Albert–Ludwigs–Universität
Freiburg i. Br.

Grundlagen der **Programmiersprache C** für Studierende der Naturwissenschaften

Rebekka Axthelm

Freiburg im Breisgau
SS 2009

Bisher haben an diesem Skript einige fleißige "Lehrer" mitgewirkt, die hier auf einer kleinen Gedenktafel zusammengefaßt wurden:

Stefan Wiedenbauer

Bernard Haasdonk, 1997

Claudius Link, 1998

Micha Fried, 2001

Rebekka Axthelm, 2002

Micha Fried, $\star$

.....

Micha Fried, $\star$

Rebekka Axthelm, 2009

Inhaltsverzeichnis

–1–
Programmieren

Die Erstellung eines C Programms erfolgt in mehreren Arbeitsschritten:

1. Der Programmquelltext wird mit einem Editor erstellt und abgespeichert, etwa unter dem Namen `hello.c`
2. Anschließend wird der Quelltext mit einem Compiler (hier gcc) mittels
   ```
   gcc -o hello hello.c
   ```
   in ein ausführbares Programm namens `hello` übersetzt. Im Computer passiert hier eine Abfolge von mehreren Schritten, die für den Programmierer im Verborgenen bleiben.
3. Mit der Eingabe von `hello` in der Kommandozeile führt man es dann aus.

Ein laufendes Programm lässt sich mit `<Ctrl>+<C>` oder durch schließen des xterms abbrechen.

1.1 Das erste C Programm

Anhand von folgenden Programmen steigen wir direkt in die C Programmierung ein. Die Details der Befehle werden in den folgenden Lektionen erläutert, aber mit dem intuitiven Verständnis können wir schon das erste eigene Programm schreiben. Eine Datei namens `hello.c` enthalte folgenden Text (ohne die Zahlen in den jeweiligen Zeilenanfängen, sie dienen nur der Beschreibung):

```
1 #include <stdio.h>
2
3 main()
4 {
5   printf("hello world!\n");
6 }
```

Im Einzelnen passiert hier einfach folgendes: In Zeile 1 wird eine Hilfsbibliothek eingebunden. In diesem Fall `stdio.h` (**st**andar**d** **i**nput/**o**utput. Sie enthält standardisierte Ein– und Ausgaberoutinen. Das eigentliche Hauptprogramm beginnt in Zeile 3 und umfasst die Klammerung in den Zeilen 4 und 6. Geschweifte Klammern bündeln immer einen Block von Befehlen. In Zeile 5 wird die Funktion `printf()` aufgerufen. Wir werden später mehr über Funktionen erfahren. `printf(string);` gibt die Zeichenkette `string` nach stdout aus, was den Bildschirm meint. `n` ist eine Zeichenkonstante, die für den Zeilentrenner (newline) steht.

Wichtig: Jede logische Zeile eines C Programms muss mit einem Semikolon abgeschlossen werden!

C Programme können formatfrei geschrieben werden, das heißt dass sie keine bestimmte Zeilenstruktur haben müssen. Ausnahme: Zeichenketten, das heißt durch eingegrenzte Textbereiche, dürfen nicht über das Ende einer Zeile hinausgehen. Trotz dieser Freiräume beim Programmieren, sollte man sich gleich zu Anfang einen Programmierstil angewöhnen, der lesbare Programme ergibt. Eine beliebte Formatierung ist die nach Kernighan & Ritchie (kurz K&R oder nur KR) den Entwicklern von C benannte. Sie rücken jeden Programmblock einen Tab weiter ein. Die geschweiften Klammern, die den Block umschließen stehen dann entweder auf der selben Höhe wie der äußere Block (bei Funktionen immer) oder es wird bei Schleifen und zum Beispiel `if-else`–Bedingungen die öffnende Klammer noch in der selben Zeile geschrieben. Beispiel:

```
if(i>=10) {
   i=0;
} else {
   i++;
}
```

Kommentare dienen dazu ein Programm lesbarer und auch verständlicher zu machen. Sie sollten dementsprechend häufig verwendet werden und man sollte versuchen, sie sinnvoll zu formulieren. Sie können überall im Programmtext erscheinen und müssen mit `/* Kommentartext */` eingeklammert werden. Kommentare werden vom Compiler ignoriert. Dies ist beim Debuggen (= Fehler suchen) sehr nützlich, da man damit selektiv ganze Programmteile auskommentieren und so Fehlerbereiche einschränken kann.

1.2 Das zweite C Programm

Jetzt ein etwas komplexeres Programm. Wir editieren die Datei `first.c`.

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 /* Programm zur Berechnung der Quadratwurzel */
 5
 6 main()
 7 {
 8   float x, y;
 9
10   printf("Berechnung der Quadratwurzel einer positiven Zahl\n");
11   printf("Gib eine Zahl ein: ");
12
```

```
13   scanf("%f",&x);
14
15   printf("\n");
16
17   if( x >= 0 ){
18     y = sqrt(x);
19     printf("Die Wurzel von %f lautet %f\n",x,y);
20   } else
21     printf("Eine reelle Wurzel existiert nicht!\n");
22
23   return 0;
24 }
```

In Zeile 1 und 2 werden wieder Hilfsbibliotheken eingebunden. `math.h` enthält verschiedene mathematische Funktionen. Mit dem Unixbefehl `locate stdio.h` kann man herausfinden, wo die headerfiles der Hilfsbibliotheken zu finden sind. Meist in `/usr/include`. Was headerfile und Bibliothek genau bedeutet werden wir später besprechen. Zunächst benötigen wir das, um überhaupt etwas machen zu können. Zeile 4 enthält ein Kommentar. Alles was zwischen `/*` und `*/` geschrieben steht, wird später für den Compiler nicht sichtbar sein. Alles in einer Zeile, das dem Doppelslash (`//`) folgt wird als Kommentar interpretiert.

