

C++ für Wissenschaftliches Rechnen

Stefan Lang

Interdisziplinäres Institut für Wissenschaftliches Rechnen, Universität Heidelberg

16. Oktober 2013

C++ für Wissenschaftliches Rechnen I

① Warum C++?

Motivation

Konzepte von C++

② Das erste Programm

③ Grundlagen C++

Datentypen

Kontrollfluss

Funktionen

④ Zeiger und Referenzen

⑤ Abstrakte Datentypen und ihre Realisierung in C++

Klassen

Konstruktoren und Destruktoren

⑥ Templates und generische Programmierung

⑦ Die Standard Template Library (STL)

Beispiel einer Container-Klasse: Vektoren

Das Iterator-Interface

⑧ Built-in Algorithmen der STL

⑨ Vererbung in C++

C++ für Wissenschaftliches Rechnen II

⑩ Virtuelle Funktionen und abstrakte Basisklassen

Virtuelle Funktionen

Rein virtuelle Funktionen und abstrakte Basisklassen

⑪ Statischer vs. Dynamischer Polymorphismus

Dynamischer Polymorphismus

Statischer Polymorphismus und Engines

⑫ Template Meta Programming

⑬ Template Spezialisierungen

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

Konzepte von C++

C++ ist eine objektorientierte Sprache

D.h. C++ unterstützt

- 1 Abstraktion durch Klassen und Objekte,
- 2 Vererbung und
- 3 Polymorphismus zur Laufzeit.

Polymorphismus bedeutet „Viele Gestalten“:

- Eine Variable kann ihren Typ zur Laufzeit ändern,
- Eine Funktion mit polymorphen Argumenten,
- Ein Funktionsname, der von verschiedenen Funktionen verwendet wird.

Literatur

Literatur zu C++

- B. Stroustrup: C++ – Die Programmiersprache (Die Bibel)
- B. Eckel: Thinking in C++, Volume 1 + 2
- A. Willms: C++ Programmierung (Für Anfänger gut geeignet)

Grundlegende C++-Kenntnisse

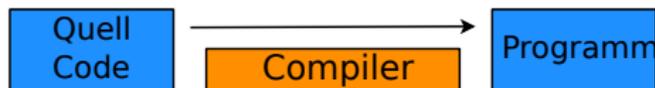
Um die Vorzüge von C++ auszunutzen, sind abstrakte Techniken notwendig. Folgende grundlegenden Konzepte sind als Basis unumgänglich:

- 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

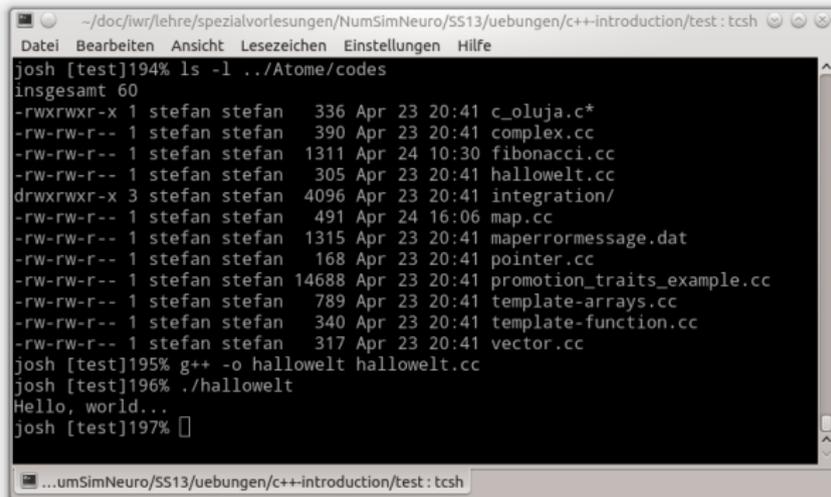
Hallo, Welt!

```
1 // include I/O-library
2 #include <iostream>
3
4 // main is always the first function to be called
5 // argc: program argument counter
6 // argv: pointer to C-Strings containing the arguments
7 int main(int argc, char** argv)
8 {
9     std::cout << "Hello, world..." << std::endl;
10
11     // return value of main
12     return 0;
13 }
```

Das Erstellen des Executables erfordert hier nur einen Compiler (g++):



