

Grundlagen - C++ Einführungskurs

Manfred Kirchgessner

Institut für Technische Informatik
Universität Heidelberg

4. - 7. Oktober 2016

Kursinhalt

- ▶ Einführung
 - ➔ Was heißt eigentlich "Programmieren" ?
 - ➔ Erstes Programm: **Hallo Welt!**
 - ➔ Werkzeuge, Einführung in die Linux Konsole
 - ➔ Elementare Datentypen & Operatoren
- ▶ Strukturierte und Prozedurale Programmierung
 - ➔ Kontrollstrukturen **If ,Then, Else...**
 - ➔ Funktionen
 - ➔ Zeiger & Speicherverwaltung
 - ➔ Zusammengesetzte Datentypen

Kursinhalt

- ▶ **Objektorientierte Programmierung**
 - ➔ Klassen & Methoden
 - ➔ Vererbung
- ▶ **Weiterführende Themen**
 - ➔ Die C++ Standardbibliothek
 - ➔ Weitere nützliche Bibliotheken

Literatur

- ▶ Die "Bibel" von Bjarne Stroustrup, dem Erfinder:
Die CPP Programmiersprache (Orig.: The C++ Programming Language)
- ▶ Gutes Nachschlagewerk auch von Bjarne Stroustrup incl. c++11:
A Tour of C++
- ▶ Online Dokumentation der Standardbibliothek:
<http://www.cplusplus.com>
 - ➔ Beschreibung der Klassen string, vector, iostream, fstream, uvm.
 - ➔ Nützliche Funktionen und Typumwandlungen.
 - ➔ Mathematik Bibliothek cmath: sin(), cos(), sqrt(), round(), uvm.

- 1** Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Wieso eigentlich C++ lernen?

- ▶ **C++** ist eine sehr hardwarenahe Programmiersprache:
 - ➔ Direkte und effiziente Übersetzung möglich
 - ➔ Direkten Zugriff auf Speicher: Belegung und Freigabe
 - ➔ Direkte Unterstützung für paralleles Arbeiten auf mehreren Prozessoren
- ▶ Compiler sind frei verfügbar und in jeder Linux Distribution vorinstalliert
- ▶ Sehr flexibel und erweiterbar

Wofür eigentlich C++ lernen?

C++ bietet eine umfangreiche Standardbibliothek. Eine Vielzahl von freien Bibliotheken stellen zusätzliche Funktionalitäten zur Verfügung:

- ▶ Mathematik und Fit algorithmen (z.B. **boost** oder **root**)
- ▶ Erweiterungen für paralleles Rechnen (z.B. **Open MPI**)
- ▶ Bibliotheken zum Rechnen auf Grafikkarten (z.B. unterstützt **CUDA C++**)
- ▶ Zugriff auf Datenbanken
- ▶ Webservices und Netzerkennung
- ▶ (Handy Apps)

Wie entstand C++ ?

C++ wurde 1985 als Erweiterung der Programmiersprache **C** entwickelt mit dem Ziel Objekt Orientierung (Klassen) in die Sprache **C** einzubringen.

- ▶ Die Sprache C wurde 1970 zusammen mit dem Betriebssystem UNIX eingeführt
- ▶ Aus **C mit Klassen** entwickelte sich dann **C++** als eine Object Orientierte Programmiersprache.
- ▶ 1985 gilt mit der Veröffentlichung des Buches "The C++ Programming Language" von Bjarne Stroustrup als das Geburtsjahr von **C++** .
- ▶ 1998 wurde die Sprache erstmals standardisiert.
- ▶ Der Standard von 2003 kann als Nachbesserung des ersten Standards verstanden werden
- ▶ Der Standard **c++11** bringt erstmals deutliche Neuerungen und nützliche Erweiterungen
- ▶ Aktuell gibt es noch c++14 und c++17 die Verbesserungen zu c++11 nachreichen.

Was heißt eigentlich “Programmieren”?

Programmieren fängt bereits beim Erfassen der Problemstellung an, und umfasst nicht nur die Umsetzung der Lösung auf einem Computer, sondern auch deren Wartung

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

Wie setze ich meine Lösung in Code um?

