Programmieren in **C++**

Christian Kreidl

Institut für Technische Informatik (ziti)
Universität Heidelberg

Kurs-Infos

- Montag - Freitag, 9:15 - 16:30Uhr
- Mittagspause: 12:30 - 13:30Uhr
- 1 LP/CP bei regelmäßiger, aktiver Teilnahme
- Anwesenheitsliste wird geführt

Kursinhalt

- Einführung
  - Was heißt eigenlich “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

ziti

Kursinhalt

- Objektorientiere 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.

Lektion 1

Was heißt eigentlich Programmieren?

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 vorhanden. z.B. GNU Compiler Collection
- Sehr flexibel und erweiterbar: riesige Menge an Bibliotheken verfügbar

ziti

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. **OpenMPI**)
- Bibliotheken zum Rechnen auf Grafikkarten (z.B. **CUDA**)
- Zugriff auf Datenbanken
- Webservices und Netzwerkkommunikation

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 objektorientierte 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

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

ziti

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, syntaktische Fehler
    Werden vom Übersetzer (compiler) 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' ($Z[1] \dots Z[N]$)
  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
  - ...

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.

ziti

Lektion 2

Hallo Welt!

Das erste Programm in **C++**

```cpp
#include <iostream>


int main ()
{
  // Textausgabe:
  std::cout << "Hallo Welt!" << std::endl;

  // Programm beenden
  return 0;
}
```

Programm1:hello.cpp

Übersetzen und Binden

compiling and linking

- 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
  - Eine Anweisung pro Zeile

    ```
    int i=3;float d=6.124;float z=i*d;i++; //so nicht
    ```
- 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()

Lektion 3

Werkzeuge

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>	(leeres) Verzeichnis löschen

Unix Programme

- Dateien

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

- Platzhalter

?	exakt ein beliebiges Zeichen
*****	beliebig viele (oder kein) Zeichen

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 <codefile>	**C++** 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.

Texteditor

- Nicht zu verwechseln mit Textverarbeitung: Word, LibreOffice
- speichert Text als reinen ASCII- oder UTF8-Text **ohne** Formatierungen
- Linux Console: vim, nano, joe und viele mehr
- Beispiele:
  - Linux GUI: kate, gvim, Pluma
  - Windows: Notepad, notepad++, Programmers Notepad
  - MacOS: TextEdit, BBEdit, TextWrangler

Dateien

- keine Umlaute, Leer- oder Sonderzeichen in Dateinamen verwenden
- C++ Quellcode-Dateien mit Endung .cpp oder .cc anlegen, sonst akzeptiert g++ die Datei nicht
- ausführbare Dateien haben keine Endung

GNU Compiler

erzeugen der Objektdateien

```
g++ -Wall -c hello.cpp
```

- g++ : Compiler-Programm-Datei
- -Wall : Optionsschalter der alle Compiler-Warnungen einschaltet
- -c : nur übersetzen, nicht linken
- hello.cpp : die C++ Programmcode-Textdatei

Dieser Aufruf erzeugt eine Objektdatei mit der Endung `.o`
Die Objektdatei enthält den Programmcode in Maschinensprache.

GNU Compiler

erzeugen des Programms

```
g++ -Wall -o hello hello.o
```

- g++ : Compiler-Programm-Datei
- -o : Name der zu erzeugenden Programmdatei
- hello.o : die Objektdatei

Dieser Aufruf erzeugt eine ausführbare Programmdatei durch das verbinden (linken) von Objektdateien und Bibliotheken.

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

ziti

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>)

- Reele Zahlen

$1.0, .5, 3.1415, 1.6022E-19$

float	normale Genauigkeit
double	höhere Genauigkeit, mehr Stellen, größerer Wertebereich
long double	noch höhere Genauigkeit

- Wahrheitswerte

  bool true, false (1,0)

- 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 `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 Variablen sollten immer initialisiert werden.
- 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: $0 \dots 9$ Nicht als erstes Zeichen!
- Umlaute und Sonderzeichen sind nicht erlaubt
- Sinnvolle Namen verwenden!
  - Namen sollten für die jeweilige Funktion sprechen: $kontostand, start_value, numberOfValues, \dots$
  - Für Zähler und Indizes (int) reicht meist ein Buchstabe. Oft: $i, j, k, l, m, n \dots$
  - Für reele Zahlen: $a, x, y, \dots$
- 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**
23f	**float**	1., 1e0, 42lf	**double**
1.18973e+4932L	**long 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