Übersetzen unter Linux

A terminal window titled "...umSimNeuro/SS13/uebungen/c++-introduction/test : tcsh" showing the execution of 'ls -l' and 'g++' commands. The 'ls' command lists several C++ source files with their permissions, owner, size, and modification date. The 'g++' command compiles 'hallowelt.cc' into an executable named 'hallowelt'. The terminal output shows the file listing and the successful compilation and execution of the program, which prints "Hello, world...".

```
~/doc/iwr/lehre/spezialvorlesungen/NumSimNeuro/SS13/uebungen/c++-introduction/test : tcsh
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
josh [test]194% ls -l ../Atome/codes
insgesamt 60
-rwxrwxr-x 1 stefan stefan 336 Apr 23 20:41 c_oluja.c*
-rw-rw-r-- 1 stefan stefan 390 Apr 23 20:41 complex.cc
-rw-rw-r-- 1 stefan stefan 1311 Apr 24 10:30 fibonacci.cc
-rw-rw-r-- 1 stefan stefan 305 Apr 23 20:41 hallowelt.cc
drwxrwxr-x 3 stefan stefan 4096 Apr 23 20:41 integration/
-rw-rw-r-- 1 stefan stefan 491 Apr 24 16:06 map.cc
-rw-rw-r-- 1 stefan stefan 1315 Apr 23 20:41 maperrormessage.dat
-rw-rw-r-- 1 stefan stefan 168 Apr 23 20:41 pointer.cc
-rw-rw-r-- 1 stefan stefan 14688 Apr 23 20:41 promotion_traits_example.cc
-rw-rw-r-- 1 stefan stefan 789 Apr 23 20:41 template_arrays.cc
-rw-rw-r-- 1 stefan stefan 340 Apr 23 20:41 template_function.cc
-rw-rw-r-- 1 stefan stefan 317 Apr 23 20:41 vector.cc
josh [test]195% g++ -o hallowelt hallowelt.cc
josh [test]196% ./hallowelt
Hello, world...
josh [test]197%
```

Für größere Projekte ist der C++-Build-Prozess in der Regel recht kompliziert.

Datentypen in C++

Die elementaren Datentypen in C++ sind:

int	Ganzzahlen	<code>int a = 2;</code>
long	Große Ganzzahlen	<code>long a = 1e15;</code>
char	Zeichen	<code>char a = 'b';</code>
float	Gleitkommazahlen 4 Byte	<code>float b = 3.14;</code>
double	Gleitkommazahlen 8 Byte	<code>double c = 3.1415;</code>
bool	Wahrheitswerte	<code>bool d = false;</code>

Verzweigungen

if-Verzweigungen:

```
1  #include <iostream>
2
3  int main(int argc, char** argv)
4  {
5      int a = 5;    // an integer variable
6      if (a > 0)
7      {
8          std::cout << "Hello, World..." << std::endl;
9      }
10     else
11     {
12         return 1; // emit an error
13     }
14
15     return 0;
16 }
```

Realisierung von Schleifen

- for-Schleifen,
- while-Schleifen,
- do..while-Schleifen.

```
1  #include <iostream>
2
3  int main(int argc, char** argv)
4  {
5      for (int i=1; i<10; ++i)
6          std::cout << "i: " << i << std::endl;
7
8      int j = 5;
9      while (j > 0)
10     {
11         std::cout << "j: " << j << std::endl;
12         j--;
13     }
14
15     return 0;
16 }
```

Funktionen

Funktionen

Funktionen dienen zur Kapselung von Programmabschnitten und können bei Bedarf aufgerufen werden.

In C++ haben sie immer die Syntax

```
Rueckgabetyf Funktionsname(Parameter1, Parameter2, ..);
```

Ein Beispielprogramm mit Funktion

```
1  #include <iostream>
2
3  using namespace std; // use namespace std globally (here ok,
4                       // avoid this in the general case)
5
6  // A function that greets everyone
7  void greet()
8  {
9      // do not need namespace-selector std:: any more
10     cout << "Hello, World." << endl;
11 }
12
13 // main function
14 int main(int argc, char** argv)
15 {
16     greet();
17     return 0;
18 }
```

Call-by-Reference und Call-by-Value

Bei Call-by-Value wird die Adresse des Objekts als Funktionsparameter übergeben und keine Kopie des Objekts erzeugt:

```
1 // call-by-value
2 void swap_wrong (int a, int b)
3 {
4     int tmp = a;
5     a = b;           // does not work, a and b are local copies
6     b = tmp;       // in the scope of the function
7 }
8
9 // call-by-reference
10 void swap_right (int& a, int& b)
11 {
12     int tmp = a; // a, b are reference parameters
13     a = b;       // That means changes to them are
14     b = tmp;     // persistant after end of function call
15 }
```

Call-by-Reference und Call-by-Value

```
1 // main function
2 int main(int argc, char** argv)
3 {
4     int a=5, b=6;
5
6     // Ausgabe 5, 6
7     swap_wrong(a, b)
8     std::cout << a << ", " << b << std::endl;
9
10    // Ausgabe 6, 5
11    swap_right(a, b)
12    std::cout << a << ", " << b << std::endl;
13
14    return 0;
15 }
```

Sollen Änderungen einer Funktion Bestand haben, müssen immer Referenz-Variablen verwendet werden (wie bei `swap_right`).

Zeiger und Referenzen

Eines der komplizierteren Themen in C/C++ sind Zeiger und Referenzen.

