Objektorientiertes Programmieren im Wissenschaftlichen Rechnen

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

21. Juli 2010

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
- Expression Templates
- Neuerungen durch den C++0x-Standard
- (Python-Wrapping für C++-Funktionen)

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	3333	6	4'096	2'000'000	437 (2.6.33.2)
2010	AMD 6174	2200	12	4'096	2'000'000	437 (2.6.33.2)

Beispiel DUNE

- Framework für die Lösung Partieller Differentialgleichungen
- entwickelt von Arbeitsgruppen an den Universitäten Freiburg, Heidelberg, Münster und der Freien Universität Berlin

- 8 Core Developer, viele weitere Entwickler
- zur Zeit 147'000 Zeilen Programmcode (plus 38'000 Zeilen Kommentare)
- Anwender an vielen anderen Universitäten
- verwendet intensiv moderne C++-Konstrukte die in dieser Vorlesung vorgestellt werden

Programmierparadigmen

- Funktionale Programmierung
  - 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 Variable 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 werden

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 int` für ganze Zahlen

- `bool` für logische Zustände
- `std::string` für Zeichenketten

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 List 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);
    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.

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.

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.

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

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

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.
- 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=0;  // 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 Element in eckigen Klammern hinter den Variablentyp schreibt.
- Um Felder zu allozieren braucht eine Klasse einen argumentlosen Konstruktor.
- Felder werden mit `delete []` gelöscht.

```
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;
}
```

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 konstante 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

```
        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)
```

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<dou >>`.
- 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 der 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 automatisch zur Klasse dazu, da sie auf der 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 Funnktionsumfang:

- 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` is a program which makes it possible to compile only the files which are affected by the change
- A `Makefile` describes which files belong to a project and how they are 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 Schlüssword `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 "verseucht".
  - 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 Welchselspiel 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

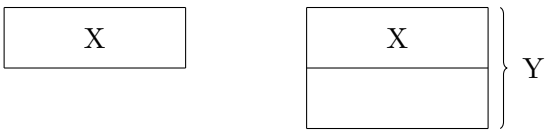
```
class X
{
  public:
    void a();
};

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	–	–	–

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.

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:

- 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.

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, 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 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 beendet, dann ist nicht garantiert, dass die Destruktoren der Objekte aufgerufen werden.

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.
- 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.
- Alle von der Standardbibliothek geworfenen Ausnahmen sind von der Klasse `exception` abgeleitet.

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
    }
}
```

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.

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. Xerox Cedar/Mesa system. $\tilde{5}$00.000 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.

10 Dynamischer Polymorphismus

10.1 Virtuelle Funktionen

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

Beispiel

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

- 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 Funktionspointers auf) `void less(int,int)` und `void 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, sind sie in der Regel schneller.

10.2 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.
- `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.

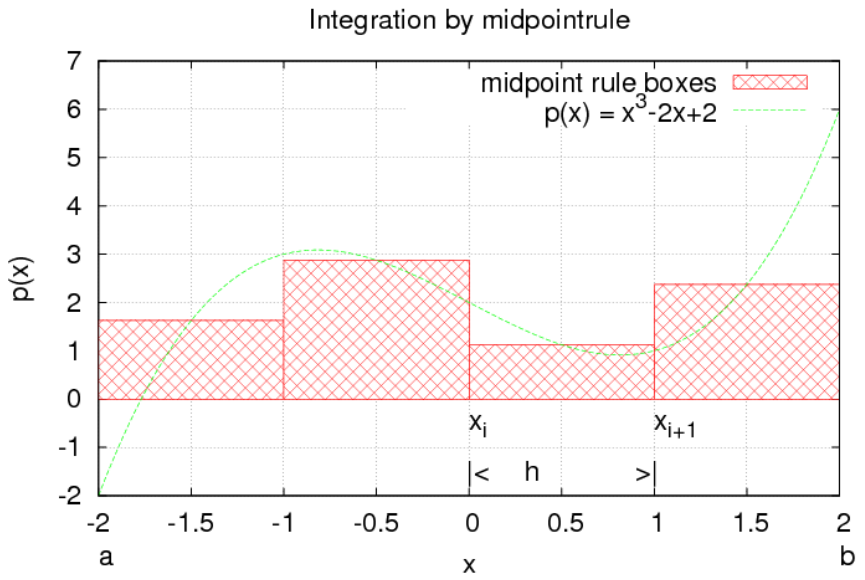


Abbildung 1: Anwendung der Mittelpunktsregel mit $n = 4$ für $p(x) = x^3 - 2x + 2$.

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"

// 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)
    {}
    virtual double operator()(double x) {
        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)
    {
        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 "mittelpunkt_class.h"
#include "simpson_class.h"
#include "cosinus.h"

int main()
```

```
{
  Cosinus cosinus(1.0,-1.0);
  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;
  delete integrate;
  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 an verschiedenen Orten die eine Simulation verwendet.
- 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 initialisiert werden.

```
std::vector<Funktor *> Funktion(4,0);
Funktion[0] = new Cosinus(1.0,-1.0);
...
for (size_t i=0;i<Funktion.size();++i)
{
    if (Funktion[i]!=0)
        delete Funktion[i];
}
```

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()
    {};
};
```

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 0, wenn die Konvertierung nicht möglich ist.
- `dynamic_cast` funktioniert auch mit Referenzen:

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

  Wenn die Konvertierung nicht durchgeführt werden kann, dann wird eine Ausnahme vom Typ `std::bad_cast` geworfen.

```
#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 SimpsonRegel(f,a,b);
    else
        return MittelpunktsRegel(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);
    }
}
```

10.3 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 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.

UML Diagrammarten

UML stellt 9 verschiedene Arten von Programmen bereit:

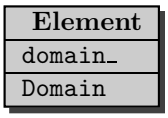
1. Class diagram
2. Object
3. Use case
4. Sequence

5. Collaboration
6. Statechart
7. Activity
8. Component
9. Deployment

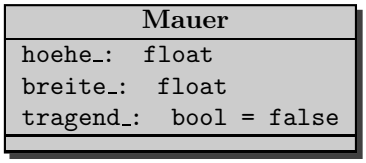
Wir behandeln nur einen winzigen Ausschnitt aus diesem umfangreichen Werkzeugset.

Klassen

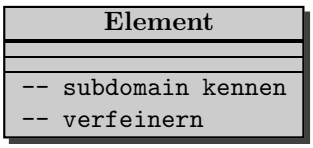
- Klassen werden durch einen rechteckigen Kasten dargestellt, der durch horizontale Linien in verschiedene Abschnitte unterteilt ist.
- Zuerst kommt der Name der Klasse, dann ihre Attribute und dann ihre Methoden.



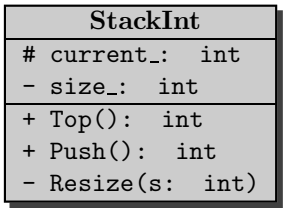
- Attribute können einen Typ und gegebenenfalls einen default-Wert haben.



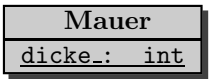
- Klassen können als vierten Abschnitt auch "Responsibilities" tragen:



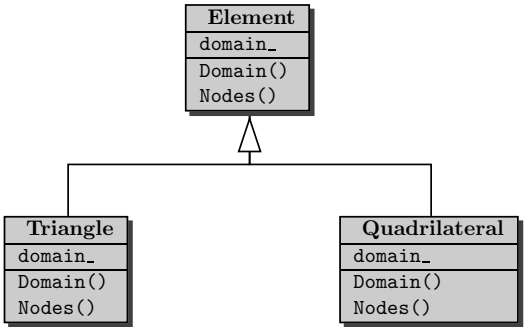
- Die Zugriffsrechte auf Attribute werden durch die vorangestellten Zeichen `+` für `public`, `#` für `protected` und `-` für `private` gekennzeichnet.



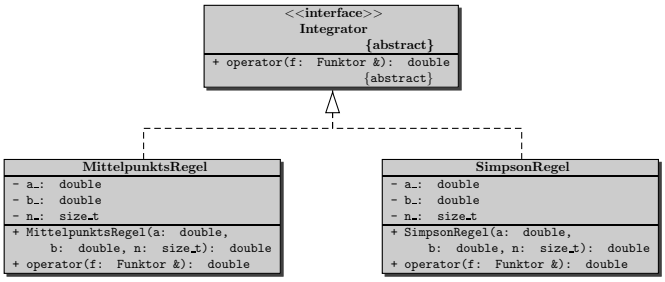
- Statische Klassenelemente werden durch Unterstreichen gekennzeichnet.



- 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:

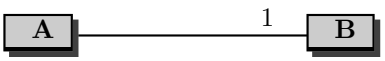


- Schnittstellenbasisklassen werden durch die Zeile <<interface>> gekennzeichnet.
- Die Verbindung zu Klassen die das Interface implementieren ist wie eine Vererbung aber gestrichelt.
- Abstrakte Basisklassen und rein virtuelle Funktionen haben die Eigenschaft {abstract}

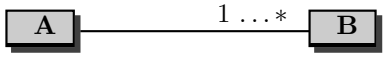


Die Assoziation zwischen zwei Elementen wird durch eine verbindende Linie dargestellt. Zahlen an der Linie können die Anzahl der Verbindungen und einen Namen für die Verbindung angeben.

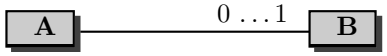
A ist assoziiert mit einem B.



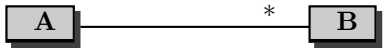
A ist assoziiert mit einem oder mehreren B.



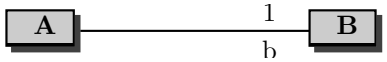
A ist assoziiert mit keinem oder einem B.



A ist assoziiert mit keinem, einem oder mehreren B.



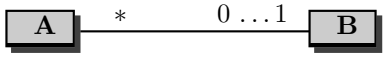
Eine Assoziation kann auch einen Namen haben:



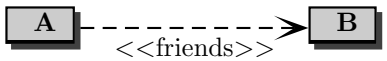
z.B.

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

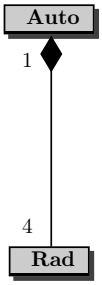
und es gibt auch Assoziationen in beide Richtungen:



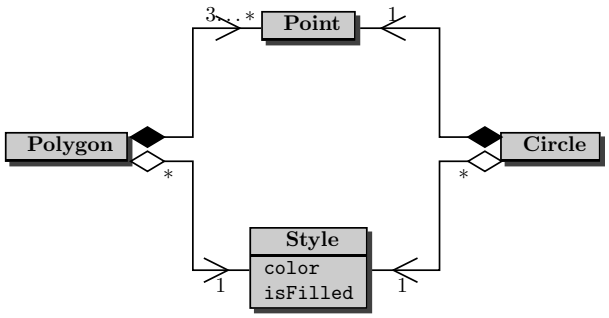
Eine Klasse kann von einer anderen Abhängen, z.B. weil sie ein `friend` ist:



Komposition wird durch verbindende Linie mit einer gefüllten Raute auf der Seite der zusammengesetzten Klasse dargestellt.



Aggregation wird durch verbindende Linie mit einer leeren Raute auf der Seite der aggregierten Klasse dargestellt.



12 Statischer Polymorphismus

12.1 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.

12.2 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.

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
  ::max(7.0, 42.0);    // calss 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 `int c = std::max(a,b);`
- `void std::swap(T &, T &)` Vertauscht den Inhalt von a und b `std::swap(a,b);`

12.3 Klassentemplates

- Es ist oft hilfreiche 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;
    }
}
```

Spezialisierung von Klassentemplates

- Auch Klassentemplates lassen sich für bestimmte Argumentwerte spezialisieren, wenn 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*>
```

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 zusätzlichen 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;
    }
}
```

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.

12.4 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
```

12.5 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.

12.6 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

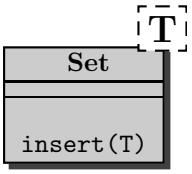
⇒ Statischer Polymorphismus ist besonders geeignet, wenn viele kurze Funktionsaufrufe benötigt werden (wie z.B. der Zugriff auf Matrixelemente ...)

12.7 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.

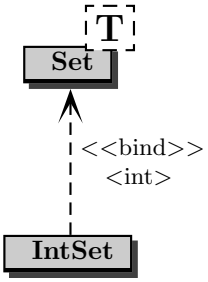
12.8 Templates in UML

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



Realisierung von Set mit `T=int`.

oder



12.9 Template Besonderheiten

12.9.1 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 List von Basisklassenspezifikationen oder einer Initialisierungsliste.

12.9.2 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 "->"

12.9.3 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 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
}
```

12.9.4 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
}
```

12.9.5 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
      {
      }
      ...
  };
  ```

12.9.6 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 Name 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 Name 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 Basisklassenattribute 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)
};
```

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.

Abbildung 2: Struktur eines `vector`

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.

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. 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).

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)
```

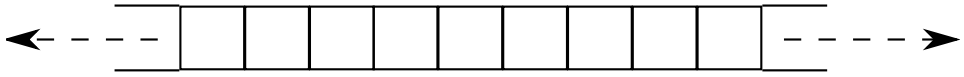


Abbildung 3: Struktur einer `deque`

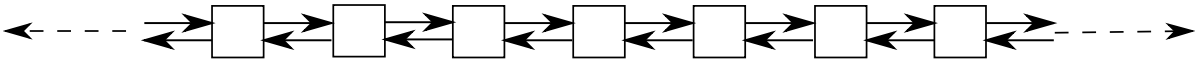


Abbildung 4: Struktur einer `list`

```
        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);
```

```
    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 << vals.front() << std::endl;
      vals.pop_front();
    }
    std::cout << std::endl;
    for (int i=0;i<copy.size();++i)
    {
      std::cout << copy.back() << std::endl;
      copy.pop_back();
    }
}
```

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 von Paaren aus zwei Werten, einem Schlüssel und einem Wert. Die Map ist 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.
- Die 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`).

