Einführung in die Programmierung mit C++

Peter Fischer

Folien basierend auf früheren Kursen von
Florian Painke, Volker Lindenstruth, Udo Kebschull

Inititut für Technische Informatik
ZITI
B6, 26, Mannheim
Universität Heidelberg

Kursinhalt

Einleitung

- Was heißt eigentlich „Programmieren“?
- Hallo Welt!
- Werkzeuge

Strukturierte und Prozedurale Programmierung

- Elementare Datentypen & Operatoren
- Kontrollstrukturen
- Zusammengesetzte Datentypen
- Funktionen & Gültigkeitsbereiche
- Zeiger & Speicherverwaltung

Kursinhalt

Objektorientierte Programmierung

- Grundzüge der Objektorientierung
- Klassen & Methoden
- Vererbung & Polymorphie

Weiterführende Themen

- Ausnahmebehandlung
- Schablonen
- Die C++ Klassenbibliothek

Literatur

- Die 'Bibel' von Bjarne Stroustrup:
  Die C++ Programmiersprache (Addison-Wesley)
  (eher ein / *das* Nachschlagewerk)
- Auf unserem Niveau: fast alle Bücher über C++
- `http://www.cplusplus.com/`

Lektion 1

Was heißt eigentlich „Programmieren“?

Vom Problem zum Programm

Definition

Programmieren ist der gesamte Prozess von der Erfassung eines Problems bis zur fertigen Umsetzung der Lösung auf einem Computer und deren Wartung.

- Problemstellung
- Forderungsanalyse
- Architektur, Algorithmen
- Implementierung
- Tests
- Wartung

Vom Problem zur Architektur

Was wollen die eigentlich von mir?

- Konkrete Problemstellung
  z.B. Ballistischer Wurf
- Analyse der Anforderungen
  z.B. Plot verschiedener Trajektorien
- Festlegen der Architektur
  z.B. Struktogramm des groben Programmablaufes
- Fehlerquellen
  - Unklare Problemstellung
  - Falsch verstandene Anforderungen
  - Denkfehler in der Architektur

Von der Architektur zum Programm

Wie bringe ich mein Problem dem Computer bei?

- Implementieren der Architektur
  z.B. C++Quellcode schreiben
- Test des Programmes
  z.B. Vergleich der Plotdaten mit von Hand gerechneten Werten
- Fehlerquellen
  - Tippfehler, syntatktische Fehler
    Werden vom Übersetzer gefunden
  - Semantische Fehler, Denkfehler im Algorithmus
    Müssen vom Programmierer gefunden werden

Was ist ein Algorithmus ?

- Ein Algorithmus ist eine *genaue* Beschreibung *aller Einzel*schritte, die zur Lösung eines Problems notwendig sind.
- Beispiel: 'Finde die kleinste von N Zahlen' ($Z[1] \ldots Z[N]$)
  Ein erster Versuch (nicht gut... Was ist bei N=1 ?)
  1. Setze $x = Z[1]$
  2. Setze $i = 2$
  3. Wenn $Z[i] < x$, dann setze $x = Z[i]$
  4. Setze $i = i + 1$. Wenn $i > N$, dann fertig, sonst mache bei 3 weiter.
- Unterschiedliche Algorithmen, die das gleiche Problem lösen, können
  - unterschiedlich schnell sein
  - unterschiedlich viel Speicher benötigen
  - ...
- Beispiel (Tafel): Sortieren

Ziele dieses Kurses

- Erfassen einfacher Problemstellungen
- Erlernen der syntaktischen Struktur von C++
- Erlernen der Sprachsemantik von C++
- Erlernen des Umgangs mit den nötigen Werkzeugen

Am Ende dieses Kurses sollte jeder Teilnehmer befähigt sein, einfache Programme zur Lösung einfacher Probleme zu schreiben und sich die für komplexere Aufgaben nötigen Kenntnisse eigenständig zu erarbeiten.

Lektion 2

Hallo Welt!

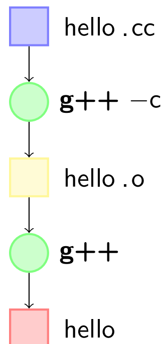
Das erste Programm in C++

```
 1 #include <iostream>
 2
 3 // Uebersetzen mit        g++ -Wall -c hello.cc
 4 // Binden mit             g++ -o hello hello.o
 5
 6 int main( void )
 7 {
 8   // Begrueszung ausgeben
 9   std::cout << "Hallo Welt!" << std::endl;
10
11   // Programm beenden
12   return 0;
13 }
```

Programm 1: hello.cc

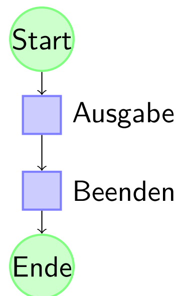
Übersetzen und Binden

- Quelltext ist von Menschen lesbare Form
- Computer will ausführbare Form
  - Übersetzen erzeugt Objektdatei
    g++ −c hello.cc
  - Binden erzeugt ausführbare Datei
    g++ −o hello hello.o
- Ausführbarung in der Shell
  ./ hello
  Hallo Welt!
- Änderungen immer im Quelltext
  Erneutes Übersetzen und Binden nötig



Zurück zum Quelltext...

- **#include** <iostream> lädt Ein-/Ausgabe
- Eigentliche Funktionalität in main()
- Kommentare beginnen mit //
- Blöcke von Kommentaren mit
  / ... Text ... */*
- Ausgaben über std :: cout
  std :: cout kann mit << alles „zugeworfen“ werden, was ausgegeben werden soll
- Programm endet mit **return** 0;



Eine Frage des guten Stiles

- Achten Sie auf die Struktur Ihrer Programme
  - Einrückungen deuten logische Struktur an
  - Leerzeichen verbessern die Lesbarkeit
  - Kommentare vermitteln Verständnis (triviale Kommentare jedoch vermeiden!)
- Bereits nach einem Monat ist selbstgeschriebener Quellcode ohne Kommentare nur noch schwer zu verstehen
- Bedenken Sie, dass möglicherweise andere Ihren Quellcode lesen und warten müssen
- Schauen Sie sich andere Programme an.
- Gewöhnen Sie sich eine einheitlichen Stil an.

