Tutorium Programmieren in C(++)

Daniel Schmidt

17. Juni 2003

Zusammenfassung

Das Tutorium Programmieren in C(++) für Physiker ist eine Ergänzung zum Skript Programmieren in C der Lehrveranstaltung. Es werden alle Punkte wiederholt bzw. ergänzt, die zum grundlegenden Verständnis von C nötig sind. Es wird ein Kurzabriß über Unix–Datenaustausch, dem Graphzeichnungsprogramm Gnuplot, Adress–Zeiger, Modul–Strukturen, Make geboten. Besonderen Wert wird in den sinnvollen Einsatz bei numerischen Probellösungen physikalischer Probleme gelegt.

Inhaltsverzeichnis

1 Grunlagen zu Unix

1.1 Die wichtigsten Kommandos

Hier die ersten 10 von noch 10000 weiteren Kommandos.

Kommandoname	Funktion
cd *dir* cd .. cd	gehe zu Verzeichnis *dir* gehe ein Verzeichnis höher (ohne Parameter) gehe ins Heimverzeichnis
pwd	akutelles Verzeichnis anzeigen - pwd=„print working directory“
ls ls -la	Inhalts eines Verzeichnisses anzeigen Inhalt des akutelles Verzeichnisses detailiert anzeigen
mkdir *dir* rm *datei* rm -r *dir* rm *	Neues Verzeichnis erstellen Datei löschen Verzeichnis *dir* löschen alle Dateien im akutellen Verzeichnis löschen **niemals** *.* bei rm unter Unix verwenden!
mv *oldname newname* mv *datei neuesdir/*	Datei umbennen Datei verschieben (mv = „move“)
cat *datei*	Anzeige der Datei „datei“ auf dem Bildschirm
more *datei*	seitenweises Anzeigen der Datei *datei*
cp *original kopie* cp *vers3.c vers4.c*	Datei *original* wird auf Datei *kopie* kopiert neue Version erzeugen

1.2 Umleiten von Daten

Hier ist ein Beispiel von Daten umleiten unter Unix gegeben.

```
1 ls -la | more
2 cat datei.txt |more
3 c-progamm |more
```

Die Ausgabe der Kommandos werden seitenweise angezeigt.

1.3 Verzeichnisnomenklatur

~	Heimatverzeichnis
/	Stammverzeichnis, höchstes Verzeichnis
.	aktuelles Verzeichnis
/home/u/user	absoluter Pfad (beginnt mit /)
../user2/datei.txt	relativer Pfad

1.4 Dateirechte

Jede Datei hat

- Eigentümer
- Gruppenzughörigkeit
- Zugriffsrechte für Eigentümer
- Zugriffsrechte für Gruppe
- Zugriffsrechte für alle Benutzer

Jedes Zugriffsrecht kann **lesen** (r=read) , **schreiben** (w=write) oder **ausführen** x=eXecute beinhalten. Wichtig für das Programmieren ist, daß erzeugte Binärdateien zumindest ein **x** enthalten muß, weil es sonst nicht ausgeführt werden darf. Mit `chmod 755 programm` stellen Sie auf jeden Fall sicher, daß jeder Benutzer im System das Programm ausführen darf. Vergessen Sie auch nicht, das Verzeichnis, in dem das Programm liegt, auch lesbar für andere Benutzer zu machen.

2 Grafische Darstellung unter UNIX: GNUPLOT

Im Laufe dieser Veranstaltung werden wir des öfteren Daten berechnen, die viel besser verstanden werden können, wenn sie grafisch dargestellt werden. Diese Aufgabe erfüllt das Programm **gnuplot**, das auf allen Unix-Plattformen zur Verfügung gestellt wird und - wie der GNU-Compiler - ebenfalls sehr weit verbreitet ist. Im folgengenden werden beispielhaft Möglichkeiten zum Plotten gegeben.

2.1 2D-Plot

```
plot sin(x)
plot x**2
plot sin(x),cos(x) plot 'mess.dat'
plot 'mess.dat' using 1:2, x**2
plot 'mess.dat' with lines plot [-2*pi:2*pi] sin(x)
```

2.2 3D-Plot

```
splot sin(x*y)
splot 'mess2.dat' splot 'mess3.dat' using 3:2:1
```

2.3 Einstellungen von Gnuplot

Der Befehl `set` setzt Parameter, die die Einstellungen im Gnuplot bestimmen. Mit `reset` werden alle Einstellungen wieder zurückgesetzt.

```
set xrange [-2*pi:2*pi]
set xrange [-pi:pi]
set zrange [-1:1]
set iosample
```

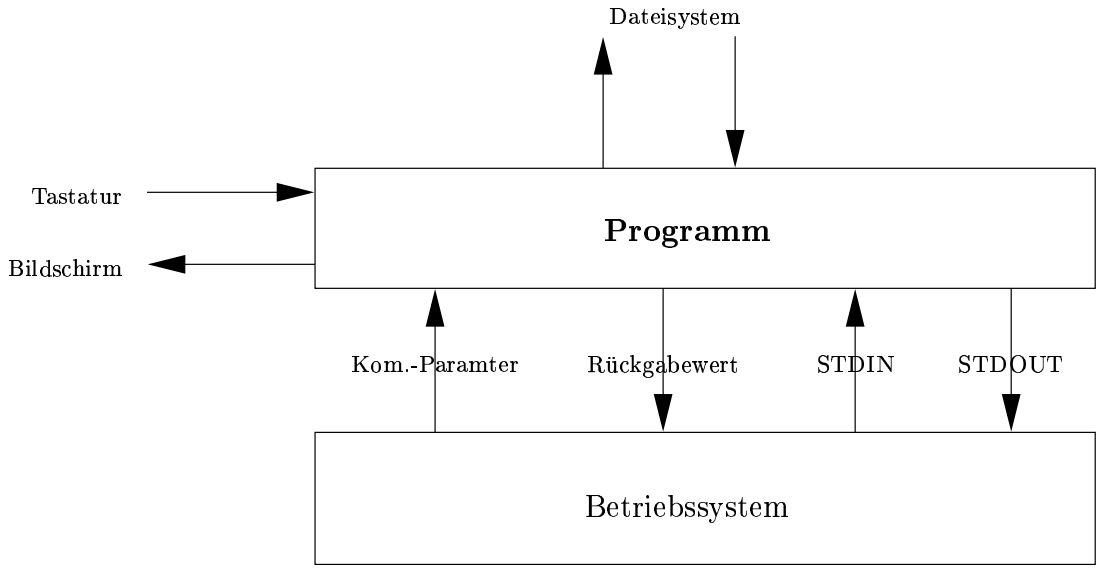
2.4 Ausgabe in druckbare Dateien

```
1 set terminal postscript portrait color
2 set output "grafik.ps"
```

2.5 Befehlsstapel in Datei

Damit Sie nicht alle Gnuplot-Befehle jedesmal wieder eingeben müssen, erzeugen Sie sich mit einen Editor eine Datei `grafik.gpl`, die zeilenweise alle Gnuplot-Kommandos enthält. Geben Sie einfach beim Aufruf von der Kommandozeile als Parameter den Dateinamen der Befehlsstapel–Datei an, also `gnuplot grafik.gpl`.

