C++ für Wissenschaftliches Rechnen

Dan Popović

Interdisziplinäres Institut für Wissenschaftliches Rechnen, Universät Heidelberg

8. April 2009

C++ für Wissenschaftliches Rechnen

Anforderungen an die Programmiersprache

- → Effizienz...
  - des Programms
  - der Entwicklung
- → Hardware-nahe Programmiersprachen
- → Integration mit existierendem Code
- → Abstraktion
- →

Vergleich von C++ mit anderen Sprachen

Fortran & C

+ schneller Code
+ gute Optimierungen

- nur prozedurale Sprachen
- wenig Flexibilität
- schlechte Wartbarkeit

C++

+ gute Wartbarkeit
+ schneller Code
+ gute Integration mit Fortran und C Bibliotheken
+ hoher Abstraktionsgrad

- schwerer zu optimieren
- meistens mehr Speicherverbrauch

Vergleich von C++ mit anderen Sprachen

Fortran & C

+ schneller Code
+ gute Optimierungen

- nur prozedurale Sprachen
- wenig Flexibilität
- schlechte Wartbarkeit

C++

+ gute Wartbarkeit
+ schneller Code
+ gute Integration mit Fortran und C Bibliotheken
+ hoher Abstraktionsgrad

- schwerer zu optimieren
- meistens mehr Speicherverbrauch

Literatur

Literatur zu C++

- B. Stroustrup: C++ – Die Programmiersprache (Die Bibel)
- A. Willms: C++ Programmierung (Für Anfänger gut geeignet)
- S. Oualline: Practical C++-Programming
- O'Reilly-Verlag: Scientific C++

Grundlegende vorausgesetzte C++-Kenntnisse

Um die Vorzüge von C++ auszunutzen, sind abstrakte Techniken notwendig. Folgende grundlegenden Konzepte sind als Basis unumgänglich:

- Grundlegende Datentypen und Kontrollstrukturen:
  - `int`, `double`, `bool`, `char`, ...
  - conditionals: `if`, `switch`, ...
  - loops: `for`, `while`
- Grundlegende Programmstrukturen:
  - Funktionen
  - Rekursive und iterative Programmierung
- Zeiger und Referenzen
- Klassen und Vererbung
  - `class` und `struct`
  - `private`, `public`, `protected`
  - Konstruktoren und Destruktoren
  - `public`, `private`-Vererbung
  - (rein) virtuelle Funktionen abstrakte Basisklassen
- Polymorphismus von Funktionen, Überladen von Operatoren
- Dynamische Speicherverwaltung (`new`, `delete`)

Ein erstes Programm: Hello World

Hallo, Welt!

```
// use the iostream extension
#include <iostream>

// main is always the first function to be called
int main(int argc, char** argv)
{
  std::cout << "Hello, World..." << std::endl;

  return 0;
}
```

Klassen und Datentypen

Eine C++-Klasse definiert einen Datentyp. Ein Datentyp ist eine Zustandsmenge mit Operationen, die die Zustände ineinander überführen. Beispiel:

```
class ComplexNumbers
{
public:
  .. .
  void print()
  {
    std::cout << u << "␣+␣i␣*␣" << v << std::endl;
  }

private:
  double u, v;
};
```

Klassen und Datentypen

```
// usage of the complex number class
int main (int argc, char** argv)
{
  ComplexNumber a, b, c;

  a.print();        // output ?
  c = a + b;        // where defined ?

  return 0;
};
```

Klassen und Datentypen

- C++ ermöglicht die Kapselung des Datentyps, d.h. Trennung von Implementierung und Interface.
  - `public`: Interface-Spezifikation,
  - `private`: Daten und Implementierung.
- Von außen kann nur auf Methoden und Daten im `public`-Teil zugegriffen werden.
- Implementierung der Methoden kann ausserhalb der Klasse geschehen.