Lektion 3

Werkzeuge

Unix Programme

- Verzeichnisse

pwd	aktuelles Verzeichnis anzeigen
ls −al [<dir>]	Dateien in einem Verzeichnis anzeigen
cd <dir>	Verzeichnis wechseln (z.B. **cd** ../ test /v1)
.	aktuelles Verzeichnis
..	übergeordnetes Verzeichnis
xxx/yyy/zzz	Verkettung von Verzeichnissen mit /
mkdir <dir>	Verzeichnis anlegen
rmdir <dir>	Verzeichnis löschen

- Dateien

cp <src> <dest>	Datei kopieren
mv <src> <dest>	Datei verschieben
rm <file>	Datei löschen
cat <file>	Datei ausgeben
less <file>	Datei seitenweise ausgeben (q zum Beenden)

Weitere Programme

- Suchen

find <dir> −name <file>	Datei suchen
grep <pattern> <file>	Ausdruck in Datei suchen

- Hilfe

man <cmd>	Beschreibung zu Kommando anzeigen
info	Hilfesystem anzeigen

- Programmierung

g++ −c <file>	Datei übersetzen
g++ −o <file> <obj> [...]	Objektdateien binden
gdb <file>	Debugger aufrufen

Weitere Linux Tipps

- Mit der Taste 'Hochpfeil' kann durch frühere Kommandos geblättert werden.
- Mit 'Tabulator' wird eine begonnene Eingabe automatisch vervollständigt, soweit das möglich ist (z.B. werden Dateinamen vervollständigt bis es mehrere Möglichkeiten gibt).
- Ein Programm kann mit ctrl–c abgebrochen werden.
- Ein Programm wird mit '&' am Ende der Kommandozeile unabhängig vom Eingabefenster ausgeführt, z.B.
  gedit file.cc &
- Blockiert ein Programm die Shell (das Eingabefenster), so kann es mit ctrl–z bg<ret> in den Hintergrund geschickt werden.

Lektion 4

Elementare Datentypen & Operatoren

Programme arbeiten mit Daten

- Zum Rechnen werden Variablen und Operatoren benötigt
- Variablen enthalten Daten des Programmes
  z.B. Position des Geschosses auf Trajektorie
- Variablen haben jeweils Typ und Namen
  z.B. ganzzahlig, reell, ...
- Es gibt jeweils unterschiedliche Genauigkeiten (byte, int, ...)
- Operatoren (+, -, ...) manipulieren die in Variablen gespeicherten Werte

Elementare Datentypen

- Ganze Zahlen

  short, **int**, **long** — -7, 42, 123456789

  unsigned short, **unsigned int** — positive Zahlen incl. 0
- Reele Zahlen

  float — 1., .5, 3.1415, 1.6022e$-$19

  double — höhere Genauigkeit (mehr Stellen, größerer Wertebere

  long double — noch höhere Genauigkeit
- Wahrheitswerte

  bool — **true**, **false**
- Zeichen (Buchstaben,...)

  char — 'a', 'Z', '3', '?', ...
- Leerer Datentyp

  void

Definition von Variablen

- Einfache Definition

```
int index; float x;
```

- Mehrfache Definition (alle gleicher Typ)

```
float a, b, c;
```

- Definition mit Initialisierung

```
char c = 'a';
unsigned long grosse_zahl = 123456789;
float g = 9.81, v = 3.4;
```

- Definition einer Konstanten, die später im Programm nicht mehr verändert werden kann

```
const float pi = 3.14159;
```

Namen von Variablen, Funktionen,...

- Buchstaben: a .. z und A .. Z
  Groß-/Kleinschreibung wird unterschieden
- Unterstrich: _
- Ziffern: 0 .. 9 Nicht als erstes Zeichen!
- Umlaute und Sonderzeichen sind nicht erlaubt
- Sinnvolle Namen verwenden!
  Namen sollten für die jeweilige Funktion sprechen:
    kontostand, start_value, ...
  Für Zähler und Indizes (int) reicht meist ein Buchstabe.
    Oft: i, j, k, l, m, n...
  Für reele Zahlen: a, x, y, ...
- Einheitliche Konventionen verwenden, auch für Groß-/Kleinschreibung

Literale

- 'Zeichenfolgen, die zur Darstellung der Werte von Basistypen definiert bzw. zulässig sind'
  z.B. **true**, **false**, 'A', '4', 15, 3.14159, ...
- Literale sind typisiert

true, **false**	**bool**	'a', 'Z', '3'	**char**
−7, 23, 42	**int**	1234l	**long**
23u	**unsigned**	42ul	**unsigned long**
1., 1e0, 23f	**float**	42lf	**double**

- Ganzzahlige Zahlenbasis

Dezimal:	−7, 23, 42
Hexadezimal (0x..):	0xaffe, 0xD00F
Oktal (führende Null)	077, 0123

Vorsicht!

```
1   int   i = 3.5;      // i ist 3!
2   int   i = 011;      // i ist 9!
3   float x = 6  / 5;   // x ist 1!
4   float x = 6. / 5;   // x ist 1.20000004768372
5                       //   (maschinenabhängig)
```

Fluchtsequenzen

- Problem: ' und " haben spezielle Bedeutung
  Begrenzung von Zeichenliteralen bzw. -ketten
  wie soll ' selbst als Zeichen in einem Literal dargestellt werden?
- Linksseitiger Schrägstrich \ leitet Fluchtsequenz ein

`'\''`	' als Literal
`'\"'`	" als Literal
`'\\'`	\ als Literal

- Fluchtsequenz auch für sogenannte nichtdruckende Zeichen

`'\0'`	ASCII Code 0	`'\n'`	New Line
`'\a'`	Alarm	`'\r'`	Carriage Return
`'\b'`	Backspace	`'\t'`	Horizontal Tab
`'\f'`	Form Feed	`'\v'`	Vertical Tab

ziti

Wertebereiche

- Tatsächliche Größe nicht standardisiert
  Lediglich Mindestgröße und aufsteigende Reihenfolge
- Wertebereiche über **#include** <limits> zugänglich

```
1  #include <iostream>
2  #include <limits>
3
4  int main()
5  {
6    std::cout << "int: "
7      << std::numeric_limits<int>::min()
8      << " .. "
9      << std::numeric_limits<int>::max()
10     << std::endl;
11   return 0;
12 }
```

Programm 2: limits.cc

Arithmetische Operatoren

- Einfache Operatoren

()	Klammerung von Ausdrücken
+ −	Addition und Subtraktion, bzw. unäres Minus
* /	Multiplikation und Division
%	Modulo (Rest einer ganzzahligen Division)

- Inkrement und Dekrement

++ −−	präfix bzw. postfix Inkrement und Dekrement
	b = a++; ist dasselbe wie b = a; a = a + 1;
	b = ++a; ist dasselbe wie a = a + 1; b = a;

- Zuweisung

=	einfache Zuweisung
+= *= ...	zusammengesetzte Zuweisung
	b += a; ⇔ b = b + a; etc.
x ? A1 : A2;	Ausdruck ergibt A1 oder A2, je nach Wahrheitswert von x, z.B. y = (x>3) ? 5.0 : 7.0