```
// Ich bin auch ein Kommentar
```

In Zeile 8 werden zwei Variablen `x` und `y` deklariert. Wir stellen damit klar, dass wir auf im Speicher des Computers Platz für zwei Variablen benötigen, die Fließkommazahlen darstellen können sollen. Wir werden diesen Punkt in Kürze etwas detaillierter Besprechen. Die `printf`s in den Zeilen 10 und 11 kennen wir schon. `scanf` in Zeile 13 ist das Gegenstück zu `printf`. Es liest Zeichen ein von stdin, was in diesem Fall die Tastatur meint. `%f` beschreibt das erwartete Format der Eingabe, in diesem Fall Fließkomma. Das & vor der Variablen ist der Adressoperator. Er liefert die Adresse im Speicher des Computers zurück, an der der Inhalt der folgenden Variable (hier also `x`) gespeichert wird. Der Rest ist selbsterklärend und wird syntaktisch in späteren Kapiteln genauer beschrieben werden.

Das Programm wird übersetzt mit dem Kommando

```
gcc -o first first.c -lm
```

`-lm` braucht man hier, damit die Bibliothek, in der die Wurzelfunktion `sqrt()` definiert ist dazugelinkt wird. Dem ausführbaren Programm `first` ist dann die Wurzelfunktion bekannt. Jeder Compiler besitzt eine Vielzahl an Optionen, von denen wir einige besprechen werden. Das aber später.

Was passiert beim Compilieren?

1. (preprocessing) Vorverarbeitung des Textes:
   - Einsetzen von allen includes und Definitionen, kurz allem, was mit # beginnt

   - Entfernen der Kommentare
   - Überprüfen der Syntax

2. (compilation) Übersetzen des Quelltexts in einen Assembler Quelltext.
3. (assembly) Übersetzen des Assembler Quelltexts in Maschinen Code. Hierbei entstehen `.o`–Objektdateien.
4. (linking) Zusammensetzen aller Objektdateien und Bibliotheken zu einem ausführbaren Programm.

–2–
Variablen und Konstanten

2.1 Variablendeklaration

Eine Variable ist eine Stelle im Hauptspeicher des Computers, in der man einen Wert ablegen kann und später wieder abrufen kann. Über den Variablennamen kann man auf den Inhalt des betreffenden Speicherbereichs zugreifen, ihn verändern oder lesen. Es gibt verschiedene Typen von Variablen. Diese verschiedenen Typen belegen unterschiedlich viel Speicherplatz.

Die wichtigsten Typen	
int	ganzzahliger Wert
long	ganzzahlier Wert mit größerem Wertebereich
float	Fließkommazahl mit mindestens sechs Nachkommastellen
double	Fließkommazahl mit mindestens zehn Nachkommastellen
char	ein Zeichen

Integerzahlen können zusätzlich noch in den Formaten `unsigned` und `signed` variiert werden. Die Größen der Speicherbereiche sind maschinenabhängig. Bevor eine Variable im Programm benutzt werden kann, muss sie deklariert werden. Dies geschieht durch Angabe des Typs und Namens der Variablen. Außerdem kann man die Variable direkt initialisieren, das heißt einen bestimmten Wert zuweisen.

```
int i;                          //Deklaration
int j = 0;                      //Deklaration und Initialisierung
int iteration1, iteration2      //Zwei Deklarationen
char a = "2";                   //Deklaration eines Zeichens
```

In der ersten Zeile wird die ganzzahlige Variable `i` deklariert, die zweite Zeile deklariert die ganzzahlige Variable `j` und initialisiert sie mit 0. Es können auch mehrere Variablen desselben Typs in einer Zeile deklariert werden.

Konstanten sind, wie Variablen, benannte Speicherstellen. Allerdings behalten Konstanten immer ihren Wert. Eine Konstante muss bei ihrer Deklaration initialisiert werden und behält ihren Wert während der gesamten Programmausführung. Konstanten werden mit dem Schlüsselwort `const` definiert:

```
const float pi = 3.14;
```

Dieser Wert wird sich nicht ändern, weshalb er auch direkt bei der Daklaration definiert werden muss.

Achtung: Es ist compilerabhängig, was passiert, wenn man mit Variablen arbeitet, denen man nie einen Wert zugeordnet hat! Nicht immer erhält man eine Fehlermeldung oder auch nur eine Warnung. Es sei denn, man hat dem Compiler entsprechende Optionen übergeben. Doch dazu später noch.

Klassischer Fehler: Bei der Summation wird "gerne" die Initialisierung vergessen. Beispiel:

```
int i, sum;

for(i=0 ; i<10 ; i++)
  sum += i;
```

Richtig wäre es, vor der `for`–Schleife, `sum=0;` zu setzen.

2.2 Ausgabeanweisung `printf()`

Die Ein– und Ausgabe ist nicht als Teil der Sprache C definiert. Sämtliche ein– und Ausgaben sind in der header–Datei `stdio.h` als externe Funktionen deklariert. Sollen in einem Programm Ein– und Ausgabe erfolgen, muss diese Datei mit der `include`–Anweisung in das Programm eingebunden werden: `#include <stdio.h>`.

Ausgaben an den Bildschirm erfolgen mit dem Befehl `printf`, was soviel wie formatierte Ausgabe heißt. Dabei können Text und der Inhalt von Variablen ausgegeben werden. Zur Veranschaulichung sei folgendes Beispiel gegeben:

```
#include<stdio.h>

main()
{
   int i;
   unsigned j;
   float y;
   double x;

   i = -10;
   j = 4;
   x = 3.7235267335643;
   y = 1.23;


   printf("i=%d, j=%u, x=%f und y=%f  \n",i,j,x,y);
}
```

Dieses Programm liefert die Ausgabe

```
i=-10, j=4, x=3.723527 und y=1.230000
```

Im einzelnen funktioniert das so: Für jede Variable, deren Inhalt man ausgeben möchte, muss eine Formatzeichenfolge (hier `%d`, `%f`, etc) angegeben werden. Diese muss dem Typ der Variablen entsprechen. Die Formatzeichenfolge `\n` bewirkt einen Zeilenumbruch. Dieses Zeichen sollte man nicht vergessen, vor allem nicht bei der letzten Ausgabe des Programms!

Die wichtigsten `printf()` Umwandlungen	
d	int, Dezimal
f	double, Gleitkommazahl (auch float)
c	char, Einzelnes Zeichen
s	char*, String, bzw. Zeichenkette

Außerdem ist es möglich, eine Formatierung der Ausgabe durchzuführen: `%12.8f` bewirkt zum Beispiel, dass die Ausgabe einer Fließkommazahl in mindestens 12 Zeichen (eventuelle führende Leerstellen) und 8 Nachkommastellen.

Damit ergäbe im obigen Beispiel die Anweisung

```
printf("x=%8.2f text\n",x);
```

folgende Ausgabe:

```
x=    3.72 text
```

Die Ausgabe wird standardmäßig rechtsbündig ausgegeben. Eine Linksbündigkeit kann man mit einem − in der Formatzeichenfolge erzwungen werden. Ein Beispiel:

Durch

```
printf("x=%-8.2f text\n",x);
```

erfolgt Ausgabe:

```
x=3.72     text
```

Damit kann man eine Ausgabe schön und übersichtlich gestalten.

2.3 Definition von Variablentypen

Vorhandenen Typen (z.B. `unsigned int`) kann mittles `typedef` ein Name gegeben werden:

```
typedef unsigned int UINT
```

Statt

```
unsigned int x
```

kann nun einfach

```
UINT x
```

geschrieben werden.

2.4 Felder

Oft braucht man bei der Programmierung Felder. Zum Beispiel zum Abspeichern von Vektoren und Matrizen. Bei der Deklaration eines Feldes muss der Typ und die Größe angegeben werden, zum Beispiel:

```
int    i[10];
float  a[10], b[10];
double A[4][3];
```

In der ersten Zeile wird ein Feld vom Typ int mit Indizes 0...9 angelegt, in der zweiten Zeile zwei Felder vom Typ float der gleichen Größe und in der dritten ein zweidimensionales Feld mit Indizes 0...3, 0...2. Die Zuweisung von Werten erfolgt bei Feldern wie bei "normalen" Variablen:

```
i[0] = 3;
i[7] = 0;
A[2][2] = 3.5436;
A[0][2] = 0.;
```

Klassischer Fehler: Ein Feld der Größe N hat Indizes 0...N-1, nicht 1...N! Im Beispiel von gerade eben ist zum Beispiel `i[10]` nicht sinnvoll und kann zu einem Abbruch des Programms führen. Das Überschreiben eines Feldes oder Arrays ist auch einer der häufigsten und leider oft auch am schwersten zu findende Programmierfehler in C.

–3–
Operatoren

Mit Hilfe von Operatoren lassen sich Ausdrücke formulieren. C beinhaltet unter anderem folgende Zeichen, beziehungsweise Zeichenfolgen als Operatoren:

```
+ - * / %
++ -- += -=
< <= > >= == !=
&& ||
<< >>
```

Es gibt noch einige mehr, aber für uns soll das zunächst genügen. Einige Operatoren haben je nach Kontext unterschiedliche Wirkungen. So bewirkt zum Beispiel der Stern * in arithmetischen Ausdrücken eine Multiplikation, vor Variablenbezeichnungen wirkt der Stern als Zeigeroperator.

3.1 Arithemtische Operatoren

In C arbeiten die Operatoren

`+, -, *, /` und `%`

wie in den meisten anderen Programmiersprachen. Sie können auf die Datentypen `int`, `float` und `double` angewendet werden, wobei die Verknüpfungen nur auf Daten des gleichen Typs angewendet werden sollten. Es sei denn, man weiß was man tut. Es kann zum Beispiel zu falschen Ergebnissen führen, wenn man eine `float`– mit einer `int`–Variablen addiert.

Achtung: Bei der Anwendung von `/` auf eine `int`–Variable wird der Nachkommaanteil abgeschnitten. Zum Beispiel liefert `5/2` den Wert 2 hingegen `5./2.` den Wert `2.5`.

`x%y`

liefert den Rest, der entsteht, wenn `x` durch `y` dividiert wird, und ist gerade Null wenn `x=y`. Der Operator `%` kann auf `float`– oder `double`–Werte nicht angewendet werden.

Vorrang:

```
1. +, - (unaer, positive und negative Vorzeichen)
2. *, /, %
3. +, - (binaer, Operation)
```

3.2 Inkrement– und Dekrement–Operatoren

Zwei C–typische Operatoren sind die Inkrement– und Dekrement–Operatoren ++ und --. Der Operator ++ addiert 1 und -- subtrahiert 1.