- ▶ Oft führt mehr als eine Lösung zum Ziel.
 - ➔ Welche ist die Einfachste / Eleganteste?
 - ➔ Welches die Schnellste?
 - ➔ Welches die Flexibelste?
 - ➔ Welche kann ich an anderer Stelle wiederverwenden?

Ziel des 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 Probleme nötigen Kenntnisse eigenständig zu erarbeiten.

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!**
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Das erste Programm in C++

```
1  #include <iostream>
2
3
4  int main ()
5  {
6      // Textausgabe:
7      std::cout << "Hallo Welt!" << std::endl;
8
9      // Programm beenden
10     return 0;
11 }
```

Programm1:hello.cpp

Übersetzen und Binden

- ▶ Quelltext ist von Menschen lesbare Form
- ▶ Computer will ausführbare Form
 - ➔ Übersetzen erzeugt Objektdatei
g++ -c hello.cpp
 - ➔ Binden erzeugt ausführbare Datei
g++ -o hello hello.o
- ▶ Aufrufen des Programms
./hello
Hallo Welt!
- ▶ Änderungen immer im Quelltext, Erneutes übersetzen und Binden nötig.

Zurück zum Quelltext...

- ▶ **#include** <iostream> lädt Ein-/Ausgabe Bibliothek
- ▶ Eigentliche Funktionalität in main()
- ▶ Kommentare beginnen mit //
- ▶ Blöcke von Kommentaren über mehrere Zeilen mit
*/ * ...Kommentar – Block... * /*
- ▶ Ausgaben über std::cout
std::cout kann mit << alles "zugeworfen" werden,
was ausgegeben werden soll
- ▶ Programm endet mit Rückgabe an Betriebssystem
return 0;

Eine Frage des guten Stiles

- ▶ Die Strukturierung des Programms ist wichtig
 - ➔ Einrückungen deuten logische Struktur an
 - ➔ Leerzeichen verbessern die Lesbarkeit
 - ➔ Kommentare vermitteln Verständnis
- ▶ 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

Eine Frage des guten Stiles

Gewöhnen Sie sich einen einheitlichen Stil an. z.B.:

- ▶ Objekte immer mit Großbuchstaben:

MyVector

- ▶ Funktionen entweder:

addVectorToOtherVector()

(CamelCase)

- ▶ oder **add_vector_to_other_vector()**

- ▶ so ist es schwieriger zu lesen:

addvectortoothervector()

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!**
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Unix Programme

Gutes Unix Tutorial: <http://www.tutorialspoint.com/unix/>

▶ Verzeichnisse

pwd	Aktuelles Verzeichnis anzeigen
ls -la [<dir>]	Dateien in einem Verzeichnis anzeigen
cd <dir>	Verzeichnis wechseln (z.B. cd ../test/v1)
.	Aktuelles Verzeichnis
..	Übergeordnetes Verzeichnis
path/to/dir	Verkettung von Verzeichnissen mit /
mkdir <dir>	Neues Verzeichnis anlegen
rmdir <dir>	Verzeichnis löschen

Unix Programme

▶ Dateien

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

Weitere Programme

▶ Suchen

find <dir> -name <file> Datei suchen

grep <pattern> <file> Ausdruck in Datei <file> suchen

▶ Hilfe

man <cmd> Beschreibung zu Kommando anzeigen

info Hilfesystem anzeigen

▶ Programmierung

g++ -c <codefile> Quellcode Datei übersetzen

g++ -o <file> <obj> [<obj>] Objektdateien binden

gdb <executable> 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 **Strg + c** (auf engl. Tastaturen Ctrl) abgebrochen werden
- ▶ Ein Programm wird mit **&** am Ende der Kommandozeile unabhängig vom Eingabefenster ausgeführt, z.B. **kate file.cpp &**
- ▶ Blockiert ein Programm die Shell (das Eingabefenster), so kann es mit **Strg + z bg<ret>** in den Hintergrund geschickt werden.