Logische und Bitweise Operatoren

- Vergleiche

== !=	Gleichheit, Ungleichheit
< >	kleiner, größer
<= >=	kleiner oder gleich, größer oder gleich

- Logische Verknüpfungen

!	unäres logisches nicht
&& ||	und, oder

- Bitweise Operatoren

~	unäres bitweises nicht
& | ^	bitweises und, oder, exklusiv-oder
<< >>	Bits nach links- bzw. rechts schieben

Typumwandlung

- Implizite Typumwandlung

```
int a, b = 10;
float pi = 3.14159;
a = b * pi; // b nach float gewandelt, a ist 31
```

- Zwischen elementaren Typen wird automatisch umgewandelt
- Bei Rechnungen mit Elementen unterschiedlichen Typs
  Wandlung aller Elemente in Typ mit größtem Wertebereich
  Warnung bei impliziter Typumwandlung mit Verlust
- Explizite Typumwandlung

```
int a, b = 10;
float pi = 3.14159;
a = b * (int) pi ; // a ist 30 oder
a = b * static_cast<int>( pi );
ch2 = (char) ((int)'A' + 2); // ch2 ist 'C'
```

Rangfolge der Operatoren

::

. −> (..) [..] *postfix* ++ −− *typecast*

präfix ++ −− *unäre* ~ ! − & ∗ **sizeof** **new** **delete**

∗ / %

+ −

<< >>

< <= > >=

== !=

&

^

|

&&

||

?:

= += −= ∗= /= %= <<= >>= &= ^= |=

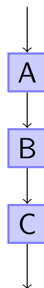
throw ,

Lektion 5

Kontrollstrukturen

Programmverlauf und Anweisungen

- Bisher streng monotoner Verlauf
- Ermöglicht nur feste Anzahl von Berechnungen
- Weitere Anweisungen sind nötig, um Programme flexibler zu gestalten



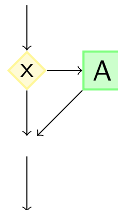
Anweisungen

- Leere Anweisung
  ;
- Einfache Anweisungen (was wir bisher kennen)
  - Definition von Variablen
  - Einfache Anweisungen mit Operatoren
  - Beenden von main() mit **return** (...);
- Zusammengesetzte Anweisungen (Anweisungsblock)
  { Anweisung_A; Anweisung_B; ... }
- Kontrollstrukturen (brechen Monotonie auf)
  - *Bedingungen* ermöglichen Alternativen
  - *Schleifen* ermöglichen Wiederholungen

ziti

Bedingte Anweisung

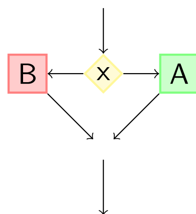
```
1  if ( Ausdruck_x ) Anweisung_A;
```



- Ausführung von Anweisung_A, wenn Ausdruck_x wahr
- Ausführen von mehreren Anweisungen mit {...}
- *WICHTIG:* C++ interpretiert jeden Wert ungleich 0 als „wahr“

Bedingte Anweisung mit Alternative

```
1  if ( Ausdruck_x ) Anweisung_A;
2  else Anweisung_B;
3                         // oder:
4  if ( Ausdruck_x )
5     Anweisung_A;
6  else
7     Anweisung_B;
```



- Ausführung von Anweisung_A, wenn Ausdruck_x wahr
- Ausführung von Anweisung_B, wenn Ausdruck_x falsch

ziti

Schachtelung bedingter Anweisungen

```
1  if ( Ausdruck_x )
2     if ( Ausdruck_y )
3        Anweisung_A;
4     else
5        Anweisung_B;
6  Anweisung_C;
```

```
1  if ( Ausdruck_x )
2     if ( Ausdruck_y )
3        Anweisung_A;
4  else
5     Anweisung_B;
6  Anweisung_C;
```

- Gefahr von Mißverständnissen der Zugehörigkeit von **else**
- Verwendung von geschweiften Klammern schafft Klarheit

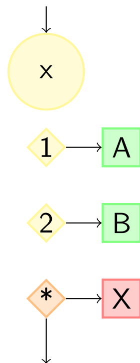
Schachtelung mit Klammern

```
 1  if ( Ausdruck_x ) {
 2    if ( Ausdruck_y ) {
 3      Anweisung_A1;
 4      Anweisung_A2;
 5    } else
 6      Anweisung_B;
 7  } else {
 8    Anweisung_C1;
 9    Anweisung_C2;
10  }
```

- Andere Arten der Einrückung möglich

Mehrfachverzweigung

```
1   switch ( Ausdruck_x ) {
2   case Konstante_1:
3       Anweisung_A; ...
4       break;
5
6   case Konstante_2:
7       Anweisung_B; ...
8       break;
9
10  default:
11      Anweisung_X; ...
12  }
```



Mehrfachverzweigung

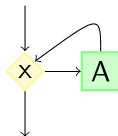
- Auswertung von Ausdruck_x
- Ergibt sich Ausdruck_x zu einer der Konstanten eines **case**-Zweiges, dann Abarbeitung der darauf folgenden Anweisungen bis zur nächsten **break**-Anweisung
- Steht zwischen zwei **case**-Zweigen kein **break**, dann wird Abarbeitung einfach fortgesetzt!
  case-Zweige immer mit **break** beenden!
- Wird kein passender **case**-Zweig gefunden, dann wird **default**-Zweig gewählt falls vorhanden
- **default**-Zweig ist optional

ziti

Einfache kopfgesteuerte Schleife

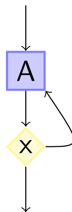
```
1  while ( Ausdruck_x )
2    Anweisung_A;
```

- So lange Ausdruck_x wahr ist, wird Anweisung_A ausgeführt
- Anweisung_A wird evtl. gar nicht ausgeführt
- Wenn Anweisung_A nicht dazu führt, dass Ausdruck_x irgendwann falsch wird, führt dies zu einer Endlosschleife



Einfache fußgesteuerte Schleife

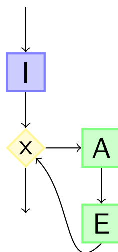
```
1  do Anweisung_A
2  while ( Ausdruck_x );
```



- Anweisung_A wird mindestens einmal ausgeführt
- So lange Ausdruck_x wahr ist, wird Anweisung_A wiederholt ausgeführt

Flexible Schleife

```
1  for ( Anweisung_I;
2        Ausdruck_x;
3        Anweisung_E )
4    Anweisung A;
```



- Zunächst wird Anweisung_I (Initialisierung) ausgeführt
- So lange Ausdruck_x wahr ist, werden Anweisung_A und Anweisung_E (Inkrement) ausgeführt

