

Ein kleiner Programmierkurs

Peter Bastian

Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 205, D-69120 Heidelberg
email: `Peter.Bastian@iwr.uni-heidelberg.de`

18. Februar 2022

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.

Workflow

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** oder **grep**.
5. Falls Ergebnis nicht korrekt, gehe nach 1!

HDNUM

- ▶ C++ kennt keine Matrizen, Vektoren, Polynome, ...
- ▶ Wir haben C++ erweitert um die **Heidelberg Educational Numerics Library**, kurz **HDNum**.
- ▶ Alle in der Vorlesung behandelten Beispiele sind dort enthalten.
- ▶ Dieser Programmierkurs ist auch Teil von HDNUM

Herunterladen von HDNUM

1. Einloggen
2. Herunterladen von HDNUM

```
git clone https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hdnum.git
```

3. Wechsle in das Verzeichnis
\$ cd hdnum/examples/progkurs
4. Anzeigen der Dateien mittels
\$ ls

Wichtige UNIX-Befehle

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

Hallo Welt !

Öffne die Datei hallohdnum.cc mit einem Editor:

```
$ gedit hallohdnum.cc
```

```
1 // hallohdnum.cc
2 #include <iostream>      // notwendig zur Ausgabe
3 #include <vector>
4 #include "hdnum.hh"      // hdnum header
5
6 int main ()
7 {
8     std::cout << "Numerik_0_ist_ganz_leicht!" << std::endl;
9     std::cout << "1+1=" << 1+1 << std::endl;
10
11     return 0;
12 }
```

- ▶ `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.

Hallo Welt laufen lassen

- ▶ Gebe folgende Befehle ein:

```
$ g++ -I../.. -o hallohdnum hallohdnum.cc  
$ ./hallohdnum
```

- ▶ Dies sollte dann die folgende Ausgabe liefern:

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


(Zahl-) Variablen

- ▶ 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);`
- ▶ Wert von Variablen der „eingebauten“ Typen ist sonst nicht definiert.

```
1 // zahlen.cc
2 #include <iostream>
3
4 int main ()
5 {
6     unsigned int i; // uninitialisierte natuerliche Zahl
7     double x(3.14); // initialisierte Fließkommazahl
8     float y(1.0);    // einfache Genauigkeit
9     short j(3);      // eine 'kleine' Zahl
10    std::cout << "(i+x)*(y+j)=" << (i+x)*(y+j) << std::endl;
11
12    return 0;
13 }
```

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.

```
1 // string.cc
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string m1("Zeichen");
8     std::string leer("   ");
9     std::string m2("kette");
10    std::cout << m1+leer+m2 << std::endl;
11
12    return 0;
13 }
```

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

Mehr Zahlen

```
1 // mehrzahlen.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <complex> // header für komplexe Zahlen
4
5 int main ()
6 {
7     std::complex<double> y(1.0,3.0);
8     std::cout << y << std::endl;
9
10    return 0;
11 }
```

- ▶ GNU Multiprecision Library <http://gmplib.org/> erlaubt Zahlen mit vielen Stellen (hier 512 Stellen zur Basis 2).
- ▶ übersetzen mit:

```
$ g++ -I../.. -o mehrzahlen mehrzahlen.cc -lgmpxx -lgmp
```
- ▶ Komplexe Zahlen sind Paare von Zahlen.
- ▶ `complex<>` ist ein Template: Baue komplexe Zahlen aus jedem anderen Zahlentyp auf (später mehr!).

Mehr Ein- und Ausgabe

```
1 // eingabe.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <iomanip>   // für setprecision
4 #include <cmath>     // für sqrt
5
6 int main ()
7 {
8     double x(0.0);
9     std::cout << "Gebe_eine_Zahl_ein: ";
10    std::cin >> x;
11    std::cout << "Wurzel(x)= "
12              << std::scientific << std::showpoint
13              << std::setprecision(15)
14              << sqrt(x) << std::endl;
15
16    return 0;
17 }
```

- ▶ Eingabe geht mit `std::cin >> x;`
- ▶ Standardmäßig werden nur 6 Nachkommastellen ausgegeben. Das ändert man mit `std::setprecision`.
- ▶ Dazu muss man die Headerdatei `iomanip` einbinden.
- ▶ Die Wurzel berechnet die Funktion `sqrt`.