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren**
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

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
 - unsigned long**
 - uint8_t, uint16_t, uint32_t** positive Zahlen mit
fester Bit-Zahl (in <stdint.h>)

- ▶ Reelle Zahlen
 - float** 1.0, .5, 3.1415, 1.6022E - 19
 - double** höhere Genauigkeit, mehr Stellen,
größerer Wertebereich
 - long double** noch höhere Genauigkeit

- ▶ Wahrheitswerte
 - bool** true, false (1,0)

- ▶ Zeichen (Buchstaben)
 - char** ä", SZ", "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 **int** i = 5;
unsigned long grosse_zahl = 123456789;
float g = 9.81, v = 3.4;
float g, v = 3.4; **Achtung!** Nur v bekommt einen Wert
- ▶ 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 `_` oder camelCase für Namen benutzen
- ▶ 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, numberOfValues, ...
 - ➔ Für Zähler und Indizes (int) reicht meist ein Buchstabe.
Oft: *i, j, k, l, m, n...*
 - ➔ Für reelle Zahlen: *a, x, y, ...*
- ▶ Einheitliche Konventionen verwenden, auch für Groß-/Kleinschreibung

Literale

Definition

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
Binär (0b..):	0b0010101001

Vorsicht!

```
1 int i = 3.5; // i ist 3
2 int i = 011; // i int 9 (Oktal-schreibweise)
3
4 float x = 6 / 5; // x ist 1 !
5 float x = 6. / 5.0; // so ist x = 1.2 korrekt!
```

Wertebereiche

► Übersicht der Wertebereiche auf einem 64 bit System

Typ	Anzahl Bits	Min	Max
bool	1	0	1
char	8	0	255
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807
long long	64	-9223372036854775808	9223372036854775807
float	32	$1.17549E - 38$	$3.40282E + 38$
double	64	$2.22507E - 308$	$1.79769E + 308$
long double	128	$3.3621E - 4932$	$1.18973E + 4932$
uint8_t	8	0	255
uint16_t	16	0	65535
...			

int8_t, uint16_t sind Teil der Standardbibliothek `<stdint.h>` und bieten Datentypen mit garantierter Größe.

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;` \Leftrightarrow `b = b + a;` etc.
- `x ? A1 : A2;` Ausdruck ergibt A1 oder A2, abhängig 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*
&& || logisches *und*, logisches *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 wird nach float gewandelt, a ist 31
```

- ▶ Zwischen elementaren Typen wird automatisch umgewandelt
- ▶ Bei Rechnungen mit Elementen unterschiedlichen Typs:
Umwandlung aller Elemente in Typ mit größtem Wertebereich
Der Compiler gibt eine Warnung aus bei impliziter Typumwandlung mit Verlust mit Compiler Flag **-Wall**

Typumwandlung

► Explizite Typumwandlung

```
1 int a,b = 10;
2 float pi = 3.14159;
3 a = b * (int) pi; // a ist 30 oder
4 a = b * static_cast<int>(pi);
5
6 char ch2;
7 ch2 = (char) ((int)'A' + 2); // ch2 ist 'C'
```

Operator Reihenfolge

```

::
. -> (..)
[..] postfix ++ -- typedef
präfix ++ -- unäre ~ ! - & * sizeof new delete
* / %
+ -
<< >>
< <= > >=
== !=
& ^ |
&& ||
? :
= += -= *= /= %= <<= >>= &= ^= |=
throw ,

```

- ▶ Falls man nicht sicher ist: besser **()** Klammern benutzen.
Compiler gibt bei Mehrdeutigkeiten eine Warnung aus:

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen**
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Programmverlauf und Anweisungen

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



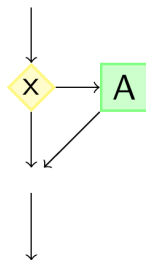
Anweisungen

- ▶ Leere Anweisung
;
- ▶ Einfache Anweisungen (was wir bisher kennen)
 - ➔ Definition von Variablen
 - ➔ Einfache Anweisungen mit Operatoren
 - ➔ Beenden von main() mit **return 0**;
- ▶ Zusammengesetzte Anweisungen (Anweisungsblock)
{ *Anweisung_A*; *Anweisung_B*; ... }
- ▶ Kontrollstrukturen (brechen Monotonie auf)
 - ➔ **Bedingungen** ermöglichen Alternativen
 - ➔ **Schleifen** ermöglichen Wiederholungen

Bedingte Anweisung

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

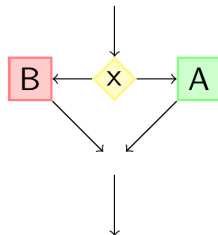
- ▶ Ausführung von Anweisung_A, wenn Ausdruck_x wahr
- ▶ Ausführung von mehreren Anweisungen mit {...}
- ▶ **WICHTIG:** C++ interpretiert jeden Wert ungleich 0 als "wahr"



Bedingte Anweisung