Konstruktoren

- Der Befehl `ComplexNumber a;` veranlasst den Compiler, eine Instanz der Klasse zu erzeugen.
- Zur Initialisierung wird ein Konstruktor aufgerufen.
- Es können verschiedene Konstruktoren existieren (Polymorphismus!).
- In gewissen Fällen erzeugt der Compiler default-Konstruktoren.

Konstruktoren

Die Klasse `ComplexNumber` mit zwei Konstruktoren:

```
class ComplexNumbers
{
public:
  // some constructors
  ComplexNumber() { u = 0; v = 0; }    // default

  ComplexNumber(double re, double im) // initialize with
  { u = re; v = im; }                 // given numbers

  void print() { ... }

private:
  double u, v;
};
```

Konstruktoren

```
// usage of the complex number class
int main (int argc, char** argv)
{
  ComplexNumber a(3.0,4.0);
  ComplexNumber b(1.0,2.0);
  ComplexNumber c;

  a.print();          // output: 3 + i * 4
  c = a + b;          // where defined ?

  return 0;
};
```

Destruktoren

- Dynamisch erzeugte Objekte können vernichtet werden, falls sie nicht mehr benötigt werden.
- Das Löschen von Objekten übernimmt der Destruktor.
- Destruktoren sind insbesondere auszuimplementieren, wenn die Klasse Zeiger (etwa Felder!) enthält.
- Ebenso bei Verwendung von dynamischen Speicher in einer Klasse.
- Stichworte zur Dynamischen Speicherverwaltung: `new`, `delete`.

Vererbung in C++

Vererbung

- bedeutet, dass das Verhalten eines Datentyps an einen abgeleiteten Datentyp weitergegeben werden kann.
- Die Datentypen stehen dabei in „Ist-ein“ Relation: Ein Mensch ist auch ein Säugetier, d.h. Klasse Mensch ist von Klasse Säugetier abzuleiten.

Vererbung in C++

```
// example of inheritance in C++
class Matrix{
public:
  ...

private:
  double data[3][3]; // (3 x 3)-Matrix
};

// the derived class
class SymMatrix: public Matrix{
public:
  double getEntry(int i, int j) { return data[i][j]; }
      // error: data private in base class
      // performance?
  ...
  // derived constructor calls a constructor of base class
  SymMatrix() : Matrix() { ... }
};
```

Verschiedene Arten der Vererbung in C++

Bei Vererbung ist darauf zu achten, auf welche member die abgeleitete Klasse Zugriff erhält → verschiedene Arten der Vererbung:

- `private`-Vererbung: Alle Elemente der Basisklasse werden private Member der abgeleiteten Klasse.
- `public`-Vererbung: `public`-Member der Basisklasse werden `public`-Member der abgeleiteten Klasse, `private` wird zu `private`.

Verschiedene Arten der Vererbung in C++

- Private member der Basisklasse bleiben immer privat (sonst macht die Kapselung keinen Sinn).
- Problem: `private`-Member sind zu stark gekapselt, `public`-Member überhaupt nicht.
- Ausweg: `protected`-Member, auf die abgeleitete Klassen zugreifen können.

Virtuelle Funktionen

Klassenhierarchie mit virtuellen Funktionen

```
class GeomObject{ // base class for geo objects
public:
  virtual double area() { return 0.0; }
  ...
};

class Triangle : public GeomObject{ // a derived class
public:
  double area()
  {
    return 0.5 * a * h;
  }
  ...
private:
  double h, a;
};
```

Virtuelle Funktionen

Wofür benötigen wir virtuelle Funktionen?

Virtuelle Funktionen

Wenn Basis- und abgeleitete Klasse enthalten Mitglieder gleichen Namens enthalten – Welche Methode wird aufgerufen?

```
int main() {
  GeomObject* geo;
  Triangle t;

  geo = &t;
  std::cout << geo->area << std::endl; // ?

  return 0;
};
```

Virtuelle Funktionen

Lösung:

- Falls nicht anders angegeben, die Methode des Basisobjekts (!).
- Durch das Schlüsselwort `virtual` wird der Aufruf an die abgeleitete Klasse durchgereicht.
- Stichwort **Late Binding**, d.h. Zuordnung Methodenname $\leftrightarrow$ Implementierung erst zur Laufzeit.