Zuweisung

- ▶ 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, y unverändert
y = (y*3)+4;    // Werte Ausdruck rechts von = aus
                // und weise das Resultat y zu!
```

Blöcke

- ▶ Block: Sequenz von Variablendefinitionen und Zuweisungen 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, notwendig ist es (fast) nicht.
- ▶ `#include`-Direktiven müssen *immer* einzeln auf einer Zeile stehen.
- ▶ Ist das folgende Programm lesbar?

```
1 // whitespace.cc
2 #include <iostream> // includes auf eigener Zeile!
3 #include <iomanip>
4 #include <cmath>
5
6 int main(){
7     double x(0.0);
8     std::cout<<"Gebe_eine_lange_Zahl_ein:_";std::cin >> x;
9     std::cout<<"Wurzel(x)=_"<<std::scientific<<std::showpoint
10         <<std::setprecision(16)<<sqrt(x)<< std::endl;
11
12     return 0;
13 }
```

If-Anweisung

- 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 Objekte!

While-Schleife

- ▶ Bisher: Sequentielle Abfolge von Befehlen wie im Programm angegeben. Das ist langweilig :-)
- ▶ 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:

Pendel (analytische Lösung, while-Schleife)

```
1 // pendelwhile.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <cmath>     // mathematische Funktionen
4 int main ()
5 {
6     double l(1.34); // Pendellänge in Meter
7     double phi0(0.2); // Amplitude im Bogenmaß
8     double dt(0.05); // Zeitschritt in Sekunden
9     double T(30.0); // Ende in Sekunden
10    double t(0.0); // Anfangswert
11
12    while ( t<=T )
13    {
14        std::cout << t << "□"
15                    << phi0*cos(sqrt(9.81/l)*t)
16                    << std::endl;
17        t = t + dt;
18    }
19
20    return 0;
21 }
```

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;
}
```

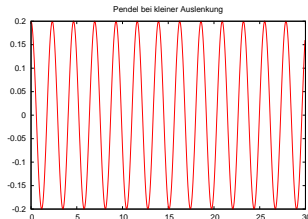
- Enthält der Block nur eine Anweisung dann kann man die geschweiften Klammern weglassen.
- Die *Schleifenvariable* ist so nur innerhalb des Schleifenkörpers sichtbar.
- Die **for**-Schleife kann auch mittels einer *while*-Schleife realisiert werden.

Pendel (analytische Lösung, **for**-Schleife)

```
1 // pendel.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <cmath>     // mathematische Funktionen
4
5 int main ()
6 {
7     double l(1.34); // Pendellänge in Meter
8     double phi0(0.2); // Amplitude im Bogenmaß
9     double dt(0.05); // Zeitschritt in Sekunden
10    double T(30.0); // Ende in Sekunden
11    for (double t=0.0; t<=T; t=t+dt)
12    {
13        std::cout << t << "␣"
14                << phi0*cos(sqrt(9.81/l)*t)
15                << std::endl;
16    }
17
18    return 0;
19 }
```

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 Programmes in eine Datei:
`$./pendel > pendel.dat`
- ▶ Starte gnuplot
`gnuplot> plot "pendel.dat" with lines`



Geschachtelte Schleifen

- ▶ 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;
```


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

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

Pendel (expliziter Euler)

