

3 Ein kleiner Programmierkurs

- Hallo Welt
- Variablen und Typen
- Entscheidung
- Wiederholung
- Funktionen
- Funktionsschablonen
- HDNUM

Programmierungsumgebung

- Wir benutzen die Programmiersprache C++.
- Wir behandeln nur die Programmierung unter LINUX mit den GNU Compilern.
- Windows: On your own.
- Wir setzen Grundfertigkeit im Umgang mit LINUX-Rechnern voraus:
 - Shell, Kommandozeile, Starten von Programmen.
 - Dateien, Navigieren im Dateisystem.
 - Erstellen von Textdateien mit einem Editor ihrer Wahl.
- Idee des Kurses: „Lernen an Beispielen“, keine rigorose Darstellung.
- Blutige Anfänger sollten zusätzlich ein Buch lesen (siehe Literaturliste).

C++ ist eine „kompilierte“ Sprache. Um ein Programm zur Ausführung zu bringen sind folgende Schritte notwendig:

- 1 Erstelle/Ändere den Programmtext mit einem **Editor**.
- 2 Übersetze den Programmtext mit dem **C++-Übersetzer** (auch C++-Compiler) in ein Maschinenprogramm.
- 3 Führe das Programm aus. Das Programm gibt sein Ergebnis auf dem Bildschirm oder in eine Datei aus.
- 4 Interpretiere Ergebnisse. Dazu benutzen wir weitere Programme wie **gnuplot**.
- 5 Falls Ergebnis nicht korrekt, gehe nach 1!

Wichtige UNIX-Befehle

<code>ls</code>	Zeige Inhalt des aktuellen Verzeichnisses
<code>cd</code>	Wechsle ins Home-Verzeichnis
<code>cd <verzeichnis></code>	Wechsle in das angegebene <code>verzeichnis</code> (im aktuellen Verzeichnis)
<code>cd ..</code>	Gehe aus aktuellem Verzeichnis heraus
<code>mkdir <verzeichnis></code>	Erstelle neues <code>verzeichnis</code>
<code>cp <datei1> <datei2></code>	Kopiere <code>datei1</code> auf <code>datei2</code> (<code>datei2</code> kann durch Verzeichnis ersetzt werden)
<code>mv <datei1> <datei2></code>	Benenne <code>datei1</code> in <code>datei2</code> um (<code>datei2</code> kann durch Verzeichnis ersetzt werden, dann wird <code>datei1</code> dorthin verschoben)
<code>rm <datei></code>	Lösche <code>datei</code>
<code>rm -rf <verzeichnis></code>	Lösche <code>verzeichnis</code> mit allem darin

Öffne die Datei `hallowelt.cc` mit einem Editor:

```
$ gedit hallowelt.cc &
// hallowelt.cc (Dateiname als Kommentar)
#include <iostream> // notwendig zur Ausgabe

int main ()
{
    std::cout << "Numerik_0_ist_leicht:" << std::endl;
    std::cout << "1+1=" << 1+1 << std::endl;
}
```

- `iostream` ist eine sog. „Headerdatei“
- `#include` erweitert die „Basissprache“.
- `int main ()` braucht man immer: „Hier geht’s los“.
- `{ ... }` klammert Folge von Anweisungen.
- Anweisungen werden durch Semikolon abgeschlossen.

- Gebe folgende Befehle ein:

```
$ g++ -o hallowelt hallowelt.cc
$ ./hallowelt
```

- Dies sollte dann die folgende Ausgabe liefern:

```
Numerik 0 ist ganz leicht:
1+1=2
```

- Aus der Mathematik: „ $x \in M$ “. Variable x nimmt einen beliebigen Wert aus der Menge M an.
- Geht in C++ mit: `M x;`
- **Variablendefinition:** x ist eine Variable vom Typ M .
- Mit **Initialisierung:** `M x(0);`
- Ohne Initialisierung ist der Wert von Variablen der „eingebauten“ Typen nicht definiert (hängt davon ab was gerade zufällig an der Stelle im Speicher stand).

```
// zahlen.cc
#include <iostream>
int main ()
{
    unsigned int i; // uninitialisierte natürliche Zahl
    double x(3.14); // initialisierte Fließkommazahl
    float y(1.0); // einfache Genauigkeit
    short j(3); // eine "kleine" Zahl
    std::cout << "(i+x)*(y+j)=" << (i+x)*(y+j) << std::endl;
}
```