3 Datenströme



Programm können erkennen, welcher Datenstrom benutzt wird, es wird eine Adresse im Hauptspeicher benutzt

Kommandozeilenparameter werden durch die Shell durch Leerzeichen getrennt, der nullte ist das Programm selbst

Rückgabewert Wird in `main`-Funktion durch `return` übergeben und kann vom Betriebssystem verarbeitet werden. Die `main`-Funktion sollte vom Typ `int` sein.

Standard-Ströhme Beispiele:

```
ls -l | more
date | more
ls | sort | more
```

Standardströhme umleiten erfolgt mit < oder >. Sei ein C-Programm `anzeigen`, welches den Inhalt eines Stroms anzeigt. Der Strom kann direkt von einer Datei oder von einem STD-Strom genommen werden.

```
anzeigen <textdatei
anzeigen <textdatei >ausgabe.txt
```

Bildschirm und Tastatur Der Befehl `printf` gibt Daten aus, hingegen gibt `scanf` Daten aus. Ohne weiteren Einstellungen ist `STDIN` die Tastatur und `STDOUT` der Bildschirm. Die Befehle lauten schematisch

```
printf(stdout,"Textformat",Variablen);
...
scanf(stdin,"Textformat",Variablen);
```

4 Grundlege Operationen

4.1 Einfache Datentypen

Man unterscheidet Integer- und Fließkomma–Datentypen.

1. Integer–Werte sind ganze Zahlen
2. Fließkomma haben das Format

$$\pm \quad X.XXX \quad \cdots \quad e \quad \pm \quad XXX$$

Es können nicht einmal alle rationalen Zahlen dargestellt werden

4.2 Typ-Erzwingung

Werden zwei unterschiedliche einfache Typen durch einen Operator verbunden, ist der Zieloperator des Typ, welchen auch der komplexere der beiden Operator hat.

```
double + real => double
double + int  => double
```

Mit Hilfe des **Cast**-Operators kann bei Ausführen der operierens der Zieldatentyp bestimmt werden. Das Prinzip am Beispiel

```
double d;
int x; int y;
d=(double) x * y;
```

So ist der Divisionsoperator `/` ohne Typ-Erzwingung nur für integeres Rechnes gedacht.

4.3 Grundlegende mathematische Operationen

Die meist benutzten mathematischen Funktionen befinden sich in der Bibliothek `math.h`. Um die Eigenschaften zu benutzen, fügt man im Kopf der C-Datei die Zeile `#include <math.h>` ein. Beim Kompilieren wird zusätzlich die Option `-lm` angefügt, also beispielsweise `gcc rechnen.c -o rechnen -lm`.

Grundlegende Funktionen sind in der folgenden Liste

```
sin,cos,tan,asin,acos,atan,sinh,
cosh,tanh,asinh,acosh,atanh,exp,log,log10
```

Oft benutzte Funktionen sind im folgenden aufgeführt.

`atan2(x,y)`	$\mathrm{atan}(x/y)$
`expm1(x)`	$e^x - 1$
`log1p`	$\log(1+x)$
`pow(x,y)`	Die Potenz mit Basis und Exponent.
`sqrt(x)`	Wurzel von x
`csqrt(x)`	$\sqrt[3]{x}$
`hypot(x,y)`	$\sqrt{x^2+y^2}$
`z=frexp(x,&n)`	$x = z2^n$ mit $0.5 \leq z < 1$
`ldexp(x,n)`	$x2^n$
`z=modf(x,&n)`	$x = z + n$ mit $0 \leq \lvert z\rvert < 1$
`ceil(x)`	Kleinste ganze Zahl größer als x
`floor(x)`	größte ganze kleiner als x
`rint(x)`	rundet auf die nächste ganze Zahl ab
`mod(x,y)`	Rest von $\frac{x}{y}$, wie bei Ganzzahldivision
`copysign(x,y)`	gibt x zurück, welches das Vorzeichen von y hat
`sign(x)`	gibt das Vorzeichen von x zurück
`j0(x),j1(x),jn(n,x)`	Besselfunktion n-ter Ordnung
`y0(x),y1(x),yn(n,x)`	Lösungen der DGL $x^2y'' + xy' - (x^2 - n^2)y = 0$
`erf(x)`	Fehlerfunktion $\frac{2}{\pi}\int_0^x \exp(-z^2)\mathrm{d}z$
`erfc(x)`	1−Fehlerfunktion
`lgamma(x)`	Logarithmus der Gammafunktion $\log\int_0^\infty t^{x-1}e^{-t}\mathrm{d}t$
`lgamma(n+1)`	Logarithmus von n-Fakultät $\log(n!)$

Konstanten Bei der Verwendung der Mathebibliothek können auch Konstanten benutzt werden.

Inhalt der Variablen prüfen Die folgende Abfolge ändert den Wert der Variable `x`.

M_E	e
M_PI	π
M_PI_2	$\pi/2$

```
double x;
if (x=M_PI) {
  printf("x ist (nun) pi\n");
}
```

Mit dem Operator == wird nur verglichen, nicht geändert, also

```
double x;
if (x==M_PI) {
  printf("x ist pi\n");
}
```

4.4 Rangfolge der Operatoren

Es sollte auf jeden Fall beim Programmieren die Reihenfolge der Operatoren beachtet werden. Lieber eine Klammer zu viel setzen als sich Fehler einzubauen. Die Tabelle 1 auf Seite 9 zeigt die Prioritäten der einzelnen Operatoren.

5 Kompilieren

Beim Prozeß des **Kompilierens** wird aus der Quelldatei eine Binärdatei erzeugt. Es können mehrere C-Quellen getrennt kompiliert werden, Das Zusammenfügen der Binärmodule nennt man **Linken** (=Verknüpfen).