```
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



Bedingte Anweisung

```
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 Mißverständen der Zugehörigkeit von **else**
- ▶ Verwendung von { } schafft Klarheit

Schachtelung mit Klammern

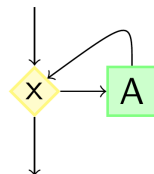
```
1  if ( Ausdruck_x ) {  
2      if ( Ausdruck_y ) {  
3          Anweisung_A1;  
4          Anweisung_A2;  
5      } else {  
6          Anweisung_B;  
7      }  
8  } else {  
9      Anweisung_C1;  
10     Anweisung_C2;  
11 }
```

- ▶ Andere Arten der Einrückung möglich
- ▶ Tabulatoren können durch Leerzeichen ersetzt werden

Einfache kopfgesteuerte Schleife

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

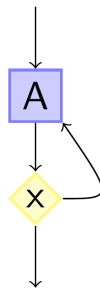
- ▶ So lange Ausdruck_x wahr ist, wird Anweisung_A ausgeführt
- ▶ Anweisung_A wird evtl. gar nicht ausgeführt
- ▶ Wenn Anweisung_A nicht dazu führt, dass Ausdruck_x irgendwann falsch wird, **führt dies zu einer Endlosschleife**



Einfache fußgesteuerte Schleife

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

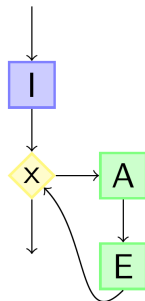
- ▶ Anweisung_A wird mindestens einmal ausgeführt
- ▶ So lange Ausdruck_x wahr ist, wird Anweisung_A ausgeführt



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 ausgeführt



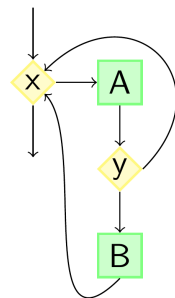
```
1  for (int i=0; i<5; i++)  
2  cout << i << " Hallo Welt!" << endl;
```

Beispiele für Flexible Schleife

```
1 float x = 0.123;
2 for ( int i=0; i<100; i++) x = x * x - 1;
3
4 int hexzahl = 0xA;
5 for ( int bit = 0b1000; bit > 0; bit >>= 1)
6     std::cout << ( hexzahl&bit)? '1' : '0');
7 std::cout << std::endl;
```

Frühzeitiges Fortsetzen

```
1 while ( Ausdruck_x ) {  
2     Anweisung_A;  
3     if ( Ausdruck_y) continue;  
4     Anweisung_B;  
5 }
```

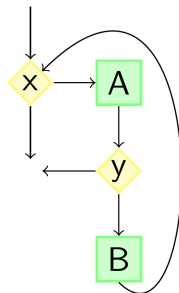


- ▶ Anweisung_A wird in jedem Schleifendurchlauf ausgeführt
- ▶ Falls Ausdruck_y wahr ist, wird Schleife *frühzeitig* fortgesetzt
- ▶ Anweisung_B wird nur ausgeführt, wenn Ausdruck_y falsch ist

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



- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen**
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Strukturen

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

- ▶ Zusammenfassung benannter Elemente beliebigen Typs
- ▶ Typdefinition vor der **int** main() Funktion
- ▶ *Definition* einer Variablen vom Typ der Struktur:
z.B. MyVector 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. MyVector start = {.0, 1.8, .0};

Aufzählungen