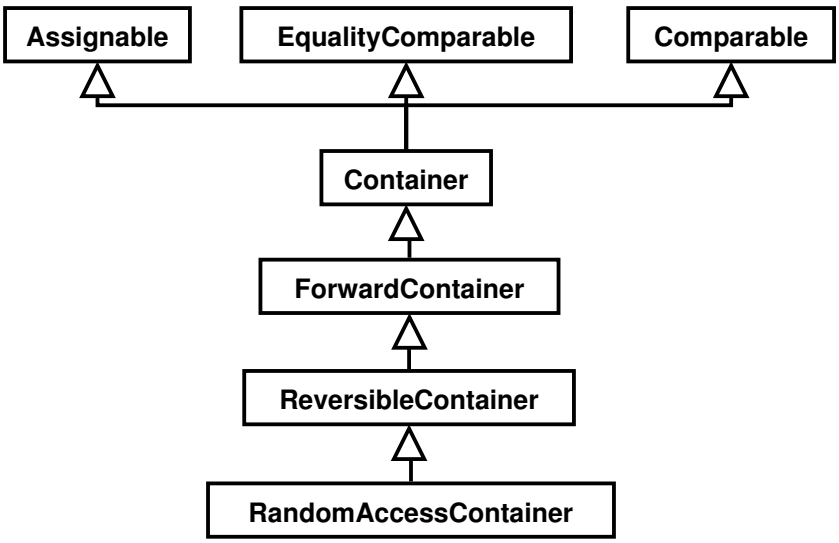


Abbildung 5: Container Konzepte

`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

- 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`

`Sequence`

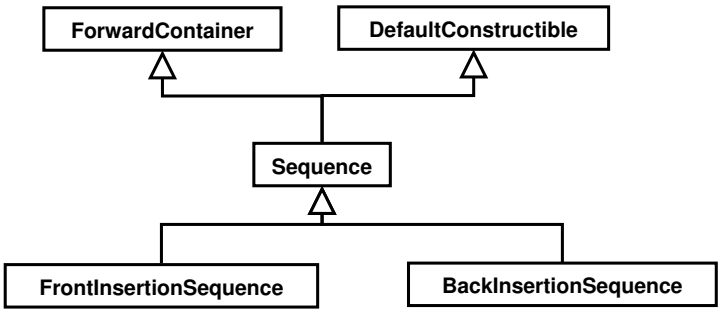


Abbildung 6: Sequenz Konzepte

`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

Assoziative Container

`AssociativeContainer`

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

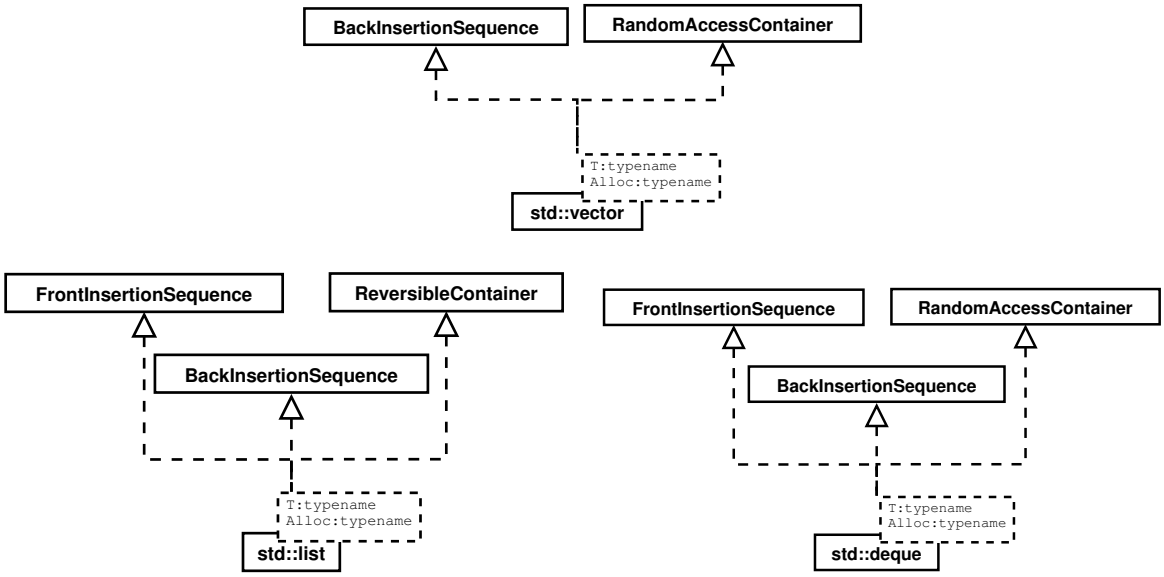


Abbildung 7: STL Sequenzcontainer

`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

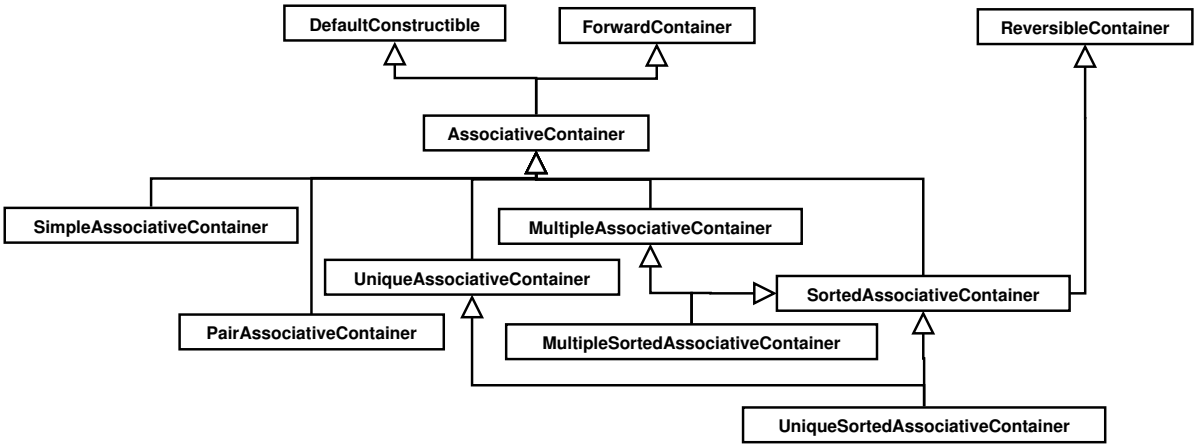


Abbildung 8: Assoziative Container Konzepte

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)==k` 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

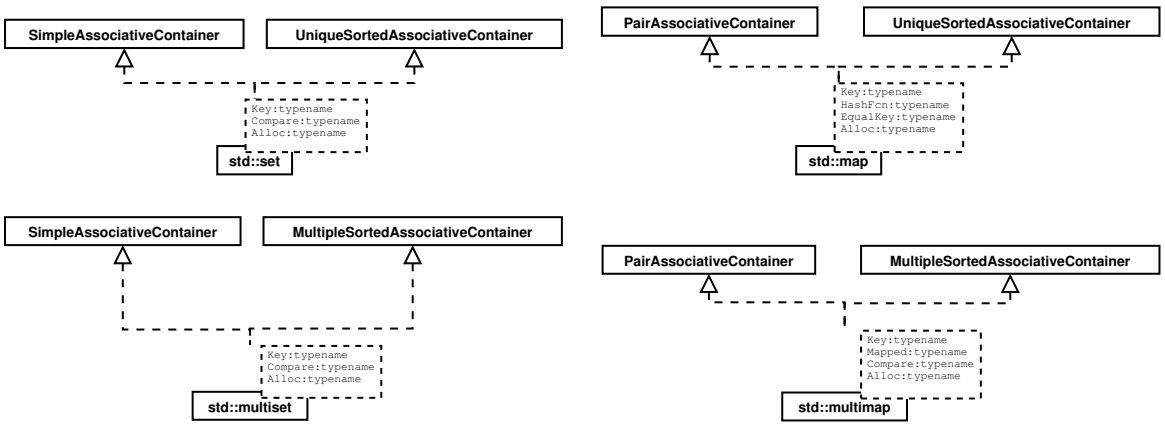


Abbildung 9: Assoziative Containerklassen

	`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

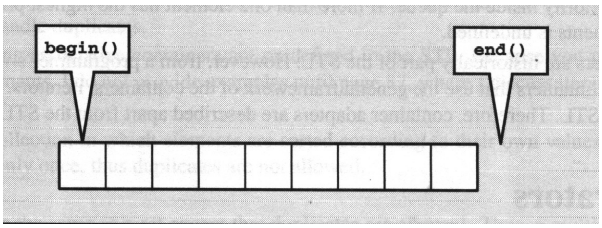


Abbildung 10: Iterator über einen Container

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

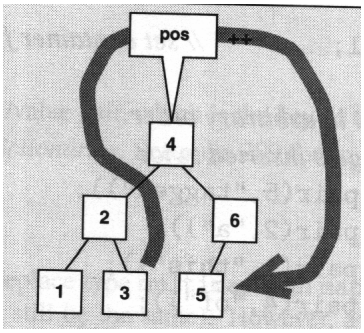


Abbildung 11: Iterator über ein Set

- 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.

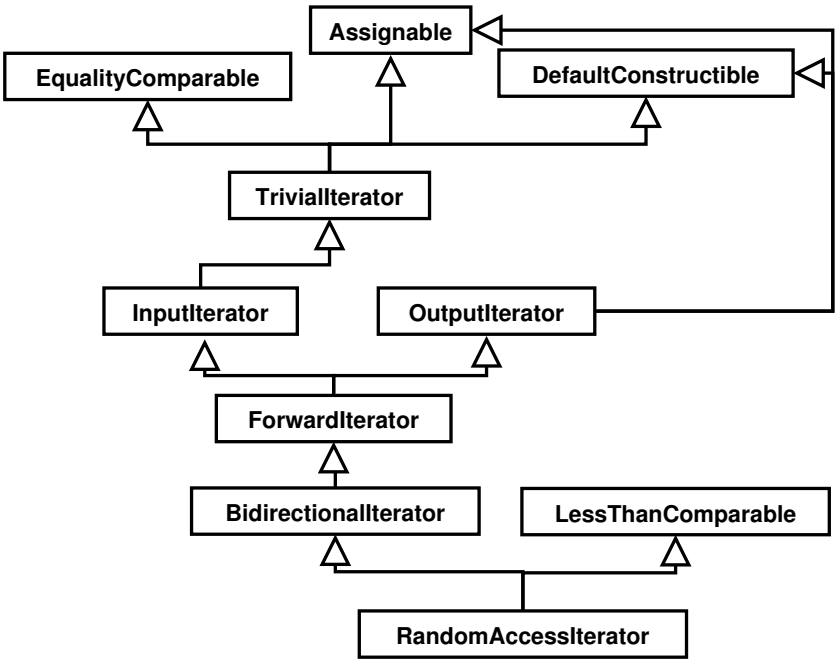


Abbildung 12: Iterator Konzepte

`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..
- Die Menge 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 zusätzliches Suffix gekennzeichnet werden.

_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 werden 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 Suffix `_copy_if`

Nicht verändernde Algorithmen

`for_each`	Performs operation for each elements	
`count`	Counts the number of elements equal to a value	`_if`
`min_element`	Returns the smallest element	
`max_element`	Returns the largest element	

Suchalgorithmen

`find`	Search for first element with the passed value	`_if`
`search_n`	Searches for the first n consecutive elements with certain properties	
`search`	Searches for the first occurrence of the passed subrange	
`find_end`	Search for last occurrence with the passed subrange	
`find_first_of`	Search for the first of several possible elements (passed as range)	
`adjacent_find`	Searches for two adjacent elements that are equal	
`equal`	Returns whether two ranges are equal	
`mismatch`	Returns the first elements of two sequences that differ	
`lexicographical_compare`	Returns whether a range is lexicographically less than another range.	

Verändernde Algorithmen

`for_each`	Perform an operation for each element	
`copy`	Copy a range starting with the first element	
`copy_backward`	Copy a range starting with the last element	
`transform`	Modifies (and copies) elements; combines elements of two ranges	
`merge`	Merges two ranges	
`swap_ranges`	swaps elements of two ranges	
`fill`	Replace all elements with a given value	
`fill_n`	Replace n elements with a given value	
`generate`	Replace all elements with the result of a given operation	
`generate_n`	Replace$n$ elements with the result of a given operation	
`replace`	Replace all elements that have a special value with another value	`_if`, `_copy`, `_copy_if`

`transorm` 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>
void multAssign(T& elem)
{
  elem=std::bind1st(std::multiplies<T>(),10)(elem);
}

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(),
                 std::bind1st(std::multiplies<int>(),10));
  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`	Remove elements with a given value	`_if`,`_copy`, `_copy_if`
`unique`	Removes adjacent duplicates	`_copy`

Tabelle 3: Removing Algorithms

- Elements are removed by overwriting them with following elements that were not removed.
- New end iterator is returned. May be used to remove elements physically.

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`	Reverse the order of the elements	`_copy`
`rotate`	Rotates the order of the elements	`_copy`
`next_permutation`	Permutes the order of the elements	
`prev_permutation`	Permutes the order of the elements	
`random_shuffle`	Brings the elements into random order	
`partition`	Changes the order of the elements such that elements that match a criterion are in front.	
`stable_partition`	Same as `partition` but preserves the order of matching and non-matching elements	

Verändernde Algorithmen and 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`	Sort all elements (based on quicksort)	
`stable_sort`	Sorts while preserving order of equal elements (based on mergesort)	
`partial_sort`	Sorts until the first n elements are correct (based on heapsort)	`_copy`
`n_th_element`	Sorts according to the nth element	
`partition`	Changes the order of the elements such that elements that match a criterion are in front.	
`stable_partition`	Same as `partition` but preserves the order of matching and non-matching elements	
`make_heap`	Convert a range into a heap	
`push_heap`	Adds an element to a heap	
`pop_heap`	Removes an element from a heap	
`sort_heap`	Sorts the heap (it is no longer a heap after the call)	

Algorithmen für sortierte Bereiche

`binary_search`	Returns whether a range contains an element
`includes`	Returns whether each element of range is also an element of another range
`lower_bound`	Finds the first element greater or equal to a given value.
`upper_bound`	Finds the first element greater than a given value
`equal_range`	Returns the range of elements equal to a given value
`merge`	Merges two ranges
`set_union`	Processes the sorted union of two ranges
`set_intersection`	Processes the sorted intersection of two ranges
`set_difference`	Processes a a sorted range that contains all elements of a range that are not part of another
`set_symmetric_difference`	Processes a sorted range that contains all elements that are exactly in one of two ranges
`inplace_merge`	Merges two consecutive sorted ranges

Numerische Algorithmen

`accumulate`	Combines all element values using a given binary function or `plus` by default
`inner_product`	Combines all elements of two ranges (using two optional binary function objects
`adjacent_difference`	Combines all elements with its predecessor (using `minus` or optional function object
`partial_sum`	Combines all elements with all of its predecessors (using `plus` or optional function object

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 Generic Programming

14.1 Building Blocks of Generic Programming

Generic Programming

Generic programming is a style of computer programming where algorithms are written in terms of unknown types that are then somehow instantiated later by the compiler. (In C++ this id done with templates.)

- A methodology for the development of reusable software libraries.
- Three primary tasks:
  - Categorise the abstractions in a domain into concepts.
  - Implement generic algorithms based on these concepts.
  - Build concrete models of the concepts.

Generic Programming Methodology

Lifting seeks to discover a generic algorithm by answering: What are the minimal requirements that data types need to fulfil for the algorithm to operate correctly and efficiently?

Concepts bundle together coherent sets of requirements into a single entity. They describe families of abstractions based on what these abstractions can do.

Models are data types that implement the concepts

Lifting

- Iterative process.
- Start with multiple, concrete implementations of the same algorithm.
- Identify the functionality needed for the data types.
- Write a common generic algorithm that only uses the identified functionality.
- Maybe start over for a new data type

Lifting Example: Concrete Implementations

```
int sum(int *ar, int n){
  int result=0;
  for(int i=0; i < n; ++i)
    result = result+ar[i];
  return result;
}

double sum(double *ar, int n){
  double result=0;
  for(int i=0; i < n; ++i)
    result = result+ar[i];
  return result;
}
```

- The two functions are nearly identical.
- One works on integral types, the other on floating point types
- We can lift these algorithm to create a single generic algorithm which works for both `int` and `float`.