Hierfür sind grundlegende Kompiler–Befehle nötig.

Makros definieren Legt ein Makro fest. Die Syntax ist `#define makro` zum setzen, oder `#define makro wert` zum Setzen mit einem Wert `wert`. Taucht im nachfolgenden C–Quelltext der Makroname auf, so wird an der entsprechenden Stelle der Wert des Makros eingefügt. D.h. alle Makrowerte sind in der Binärdatei bereits gesetzt.

Die Zuordnung von Variablen oder Makrod erfolgt zu unterschiedlichen Zeitpunkten:

beim Kompilieren Überall in der C–Quelle wird `x` vom Kompiler durch `3` ersetzt.

```
#define x 3
printf("Wert %f\n",x);
```

zur Laufzeit Hierbei erhält `x` seinen Wert erst wenn das Programm läuft.

```
int x;
x=3;
printf("Wert %f\n",x);
```

Bezeichnung	**Operatorsymbol**	**Priorität**	**Bewertungs-reihenfolge**
Klammern	`( ) [ ]`	15	von links nach rechts
Komponentenauswahl	`. ->`	15	von links nach rechts
Unäre Operatoren			
Cast	`(Datentyp)`	14	von rechts nach links
Größe	`sizeof`	14	von rechts nach links
Adreoperator	`&`	14	von rechts nach links
Verweis	`*`	14	von rechts nach links
Negation			
arithmetisch	`-`	14	von rechts nach links
logisch	`!`	14	von rechts nach links
bitlogisch	`| &`	14	von rechts nach links
Inkrement	`++`	14	von rechts nach links
Dekrement	`--`	14	von rechts nach links
Binäre und ternre Operatoren			
arithmetisch	`* / %`	13	von links nach rechts
	`+ -`	12	von links nach rechts
Verschieben	`<< >>`	11	von links nach rechts
Vergleich	`> >= < <=`	10	von links nach rechts
	`== !=`	9	von links nach rechts
Bitoperatoren	`&`	8	von links nach rechts
	`^`	7	von links nach rechts
	`|`	6	von links nach rechts
logisch	`&&`	5	von links nach rechts
	`||`	4	von links nach rechts
Bedinungen (ternär)	`?:`	3	von rechts nach links
Zuweisung	`= += -= *= /= %= >>= <<= &= |=`	2	von rechts nach links
Sequenz	`,`	1	von links nach rechts

Tabelle 1: Die Rangfolge der Operatoren in C

Bedingtes Kompilieren Es gibt auch Wenn-Dann-Abfragen. Die Syntax ist im Beispiel gezeigt:

```
#define makroname "Text"
#ifdef makroname
  printf("Makro %s ist definiert\n",makroname);
#else
  printf("Makro %s ist undefiniert\n",makroname);
#endif
```

Man kann auch beim Aufrufen des Kompiler von der Kommandozeile Makro setzen. Die Syntax hierfür ist:

```
gcc quelle.c binaer -Dmakroname=wert
```

Einfügen Mit dem Befehl `#include quelle.datei` werden zusätzliche Quelle-Dateien eingefügt. An der Stelle des `include`-Befehls wird einfach der Inhalt der einzufügenden Datei an den Kompiler zur Bearbeitung übergeben. Wird die einzufügende Datei in Anführungsstrichen angegeben, also

```
#include "einfuegen.h"
```

gilt der absolute oder relative Pfad. In diesem Fall werden meist selbstgeschriebene Quelldateien verwendet.

Mit spitzen Klammern werden Standardbibliotheken eingebunden, also

```
#include <math.h>
#include <stdlib.h>
```

Die Dateiendung **.h** steht für Kopf-Dateien (englisch header), die aber die C–Syntax haben.

6 Schleifen

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  //Beispiel fuer Kommandozeilenparameter
5  //gcc schleifen.c -o schleifen -Wall -g
6  int main(void) {
7          //Variablen
8          //int auswahl;
9          int i;
10         double Te;
11         double t;
12         double t_b;
13         double t_u;
14
15                 //while-Schleife
```

```
16                  t=0;
17                  while (t<10) {
18                          printf("Zeit: %f\n",t);
19                          t+=0.1;
20                  }
21
22                  //while-Schleife mit Abbruch-Befehlen
23                  t=3;
24                  while (t>0) {
25                          printf("Zeit: %f\n",t);
26                          if (t==3.2) {
27                                  printf("Singuluaritaet bei %f erreicht\n",t);
28                                  continue;
29                          }
30                          if (t==6.2) {
31                                  printf("Periode erreicht\n");
32                                  break;
33                          }
34                          printf("tan(%f)=%f\n",t,tan(t));
35                          t+=0.1;
36                  }
37
38                  //do-Schleife
39                  do {
40                          if (t==3.2) {
41                                  printf("Singuluaritaet bei %f erreicht\n",t);
42                                  continue;
43                          }
44                          printf("tan(%f)=%f\n",t,tan(t));
45                          t+=0.1;
46                  } while (t<6.2);
47
48
49                  //integer hochzaehlen
50                  for (i=0;i<10;i++) {
51                          printf("%d. Buchstabe des Alphabets ist %c\n",i,i+55);
52                  }
53
54
55
56                  //Fliesskommazahl hochzaehlen
57                  for (Te=273;Te<290;Te+=0.25) {
58                          printf("Temperatur: %f\n",Te);
59                  }
60
61
62                  //als for
63                  for (t=3;t<6.2; t+=0.1) {
64                          if (t==3.2) {
65                                  printf("Singuluaritaet bei %f erreicht\n",t);
66                                  continue;
67                          }
68                  }
```

```
69
70                  for (
71                     t_b=0, t_u=1;
72                     t_b<1;
73                     t_b+=0.1, t_u-=0.1) {
74                          printf("unbesetzt=%f besetzt=%f\n",t_u,t_b);
75                  }
76          //
77          return 0;
78  }
```

7 Ein– und Ausgabe

Daten einlesen Die Eingabe von Daten erfolgt allgemein mit

```
1  fscanf(DATENSTROM,FORMAT,&VARIABLE1,&VARIABLE2,...);
```

Für die Tastatur wählt man `stdin`. Das Format ist eine Zeichenkette, die angibt wieviele und mit welchen Typ eine oder mehrere Variablen eingelesen werden. Vor jeder Variable muß ein kaufmännisches Und–Zeichen (`&`) stehen. Nur so können die Werte der Variablen durch `fscanf` geändert werden. Zum Verständnis hiefür sind die Kapitel über Funktionen und Zeiger notwendig.

Daten ausgeben Die Ausgabe von Daten erfolgt mit

```
1  fprintf(DATENSTROM,FORMAT,VARIABLE1,VARIABLE2,...);
```

Das Format hat leichte Unterschiede zu `fscanf`. Das standartisierte Datenstrom, was ohne weitere Angaben der Monitor ist, lautet `stdout`.

8 Funktionen

Es ist immer sinnvoll, das C–Programm in Blöcke zu unterteilen. Jeder Block erfüllt einen Aufgabenbereich. Wird ein Block öfter wiederholt, eignet sich hierfür eine Funktion. Sie kann mehrmals mit verschiedenen Werten aufgerufen werden.