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

```
#include <iostream>
#include <limits>

int main()
{
  std::cout << "int: "
  << std::numeric_limits<int>::min()
  << " .. "
  << std::numeric_limits<int>::max()
  << std::endl;
  return 0;
}
```

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;` ⇔ `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, gößer
`<=` `>=`	kleiner oder gleich, größer oder gleich

- Logische Verknüpfungen

`!`	unäres logisches *nicht*
`&&` `||`	logisches *und*, logisches *oder*

- Bitweise Operatoren

`~`	unäres bitweises *nicht*
`&` `|` `^`	bitweises *und,oder,exclusiv-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* ++ -- *typecast*
präfix ++ -- *unäre* ~ ! - & * **sizeof new delete**
* / %
+ -
<< >>
< <= > >=
== !=
& ^ |
&& ||
? :
= += -= *= /= %= <<= >>= &= ^= |=
throw

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

Fluchtsequenzen

Problem: es gibt Zeichen die vom Kompiler als Anweisung interpretiert werden, und somit nicht einfach als Literal in eine Zeichenkette eingefügt werden können.
Lösung: Fluchtsequenzen (Escape-Sequenzen, ESC)

- Linksseitiger Schrägstrich \ leitet Fluchtsequenzen ein
  - '\' ' — ' als Literal
  - '\" ' — " als Literal
  - '\\ ' — \ als Literal
- Fluchtsequenz auch für sogenannte nichtdruckende Zeichen

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

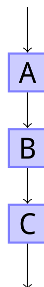
ziti

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

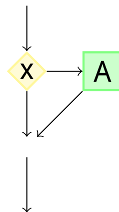
- Leere Anweisung
  ;
- Einfache Anweisungen (was wir bisher kennen)
  - Definition von Variablen
  - Einfache Anweisungen mit Operatoren
  - Beenden von main() mit **return** 0;
- Zusammengesetzte Anweisungen (Anweisungsblock)
  { Anweisung_A; Anweisung_B; ... }
- Kontrollstrukturen (brechen Monotonie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

ziti

Bedingte Anweisung

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

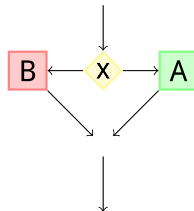
- Ausführung von Anweisung_A, wenn Ausdruck_x wahr
- Ausführung von mehreren Anweisungen mit {...}
- *WICHTIG:* C++ interpretiert jeden Wert ungleich 0 als “wahr”



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

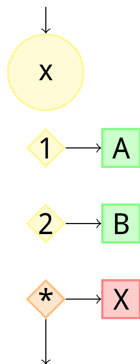
```
1  if ( Ausdruck_x) {
2    if ( Ausdruck_y ) {
3      Anweisung_A1;
4      Anweisung_A2;
5    } else {
6      Anweisung_B;
7    }
8  } else {
9    Anweisung_C1;
10   Anweisung_C2;
11 }
```

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

ziti

Mehrfachverzweigung

```
1  switch ( Variable_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

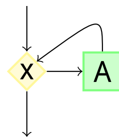
- Auswertung von `Ausdruck_x`
- Ergibt sich `Ausdruck_x` zu einer der Konstanten eines `case`-Zweiges, dann Abarbeitung der darauf folgenden Anweisungen bis zur nächsten `break`-Anweisung
- Steht zwischen zwei `case`-Zweigen kein `break`, dann wird Abarbeitung einfach fortgesetzt!
  `case`-Zweige immer 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;
```

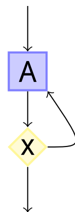
- So lange Ausdruck_x wahr ist, wird Anweisung_A ausgeführt
- Anweisung_A wird evtl. gar nicht ausgeführt
- Wenn Anweisung_A nicht dazu führt, dass Ausdruck_x irgendwann falsch wird, führt dies zu einer Endlosschleife



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 ausgeführt

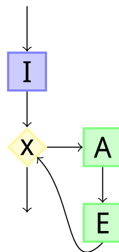


Flexible Schleife

```
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;
```

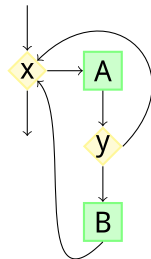


Beispiele für Flexible Schleife

```
1  float x = 0.123;
2  for ( int i=0; i<100; i++) {
3    x = x * x - 1;
4  }
5
6  int hexzahl = 0xA;
7  for ( int bit = 0b1000; bit > 0; bit >>= 1) {
8     std::cout << ( (hexzahl&bit)? '1' : '0');
9  }
10 std::cout << std::endl;
```