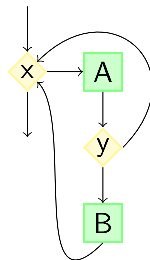
ziti

Beispiele für Flexible Schleifen

```
1   float x = 0.123;
2   for ( int i=0; i<100; i++) x = x * x - 1;
3
4   int hexzahl = 0xa5;
5   for (int bit = 0x80; bit>0; bit>>=1)
6     std::cout << ( (hexzahl & bit) ? '1' : '0');
7   std::cout << "\n";
```

Frühzeitiges Fortsetzen

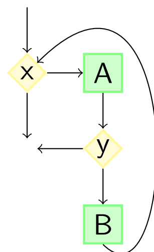
```
1  while ( Ausdruck x ) {
2    Anweisung A;
3    if ( Ausdruck y ) continue;
4    Anweisung B;
5  }
```



- Anweisung A wird in jedem Schleifendurchlauf ausgeführt
- Falls Ausdruck y wahr ist, wird Schleife *frühzeitig* fortgesetzt
- Anweisung B wird nur ausgeführt, wenn Ausdruck y falsch ist
- Bei **for**-Schleife wird Inkrement in jedem Fall ausgeführt

Frühzeitiges Beenden

```
1  while ( Ausdruck x ) {
2    Anweisung A;
3    if ( Ausdruck y ) break;
4    Anweisung B;
5  }
```



- Anweisung A wird in jedem Schleifendurchlauf ausgeführt
- Falls Ausdruck y wahr ist wird, Schleife *frühzeitig* beendet
- Anweisung B wird nur ausgeführt, wenn Ausdruck y falsch ist

Lektion 6

Zusammengesetzte Datentypen

Strukturen

```
1  struct vector
2  {
3    float x, y, z;
4  };
```

- Zusammenfassung benannter Elemente beliebigen Typs
- Typdefinition vor main()
- *Definition* einer Variablen:
  z.B. `vector pos, r;`
- *Zugriff* auf Strukturelemente über den Zugriffsoperator .
  z.B. `pos.x = 1.0; float length = sqrt(r.x * r.x + r.y * r.y)`
- *Optionale Initialisierung* (bei Deklaration):
  z.B. `vector start = { .0, 1.8, .0 };`

Aufzählungen

```
1  enum language
2  {
3     unknown,
4     german,
5     french,
6     klingon
7  };
```

- Eine Variable vom Typ ‘language ’kann (nur) die Werte unknown, german,... annehmen.
- Typdefinition vor main()
- Definition & Initialisierung einer Variable dieses Typs: z.B. language ConfLang = french;
- Abfrage z.B. **if** (ConfLang == german) {...}
- Explizite Zuordnung zu int möglich, z.B. **enum** newtype {ok = 3, bad = 7};

Felder (Arrays)

- Indizierte Zusammenfassung von Elementen gleichen Typs
- Einfache Definition
  z.B. **int** koeffizienten [5];
- Definition mit Initialisierung
  z.B. **int** koeffizienten [] = { 1, 2, 4, 8, 16 };
- Mehrdimensionale Felder
  z.B. **float** matrix [2][3] =
  { { 1., .0, .0 }, { .0, 1., 1.} };
- Zugriff auf Feldelemente über Indexoperator [..]
  z.B. koeffizienten [4] = 23;
- Erstes Element hat Index 0
- Ausblick; Nach **int** A[5]; ist A einfach ein Zeiger auf das erste int im Feld. Die Größe des Feldes kann daraus nicht ersehen werden

Primitive Zeichenketten (‘c-strings’)

- Primitive Zeichenketten (‘strings ’) sind Zeichenfelder
- Definition z.B.: **char** name[] = "Bond,␣James␣Bond";
- Um das Ende der Zeichenkette erkennen zu können, wir (automatisch) eine Null (0) angehängt.
  Daher hat **char** str [] = "Tom"; die Länge 4!
- Nur sehr eingeschränkte Manipulationsmöglichkeiten (kein automatisches Kopieren,...)
- C++ string -Klasse bietet mehr Flexibilität
  Zugänglich über **#include** <string>
  z.B. std :: string name = "Tom";

Mischungen

```
1  struct point3 { float x, y, z; };
2  const int N = 100;
3  point3 v[N];
4
5  const int NDAY = 365;
6  enum gender {male, female};
7  struct Mitarbeiter
8    {
9      char    Name;     // Erst mal nur ein Buchstabe
10     gender  g;
11     bool    IsOnHoliday[NDAY];
12   };
13   Mitarbeiter XYZ;
14   XYZ.g = female;
15   XYZ.IsOnHoliday[13] = false;
16   XYZ.Name = 'A';
17
18   if (XYZ.g == male)...
```

Noch ein Beispiel

```
1  struct T_Land {
2    string name;            // z.B. "Frankreich"
3    int    Vorwahl           // 33
4  };
5
6  struct T_Adresse {
7    T_Land land,            // s.o.
8    int    PLZ              // eine Zahl
9  };
10
11 struct T_Mitarbeiter {
12   string    name,
13   T_Adresse adr
14 };
15
16 Mitarbeiter Team1[100]; // Platz für 100 Mitarbeiter
17
18 Team1[1].name = "Tom";
19 Team1[1].adr.land.name = "England";
20 ...
```

Typdefinition

- Definition eines neuen Typs:
  - Implizit über **struct** oder **enum**
  - Explizit mit **typedef**

```
1 typedef unsigned int UINT;
2 UINT k = 3;
```

Lektion 7

Funktionen & Gültigkeitsbereiche

Strukturierte Programmierung

- Bisher Programm komplett in `main()`
- Beobachtungen:
  - Programmteile können mehrfach vorkommen
  - Programmteile können für andere Programme interessant sein
  - Bereits mittlere Programme werden schnell unübersichtlich
- Abhilfe: *Funktionen* kapseln Programmteile in Block
- Übergang zur *prozeduralen Programmierung*

Funktionsdefinition

- Funtionen bestehen aus *Kopf* und *Rumpf*:

```
1  Rückgabetyp Name (typ name, ...)          // Kopf
2  {
3    Etwas sinnvolles machen...;               // Rumpf
4    return Ergebnis;       // Ergebnis vom Typ Rückgabet
5  };
```