```
1  int d; //normale Variable, kein Parameter
2  int quadrat(int x); //die sogenannte Deklaration einer Funktion
3  ...
4  int quadrat(int x) {
5          return x*x;  //Definition
6  }
```

- Der Typ der Funktion sollte ein einfacher Datentyp sein. Es gibt ausgewählte C–Kompilatoren, welche auch komplexe Datentypen akzeptieren.
- Der Rückgabewert muß mit dem Typ der Funktion übereinstimmen.

Hier nun das Beispiel der Summenbildung aller Werte zwischen zwei natürlichen Zahlen a und b:

```
int summe(int a, int b) {
        int i;
        int s;
        s=0;
        for (i=a;i<b=b;i++) {
                s=s+i;
        }
        return s;
}
```

Die Werte der Variablen werden kopiert, und die Funktion arbeitet mit der Kopie der Werte. Die ursprünglichen Werte bleiben unverändert.

```
int x;
int y;
y=quadrat(x); //x bleibt unveraendert, y wird geaendert
```

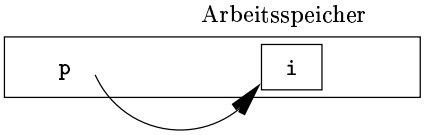
Aus diesem Grund wird bei `scanf` ein `&`–Zeichen vor die zu verändernden Variablen gesetzt. Zum Verständ lese man das Kapitel über Zeiger.

9 Wichtige Datentypen

9.1 Zeiger

Jede Variable hat eine Adresse im Speicher. Also kann man eine integere Variable für den Zugriff verwenden. Man bezeichnet diese Variable als Zeiger (=Pointer). Der Zeiger weist auf die Speicheradresse einer Variable. Es wird eine Va-

```
int i;
int *p;
p=&i;
```



riable `i` erzeugt, der Inhalt des Zeigers `p` ist die Variable `i`.
Andere Programmiersprachen arbeiten ohne Zeiger. Ein C–Programmierer kommt ohne Zeiger nicht aus. Der direkte Zugriff auf den Arbeitsspeicher macht daher C–Programm schnell. Es können mehrere Programmteile auf eine ein und die selbste Variable zugreifen ohne eine Kopie zu erstellen. Jedoch haben Fehlzugriffe schwere Folgen: Das Programm kann abstürzen. Deswegen sollte man viel Zeit in die Zeiger–Arithmetrik legen. Das entspricht der Rechnzeit zur Vermeidung von Fehlzugriffen, welche Programmieresprachen ohne Zeiger aufbringen müssen.

9.2 Zeichenketten

Hier ein Minimalwissenssatz zu Zeichenketten.

Anhängen	`strcat(STAMM,ANHANG)`
Länge ermitteln	`LAENGE=strlen(ZEICHENKETTE)`
Vergleichen	`strcmp(ZK1,ZK2)` liefert Null, wenn `ZK1` und `ZK2` gleich sind.
Einzelnes Zeichen vergleichen	`if (ZK[INDEX]=='ZEICHEN') ...`
Integer zu Zeichenkette	`sprintf(z,"%d",i);`
Zeichenkette zu Integer oder Fließkomma	`l=atoi(z)` bzw. `d=atof(z)`

9.3 Doppelzeiger

Das Prinzip von Zeigern sollte verstanden werden, hier wird aber eine weiterführende Hilfe geboten, welche eine häufige Anwendung in der Algebra hat.

Wie stellt man Matrizen dar? Die 3×3–Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{1}$$

kann statisch mittels `double A[3][3]` deklariert werden. Was ist aber, wenn der Benutzer des Programms mehr als 3 Spalten verwende möchte. Bei größeren Datenmengen ist ungenutzter statischer Speicher Verschwedung. Hierführ eignet sich daher ein dynamisches Variablensystem, welches oft benutzt wird. Es sollte richtig verstanden werden.

Vektor darstellen Das 1–dimensionale Feld `double a[3]` ist statisch. Hingegen kann der Zeiger `double *a` mit

```
a = (double*) malloc( sizeof(double) * n )
```

ein Vektor mit n Komponenten reserviert werden. Die Speicherung sieht als wie in Abbildung 1 aus. Der Zugrif auf das i–te Elemente des Vektors ist genauso

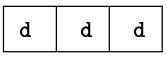


Abbildung 1: Ein 1–dimensionales Feld von `double`–Variablen (kurz: `d`)

wie beim statischen Feld, nämlich

```
1 printf("%f\n",a[i]);
```

Nach Abschluß der Benutzung von `a` wird der Speicher wieder mit `free(a)` freigegeben. Noch einmal ein Beispiel:

```
1 #include <stdlib.h>
2 #include <stdio.h>
```

```
3  int main(void) {
4          int i;
5          double a_st[3];
6          double *a_dy;
7          a_dy = (double*) malloc( sizeof(double) * 3 );
8          a_st[0]=1.0; a_st[1]=2.0; a_st[2]=3.0;
9          a_dy[0]=4.0; a_dy[1]=5.0; a_dy[2]=6.0;
10         for (i=0; i<3; i++) {
11                 printf("%f\n",a_st[i]);
12                 printf("%f\n",a_dy[i];
13         }
14         free(a_dy);
15         return 0;
16 }
```

Matrix darstellen Der Datentyp `double **b` ist ein **Doppelzeiger**. Das Äußere ist ein 1–dimensionales Feld von Zeigern, welches mit