```
1 enum Language { Unknown, German, French };
```

- ▶ Eine Variable vom Typ 'Language' kann (nur) die Werte *Unknown, German, ...* annehmen.
- ▶ Typdefinition vor `main()`;
- ▶ Definition & Initialisierung einer Variablen diese Typs:
z.B. `Language lang = French;`
- ▶ Abfrage z.B. `if (lang == 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** array[5];
- ▶ Definition mit Initialisierung
z.B. **int** array[] = { 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. array[4] = 23;
- ▶ Erstes Element hat Index 0
- ▶ **Ausblick:** Nach **int** A[5]; kann man mit der **sizeof(A)** Funktion nach der Größe des Arrays fragen. Bei **Pointern** geht das nicht.

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, wird (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 Funktionalität
Zugänglich über **#include** `<string>` z.B. **std::string** `name = "Tom";`
 - ➔ Bibliothek `<string>` bietet viele Funktionen:
`size()`, `clear()`, `find()`, `compare()`, `substr()` ...

Typdefinitionen

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

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

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche**
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

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

- ▶ Funktionen bestehen aus *Kopf* und *Rumpf*

```
1   RückgabeTyp funktionName (Typ variable, ...) //Kopf
2   {
3       Etwas sinnvolles machen...;           //Rumpf
4       return ergebnis; //Ergebnis vom Typ Rückgabety
5   }
```

- ▶ Kopf:

- ➔ Name
- ➔ Rückgabety 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**;

Beispiel für Funktionsdefinition und Aufruf

```
1  float arithmetischesMittel (float x1, float x2){
2      float mittel;
3      mittel = (x1 + x2) / 2.0;
4      return mittel;
5  }
6
7  float geometrischesMittel (float x1, float x2){
8      return sqrt(x1*x2);
9  }
10
11 int main(void){
12     cout << arithmetischesMittel(3.0,4.0) << endl;
13     cout << geometrischesMittel(3.0,4.0) << endl;
14     return 0;
15 }
```

Beispiel Fakultät iterativ

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

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()

- ▶ Es ist möglich dem Programm beim **Aufrufen** Parameter zu übergeben
- ▶ 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) //Aequivalent
6      cout << "Anzahl parameter = " << argc << endl;
7      for(int i=0; i<argc; i++){
8          cout << "Parameter " << i << ": ";
9          cout << argv[i] << endl;
10     }
11     return 0;
12 }
```

Funktionen der Standardbibliothek: z.B. Mathematik

- ▶ die Standardbibliothek bietet bereits einige sehr nützliche Funktionen z.B. Mathematik Funktionen
- ▶ Mathematik Funktionen zugänglich durch den Befehl **#include <cmath>**
 - ➔ `sqrt()`, `pow(base,exponent)`
 - ➔ `exp()`, `log()`
 - ➔ `sin()`, `cos()`,..., `sinh()`, `cosh()`,...
 - ➔ `abs()`, `floor()`, `ceiling()`
 - ➔ Konstanten wie `M_PI` (π),...

Gültigkeitsbereiche

Definition

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

- ▶ Zwei Gültigkeitsbereiche: global und lokal
- ▶ Funktionen sind global
- ▶ Variablen sind lokal zu Funktion
- ▶ Jeder Block stellt geschlossenen Gültigkeitsbereich dar
z.B. **for (int i = 0; i<5; ++i) {..}**
i besitzt Gültigkeit nur innerhalb des **for**-Blocks

Globale Variablen

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

Aufruf mit Argumentwerten

```
1 float mult(float wert, int anzahl) {  
2     ..  
3     wert = 7.0;  
4     return wert * anzahl;  
5 }  
6 ...  
7 float z,x = 2.0;  
8 z = mult(x,3); // x bleibt hier 2.0
```

- ▶ 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 Werte 'außerhalb' aus (im Beispiel bleibt x = 2.0)

Aufruf mit Argumentreferenz

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

- ▶ Die Referenz bewirkt, dass alle Änderungen innerhalb der Funktion sich auch auf die Variablen ausserhalb der Funktion auswirken
- ▶ Argumente werden nicht in den Funktionsrumpf kopiert, bei großen Objekten ist dies viel effizienter
- ▶ Ausblick: & Operator bewirkt die Übergabe als Pointer = Adresse
- ▶ Effiziente Übergabe kann auch benutzt werden, ohne dass sich Änderungen nach außen auswirken: funkt(**const GroßesObjekt & x**)