Lifting Example: Generic Algorithm

```
template<typename T>
T sum(T* array, int n){
  T result = 0;
  for(int i=0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Requirements for type `T`:

- Initialisation with zero.
- Addition of two values of type `T`.
- Assignment of two values of type `T`.
- `T` must have a copy constructor.

Lifting Example: Further Lifting

There might be scenarios requiring further lifting iterations, e.g:

```
template<typename T>
T sum(T* ar, int n){
  T result = 0;
  for(int i=0; i < n; ++i)
    result = result + ar[i];
  return result;
}
```

```
std::string concatenate(std::string* ar, int n){
  std::string result = "";
  for (int i=0; i < n; ++i){
    result = result + ar[i];
  return result;
}
```

- Algorithms are nearly identical,
- but generic algorithm expects numeric types, initialisable to zero
- while concatenate algorithm operates on strings and is initialised with the empty string.
- We need to abstract away the initialisation value, such that numeric types are initialised to zero and strings to the empty string.
- In theses cases the default constructor does the right thing!

Lifting Example: Lifted Generic Algorithm

```
template<typename T>
T sum(T* array, int n){
  T result = T();
  for(int i=0; i<n; ++i)
    result = result + array[i];
  return result;
}
```

Requirements:

- `T` must have a default constructor that produces the identity value.
- `T` must have an additive operator+.
- `T` must have an assignment operator.
- `T` must have a copy constructor.

Lifting Containers

- Algorithm supports summation of numeric values, concatenation of strings, etc.
- Currently our algorithm only works for arrays of various types.
- What about containers, such as `std::vector`?
- If they provide `operator[]`, we only need to adapt our algorithm a little bit:

```
template<typename Cont, typename T>
T sum(Cont& array, int n){
  T result = T();
  for(int i=0; i<n; ++i)
    result = result + array[i];
  return result;
}
```

Requirements

- `T` must have a default constructor that produces the identity value.
- `T` must have an additive operator+.
- `T` must have an assignment operator.
- `T` must have a copy constructor.
- `Cont` must have an indexing `operator[]` that returns a `T`

What about lists?

- Consider this very simple list:

```
template<typename T>
struct ListNode{
  ListNode(const T& t)
    : value(t), next(0)
  {}

  T value;
  ListNode* next;
};

template<typename T>
struct List{
  ListNode<T>* values;
};
```

- Implementing an `operator[]` to fulfil the requirements of our generic algorithm is possible,
- but results in being a linear time operation.
- Algorithm is correct but not efficient.
- Our abstraction is either poor or too much.
- Lets take another look.

Lifting Iteration

```
template<typename T>
T sum(T* ar, int n){
  T result = T();
  for(int i=0; i<n; ++i)
    result = result + ar[i];
  return result;
}

template<typename T>
T sum(List<T> list, int n) {
  T result = T();
  for (ListNode<T>* current = list.values;
       current != 0; current = current->next)
    result = result + current->value;
  return result;
}
```

- Much more differences than before!
- The container does not get referenced once the values are extracted.
- Maybe the container is not a good candidate for the core abstraction?

Iterating over a sequence

Let us modify our algorithm to use pointer arithmetic:

```
template<typename T>
T sum(T* ar, int n){
  T result = T();
  for(T* cur=ar; cur != array + n; ++cur)
    result = result + *cur;
  return result;
}
```

and compare it with sum over the list, again:

```
template<typename T>
T sum(List<T> list, int n) {
  T result = T();
  for (ListNode<T>* current = list.values;
       current != 0; current = current->next)
    result = result + current->value;
  return result;
}
```

- Much more similarities now.
- Still we need a generic way to
  - get the end of the sequence or let the user provide it,
  - move to the next sequence member,
  - and get the current value in the sequence.

Free Functions

We can provided the needed additional functionality with free functions as opposed to member functions:

```
template<typename T>
T* next(T* t){
  return ++t;
}
template<typename T>
ListNode<T> next(ListNode<T>* n) {
  return n->next;
}

template<typename T>
T get(T* p) {
  return *p;
}
template<typename T>
T get(ListNode<T>* n){
  return n->value;
}
```

This is noninvasive and lets us use third party data types with our generic algorithms.

Lifted Iteration Version

```
template<typename I, typename T>
T sum(I start, I end, T init){
  for(I cur=start; cur !=end; cur = next(cur))
    init = init + get(cur);
  return init;
}
```

Requirements

- `T` must have an additive operator `+`.
- `T` must have an assignment operator.
- `T` must have a copy constructor.
- `I` must have an inequality operator `!=`.
- `I` must have a copy constructor.
- `I` must have an operation `next()` moving to the next sequence value.
- `I` must have an operation `get()` that returns the current value convertible to type `T`

Concepts

A *concept* describes a set of requirements that a template parameter must meet for the template function or class template to compile and operate properly.

- Concepts describe a family of related abstractions based on what these abstractions can do.

- Concepts are discovered through the process of lifting.
- E.g. `Iterator` concept describing abstraction for iterators or a `Shape` abstraction describing abstractions for shapes (polygon, circle, ...)

Sum Algorithm Requirements

```
template<typename I, typename T>
T sum(I start, I end, T init){
  for(I cur=start; cur !=end; cur = next(cur))
    init = init + get(cur);
  return init;
}
```

Requirements

- `T` must have an additive operator `+`.
- `T` must have an assignment operator.
- `T` must have a copy constructor.
- `I` must have an inequality operator `!=`.
- `I` must have a copy constructor.
- `I` must have an operation `next()` moving to the next sequence value.
- `I` must have an operation `get()` that returns the current value of type `T`

Packaging Requirements into Concepts

- There are many ways to package requirements into concepts, e.g. the two extreme cases:
  - Package all requirements into one concept.
  - Make each requirement a concept on its own.
- In general the correct (optimal) solution lies in between of course:
  - The concept should have enough requirements to give some common identity to the abstraction it describes.
  - The concept number of requirements should not unnecessarily restrict the family of abstractions.
- Packaging requirements relies on the lifting process again.

Another Example Algorithm

Consider searching for a specific value in a sequence:

```
template<typename I, typename T>
I find(I start, I end, T value){
  for(I cur=start; cur != end; ++cur)
    if(get(cur)==value)
      return cur;
  return end;
}
```

Requirements

- `T` must have an equality operator `==`
- `T` must have a copy constructor
- `I` must have an inequality operator `!=`
- `I` must have a copy constructor.
- `I` must have an operation `next()` moving to the next sequence value.
- `I` must have an operation `get()` that returns the current value of type `T`

Comparison of Requirements

Differences

- Only `sum` requires `operator+` for `T`
- Only `find` requires `operator==` for `T`

Similarities

- Both require a copy constructor for type `T`.
- Requirements on `I` are identical.

Concept Requirement Mapping

Concept	Requirements
`CopyConstructible<T>`	`T` must have a copy constructor.
`Assignable<T>`	`T` must have an assignment operator.
`Addable<T>`	`T` must have an `operator+` that take tow `T` values and returns `T`
`EqualityComparable<T>`	`T` must have an `operator==` comparing two `T`s and returning a `bool`. `T` must have an `operator!=` comparing two `T`s and returning a `bool`.
`Iterator<I, T>`	`I` must have an `operator!=` returning a bool `I` must have an `operator==` returning a bool `I` must have a copy constructor. `I` must have an operation `next()` moving to the next sequence value. `I` must have an operation `get()` that returns the current value of type `T`

Nested Requirements

- Our requirements are redundant:
  - The `Iterator<I,T>` concept requires `I` to have `operator==` and `operator!=`,
  - `I` to have a copy constructor,
  - and `I` to have an assignment operator.
- We already have specified these concepts.
- We can reuse them by nesting them.

A *nested requirement* is when a concept references another concepts as one of its own requirements.

Concept	Requirements

`Iterator<I,T>`	`EqualityComparable<I>`, `CopConstructable<I>`, `Assignable<I>`
	`I` must have an operation `next()` moving to the next sequence value.
	`I` must have an operation `get()` that returns the current value of type `T`

Associated Types

Consider an example function

```
/* Requirements: Iterator<I,T> */
template<typename I, typename T>
int distance(I start, I end){
  int i = 0;
  for(; start != end; ++start)
    ++i
  return i;
}
```

- No reference of `T` in the whole function.

- `distance()` hard to use as `T` has to be explicitly specified.
- `T` is implicitly known if `I` is known.
- Use *associated types* to store it in the concept definition.

Concept with Associated Types

Concept	Requirements

`Iterator<I>`	`EqualityComparable<I>`, `CopConstructable<I>`, `Assignable<I>`
	`I` must have an operation `next()` moving to the next sequence value.
	`value_type` is an associated type, accessible via `iterator_traits<I>::value_type`
	`I` must have an operation `get()` that returns the current value of type `value_type`

Using this concept leads to the following algorithm specification:

```
/* Requirements: Iterator<I> */
template<typename I>
int distance(I start, I end){
  int i = 0;
  for(; start != end; ++start)
    ++i
  return i;
}
```

Concept with Associated Types II

- Associated types are stored in traits classes (see last lecture), which can be specialised, e.g

```
template<typename T>
struct iterator_traits{};

template<typename T>
struct iterator_traits<T*>{
  typedef T value_type;
}
```

- Using these we can rewrite our algorithm as:

```
/* Requirements: Iterator<I>, Addable<value_type>,
Assignable<value_type>,  CopyConstructible<value_type> */
template<typename I>
typename iterator_traits<I>::value_type
sum(I start, I end, typename iterator_traits<I>::value_type init){
  for (I current = start; current != end; current = next(current))
    init = init + get(current);
  return init;
}
```

Concept Refinement

- nested refinements allow us to reuse concepts to describe other concepts. We can express arbitrary concepts with it.
- *Concept refinement* describes a much more specific, hierarchical relationship.

Concept refinement describes a hierarchical relationship between concepts. If a concept `C2` refines a concept `C1`, then `C2` includes all of the requirements of `C1` and adds its own new requirements. (So every `C2` is also a `C1`, but `C2` is more specific, and enables more and better algorithms.

Concept Refinement Example

- The `Iterator` concept permits forward movement and reading each value.
- To reverse an iterator, we would need the methods
  - `prev` to move to the previous entry,
  - and `set` to set a value in the sequence
- Instead of making `Iterator` bigger, we will create a new concept `BidirectionalIterator` that *refines* `Iterator` but adds the methods.

Concept	Requirements
`BidirectionalIterator<I>`	Refines `Iterator<I>` I must have an operation `prev()` that moves to the previous value in the sequence. `I` must have an operation `set()` that sets the current value

Reverse Algorithm

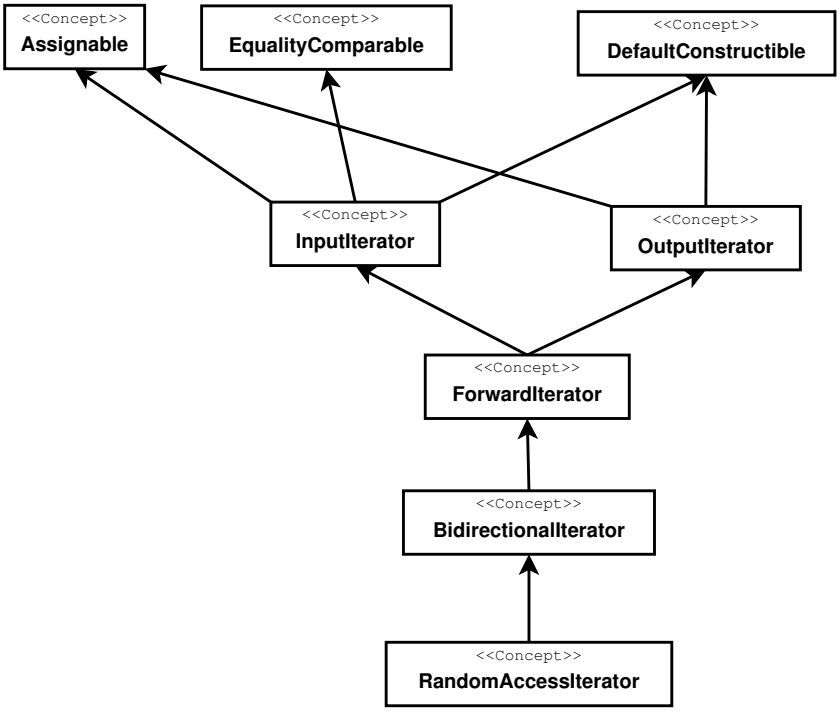
```
/* Requirements: BidirectionalIterator<I>, Assignable<value_type>,
   CopyConstructible<value_type> */
template<typename BI>
void reverse(BI start, BI end){
  while (start!=end) {
    end = prev(end);
    if(start==end)
      break;

    // swap the value
    typename iterator_traits<BI>::value_type tmp = get(start);
    set(start, get(end));
    set(end, tmp);

    start = next(start);
  }
}
```

Concept Taxonomy (STL Iterators)

- Using iterator refinement we can have a whole concept taxonomy.
- E.g. the STL iterator concepts are refined like in the following Figure:



- We can identify corresponding concepts via static polymorphism using compile time tags.

Models

- Concepts describe a set of requirements.
- Models of a concept are abstraction fulfilling these requirements, typically data types or sets of data types.
- E.g. a pointer is a model of the `Iterator` concept.
- The set of models for a given concept is neither fixed nor known, but open for future models.
- Therefore generic programming is an ideal candidate to design reusable libraries.
- When a data type is created we do not need to know all concepts that it will model.
- We can model a new concept without changing the data type, as long as the concept does not require member functions but relies on free functions and traits.

Specialisation

- Concept refinement allows better algorithms because of richer abstractions.
- Additional operations permit implementation of new algorithm, e.g. `reverse()` for `BidirectionalIterator`
- But refined concepts might also allow for more efficient algorithm.

Specialisation Example

We can identify the corresponding concept using a traits class and use function overloading to choose the correct algorithm, e.g:

```
struct IteratorTag{};
struct BidirectionalIteratorTag{};
struct RandomAccessIteratorTag{};
template<typename T>
struct iterator_traits<T*>{
  typedef T value_type;
  typedef RandomAccessIteratorTag tag;
};
template<typename T, typename TAG>
int distance_impl(T start, T end, TAG){
  int i=0;
  for(;start!=end; ++start) ++i;
  return i;
}
template<typename T>
int distance_impl(T start, T end, RandomAccessIteratorTag){
  return end-start;
}
template<typename T>
void distance(T start, T end){
  distance_impl(start, end, typename ::iterator_traits<T>::tag());
}
```

14.2 Concept Checking

The Problem with Concepts

Consider the following erroneous code:

```
int main(){
  std::list<double> dlist;
  dlist.push_back(3);
  dlist.push_back(4);

  std::sort(dlist.begin(), dlist.end()); /*Error: list is no random access
      container! */
}
```

The problem here is:

- `std::list` only provides an iterator adhering to the `BidirectionalIterator` concept.
- `std:sort()` assumes the iterator fullfills the `RandomAccessIterator` concept, of which expects additional methods to `BidirectionalIterator`

- One would want to get an error message similar to this:

```
concept_error.cc: In function 'int main()'
concept_error.cc:10: error: std::list::iterator is not a model of
RandomAccessIterator concept, because ...
```

- Instead the compiler spits out a whole error novel:

The Error Novel

```
include/c++/4.1.2/bits/stl_algo.h: In function 'void std::sort(_RandomAccessIterator,
    _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<double>]':
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/stl_algo.h:2713: error: no match for 'operator-' in '__last -
    __first'
include/c++/4.1.2/bits/stl_algo.h: In function 'void
    std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]':
include/c++/4.1.2/bits/stl_algo.h:2714:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/stl_algo.h:2357: error: no match for 'operator-' in '__last -
    __first'
include/c++/4.1.2/bits/stl_algo.h:2359: error: no match for 'operator+' in '__first +
    16'
include/c++/4.1.2/bits/stl_algo.h:2360: error: no match for 'operator+' in '__first +
    16'
include/c++/4.1.2/bits/stl_algo.h: In function 'void
    std::__insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]':
include/c++/4.1.2/bits/stl_algo.h:2363:   instantiated from 'void
    std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
include/c++/4.1.2/bits/stl_algo.h:2714:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/stl_algo.h:2273: error: no match for 'operator+' in '__first +
    1'
include/c++/4.1.2/bits/stl_algo.h:2363:   instantiated from 'void
    std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
include/c++/4.1.2/bits/stl_algo.h:2714:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/stl_algo.h:2279: error: no match for 'operator+' in '__i + 1'
```

Error Using Concepts

```
include/c++/4.1.2/bits/boost_concept_check.h: In member function 'void
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]':
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/stl_algo.h:2706:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/boost_concept_check.h:555: error: no match for 'operator[]' in
    '((__gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double>
```

```
    >::__i[((__gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n]'
include/c++/4.1.2/bits/boost_concept_check.h: In member function 'void
    __gnu_cxx::_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]':
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/boost_concept_check.h:553:   instantiated from 'void
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]'
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/stl_algo.h:2706:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/boost_concept_check.h:536: error: no match for 'operator+=' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i += ((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n'
include/c++/4.1.2/bits/boost_concept_check.h:537: error: no match for 'operator+' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i + ((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n'
include/c++/4.1.2/bits/boost_concept_check.h:537: error: no match for 'operator+' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n + ((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i'
include/c++/4.1.2/bits/boost_concept_check.h:538: error: no match for 'operator-=' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i -= ((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n'
include/c++/4.1.2/bits/boost_concept_check.h:539: error: no match for 'operator-' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i - ((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n'
include/c++/4.1.2/bits/boost_concept_check.h:541: error: no match for 'operator-' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i - ((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__j'
include/c++/4.1.2/bits/boost_concept_check.h:542: error: no match for 'operator[]' in
    '((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__i[((__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double>
    >::__n]'
```

Error Using Concepts (Contd.)