Zeiger und der Adressoperator &

- `int x = 12`

Die Variable `x` ist definiert durch Adresse, Größe (benötigter Speicherplatz), Name und Inhalt.

- Um den Wert der Adresse (nicht der Variablen `x`!) zu ermitteln, gibt es den **Adressoperator** `&`:

```
std::cout << &x << std::endl // Ausgabe: 0xA0000000
```

- Adresswerte können in **Zeigervariablen** gespeichert werden. Zeigervariablen haben die Syntax `Typ* name`, `Typ` ist der Typ des Objekts, auf den der Zeiger `name` zeigt:

```
int* z = &x; // z ist eine Zeigervariable
```

Zeiger und Referenzen

Der Dereferenzierungsoperator *

- Über die Zeigervariable z

```
int* z = &x; // z ist eine Zeigervariable
```

kann auch der Wert der Variablen x verändert werden. Dazu gibt es den (**Dereferenzierungsoperator** $*$):

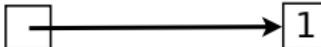
```
*z = 4711; // z wird dereferenziert, x hat nun den Wert 4711
```

- Achtung:
 - Mit dem Dereferenzierungsoperator wird nicht der Zeiger z verändert (z zeigt immer noch auf die Speicheradresse von x).
 - Das Symbol $*$ bezeichnet je nach Kontext den Dereferenzierungsoperator oder eine Zeigervariable.

Zeiger und Referenzen

Den Zusammenhang zwischen Zeigervariablen, Adress- und Dereferenzierungsoperator verdeutlicht folgende Abbildung:

Zeigervariable `intPtr`

`int* intPtr = &i` 

Adressoperator `&`

`*intPtr = 6` 

Dereferenzierungsoperator `*`

Zeiger und Referenzen

Referenzen

Neben Zeigervariablen gibt es *Referenzen*.

- Referenzen sind intern Zeiger.
- Referenzen kann man sich als „anderen Namen“ für eine Variable vorstellen:

```
1  int  x = 5;  
2  int& y = x; // anderer Name fuer x  
3  y = 4;     // bedeutet x = 4!
```

Zeiger und Referenzen

Beispiele für Zeiger und Referenzen:

```
1  int i, j, *p, *q;
2  int &s = i, &r = j; // Referenzen muessen initialisiert werden
3
4  r = 2;           // OK, j (==r) hat jetzt Wert 2
5  r = &j;         // BAD, &j hat falschen Typ 'int *' statt 'int'
6
7  p = 2;          // BAD, 2 hat falschen Typ 'int' statt 'int *'
8  p = &j;         // OK, p enthaelt nun Adresse von j
9
10 if (p == q)    // WAHR, falls p, q auf die gleiche Adresse zeigen
11               // Der Inhalt der Adresse ist egal.
12
13 if (r == s)    // WAHR, falls Inhalt von j (Referenz von r) und i
14               // (Referenz von s) gleich ist. Die Adresse der
15               // Variablen ist egal!
```

Zeiger und Referenzen

(Mehrdimensionale) Felder sind nichts anderes als Zeiger auf den ersten Feldeintrag:

```
1  int a[5];           // Feld von 5 int-Variablen
2
3  a[0] = 3;
4  std::cout << *a;   // output: 3 (= a[0])
5  std::cout << &a;   // output: Adresse von a[0]
6
7  int a[3][20];      // 3 x 20 - Feld
```

Zeiger und Referenzen

Zeiger erlauben beliebig komplizierte Konstrukte:

```
1  int **p;           // p enthaelt Zeiger, die auf Variablen des
2                      // Typs 'int' zeigen
3
4  int *p[10];        // p ist ein Feld, das 10 int-Variablen enthaelt,
5                      // denn die Klammern [] binden staerker als *.
6                      // D.h. int * ist der Typ der Feldelemente!
7
8  int (*p)[10];      // Jetzt hingegen ist p ein Zeiger auf ein
9                      // Feld mit 10 int-Komponenten
10
11 int* f()            // f ist eine parameterlose Funktion, die
12                      // einen Zeiger auf int zurueckgibt.
13                      // Runde Klammern binden staerker, wie oben!
```

Klassen und Datentypen

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

```
1 #include <iostream>
2
3 class ComplexNumber { // a class definition
4 public:
5     void print()
6     {
7         std::cout << u << " + i * " << v << std::endl;
8     }
9
10 private:
11     double u, v;
12 }; // ';' is very important!
13
14 int main(int argc, char** argv)
15 {
16     ComplexNumber a, b, c;
17     a.print(); // print uninitialized (!) number
18
19     //c = a + b; // where defined?
20
21     return 0;
22 }
```

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:

```
1  class ComplexNumbers
2  {
3  public:
4      // some constructors
5      ComplexNumber() { u = 0; v = 0; }    // default
6
7      ComplexNumber(double re, double im) // initialize with
8      { u = re; v = im; }                // given numbers
9
10     void print() { ... }
11
12 private:
13     double u, v;
14 };
```

Konstruktoren

```
1 // usage of the complex number class
2 int main (int argc, char** argv)
3 {
4     ComplexNumber a(3.0,4.0);
5     ComplexNumber b(1.0,2.0);
6     ComplexNumber c;
7
8     a.print();           // output: 3 + i * 4
9     c = a + b;         // where defined ?
10
11     return 0;
12 };
```

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`.