Dynamischer Polymorphismus

Die Technik der späten Typ-Bindung mit virtuellen Funktionen hat einen eigenen Namen:

Dynamischer Polymorphismus

- Genaue Typbestimmung zur Laufzeit.
- Realisierung über:
  - Virtuelle Funktionen (*function lookup table*),
  - Überschreiben von Funktionen.

Dynamischer Polymorphismus

Die Technik der späten Typ-Bindung mit virtuellen Funktionen hat einen eigenen Namen:

Dynamischer Polymorphismus

- Genaue Typbestimmung zur Laufzeit.
- Realisierung über:
  - Virtuelle Funktionen (*function lookup table*),
  - Überschreiben von Funktionen.

Vorteile des dynamischen Polymorphismus

- Basisklassen sind Obermengen der abgeleiteten Klassen
- Algorithmen, die auf Basisklasse operieren, können auch auf den abgeleiteten Klassen operieren.
- Beispiel: Liste, die Pointer auf `GeomObject`s speichert. Pointer kann auf ein `Triangle`-Objekt oder jedes andere `GeomObject`-Objekt zeigen!

Abstrakte Basisklassen und Schnittstellen

Oftmals sind virtuelle Funktionen nicht sinnvoll in der Basisklasse definierbar. Dann

- Deklararation der Funktion in der Basisklasse als „rein virtuell“:
  `virtual printArea() = 0`.
- Abgeleitete Klassen müssen rein virtuelle Funktionen implementieren.

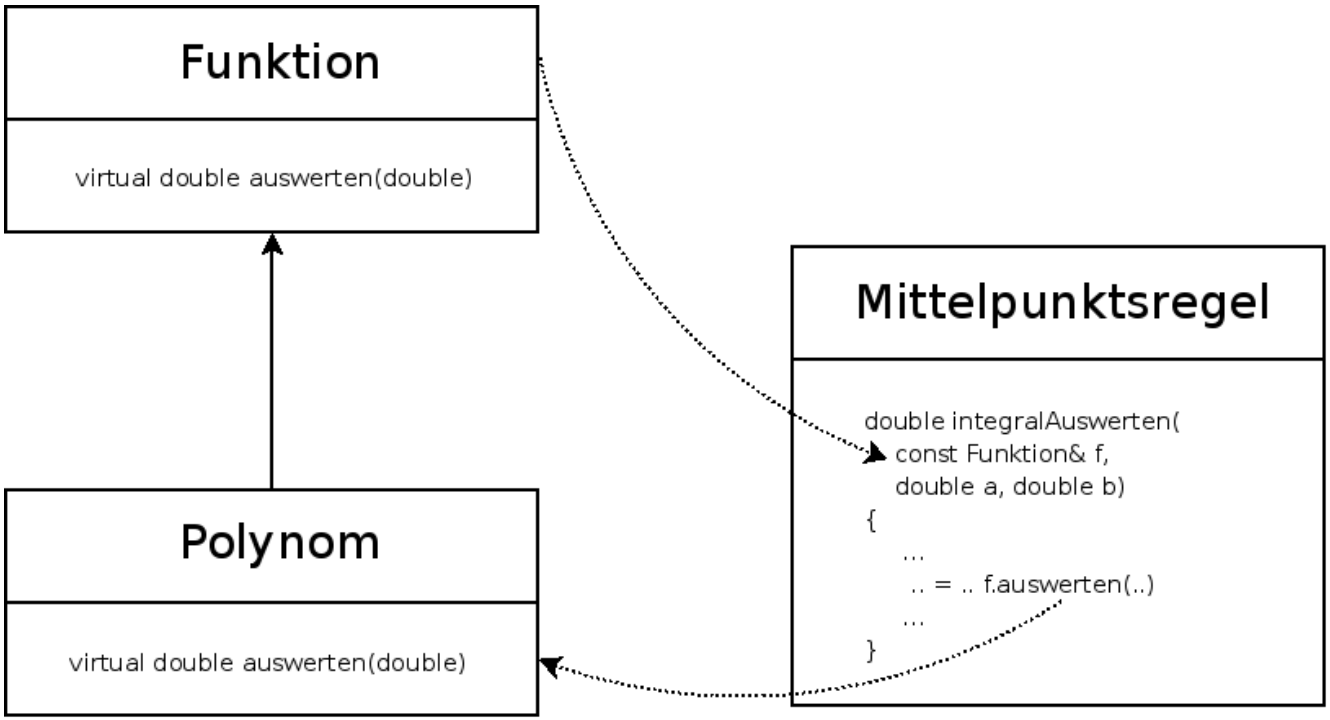
Abstrakte Basisklassen und Schnittstellen