Andere Typen

- C++ kennt noch viele weitere Typen.
- Typen können nicht nur Zahlen sondern viele andere Informationen repräsentieren.
- Etwa Zeichenketten: `std::string`
- Oft muss man dazu weitere Headerdateien angeben.

```
// string.cc
#include <iostream>
#include <string>
int main ()
{
    std::string m1("Zeichen");
    std::string leer("   ");
    std::string m2("kette");
    std::cout << m1+leer+m2 << std::endl;
}
```

- Jede Variable *muss* einen Typ haben. Strenge Typbindung.

```
// eingabe.cc
#include <iostream> // header für Ein-/Ausgabe
#include <iomanip> // für setprecision
#include <cmath> // für sqrt
int main ()
{
    double x(0.0);
    std::cout << "Gebe eine Zahl ein: ";
    std::cin >> x;
    std::cout << "Wurzel(x)= "
                << std::scientific << std::showpoint
                << std::setprecision(15)
                << sqrt(x) << std::endl;
}
```

- Eingabe geht mit `std::cin >> x;`
- Standardmäßig werden nur 6 Nachkommastellen ausgegeben. Das ändert man mit `std::setprecision`. `std::scientific` sorgt für eine Ausgabe im Format `mantisseexponent` (z.B. `1.0e+04`, `std::showpoint` erzwingt die Ausgabe von Nullen am Ende).
- Für die Verwendung der Manipulatoren mit Argument muss die Headerdatei `iomanip` eingebunden werden.

- Den Wert von Variablen kann man ändern. Sonst wäre es langweilig :-)
- Dies geht mittels Zuweisung:

```
double x(3.14); // Variablendefinition mit Initialisierung
double y;      // uninitialisierte Variable
y = x;        // Weise y den Wert von x zu
x = 2.71;     // Weise x den Wert 2.71 zu, y unverändert
y = (y*3)+4;  // Werte Ausdruck rechts von = aus
              // und Weise das Resultat y zu!
```

- Block: Sequenz von Variablendefinitionen und Anweisungen in geschweiften Klammern.

```
{
    double x(3.14);
    double y;
    y = x;
}
```

- Blöcke können rekursiv geschachtelt werden.
- Eine Variable ist nur in dem Block *sichtbar*, in dem sie definiert ist sowie in allen darin enthaltenen Blöcken:

```
{
    double x(3.14);
    {
        double y;
        y = x;
    }
    y = (y*3)+4; // geht nicht, y nicht mehr sichtbar.
}
```

Whitespace

- Das Einrücken von Zeilen dient der besseren Lesbarkeit, es ist (fast) nicht vorgeschrieben, aber extrem nützlich.
- `#include`-Direktiven müssen *immer* einzeln auf einer Zeile stehen.
- Ist das folgende Programm lesbar?

```
// whitespace.cc
#include <iostream> // includes auf eigener Zeile!
#include <iomanip>
#include <cmath>
int main(){double x(0.0);
std::cout<<"Gebe_eine_lange_Zahl_ein:";std::cin >> x;
std::cout<<"Wurzel(x)=_"<<std::scientific<<std::showpoint
<<std::setprecision(16)<<sqrt(x)<< std::endl;}
```

- Aus der Mathematik kennt man eine „Zuweisung“ der folgenden Art.

Für $x \in \mathbb{R}$ setze

$$y = |x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{sonst} \end{cases}$$

- Dies realisiert man in C++ mit einer If-Anweisung:

```
double x(3.14), y;
if (x>=0)
{
    y = x;
}
else
{
    y = -x;
}
```

Varianten der If-Anweisung

- Die geschweiften Klammern kann man weglassen, wenn der Block nur eine Anweisung enthält:

```
double x(3.14), y;
if (x>=0)
    y = x;
else
    y = -x;
```

- Der `else`-Teil ist optional:

```
double x=3.14;
if (x<0)
    std::cout << "x ist negativ!" << std::endl;
```

- Weitere Vergleichsoperatoren sind `<` `<=` `==` `>=` `>` `!=`
- Beachte: `=` für Zuweisung, aber `==` für den Vergleich zweier Variablen/Werte!