Überladen von Operatoren

Operationen für abstrakte Datentypen (Klassen)

- Die Anweisung $a + b$ ist für `ComplexNumber` nicht definiert und muss erklärt werden.
- Für Klassen können verschiedene Operationen wie $++$, $+$, $*$, $/$, $-$, $--$, $=$, $!=$, $!$, $==$, $[]$, \dots ausimplementiert werden.
- Klassen, die den Operator $()$ implementieren, heißen *Funktoren*.

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

```
1 #include <iostream>
2
3 // example for a function template
4 template <class T>
5 T getMax(const T& a, const T& b)
6 {
7     return (a>b) ? a : b;
8 }
9
10 int main ()
11 {
12     int    i = 5, j = 6, k;
13     double l = 10.4, m = 10.25, n;
14
15     k = getMax<int>(i,j); n = getMax<double>(l,m);
16     std::cout << k << ", " << n << std::endl;
17     // output: 6, 10.4
18
19     return 0;
20 }
```

Beispiel: Templatisierte Array-Klasse

```

1 // a class that takes a template parameter
2 template <typename T> class Array
3 {
4 public:
5     int add(const T& next, int n);           // add 'next' at data[n]
6     T& at(int n);
7     T& operator [] (int n) { return at(n); } // overloaded operator
8
9 private:
10    T data[10];
11 };
12
13 // add a new data member
14 template <class T> int Array<T>::add(const T& next, int n)
15 {
16     if (n>=0 && n<10)
17     {
18         data[n] = next; return 0;
19     }
20     else return 1;
21 }

```

Beispiel: Templatisierte Array-Klasse

```
23 // get a certain data member
24 template <class T> T& Array<T>::at(int n)
25 {
26     if (n>=0 && n<10) return data[n];
27 }
28
29 // main program
30 #include <iostream>
31 int main()
32 {
33     Array<int> c; c.add(3,0); c.add(4,5); c.add(0,1);
34     std::cout << c.at(5) << std::endl;
35     // output: 4
36
37     Array<char> d; d.add('x',9);
38     std::cout << d.at(9) << std::endl;
39     // output: x
40
41     return 0;
42 }
```

Weiteres zu Templates

- Templates sind die Grundlage der generischen Programmierung in C++!
- Templates können ausspezialisiert werden (für Sonderfälle).
- Mehrere Template-Parameter sind möglich.
- Parameter können default-Werte haben.

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 Speichermanagement
- + 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

```
1 #include <iostream>
2 #include <vector>
3
4 int main() {
5     // example usage of an STL vector
6     int result = 0;
7     std::vector<int> x(100);
8
9     for (int j=0; j<100; j++) x[j] = j;
10
11     x.push_back(100);
12
13     for (int j=0; j<x.size(); j++)
14         result += x[j];
15
16     // output: 5050
17     std::cout << result << std::endl;
18
19     return 0;
20 }
```

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 „r“- und „w“-Varianten,
- Helfen, array-Überläufe zu vermeiden.
- Die Iteratoren werden von vielen STL-Algorithmen wie Sortieren, Suchen u. ä. verwendet.

Beispiel: Iteratoren über eine Map

```
1 #include <iostream>
2 #include <map>
3 #include <cstring>
4
5 int main()
6 {
7     // example usage of an STL-map
8     std::map <std::string, int> y;
9
10    y["eins"] = 1; y["zwei"] = 2;
11    y["drei"] = 3; y["vier"] = 4;
12
13    std::map<std::string, int>::iterator it;
14    //std::map::string, double::iterator it; // nice error message :-)
15    for (it=y.begin(); it!=y.end(); ++it)
16        std::cout << it->first << ": " << it->second << std::
17            endl;
18    // output: eins: 1
19    // zwei: 2 ... usw.
20    return 0;
21 }
```