Abstrakte Basisklassen

- Enthält eine Basis-Klasse eine rein virtuelle Funktionen, heisst die Klasse abstrakt.
- Von abstrakten Klassen können keine Objekte instanziiert werden.
- Eine abstrakte Basisklasse definiert einheitliches Erscheinungsbild (Interface) einer Abstraktion.
- Algorithmen operieren auf diesem Interface, d.h. unabhängig der tatsächlichen Implementierung.

Abstrakte Basisklassen und Schnittstellen

Beispiel:



Abstrakte Basisklassen und Schnittstellen

Erklka:rung des Beispiels:

- Der Algorithmus `Mittelpunktsregel` integriert beliebige Funktionen
- Es existiert eine (u.U. abstrakte) Basis-Klasse für Funktionen
- Allgemeine Funktionen wie Polynome, Sinus, ... werden von der Basisklasse abgeleitet.
- `Mittelpunktsregel` operiert nur auf der Funktionsschnittstelle!

Es folgt der Code zum Beispiel, es wird ein Sinus integriert:

Abstrakte Basisklassen und Schnittstellen

```
// main.cpp: Test der Integration mit der Funktions-Schnittstelle

// System-Header inkludieren
#include <cstdlib>
#include <iostream>
#include <cmath>

// eigene Header inkludieren
#include "sinus.h"
#include "mittelpunktsregel.h"

// main-Funktion
int main(int argc, char** argv)
{
    // Objekt der Klasse Mittelpunktsregel anlegen
    MittelpunktsRegel mipur(100);

    // Sinus-Objekt erzeugen
    Sinus s1;

    // Integration der Polynome testen
    std::cout << "Integral Sinus: " << mipur.integralAuswerten(s1,-2.0,2.0) << std::endl;
    std::cout << "Integral Sinus: " << mipur.integralAuswerten(s1,-3.1415,6.2890) << std::endl;
    std::cout << std::endl;

    return 0;
}
```

Abstrakte Basisklassen und Schnittstellen

```
// mittelpunktsregel.h: Die Klasse Mittelpunktsregel

#include "funktion.h"

#ifndef __MIPUREGEL_H_
#define __MIPUREGEL_H_

// Mittelpunktsregel-Klasse
class MittelpunktsRegel
{
public:
    MittelpunktsRegel(int anzahl) : n(anzahl) {}
    ~MittelpunktsRegel() {};

    // Integral einer Funktion auswerten
    double integralAuswerten(Funktion& f, double a, double b) const
    {
      double erg = 0.0;
      double h = (b-a)/(1.0*n);        // Laenge der Intervalle

      // Anteile der einzelnen Boxen aufsummieren
      for (int i=0; i<n; ++i)
      {
          double x = a + i*h + 0.5*h; // Intervall-Mittelpunkt
          erg += h * f.auswerten(x);  // Funktionsauswertung
      }

      return erg;
    }

private:
    int n;
};

#endif
```