```
1  b = (double*)malloc( sizeof(double*) * m );
```

reserviert wird und m Zeiger erhält. Jedes Elemente des Feldes zeigt wieder auf ein `double *b`. Die Abbildung 2 gibt die Struktur von

```
double ** B;
```

wieder. Es existieren also innere `double`–Vektoren und äußere `double*`–Zeiger–

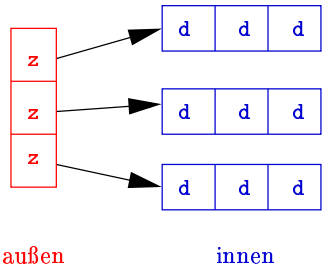


Abbildung 2: Ein 2–dimensionales Feld von `double`–Variablen (kurz: `d`) und `double*`–Zeigern (kurz: `z`)

Vektoren.

Zugriff auf dynamische Matrizen Der Zugriff des Elements B_{ij} der Matrix B erfolgt wie im statischen Fall, namlich mit

```
printf("%f\n",B[i][j]);
```

Reservierung von dynamischen Matrizen Die Reservierung des gesamten Feldes `b` einer Matrix $B^{n\times m}$ wird wie folgt gemacht:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  int main(void) {
```

```
4           int i;
5           //Reservieren
6           double **b = (double**) malloc(sizeof(double*) * n );
7           for (i=0;i<n;i++) {
8                   b[i] = (double*) malloc( sizeof(double) * m );
9           }
10          //Zugriff und Benutzen
11          for (i=0;i<n;i++) {
12                  for (j=0;j<m;j++) {
13                          printf("%f\t",b[i][j]);
14                  }
15                  printf("\n");
16          }
17          //Freigabe
18          for (i=0;i<n;i++) {
19                  free(b[i]);
20          }
21          free(b);
22          return 0;
23  }
```

Komplere Matrizen verwenden Natürlich kann mit

```
1   typedef struct {
2           double re;
3           double im;
4   } tcomplex;
5   typedef double tmat[3][3];
6   tcomplex **c;
7   tmat **d;
```

komplexe bzw. Matrizen mit Matrizen–Elementen erzeugen. Es wird einfach an allen Stellen des C–Codes der neue Datentyp statt `double` verwendet.

10 Module

- Quelltext auf mehreer Dateien verteilen
- Hauptdatei fuegt Nebenmodule ein
- Nebenmodule werden zum Hauptbinaermodul angefuegt

10.1 Nebenmodul

Dem Nebenmodul liegen 2 Dateien zugrunde:

H-Datei
- alle Deklarationen, die vom Hauptmodul verwendet werden
- gleiche Syntax wie C-Quelle
- Hauptmodul braucht keine H-Datei

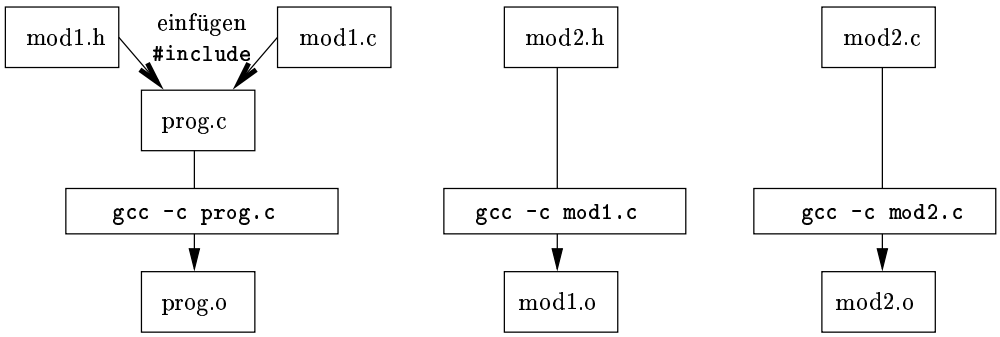
```
int var;
int func(int var);
```

C-Datei

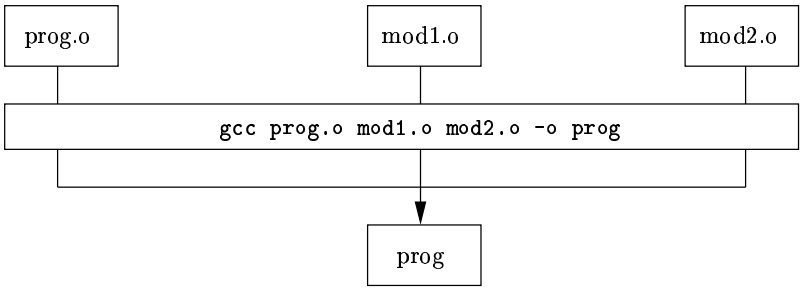
- enthaelt alle Definitionen und Befehle und interne Deklarationen
- es wird die dazugehoerige H-Datei eingefuegt

```
var=3;
int func(int var) {
  return var;
}
```

10.2 Kompilieren



10.3 Verknüpfen



10.4 Nebenmodul

Hier das Beispiel eines Nebenmoduls `trigono`. Die Befehle zum Kompilieren lauten

```
1 gcc -c trigono.c -Wall
2 gcc -c triometrie.c -Wall
3 gcc trigono.o triometrie.o -lm -o triometrie
```

```
//trigono.h
typedef struct {
        double re;
        double im;
} tcomplex;
double tangens(double x);
double cotagens(double x);
tcomplex *conj_complex(
        tcomplex c,
           tcomplex *cc
   );
```

```
//trigono.c
#include <math.h>
#ifndef _TRIGONO_H_
#define _TRIGONO_H_
double tangens(double x) {
        return sin(x)/cos(x);
}
double cotagens(double x) {
        return cos(x)/sin(x);
}
tcomplex *conj_complex(
                tcomplex c,
                tcomplex *cc
        ) {
        cc->re=c.re;
        cc->im=c.im;
        return cc;
}
#endif
```

```
//triometrie.c
#include "trigono.h"
int main(void) {
        double x=0.5;
        tcomplex a,b;
        c.re=0.5;
        c.im=1.0;
        printf("tan(%f)=%f cot(%f)=%f\",x,tangens(x),x,cotangens(x));
        conj_complex(a,&b);
        printf("Im(complex_conj(%f))=%f\n",a.im,b.im);
        return 0;
}
```

11 Make

- wie eine Stapeldatei fuer Kompilierbefehle von der Kommandozeile aus
- besitzt Schalter zur Auswahl gewuenschter Kompilieraktion

Was ist zu tun:

1. `Makefile` erstellen
2. `make` ausfuehren

Beispiel:

```
mod1:
        gcc -c mod1.c -Wall
mod2:
        gcc -c mod2.c -Wall
prog:
        gcc -c prog.c -Wall
all:        mod1 mod2 prog
        gcc mod1.o mod2.o prog.o -o prog
```

Beachten Sie, daß hinter den Zielen ein Tabulatorzeichen steht. Außer dem dem Kompilierbefehl können Sie beliebig weitere Unix–Kommandos ausführen lassen. Es muß nur ein Tabulatorzeichen vor jedem Befehl stehen.