Ein Nachteil der STL: Die Fehlermeldung

Falls in diesem Beispiel der falsche Typ eines Iterators instantiiert wird, spuckt der Compiler folgende Fehlermeldung aus:

```
1 map.cc: In function 'int main()':
2 map.cc:15: error: no match for 'operator=' in 'it = y.std::map<_Key,
  _Tp, _Compare, _Alloc>::begin [with _Key = std::basic_string<char,
  std::char_traits<char>, std::allocator<char> >, _Tp = int,
  _Compare = std::less<std::basic_string<char, std::char_traits<
  char>, std::allocator<char> > >, _Alloc = std::allocator<std::
  pair<const std::basic_string<char, std::char_traits<char>, std::
  allocator<char> >, int> >]()'
3 /usr/include/c++/4.4/bits/stl_tree.h:154: note: candidates are: std::
  _Rb_tree_iterator<std::pair<const std::basic_string<char, std::
  char_traits<char>, std::allocator<char> >, double> >& std::
  _Rb_tree_iterator<std::pair<const std::basic_string<char, std::
  char_traits<char>, std::allocator<char> >, double> >::operator=(
  const std::_Rb_tree_iterator<std::pair<const std::basic_string<
  char, std::char_traits<char>, std::allocator<char> >, double>
  >&)
4 map.cc:15: error: no match for 'operator!=' in 'it != y.std::map<_Key,
  _Tp, _Compare, _Alloc>::end [with _Key = std::basic_string<char,
  std::char_traits<char>, std::allocator<char> >, _Tp = int,
  _Compare = std::less<std::basic_string<char, std::char_traits<
  char>, std::allocator<char> > >, _Alloc = std::allocator<std::
  pair<const std::basic_string<char, std::char_traits<char>, std::
  allocator<char> >, int> >]()'
5 [...]
```

Algorithmen

Algorithmen, die die STL bereitstellt

Die STL enthält viele hilfreiche Algorithmen, die

- Elemente eines Datencontainers 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

```
1 // a vector for integers
2 vector<int> x;
3
4 x.push_back(23); x.push_back(-112);
5 x.push_back(0); x.push_back(9999);
6 x.push_back(4); x.push_back(4);
7
8 // sort the integer vector
9 sort(v.begin(), v.end());
10
11 // output: -112 0 4 4 23 9999
12 for (int i = 0; i<x.size(); i++)
13     cout << x[i] << "\\t";
```

Vererbung in C++

Vererbung

- Datentyp gibt seine Abstraktion an anderen Datentyp weiter.
- „Ist-ein“ Relation: Dreieck ist ein geometrisches Objekt, d.h. Klasse Dreieck ist von Klasse GeomObject abzuleiten.
- Nicht zu verwechseln mit einer „Enthält-ein“ Relation: Ein Dreieck enthält drei Punkte (aber ein Dreieck ist kein Punkt → keine Vererbung).

Vererbung in C++

```
1  // example of inheritance in C++
2  class Matrix
3  {
4  public:
5      ...
6  private:
7      double data[3][3]; // (3 x 3)-Matrix
8  };
9
10 // the derived class: symmetrical matrix is a matrix
11 class SymMatrix: public Matrix
12 {
13 public:
14     double getEntry(int i, int j) { return data[i][j]; }
15         // error: data private in base class
16     ...
17     // constructor calls a constructor of base class
18     SymMatrix() : Matrix() { ... }
19 };
```

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

Virtuelle Funktionen erlauben, dass abgeleitete Klassen Methoden der Basisfunktionen überdecken:

```
1  class GeomObject           // base class for geo objects
2  {                           // 'area' is a function member
3  public:
4
5     virtual double area() { return 0.0; }
6     ...
7 };
8
9  class Triangle : public GeomObject
10 {                             // a derived class
11 public:                          // has a specific member 'area' as well
12
13     double area() { return 0.5 * a * h; }
14     ...
15 private:
16
17     double h, a;
18 };
```

Virtuelle Funktionen

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

```
19 int main() {
20     GeomObject* geo;
21     Triangle t;
22
23     geo = &t;
24     std::cout << geo->area() << std::endl; // ??
25
26     return 0;
27 };
```

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 \longleftrightarrow 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 der Basisklasse operieren, können auch auf den abgeleiteten Klassen operieren.
- Beispiel: Liste, die Pointer auf `GeomObjects` 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

- Deklaration der Funktion in der Basisklasse als „rein virtuell“:
- Abgeleitete Klassen müssen rein virtuelle Funktionen implementieren.

Klassen mit einer (oder mehreren) rein virtuellen Funktionen heißen **abstrakte Basisklassen**. Sie sind reine Schnittstellen-Spezifikationen.