Abstrakte Basisklassen und Schnittstellen

```
// funktion.h: Abstrakte Schnittstellenklasse fuer Funktionen

// Inklusions-Waechter
#ifndef __FUNKTION_H_
#define __FUNKTION_H_

// Abstrakte Basisklasse fuer Funktionen
class Funktion
{
public:
  // Konstruktoren
  Funktion() {};

  // virtueller Destruktor
  virtual ~Funktion() {};

  // Funktion auswerten, rein virtuell !
  virtual double auswerten(double x) const = 0;

private:
};

#endif
```

Abstrakte Basisklassen und Schnittstellen

```
#include <cmath>

// inkludiere Basisklasse / Schnittstelle
#include "funktion.h"

#ifndef __SINUS_H_
#define __SINUS_H_

// Kapselungs-Klasse fuer den Sinus
class Sinus : public Funktion
{
public :
    Sinus() {}

    // Erfuellung der Schnittstelle
    double auswerten(double x) const
    {
      return sin(x);
    }

private :
};

#endif
```

Templates

Templates – Code-Schablonen

- Templates ermöglichen die Parametrisierung von Klassen und Funktionen.
- Templates entkoppeln Funktionen oder Algorithmen vom Datentyp.
- Zulässige Parameter:
  - Standard-Typen wie `int`, `double`, ...,
  - Eigene Typen (Klassen),
  - Templates.
- Templates ermöglichen statischen Polymorphismus (siehe später).
- Templates verallgemeinern Code → „Generische Programmierung“.

Beispiel: Templatisierte Funktion

```
#include <iostream>

// example for a function template
template <class T>
T getMax(const T& a, const T& b)
{
  return (a>b) ? a : b;
}

int main ()
{
  int i=5, j=6, k;
  double l=10.4, m=10.25, n;

  k = getMax<int>(i,j); n = getMax<double>(l,m);
  std::cout << k << ", " << n << std::endl;
  // output: 6, 10.4

  return 0;
}
```

Beispiel: Templatisierte Array-Klasse

```
// a class that takes a template parameter
template <class T> class Array
{
public:
  int add(const T& next, int n);
  T& at(int n);
  T& operator[](int n) { return at(n); }

private:
  T data[10];
};

// add a new data member
template <class T> int Array<T>::add(const T& next, int n)
{
  if (n>=0 && n<10)
  {
    data[n] = next; return 0;
  }
  else return 1;
}
```

Beispiel: Templatisierte Array-Klasse

```
// get a certain data member
template <class T> T& Array<T>::at(int n)
{
  if (n>=0 && n<10) return data[n];
}

// main program
#include <iostream>
int main()
{
  Array<int> c; c.add(3,0); c.add(4,5); c.add(0,1);
  std::cout << c.at(5) << std::endl;
  // output: 4

  Array<char> d; d.add('x',9);
  std::cout << d.at(9) << std::endl;
  // output: x

  return 0;
}
```

Weiteres zu Templates

- Mehrere Template-Parameter sind möglich
- Parameter können default-Werte haben
- Templates können ausspezialisiert werden

STL – Die Standard Template Library

In C++ gibt es viele vorgefertigte Template-Container, die ohne Kenntnis der Implementierung verwendet werden können. Sie sind in einer Bibliothek, der STL, zusammengefasst.

Die STL

- ist eine Sammlung von Template Klassen und Algorithmen,
- bietet viele Containerklassen (Klasse, die eine Menge anderer Objekte verwaltet),
- hat dabei vereinheitlichte User-Interfaces für die Container,
- ist in der C++-Standardbibliothek enthalten.