- Kopf:
  - Name
  - Rückgabetyp oder **void** für Funktionen ohne Rückgabe
  - Parameter: Jeweils Typ und Name, durch Komma getrennt
- Rumpf:
  - definiert Verhalten
  - Funktionen mit Ergebnistyp müssen **return** xxx; enthalten
  - Ohne Rückgabewert (**void**): optionales leeres **return**;
- Parameter und im Rumpf definierte Variablen sind lokal.
  Sie werden bei jedem Aufruf reserviert und initialisiert
- Funktionsdefinitionen können nicht geschachtelt werden

ziti

Beispiel für Funktionsdefinition und Aufruf

```
1  ...
2  float ArithmetischesMittel(float x1, float x2){
3    float mittel;
4    mittel = (x1 + x2) / 2;
5    return mittel;
6  }
7
8  float GeometrischesMittel(float x1, float x2){
9    return sqrt(x1 * x2);
10 }
11
12 int main (void) {
13   cout << ArithmetischesMittel(3,4) << endl;
14   cout << GeometrischesMittel(3,4) << endl;
15 }
```

Fakultät iterativ

```
 1  #include <iostream>
 2
 3  // Funktion zur Berechnung der Fakultaet
 4  unsigned factorial( unsigned n )
 5  {
 6     if ( n == 0 ) return 1; // 0! ist 1
 7
 8     // n! ist n * (n-1) * .. * 2 * 1
 9     unsigned fact = n;
10     while ( n > 1 ) fact *= --n;
11
12     return fact;
13  }
14
15  int main()
16  {
17      std::cout << "10! ist "
18        << factorial( 10 )
19        << std::endl;
20
```

Funktionsaufruf

- Aufruf durch Name und Funktionsoperator (..)
  Übergabe von Argumenten in der Klammer
  z.B. **unsigned** x = factorial (10);
- Funktionen können Funktionen aufrufen
  Funktionen können auch sich selbst wieder aufrufen
  Rekursion; Abbruchbedingung wichtig, sonst Endlosrekursion
- Auch main() ist eine Funktion
  Wird von Laufzeitumgebung aufgerufen
  return in main() kehrt zur Laufzeitumgebung zurück
- Funktion muß vor Aufruf bekannt sein
  Entweder Deklaration oder Definition
- *Deklaration* über Funktionskopf abgeschlossen durch ;
  z.B. **unsigned** factorial (**unsigned** n);
- *Definition* muss dann später erfolgen!

Übergabe von Parametern nach main()

- Die Parameter aus der linux - shell werden als array von **char*** übergeben

```
1  #include <iostream>
2  using namespace std;
3
4  int main (int argc, char* argv[])
5  //int main (int argc, char** argv)    // Äquivalent
6  {
7     cout << "Anzahl␣parameter␣=␣" << argc << endl;
8     for (int i=0; i<argc; i++)
9        cout << "Parameter" << i
10            << ":␣" << argv[i] << endl;
11 }
```

Mathematik - Funktionen

- Die math-Bibliothek (**#include** <cmath>) stellt viele nützliche Funktionen zur Verfügung
- sqrt(), pow(base, exponent)
- exp(), log()
- sin(), cos(), . . . , sinh(), cosh(), . . .
- fabs() (Betrag)
- Konstanten wie M_PI (π),...

Gültigkeitsbereiche

Definition

Der Gültigkeitsbereich beschreibt, in welchem Kontext ein Name in einem Programm verwendet werden kann.

- Zwei Gültigkeitsbereiche: global und lokal
- Funktionen sind global gültig
- Variablen lokal zu Funktion
- Jeder Block stellt abgeschlossenen Gültigkeitsbereich dar
  z.B. **for** (**int** i = 0; i < 5; i++) {..}
  i besitzt Gültigkeit nur innerhalb des **for**-Blocks
- Zugriff auf globale Variable (unschön) mit ::name

Globale Variablen

- Variablen können außerhalb von Funktion definiert werden
- Zugriff aus allen Funktionen möglich
- Gefahr von Inkonsistenzen hoch
- Fehlersuche erschwert
- Vermeiden!

Aufruf mit Argumentwert

```
1  float fff (float wert, int anzahl) {
2     ..
3     wert = 7.0;
4     return 3.0;
5  }
6  ...
7  float x = 2.0;
8  z = fff(x,3);
```

- Funktionen können beliebige Argumente erhalten
- Funktionen können *nur einen* Wert (via *return*) zurückgeben
- Die Funktionsparameter (wert, anzahl) sind *lokale* Variablen
- Argumente werden in Parameter kopiert
- Änderungen an Parametern *innerhalb* einer Funktion wirken sich nicht auf die Wert 'außerhalb' aus (i.e. x bleibt 2.0)!

Aufruf mit Argumentreferenz

```
1  void swap( int & x, int & y )
2  {
3    int t = x;
4    x = y;
5    y = t;
6  }
```

- Argumente werden hier an die Funktion „durchgereicht“ (es wird ein Pointer = eine Adresse übergeben)
- Das ist (bei großen Argumenten) viel effizienter.
- Änderungen wirken auf Argumente zurück!
- Referenzparameter werden mit & vor dem Namen definiert
- Mit dem **const**-Modifier wird der Compiler angewiesen, keine Änderungen zu erlauben. Damit ist die Übergabe 'schnell' und 'sicher'.

Warum ist **const** bei Referenzen wichtig?

Die Übergabe eines Parameters als Referenz mit & erlaubt es der Funktion, den Parameter zu ändern. Nun gibt es Fälle, in denen das kaum möglich ist:

- die Funktion wird mit einer Konstanten aufgerufen (z.B. einer Zahl)
- das Argument ist ein temporärer Ausdruck, z.B. a+b

In solchen Fällen scheitert der Compiler daher. Wird das Argument (die Referenz) in der Funktion jedoch als **const** qualifiziert, so ist dem Compiler klar, dass er diese problematischen Argumente nie ändern muss, und es klappt.

Namensräume

- Strukturierung großer Programme über Namensräume
- Namensraum wird über Schlüsselwort **namespace** und Name definiert und leitet Block ein, der Elemente kapselt: **namespace** Space1 { ... definitionen ...}
- Elemente eines Namensraumes können über Namen und Auflösungsoperator :: erreicht werden
- Globaler Namensraum hat leeren Namen
- C++ verwendet Namensraum std

Überladen von Funktionen und Operatoren

- Funktionen mit unterschiedlicher Parameterliste können gleiche Namen haben
- Auswahl der aufzurufenden Funktion erfolgt anhand des Typs der Argumente
  Bei Mehrdeutigkeiten explizite Typwandlung nötig
- Funktionen mit gleichem Namen sollten ähnliche Funktionalität haben!
- Operatoren sind in C++ Funktionen zugeordnet
- Name einer Operatorfunktion beginnt mit Schlüsselwort **operator**, gefolgt von Operatorsequenz
  z.B. a = b + c; wird **operator**=(a, **operator**+(b, c));
- Auch Operatoren können überladen werden

Operator überladen für selbstdefinierte Typen

```
 1  vector
 2  operator+( const vector & a,
 3             const vector & b )
 4  {
 5    vector c;
 6
 7    c.x = a.x + b.x;
 8    c.y = a.y + b.y;
 9    c.z = a.z + b.z;
10
11    return c;
12  }
```

Überladen des Ausgabeoperators

- cout und cin sind im Namensraum std definierte Variablen
- Ausgabe mit std :: cout über spezielle Überladungen von **operator**<<() für Typ von std :: cout
- Typ von cout ist std :: ostream
- Eingabe mit std :: cin entsprechend, Typ ist std :: istream

Ausgabe für selbstdefinierte Typen

```
1  std::ostream & operator<<(
2                 std::ostream & os, const vector & v )
3  {
4    os << "( " << v.x << ", "
5      << v.y << ", " << v.z << " )";
6
7    return os;
8  }
```

Lektion 8

Zeiger & Speicherverwaltung

ziti

Was sind Zeiger?