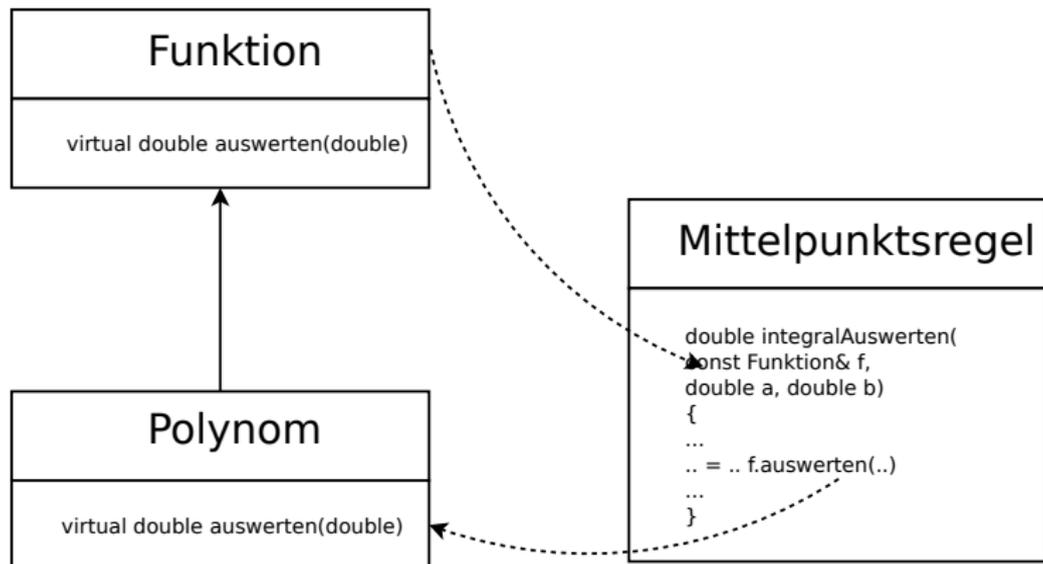
Abstrakte Basisklassen und Schnittstellen

Abstrakte Basisklassen

- Enthält eine Basis-Klasse mind. eine rein virtuelle Funktion, heisst die Klasse abstrakt.
- Von abstrakten Klassen können keine Objekte instanziiert werden.
- Eine abstrakte Basisklasse definiert ein einheitliches Interface.
- Algorithmen operieren auf diesem Interface, d.h. unabhängig von der tatsächlichen Implementierung.

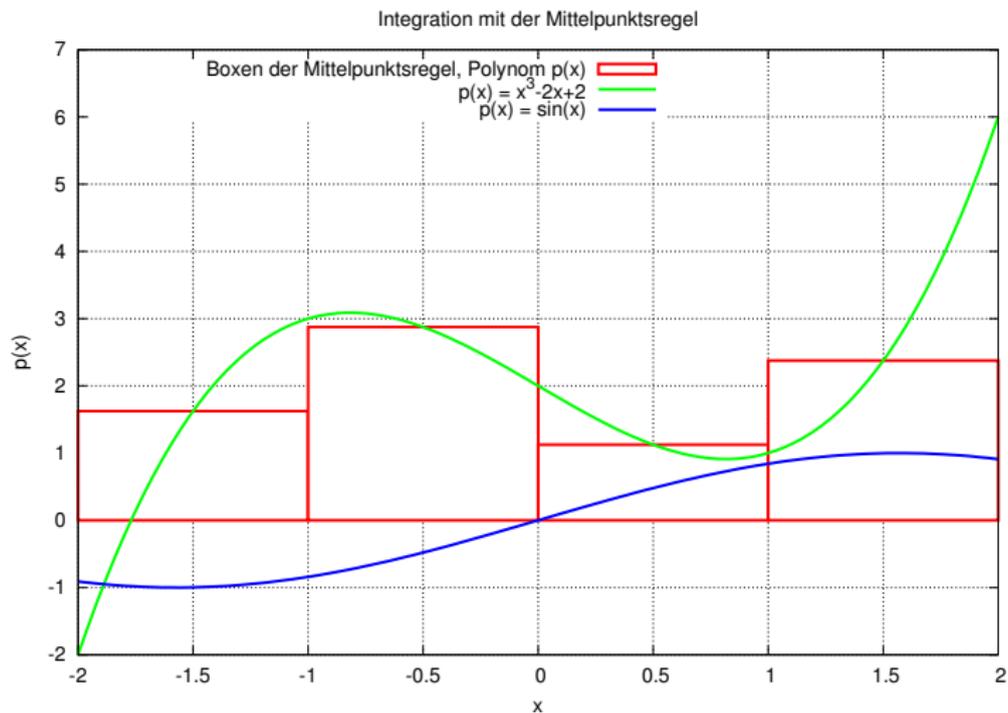
Abstrakte Basisklassen und Schnittstellen

Beispiel:



Abstrakte Basisklassen und Schnittstellen

Beispiel:



Abstrakte Basisklassen und Schnittstellen

Erklärung 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

```
1 // main.cpp: Test der Integration mit der Funktions-Schnittstelle
2
3 // System-Header inkludieren
4 #include <cstdlib>
5 #include <iostream>
6 #include <cmath>
7
8 // eigene Header inkludieren
9 #include "sinus.h"
10 #include "mittelpunktsregel.h"
11
12 // main-Funktion
13 int main(int argc, char** argv)
14 {
15     // Objekt der Klasse Mittelpunktsregel anlegen
16     MittelpunktsRegel mipur(100);
17
18     // Sinus-Objekt erzeugen
19     Sinus s1;
20
21     // Integration der Polynome testen
22     std::cout << "Integral Sinus: " << mipur.integralAuswerten(s1, -2.0, 2.0) << std::endl;
23     std::cout << "Integral Sinus: " << mipur.integralAuswerten(s1, -3.1415, 6.2890) << std::endl;
24     std::cout << std::endl;
25
26     return 0;
27 }
```