Container-Arten der STL

Die STL stellt verschiedene Arten von Containern bereit:

- Sequentielle Container
  Beispiele: Vektoren, Listen
- Container adapter
  Eingeschränktes Interface zu beliebigen Containern
  Beispiele: Stacks, Queues
- Assoziative Container
  Schlüssel-Wert Container
  Beispiel: Maps, Multimaps

Vor- und Nachteile der STL

Vor- und Nachteile der STL

+ Dynamisches Speichermanagment
+ Vermeidung von array-Überläufen
+ Hohe Qualität der Container
+ Optimierbarkeit durch statischen Polymorphismus

- Unübersichtliche Fehlermeldungen
- Hohe Anforderungen an Compiler und Entwickler
- Nicht alle Compiler sind STL-fähig (obwohl die STL im C++-Standard enthalten ist)

Beispiele für die Verwendung von STL-Containern: vector

Vector-Container

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

int main()
{
  // example usage of an STL vector
  int result = 0;
  std::vector<int> x(100);

  for (int j=0; j<100; j++) x[j] = j;

  x.push_back(100);

  for (int j=0; j<x.size(); j++)
    result += x[j];

  // output: 5050
  std::cout << result << std::endl;

  return 0;
}
```

Das Iterator-Interface

Iteratoren bieten Zugriff auf die Elemente eines Containers. Sie

- Iterieren über die Elemente eines Containers,
- Liefern Zeiger auf Container-Elemente,
- Werden von jeder Container-Klasse bereitgestellt,
- Gibt es in „rw“- und einer „w“-Varianten,
- Helfen, array-Überläufe zu vermeiden.
- Die Iteratoren werden von vielen STL-Algorithmen wie Sortieren, Suchen u. ä. verwendet.

Beispiel: Iteratorieren über eine Map

Map-Container mit Iterator

```
#include <iostream>
#include <map>
#include <cstring>

int main()
{
  // example usage of an STL-map
  std::map <std::string, int> y;

  y["eins"] = 1; y["zwei"] = 2;
  y["drei"] = 3; y["vier"] = 4;

  std::map<std::string, int>::iterator it;
  for (it=y.begin(); it!=y.end(); ++it)
    std::cout << it->first << ": " << it->second << std::endl;
    // output: 1: eins
    //         2: zwei ... usw.

  return 0;
}
```

Algorithmen

Algorithmen, die die STL bereitstellt

Die STL enthält viele hilfreiche Algorithmen, die

- Elemente eines Datencontainerns manipulieren können,
- die Iteratoren zum Elementzugriff verwenden.

Beispiele:

- Sortieren
- Suchen
- Kopieren
- Umkehren der Reihenfolge im Container
- ...

Algorithmen

Beispiel: Sortier-Algorithmen für Vektoren

- Verschiedene Sortierungen für Vektoren stehen bereit
- Unterscheidung z.B. durch:
  - Benutzte Vergleichsoperation
  - Bereich der Sortierung
  - Stabilität
- Komplexität des Standard-Sortierers für Vektoren:
  - $O(n \cdot \log n)$ ideal
  - $O(n^2)$ ungünstigster Fall
- eigene Vergleichsfunktionen möglich
- Achtung: (doppelt verkettete) Listen sind auf Einfügen und Löschen von Elementen optimiert $\Rightarrow$ spezielle Sortier-Algorithmen

Algorithmen

Beispiel: Verwendung eines Sortier-Algorithmus für Vektoren

```
// a vector for integers
vector<int> x;

x.push_back(23); x.push_back(-112);
x.push_back(0); x.push_back(9999);
x.push_back(4); x.push_back(4);

// sort the integer vector
sort(v.begin(), v.end());

// output: -112 0 4 4 23 9999
for (int i = 0; i<x.size(); i++)
  cout << x[i] << "\t";
```

Statischer vs. Dynamischer Polymorphismus

Dynamischer Polymorphismus

- Der „ganz normale“ Polymorphismus.
- Anwendung: Interface-Definitionen über abstrakte Basisklassen.
- Erlaubt Austauschbarkeit zur Laufzeit.
- Verhindert eine Vielzahl von Optimierungen, z.B.
  - inlining,
  - loop unrolling.
- Zusätzlicher Overhead (function lookup table).

Statischer vs. Dynamischer Polymorphismus

Dynamischer Polymorphismus

- Der „ganz normale“ Polymorphismus.
- Anwendung: Interface-Definitionen über abstrakte Basisklassen.
- Erlaubt Austauschbarkeit zur Laufzeit.
- Verhindert eine Vielzahl von Optimierungen, z.B.
  - inlining,
  - loop unrolling.
- Zusätzlicher Overhead (function lookup table).

Statischer Polymorphismus

- Erlaubt lediglich Austauschbarkeit zur Compile-Zeit.
- Erlaubt alle Optimierungen.
- Längere Kompilierzeiten.
- Reduziert den Overhead der Interfaces.

Statischer vs. Dynamischer Polymorphismus

Techniken zur Realisierung der Polymorphismen:

statisch:

- Templates
- Überladen von Funktionen
- „Engine“-Technik

dynamisch:

- virtuelle Funktionen
- Überschreiben von Funktionen

Beispiel: Dynamischer Polymorphismus bei Matrix-Klasse