```
include/c++/4.1.2/bits/boost_concept_check.h: In member function 'void
    __gnu_cxx::_ComparableConcept<_Tp>::__constraints() [with _Tp =
```

```
    std::_List_iterator<double>]':
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_ComparableConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/boost_concept_check.h:529:   instantiated from 'void
    __gnu_cxx::_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]'
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/boost_concept_check.h:553:   instantiated from 'void
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]'
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/stl_algo.h:2706:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/boost_concept_check.h:266: error: no match for 'operator<' in
    '((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__a <
    ((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__b'
include/c++/4.1.2/bits/boost_concept_check.h:267: error: no match for 'operator>' in
    '((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__a >
    ((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__b'
include/c++/4.1.2/bits/boost_concept_check.h:268: error: no match for 'operator<=' in
    '((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__a <=
    ((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__b'
include/c++/4.1.2/bits/boost_concept_check.h:269: error: no match for 'operator>=' in
    '((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__a >=
    ((__gnu_cxx::_ComparableConcept<std::_List_iterator<double>
    >*)this)->__gnu_cxx::_ComparableConcept<std::_List_iterator<double> >::__b'
include/c++/4.1.2/bits/boost_concept_check.h: In member function 'void
    __gnu_cxx::_ConvertibleConcept<_From, _To>::__constraints() [with _From =
    std::bidirectional_iterator_tag, _To = std::random_access_iterator_tag]':
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_ConvertibleConcept<std::bidirectional_iterator_tag,
    std::random_access_iterator_tag>]'
include/c++/4.1.2/bits/boost_concept_check.h:530:   instantiated from 'void
    __gnu_cxx::_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]'
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/boost_concept_check.h:553:   instantiated from 'void
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<_Tp>::__constraints() [with _Tp =
    std::_List_iterator<double>]'
include/c++/4.1.2/bits/boost_concept_check.h:62:   instantiated from 'void
    __gnu_cxx::__function_requires() [with _Concept =
    __gnu_cxx::_Mutable_RandomAccessIteratorConcept<std::_List_iterator<double> >]'
include/c++/4.1.2/bits/stl_algo.h:2706:   instantiated from 'void
    std::sort(_RandomAccessIterator, _RandomAccessIterator) [with
    _RandomAccessIterator = std::_List_iterator<double>]'
concept_error.cc:9:   instantiated from here
include/c++/4.1.2/bits/boost_concept_check.h:223: error: conversion from
    'std::bidirectional_iterator_tag' to non-scalar type
    'std::random_access_iterator_tag' requested
```

Shortcoming of standard C++ errors

- It needs experience to find the relevant error message as well as the position in the code where the bug is.
- There is no obvious correlation between the error message and the documented requirements of sort.
- The error message is overly long, and lists STL internal functions the user should not know.
- Inexperienced users might guess that this is an error in the library code.

Concept Checks

- Up to now all concepts were just documented in the source code.
- But concepts are known at compile already and the models should be checked for validity!
- Still we do not want to unnecessarily execute the required functions at runtime.
- This tricky, but manageable with standard C++.

[Siek and Lumsdaine(2000)] approach

- Exercise requirements in a separate function and assign it to function pointer.
- The compiler will initiate the function but not invoke it.
- An optimising compiler will remove the pointer assignment as dead code.

The Concept Check Class

For each concept we create one class template with the following properties:

- Uniform naming scheme (e.g. `Concept` suffix): `AssignableConcept`, `CopyConstructibleConcept`.
- The template parameter is the type to be checked.
- All concept requirements are exercised in the `constraints()` member function.
- All objects used are declared as data members

```
template<class T>
struct AddableConcept
{
  void constraints()
  {
    i+j;
  }

private:
  T i, j;;
};
```

Exercising Concept Checks for Functions

- Requirements are checked via assignment of the `constraints` member functions to a function pointer.
- This can easily be done with the following function template:

```
template<class Concept>
void function_requires()
{
  void (Concept::*x)() = &Concept::constraints;
}
```

- Example usage:

```
template<typename I>
typename ::iterator_traits<I>::value_type
sum(I start, I end, typename ::iterator_traits<I>::value_type init) {
  function_requires<IteratorConcept<I> >();
  typedef typename ::iterator_traits<I>::value_type T;
  function_requires<AddableConcept<T> >();
  function_requires<AssignableConcept<T> >();
  function_requires<CopyConstructibleConcept<T> >();

  for (I current = start; current != end; current = next(current))
    init = init + get(current);
  return init;
}
```

Concept Checks for Classes

We define the following macro to check concepts inside of the class body:

```
#define CLASS_REQUIRE(type_var, concept)       \
  typedef void (concept <type_var> ::* func##type_var##concept)(); \
                                               \
  template <func##type_var##concept FuncPtr>   \
  struct dummy_struct_##type_var##concept {}; \
                                               \
  typedef dummy_struct_##type_var##concept<    \
    &concept <type_var>::constraints>          \
    dummy_typedef_##type_var##concept
```

1. First we create a type for `void` member functions of class `concept` taking no arguments.
2. Next we define a nested class taking such a function as its template parameter.
3. Last we define a type with this nested class and the `constraints()` function as the template parameter, which causes the function to be instantiated.

Exercising Concept Checks for Classes

Exercise the check like this:

```
template<typename T>
struct Tester{
  CLASS_REQUIRE(T, BidirectionalIteratorConcept);
};
```

An Example Check for a Refined Concept

The refinement is checked using `function_requires()` inside of the `constraints()` member function:

```
template<typename T>
struct IteratorConcept{
  typedef typename ::iterator_traits<T>::value_type value_type;

  void constraints(){
    function_requires<EqualityComparableConcept<T> >();
    function_requires<CopyConstructibleConcept<T> >();
    function_requires<AssignableConcept<T> >();
    get(t);
    next(t);
  }

protected:
  value_type v;
  T t;
};
```

Concept Error Message

If we make the copy constructor of our `ListNode` private., our approach triggers the following meaningful error message:

```
list.hh: In member function 'void␣CopyConstructibleConcept<T>::constraints()␣
    [with␣T␣=␣ListNode<double>]':
concept_check.hh:12:␣␣␣instantiated␣from␣'void function_requires() [with
    Concept = CopyConstructibleConcept<ListNode<double> >]'
concept_check.hh:74:   instantiated from 'void␣
    IteratorConcept<T>::constraints()␣[with␣T␣=␣ListNode<double>]'
concept_check.hh:12:␣␣␣instantiated␣from␣'void function_requires() [with
    Concept = IteratorConcept<ListNode<double> >]'
concept_check.hh:91:   instantiated from 'void␣
    BidirectionalIteratorConcept<T>::constraints()␣[with␣T␣=␣ListNode<double>]'
sum.cc:9:␣␣␣instantiated␣from␣'Tester<ListNode<double> >'
sum.cc:22:   instantiated from here
list.hh:14: error: 'ListNode<T>::ListNode(const␣ListNode<T>&)␣[with␣T␣=␣
    double]'␣is␣private
concept_check.hh:45:␣error:␣within␣this␣context
```

Concept Coverage

- We discussed how to check whether template parameters satisfy a concepts requirements.
- We need to check whether the requirements we pose via concepts cover all requirements our algorithm poses onto the data types.
- This can be covered through *archetype classes*.

An *archetype class* is an exact implementation of the interface associated with a particular concept. The runtime behaviour of the archetype class is not important.

- The function can be empty.

- Test programs can then be compiled with archetype classes.
- If the program compiles then we can be sure that the concepts cover the component/algorithm

Example Archetype class

```
template<typename T>
struct Proxy{
  operator T(){ return t;}
  static T t;
};

template<typename T>
T Proxy<T>::t;


class ValueArchetype{
public:
  ValueArchetype(const ValueArchetype&){}
  ValueArchetype& operator=(const ValueArchetype&){
    return *this;
  }
  ValueArchetype operator+(const ValueArchetype&){
    return ValueArchetype();
  }
private:
  ValueArchetype(){}
  friend class Proxy<ValueArchetype>;
  friend void testSum();
};

template<typename T>
class IteratorArchetype
{
public:
  IteratorArchetype(const IteratorArchetype&){}
  friend bool operator!=(const IteratorArchetype<T>&, const
      IteratorArchetype<T>&){ return true;}
  friend bool operator==(const IteratorArchetype<T>&, const
      IteratorArchetype<T>&){ return true;}
private:
  IteratorArchetype(){}
  friend void testSum();
};template<typename T> struct iterator_traits;

template<typename T>
struct iterator_traits<IteratorArchetype<T> >{
  typedef T value_type;
};

template<typename T>
IteratorArchetype<T> next(IteratorArchetype<T> i){ return i;}
template<typename T>
Proxy<T> get(IteratorArchetype<T> i){ return Proxy<T>(); }
```

Summary and Outlook

- Generic programming is done by means of lifting, concepts, and models.
- Concepts can be used to specify the requirements of generic algorithm.
- These requirements can be checked at compile time using concept checks.
- Further more archetypes that exactly conform to the concepts can be used to check the coverage of the algorithm.
- Concept checking is being incorporated into modern compilers (experimental in g++ with option `-D_GLIBCXX_CONCEPT_CHECKS`)
- A complete implementation is available in the [Boost Concepts(2008)] library.

Literatur

[Gregror(2008)] Doug Gregor. `www.generic-programming.org`

[Siek and Lumsdaine(2000)] Jeremy Siek and Andrew Lumsdaine. *Concept Checking: Binding Parametric Polymorphism in C++ Proceedings First Workshop on C++ Template Programming, 2000*

[Boost Concepts(2008)] Boost C++ Concept Checking Reference `http://www.boost.org/doc/libs/1_35_0/libs/concept_check/`

15 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.

[2] D. Vandevoorde, N. M. Josuttis: C++ Templates - The Complete Guide, Addison Wesley 2003

Beispiel

Summe über eine Sequenz

Die Summe über eine Reihe von Werten die in einem C-Array gespeichert sind, lässt sich wie folgt schreiben:

```
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;
}
```

15.1 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;
}
```

15.2 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;
}
```

15.3 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;

}
```

15.4 Iterator Traits

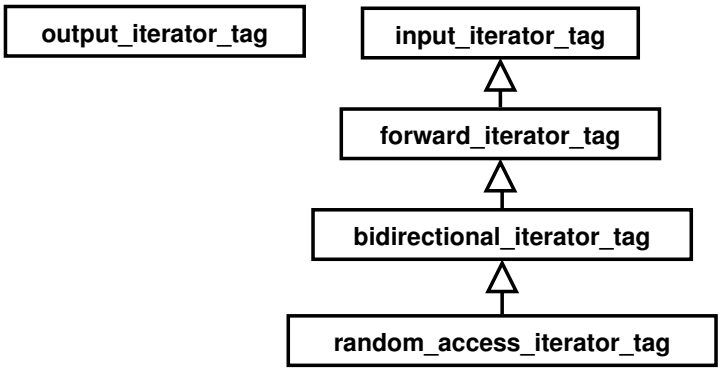
- Auch STL Iteratoren exportieren viele ihrer Eigenschaften über Traits:
- Laut C++ Standard werden folgende Informationen über Iteratoren bereitgestellt:

```
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>::category>
  ::advance(pos, dist);
}
```

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 Konstructor 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 und 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 T* const_pointer_type;
  typedef 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){
    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`s:

  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.

16 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 er ein Konzept korrekt implementiert (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).

16.1 Engine Konzept

- Es wird ein Klassentemplate definiert, dass einen Wrapper für das Modell darstellt, das 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);
    }
};
```

16.2 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 Basisklasse 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);
   }
};
```

Eigenschaften

- 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.

17 Template Metaprogramming

17.1 Grundlagen des Template Metaprogramming

Was ist ein Metaprogramm?

Ein Metaprogramm ist ein Programm das andere Programme (oder sich selbst) schreibt und verändert oder das eine 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 [Unruh1994, Veldhuizen(1995)]
- 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 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 auch 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<10)? 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 ausreichende 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_unroling.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_unroling2.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';
}
```

17.2 Template Metaprogramming mit Boost

Boost

- Boost ist eine Sammlung von C++ Bibliotheken, die vielerlei Aufgaben erfüllen, die über die Standardbibliothek hinausgehen.
- Viele Erweiterungen von Boost werden in den neuen C++ Standard C++1x einfließen.
- Boost enthält auch `MPL` eine Bibliothek die die Programmierung von Template Metaprogrammen stark erleichtert.
- `MPL` stellt unter anderem Sequenzen und Metafunktionen zur Verfügung.
- Neben der Bereitstellung von Funktionalität soll auch eine konzeptionelle Grundlage für die Metaprogrammierung in C++ zur Verfügung stellen um diese so einfach und freudvoll zu machen wie normale Programmierung.

Boost MPL

Boost.MPL stellt ähnlich wie die STL folgende Sprachmittel zur Verfügung:

- Sequenzen (allerdings von Typen nicht von Werten)

- Iteratoren
- Algorithmen (die mit Typen arbeiten statt mit Werten)
- Metafunktionen (die ebenfalls mit Typen arbeiten)
- Datentypen

Diese befinden sich alle im Namespace `boost::mpl`.

Berechnungen mit Dimensionsanalyse

Als Beispiel für die Verwendung von Boost.MPL wollen wir eine Berechnung mit Dimensionsanalyse durchführen um die Korrektheit der Berechnung überprüfen zu können.

Die Einheit stellen wir durch ein Array von Potenzen der einzelnen Dimensionen dar:

```
typedef int dimension[7]; // m  l  t  ...
dimension const mass      = {1, 0, 0, 0, 0, 0, 0};
dimension const length    = {0, 1, 0, 0, 0, 0, 0};
dimension const time      = {0, 0, 1, 0, 0, 0, 0};
```

Einheiten im Nenner werden durch negative Exponenten dargestellt, da $1/x = x^{-1}$.

In dieser Darstellung wäre die Einheit einer Kraft $[ml/t^2]$

```
dimension const force  = {1, 1, -2, 0, 0, 0, 0};
```

Dimensionsanalyse mit Template Metaprogramming

Die Dimensionsanalyse soll zur Übersetzungszeit erfolgen, so dass die Programmausführung nicht verlangsamt wird. Deshalb wollen wir Template Metaprogramming verwenden. Dafür müssen wir jedoch eine Array von Typen verwenden statt einem Array von Werten.

Boost definiert dafür den Metadatentyp `mpl::vector`

```
#include <boost/mpl/vector.hpp>

typedef boost::mpl::vector<signed char, short, int, long> signed_types;
```

Auch Zahlenwerte müssen über Typen dargestellt werden:

```
#include <boost/mpl/int.hpp>

namespace mpl = boost::mpl; // namespace alias
static int const five = mpl::int_<5>::value;
```

Um Schreibarbeit zu sparen wird hier ein Namespace Alias definiert, der dann anstelle des ursprünglichen Namespaces verwendet werden kann:

```
namespace mpl = boost::mpl; // namespace alias
```

Wrapperklassen für Ganzzahlen

Boost stellt auch Wrapperklassen für andere Ganzzahlentypen bereit wie `long_` und `bool_`.

Jetzt können wir die Masse schreiben als:

```
typedef mpl::vector<
   mpl::int_<1>, mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
 , mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
> mass;
```

und die Länge als:

```
typedef mpl::vector<
   mpl::int_<0>, mpl::int_<1>, mpl::int_<0>, mpl::int_<0>
 , mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
> length;
...
```

Initialisierung von Metavektoren

Da sich das doch sehr sperrig liest, gibt es dafür in Boost eine Abkürzung:

```
#include <boost/mpl/vector_c.hpp>

typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length; // or position
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
typedef mpl::vector_c<int,0,0,0,1,0,0,0> charge;
typedef mpl::vector_c<int,0,0,0,0,1,0,0> temperature;
typedef mpl::vector_c<int,0,0,0,0,0,1,0> intensity;
typedef mpl::vector_c<int,0,0,0,0,0,0,1> angle;
```

Der Typ dieser `mpl::vector_c` Spezialisierungen unterscheidet sich zwar von den vorhergehenden, aber sie sind trotzdem äquivalent zu den ausführlicheren Versionen von `mpl::vector`.

Definition gemischter Einheiten und von Skalaren

Wir können auch gemischte Einheiten definieren:

```
// base dimension:        m l  t ...
typedef mpl::vector_c<int,0,1,-1,0,0,0,0> velocity;     // l/t
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration; // l/(t2)
typedef mpl::vector_c<int,1,1,-1,0,0,0,0> momentum;     // ml/t
typedef mpl::vector_c<int,1,1,-2,0,0,0,0> force;        // ml/(t2)
```

Und Skalare (wie z.B. π) lassen sich auch darstellen:

```
typedef mpl::vector_c<int,0,0,0,0,0,0,0> scalar;
```

Kombination von Daten und Metadaten

Die Vektortypen sind noch reine Metadaten. Mit Hilfe einer Wrapperklasse lassen sie sich mit den echten Zahlenwerten zusammenbringen. Die Wrapperklasse wird mit dem Zahlentyp und dem Dimensionstyp parametrisiert:

```
template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
       : m_value(x)
    {}

    T value() const { return m_value; }
 private:
    T m_value;
};
```

Darstellung dimensionsbehafteter Zahlen

Jetzt haben wir eine Möglichkeit dimensionsbehaftete Zahlen darzustellen:

```
quantity<float,length> l( 1.0f );
quantity<float,mass> m( 2.0f );
```

Die Dimensionen erscheinen nirgendwo in der Definition von Quantity außer in der Templateparameterliste. Ihr Zweck ist es einzig sicherzustellen, dass l und m verschiedene Typen haben. Da dies der Fall ist, ist es nun unmöglich einer Masse eine Länge zuzuweisen:

```
m = l;    // compile-time type error
```

Addition und Subtraktion

Die Operatoren für Addition und Subtraktion sind ganz einfach, da die Dimensionen der Argumente hier immer identisch sein müssen:

```
template <class T, class D> quantity<T,D>
operator+(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() + y.value());
}

template <class T, class D> quantity<T,D>
operator-(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() - y.value());
}
```

Addition und Subtraktion

Mit diesen Operatoren können wir jetzt folgenden Code schreiben:

```
quantity<float,length> len1( 1.0f );
quantity<float,length> len2( 2.0f );

len1 = len1 + len2;   // OK
```

aber wir dürfen keine inkompatiblen Dimensionen addieren:

```
len1 = len2 + quantity<float,mass>( 3.7f ); // error
```

Multiplikation

Die Multiplikation ist deutlich schwieriger. Hier müssen die Einheiten addiert werden. In der MPL gibt es ein Äquivalent zum Funktor, eine Metafunktion. Eine Metafunktion muss ein `struct apply` definieren, dass einen Typ `type` definiert:

```
struct plus_f
{
    template <class T1, class T2>
    struct apply
    {
       typedef typename mpl::plus<T1,T2>::type type;
    };
};
```

Es gibt auch ein Äquivalent zum Transformalgorithmus der STL, dass eine Metafunktion für jedes Element des Metavektors aufruft:

```
template <class T, class D1, class D2>  // new dimensions
quantity<T,typename mpl::transform<D1,D2,plus_f>::type>
operator*(quantity<T,D1> x, quantity<T,D2> y)
{
    typedef typename mpl::transform<D1,D2,plus_f>::type dim;
    return quantity<T,dim>( x.value() * y.value() );
}
```

Jetzt können wir auch Multiplikationen ausführen:

```
quantity<float,mass> m(5.0f);
quantity<float,acceleration> a(9.8f);
std::cout << "force = " << (m * a).value();
```

Unser Multiplikationsoperator multipliziert die Fließkommazahlen (mit dem Ergebnis `49.0f`) und das Metaprogramm berechnet mit `transform` die Summe der Dimensionsexponenten, wobei sich eine neue Dimension ergibt:

```
mpl::vector_c<int,1,1,-2,0,0,0,0>
```

Leider lässt sich immer noch nicht schreiben:

```
quantity<float,force> f = m * a;
```

da die Typen nicht als identisch erkannt werden.

Um dies zu realisieren, können wir der Klasse `quantity` einen Konvertierungsoperator hinzufügen:

```
template <class OtherDimensions>
quantity(quantity<T,OtherDimensions> const& rhs)
  : m_value(rhs.value())
{
    BOOST_STATIC_ASSERT((
       mpl::equal<Dimensions,OtherDimensions>::type::value
    ));
}
```

dieser verwendet die Template Metafunktion `mpl::equal`, die Element für Element überprüft ob die Inhalte von zwei Metavektoren gleich sind.

Das Makro `BOOST_STATIC_ASSERT` prüft zur Übersetzungszeit, ob eine Bedingung erfüllt ist.

Division

Die Division schreiben wir mit weiteren Sprachmitteln noch etwas kürzer:

```
#include<boost/mpl/placeholders.hpp>
using namespace mpl::placeholders;

template <class T, class D1, class D2>
quantity<T,typename mpl::transform<D1,D2,mpl::minus<_1,_2> >::type>
operator/(quantity<T,D1> x, quantity<T,D2> y)
{
   typedef typename
     mpl::transform<D1,D2,mpl::minus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() / y.value() );
}
```

`_1` und `_2` sind sogenannte Placeholder. Transform wendet dann `mpl::minus` für jeden Wert von `D1` und `D2` an.

Anwendung der Dimensionsanalyse

Jetzt funktioniert unsere Dimensionsanalyse für alle Grundrechenarten und wir können Sie auch dazu verwenden zu prüfen ob wir richtig gerechnet haben:

```
quantity<float,mass> m2 = f/a;
float rounding_error = std::abs((m2 - m).value());
```

Komplettes Programm

```
#include <cmath>
#include <iostream>
#include <boost/mpl/vector.hpp>
#include <boost/mpl/int.hpp>
#include <boost/mpl/vector_c.hpp>
#include <boost/mpl/transform.hpp>
#include <boost/mpl/equal.hpp>
#include <boost/mpl/placeholders.hpp>
#include <boost/static_assert.hpp>

namespace mpl = boost::mpl; // namespace alias
using namespace mpl::placeholders;

// base dimension:        m l  t ...
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass_;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length_; // or position
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time_;
typedef mpl::vector_c<int,0,0,0,1,0,0,0> charge_;
typedef mpl::vector_c<int,0,0,0,0,1,0,0> temperature_;
typedef mpl::vector_c<int,0,0,0,0,0,1,0> intensity_;
typedef mpl::vector_c<int,0,0,0,0,0,0,1> angle_;

// combined dimensions
typedef mpl::vector_c<int,0,1,-1,0,0,0,0> velocity_;     // l/t
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration_; // l/(t2)
typedef mpl::vector_c<int,1,1,-1,0,0,0,0> momentum_;     // ml/t
typedef mpl::vector_c<int,1,1,-2,0,0,0,0> force_;        // ml/(t2)
// scalar
typedef mpl::vector_c<int,0,0,0,0,0,0,0> scalar_;

struct plus_f
{
    template <class T1, class T2>
    struct apply
    {
       typedef typename mpl::plus<T1,T2>::type type;
    };
};

template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
       : m_value(x)
    {}

    T value() const { return m_value; }
```

```
template <class OtherDimensions>
    quantity(quantity<T,OtherDimensions> const& rhs)
      : m_value(rhs.value())
    {
        BOOST_STATIC_ASSERT((
          mpl::equal<Dimensions,OtherDimensions>::type::value
        ));
    }

 private:
    T m_value;
};

template <class T, class D> quantity<T,D>
operator+(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() + y.value());
}

template <class T, class D> quantity<T,D>
operator-(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() - y.value());
}

template <class T, class D1, class D2>
quantity<T,typename mpl::transform<D1,D2,plus_f>::type>
operator*(quantity<T,D1> x, quantity<T,D2> y)
{
    typedef typename mpl::transform<D1,D2,plus_f>::type dim;
    return quantity<T,dim>( x.value() * y.value() );
}

template <class T, class D1, class D2>
quantity<T,typename mpl::transform<D1,D2,mpl::minus<_1,_2> >::type>
operator/(quantity<T,D1> x, quantity<T,D2> y)
{
   typedef typename mpl::transform<D1,D2,mpl::minus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() / y.value() );
}

int main()
{
  quantity<float,length_> l( 1.0f );
  quantity<float,mass_> m( 2.0f );

  quantity<float,length_> len1( 1.0f );
  quantity<float,length_> len2( 2.0f );

  len1 = len1 + len2;   // OK

  quantity<float,acceleration_> a(9.8f);
  std::cout << "force␣=␣" << (m * a).value() << std::endl;

  quantity<float,force_> f = m * a;
  quantity<float,mass_> m2 = f/a;
  float rounding_error = std::abs((double)(m2 - m).value());
```

```
  std::cout << "rounding error = " << rounding_error << std::endl;
}
```

Literatur

[Abrahams(2004)] D. Abrahams, A. Gurtovoy *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Adisson Wesley, 2004.

[Boost.MPL] The Boost.MPL Library `http://www.boost.org`

[Veldhuizen(1995)] Todd L. Veldhuizen *Expression Templates* C++ Report, Vol. 7 No. 5 (June 1995), pp. 26-31 `http://ubiety.uwaterloo.ca/~tveldhui/papers/Expression-Templates/exprtmpl.html`

[Veldhuizen(1995)] Todd L. Veldhuizen *Using C++ template metaprograms* C++ Report Vol. 7 No. 4 (May 1995), pp. 36-43. `http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html`

[Unruh1994] E. Unruh Prime number computation, 1994. newblock ANSI X3J16-94-0075/ISO WG21-462.

18 Expression Templates

Array für numerische Berechnungen

```
#include <cstdlib>
#include <cassert>

template<typename T>
class SArray {
  public:
    // create array with initial size
    explicit SArray (size_t s)
     : storage(new T[s]), storage_size(s)
    {
        init();
    }
    // copy constructor
    SArray (SArray<T> const& orig)
     : storage(new T[orig.size()]), storage_size(orig.size())
    {
        copy(orig);
    }
    // destructor: free memory
    ~SArray()
    {
        delete[] storage;
    }

    // assignment operator
    SArray<T>& operator= (SArray<T> const& orig)
    {
        if (&orig!=this)
            copy(orig);
```

```
        return *this;
    }
    // return size
    size_t size() const
    {
        return storage_size;
    }
    // index operator for constants and variables
    T operator[] (size_t idx) const
    {
        return storage[idx];
    }
    T& operator[] (size_t idx)
    {
        return storage[idx];
    }

  protected:
    // init values with default constructor
    void init()
    {
        for (size_t idx = 0; idx<size(); ++idx)
            storage[idx] = T();
    }
    // copy values of another array
    void copy (SArray<T> const& orig)
    {
        assert(size()==orig.size());
        for (size_t idx = 0; idx<size(); ++idx)
            storage[idx] = orig.storage[idx];
    }
  private:
    T*     storage;       // storage of the elements
    size_t storage_size;  // number of elements
};
```

Fragestellung

- Wir möchten nun mit diesem Array mathematische Berechnungen durchführen.
- Dafür müssen wir in der Lage sein z.B. Additionen und Multiplikationen mit Skalaren und anderen Arrays durchführen zu können.
- z.B. sollte die folgende Berechnung möglich sein:

```
SArray<double> x(1000), y(1000);
x = 1.2*x+x*y;
```

Implementierung von Operatoren

```
// addition of two SArrays
template<typename T>
SArray<T> operator+ (SArray<T> const& a, SArray<T> const& b)
{
    SArray<T> result(a.size());
    for (size_t k = 0; k<a.size(); ++k)
```

```
        result[k] = a[k]+b[k];
    return result;
}

// multiplication of two SArrays
template<typename T>
SArray<T> operator* (SArray<T> const& a, SArray<T> const& b)
{
    SArray<T> result(a.size());
    for (size_t k = 0; k<a.size(); ++k)
        result[k] = a[k]*b[k];
    return result;
}

// multiplication of scalar and SArray
template<typename T>
SArray<T> operator* (T const& s, SArray<T> const& a)
{
    SArray<T> result(a.size());
    for (size_t k = 0; k<a.size(); ++k)
        result[k] = s*a[k];
    return result;
}

// multiplication of SArray and scalar
// addition of scalar and SArray
// addition of SArray and scalar
//...
```

Nachteil dieses Ansatzes

```
SArray<double> x(1000), y(1000);
x = 1.2*x+x*y;
```

1. Jeder Operator generiert im allerbesten Fall ein temporäres Array (und auch das nur, wenn der Compiler perfekt arbeitet, sonst sind es mehr). In unserem Beispiel heisst das zumindest 3 temporäre Objekte.

   ```
   SArray<double> t1=1.2*x;
   SArray<double> t2=x*y;
   SArray<double> t3=t1+t2;
   x = t3
   ```

   - Bei kleinen Feldern bremst hier insbesondere die Allokation und Deallokation des Speichers die Berechnung aus.
   - Bei großen Arrays hingegen ist der zusätzliche Speicherbedarf völlig inakzeptabel.

2. Bei jeder Operatoranwendung ist ein Schleifendurchlauf über die Elemente des Arrays notwendig. Im konkreten Fall werden:
   - ca. 6000 `double` Werte gelesen
   - ca. 4000 `double` Werte geschrieben

Teilweise Verbesserung

Einige der temporären Objekte lassen vermeiden, wenn die kombinierten Berechnungs-/Zuweisungsoperatoren (+=, *=, ...) verwendet werden:

```
SArray<double> x(1000), y(1000), t(x);
t*=y;
x*=1.2;
x+=tmp;
```

```
// additive assignment of SArray
template<typename T>
SArray<T>& SArray<T>::operator+= (SArray<T> const& b)
{
    for (size_t k = 0; k<size(); ++k)
        (*this)[k] += b[k];
    return *this;
}

// multiplicative assignment of SArray
template<typename T>
SArray<T>& SArray<T>::operator*= (SArray<T> const& b)
{
    for (size_t k = 0; k<size(); ++k)
        (*this)[k] *= b[k];
    return *this;
}

// multiplicative assignment of scalar
template<typename T>
SArray<T>& SArray<T>::operator*= (T const& s)
{
    for (size_t k = 0; k<size(); ++k)
        (*this)[k] *= s;
    return *this;
}
```

Nachteile

- Die Lesbarkeit des Programmes leidet.
- Es ist immer noch ein temporäres Objekt notwendig.
- Die Anzahl der Schleifendurchläufe und Schreiblesenzugriffe bleibt gleich hoch.

Der Idealfall

Wir möchten die folgende einfache Notation verwenden können:

```
SArray<double> x(1000), y(1000);
x = 1.2*x+x*y;
```

Der Compiler soll daraus Code generieren, der nur einen Schleifendurchlauf benötigt:

```
SArray<double> x(1000), y(1000);
for(int i=0; i < x.size(), ++i)
    x[i] = 1.2*x[i]+x[i]*y[i];
```

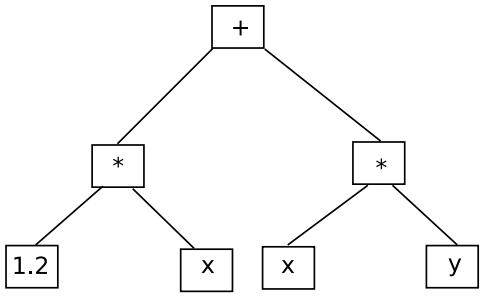
Vorteile:

- Keine temporären Objekte mehr nötig.
- Pro Iteration: einmal lesen von `x[i]` und `y[i]` aus dem Hauptspeicher, einmal schreiben von `x[i]`.
- 2000 Lese- und 1000 Schreibzugriffe (also weniger als ein Drittel).

Mit Templatemetaprogramming lässt sich das umsetzen.

Expression Templates

- Teiloperationen, wie z.B. `1.2*x` werden nicht gleich berechnet.
- Stattdessen werden die Operationen gesammelt, bis ihre Auswertung notwendig wird (z.B. durch eine Zuweisung).
- Da die Operationen zur Übersetzungszeit feststehen, können sie als Templateargumente verwendet werden.
- Dies führt zu einem Baum von Operatione, wie z.B.



- Dies liese sich z.B. speichern als:

```
Expression<Expression<Scalar,mult,Array>,
          plus,Expression<Array,mult,Array> >
```

- oder als:

```
A_Add<double, A_Mult<double,A_Scalar<double>,Array<double> >,
      A_Mult<double,Array<double>,Array<double> > >
```

Addition

```
template <typename T, typename OP1, typename OP2>
class A_Add
{
  private:
    typename A_Traits<OP1>::ExprRef op1;    // first operand
    typename A_Traits<OP2>::ExprRef op2;    // second operand
  public:
    // constructor initializes references to operands
    A_Add (OP1 const& a, OP2 const& b)
```

```
     : op1(a), op2(b)
    {}
    // compute sum when value requested
    T operator[] (size_t idx) const
    {
        return op1[idx] + op2[idx];
    }
    // size is maximum size
    size_t size() const
    {
        assert (op1.size()==0 || op2.size()==0
                || op1.size()==op2.size());
        return op1.size()!=0 ? op1.size() : op2.size();
    }
};
```

Multiplikation

```
template <typename T, typename OP1, typename OP2>
class A_Mult
{
  private:
    typename A_Traits<OP1>::ExprRef op1;    // first operand
    typename A_Traits<OP2>::ExprRef op2;    // second operand
  public:
    // constructor initializes references to operands
    A_Mult (OP1 const& a, OP2 const& b)
     : op1(a), op2(b)
    {}
    // compute product when value requested
    T operator[] (size_t idx) const
    {
        return op1[idx] * op2[idx];
    }
    // size is maximum size
    size_t size() const
    {
        assert (op1.size()==0 || op2.size()==0
                || op1.size()==op2.size());
        return op1.size()!=0 ? op1.size() : op2.size();
    }
};
```

Skalare

```
template <typename T>
class A_Scalar
{
  private:
    T const& s;  // value of the scalar
  public:
    // constructor initializes value
    A_Scalar (T const& v)
     : s(v)
    {}
    // for index operations the scalar is the value of each element
```

```
    T operator[] (size_t) const
    {
        return s;
    }
    // scalars have zero as size
    size_t size() const
    {
        return 0;
    };
};
```

Helper Traits

```
template <typename T> class A_Scalar;

// primary template
template <typename T>
class A_Traits
{
  public:
    typedef T const& ExprRef;     // type to refer to is constant reference
};

// partial specialization for scalars
template <typename T>
class A_Traits<A_Scalar<T> >
{
  public:
    typedef A_Scalar<T> ExprRef;  // type to refer to is ordinary value
};
```

Array

```
template <typename T, typename Rep = SArray<T> >
class Array
{
  private:
    Rep expr_rep;   // (access to) the data of the array
  public:
    // create array with initial size
    explicit Array (size_t s)
     : expr_rep(s)
    {}
    // create array from possible representation
    Array (Rep const& rb)
     : expr_rep(rb)
    {}
    // assignment operator for same type
    Array& operator= (Array const& b)
    {
        assert(size()==b.size());
        for (size_t idx = 0; idx<b.size(); ++idx)
            expr_rep[idx] = b[idx];
        return *this;
    }
```

```
    // assignment operator for arrays of different type
    template<typename T2, typename Rep2>
    Array& operator= (Array<T2, Rep2> const& b)
    {
        assert(size()==b.size());
        for (size_t idx = 0; idx<b.size(); ++idx)
            expr_rep[idx] = b[idx];
        return *this;
    }
    // size is size of represented data
    size_t size() const
    {
        return expr_rep.size();
    }
    // index operator for constants and variables
    T operator[] (size_t idx) const
    {
        assert(idx<size());
        return expr_rep[idx];
    }

    T& operator[] (size_t idx)
    {
        assert(idx<size());
        return expr_rep[idx];
    }
    // return what the array currently represents
    Rep const& rep() const
    {
        return expr_rep;
    }
    Rep& rep()
    {
        return expr_rep;
    }
};
```

Operatoren

```
// addition of two Arrays
template <typename T, typename R1, typename R2>
Array<T,A_Add<T,R1,R2> >
operator+ (Array<T,R1> const& a, Array<T,R2> const& b)
{
    return Array<T,A_Add<T,R1,R2> >
           (A_Add<T,R1,R2>(a.rep(),b.rep()));
}

// multiplication of two Arrays
template <typename T, typename R1, typename R2>
Array<T, A_Mult<T,R1,R2> >
operator* (Array<T,R1> const& a, Array<T,R2> const& b)
{
    return Array<T,A_Mult<T,R1,R2> >
           (A_Mult<T,R1,R2>(a.rep(), b.rep()));
}
```

```
// multiplication of scalar and Array
template <typename T, typename R2>
Array<T, A_Mult<T,A_Scalar<T>,R2> >
operator* (T const& s, Array<T,R2> const& b)
{
    return Array<T,A_Mult<T,A_Scalar<T>,R2> >
           (A_Mult<T,A_Scalar<T>,R2>(A_Scalar<T>(s), b.rep()));
}

// multiplication of Array and scalar
//...
```

Was passiert genau?

```
  Array<double> x(1000000), y(1000000);
  x=1.2*x+x*y;
```

Der Compiler übersetzt zunächst die $*$ Operation ganz links, bei der ein Array mit einem Skalar multipliziert wird. Der Operator

```
// multiplication of scalar and Array
template <typename T, typename R2>
Array<T, A_Mult<T,A_Scalar<T>,R2> >
operator* (T const& s, Array<T,R2> const& b)
{
    return Array<T,A_Mult<T,A_Scalar<T>,R2> >
           (A_Mult<T,A_Scalar<T>,R2>(A_Scalar<T>(s), b.rep()));
}
```

liefert

```
Array<double, A_Mult<double, A_Scalar<double>, SArray<double> > >

  Array<double> x(1000000), y(1000000);
  x=1.2*x+x*y;
```

als nächstes wird die zweite Multiplikation ausgewerte, eine Array-Array Operation:

```
// multiplication of two Arrays
template <typename T, typename R1, typename R2>
Array<T, A_Mult<T,R1,R2> >
operator* (Array<T,R1> const& a, Array<T,R2> const& b)
{
    return Array<T,A_Mult<T,R1,R2> >
           (A_Mult<T,R1,R2>(a.rep(), b.rep()));
}
```

dies resultiert in

```
Array<double, A_Mult<double, SArray<double>, SArray<double> > >

  Array<double> x(1000000), y(1000000);
  x=1.2*x+x*y;
```

zuletzt wird die Addition ausgeführt (wieder eine Array-Array Operation):

```
// addition of two Arrays
template <typename T, typename R1, typename R2>
Array<T,A_Add<T,R1,R2> >
operator+ (Array<T,R1> const& a, Array<T,R2> const& b)
```

```
{
    return Array<T,A_Add<T,R1,R2> >
           (A_Add<T,R1,R2>(a.rep(),b.rep()));
}
```

es ergibt sich

```
Array<double, A_Add<double,A_Mult<double, A_Scalar<double>,
                                  SArray<double> >
                            A_Mult<double, SArray<double>,
                                  SArray<double> > > >
```

Dieser Typ wird dem Zuweisungsoperator übergeben

```
// assignment operator for arrays of different type
template<typename T2, typename Rep2>
Array& operator= (Array<T2, Rep2> const& b)
{
    assert(size()==b.size());
    for (size_t idx = 0; idx<b.size(); ++idx)
        expr_rep[idx] = b[idx];
    return *this;
}
```

Dieser berechnet jedes Element indem er den `operator[]` des Arguments aufruft, dessen Typ

```
A_Add<double,A_Mult<double, A_Scalar<double>, SArray<double> >
             A_Mult<double, SArray<double>, SArray<double> > > >
```

ist. Wenn man diesen auswertet zeigt sich, das folgendes berechnet wird:

```
(1.2*x[idx]) + (x[idx]*y[idx])
```

Schlussfolgerungen

- Die Implementatierung von Expression templates ist ziemlich aufwändig.
- Dies wird in der Regel durch den Performancegewinn gerechtfertigt. In unserem Beispiel sind Expression Templates 4-5 mal so schnell wie eine traditionelle Implementierung!
- Es können allerdings Schwierigkeiten auftreten, z.B. liefert folgende Matrix-Vektormultiplikation ein falsches Ergebnis: `x = A * x`, da hier `x[0]` schon im ersten Schritt überschrieben wird, aber noch für die weiteren Berechnungen benötigt wird. Bei `y = A * x` gibt es kein Problem.
- Möchte man Zwischenergebnisse speichern ohne sie auszuwerten, dann ist das mühsam, da man dann den Typ von Hand schreiben muss:

  ```
  A_Add<double, A_Add<double,SArray<double>,SArray<double> >,
                A_Minus<double,SArray<double>,SArray<double> > >
              intermediate = a + b + (c - d);
  ```

- Expression Templates sind nicht auf numerische Berechnungen beschränkt. Boost verwendet sie z.B. zum implementieren seiner “lambda expression library”.

The Expression Class

```
template<typename L, typename Op, typename R>
struct Expression{
  typedef typename Op::result_type value_type;

  Expression(const L& l, const R& r)
    : left(l), right(r)
  {}

  typename Op::result_type operator[](std::size_t i) const{
    Op()(left[i], right[i]);
  }

  std::size_t size(){
    assert(left.size()==0 || right.size()==0 ||
           left.size() == right.size());
    return left.size()? left.size(): right.size();
  }

private:
  typename expression_traits<L>::const_reference left;
  typename expression_traits<R>::const_reference right;
};
```

Representing Scalars as Expressions

```
template<typename T>
class ScalarExpression{
public:
  typedef T value_type;
  ScalarExpression(const T& v)
    : val(v){}

  T operator[](std::size_t i) const{
    return val;
  }

  std::size_t size() const{
    return 0;
  }

private:
  const T& val;
};
```

- `ScalarExpression` has to model a vector expression because we need to evaluate using `operator[]`.
- Scalars have to be handled with care: They have to be copied as they are temporary and thus only until the expression is evaluated but still may be bound in an expression!

Traits to Customize Storage

```
template<typename T>
struct expression_traits{
```

```
  typedef const T& const_reference;
};
template<typename T>
struct expression_traits<ScalarExpression<T> >
{
  typedef ScalarExpression<T> const_reference;
};
```

- Save scalar expressions by value to avoid dangling references.
- Default is storage by reference , of course.

Automatic Expression Creation by Operator Overloading

```
template<typename L1, typename T>
Expression<ScalarExpression<L1>,std::plus<T>, Array<T> >
operator+(const L1& l, const Array<T>& array){
  return Expression<ScalarExpression<L1>,std::plus<T>, Array<T>
      >(ScalarExpression<L1>(l),array);
}
template<typename R1, typename T>
Expression<Array<T>,std::plus<T>, ScalarExpression<R1> >
operator+(const Array<T>& array, const R1& r){
  return Expression<Array<T>,std::plus<T>, ScalarExpression<R1>
      >(array,ScalarExpression<R1>(r));
}
template<typename T>
Expression<Array<T>,std::plus<T>, Array<T> >
operator+(const Array<T>& l, const Array<T>& r){
  return Expression<Array<T>,std::plus<T>, Array<T> >(l,r);
}

template<typename L1, typename Op1, typename R1,
         typename L2, typename Op2, typename R2>
Expression<Expression<L1,Op1,R1>,
           std::plus<typename Expression<L1,Op1,R1>::value_type>,
           Expression<L2,Op2,R2> >
operator+(const Expression<L1,Op1,R1>& l,
          const Expression<L2,Op2,R2>& r)
{
  return Expression<Expression<L1,Op1,R1>,
    std::plus<typename Expression<L1,Op1,R1>::value_type>,
    Expression<L2,Op2,R2> >(l,r);
}
```

Assignment of Expressions

```
template<typename T>
class Array
{
public:
  template<typename L, typename Op, typename R>
  Array& operator=(const Expression<L,Op,R>& expr){
    for(std::size_t i=0; i < size_; ++i)
      vals[i]=expr[i];
  }
```

```
  const T& operator[](std::size_t i) const{
    return vals[i];
  }

  std::size_t size()const{
    return size_;
  }
private:
  std::size_t size_;
  T* vals;
};
```

Review of the Expression Setup

```
Array<double> x(1000000), y(1000000);
x=1.2*x+x*y;
```

1. Compiler applies leftmost * operation, which is scalar-times-array and returns the type

   ```
   Expression<ScalarExpression<double>,std::multiply<double>,Array<double> >
   ```

2. Second multiplication is evaluated, which is array-times-vector and returns the type

   ```
   Expression<Array<double>,std::multiplies<double>,Array<double> >
   ```

3. Finally + is evaluated, which expression-plus-expression and returns the type

   ```
   Expression<Expression<ScalarExpression<double>,std::multiply<double>,Array<double>
       >,
               std::plus<double>,
               Expression<Array<double>,std::multiplies<double>,Array<double>
                   > >
   ```

Notes on Expression Templates

- Implementation is of expression templates is rather complex.
- Justified by the performance gain. In our example the expression templates are 4-5 times faster than the naive implementation!
- Using expression templates to reuse intermediate result without evaluating it early is cumbersome:

  ```
  Expression<Expression<Array<T>,std::plus<T>,Array<T> >,
              std::plus<T>,
              Expression<Array<T>,std::minus<T>,Array<T> >
  > intermediate = a + b + (c - d);
  ```

- Expression templates are not limited to numeric computations. E.g. the boost lambda expression library is implemented using expression templates.

19 C++0X

19.1 Automatische Typerkennung

Automatische Typerkennung (GCC 4.4,ICC 11,VS2010)

Der Compiler kann den benötigten Typ aus der rechten Seite einer Zuweisung automatisch ermitteln:

```
auto x = expression;
```

z.B.