```
1 // pendelnumerisch.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <cmath>     // mathematische Funktionen
4
5 int main ()
6 {
7     double l(1.34); // Pendellänge in Meter
8     double phi(3.0); // Anfangsamplitude in Bogenmaß
9     double u(0.0);   // Anfangsgeschwindigkeit
10    double dt(1E-4); // Zeitschritt in Sekunden
11    double T(30.0);  // Ende in Sekunden
12    double t(0.0);   // Anfangszeit
13
14    std::cout << t << "□" << phi << std::endl;
15    while (t<T)
16    {
17        t = t + dt; // inkrementiere Zeit
18        double phialt(phi); // merke phi
19        double ualt(u);     // merke u
20        phi = phialt + dt*ualt; // neues phi
21        u = ualt - dt*(9.81/l)*sin(phialt); // neues u
22        std::cout << t << "□" << phi << std::endl;
23    }
24
25    return 0;
26 }
```

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:

Ergebnistyp Funktionsname (Argumente)
{ Funktionsrumpf }

- ▶ Beispiel:

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

Komplettbeispiel zur Funktion

```
1 // funktion.cc
2 #include <iostream>
3
4 double f (double x)
5 {
6     return x*x;
7 }
8
9 int main ()
10 {
11     double x(2.0);
12     std::cout << "f(" << x << ")=" << f(x) << std::endl;
13
14     return 0;
15 }
```

- ▶ 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 !
}
```

Weiteres zum Verständnis der Funktion

- ▶ 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 Argument werden durch Kommata getrennt:

```
double g (int i, double x)  
{  
    return i*x;  
}  
std::cout << g(2,3.14) << std::endl;
```

Pendelsimulation als Funktion

```
1 // pendelmitfunktion.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <cmath>     // mathematische Funktionen
4
5 void simuliere_pendel (double l, double phi, double u)
6 {
7     double dt      = 1E-4;
8     double T       = 30.0;
9     double t       = 0.0;
10
11     std::cout << t << " " << phi << std::endl;
12     while (t<T)
13     {
14         t = t + dt;
15         double phialt(phi), ualt(u);
16         phi = phialt + dt*ualt;
17         u = ualt - dt*(9.81/l)*sin(phialt);
18         std::cout << t << " " << phi << std::endl;
19     }
20 }
21
22 int main ()
23 {
24     simuliere_pendel(1.34,3.0,0.0);
25
26     return 0;
```

Funktionsschablonen

- ▶ Oft macht eine Funktion mit Argumenten verschiedenen Typs einen Sinn.
- ▶ `double f (double x) {return x*x;}` macht auch mit `float`, `int` oder `mpf_class` Sinn.
- ▶ Man könnte die Funktion für jeden Typ definieren. 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 f (T y)
{
    return y*y;
}
```

- ▶ T steht hier für einen beliebigen Typ.

Pendelsimulation mit Templates I

```
1 // pendelmitfunktionstemplate.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <cmath>     // mathematische Funktionen
4
5 template<typename Number>
6 void simuliere_pendel (Number l, Number phi, Number u)
7 {
8     Number dt(1E-4);
9     Number T(30.0);
10    Number t(0.0);
11    Number g(9.81/l);
12
13    std::cout << t << "□" << phi << std::endl;
14    while (t<T)
15    {
16        t = t + dt;
17        Number phialt(phi), ualt(u);
18        phi = phialt + dt*ualt;
19        u = ualt - dt*g*sin(phialt);
20        std::cout << t << "□" << phi << std::endl;
21    }
22 }
23
```

Pendelsimulation mit Templates II

```
24 int main ()
25 {
26     float l1(1.34);    // Pendellänge in Meter
27     float phi1(3.0);   // Anfangsamplitude in Bogenmaß
28     float u1(0.0);     // Anfangsgeschwindigkeit
29     simuliere_pendel(l1,phi1,u1);
30
31     double l2(1.34);   // Pendellänge in Meter
32     double phi2(3.0);  // Anfangsamplitude in Bogenmaß
33     double u2(0.0);    // Anfangsgeschwindigkeit
34     simuliere_pendel(l2,phi2,u2);
35
36     return 0;
37 }
```

Referenzargumente

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

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

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

- ▶ Statt eines Rückgabewertes kann man 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.