- Verweis auf andere Variable
- "Die Adresse im Speicher"
- Eine Referenz ist letzlich ein Zeiger, aber anders 'verpackt'
- Die dynamische Erzeugung der Referenzierten ist möglich
- Syntax ist näher an der Hardware
- Flexibles Konzept der dynamischen Speicherverwaltung
  Flexibilität bringt neue Fehlerquellen

Definition und Speicherverwaltung

- Einfache Definition über Typ und Typmodifizierer *
  z.B. **int** * ptr;
  - Leseweise: ptr ist ein Zeiger auf ein int (also ein **int** *)
  - Alternativ: * ptr ist ein **int**
  - ptr zeigt zunächst *irgendwo* hin.
- Verweis auf benannte Variable über *Adressoperator* &
  z.B. **int** i; ptr = & i;
- Alternativ: dynamische *Erzeugung* von neuem Speicherplatz (für ein int) mit **new**, z.B. ptr = **new int**;
- Dynamisch erzeugte Variablen müssen nach Benutzung mit **delete** *freigegeben* werden, z.B. **delete** ptr;
- Zeiger sind i.A. typgebunden, Ausnahme sind **void**-Zeiger

Verwendung von Zeigern

- Zeiger müssen vor Zugriff initialisiert sein
- Nach Freigabe darf kein Zugriff auf Zeiger mehr erfolgen
- Zugriff auf Referenzierte über Dereferenzierungsoperator $*$
  z.B. `* ptr = 42;`
- **void**-Zeiger können nicht dereferenziert werden
- Zeiger können auch auf Zusammengesetzte Typen zeigen
  z.B. `vector v; vector * pv; pv = & v;`
- Zugriff auf Strukturelemente
  z.B. `(* pv).x = .5; pv->y = 1.;`

ziti

Merke

- '&' heißt: "die Adresse von (nachfolgendes Objekt)"
  z.B.: '& x' ist die Adresse von x.
  x ist hier ein beliebiges Objekt.
- '*' heißt: "Der Wert auf den (nachfolgender Zeiger) zeigt"
  z.B.: '* x' ist der Wert, auf den x zeigt.
  x muss ein Zeiger (= Pointer) sein!

Identität von Zeigern und Feldern

- Dynamische Erzeugung von Feldern mit **new**[]
  z.B. ptr = **new int**[16];
- Zugriff auf Feldelemente
  z.B. *(ptr + 5) = 17; ptr[8] = 42;
- Freigabe dynamisch erzeugter Felder mit **delete**[]
  z.B. **delete**[] ptr;
- Zeigerarithmetik von C++
  - Element bei Index n: *(ptr + n) oder ptr[n]
  - Zeiger auf n. Element: ptr + n oder &ptr[n]
- Zeiger sind syntaktisch und semantisch identisch zu Feldern

ziti

Lektion 9

Grundzüge der Objektorientierung

Paradigmenwechsel

- Bisher: Funktionen und Datensätze (weitgehend getrennt behandelt)
- Jetzt: Objekte, die 'Dinge können'und wechselwirken
- Schlagworte:
  - Abstrakte Datentypen
  - Kapselung
  - Vererbung
  - Polymorphie

ziti

Abstrakte Datentypen

- Datentypen beschreiben zunächst nur, was Inhalt/Funktion sein kann
  Beispiel: Ein Objekt 'Form' hat eine Farbe, es kann aber so erst mal nicht gedruckt werden
  Eine spezielle Form, z.B. ein Dreieck, läßt sich drucken.
- Ein Vektor als abstrakter Typ ist Zusammenfassung von:
  - Vektorkomponenten
  - Addition und Subtraktion, evtl. äußeres Produkt
  - Multiplikation und Division mit Skalar
  - Skalarprodukt und Betrag
- Die Größe des Vektors spielt hier erst mal keine Rolle!

Kapselung

- Durch Kapselung werden Daten vor der Außenwelt verborgen
- Die Details der Implementierung einer Funktionalität sind unwichtig
- Durch kontrollierte Zugegriffe auf die Daten passieren weniger Fehler

Vererbung

- Abstrakte Datentypen stehen oft in Beziehung zueinander
- Beispiel:
  - Abstrakte Form hat Eigenschaften wie Linienfarbe, Füllung, ...
  - Spezielle Formen haben auch einen Rand, ..., z.B. Dreieck oder Kreis
  - Das Zeichnen ist bei jeder Form anders, das Ändern der Linienfarbe bei allen gleich
    $\rightarrow$ alle speziellen Formen *erben* Eigenschaften von Form
- Ableiten der Gemeinsamkeiten von Basis durch Vererbung
  - Gemeinsame Programmteile nur einmal implementiert
  - Erweiterungen der Basis kommen allen Erben zugute

Polymorphie

- Abgeleitete Objekte lassen sich auf Basis reduzieren
- Kenntnis verallgemeinerter Basis oft ausreichend
- Unterscheidung zwischen Spezialisierungen oft unnötig
- Voraussetzung:
  spezialisierte Objekte verhalten sich trotz Reduktion korrekt
- Beispiel: ostream, fstream, sstream etc. verhalten sich sehr ähnlich

Lektion 10

Klassen & Methoden

Klassen: Beispiel

```
1  class vector {                       //
2  public:
3    vector();
4    vector(float, float, float);
5    ~vector();
6    float Abs(void);
7  private:
8    float x,y,z;
9  };
10
11 vector v(1.1, 2.1, 3.0);
12 float laenge = v.Abs();
```

