'_0 = f_0, f'_1 = f_2, f''_2 = f_4, \ldots$ sowie $a''_0 = a_1, a''_1 = a_2, a''_2 = a_3$ und $f''_0 = f_1, f''_1 = f_3, f''_2 = f_5, \ldots$

$$\begin{aligned} f_k &= \sum_{j=0}^{n-1} a'_j \cdot e^{-\frac{2\pi i}{n}jk} + e^{-\frac{\pi i}{n}k} \sum_{j=0}^{n-1} a''_j e^{-\frac{2\pi i}{n}jk} \\ &= \begin{cases} f'_k + e^{-\frac{\pi i}{n}k} f''_k \text{ falls } k < n \\ f'_{k-n} + e^{-\frac{\pi i}{n}(k-n)} f''_{k-n} \text{ falls } k \geq n \end{cases} \end{aligned}$$

- Damit können wir das Problem jetzt durch Berechnung von zwei Fourier Transformationen der Länge $N/2$ lösen.
- Das lässt sich rekursiv anwenden. Da die Fourier-Transformierte eines Wertes der Wert selbst ist, müssen auf der untersten Stufe $N/2$ Summen aus jeweils zwei Werten berechnet werden.
- Da auf jeder Stufe N komplexe Multiplikationen mit der Einheitswurzel und N komplexe Additionen notwendig sind, lässt sich die Komplexität von $O(N^2)$ auf $O(N \cdot \log(N))$ senken.

Rekursiver Algorithmus im Pseudocode

```
procedure R_FFT(X, Y, n, w)
if (n==1) then Y[0] := X[0];
else begin
  R_FFT(<X[0],X[0],...,X[n-2]>,<Q[0],Q[1],...,Q[n-2]>,n/2,w^2);
  R_FFT(<X[1],X[3],...,X[n-1]>,<T[0],T[1],...,T[n-2]>,n/2,w^2);
  for i:= 0 to n-1 do
    Y[i] := Q[i mod (n/2)] + w^i T[i mod (n/2)];
end R_FFT
```

mit $w = e^{-\frac{2\pi\sqrt{-1}}{n}}$ aber: Rekursion schlecht parallelisierbar.

Iterativer Algorithmus im Pseudocode

```
procedure ITERATIVE_FFT(X, Y, N)
  r:= log n;
  for i := 0 to N-1 do R[i] := X[i];
  for m := 0 to r-1 do
    for i:= 0 to N-1 do S[i] := R[i];
    for i:= 0 to N-1 do
      /* Sei (b_0,b_1,...,b_r-1) die binaere Darstellung von i */
        j := (b_0,...,b_m-1,0,b_m+1,...,b_r-1);
        k := (b_0,...,b_m-1,1,b_m+1,...,b_r-1);
        R[i] := S[j] + S[k] x w^(b_m,b_m-1,...,b_0,0,...,0);
    endfor
  endfor
  for i := 0 to N-1 do Y[i] := R[i];
end ITERATIVE_FFT
```

mit $w = e^{-\frac{2\pi\sqrt{-1}}{n}}$

Sequentielle Implementierung der FFT

```
std::vector<std::complex<double>> fft(const std::vector<std::complex<double>> &data)
{
    const size_t numLevels=(size_t)std::log2(data.size());
    std::vector<std::complex<double>> result(data.begin(),data.end());
    std::vector<std::complex<double>> resultNeu(data.size());
    std::vector<std::complex<double>> root(data.size());
    for (size_t j=0;j<root.size();++j)
        root[j]=unitroot(j,root.size());
    for (size_t i=0;i<numLevels;++i)
    {
        size_t mask=1<<(numLevels-1-i);
        size_t invMask=~mask;
        for (size_t j=0;j<data.size();++j)
        {
            size_t k=j&invMask;
            size_t l=j|mask;
            resultNeu[j]=result[k]+result[l]*root[Reversal(j/mask,i+1)*mask];
        }
        if (i!=numLevels-1)
          result.swap(resultNeu);
    }
    return resultNeu;
}
```