Frühzeitiges Fortsetzen

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

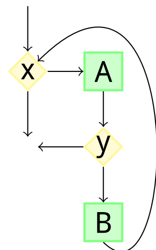


- Anweisung_A wird in jedem Schleifendurchlauf ausgeführt
- Falls Ausdruck_y wahr ist, wird Schleife *frühzeitig* fortgesetzt
- Anweisung_B wird nur ausgeführt, wenn Ausdruck_y falsch ist

ziti

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 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 len=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;`

ziti

Felder (Arrays)

- Ein Array hat eine statische Größe. Sie muss zur Compilezeit festgelegt werden!
- Das erste Element hat Index 0
- Das letzte Element eines Array's der Größe n: n-1
- Der Programmierer ist für die Einhaltung der Array-Grenzen selbst verantwortlich!
- Ein Zugriff außerhalb der Grenzen kann andere Daten verändern oder das Programm zum Absturz bringen.
- Array-Index kann auch Berechnung mit ganzahligen Typ sein: z.B.

```
array[(i*2)-1 % 5] = 23;
```

Felder (Arrays)

- Zuweisung von Arrays nicht möglich!

```
1 int a[5] = { 1, 2, 3, 4, 5 };
2 int b[5];
3 b=a; // error: invalid array assignment
```

- Es müssen die einzelne Elemente des Arrays zugewiesen werden.

```
1 for(int i=0; i<5; i++)
2 b[i]=a[i];
```

- Gleiches gilt bei Vergleichen: `if (a==b) {...}` ist falsch

ziti

Felder (Arrays) beim g++ Compiler

- "Array hat eine statische Größe. Sie muss zur Compilezeit festgelegt werden!"
- Aber dieser Code funktioniert! Warum?

```
1 int i = 0;
2 std::cin >> i;
3 int a[i];
```

- "Variable-length automatic arrays are allowed in ISO C99, and as an extension GCC accepts them in C90 mode and in C++" [1]
- Jeder Compiler hat andere Spracherweiterung. Fallstricke drohen beim Compilieren auf anderen Systemen!
- `g++ -Wall -pedantic-errors -o hello hello.cpp`

[1] Using the GNU Compiler Collection (GCC): Variable Length

Primitive Zeichenketten ('c-strings')

- Primitive Zeichenketten ("strings") sind Zeichenfelder
- Definition z.B.: `char name[] = "Bond, James Bond";`
-
```
char str[3] = "Tom";
//Error: initializer-string for array of chars is too long
```
- Um das Ende der Zeichenkette erkennen zu können, wir (automatisch) Null-Zeichen (\0) angehängt. Daher hat `char str[] = "Tom"` die Länge 4!
- Nur sehr eingeschränkte Möglichkeiten:
  kein automatisches Kopieren durch Zuweisung (s. Arrays), kein Vergleich
- Kopieren mit `strcp(Ziel, Quelle);` Vergleich mit `strcmp(s1, s2);` aus
```
#include <string.h>
```

C++ Zeichenketten

- Zugänglich über `#include <string>`
  z.B. `std::string name = "Tom";`
- automatische Speicherverwaltung
- Zuweisung und Vergleiche möglich:

```
std::string b, a = "Hallo Welt!";
b=a;
if ( b==a ) {...};
if ( b=="Hallo Welt!" ) {...};
```

- Zugriff auf einzelne Buchstaben:
  `name[i]`: keine Prüfung der Index-Position (s. Array)!
  Besser: `name.at(i)` verwenden
- Bibliothek <string> bietet viele Funtionen:
  size(), clear(), find(), compare(), substr() ...