Klassen

- Klassen sind ein sprachliches Mittel zur objektorientierten Programmierung
- Sie erlauben die Definition abstrakter Datentypen
  - Klassenelemente beschreiben Inhalt
  - Methoden beschreiben Schnittstelle
- Kapselung über Schutzattribut **private**
- Veröffentlichung über Schutzattribut **public**
- Vererbung über Basisklassen
- Polymorphie über Zeiger

ziti

Definition

- Definition beginnt mit Schlüsselwort **class** und Name
- Klassenelemente und Methoden in Block danach
- Klassenelemente analog zu Strukturelementen
- Methoden sind gewöhnliche Funktionen
  In Klasse deklarierte Methoden müssen außerhalb um Klassenname erweitert definiert werden
- Öffentliche Teile mit **public**: eingeleitet
- Private Teile mit **private**: eingeleitet

Verwendung

- Typdefinition heißt Klasse
- Variable abstrakten Typs heißt Instanz
- Zugriff analog zu Strukturen
- Zusätzlich Aufruf von Methoden über . bzw. ->
- Methoden erhalten implizites Argument **this**
  Zeiger auf konkrete Instanz
  Operatoren jeweils um ein Argument reduziert

Spezielle Methoden

- Konstruktor wird bei Erzeugung einer Instanz aufgerufen
  - Definition mit Name der Klasse *ohne* Rückgabetyp
  - Konstruktor kann überladen werden
  - Implizit statische Methode
  - Aufruf durch Angabe von Argumenten bei Instanzierung
- Destruktor wird bei Freigabe einer Instanz aufgerufen
  - Definition wie Konstruktor mit vorangestelltem ∼
  - Erhält außer implizitem keine weiteren Argumente
- Methoden ohne Wirkung auf Inhalt als **const** definieren
- *Statische* Elemente (Schlüsselwort **static**) sind nur einmal vorhanden, *nicht* für jede Instanz!
- Funktionen außerhalb des Klassenkontext erhalten als **friend** Zugriff auf private Teile

Automatische Methoden

- Standardkonstruktor ohne weitere Argumente
  Implizit definiert, falls kein Konstruktor
- Kopierkonstruktor mit Referenzargument auf Klassentyp
- Typumwandlung durch Konstruktoren mit einem Argument
  Verhinderung durch **explicit**
- Zuweisungsoperator mit Referenzargument auf Klassentyp
  Implizit definiert, Wertekopie

Lektion 11

Vererbung & Polymorphie

Vererbung

- Klassen können von anderen abgeleitet werden
- Elemente und Methoden werden dabei vererbt
- Private Teile der Basisklasse nicht zugänglich
  Schutzattribut **protected** erlaubt Zugriff bei Vererbung
- Schutzattribute der Basisklasse können modifiziert werden
- Teile der Basisklasse können überdeckt werden

Verwendung

- Name der Basisklasse wird bei Definition nach dem Klassennamen durch : getrennt angegeben
- Vererbungsattribut steht vor Name der Basisklasse
- Konstruktor muß Konstruktor der Basisklasse aufrufen
  Aufruf folgt bei Definition Konstruktorkopf durch : getrennt
  Ggf. impliziter Aufruf des Standardkonstruktors
- Methoden können auf überdeckte Teile zugreifen
  Zugriff erfolgt über Name der Basisklasse und ::

Polymorphie

- Zeiger auf Basisklasse kann auf abgeleitete Klasse zeigen
- Nur Aufruf von Methoden der Basisklasse möglich
- Virtuelle Methoden ermöglichen weiterreichen von Aufrufen
  Defintion über Schlüsselwort **virtual** vor Rückgabetyp
- Frage: Wie soll sich reduzierte Instanz verhalten?
  - Wie Instanz der Basisklasse → Nicht-virtuelle Überdeckung
  - Wie Instanz der abgeleiteten Klasse → Virtuelle Überdeckung
- Destruktoren sollten Virtuell definiert werden
- Rückumwandlung mit **dynamic_cast**<..>(..)

Abstrakte Basisklassen

- Basisklasse oft nicht zur Instanzierung konzipiert
- Einzelne Methoden der Basisklasse oft leer
- Leere Methoden können rein-virtuell definiert werden
  Methodenkopf gefolgt von Nullzuweisung
- Basisklassen mit rein-virtuellen Methoden heißen abstrakt
- Abstrakte Basisklassen können nicht instanziert werden

Lektion 12

Ausnahmebehandlung

Klassische Fehlerbehandlung

- Zunächst: Betrachtung von Laufzeitfehlern
  z.B. Datei nicht gefunden, nicht genug Speicher, ...
- Laufzeitfehler müssen behandelt werden
  z.B. Fehlermeldung ausgeben, Aktion wiederholen, ...
- Klassisches Konzept: Behandlung vor Ort
  - Aktion ausführen
  - Auf Fehler prüfen
  - Fehler ggf. behandeln
- Nachteil: Funktionalität und Fehlerbehandlung verwoben
  Unübersichtlich, Umständlich

Ausnahmen (‘Exceptions ’)

- Ausnahmen ermöglichen nachgelagerte Fehlerbehandlung
  - Aktionen kann Ausnahme auslösen
  - Ausnahme unterbricht Programmfluss
  - Kann an anderer Stelle aufgefangen werden
- Fehlerbehandlung z.B. am Ende eines Abschnitts
- Vorteil: Funktionaler Programmfluss nicht unterbrochen
- Nachteil: erhöhter Aufwand durch die Laufzeitumgebung

Ausnahmen in C++

- Ausnahme kann mit **throw** ausgelöst werden
- Auslösung immer mit Objekt zur Repräsentation verbunden
  Repräsentant kann bel. Instanz sein, auch elementaren Typs
- Bei Auslösung wird aktueller Block sofort verlassen
- Auffangen in **catch**-Block nur nach **try**-Block möglich
- **try**-Block können mehrere **catch**-Blöcke folgen
- **catch**-Block fängt Repräsentanten bestimmten Typs auf
  Nur erster passender **catch**-Block wird verwendet
- **catch** (...) fängt alle Repräsentanten auf

Standardausnahmeklassen

- C++ stellt Standardausnahmeklassen bereit
  Einbinden mit **#include** <exceptions>
- Basisklasse aller Repräsentanten ist std :: exception
- Laufzeitfehler sind von std :: runtime_error abgeleitet
- Logikfehler sind von std :: logic_error abgeleitet
- Diverse spezifische Klassen
  z.B. std :: bad_alloc, wird ausgelöst, wenn **new** fehlschlägt

Struktur

```
1   try {
2     int * feld = new int[anzahl];
3     ...
4   }
5
6   catch ( const std::bad_alloc & e ) {
7     ...
8   }
9
10  catch ( const std::exception & e ) {
11    ...
12  }
```