12 Objektorientiertes Programmieren

12.1 Einführung

Das doppelte Plus zeigt an, das es sich bei C, Java oder Perl um objektorientiertes Programmieren handelt. Der Unix–Befehl zum kompilieren lautet wie folgt:

```
1 g++ quelldatei.cpp
2 cpp quelldatei.cpp
```

Die Datei—Endungen sind `.cpp` bzw `.hh`. Die C–Syntax bleibt erhalten wird nur um neue Elemente erweitert.

12.2 Den Vorteil am Beispiel zeigen

Was ist am objektorientierten Programmieren besser?

```
1  typedef struct {
2          double *x;
3          double *y;
4          int anzahl;
5  } twerte;
6  double mittelwert(twerte *w);
7  int datenanhaengen(char *dateiname, twerte *w);
8  int datenfreigeben(twerte *w);
9  int datenausgeben(twerte *w);
```

Für jede Datensammlung muß man jeweils einzeln die Werte angeben, Speicher reservieren und freigeben.Aus diesem Grund hat man Klassen eingeführt.

```
1  class cwerte {
2          public:
3          cwerte(void);
4          ~cwerte(void);
5          double mittelwert(void);
6          double datenanhaengen(char *dateiname);
7          int datenausgeben(void);
8          privat:
9          double *x;
10         double *y;
11         int anzahl;
12 };
```

Klassen sind ähnlich wie Strukturen, können aber zusätzlich Funktion als Elemente enthalten. Das Word `public` besagt, daß die folgenden Variablen/Funktion von außen benutzt werden können. Der private Teil wird nur intern für Berechnunen benötigt. Der Clou wird erst bei der Definition deutlich:

```
1  cwerte::cwerte(void) {
2          anzahl=0;
3          x=NULL;
4          y=NULL;
5  }
6  cwerte::~cwerte(void) {
7          free(x);
8          free(y);
9  }
10 double cwerte::mittelwert(void) {
11         double m=0;
12         for (int i=0;i<anzahl;i++) {
13                 m+=y[i];
14         }
15         return m/anzahl;
16 }
17 int cwerte::datenanhaengen(char *dateiname) {
18         FILE *f;
19         double fx; double fy;
20         f=fopen(dateiname,"r");
21         while (fscanf("%f%f",&fx,&fy)!=EOF) {
22                 x=realloc(x,sizeof(double)*(anzahl+1);
23                 y=realloc(y,sizeof(double)*(anzahl+1);
24                 x[anzahl]=fx;
25                 y[anzahl]=fy;
26                 anzahl++;
27         }
28         fclose(f);
29         return 0;
30 }
31 int cwerte::datenausgeben(void) {
32         for (int i=0;i<anzahl;i++) {
33                 printf("%d\t%lf\t%lf\n",i,x[i],y[i]);
34         }
35         return 0;
36 }
```

Die Doppelpunkte besagen, daß die Definition der Funktion zu der Klasse `cwerte` gehört.

12.3 Konstruktor und Destruktor

Wenn der Funktionsname gleich dem Namen der Klasse ist, so wird die Funktion automatisch beim Initialisieren der Klasse als Variable an beliebiger Stelle des Programms ausgeführt. Das kann beliebig oft geschehen. Man nennt die Initiator–Funktion **Kontruktor**. Ist der Funktionsname wiederum gleich dem Klassenname aber eine Tilde davor steht, handelt es sich um den sogenannten **Destruktor**. Der Destruktor wird ausgeführt, wenn die Klasse als Variable terminiert wird. Bis dahin bleiben alle zugehörigen Variablen und Funktionen erhalten.

12.4 Variablen für den Konstruktor

Es ist möglich, beim Aufruf der Klasse dem Konstruktur Variablen zu übergeben. In der Deklaration wird einfach wie bei Funktionen eine Variablenliste angehängt. Werden mehrere Konstruktoren verwendet, so wird derjenige Konstruktor ausgeführt, welcher die passende Anzahl der übergebenen Parameter hat.

```
1  //Deklaration--------------
2  class cwerte {
3          public:
4          cwerte(int wertanzahl);
5          cwerte(int wertanzahl, double xwert, double ywert);
6          //...
7  };
8  //Definiton---------
9  cwerte::cwerte(int wertanzahl) {
10         anzahl=wertanzahl;
11         x=(double*)malloc(sizeof(double)*wertanzahl);
12         y=(double*)malloc(sizeof(double)*wertanzahl);
13 }
14 cwerte::cwerte(int wertanzahl, double xwert, double ywert) {
15         int i;
16         anzahl=wertanzahl;
17         x=(double*)malloc(sizeof(double)*wertanzahl);
18         y=(double*)malloc(sizeof(double)*wertanzahl);
19         for (i=0;i<anzahl;i++) {
20                 x[i]=xwert;
21                 y[i]=ywert;
22         }
23 }
24 //...Implementierung---------
25 int main(void) {
26         cwerte A(3);
27         cwerte B(5,0,10);
28         return 0;
29 }
```

In diesem Beipiel hat die Variablen–Menge `A` 3 unbestimmte Werte und die Variablen–Menge `B` 5 x–Variablen des Werts Null sowie 5 y–Variablen des Werts Zehn.