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, divers};
7  struct Mitarbeiter
8  {
9    std::string   Name;
10   gender        g;
11   bool          IsOnHoliday[NDAY];
12 };
13 Mitarbeiter XYZ;
14 XYZ.g = female;
15 XYZ.IsOnHoliday[13] = false;
16 XYZ.Name = "Anton";
17 if (XYZ.g == male)...
```

Noch ein Beispiel

```
1  struct T_Land {
2    std::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   std::string    name,
13   T_Adresse      adr
14 };
15
16 T_Mitarbeiter Team1[100];        // Platz für 100 Mitarbeiter
17
18 Team1[1].name = "Tom";
19 Team1[1].adr.land.name = "England";
20 Team1[1].adr.PLZ = 12345;
```

Typdefinitionen

- 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

- Funktionen bestehen aus *Kopf* und *Rumpf*

```
1 RuckgabeTyp funktionName (Typ variable, ...) //Kopf
2 {
3   Etwas sinnvolles machen...;                 //Rumpf
4   return ergebnis;    //Ergebnis vom Typ Ruckgabetyp
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**;

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 }
```

ziti

Beispiel Fakultät iterativ

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

Funktionsaufruf

- Aufruf durch Name und Funktionsoperator (..)
  Übergabe von Argumenten in der Klammer
  z.B. `unsigned int 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 int factorial (unsigned int n);`
- *Definition* muss dann später erfolgen!

ziti

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
3  int main(int argc, char* argv[]) {
4    //int main (int argc, char **argv) //Aquivalent
5    std::cout << "Anzahl parameter = " << argc << std::endl;
6    for(int i=0; i<argc; i++){
7      std::cout << "Parameter " << i << ": ";
8      std::cout << argv[i] << std::endl;
9    }
10   return 0;
11 }
```

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(), fabs(), floor(), ceil()
  - 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

ziti

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)

ziti

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)`

ziti

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

- 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!

Überladen von Funktionen

```
void ausgabe(int variable){
 std::cout << "integer variable=" << variable
  << std::endl;
}
void ausgabe(float variable){
 std::cout <<   "float variable=" << variable
  << std::endl;
}
```

Operator überladen für selbstdefinierte Typen

- 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

```
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 }
```

ziti

Lektion 8

Zeiger & Speicherverwaltung

Was sind Zeiger?

- Verweis auf eine Variable
- "Die Adresse im Speicher""
- Eine Referenz ist letzlich 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

ziti

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;`
- 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!

ziti

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)

void-Pointer

- generischer Pointer der auf jeden Typ zeigen kann.
- **void**-Zeiger können nicht dereferenziert werden.
- Zugriff über cast möglich.

```
1 int i = 3;
2 float f;
3 void* voidPtr = NULL;
4 voidPtr = &f;
5 voidPtr = &i;
6 int* x = static_cast<int*>(voidPtr);
```

Referenz

- Alias einer Variable
- **Muss** initialisiert werden!
- **Keine** erneute Adresszuweisung möglich!

```
1 int i,j;
2 int &a = i; // ok
3 &a = j;     // Fehler!
```

Bemerkenswertes zu Zeigern und Feldern (Arrays)

```
1  void myFunct(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 myFunct(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

Lektion 9

Klassen & Methoden

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 recyclen und leicht zu erweitern
  - Gemeinsame Programmteile nur einmal implementiert
  - Erweiterungen der Basis kommen allen Erben zugute

ziti

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  }; // Semikolon nicht vergessen!
10
11 MyVector t;
12 MyVector v(1.1,1.5,15);
13 MyVector *p = new MyVector();
14 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.

ziti

Spezielle Methoden

- Konstruktor wird bei Erzeugung einer Instanz aufgerufen
  - Definition mit Name der Klasse *ohne* Rückgabetyp
  - 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 $\sim$
  - Destruktor muss dafür sorge tragen, dass aller Speicher, der von einer Instanz belegt wurde, wieder ordnungsgemäß freigegeben wird

Methoden definieren

```
class MyVector {
  public:
  MyVector(float , float , float );
  void print();
  private:
  float x,y,z;
};

MyVector::MyVector(float a, float b, float c)
: x(a), y(b), z(c) // Initialisierung von x,y,z
{}

void MyVector::print() {
  std::cout << "(" << x << "," << y << "," << z << ")";
}
```

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.

Lektion 10

Code Management

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

Header-Beispiel

- Die Klassendefinition muss die -deklaration auch kennen! Daher die Headerdatei in der .cpp-Datei mit `#include` einbinden:

```
1 class MyVector {
2   public:
3     void setX(float newX);
4   private:
5     float x;
6 };
```

Deklaration: MyVector.h

```
1 #include "MyVector.h"
2
3 void MyVector::setX(float newX)
4 {
5   x = newX;
6 };
```

Definition: MyVector.cpp

Code Management im Projekt

- Das Hauptprogramm main() steht ebenfalls in einer eigenen Datei. Um die Klasse in der main() verwenden zu können wird die Headerdatei dort ebenfalls eingebunden:

```
1 #include "MyVector.h"
2
3 int main()
4 {
5   MyVector v;
6   v.setX(1.23);
7 };
```

Hauptprogramm: main.cpp

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]

Build-Tools

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

Makefile Beispiel

```
1  all: geburtstag
2
3  # Einrückung des Compile-Befehls muss mit einem Tabulator-Zeichen
4  # erfolgen! keine Leerzeichen!
5  geburtstag: geburtstag.o torte.o wackeligetorte.o
6  	g++ -o geburtstag $^
7
8  geburtstag.o: geburtstag.cpp torte.h wackeligetorte.h
9  torte.o: torte.cpp torte.h
10 wackeligetorte.o: wackeligetorte.cpp wackeligetorte.h torte.h
11
12 %.o: %.cpp
13 	g++ -c $<
```

Lektion 11

Vererbung

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

```
class MyVector{
  public:
  void print();
  ...
};

class SmartVector : public MyVector {
  public:
  float lenght();
};
```

- 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**	**protected**	**private**
Wird es in D&E zu..	**private**	**private**	**private**

- **public** Vererbung ist die geläufigste Methode

Vererbung und Konstruktor mit Parametern

```
1  class MyVector{
2    public:
3      MyVector(int x); // Nur dieser Konstruktor vorhanden
4    ...
5  };
6
7  class SmartVector : public MyVector {
8    public:
9      SmartVector();
10 };
11
12 SmartVector::SmartVector() : MyVector(5) {
13   ...
14 }
```

- SmartVector erbt von MyVector auch Konstruktor. Dieser benötigt aber einen Parameter!
- Aufruf von MyVector(int) durch *initializer lists*

ziti

Vererbung und Header-Dateien

- Die Klassendeklaration von MyVector muss in SmartVector eingebunden werden:

```
#include "MyVector.h"
class SmartVector : public MyVector {
  public:
  float lenght();
};
```

Deklaration: SmartVector.h

```
class MyVector {
  public:
    void setX(float newX);
  private:
    float x;
};
```

Deklaration: MyVector.h

Einbinden der Header-Dateien

- Um beide Klassen in der main() verwenden zu können müssen beide Headerdateien dort eingebunden werden:

```
1 #include "MyVector.h"
2 #include "SmartVector.h"
3
4 int main() {
5   MyVector v;       v.setX(1.23);
6   SmartVector s;   s.setX(11.23);
7 };
```

Hauptprogramm: main.cpp

- Aber dann gibt es ein Fehler: `error: redefinition of ‘class MyVector’`
- MyVector ist direkt und nochmal über SmartVector doppelt eingebunden!

include guards

- Präprozessor-Direktiven verhindern das mehrfache Einbinden:

```
#ifndef MyVector_h
#define MyVector_h

class MyVector {
  public:
    void setX(float newX);
  private:
    float x;
};

#endif
```

Deklaration: MyVector.h

Lektion 12

Schablonen

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> void printFancy(T value)
2    { 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>(c);
6 printFancy<float>(b);
```

Lektion 13

Bibliotheken

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 nütliche Containerklassen:
    std::vector, std::pair, std::list ...
  - **std::thread** für parallele Ausführung
  - **std::socket** für Netzwerk-Komunikation 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.

Containerklassen - std::vector

- Verwendet intern ein Array: Bei Größenänderung müssen alle Daten kopiert werden $\rightarrow$ Langsam!
- Vorteil: Direkter Zugriff auf Einträge möglich

```
std::vector<int> v(5); // vector mit 5 integers erzeugen

for (int i=0; i<v.size(); i++)
  v.at(i)=i; // Werte zuweisen

v.push_back(111); // ein 11. Element am Ende hinzufügen

for (int i=0; i<v.size(); i++)
  std::cout << v.at(i) << ' ';

//Ausgabe: 0 1 2 3 4 111
```

ziti

Containerklassen - std::list

- Verwendet intern eine verkettete Liste: Schnelles hinzufügen und löschen von Elementen.
- Aber: Keinen direkten Zugriff auf n-tes Element! Nur der Reihe nach (iterator)!

```
std::list<int> l(5,0); // Liste mit 5 integers erzeugen und alle mit 0 initialisieren

for (int i=1; i<=5; i++)
  l.push_back(-1*i); // Werte am Ende der Liste anfügen

for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it)
  std::cout << *it << ' ';

//Ausgabe: 0 0 0 0 0 -1 -2 -3 -4 -5
```

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