- Bisher: Sequentielle Abfolge von Befehlen wie im Programm angegeben.
- Eine Möglichkeit zur Wiederholung bietet die `while`-Schleife:

```
while ( Bedingung )  
{ Schleifenkörper }
```
- Beispiel:

```
int i=0; while (i<10) { i=i+1; }
```
- Bedeutung:
 - 1 Teste Bedingung der `while`-Schleife
 - 2 Ist diese *wahr* dann führe Anweisungen im Schleifenkörper aus, sonst gehe zur ersten Anweisung nach dem Schleifenkörper.
 - 3 Gehe nach 1.
- Anweisungen im Schleifenkörper beeinflussen normalerweise den Wahrheitswert der Bedingung.
- Endlosschleife: Wert der Bedingung wird nie *falsch*.

Pendel (analytische Lösung; `while`-Schleife)

- Die Auslenkung des Pendels mit der Näherung $\sin(\phi) \approx \phi$ und $\phi(0) = \phi_0$, $\phi'(0) = 0$ lautet:

$$\phi(t) = \phi_0 \cos\left(\sqrt{\frac{g}{l}} t\right).$$

- Das folgende Programm gibt diese Lösung zu den Zeiten $t_i = i\Delta t$, $0 \leq t_i \leq T$, $i \in \mathbb{N}_0$ aus:

```
// pendelwhile.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen
int main ()
{
    double l(1.34); // Pendellänge in Meter
    double phi0(0.2); // Amplitude im Bogenmaß
    double dt(0.05); // Zeitschritt in Sekunden
    double T(30.0); // Ende in Sekunden
    double t(0.0); // Anfangswert

    while ( t<=T )
    {
        std::cout << t << " "
                    << phi0*cos(sqrt(9.81/l)*t)
                    << std::endl;
        t = t + dt;
    }
}
```

Wiederholung (for-Schleife)

- Möglichkeit der Wiederholung: `for`-Schleife:

```
for ( Anfang; Bedingung; Inkrement )
{ Schleifenkörper }
```

- Beispiel:

```
for (int i=0; i<=5; i=i+1)
{
    std::cout << "Wert von i ist " << i << std::endl;
}
```

- Enthält der Block nur eine Anweisung dann kann man die geschweiften Klammern weglassen.
- Wird die *Schleifenvariable* direkt in der `for`-Anweisung definiert, so ist sie nur innerhalb des Schleifenkörpers sichtbar.
- Die `for`-Schleife kann auch mittels einer `while`-Schleife realisiert werden.

```
// pendel.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen
int main ()
{
    double l(1.34); // Pendellänge in Meter
    double phi0(0.2); // Amplitude im Bogenmaß
    double dt(0.05); // Zeitschritt in Sekunden
    double T(30.0); // Ende in Sekunden
    for (double t=0.0; t<=T; t=t+dt)
    {
        std::cout << t << " "
                    << phi0*cos(sqrt(9.81/l)*t)
                    << std::endl;
    }
}
```

Visualisierung mit `Gnuplot`

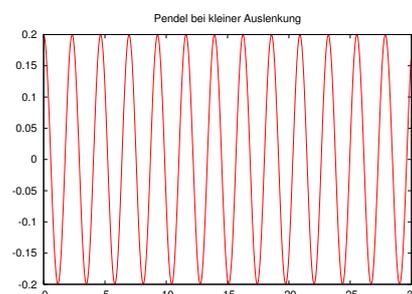
- `Gnuplot` erlaubt einfache Visualisierung von Funktionen $f : \mathbb{R} \rightarrow \mathbb{R}$ und $g : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$.
- Für $f : \mathbb{R} \rightarrow \mathbb{R}$ genügt eine zeilenweise Ausgabe von Argument und Funktionswert.
- Umlenken der Ausgabe eines Programms in eine Datei:

```
$ ./pendel > pendel.dat
```

- Starte `gnuplot`

```
$ gnuplot
```

```
gnuplot> plot "pendel.dat"with lines
```



- Ein Schleifenkörper kann selbst wieder eine Schleife enthalten, man spricht von *geschachtelten* Schleifen.