Lektion 13

Schablonen

Generische Programmierung

- Funktionen oft für verschiedene Typkombinationen benötigt.
  Beispiel:
  <max(**int** a, **int** b) ...> funktioniert für **float** nicht richtig (warum?) und muss daher für **float** neu definiert werden
- C++ erlaubt Überladen von Funktionen
- Nachteil: oft identischer Quelltext
  Gefahr von Tippfehlern, Fehler müssen an mehreren Stellen behoben werden
- Lösungsansatz: Generische Programmierung
  - Funktion wird für generischen Typ nur einmal implementiert
  - Übersetzer generiert Implementierung für konkreten Typ

Schablonen (Templates)

- Schablonen ermöglichen in C++ generische Programmierung
- Schablone wird mit **template** eingeleitet
- Generischer Typ in spitzen Klammern mit <**typename** ...>
- In Definition wird generischer Typ verwendet
  Schablonen erlaubt für Funktionen und Klassen
- Bei Verwendung konkreter Typ in spitzen Klammern

```
1  template <typename T> T abs( T x )
2  { if ( x < 0 ) return -x; return x; }
3  ...
4  int a = abs<int>( b );
```

Lektion 14

Die C++ Klassenbibliothek

Überblick

- Die ‘standard template library’stellt viele nützliche Objekte zur Verfügung!
- Namensraum std
- Ein-/Ausgabe
- Zeichenketten
- Kontainerklassen
- Algorithmen
- Verwendung von stl Objekten kann viel Arbeit sparen und macht das Programm kurz und sicher!

Ein-/Ausgabe

- Ist ein spezieller stream
- **#include** <iostream>
- Objekte

cin , cout	Standardeingabe, -ausgabe
cerr	Fehlerausgaben
clog	Verlaufskontrolle
endl	Zeilenende

- Operatoren

<<, >>	Ausgabe an bzw. Eingabe von Stream

Dateizugriffe

- Ein weiterer stream
- **#include** <fstream>
- Klassen

ifstream , ofstream	Lese- bzw. Schreibzugriff
fstream	Wahlfreien Zugriff

- Methoden

open(), close ()	Öffnen und Schließen
read (), write ()	Zugriff auf Binärdateien
tellg (), seekg (),	Abfragen/Setzen des
tellp (), seekp()	Lese-/Schreibzeigers

- Beispiel: `ofstream f; f.open("file.txt"); f << 'A'; f.close();`

Zeichenketten

- Header **#include** <string>
- Konstruktoren

string (**const char** ∗)	Kopie von normalem C String
string (**const** string &)	Kopierkonstruktor

- Operatoren

=, +, +=	Kopieren, Zusammenführen und Anhängen
[..]	Zugriff auf einzelne Zeichen (besser at())

- Methoden

find (), rfind (), substr ()	Suchen und extrahieren von Substrings
insert (), erase (), replace ()	Operationen auf Substrings
length (), size ()	Länge bzw. Größe abfragen
c_str ()	zu C String umwandeln

Container

- Container-Objekte enthalten andere Objekte
  Oft benötigt, mit verschiedenen Inhaltstypen
  $\rightarrow$ Alle Container nutzen Schablonen (templates)
- Klassen für verschiedene Zugriffsmuster
  - vector
    Wahlfreier Zugriff ([]), schnelles Anfügen am Ende (push_back), langsames Einfügen am Anfang (push_front), Integer als Indizes
  - list
    Schnelles Einfügen überall, nur serieller Zugriff (kein [])
  - deque (double-ended queue)
    Wahlfreier Zugriff, schnelles Einfügen am Anfang und Ende, Integer als Indizes
  - stack: Einfügen/entnehmen nur am Ende
  - map
    Wahlfreier Zugriff, beliebige Typen als Indizes (Schlüssel)
    Oft auch Dictionary genannt
  - set: Menge. Jeder Wert kann nur einmal vorkommen.

ziti

Container

- Methoden

insert ()	Einfügen eines Elements
erase ()	Löschen eines Elements
clear ()	Löschen aller Elemente
size ()	Anzahl der gespeicherten Elemente abfragen
capacity ()	Anzahl der Elemente, die gespeichert werden können bevor Speicher allokiert werden muss
empty()	Prüfen, ob Kontainer leer ist
push_back()	Anhängen ans Ende
pop_back()	Entfernen vom Ende
push_front()	Einfügen am Anfang (außer vector)
pop_front()	Entfernen vom Anfang (außer vector)

- Operatoren

[..]	Wahlfreier Zugriff (vector, deque)

Iteratoren

- Bei allgemeinen Container (z.B. set) ist Zugriff auf die Elemente mit Indices nicht sinnvoll.
- Iteratoren sind ein allgemeines Konzept für Zugriff auf Elemente in containern
- Idee: 'Erstes Element', 'Nächstes Element' etc.
- Je nach Kontainer Vorwärts- und Rückwärtsiteratoren (iterator , reverse_iterator)

begin(), end()	Iterator auf Anfang, hinter Ende
rbegin(), rend()	Iterator auf Ende, vor Anfang

- Operatoren

++	Auf nächstes Element setzen
*	Dereferenzierung

Lektion 15

Häufige Fehler

Fehler 1

- Versehentliche Integer-Division: **int** i; **float** x = i / 3;
- Zugriff auf Elemente außerhalb von Arrays:
  int k[10]; **int** i=k[10];
- Keine Initialisierung von Variablen
- Vergessene {} - Klammern
- Fehlende '()'-Klammern: x = x − (x∗x − r) / 2∗x; (Newton Iteration)
- '==' Vergleich von floats: **float** x,y; **if** (x==y) ...
- Verwechslung von Vergleich '==' und Zuweisung '='
- Verwechslung von logischen und bitweisen Operatoren ('&' und '&&'...)
- ';' nach Klassendefinition vergessen
- Argumenttypen in .h und .cc file nicht EXAKT gleich

ziti

Fehler 2

- Das geht:
  ```
  for (unsigned int i = 0; i <= 3; i++) {...},
  ```
  aber rückwärts geht es *nicht*:
  ```
  for (unsigned int i = 3; i >= 0; i--) {...}
  ```
- ```
  for (int i=0; i<10; i++);
  {
   ...
  }
  ```
  ```

Kurioses aus dem Kurs

Alle folgenden Beispiele sind syntaktisch korrekt, jedoch falsch oder unschön.

- **if** (i == (2 || 3))..
- **for** (**int** i=1; i<=NN; i++) found[i = **false**]
- **bool** a[10]; **if** (a[i] == **true**) ... (überflüssiger Vergleich!)
- **for** (**int** i=0; j−1 <= i <= j+1; i++) {...}

Was ist das?

- Was passiert hier: `x == y == z`
- Vergleiche `a[i++]` mit `a[i]++`
- Was passiert bei `a[i=0] = 3;`
- Was ergibt `float x = '4';`? Was passiert bei `float x = "3.141";`?