```
// base class
class Matrix {
  virtual bool isSymmetricPositiveDefinit();
};

// symmetric matrices
class SymmetricMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit() { ... };
};

// upper triangular matrices
class UpperTriangularMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit()
  { return false };
};
```

Die Abfrage „Ist die Matrix symmetrisch positiv definit wird von der Basisklasse an die abgeleiteten Klassen durchgereicht.

Beispiel: Dynamischer Polymorphismus bei Matrix-Klasse

```
// base class
class Matrix {
  virtual bool isSymmetricPositiveDefinit();
};

// symmetric matrices
class SymmetricMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit() { ... };
};

// upper triangular matrices
class UpperTriangularMatrix : public Matrix {
  virtual bool isSymmetricPositiveDefinit()
  { return false };
};
```

$\Rightarrow$ Der Ansatz mit virtuellen Funktionen ist hier unter Umständen nicht performant. Ausweg: Engine-Konzept.

Das Engine-Konzept

```
// example delegation of a method to an engine
template<class Engine> class Matrix {
  Engine engineImp;

  bool IsSymmetricPositiveDefinit()
  { return engineImp.isSymmetricPositiveDefinite(); }
};

// some engine classes
class Symmetric {
  bool isSymmetricPositiveDefinite(){ // check if matrix is spd. }
};

class UpperTriangle {
  bool isSymmetricPositiveDefinite(){ return false; }
};

// use in main function
Matrix<UpperTriangle> A;
std::cout << A.isSymmetricPositiveDefinite() << std::endl;
```

Das Engine-Konzept

Der Engine-Ansatz

- Aspekte der verschiedenen Matrizen sind in den Engines (`Symmetric` oder `UpperTriangular`) „verpackt“.
- `Matrix` delegiert die meisten Operationen an die Engine – zur Compile-Zeit!
- Dynamischer Polymorphismus durch statischen (Templates) ersetzt.
- Nachteil: Der Basis-Typ (`Matrix`) muss alle Methoden *aller* Subklassen enthalten.
- Der Trick, dies zu vermeiden, nennt sich „Barton-Nackmann-Trick“.

Template Spezialisierungen

Eine wichtige Technik bei der Arbeit mit Templates ist die sogenannte „Template-Spezialisierung“:

- Abweichungen von der Template-Schablone können explizit ausprogrammiert werden,
- Etwa für Datentypen, die Laufzeit- oder Speicher-effizient implementiert werden können.

Template Spezialisierungen

Beispiel zur Spezialisierung von Templates: Sortierung

```
// a sorter class with two template parameters
template <class T, int N> Sorter
{
  void sort(T* array) { // sort here };
  ...
};

// sorting a single field array is simple...
template <class T> Sorter<T,1>
{
  void sort(T* array) {};
  ...
};
```

Template Spezialisierungen

Wofür brauchen wir Template-Spezialisierung?

Viele Algorithmen (auch untemplatisierte) können durch Spezialisierung beschleunigt werden. Beispiel:

```
// dot-product
double dotproduct(const double *a, const double *b, int N)
{
  double result = 0.0;
  for (int i=0; i<N; i++)
    result += a[i]*b[i];
  return result;
}

// specialisation for small N (e.g. N=3) speeds up calculation
double dotproduct(const double *a, const double *b, int N)
{
  return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

Motivation

Templates parametrisieren Klassen und Funktionen im Typ. Oft kann man Code durch fortgeschrittenere Techniken verbessern.