Namensräume

- ▶ Strukturierung großer Programme über Namensräume
- ▶ Namensraum wird über Schlüsselwort **namespace** und Name definiert und leitet Block ein, der Elemente kapselt:
namespace MyDebug { **int** level = 2; }
- ▶ Elemente eines Namensraumes können über Namen und Auflösungsoperator :: erreicht werden z.B.
MyDebug::level = 5;
- ▶ Globaler Namensraum hat leeren Namen
- ▶ C++ Standardbibliothek 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 MyVector operator+( MyVector a, MyVector b)
2 {
3     MyVector c;
4     c.x = a.x + b.x;
5     c.y = a.y + b.y;
6     c.z = a.z + b.z;
7     return c;
8 }
```

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung**
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Was sind Zeiger?

- ▶ Verweis auf eine Variable
- ▶ "Die Adresse im Speicher"
- ▶ Eine Referenz ist letztlich ein Zeiger, aber anders "verpackt"
- ▶ Die dynamische Erzeugung von Referenzen ist möglich
- ▶ 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

Definition und Speicherverwaltung

- ▶ Zeiger müssen vor Zugriff initialisiert sein
- ▶ Nach Freigabe darf kein Zugriff auf Zeiger mehr erfolgen
- ▶ Zugriff auf Referenzierte Objekte über Dereferenzierungsoperator `*`
z.B. `*ptr = 42;`
- ▶ **void**-Zeiger können nicht dereferenziert werden
- ▶ Zeiger können auch auf Zusammengesetzte Typen zeigen
z.B. `MyVector v; MyVector *pv; pv = &v;`
- ▶ Zugriff auf Strukturelemente
z.B. `pv->y = 1.0; (*pv).x = 0.5;`

Merke

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

Identität von Zeigern und Feldern

- ▶ Dynamische Erzeugung von Feldern mit **new[]**
z.B. **int *ptr = new int[16];**
- ▶ Zugriff auf Feldelemente
z.B. **ptr[8] = 42; *(ptr+5) = 17;**
- ▶ Wird ein Feld mit [] erzeugt muss es auch mit []
freigegeben werden: **delete []**
z.B. **delete [] ptr;**
- ▶ Zeigerarithmetik von C++
 - ➔ Element bei Index n: ***(ptr+n)** oder **ptr[n]**
 - ➔ Zeiger auf n-tes Element: **ptr+n** oder **&ptr[n]**
- ▶ Zeiger sind syntaktisch und semantisch identisch zu
Feldern (Arrays)

Bemerkenswertes zu Zeigern und Feldern (Arrays)

```
1 void myFuncnt(int &a, int *b, int size)
2 {
3     int arr[20];
4     int *field = new int[size];
5     b[0] = 5; //oder *b = 5;
6     a = 10;
7     delete [] field;
8 }
9 ..
10 int x = 10;
11 myFuncnt(x,&x,100);
```

- ▶ arr ist lokal in der Funktion, und wird automatisch bei beenden wieder freigegeben
- ▶ field muss manuell freigegeben werden, kann aber dynamisch mit dem Parameter size erzeugt werden
- ▶ Änderungen an a und b sind in beiden Fällen außerhalb sichtbar
- ▶ Die Referenz a kann in der Funktion wie ein **int** benutzt werden
- ▶ Referenzen benutzt man zur übergabe von Einzel-Objekten, Zeiger (meistens) für Felder von Objekten

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden**
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

Grundzüge der Objektorientierung

- ▶ Bisher: Funktionen und Datensätze (weitgehend getrennt behandelt)
- ▶ Jetzt: Objekte, die "Dinge können und wechselwirken
- ▶ Zusammengesetzte Datentypen bekommen nun Funktionen die mit dem Objekt wechselwirken
- ▶ Durch Kapselung werden Daten verborgen und durch kontrollierte Zugriffe auf die Daten passieren weniger Fehler
- ▶ Vererbung ermöglicht es Code zu recyceln und leicht zu erweitern
 - ➔ Gemeinsame Programmteile nur einmal implementiert
 - ➔ Erweiterungen der Basis kommen allen Erben zugute

Klassen: Beispiel