```
auto x = 7;
```

Damit lassen sich insbesondere Schleifen eleganter schreiben. Statt bisher:

```
template<typename T> void printall(const std::vector<T> &v)
{
  for (typename std::vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
    std::cout << *p << "\n";
}
```

genügt:

```
template<typename T> void printall(const std::vector<T>& v)
{
  for (auto p = v.begin(); p!=v.end(); ++p)
    std::cout << *p << "\n";
}
```

Besonders hilfreich ist die automatische Typerkennung auch in Templatefunktionen mit mehreren Templateargumenten:

```
template<typename T, typename U> void (const std::vector<T>& vt, const
    std::vector<U>& vu)
{
    // ...
    auto tmp = vt[i]*vu[i];
    // ...
}
```

Typdefinition mit `decltype` (GCC 4.3,ICC 11,VS2010)

Mit `decltype` lässt sich der Typ bestimmen, der bei der Auswertung eines Ausdrucks entsteht (z.B. durch implizite Typkonvertierung):

```
void f(const std::vector<int> &a, std::vector<float> &b)
{
  typedef decltype(a[0]*b[0]) TmpType;
  for (int i=0; i<b.size(); ++i)
  {
    TmpType* p = new Tmp(a[i]*b[i]);
    // ...
  }
  // ...
}
```

Automatische Return-Type-Bestimmung

Ein Problem ist es oft auch den geeigneten Returntype zu bestimmen, wenn ein Template mehr als ein Argument hat:

```
template<typename T, typename U>
??? mul(T x, U y)
{
  return x*y;
}
```

Oft wird dies mit Promotion Traits gelöst. Wir würden hier gerne `decltype` verwenden. Dies ist aber zur Definition des Returntype nicht möglich, weil die Variablen dort noch nicht bekannt sind:

```
template<typename T, typename U>
decltype(x*y) mul(T x, U y) // scope problem!
{
  return x*y;
}
```

Folgendes wäre zulässig (aber hässlich):

```
template<typename T, typename U>
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y)   // ugly! and error prone
{
  return x*y;
}
```

Mit Hilfe des Schlüsselwortes `auto` lässt sich die Definition ans Ende der Argumentliste schieben, wo die Variablen dann definiert sind:

```
template<typename T, typename U>
auto mul(T x, U y) -> decltype(x*y)
{
  return x*y;
}
```

`auto` heisst dabei "ein return type der später bestimmt wird."

Es handelt sich hier eigentlich nicht um ein Problem des Returntypes allein sondern vor allem des Gültigkeitsbereichs. Die extern definierte Methode `erase` muss bisher geschrieben werden als:

```
struct List
{
  struct Link { /* ... */ };
  Link* erase(Link* p);   // remove p and return the link before p
  // ...
};

List::Link* List::erase(Link* p)
{ /* ... */ }
```

Alternativ kann nun auch geschrieben werden:

```
auto List::erase(Link* p) -> Link*
{ /* ... */ }
```

Automatische For-Schleife über Container (NYI)

```
template<typename T>
void f(const vector<T> &v)
{
  // traditionelle Notation
  for (typename std::vector<T>::const_iterator i=v.begin();i!=v.end();++i)
    *i *= 2;
  // neue Notation (read only)
  for (auto x : v)
    std::cout << x << '\n';
  for (auto &x : v) // using a reference to allow us to change the value
    ++x;
}

for (const auto x : { 1,2,3,5,8,13,21,34 })
  cout << x << '\n';
```

For-Schleifen, die über einen ganzen Container iterieren lassen sich jetzt besonders elegant schreiben:

```
template<typename T>
void f(const vector<T> &v)
{
  // traditionelle Notation
  for (typename std::vector<T>::const_iterator i=v.begin();i!=v.end();++i)
    *i *= 2;
  // neue Notation (read only)
  for (auto x : v)
    std::cout << x << '\n';
  for (auto &x : v) // using a reference to allow us to change the value
    ++x;
}
```

19.2 Initialisierungslisten, Einheitliche Initialisierung

Initialisierungslisten (GCC 4.5)

Ebenso wie C-Arrays lassen sich Container in Zukunft bei der Definition mit einer Liste von Werten initialisieren, auch verschachtelt:

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string,string>> languages = {
  {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};
// this is only supported by g++ >= 4.5
map<vector<string>,vector<int>> years = {
  { {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000} },
  { {"Martin", "Ritchards"} {1982, 2003, 2007} },
  { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

Diese Art von Listen kann auch von Funktionen verwendet werden. Der Argumenttyp ist ein Container vom Typ `initializer_list<T>`:

```
void f(std::initializer_list<int>);
f({1,2});
f({23,345,4567,56789});
f({});  // the empty list
f{1,2}; // error: function call ( ) missing
```

Damit ein Container mit Initialisierungslisten funktioniert muss ein Konstruktor definiert werden, der eine `initializer_list` als Argument bekommt:

```
template<typename E> class vector
{
  public:
    vector (std::initializer_list<E> s) // initializer-list constructor
    {
      reserve(s.size());  // get the right amount of space
      uninitialized_copy(s.begin(), s.end(), elem);   // initialize elements
          (in elem[0:s.size())
      sz = s.size();  // set vector size
    }

    // ... as before ...
};
```

Mit Hilfe von Initialisierungslisten lässt sich ein adhoc-loop Schreiben wie in einer Scriptsprache:

```
for (const auto x : { 1,2,3,5,8,13,21,34 })
  cout << x << '\n';
```

Vergleich traditioneller Konstruktor, Initialisierungsliste:

```
std::vector<double> v1(7);  // ok: v1 has 7 elements
v1 = 9;                     // error: no conversion from int to vector
std::vector<double> v2 = 9; // error: no conversion from int to vector

void f(const std::vector<double>&);
f(9);                       // error: no conversion from int to vector

std::vector<double> v1{7};  // ok: v1 has 1 element (with its value 7)
v1 = {9};                   // ok v1 now has 1 element (with its value 9)
std::vector<double> v2 = {9}; // ok: v2 has 1 element (with its value 9)
f({9});                     // ok: f is called withthe list { 9 }

std::vector<std::vector<double>> vs =
{
  std::vector<double>(10),  // ok: explicit construction (10 elements)
  std::vector<double>{10},  // ok explicit construction (1 element with the value 10)
  10                        // error: vector's constructor is explicit
};
```

Initialisierungslisten können auch als read-only Container verwendet werden:

```
void f(initializer_list<int> args)
{
  for (auto p=args.begin(); p!=args.end(); ++p)
    cout << *p << "\n";
}
```

Bei Zuweisungen erfolgt in C++ oft eine Reduktion der Genauigkeit durch automatische Typkonvertierung. Bei Initialisierungslisten ist das nicht zulässig.

```
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d};      // error: narrowing (double to int)
char x3{7};     // ok: even though 7 is an int, this is not narrowing
std::vector<int> vi = { 1, 2.3, 4, 5.6 };    // error: double to int narrowing
```

Vereinheitlichte Initialisierung

Die Initialisierung von Variablen ist in C++ sehr unterschiedlich geregelt:

```
std::string a[] = { "foo", "bar" };       // ok: initialize array variable
std::vector<std::string> v = { "foo", "bar" }; // error: initializer list for
    non-aggregate vector
void f(string a[]);

f( { "foo", "bar" } );  // syntax error: block as argument
int a = 2;              // "assignment style"
int[] aa = { 2, 3 };    // assignment style with list
complex z(1,2);         // "functional style" initialization
x = Ptr(y);             // "functional style" for conversion/cast/construction

int a(1);   // variable definition
int b();    // function declaration
int b(foo); // variable definition or function declaration
```

Mit Initialisierungslisten lässt sich das alles vereinheitlichen:

```
X x1 = X{1,2};
X x2 = {1,2};   // the = is optional
X x3{1,2};
X* p = new X{1,2};
struct D : X
{
    D(int x, int y) : X{x,y} { /* ... */ };
};
struct S
{
    int a[3];
    S(int x, int y, int z) : a{x,y,z} { /* ... */ }; // solution to old problem
};
X x{a};
X* p = new X{a};
z = X{a};          // use as cast
f({a});            // function argument (of type X)
return {a};        // function return value (function returning X)
```

19.3 constexpr

constexpr (NYI)

`constexpr` werden gleich zur Übersetzungszeit ausgewertet (und ersetzen damit einen guten Teil des Template Metaprogrammings). Folgendes `switch` statement wäre bisher nicht zulässig:

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2)
{
  return Flags(int(f1)|int(f2));
}

void f(Flags x)
{
  switch (x)
  {
    case bad:          /* ... */ break;
    case eof:          /* ... */ break;
```

```
    case bad|eof:      /* ... */ break;
    default:           /* ... */ break;
  }
}
```

Eine `constexpr` muss zur Übersetzungszeit evaluierbar sein:

```
constexpr int x1 = bad|eof; // ok

void f(Flags f3)
{
    constexpr int x2 = bad|f3;  // error: can't evaluate at compile time
    int x3 = bad|f3;        // ok
}
```

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 becomes 1
```

19.4 Lambdafunktionen

Lambdafunktionen (GCC 4.5,ICC 11,VS2010)

Für ein Sortierkriterium muss bisher umständlich ein Funktor im globalen Namensraum definiert werden. Lambdafunktionen ermöglichen die Definition on the fly:

```
std::vector<int> v = {50, -10, 20, -30};

std::sort(v.begin(), v.end());  // the default sort
// now v should be { -30, -10, 20, 50 }

// sort by absolute value:
std::sort(v.begin(), v.end(), [](int a, int b){ return abs(a)<abs(b); });
// now v should be { -10, 20, -30, 50 }
```

Auch in anderen STL-Algorithmen lassen sich diese Lambdafunktionen gut verwenden:

```
void f(vector<Record>& v)
{
  std::vector<int> indices(v.size());
  int count = 0;
  fill(indices.begin(),indices.end(),[&count](){ return ++count; });

  // sort indices in the order determined by the name field of the records:
  std::sort(indices.begin(), indices.end(), [&](int a, int b) { return
      v[a].name<v[b].name; });
  // ...
}
```

`[&count]` bedeutet, dass die Funktion über eine Referenz Zugriff auf die Variable `count` in der Umgebung hat. `[=count]` würde heißen, dass eine Kopie übergeben wird. Bei `[&]` werden alle Variablen zugänglich gemacht, bei `[=]` Kopien aller Variablen (was teuer werden kann).

Es kann hilfreich sein, kompliziertere Vergleichsoperatoren doch als Funktor zu realisieren:

```
void f(vector& v)
{
  vector indices(v.size());
  int count = 0;
  fill(indices.begin(),indices.end(),[&](){ return ++count; });
  struct Cmp_names
  {
    const vector<Record> &vr;
    Comp_names(const vector<Record> &r) : vr(r) {}
    bool operator()(Record& a, Record &b) const { return vr[a]<vr[b]; }
  };
  // sort indices in the order determined by the name field of the records:
  std::sort(indices.begin(), indices.end(), Cmp_names(v));
  // ...
}
```

Lokale Funktordefinition (GCC 4.5,ICC 11,VS2010)

In C++0x ist es erlaubt Funktoren lokal innerhalb einer Funktion zu definieren:

```
void f(vector<X>& v)
{
  struct Less
  {
    bool operator<(const X& a, const X& b) { return a.v<b.v; }
  };
  sort(v.begin(), v.end(), Less());    // C++98: error: Less is local
                                       // C++0x: ok
}
```

19.5 Template Verbesserungen

Template Aliases (NYI)

Templatenamen inklusive Argumenten können ziemlich lange werden. Für Templates lassen sich ähnlich wie bei Typedefs neue Namen definieren:

```
template<class T>
using Vec = std::vector<T,My_alloc<T>>;  // standard vector using my allocator

Vec<int> fib = { 1, 2, 3, 5, 8, 13 };    // allocates elements using My_alloc

std::vector<int,My_alloc<int>> verbose = fib; // verbose and fib are of the
    same type
```

Spitze Klammern bei Templates (GCC 4.3,ICC 11,VS2010)

In C++0x ist es jetzt nicht mehr notwendig ein Leerzeichen zwischen die spitzen Klammern am Ende einer Templateargumentliste zu setzen, da alle existierenden Compiler das sowieso erkennen können:

```
list<vector<string>> lvs;
```

Variadic Templates

Die Funktion `printf()` ist in C++ bisher immer noch eine C-Funktion. In C++0x lässt sich dies ändern. Die Anwendung sieht wie folgt aus:

```
const string pi = "pi";
const char* m = "The␣value␣of␣%s␣is␣about␣%g␣(unless␣you␣live␣in␣%s).\n";
printf(m,  pi, 3.14159, "Indiana");
```

Der einfachste Fall von `printf()` ist der in dem es außer dem Formatstring keine Argumente gibt:

```
void printf(const char* s)
{
  while (s && *s)
  {
    if (*s=='%' && *++s!='%') // make sure that there wasn't meant to be more
        arguments
                              // %% represents plain % in a format string
        throw 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>      // note the "..."
void printf(const char* s, T value, Args... args)   // note the "..."
{
  while (s && *s)
  {
    if (*s=='%' && *++s!='%') // a format specifier (ignore which one it is)
    {
      std::cout << value;     // use first non-format argument
      return printf(++s, args...);  // "peel off" first argument
    }
    std::cout << *s++;
  }
  throw std::runtime error("extra␣arguments␣provided␣to␣printf");
}
```

Tuples (GCC 4.5)

Eine weitere Anwendung von variadic Templates sind Tuple, eine Verallgemeinerung von Pairs auf beliebig viele Komponenten. Deren Typ kann mit Hilfe von `auto` und `make_tuple` auch automatisch generiert werden:

```
std::tuple<std::string,int> t2("Kylling",123);
auto t = make_tuple(string("Herring"),10, 1.23);
// t will be of type tuple<string,int,double>
std::string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

Externe Templates (g++,ICC 11,)

Bei Projekten, die aus vielen Einzeldateien bestehen, werden Templates oft an mehreren Stellen instanziiert. Durch die Deklaration von Templates unter Verwendung des Schlüsselwortes `explicit` erfolgt das nicht.

```
#include "MyVector.h"

extern template class MyVector<int>; // Suppresses implicit instantiation
    below --
                // MyVector<int> will be explicitly instantiated elsewhere

void foo(MyVector<int>& v)
{
    // use the vector in here
}
```

Die eigentliche Instantiierung erfolgt an einer wohldefinierten Stelle explizit:

```
#include "MyVector.h"

template class MyVector<int>; // Make MyVector available to clients (e.g.,
    of the shared library
```

19.6 Multitasking/Multithreading

Multithreading (NYI)

In C++ gibt es eine Platformübergreifende Art Threads zu erzeugen, um Aufgaben parallel erledigen zu lassen.

```
#include<thread>
struct Foo
{
  std::string msg;
  Foo(const string &m) : msg(m)
  {}
  void operator()()
  {
    cout << "Hier␣ist␣Thread␣" << msg << endl;
  }
};

void run()
{
  Foo foo("Nummer␣1");
  std::thread th1(foo);
  std::thread th2{[]() { cout << "Hier␣ist␣Thread␣Nummer␣2" << endl; } };
  th1.join();
  th2.join();
}
```

Mutual Exclusion/Locks (NYI)

Bei Multithreading-Applikationen ist es notwendig, den Zugriff auf gemeinsame Variablen zu kontrollieren, so dass z.B. bestimmte Variablen nur von einem Prozess geändert werden können. Eine Möglichkeit dies zu realisieren ist mutual exclusion. C++0x bietet dafür direkte Unterstützung. Ein `mutex` kann einen Berech sperren, so dass er nur von einem Prozess betreten werden kann. Ein `lock` kann ihn verwalten:

```
std::mutex m;
int sh; // shared data
// ...
void f()
{
  // ...
  std::unique_lock lck(m);
  // manipulate shared data:
  sh+=1;
}
```

Es lässt sich auch Prüfen, ob ein Bereich betreten werden darf ohne dass der Prozess gleich steckenbleibt:

```
std::mutex m;
int sh; // shared data
// ...
void f()
{
  // ...
  std::unique_locklck(m,std::defer_lock); // make a lock, but don't aquire the
      mutex
  // ...
  if (lck.try_lock())
  {
    // manipulate shared data:
    sh+=1;
  }
  else
  {
    // maybe do something else
  }
}
```

Threaderzeugung mit `async` (NYI)

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, const 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})};

  return f0.get()+f1.get()+f2.get()+f3.get();
}

int main()
{
  double result=comp(blub);
  std::cout << result << std::endl;
}
```

19.7 Statische Assertions

Statische Assertions (GCC 4.3,ICC 11,VS2010)

Mit der Funktion `assert()` lassen sich nur dann Fehler finden, wenn Sie auch ausgeführt wird. Dies setzt das Schreiben umfangreicher Tests voraus. `static_assert` hingegen wird bereits bei der Übersetzung geprüft:

```
static_assert(expression,std::string);
```

Der Compiler überprüft den Ausdruck und schreibt den Inhalt des String, wenn der Ausdruck `false` ergibt. Zum Beispiel:

```
static_assert(sizeof(long) >= 8, "64-bit code generation required for this
    library.");
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y),"unexpected padding in S");
```

Ein `static_assert` kann nützlich sein um Annahmen über das Programm und seine Behandlung durch den Compiler zu prüfen. Da `static_assert` zur Übersetzungszeit geprüft wird, können damit keine Bedingungen formuliert werden, die sich erst zur Laufzeit prüfen lassen:

```
int f(int* p, int n)
{
  static_assert(p==0,"p is not null");     // error: static_assert() expression
      not a constant expression
  // ...
}
```

19.8 Neue Container

19.8.1 Sequence Container

`std::array` (GCC 4.4)

Das `std::array` ist ein Container, dessen Größe bereits zur Übersetzungszeit bekannt ist und der möglichst Overhead erzeugt (Speicherbedarf, Laufzeit). Er soll das C-Array ersetzen:

```
std::array<int,6> a = { 1, 2, 3 };
a[3]=4;
int x = a[5];        // x becomes 0 because default elements are zero
    initialized
```

```
int* p1 = a;        // error: std::array doesn't implicitly convert to a
    pointer
int* p2 = a.data(); // ok: get pointer to first element

std::array<int> a3 = { 1, 2, 3 };    // error: size unknown/missing
std::array<int,0> a0;        // ok: no elements
int* p = a0.data();     // unspecified; don't try it
```

Er lässt sich wie ein `std::vector` verwenden (ausser dass die Grösse fix ist):

```
template<class C> C::value_type sum(const C& a)
{
    return accumulate(a.begin(),a.end(),0);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
int x1 = sum(a10);
int x2 = sum(a1000);
int x3 = sum(v);
```

Da beim C-Array nur ein Pointer übergeben wird, der auch für einen Pointer der Basisklasse eingesetzt werden darf, kann es zur Vermischung von Äpfeln und Birnen kommen, die mit dem `std::array` vermieden wird:

```
struct Apple : Fruit { /* ... */ };
struct Pear : Fruit { /* ... */ };

void nasty(array<Fruit*,10>& f)
{
  f[7] = new Pear();
};

array<Apple*,10> apples;
// ...
nasty(apples);  // error: can't convert array<Apple*,10> to array<Fruit*,10>;
```

`std::forward_list` (GCC 4.5)

Es gibt auch eine sehr kompakte neue einfach-verkettete Liste:

```
template <ValueType T, Allocator Alloc = allocator<T> >
class forward_list
{
   public:
    // the usual container stuff
    // no size()
    // no reverse iteration
    // no back() or push_back()
};
```

19.8.2 Smart Pointer

`unique_ptr` (GCC 4.5)

Normale Pointer sind nicht sicher, wenn es zwischen der Allokation von Speicher und seiner Freigabe zu einer Exception kommt:

```
X* f()
{
  X* p = new X;
  // do something - maybe throw an exception
  return p;
}
```

Eine Lösung dieses Problems ist es den Pointer auf das allozierte Objekt in einem `unique_ptr` zu speichern:

```
X* f()
{
  std::unique_ptr<X> p(new X);      // or {new X} but not = new X
  // do something - maybe throw an exception
  return p.release();
}
```

Ein Unique-Pointer hat keinen Copykonstruktor, nur einen Movekonstruktor. Geht er out-of-scope wird ein passendes `delete` aufgerufen.

Noch besser ist es gleich einen `unique_ptr` zurückzuliefern:

```
std::unique_ptr<X> f()
{
  std::unique_ptr<X> p(new X);      // or {new X} but not = new X
  // do something - maybe throw an exception
  return p;   // the ownership is transferred out of f()
}
```

`shared_ptr` (GCC 4.5)

Ein `shared_ptr` kann kopiert werden. Er zählt die Anzahl der Kopien mit und zerstört das Objekt, wenn die letzte Kopie zerstört wird:

```
void test()
{
  shared_ptr<int> p1(new int);    // count is 1
  {
    shared_ptr<int> p2(p1); // count is 2
    {
      shared_ptr<int> p2(p1); // count is 3
    }   // count goes back down to 2
  } // count goes back down to 1
}   // here the count goes to 0 and the int is deleted.
```

19.9 Reguläre Ausdrücke/RawString Literale

Reguläre Ausdrücke (VS2010)

Wie in vielen anderen Programmiersprachen lassen sich auch in C++0x 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;
}
```

Raw String Literale (GCC 4.5)

Das schreiben von regulären Ausdrücken kann dadurch schwer lesbar werden, dass in normalen C-Strings der Backslash für Sonderzeichen verwendet wird und deshalb als \\ geschrieben werden muss. Das macht das ganze schwer lesbar:

```
    string s = "\\w\\\\\\w";     // I hope I got that right
```

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 ("])"
```

19.10 Benutzerdefinierte Literale

Literale

Mit Hilfe von Literalen lassen sich verschiedene Konstanten charakterisieren:

```
123   // int
1.2   // double
1.2F  // float
'a'   // char
1ULL  // unsigned long long
0xD0  // hexadecimal unsigned
"as"  // string
```

Benutzerdefinierte Literale

Es wäre schön, wenn sich hier weitere Literale definieren liesen, z.B.

```
"Hi!"s            // string, not ‘‘zero-terminated array of char’’
1.2i              // imaginary
123.4567891234df  // decimal floating point (IBM)
101010111000101b  // binary
123s              // seconds
123.56km          // not miles! (units)
12345678901234567890123456789012345678901234567890x   // extended-precision
```

In C++0x ist dies möglich, z.B. für komplexe Zahlen und Strings:

```
constexpr complex<double> operator "" i(long double d)  // imaginary literal
{
    return {0,d};   // complex is a literal type
}

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"); // pass pointer to char*
f("Hello"s);    // pass (5-character) string object
f("Hello\n"s);  // pass (6-character) string object

auto z = 2+1i;  // complex(2,1)
```

Es lassen sich auch direkt C-Strings übergeben:

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);
f(12345678901234567890123456789012345678901234567890x);
```

aber nicht immer:

```
string operator"" S(const char* p);     // waning: this will not work as
    expeceted

"one␣two"S; // error: no applicable literal operator
```

19.11 Zufallszahlen

Zufallszahlen (GCC 4.5,VS2010)

C++0x stellt mächtige Zufallszahlengeneratoren bereit, die mit verschiedenen Verteilungen arbeiten können, z.B. gleichverteilte Zufallszahlen:

```
default_random_engine re {};        // the default engine
uniform_int<> one_to_six {1,6};  // distribution that maps to the ints 1..6

using my_generator = variate_generator<default_random_engine,uniform_int<>>;
      // type of generator

my_generator dice(re,one_to_six);   // make a generator

int x = dice(); // roll the dice: x becomes a value in [1:6]
```

Damit lassen sich dann auch einfachere Funktionen bauen:

```
int rand_int(int low, int high)
{
  static variate_generator<default_random_engine, uniform_int<>> r
      {default_random_engine(),uniform_int<>(low,high)};
  return r();
}
```

Es gibt auch normalverteilte Zufallszahlen:

```
using my_normal = variate_generator<default_random_engine,
    normal_distribution<>>;

default_random_engine re;   // the default engine
normal_distribution<> nx(31 /* mean */,8 /* sigma */);

my_normal norm(re,nx);

vector<int> mn(64);

int main()
{
  for (int i = 0; i<1200; ++i)
    ++mn[round(norm())]; // generate

  for (int i = 0; i<mn.size(); ++i)
  {
    cout << i << '\t';
    for (int j=0; j<mn[i]; ++j) cout << '*';
    cout << '\n';
  }
}
```

19.12 Zeitmessung

Zeitmessung (GCC 4.5)

Zur Zeitmessung müssen bisher Systembibliotheken verwendet werden. C++0x bietet eine integrierte Möglichkeit zur Zeitmessung an. Dabei gibt es verschiedene Uhren und Datentypen um Zeitpunkte und Zeitspannen zu speichern.

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

void messen()
{
  monotonic_clock::time_point start = monotonic_clock::now();
  sleep(2); // do something
  auto now = monotonic_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";
}
```

19.13 Initialisierung von Attributen

Initialisierung von Attributen (NYI)

Bisher können nur statische Attribute direkt bei ihrer Definition initialisiert werden. Andere Attribute müssen im Konstruktor initialisiert werden:

```
class A
{
  public:
    A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor␣run") {}
    A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor␣run")
        {}
    A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor␣run") {}
    int a, b;
  private:
    HashingFunction hash_algorithm;  // Cryptographic hash to be applied to
        all A instances
    std::string s;                   // String indicating state in object
        lifecycle
};
```

Bei einigen Attributen wird hier immer der gleiche Wert verwendet, was das ganze unübersichtlich und anfällig macht (bei Änderungen wird gern mal ein Konstruktor übersehen).

In C++0x ist es erlaubt auch andere Attribute gleich bei der Definition zu initialisieren. Der Compiler merkt sich die Initialisierung und fügt sie bei den Konstruktoren hinzu, wenn dort nichts anderes bestimmt wurde:

```
class A
{
  public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(D d) : a(7), b(g(d)) {}
    int a, b;
  private:
    HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be
        applied to all A instances
    std::string s{"Constructor␣run"};       // String indicating state in
        object lifecycle
};
```

So lassen sich auch alle Werte die häufig gleich sind initialisieren und bei den Konstruktoren stehen nur noch die Werte die für diesen Konstruktor anders initialisiert werden:

```
class A
{
  public:
    A() {}
    A(int a_val) : a(a_val) {}
    A(D d) : b(g(d)) {}
    int a = 7;
    int b = 5;
  private:
    HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be
        applied to all A instances
    std::string s{"Constructor␣run"};       // String indicating state in
        object lifecycle
};
```

19.14 Konstruktoren

Aufrufen von Konstruktoren (NYI)

Brauchen zwei Konstruktoren gemeinsame Funktionalität so muss diese bisher in eine weitere Funktion ausgelagert werden:

```
class X
{
    int a;
    validate(int x)
    {
      if (0<x && x<=max)
        a=x;
      else
        throw bad_X(x);
    }
  public:
    X(int x)
    {
      validate(x);
    }
    X()
    {
      validate(42);
    }
    X(string s)
    {
      int x = lexical_cast<int>(s);
      validate(x);
    }
};
```

In C++0x kann ein Konstruktor einfach einen anderen Konstruktor aufrufen:

```
class X {
    int a;
  public:
    X(int x)
    {
      if (0<x && x<=max)
        a=x;
      else
        throw bad_X(x);
    }
    X() : X{42}
    {}
    X(string s) : X{lexical_cast<int>(s)}
    {}
    // ...
};
```

Vererben von Konstruktoren (NYI)

In C++0x lassen sich Konstruktoren von einer Basisklasse übernehmen:

```
class Derived : public Base
{
  public:
    using Base::f;     // lift Base's f into Derived's scope -- works in C++98
    void f(char);      // provide a new f
    void f(int);       // prefer this f to Base::f(int)
```

```
    using Base::Base; // lift Base constructors Derived's scope -- C++0x only
    Derived(char);    // provide a new constructor
    Derived(int);     // prefer this constructor to Base::Base(int)
    // ...
};
```

Werden in einer abgeleiteten Klassen neue Attribute hinzugefügt, so werden diese vom Basisklassenkonstruktor nicht initialisiert:

```
struct B1
{
  B1(int)
  {}
};

struct D1 : B1
{
  using B1::B1; // implicitly declares D1(int)
  int x;
};

void test()
{
  D1 d(6);    // Oops: d.x is not initialized
  D1 e;       // error: D1 has no default constructor
}
```

Dies lässt sich mit Hilfe von Initialisierungen vermeiden:

```
struct D1 : B1
{
  using B1::B1;   // implicitly declares D1(int)
  int x{0};   // note: x is initialized
};

void test()
{
    D1 d(6);    // d.x is zero
}
```

19.15 Defaultfunktionen/Löschen von Funktionen

Defaultfunktionen (GCC 4.4)

In C++ werden für eine Klasse automatisch ein argumentloser Konstruktor, Copy-Konstruktor, Zuweisungsoperator, Adressoperator und Destruktor erzeugt.

Bei manchen Klassen ist die Kopierbarkeit unerwünscht. Bisher muss man dazu einen eigenen Copy-Konstruktor und Zuweisungsoperator definieren und `private` machen. In C++0x lassen sich die Defaultmethoden einfach löschen:

```
class X
{
    // ...
    X& operator=(const X&) = delete;    // Disallow copying
    X(const X&) = delete;
};
```

Sollen die Defaultmethoden auf jeden Fall erzeugt werden, lässt sich auch dies ausdrücken:

```
class Y
{
    // ...
    Y& operator=(const Y&) = default;   // default copy semantics
    Y(const Y&) = default;
};
```

Das Löschen von Funktionen kann auch dazu verwendet werden eine implizite Typkonvertierung auszuschliessen:

```
struct Z
{
    // ...

    Z(long long);      // can initialize with an long long
    Z(long) = delete; // but not anything less
};
```

19.16 Rvalue Referenzen

Rvalues

Rvalues sind Ausdrücke die nur auf der rechten Seite einer Zuweisung stehen können und die keine Adresse im Speicher haben:

```
void incr(int& a)
{
  ++a;
}

int main()
{
  int i = 0;
  incr(i);    // i becomes 1
  incr(0);    // error: 0 not an lvalue
}
```

Wenn `incr(0)` erlaubt wäre, dann würde entweder eine temporäre Variable hochgezählt, die nie weiter verwendet wird oder der Wert von 0 wäre in Zukunft 1 (es gab so einen Fall in frühen Fortran Compilern).

RValues

Wenn wir den Inhalt von zwei Variablen vertauschen möchten, dann wird eine temporäre Kopie benötigt. Das Erstellen dieser Kopie kann relativ aufwändig sein und sie wird nicht weiter gebraucht:

```
template<class T> swap(T& a, T& b)      // "old style swap"
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

Rvalue Referenzen (GCC 4.3,VS2010)

Eine Lösung ist es einem Objekt zu erlauben Zugriff auf die inneren Daten von temporären Variablen zu bekommen und sich diese zu nehmen und durch einen billig zu erzeugenden anderen Wert zu ersetzen. Dies erfolgt mit Hilfe von move-Konstruktoren und move-Zuweisungen:

```
template<class T>
class vector
{
  // ...
  vector(const vector&);           // copy constructor
  vector(vector&&);              // move constructor
  vector& operator=(const vector&);   // copy assignment
  vector& operator=(vector&&);        // move assignment
};  // note: move constructor and move assignment takes non-const &&
    // they can, and usually do, write to their argument
```

`&&` bezeichnet eine "Rvalue Referenz". Eine Rvalue Referenz kann nur auf einen Rvalue erstellt werden, nicht auf einen Lvalue:

```
X a;
X f();
X &r1 = a;      // bind r1 to a (an lvalue)
X &r2 = f();        // error: f() is an rvalue; can't bind

X &&rr1 = f();  // fine: bind rr1 to temporary
X &&rr2 = a;    // error: bind a is an lvalue
```

Mit Hilfe von Rvalue Referenzen lässt sich das swap viel effizienter schreiben:

```
template<class T>
void swap(T& a, T& b)    // "perfect swap" (almost)
{
    T tmp = move(a);     // could invalidate a
    a = move(b);         // could invalidate b
    b = move(tmp);       // could invalidate tmp
}
```

19.17 Nullptr Konstante

Nullptr Konstante (VS2010)

Es gibt viele Varianten um einen Pointer als ungültig zu initialisieren: `NULL`, `null` oder `0`. Um dem Abzuhelfen wird in C++0x das Schlüsselwort `nullptr` eingeführt. Es lässt sich nicht automatisch nach `int` umwandeln, ein Pointer der auf `0` initialisiert wurde ist aber einem `nullptr` äquivalent.

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0;            // 0 still works and p==p2

void f(int);
void f(char*);

f(0);          // call f(int)
f(nullptr);    // call f(char*)

void g(int);
g(nullptr);        // error: nullptr is not an int
int i = nullptr;   // error nullptr is not an int
```

19.18 `long long`

`long long` (GCC 4.3,ICC 11,VS2010)

In C++0x gibt es einen Datentyp `long long` mit mindestens 64bit Länge. Das zugehörige Literal ist `LL`:

```
long long x = 9223372036854775807LL;
```

Es gibt weder eine Variante `long long long` noch ein `short long long`.

19.19 Weitere Erweiterungen

- Stark-typisierte Enums (GCC 4.4)
- Explizite Typkonvertierung (GCC 4.5)
- Verbesserte Unions (GCC 4.6)
- Funktionswrapper (GCC 4.5)
- Unicode Stringliterale (GCC 4.5)