12.5 Direktzugriff auf Variablen

Im ANSI–C wird beim Aufruf einer Funktion von allen Variablen eine Kopie erzeugt, mit der die Funktion arbeit, die ursprüngliche Variable der aufrufenden Funktion bleibt unverändert. In C++ kann bei der Variablenübergabe durch ein vorangestelltes kaufmännisches `&`–Zeichen symbolisiert werden, daß die Werte durch die Funktion geändert werden können. Die folgende Passage liefert als Ausgabe immer 3 aus.

```
1  int func(int &x) {
2          x=3;
```

```
3         return 0;
4 }
5 int main(void) {
6         int a=0;
7         func(a);
8         printf("%d\n",a);
9         return 0;
10 }
```

12.6 Überladene Operatoren

Um zum Beispiel 2 komplexe Zahl der Datentyp–Klassen

```
1 class ccomples {
2         double re;
3         double im;
4 };
```

zu addieren, ist es möglich in C++ den Code auf folgende Zeilen abzukürzen:

```
1 int main(void) {
2         ccomplex a,b,c;
3         c=a+b;
4         return 0;
5 }
```

Das ist mit Hilfe von **überladenen Operatoren** möglich. Die Operation Plus `+` in Kombination mit Strukturen wie der Klasse `ccomplex` gibt normalerweise einen Fehler aus, außer man sagt ihm wie die Operation plus mit der Klasse umgeht. Die allgemeine Definiton ist wie folgt:
`Typ operator` $\otimes$ `(Variablenliste)` {
`Code--Block;`
}
Hierbei ist $\otimes$ ist ein beliegibes Zeichen wie `+`, `-`, `*`, `!` oder ähnliche. Sei es am Beispiel der komplexen Addition und Multiplikation gezeigt:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  //Deklaration-----------------------
4  class ccomplex {
5          //Klassendeklaration
6          public:
7          double Re;
8          double Im;
9          ccomplex(double r, double i); //Konstruktor
10         ~ccomplex(void);  //Destruktor
11         ccomplex operator +(ccomplex rechts); //Addtion
12         ccomplex operator *(ccomplex rechts); //Multiplikation
13 };
14 //Definitionen----------------------
15 ccomplex::ccomplex(double r, double i) {
16         //Konstruktor
17         Re=r;
```

```
18          Im=i;
19  }
20  ccomplex::~ccomplex(void) {
21          //Destruktor formal vorhanden
22  }
23  ccomplex ccomplex::operator +(ccomplex rechts) {
24          //Addition
25          ccomplex aux(Re+rechts.Re,Im+rechts.Im);  //lokale Variable (Nebenrechnung)
26          return aux; //gibt lokale Variable zurueck
27  }
28  ccomplex ccomplex::operator *(ccomplex rechts) {
29          //Multiplikation
30          ccomplex aux( Re*rechts.Re-Im*rechts.Im, Re*rechts.Im+rechts.Im*Re );
31          return aux; //gibt lokale Variable zurueck
32  }
33  //Hauptfunktion
34  int main(void) {
35          ccomplex X(4,1),Y(2,3),Z(0,0);
36          Z=X+Y;
37          printf("re=%f im=%f\n",Z.Re,Z.Im);
38          Z=X*Y;
39          printf("re=%f im=%f\n",Z.Re,Z.Im);
40          return 0;
41  }
```

Bei Initialisierung der Klasse `ccomplex` werden die zwei Paramter benötigt: Realteil und Imaginärteil. Im Konstruktor

```
ccomplex::ccomplex(double r, double i)
```

werden die beiden Werte `r` und `i` an die Mitgliedvariablen `Re` und `Im` übergeben. Die Operation Addition `+` wird in der Funktion

```
ccomplex ccomplex::operator +(ccomplex rechts)
```

umgesetzt. Beim Addieren wird

- als Wert links des Operatorzeichens die Klasse selbst verwendet
- als Wert rechts des Operatorzeichnes der übergebene Parameter `rechts`

Es wird eine lokale Nebenrechnunsvariable `aux` erzeugt, welche als Startwerte die Summe von linker und rechter Seite jeweils erhält. Bei der Multiplikation verhält es sich analog.

12.7 Dynamische Speicherverwaltung

Es gibt unter C++ zwei neue Befehle, die aber ein Analogon unter C haben. Es kann auch eine eckige Klammer benutzt werden, wenn es um Felder geht. Sie hierzu Abbildung 2 mit einem 1–dimensionalen Feld `a` und einen $x \times y$ großem Feld `b`:

	1–dimensional (Vektor)

C	`double *a;`
	`a=(double *a) malloc(sizeof(double)*3);`
	`...`
	`free(a);`

C++	`double *a;`
	`a=new double[3];`
	`...`
	`delete a;`

	2–dimensionales Feld (Matrix)

C	`double **b;`
	`int i;`
	`b=(double *)malloc(sizeof(double**)*y);`
	`for (i=0;i<y;i++) {`
	`  b[i]=(double*)*malloc(sizeof(double)*x);`
	`}`
	`...`
	`for (i=0;i<y;i++) {`
	`  free(b[i]);`
	`}`
	`free(b);`

C++	`b=new (double*)[y];`
	`for(int i=0;i<y;i++) {`
	`  b[i]=new double[x];`
	`}`
	`for (int i=0;i<y;i++) {`
	`  delete b[i];`
	`}`
	`delete []b;`

Tabelle 2: Vergleich der dynamischen Speicherreservierung in C und C++

12.8 Templates und Klassen

Vorlagen sind eine Möglichkeit, wie der Benutzer der Klassen bestimmten, welcher Datentyp in einer Klasse verwendet werden soll. Die allgemeine Definition ist

```
template <class Typ> class klassenname {
        Typ Variable;
}
```

Überall in der Defintion wird an der Stelle `Typ` der Datentyp verwendet, welche bei dem Aufrufen der Klasse mittels

```
klassenname<Datentyp> K;
```

erfolgt. Es wird also eine Variable des Datentyp `Datentyp` erzeugt.

Mehrere Datentypen übergeben Bei Verwenden einer Vorlage kann nicht nur 1 Datentyp übergeben werden, sondern mehrere. Es wird einfach durch Komma getrennt.

Beispiel einer flexiblen Matrix Nun sei das Beispiel einer Matrizenklasse gegeben, welches Matrizen mit Matrix–Elementen des Typs `float`, `int` oder gar einer komplexen `ccomplex`–Klasse sein kann.