```
1  class MyVector {
2      public:
3          MyVector();
4          MyVector(float , float , float );
5          ~MyVector();
6          void print();
7      private:
8          float x,y,z;
9  };
10
11  MyVector v(1.1,1.5,15);
12  v.print();
```

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 im namespace der Klasse (mit ::) definiert werden
- ▶ öffentliche Teile mit **public:** eingeleitet
- ▶ Private Teile mit **private:** eingeleitet
- ▶ **protected:** spielt bei Vererbung eine Rolle
- ▶ Klassen Definition wird durch Semikolon beendet.

Spezielle Methoden

- ▶ Konstruktor wird bei Erzeugung einer Instanz aufgerufen
 - ➔ Definition mit Name der Klasse *ohne* Rückgabetyt
 - ➔ Konstruktor kann überladen werden
 - ➔ Konstruktor trägt dafür Sorge, dass die Klasseninstanz vollständig initialisiert wird
- ▶ Destruktor wird bei Freigabe (**delete**) einer Instanz aufgerufen
 - ➔ Definition wie Konstruktor mit vorangestelltem ~
 - ➔ Destruktor muss dafür Sorge tragen, dass aller Speicher, der von einer Instanz belegt wurde, wieder ordnungsgemäß freigegeben wird

Zugriff auf Klassen Variablen

```
1  class MyVector{
2  public:
3      void setValues(float newX, float newY, float newZ);
4      void setX(float newX);
5      float getX();
6      ...
7  };
8      ...
9  v.setValues(1.5,3.2,0.0);
10 v.setX(5.5);
11 cout << "X ist: " << v.getX() << endl;
```

- ▶ Um auf die privaten Klassenvariablen zuzugreifen werden Methoden definiert
 - ➡ set und get Methoden kontrollieren den Zugriff. Dabei muss gar nicht im einzelnen bekannt sein, wie genau der Zugriff funktioniert
 - ➡ die Methoden der Klasse verstecken die interne Datenstruktur.

Code Management im Projekt

- ▶ In einem Software Projekt gibt es oft mehrere verschiedene Klassen mit ganz unterschiedlichen Funktionen.
- ▶ Die Deklaration und Definition von Klassen vor main() ist möglich, bei mehreren verschiedenen Klassen wird das aber schnell unübersichtlich
- ▶ Um das Software-Projekt übersichtlich zu halten, speichert man die Klassen Deklaration und die Klassen Definition in jeweils eigene Dateien.
 - ▶ Deklaration einer Klasse in den Headerfile
z.B. MyVector.h (oder .hpp)
 - ▶ Definition der Klassenmethoden in den Sourcefile
z.B. MyVector.cpp
- ▶ Das Hauptprogramm main() steht ebenfalls in einer eigenen Datei. Um die Klasse in der main() Funktion bekannt zu machen wird nur die Headerdatei mit **#include** "MyVector.hpp" eingebunden.

Beispiel: Code Management im Projekt

- ▶ Benutzt ein Programm Code aus mehreren Dateien, muss das beim Compilieren berücksichtigt werden

- ➔ Erzeugen von Objektdateien für alle Code Dateien:

```
g++ -c MyVector.cpp
```

```
g++ -c main.cpp
```

- ➔ Beim Binden werden alle Objektdateien zu einem Programm zusammengefügt:

```
g++ -o prog main.o MyVector.o
```

- ➔ Man kann auch alles in einem Schritt erledigen:

```
g++ -o prog MyVector.cpp main.cpp [-Wall]
```

Ausblick:

- ▶ In größeren Projekten mit vielen Dateien werden oft Scripte benutzt, um das Programm zusammen zu bauen:
 - ➔ **Make** benutzt **Makefiles** um Programm zu bauen
 - ➔ **CMake** ist ein Programm mit einfacherer Syntax um Makefiles generieren zu lassen
- ▶ **Makefiles** sparen Zeit und bieten hohe Flexibilität

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung**
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen

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

Beispiel: Vererbung