- Beispiel:

```
for (int i=1; i<=10; i=i+1)
  for (int j=1; j<=10; j=j+1)
    if (i==j)
      std::cout << "i_gleich_j:" << std::endl;
    else
      std::cout << "i_ungleich_j!" << std::endl;
```

besser:

```
for (int i=1; i<=10; i=i+1)
{
  for (int j=1; j<=10; j=j+1)
  {
    if (i==j)
      std::cout << "i_gleich_j:" << std::endl;
    else
      std::cout << "i_ungleich_j!" << std::endl;
  }
}
```

Numerische Lösung des Pendels

- Volles Modell für das Pendel aus der Einführung:

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l} \sin(\phi(t)) \quad \forall t > 0,$$

$$\phi(0) = \phi_0, \quad \frac{d\phi}{dt}(0) = u_0.$$

- Umschreiben in System erster Ordnung:

$$\frac{d\phi(t)}{dt} = u(t), \quad \frac{d^2\phi(t)}{dt^2} = \frac{du(t)}{dt} = -\frac{g}{l} \sin(\phi(t)).$$

- Eulerverfahren für $\phi^n = \phi(n\Delta t)$, $u^n = u(n\Delta t)$:

$$\begin{aligned} \phi^{n+1} &= \phi^n + \Delta t u^n & \phi^0 &= \phi_0 \\ u^{n+1} &= u^n - \Delta t (g/l) \sin(\phi^n) & u^0 &= u_0 \end{aligned}$$

```
// pendelnumerisch.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

int main ()
{
    double l(1.34); // Pendellänge in Meter
    double phi(3.0); // Anfangsamplitude in Bogenmaß
    double u(0.0); // Anfangsgeschwindigkeit
    double dt(1E-4); // Zeitschritt in Sekunden
    double T(30.0); // Ende in Sekunden
    double t(0.0); // Anfangszeit

    std::cout << t << "□" << phi << std::endl;
    while (t<T)
    {
        t = t + dt; // inkrementiere Zeit
        double phialt(phi); // merke phi
        double ualt(u); // merke u
        phi = phialt + dt*ualt; // neues phi
        u = ualt - dt*(9.81/l)*sin(phialt); // neues u
        std::cout << t << "□" << phi << std::endl;
    }
}
```

Übungsaufgaben

- 1 Schreiben Sie ein Programm, das Sie nach Ihrem Vornamen, Nachnamen und Alter fragt und anschließend etwas in der folgenden Art auf den Bildschirm schreibt:

```
Ihr Name ist Peter Bastian.
Sie sind 49 Jahre alt.
```

- 2 Verändert Sie das Programm so, dass es ausrechnet wievielen Monaten, Tagen, Stunden, Minuten und Sekunden Ihr Alter entspricht (Sie dürfen Schaltjahre vernachlässigen). Die Ausgabe sollte in etwa so aussehen:

```
Ihr Name ist Peter Bastian.
Sie sind 49 Jahre alt.
Das entspricht
588 Monaten
oder 17885 Tagen
oder 429240 Stunden
oder 25754400 Minuten
oder 1545264000 Sekunden.
```

- 3 Bei der Fibonacci Folge: 1 1 2 3 5 8 13 21 34 ... erhält man das nächste Folgenglied durch Addieren der jeweils letzten zwei Glieder der Folge:

$$Fib(0) = 0$$

$$Fib(1) = 1$$

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$

Schreiben Sie ein Programm, das vom Benutzer die Anzahl von Folgengliedern abfragt und dann entsprechend viele Elemente der Fibonacci Folge ausgibt.

Funktionsaufruf und Funktionsdefinition

- In der Mathematik gibt es das Konzept der *Funktion*.
- In C++ auch.
- Sei $f : \mathbb{R} \rightarrow \mathbb{R}$, z.B. $f(x) = x^2$.
- Wir unterscheiden den *Funktionsaufruf*

```
double x,y;
y = f(x);
```

- und die *Funktionsdefinition*. Diese sieht so aus:

```
Rückgabetyp Funktionsname ( Argumentliste )
{ Funktionsrumpf }
```

- Beispiel:

```
double f (double x)
{
    return x*x;
}
```

```
// funktion.cc
#include <iostream>

double f (double x)
{
    return x*x;
}

int main ()
{
    double x(2.0);
    std::cout << "f(" << x << ")=" << f(x) << std::endl;
}
```

- Funktionsdefinition muss vor Funktionsaufruf stehen.
- Formales Argument in der Funktionsdefinition entspricht einer Variablendefinition.
- Beim Funktionsaufruf wird das Argument (hier) *kopiert*.
- `main` ist auch nur eine Funktion.