Zurücksortierung der Werte durch Bit-Reversal

```
void SortBitreversal(std::vector<std::complex<double>> &result)
{
    std::vector<std::complex<double>> resultNeu(result.size());
    const size_t n = std::log2(result.size());
    for (size_t j=0;j<result.size();++j)
      resultNeu[Reversal(j,n)]=result[j];
    result.swap(resultNeu);
}

void FastSortBitreversal(std::vector<std::complex<double>> &result)
{
  size_t n = result.size();
  const size_t t = std::log2(n);
  size_t l = 1;
  std::vector<size_t> c(n);
  for (size_t q=0;q<t;++q)
  {
    n=n/2;
    for(size_t j=0;j<l;++j)
      c[l+j]=c[j]+n;
    l=2*l;
  }
  std::vector<std::complex<double>> resultNeu(result.size());
  for (size_t j=0;j<result.size();++j)
    resultNeu[c[j]]=result[j];
  result.swap(resultNeu);
}
```

Bit-Reversal und Einheitswurzeln

```
size_t Reversal(size_t k, size_t n)
{
  size_t j=0;
  size_t mask=1;
  if (k&mask)
    ++j;
  for (size_t i=0;i<(n-1);++i)
  {
    mask<<=1;
    j<<=1;
    if (k&mask)
      ++j;
  }
  return j;
}

inline std::complex<double> unitroot(size_t i, size_t N)
{
    double arg = -(i*2*M_PI/N);
    return std::complex<double>(cos(arg),sin(arg));
}
```

Hauptprogramm Sequentielle FFT

```
int main()
{
    const size_t numPoints = pow(2,16);
    std::vector<std::complex<double>> data(numPoints);
    size_t i=1;
    for (auto &x : data)
      x.real(cos(i++));
    auto t0 = std::chrono::steady_clock::now();
```

```
    std::vector<std::complex<double>> result1 = fft(data);
    FastSortBitreversal(result1);
    auto t1 = std::chrono::steady_clock::now();
    for (auto &x : result1)
        std::cout << std::abs(x) << " " << x.real() << " " << x.imag()<< std::endl;
    std::cout << std::endl << std::endl;

    std::cout << "# Sequential fft took " <<
        std::chrono::duration_cast<std::chrono::milliseconds>(t1-t0).count() << "
        milliseconds." << std::endl;
}
```

Zeitmessung

Zur Zeitmessung müssen bisher Systembibliotheken verwendet werden. C++11 bietet eine integrierte Möglichkeit zur Zeitmessung an. Dabei gibt es verschiedene Uhren und Datentypen um Zeitpunkte und Zeitspannen zu speichern.

```
#include<chrono>
#include<iostream>
#include<thread>
using namespace std::chrono;

int main()
{
  steady_clock::time_point start = steady_clock::now();
  std::this_thread::sleep_for(seconds{2});
  auto now = steady_clock::now();
  nanoseconds duration = now-start; // we want the result in ns
  milliseconds durationMs = duration_cast<milliseconds>(duration);
  std::cout << "something took " << duration.count()
            << " ns which is " << durationMs.count() << " ms\n";
  seconds sec = hours{2} + minutes{35} + seconds{9};
  std::cout << "2 h 35m 9s is " << sec.count() << " s\n";
}
```

Hilfe zum Debuggen: Ausgabe von Binärzahlen

```
std::string convBase2(size_t v)
{
        std::string digits = "01";
        std::string result;
  do {
                result = digits[v % 2] + result;
                v /= 2;
        }       while(v);

        return result;
}
```

Parallele FFT: Threads

```
Barrier barrier;

void fftthread(const size_t threadNum, std::vector<std::complex<double>> &root,
    std::vector<std::complex<double>> &result, std::vector<std::complex<double>>
    &resultNeu)
{
```

```
    const size_t N = std::log2(result.size());
    size_t n = result.size()/numThreads;
    const size_t offset = threadNum*n + std::min(threadNum,result.size()%numThreads);
    n += (threadNum<(result.size()%numThreads)?1:0);
    for (size_t i=offset;i<offset+n;++i)
        root[i]=unitroot(i,root.size());
    barrier.block(numThreads);
    for (size_t level=0;level<N;++level)
    {
        size_t mask=1<<(N-level-1);
        size_t invMask=~mask;
        for (size_t i=0;i<n;++i)
        {
            size_t j=offset+i;
            size_t k=j&invMask;
            size_t l=j|mask;
            size_t e = Reversal(j/mask,level+1)*mask;

            std::cout << "#␣j:␣" << j << "␣k:␣" << k << "␣l:␣" << l << "␣e:␣" << e <<
                std::endl;
            resultNeu[j]=result[k]+result[l]*root[Reversal(j/mask,level+1)*mask];
        }
        barrier.block(numThreads);
        if (threadNum==0)
            result.swap(resultNeu);
        barrier.block(numThreads);
    }
    for (size_t j=offset;j<offset+n;++j)
      resultNeu[Reversal(j,N)]=result[j];
}

std::vector<std::complex<double>> fft(const std::vector<std::complex<double>> &data)
{
    std::vector<std::complex<double>> result(data.begin(),data.end());
    std::vector<std::complex<double>> resultNeu(data.size());
    std::vector<std::complex<double>> root(data.size());
    std::vector<std::thread> t(numThreads);

    for (size_t p = 0;p<numThreads;++p)
        t[p]=std::thread
            {fftthread,p,std::ref(root),std::ref(result),std::ref(resultNeu)};

    for (size_t p = 0;p<t.size();++p)
        t[p].join();
```