```
1  class MyVector{
2  public:
3      void print();
4      ...
5  };
6
7  class SmartVector : public MyVector {
8  public:
9      float lenght();
10 };
```

- ▶ SmartVector erbt von MyVector alle Methoden und Variablen.
- ▶ Statt **public**, kann auch **protected** oder **private** vererbt werden

Beispiel: Vererbung

```

1  class A { /* ...*/};
2  class B : public A    { /* ...*/};
3  class C : protected A { /* ...*/};
4  class D : private A   { /* ...*/};
5  class E : A           { /* ...*/}; // D und E identisch

```

Ist ein Element	public	protected	private
Wird es in B zu..	public	protected	private
Wird es in C zu..	protected	private	private
Wird es in D&E zu..	private	private	private

► **public** Vererbung ist die geläufigste Methode

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen**
- 12 Lektion 12: weiterführende Themen

Generische Programmierung

- ▶ Funktionen oft für verschiedene Typkombinationen benötigt:
Beispiel:
`int max(int a, int b)` funktioniert für **float** nicht richtig, 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 T,...>**
- ▶ In Definition wird generischer Typ verwendet
Schablonen sind erlaubt für Funktionen und Klassen
- ▶ Bei Verwendung (z.B. bei Funktionsaufruf) wird konkreter Typ in spitzen Klammern übergeben

```
1  template <typename T>
2  void printFancy(T value){cout << "Value is: " << value << endl;}
3
4  int a=5; float b=20.11;      std::string c="Hallo";
5  printFancy<int>(a);          printFancy<std::string>(b);
6  printFancy<float>(c);
```

- 1 Lektion 1: Einführung
- 2 Lektion 2: Hallo Welt!
- 3 Lektion 3: Werkzeuge!
- 4 Lektion 4: Elementare Datentypen & Operatoren
- 5 Lektion 5: Kontrollstrukturen
- 6 Lektion 6: Zusammengesetzte Datentypen
- 7 Lektion 7: Funktionen & Gültigkeitsbereiche
- 8 Lektion 8: Zeiger & Speicherverwaltung
- 9 Lektion 9: Klassen & Methoden
- 10 Lektion 10: Vererbung
- 11 Lektion 11: Schablonen
- 12 Lektion 12: weiterführende Themen**

Standardbibliothek - www.cplusplus.com

- ▶ einige Klassen der Standardbibliothek
 - ➔ **std::string** zur Manipulation von Zeichenketten
 - ➔ **std::bitset** für bool Operationen
 - ➔ **std::fstream** bietet Möglichkeit zum Schreiben und Lesen von Dateien
 - ➔ verschiedene sehr nützliche Containerklassen:
std::vector, std::pair, std::list ...
 - ➔ **std::thread** für parallele Ausführung
 - ➔ **std::socket** für Netzwerk-Kommunikation incl. TCP
 - ➔ **std::chrono** bietet Zugang zu Zeitfunktionen wie Systemzeit oder Stoppuhr
 - ➔ **std::unique_ptr<>** und **std::shared_ptr<>** bieten schlaue Pointer die sich selbst Verwalten wodurch Fehler vermieden werden können.
- ▶ Neuer c++11 standard kann im Compiler mit `-std=c++11` aktiviert werden:
g++ -o hello hello.cpp -std=c++11

Weitere nützliche Bibliotheken

Häufig benutzte Bibliotheken die viele zusätzliche Funktionalitäten bieten

- ▶ Boost
 - ➔ Viele Mathematik-Funktionen und Container-Klassen
- ▶ Qt
 - ➔ (Nicht nur) für Fenster-Anwendungen.
 - ➔ Hervorragende Dokumentation
 - ➔ Datenbanken, Internet-Kommunikation, Threads...
- ▶ root
 - ➔ CERN Bibliothek zum Speichern, Auswerten und Darstellen von Messungen & Daten

Häufige Fehler

- ▶ 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
- ▶ Nicht zu jedem **new** ein **delete**
- ▶ Vergessene {} - Klammern
- ▶ Verwechslung von Vergleich `==` und Zuweisung `=`
- ▶ Verwechslung von logischen und bitweisen Operatoren ("`&`" und "`&&`"...)
- ▶ ";" nach Klassendefinition vergessen
- ▶ Argumenttypen in .h und .cpp file nicht EXAKT gleich