Abstrakte Basisklassen und Schnittstellen

```
1 // mittelpunktregel.h: Die Klasse Mittelpunktsregel
2
3 #include "funktion.h"
4
5 #ifndef __MIPUREGEL_H_
6 #define __MIPUREGEL_H_
7
8 // Mittelpunktsregel-Klasse
9 class MittelpunktsRegel
10 {
11 public:
12     MittelpunktsRegel(int anzahl) : n(anzahl) {}
13     ~MittelpunktsRegel() {};
14
15     // Integral einer Funktion auswerten
16     double integralAuswerten(Funktion& f, double a, double b) const
17     {
18         double erg = 0.0;
19         double h = (b-a)/(1.0*n); // Laenge der Intervalle
20
21         // Anteile der einzelnen Boxen aufsummieren
22         for (int i=0; i<n; ++i)
23         {
24             double x = a + i*h + 0.5*h; // Intervall-Mittelpunkt
25             erg += h * f.auswerten(x); // Funktionsauswertung
26         }
27
28         return erg;
29     }
30
31 private:
32     int n;
33 };
34
35 #endif
```

Abstrakte Basisklassen und Schnittstellen

```
1 // funktion.h: Abstrakte Schnittstellenklasse fuer Funktionen
2
3 // Inklusions-Waechter
4 #ifndef __FUNKTION_H_
5 #define __FUNKTION_H_
6
7 // Abstrakte Basisklasse fuer Funktionen
8 class Funktion
9 {
10 public:
11     // Konstruktoren
12     Funktion() {};
13
14     // virtueller Destruktor
15     virtual ~Funktion() {};
16
17     // Funktion auswerten, rein virtuell !
18     virtual double auswerten(double x) const = 0;
19
20 private:
21 };
22
23 #endif
```

Abstrakte Basisklassen und Schnittstellen

```
1 #include <cmath>
2
3 // inkludiere Basisklasse / Schnittstelle
4 #include "funktion.h"
5
6 #ifndef __SINUS_H_
7 #define __SINUS_H_
8
9 // Kapselungs-Klasse fuer den Sinus
10 class Sinus : public Funktion
11 {
12 public :
13     Sinus() {}
14
15     // Erfuellung der Schnittstelle
16     double auswerten(double x) const
17     {
18         return sin(x);
19     }
20
21 private :
22 };
23
24 #endif
```

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 tables).

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 tables).

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

→ Statischer Polymorphismus erlaubt es, Algorithmen und Datenstrukturen zu trennen (Interfaces), wird aber zur Compilezeit ausgewertet und erlaubt exzessives Optimieren.

Beispiel: Dynamischer Polymorphismus bei Matrix-Klasse

```
1 // base class
2 class Matrix {
3     virtual bool isSymmetricPositiveDefinit();
4 };
5
6 // symmetric matrices
7 class SymmetricMatrix : public Matrix {
8     virtual bool isSymmetricPositiveDefinit() { ... };
9 };
10
11 // upper triangular matrices
12 class UpperTriangularMatrix : public Matrix {
13     virtual bool isSymmetricPositiveDefinit()
14     { return false };
15 };
```

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

Beispiel: Dynamischer Polymorphismus bei Matrix-Klasse

```
1 // base class
2 class Matrix {
3     virtual bool isSymmetricPositiveDefinit();
4 };
5
6 // symmetric matrices
7 class SymmetricMatrix : public Matrix {
8     virtual bool isSymmetricPositiveDefinit() { ... };
9 };
10
11 // upper triangular matrices
12 class UpperTriangularMatrix : public Matrix {
13     virtual bool isSymmetricPositiveDefinit()
14     { return false };
15 };
```

⇒ Der Ansatz mit virtuellen Funktionen ist hier unter Umständen nicht performant. Ausweg: Statischer Polymorphismus (hier: Engine-Konzept).

Das Engine-Konzept

```
1  // example delegation of a method to an engine
2  template<class Engine> class Matrix {
3      Engine engineImp;
4
5      bool IsSymmetricPositiveDefinit()
6      { return engineImp.isSymPositiveDefinite(); }
7  };
8
9  // some engine classes
10 class Symmetric {
11     bool isSymPositiveDefinite()
12     { /* check if matrix is spd. */}
13 };
14
15 class UpperTriangle {
16     bool isSymPositiveDefinite(){ return false; }
17 };
```