Weiteres zum Verständnis der Funktion

- Der Name des formalen Arguments in der Funktionsdefinition ändert nichts an der Semantik der Funktion (Sofern es überall geändert wird):

```
double f (double y)
{
    return y*y;
}
```

- Das Argument wird hier kopiert, d.h.:

```
double f (double y)
{
    y = 3*y*y;
    return y;
}

int main ()
{
    double x(3.0), y;
    y = f(x); // ändert nichts an x!
}
```

- Argumentliste kann leer sein (wie in der Funktion `main`):

```
double pi ()
{
    return 3.14;
}

y = pi(); // Klammern sind erforderlich!
```

- Der Rückgabetyt `void` bedeutet „keine Rückgabe“

```
void hello ()
{
    std::cout << "hello" << std::endl;
}

hello();
```

- Mehrere Argumente werden durch Kommata getrennt:

```
double g (int i, double x)
{
    return i*x;
}

std::cout << g(2,3.14) << std::endl;
```

- Das Kopieren der Argumente einer Funktion kann verhindert werden indem man das Argument als *Referenz* definiert:

```
void Square(double x, double& y)
{
    y = x*x;
}
```

```
double x(3), y;
Square(x,y); // y hat nun den Wert 9, x ist unverändert.
```

- Statt eines Rückgabewertes kann man also auch ein (zusätzliches) Argument modifizieren.
- Insbesondere kann man so den Fall mehrerer Rückgabewerte realisieren.
- Referenzargumente bieten sich auch an, wenn Argumente „sehr groß“ sind und damit das Kopieren sehr zeitaufwendig ist.
- Der aktuelle Parameter im Aufruf *muss* dann eine Variable sein.

Pendelsimulation als Funktion

```
// pendelmitfunktion.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath> // mathematische Funktionen

void simuliere_pendel (double l, double phi, double u)
{
    double dt = 1E-4;
    double T = 30.0;
    double t = 0.0;

    std::cout << t << " " << phi << std::endl;
    while (t<T)
    {
        t = t + dt;
        double phialt(phi), ualt(u);
        phi = phialt + dt*ualt;
        u = ualt - dt*(9.81/l)*sin(phialt);
        std::cout << t << " " << phi << std::endl;
    }
}
```

```
int main ()
{
    double l(1.34); // Pendellänge in Meter
    double phi(3.0); // Anfangsamplitude in Bogenmaß
    double u(0.0); // Anfangsgeschwindigkeit
    simuliere_pendel(l,phi,u);
}
```

Funktionsschablonen

- Oft macht eine Funktion mit Argumenten verschiedenen Typs einen Sinn.
- z.B. braucht man die Funktion

```
double Square (double x)
{
    return x*x;
}
```

oft auch mit `float` oder `int`.

- Man könnte die Funktion für jeden Typ definieren, z.B.

```
float Square (float x)
{
    return x*x;
}
```

Das ist natürlich sehr umständlich. (Es darf mehrere Funktionen gleichen Namens geben, sog. *overloading*).

- In C++ gibt es mit Funktionsschablonen (engl.: *function templates*) eine Möglichkeit den Typ variabel zu lassen:

```
template<typename T>
T Square(T y)
{
    return y*y;
}
```

- T steht hier für einen beliebigen Typ.

Pendelsimulation mit Templates

```
// pendelmitfunktionstemplate.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

template<typename Number>
void simuliere_pendel (Number l, Number phi, Number u)
{
    Number dt(1E-4);
    Number T(30.0);
    Number t(0.0);
    Number g(9.81/l);

    std::cout << t << " " << phi << std::endl;
    while (t<T)
    {
        t = t + dt;
        Number phialt(phi), ualt(u);
        phi = phialt + dt*ualt;
        u = ualt - dt*g*sin(phialt);
        std::cout << t << " " << phi << std::endl;
    }
}
```

```
int main ()
{
    float l1(1.34); // Pendellänge in Meter
    float phi1(3.0); // Anfangsamplitude in Bogenmaß
    float u1(0.0); // Anfangsgeschwindigkeit
    simuliere_pendel(l1,phi1,u1);

    double l2(1.34); // Pendellänge in Meter
    double phi2(3.0); // Anfangsamplitude in Bogenmaß
    double u2(0.0); // Anfangsgeschwindigkeit
    simuliere_pendel(l2,phi2,u2);
}
```