```
1  //Class Template for matrix
2  //Deklaration--------
3  typedef struct {
4          double im;
5          double re;
6  } tcomplex;
7  template <class Typ, int dx, int dy> class cmatrix {
8          public:
9          Typ **M;
10         cmatrix(void);
11         ~cmatrix(void);
12         int dimx;
13         int dimy;
14         int setval(int x,int y, Typ val);
15 };
16 typedef cmatrix<float,3,3> tmat33;
17 //Definition------------
18 template <class Typ, int dx, int dy> cmatrix<Typ, dx, dy>::cmatrix(void) {
19         M=new (Typ*)[dx];
20         for (int i=0; i<dx;i++) {
21                         M[i]=new Typ[dy];
22         }
23         dimx=dx;
24         dimy=dy;
25 }
26 template <class Typ, int dx, int dy> cmatrix<Typ, dx, dy>::~cmatrix(void) {
27         for (int i=0;i<dimx; i++) {
28                         delete M[i];
```

```
29          }
30          delete []M;
31  }
32  //Implementierung-----------------
33  int main(void) {
34          cmatrix<float,3,3> f_mat;
35          cmatrix<tcomplex,3,3> c_mat;
36          cmatrix<tmat33,3,3> m_mat;
37          return 0;
38  }
```

Es kann also im Hauptprogramm eine Matrix der ein und derselben Klasse **verschiedener Typen** eingeführt werden. In diesem Beispiel sind es die Matrixelemente des Datentyps `float`, eines komplexen `double` und einer 3×3–`double`–Matrix, also

$$\left(\begin{array}{ccc} \left(\begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array} \right) & \left(\begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{array} \right) & \ldots \\ \left(\begin{array}{ccc} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & c_{33} \end{array} \right) & \ddots & \\ \left(\begin{array}{ccc} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{32} & g_{33} \end{array} \right) & & \end{array} \right)$$

Oder es wird mit einer 1×3–Matrix ein Vektor definiert. Man erkennt also leicht, daß objektorientiertes Programmieren viele Vorteile bietet.

13 Numerische Lösungen

13.1 Mathematica und C

Es ist möglich C–Funktionen von Mathematica auszuführen. Für viele meist einfache numerische Problemfälle ist die Lßoung schneller Mathematica getan als umständliches Handtieren mit Feldern und Speicherzuweisungen unter C. Wenn aber komplexe Rechnungen mit vielen Operationen notwendig sind, ist eine Umsetzung der speziellen Funktionen in C viel rechenzeitsparender.

13.1.1 Welche Schritte sind notwenig

Mathematica installieren Das sollte der Administrator bereit getan haben.

C–Quelldatei erstellen Es wird die Bibliothek `mathlink.h` benötigt, welche bei einer korrekten Mathematica–Installation automatisch verwendet wird. Fügen Sie die Zeile `#inlude "mathlink.h"` in den Kopf der Datei ein.

```
1  #include "mathlink.h"
2
```

```
3  long facult(long x) {
4          long i; long f=1;
5          for (i=1;i<=x;i++) {
6                  f *= i;
7          }
8          return f;
9  }
10 long binominal(long n, long k) {
11         return facult(n) / ( (double) facult(k) * facult(n-k) );
12 }
13
14 int main(int argc, char *argv[]) {
15         return MLMain(argc, argv);
16 }
```

Eine TM–Datei erstellen Sie enthält ihre C–Quelle mit der speziellen Funktion sowie Zuweisung für die Syntax in Mathematica. Das geschiet im Teil zwischen den Zeilen `:Begin:` und `:End:`. E Sei ein Beispiel der Fakultätberechnung in der Datei `binominal.tm` gegeben:

```
1  //Schnittstelle fuer nb-Datei
2  :Begin:
3  :Function: binominal
4  :Pattern: Binominal[n_Integer, k_Integer]
5  :Arguments: {n, k}
6  :ArgumentTypes: {Integer, Integer}
7  :ReturnType: Integer
8  :End:
```

Kompilieren Das Kompilieren geschieht mit dem `mcc`–Kompiler. Dieser ruft den normalen `gcc`–Kompiler auf und macht zusätzliche Aktionen. Fïr das Beispiel wäre es:

```
mcc binominal.tm binominal.c -o binominal
```

Unter BSD verwende man das Paket `mathlink_bsd`, wobei der Befehl anders lautet:

```
mcc_bsd binominal.tm binominal.c -o binominal
```

Unter Windows sollte die Endung `.exe` bei Binärdateien vorliegen.

Mathematica–Datei Es muß in der Mathematica–Datei die Binärdatei installiert. Ist das geschehen, können alle für die Mathematica–Syntax freigeschalteten Funktionen genutzt werden:

```
In[1]:= Install["facul"]
Out1]= LinkObject["./facul", 2, 2]
In[2]:= Binominal[10, 2]
Out[2]= 45
```

13.2 Differentialgleichungen numerisch lösen

Es gibt Differentialgleichungen (DGLen), zu denen keine analytisch Lösungsmethoden bekannt sind.

13.3 Beispiel 1

Wie kann man die folgende DGL lösen. Es wird eine kleiner metallener Körper in das Magnetfeld eines Ringmagneten gebracht. Wie wird sich der Körper bewegen? Es liegen die Koordinaten r als Abstand zum Mittelpunkt des Ringmagneten und der Winkel ϕ vor sowie die Konstanten C und f und der Masse m.

$$\begin{aligned} m\ddot{r} &= mr\dot{\phi}^2 + \frac{3c}{(r-f)^4} \\ mr^2\ddot{\phi} &= -2mr\dot{r} \end{aligned}$$

- DGl umstellen nach Grad der Ableitung ordnen und Größen kürzen

$$\begin{aligned} \dot{t} &= -\frac{r}{2} \\ \ddot{r} &= r\ddot{\phi}^2 + \frac{3c}{m(r-f)^4} \end{aligned}$$

- **Substituieren**, d.h. es wird eine Ableitung durch eine zeitabhängige Größe ersetzt.

$$\begin{aligned} y_0 &= r \\ y_1 &= \dot{r} \\ y_2 &= \phi \\ y_3 &= \dot{\phi} \end{aligned}$$

- **Reduktion der Ordnung**. Nun liegen 4 Diffentialgleichungen in einem DGL-System erster Ordnung vor.

$$\begin{aligned} \dot{y}_0 &= y_1 \\ y_1 &= -\frac{y_0}{2} \\ \dot{y}_2 &= y_3 \\ \dot{y}_1 &= y_0 + y_3^2 + \frac{3c}{m(y_0-f)^4} \end{aligned}$$

- Mit DGl 1. Ordnung kann schrittweise integriert werden. Dies erfolgt zu jedem Zeitschritt mit jeder DGL des Systems. Beim Euler–Verfahren wird zum Koordinaten–Wert q_n zur vorherigen Wert q_{n-1} ein Term aus Produkt der Ableitung $\dot{q}$ und des Zeitschritts Δt addiert.