Das Engine-Konzept

```
1 // usage (compiler evaluates Type of A !)  
2 UpperTriangle upper; // create upper matrix  
3  
4 Matrix<UpperTriangle> A(upper); // pass upper to some  
5 // constructor of A  
6  
7 std::cout << A.isSymPositiveDefinite() << 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 Meta Programming

Entscheidende Technik des statischen Polymorphismus sind Templates. Mit den Templates ist eine Programmieretechnik für Meta-Programme entstanden:

Template Meta Programme

- Idee: Der Compiler agiert als Interpreter.
- Ersetzen von Kontrollstrukturen wie `if` und Loops durch Spezialisierung und Rekursion.
- Theoretisch: Turing-Maschine durch Template Programming möglich.

Beispiel eines Template Meta Programms: Fakultät (T. Veldhuizen)

```
// factorial realized as TMP
template<int N> class Factorial
{
public:
    enum { value = N * Factorial<N-1>::value };
};

// a specialization is needed to break
class Factorial<1>
{
public:
    enum { value = 1 };
};
```

⇒ der Wert $N!$ ist zur Kompilierzeit als `Factorial<N>::value` verfügbar durch erzeugen eines Objekts der Klasse:

```
Factorial<12> a; // ergibt 12!
```

Weiteres Beispiel: Fibonacci-Zahlen

Das folgende Listing zeigt ein Programm, dass die Fibonacci-Zahlen zur Compile-Zeit und zur Laufzeit auswertet und die Zeiten misst:

```

1 // fibonacci.cc:
2 // Compute fibonacci numbers at run- and compile time and compare
3 // the time used for it.
4 #include <iostream>
5 #include <cstdio>
6
7 // rekursive runtime variant
8 unsigned long Fibonacci_Simple(unsigned long n)
9 {
10     if (n==0) return 0;
11     else if (n==1) return 1;
12     else
13         return Fibonacci_Simple(n-1) + Fibonacci_Simple(n-2);
14 };
15
16 // rekursive template instantiations
17 template<unsigned long N>
18 class Fibonacci
19 {
20 public:
21     enum { value = Fibonacci<N-1>::value +
22             Fibonacci<N-2>::value };
23 };

```

Weiteres Beispiel: Fibonacci-Zahlen

Das folgende Listing zeigt ein Programm, dass die Fibonacci-Zahlen zur Compile-Zeit und zur Laufzeit auswertet und die Zeiten misst:

```
25 // template specializations to abort iterative template instantiation
26 template<>
27 class Fibonacci<1> {
28 public:
29     enum { value = 1 };
30 };
31
32 template<>
33 class Fibonacci<0> {
34 public:
35     enum { value = 0 };
36 };
37
38 // main program
39 int main()
40 {
41     // call of recursive Fibonacci
42     clock_t begin_rec = clock();
43     unsigned long result = Fibonacci_Simple(45);
44     clock_t end_rec = clock();
45     printf("Recursive Fib(45) = %ld  computed in %lf secs.\n",
46           result, (double)(end_rec - begin_rec)/CLOCKS_PER_SEC);
```

Weiteres Beispiel: Fibonacci-Zahlen

Das folgende Listing zeigt ein Programm, das die Fibonacci-Zahlen zur Compile-Zeit und zur Laufzeit auswertet und die Zeiten misst:

```

47
48 // call of templated Fibonacci
49 begin_rec = clock();
50 result = Fibonacci<45>::value;
51 end_rec = clock();
52 printf("Templated Fib(45) = %ld  computed in %lf secs.\n",
53        result, (double)(end_rec - begin_rec)/CLOCKS_PER_SEC);
54
55 return 0;
56 }

```

Zeiten bei mir für $n = 45$:

- Rekursive Funktion: 31 s (da nicht optimiert durch z.B. Werte-Cashes),
- Templates : 0 s (klar :-)).

Template Meta Programming

Wofür brauchen wir Template Meta Programme?

- Idee: Hybrider Ansatz, also eine Zerlegung des Programms in
 - ein TMP, läuft zur Kompilier-Zeit
 - ein „normales Programm“
- ⇒ Laufzeit-Verbesserungen (etwa durch massives Inlining)
- Generische Programmierung und TMP werden fast immer dann verwendet, wenn eine Bibliothek gleichzeitig:
 - performant und
 - flexibelsein soll!
 - Spezialisierte Algorithmen für „kleine“ Klassen
 - Beispiele: komplexe Zahlen, Tensoren, Gitter, ...

Template Spezialisierungen

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

- Abweichungen von der Template-Schablone werden explizit ausprogrammiert,
- 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> class Sorter
{
    void sort(T* array) { /* sort here */ };
    ...
};

// sorting a single field array is simple...
template <class T> class 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;  
}  
  
// specialization for small N (e.g. N=3) speeds up calculation  
double dotproduct(const double* a, const double* b, 3)  
{  
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];  
}
```

Weiterführende Literatur

Es existiert eine Vielzahl Literatur zu den ausschnittsweise vorgestellten Optimierungsmöglichkeiten durch die vorgestellten Techniken (insb. Statischer Polymorphismus).

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 Template Meta Programming)