Zwei fortgeschrittene Template-Verwendungen

- Traits – Meta-Informationen von Template-Parametern
- Policies – Verhaltens-Modifikation von Algorithmen

Definition

Definition: Traits

Repräsentieren natürliche zusätzliche Eigenschaften eines Template Parameters.

Beispiele: Meta-Informationen für Gitter (Ist Gitter konform, adaptiv, . . . ?), Typ-Promotionen.

Type Promotion Traits

Betrachte Addition von 2 Vektoren:

```
template<typename T>
std::vector<T> operator+(const std::vector<T>& a,
    const std::vector<T>& b);
```

Frage: Rückgabetyp bei Addition zweier Vektoren unterschiedlichen Typs:

```
template<typename T1, typename T2>
std::vector<???> operator+(const std::vector<T1>& a,
    const std::vector<T2>& b);
```

Beispiel:

```
std::vector<int> a;
std::vector <complex<double> > b;

std::vector<???> c = a+b;
```

Type Promotion Traits

Der Rückgabetyp ist abhängig von den beiden Input-Typen! Das Problem kann mit Traits-Klassen gelöst werden:

```
template<typename T1, typename T2>
std::vector<typename Promotion<T1, T2>::promoted_type>
operator+(const std::vector<T1> &, const std::vector<T2> &);
```

Die Type Promotion Traits werden über Spezialisierung definiert:

```
template<typename T1, typename T2>
struct Promotion {};
```

Für identische Typen kann eine partielle Spezialisierung vorgenommen werden:

```
struct Promotion<typename T, typename T> {
  public:
    typedef T promoted_type;
};
```

Type Promotion Traits

Andere Promotionen können über volle Spezialisierung erreicht werden:

```
template<>
struct Promotion<float, complex<float> > {
  public:
    typedef complex<float> promoted_type;
};

template<>
struct Promotion<complex<float>, float> {
  public:
    typedef complex<float> promoted_type;
};
```

Type Promotion Traits

Sind viele Typ-Promotionen notwendig, erleichtern kleine Makros die Arbeit:

```
#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
```

Type Promotion Traits

Eine generische Addier-Funktion mit Typ-Promotion kann dann etwa so implementiert werden:

```
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 Itera;
  typedef typename std::vector<T2>::const_iterator Iterb;

  std::vector<T3> c;
  Iterc ic = c.begin();
  Iterb ib = b.begin();
  for(Itera ia=a.begin(); ia!=a.end(); ++ia, ++ib, ++ic)
    *ic = *i1 + *i2;
  return c;
}
```

Type Promotion Traits

Weiteres Beispiel:

```
#include <iostream>

using namespace std;

// start with the basic template:
template <typename T1, typename T2>
struct Promote
{
};

// the same types are the same
template <typename T1>
struct Promote<T1,T1>
{
    typedef T1 type;
};

// specilizations for all the type promotions
template<> struct Promote<int,char> { typedef int type; };
template<> struct Promote<double,int> { typedef double type; };
```

Type Promotion Traits

Weiteres Beispiel:

```
// an example function build minima of two variables with different ty
template <typename T1, typename T2>
typename Promote<T1,T2>::type min( const T1 & x, const T2 & y )
{
    return x < y ? x : y;
}

// main
int main()
{
    std::cout << "min:␣" << min(88.9, 99) << std::endl;
    // output: 88.9

    std::cout << "min:␣" << min(4756, 'a') << std::endl;
    // output: 97

    return 0;
}
```

Weiterführende Literatur

Literatur zu „Scientific Computing with C++“

- N. Josuttis: C++ Templates – The Complete Guide
- T. Veldhuizen: Techniques for Scientific C++
- T. Veldhuizen: Template Metaprogramming
- E. Unruh: Prime Number Computation (historisches Beispiel für TMP)