Barrier

```
const size_t numThreads=1;//std::thread::hardware_concurrency();

struct Barrier
{
    std::atomic<size_t> count_;
    std::atomic<size_t> step_;
    Barrier() : count_(0), step_(0)
    {}

    void block(size_t numThreads)
    {
        if (numThreads<2)
            return;
        size_t step = step_.load();
        if (count_.fetch_add(1) == (numThreads-1))
        {
            count_.store(0);
            step_.fetch_add(1);
```

```
            return;
        }
        else
        {
            while (step_.load() == step)
              std::this_thread::yield();
            return;
        }
    }
};
```

Hauptprogramm Parallele FFT

```
    return resultNeu;
}

int main()
{
    const size_t numPoints = pow(2,2);
    std::vector<std::complex<double>> data(numPoints);
    size_t i=1;
    for (auto &x : data)
      x.real(cos(i++));
    auto t0 = std::chrono::steady_clock::now();
    std::vector<std::complex<double>> result1 = fft(data);
    auto t1 = std::chrono::steady_clock::now();
    for (auto &x : result1)
        std::cout << std::abs(x) << "␣" << x.real() << "␣" << x.imag()<< std::endl;
    std::cout << std::endl << std::endl;
```

20.8 Threaderzeugung mit `async`

Mit Hilfe der Funktion `async` lässt sich einfach ein Thread erzeugen, dessen Ergebnis später mit Hilfe der Funktion `get` abfragen lässt.

```
#include<iostream>
#include<vector>
#include<numeric>
#include<thread>
#include<future>

template<class T> struct Accum  {  // simple accumulator function object
  T *b;
  T *e;
  T val;
  Accum(T *bb, T *ee, T vv) : b{bb}, e{ee}, val{vv} {}
  T operator() () { return std::accumulate(b,e,val); }
};
```

Mit Hilfe der Funktion `async` lässt sich einfach ein Thread erzeugen, dessen Ergebnis später mit Hilfe der Funktion `get` abfragen lässt.

```
double comp(std::vector<double> &v)
  // spawn many tasks if v is large enough
{
  if (v.size()<10000)
    return std::accumulate(v.begin(),v.end(),0.0);
  std::future<double> f0 {std::async(Accum<double>{&v[0],&v[v.size()/4],0.0})};
  std::future<double> f1
      {std::async(Accum<double>{&v[v.size()/4],&v[v.size()/2],0.0})};
  std::future<double> f2
      {std::async(Accum<double>{&v[v.size()/2],&v[v.size()*3/4],0.0})};
```

```
  std::future<double> f3
      {std::async(Accum<double>{&v[v.size()*3/4],&v[v.size()],0.0})};

  // hier koennte noch ganz viel anderer Code kommen...

  return f0.get()+f1.get()+f2.get()+f3.get();
}

int main()
{
  std::vector<double> blub(100000,1.);
```

20.9 Weiterführende Literatur

Literatur

[1] C++11 multi-threading Tutorial http://solarianprogrammer.com/2011/012/16/cpp-11-thread-tutorial

[2] Übersicht aller C++11 Thread Klassen und Funktionen http://en.cppreference.com/w/cpp/thread

[3] Übersicht aller C++11 atomic Anweisungen http://en.cppreference.com/w/cpp/atomic

[4] Working draft des C++11 Standards (nahezu identisch mit dem endgültigen Standard aber kostenlos) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf

Literatur

[Str13] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.

[Unr94] E. Unruh. Prime number computation. Ansi x3j16-94-0075/iso wg21-462, ANSI / ISO, 1994.

[Vel95] Todd L. Veldhuizen. Using c++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.