```
x = x + 1
```

ist das Gleiche wie

```
x++
```

oder auch

```
x += 1.
```

+= kann auch in Kombination mit anderen Werten wie die 1 angewendet werden. Etwa erhöht `x += 5` den Wert von `x` um 5. Das Gleiche gilt für -- und -=. Diese Operatoren benötigt man zum Beispiel zur Konstruktion von Schleifen. Doch darauf und auf weitere Operatoren werden im Laufe des Kurses noch eingehen. Der Vorteil und Sinn dieser Anweisung, ist neben weniger Zeichen zur Eingabe und der erhöhten Lesbarkeit (für geübte C Programmierer), dass sie intern viel effizienter ersetzt werden können.

++ und -- können sowohl vor als auch nach dem Operanden angegeben werden. In beiden Fällen wird inkrementiert. `++i` wird inkrementiert bevor der resultierende Ausdruck verwendet wird und `i++` wird inkrementiert nachdem der resultierende Ausdruck verwendet wird.

Beispiel:

```
int n = 5;
int x, y;

x = n++;              // n erhaelt den Wert 6 und x erhaelt den Wert 5
y = ++n;              // n erhaelt den Wert 7 und y erhaelt den Wert 7
```

3.3 Vergleichsoperatoren

Die Vergleichsoperatoren sind

>, >=, < und <=

und die Äquivalenzoperatoren sind

== und !=.

a<b fragt ab, ob a echt kleines als b ist und liefert entsprechend 0 für falsch und 1 für richtig. <= fragt nach kleiner gleich und die entsprechenden Ausdrücke mit + funktionieren analog. == fragt nach Gleichheit und != nach Ungleichheit.

Es ist `if(!valid)` das Gleiche wie `if(valid==0)`.

Achtung: Ein (un)gern gesehener Fehler ist die Abfrage `if(a=b)`. Dieser Ausdruck stellt keinen Vergleich dar, sondern eine Zuweisung. Die Folge wird sein, dass `a=b` ist und somit die `if`–Abfrage ein `true` erhält, ganz egal was a vorher mal gewesen ist!

Vorrang:

```
1. >, >=, <, <=
2. ==, !=
```

3.4 Logische Operatoren

`&&` und `||`

stellen die logischen Operatoren dar. `&&` steht für logisches `UND` und `||` für logisches `OR`. Ein Beispiel:

```
a==b && a==c
```

ist wahr, wenn `a=b` und `a=c` gilt.

Ausdrücke die mehrere logische Verknüpfungen darstellen werden von links nach rechts ausgewertet.

Vorrang:

```
1. &&
2. ||
```

Beispiel:

```
a==b && b==c || b==d
```

ist das Gleiche wie

```
(a==b) && (b==c) || (b==d)
```

oder auch

```
((a==b) && (b==c)) || (b==d)
```

Es wird allerdings etwas ganz anderes durch

```
(a==b) && ((b==c) || (b==d))
```

ausgedrückt. Nehmen wir einmal an `a=2` und `b=c=d=1`, dann wäre der Ausdruck `((a==b) && (b==c)) || (b==d)` wahr, der Ausdruck `(a==b) && ((b==c) || (b==d))` hingegen falsch.

Vorrang aller bisherigen Operatoren:

```
1. +, - (unaer, positive und negative Vorzeichen)
2. *, /, %
3. +, - (binaer, Operation)
4. >, >=, <, <=
5. ==, !=
```

Beispiel:

`i<a-1` ist das Gleiche wie `i<(a-1)`

Tipp: Lieber eine Klammer zu viel als zu wenig.

3.5 Format von C–Programmen, Kommentare

C–Programme können formatfrei geschrieben werden, d.h. dass sie keine bestimmte Zeilenstruktur haben müssen. Eine Ausnahme besteht bei Zeichenketten, d.h. durch `""` eingegrenzte Textbereiche. Diese dürfen nicht über das Ende einer Zeile hinausgehen.

Wichtig: Das Semikolon am Ende einer Anweisung darf nicht vergessen werden.

Trotz dieser Freiräume beim Programmieren, sollte man sich gleich zu Anfang einen Programmierstil angewöhnen, der für Menschen lesbare Programme ergibt. Eine beliebte Formatierung ist die nach den Entwicklern von C, Kernigham und Ritchie, K&R oder nur KR benannte. Sie rücken jeden Programm–Block einen "Tab" weiter ein. Die geschweiften Klammern $\{\}$ die den Block umschließen stehen dann entweder auf der selben Höhe wie der äußere Block (bei Funktionen immer) oder es wird bei Schleifen und z.B. `if–else`–Bedingungen (vgl. das folgende Kapitel) die öffnende Klammer noch in der selben Zeile geschrieben. Beispiel:

```
if( i>=10 ) {
    i=0;
} else {
    i++;
}
```

Durch die Verwendung von Kommentaren soll ein Programm lesbarer und verständlicher werden. Dementsprechend klar sollten Kommentare denn auch formuliert werden. Sie können überall im Programmtext erscheinen und müssen mit `/* Kommentartext */` eingeklammert werden. Kommentare werden vom Compiler ignoriert.

Kommentare sind auch bei der Fehlersuche, dem "Debuggen", oft sehr nützlich, da man damit selektiv ganze Programmteile auskommentieren kann, und so Fehlerbereiche eingeschränkt werden können.

–4–
Algorithmus, Struktogramme und Flussdiagramme

Algorithmen sind genau definierte Verarbeitungsvorschriften zur Lösung einer Aufgabe. Sie beschreiben ein Schema, welches unter Verwendung von endlich vielen Arbeitsschritten ein bestimmtes Problem löst. Ein Algorithmus besteht aus einer endlichen Folge von Anweisungen, die nacheinander ausgeführt werden. Die Anweisungen können unter bestimmten Bedingungen wiederholt werden.

Beispiel eines Algorithmus: Bedienung einer Waschmaschine

1. Tür der Waschmaschine öffnen.
2. Max. 5 kg Wäsche (einer Farbe) einfüllen.
3. Tür der Waschmaschine schließen.
4. Waschmittel passend zur Farbe der Wäsche in die kleine Schublade für
5. den Hauptwaschgang füllen. Wasserzulauf öffnen.
6. Waschprogramm wählen.
7. Starttaste drücken.
8. Waschvorgang abwarten.
9. Nach Programm-Ende Maschine abstellen.
10. Wasserzulauf schließen.
11. Tür öffnen und Wäsche entnehmen.

Eigenschaften eines Algorithmus:

- Ein Algorithmus benötigt endlich viele Arbeitsschritte.
- Ein Algorithmus ist beschreibbar.
- Jeder Arbeitsschritt ist ausführbar.
- Ein Algorithmus liefert unter identischen Startbedingungen immer das

- gleiche Endergebnis.
- Der Ablauf des Verfahrens ist eindeutig definiert.

Ablaufsteuerung:

- Durch die Ablaufsteuerung wird die Abarbeitung der einzelnen Aktionen festgelegt.
- Abläufe können mit Hilfe von Flussdiagrammen oder Struktogrammen dargestellt werden.
- Folgende Elemente sind in Flussdiagrammen und Struktogrammen vorhanden:
  - Folge (Sequenz)
  - Auswahl (Selektion)
  - Schleifen (Iteration)
  - Sprünge, um Schleifen vorzeitig zu verlassen oder um zu anderen
  - Algorithmen zu springen.
  - Unterprogrammaufrufe
  - Ein-/Ausgabeoperationen, bzw. Handoperationen

4.1 Komponenten eines Algorithmus

Folge (Sequenz):

- Eine bestimmte Anzahl von Aktion werden nacheinander ausgeführt.
- Die Anweisungen werden ohne Einschränkung ausgeführt.

Auswahl (Selektion):

- Die Anweisungen werden nur unter bestimmten Bedingungen ausgeführt.
- Die Aktionen werden nur mit Einschränkungen ausgeführt.
- Man unterscheidet zwischen:
  - Einfache bedingte Anweisung.
  - Vollständige bedingte Anweisung.
  - Fallunterscheidung.

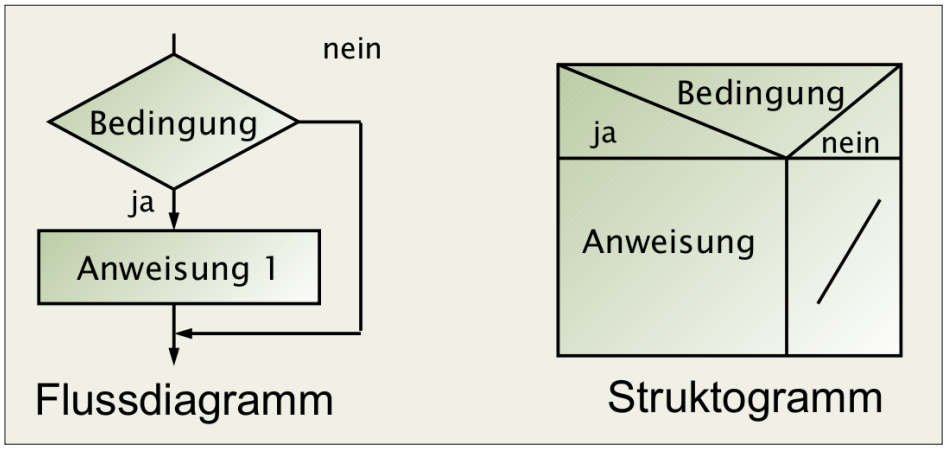
Schleifen (Iteration):

- Schleifen sind eine oder mehrere Aktionen, die in Abhängigkeit von einer Bedingung wiederholt ausgeführt werden.
- Die Schleife wird so lange durchlaufen, bis eine Abbruchbedingung erfüllt wird. Diese wird, je nach Schleifentyp, am Anfang oder am Ende der Schleife geprüft.
- Die Schleife selber besteht aus
  - Anweisungen, die wiederholt ausgeführt werden sollen. Eine dieser Anweisungen verändert auch den Wert von Variablen welche die Abbruchbedingung beeinflussen (z. B. Stand eines Zählers)
  - Der Entscheidung für den Abbruch oder das Fortsetzen der Schleife anhand der Abbruchbedingung (z. B. des Zählerstandes)

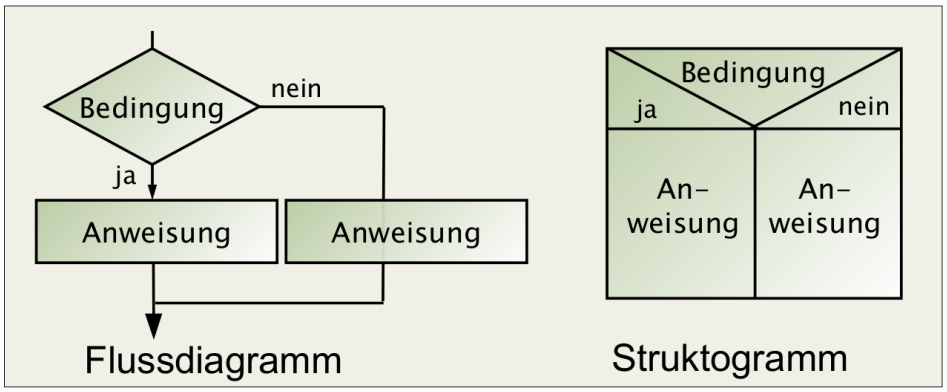
4.2 Struktogramme und Flussdiagramme

- Darstellung in einem Flussdiagramm .
  - Ein Flussdiagramm ist eine Ablaufdiagramm für Computerprogramme.
  - Die benutzten Symbole sind in der DIN 66001 genormt.
- Darstellung als Struktogramm.
  - Struktogramme werden auch als Nassi-Shneidermann-Diagramme bezeichnet.
  - Die benutzten Symbole sind in der DIN 66261 genormt.
  - Struktogramm-Editoren (Freeware) sind unter http://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm zu finden. Der dort aufgeführte HUS-Struktogrammer zeichnet sich durch eine einfache Bedienung aus.

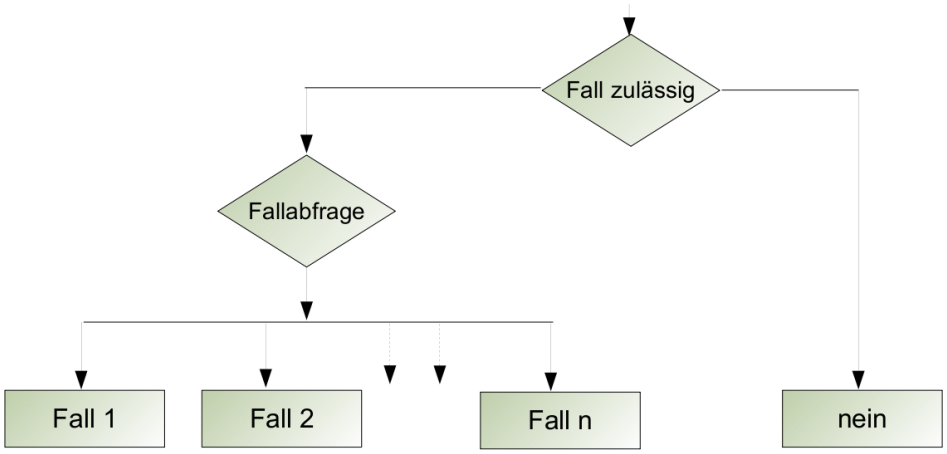
Die Anweisungen werden nur ausgeführt, wenn die angegebene Bedingung erfüllt ist.



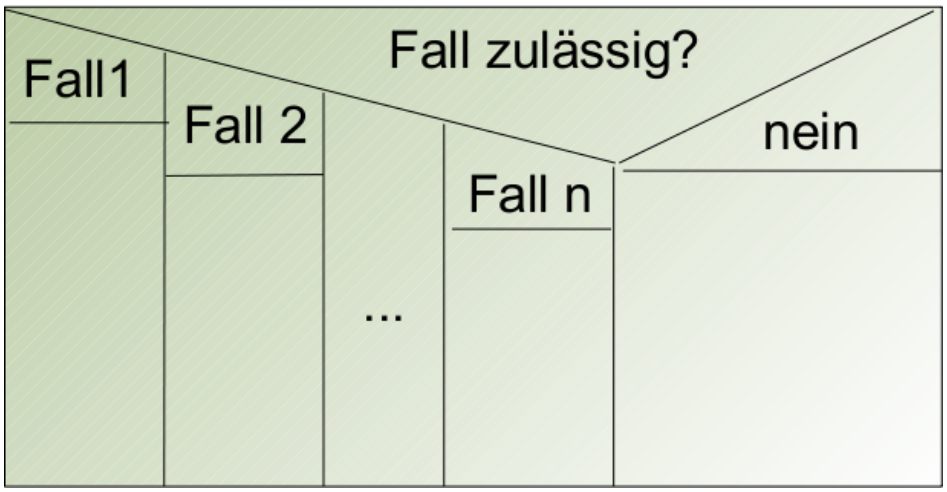
Manche Anweisungen werden ausgeführt, wenn eine Bedingung erfüllt ist und andere, wenn die Bedingung nicht erfüllt ist.



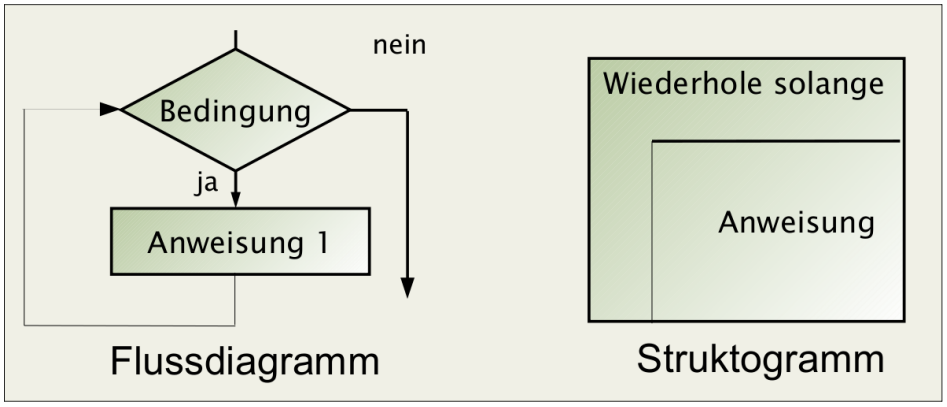
Es sind mehr als zwei Bedingungen vorhanden. Abhängig von einem Wert werden verschiedene Anweisungen ausgeführt.



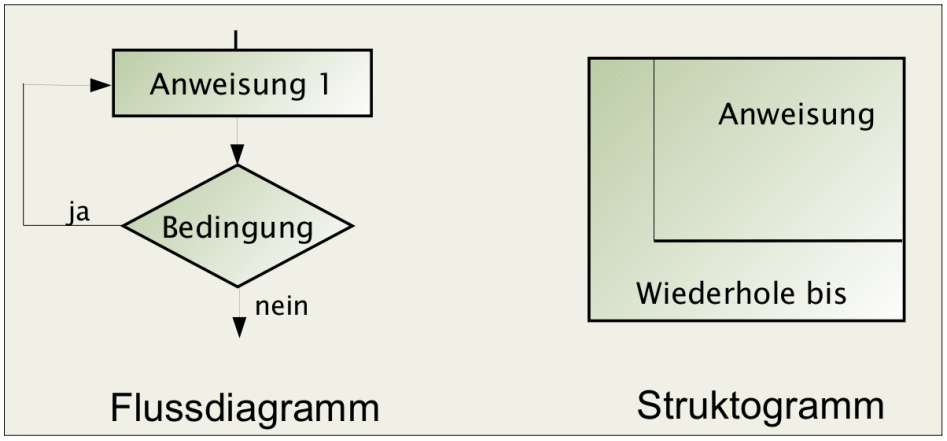
Es sind mehr als zwei Fälle vorhanden. Abhängig von einem Wert werden verschiedene Anweisungen durchgeführt.



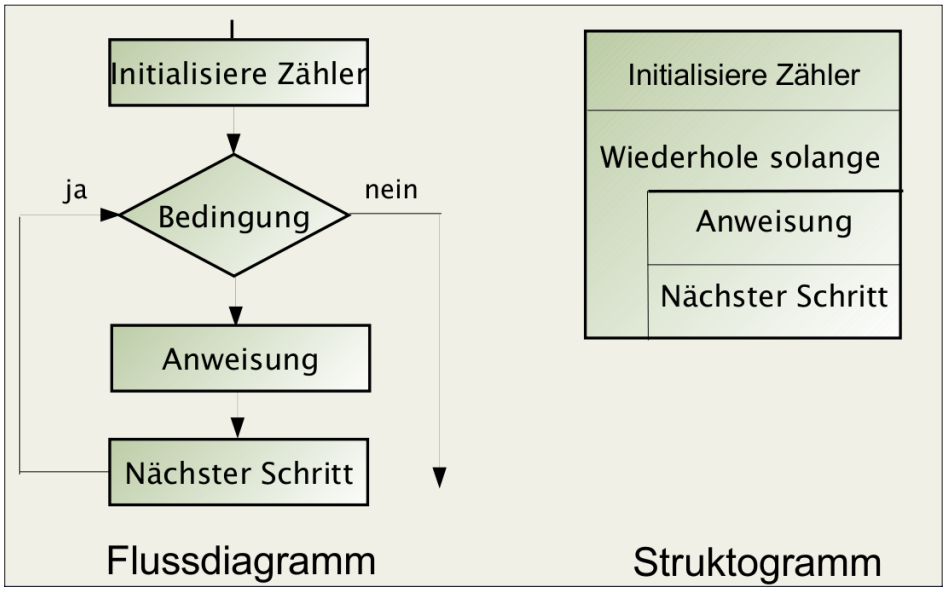
Solange die Bedingung erfüllt ist, führe die Anweisung aus.



Bis die Bedingung erfüllt ist, führe die Anweisung aus.



Definition von Schleifen:



–5–
For– und If–Else–Anweisungen

5.1 Die for–Anweisung

Die am häufigsten benötigte Schleife ist die `for`-Schleife. Sie wird eingesetzt, wenn eine bestimmte Aktion mehrfach hintereinander ausgeführt werden soll und die Anzahl der Wiederholungen vorher bekannt ist. Ein Beispiel:

```
for(i=0; i<100; i++) {
    summe = summe + i;
}
```

Innerhalb der runden Klammern stehen drei Ausdrücke, die durch Semikolon getrennt werden. An erster Stelle die Initialisierung, sie wird genau einmal ausgeführt, nämlich unmittelbar bevor die Schleife begonnen wird. Der zweite Ausdruck ist die Bedingung, welche die Schleife kontrolliert. Dieser Ausdruck wird ausgewertet, und falls die Bedingung erfüllt ist, werden die Anweisungen der Schleife ausgeführt, d. h. die Anweisungen, die im nachfolgenden Block durch die Klammern { } zusammengefasst werden. Der letzte Ausdruck enthält eine Anweisung, welche am Ende der Schleife ausgeführt wird. Normalerweise steht hier ein Inkrement oder Dekrement der Kontrollvariablen, das ist in unserem Beispiel die Variable `i`.

Wichtig: Sämtliche Variablen, die in den drei Ausdrücken auftreten, müssen vorher deklariert worden sein. Im Beispiel etwa durch die Deklaration `int i`. Natürlich müssen auch alle Variablen, die innerhalb einer Schleife auftreten, vorher deklariert worden sein. Und alle Variablen, welche auf der rechten Seite einer Zuweisung stehen (bzw. links der Zuweisungen += oder −=) müssen auch initialisiert worden sein. Das betrifft im Beispiel die Variable `summe`.

Noch eine Bemerkung: Die geschweiften Klammern können entfallen, falls die Schleife nur aus einer einzelnen Anweisung besteht. Gleichbedeutend mit unserem Beispiel ist also auch:

```
for(i=0; i<100; i++)
    summe += i;
```

Diese Schleife (ein Programm ist es nicht, weil der entsprechende Rahmen fehlt) summiert die Zahlen von 0 bis 99 auf: Angefangen bei `i=0` soll für alle `i` bis `i=99` jeweils das `i` zu `summe` addiert und schließlich um 1 erhöht werden.

Klassische Fehler:

- Vor dem Aufsummieren muss `summe` initialisiert werden! Für gewöhnlich mit Null.

- Innerhalb der `for`–Schleife darf `i` neu belegt werden. Man sollte aber wissen, was man tut. Gerne belegt man `i` neu, ohne zu bedenken, dass es die Kontrollvariable ist, die durch die Neubelegung nie das Abbruchkriterium erreicht. Falls doch dennoch nicht die gewünschte Rechenoperation durchführt. Man erreicht möglicherweise eine Endlosschleife, die man aber mit <Ctr>+<c> wieder abbrechen kann.

5.2 if–else–Verzweigung

Die wichtigste bedingte Verzweigung in der Sprache C ist die `if–else`–Anweisung. Mit ihrer Hilfe können in Abhängigkeit von einer Bedingung unterschiedliche Anweisungen ausgeführt werden. Zur Veranschaulichung ein einfaches Beispiel:

```
int i, j, m;

   ...

if (i > j)
    m = i;
else
    m = j;
```

Die Verzweigung hängt hier von dem Ausdruck "`i > j`" ab. Ist `i` tatsächlich größer als `j`, dann wird die Zuweisung `m = i;` ausgeführt, ansonsten die Zuweisung `m = j;`. Statt einer einzelnen Anweisung kann ein `if–else`–Teil auch einen Block mit mehreren Anweisungen enthalten:

```
int i, j, m;

    ...

if (i > j) {
    m = i;
    printf("i ist groesser als j  \n");
}
else {
    m = j;
    printf("i ist NICHT groesser als j \n");
}
```

Ist der Block nach dem `else`–Teil leer, so kann dieser Block und die `else`–Anweisung ganz entfallen.

Die `else–if` Anweisung ermöglicht es, mehrere Bedingungen hintereinander abzufragen und die Ausführung von Anweisungen davon abhängig zu machen. Eigentlich ist es keine neue Anweisung, sondern einfache Schachtelung von `if–else`–Anweisungen, aber die Verwendung ist sehr praktisch, und kann sehr übersichtlich eingegeben werden, ohne die Blöcke ständig einzurücken. Ein Beispiel:

```
int i, j, m;

    ...

if (i == j) {
    m = i;
    printf("i ist gleich j  \n");
}
else if (i < j) {
    m = j;
    printf("i kleiner als j \n");
}
else {
    m = 0;
    printf("i nicht gleich und nicht kleiner als j \n");
}
```

5.3 Vergleiche und logische Verknüpfungen

Im Allgemeinen kann jeder Ausdruck als Bedingung in einer `if`– Verzweigung auftreten, die Regel sind jedoch Vergleiche und logische Verknüpfungen mehrerer Vergleiche. Ein Ausdruck gilt dabei als "falsch", oder nicht zutreffend, falls sein numerischer Wert 0 ist. Andernfalls ist er "wahr".

Vergleiche haben einen geringeren Vorrang als arithmetische Operatoren, d. h. zuerst werden eventuelle arithmetische Operationen ausgeführt, und dann erst die Ergebnisse verglichen.

Logische Operatoren dienen dazu, mehrere Ausdrücke miteinander zu verknüpfen. Es gibt die beiden binären Operatoren `&&`, `||`, das logische "und" und das logische "oder". Dazu kommt noch der unäre Negationsoperator `!`. Ausdrücke, welche mit einem der ersten beiden Operatoren verknüpft werden, werden strikt von links nach rechts ausgewertet, und zwar nur solange, bis das Resultat der Auswertung feststeht. Dabei hat der Operator `&&` Vorrang vor `||`, und beide haben geringeren Vorrang als die obigen Vergleichsoperatoren.

5.4 goto und Textmarken

goto bietet die Möglichkeit von einer beliebigen Stelle im Programm zu einer beliebigen anderen, mit einer sogenannten "Textmarke" sozusagen markierten Stelle zu springen.

```
goto markierung;
...
/* hier geht's dann weiter */
markierung:
```

Die Marke wird immer mit einem anschließenden Doppelpunkt versehen und hat ihren Gültigkeitsbereich in der gesamten Funktion, in der die goto-Anweisung erfolgt. Die Regeln für gültige Textmarkennamen sind die selben, wie für Variablennamen.

Achtung! Formal ist eine goto-Anweisung nicht notwendig und kann immer durch andere Möglichkeiten erreicht werden. Da man auf diese Weise äußerst unübersichtliche "Spaghetti-Programme" schreiben kann, sollte man die goto-Anweisung wirklich nur in "Notfällen" verwenden...

Ein solcher Fall wäre zum Beispiel der gleichzeitige Abbruch mehrerer verschachtelter Schleifen:

```
for(...)
  for(...){
    ...
      if( Katastrophe ) goto error;
    ...
  }
}
...
error:
  ...
```

5.5 Die switch-Anweisung

Die switch-Anweisung ermöglicht eine Programmverzweigung in Abhängigkeit eines Ausdrucks, welcher einen ganzzahligen Wert haben muss. Dazu ein kleines Beispiel:

```
#include <stdio.h>
```

```
int main(void)
{
  int a;

  printf("Gib einen ganzzahligen Wert fuer a an:\n");
  scanf("%d",&a);

  switch(a){
    case 0:
      printf("a hat den Wert 0. \n");
      break;
    case 1:
      printf("a hat den Wert 1. \n");
      break;
    case 2:
      printf("a hat den Wert 2. \n");
      break;
    default:
      printf("a hat weder den Wert 0 noch 1 noch 2. \n");
      break;
  }
}
```

Hat a einen der Werte in den case-Marken, so werden die entsprechend folgenden Anweisungen ausgeführt. Mit dem Befehl break oder return wird die switch-Anweisung abgebrochen. Trifft keiner der case-Fälle zu so wird, falls vorhanden, bei der default-Marke fortgefahren. Die default-Marke ist optional.

Achtung! Eine case- oder default-Marke ist nichts anderes als eine spezielle Textmarke. Ohne die break-Anweisung oder eine ähnliche Anweisung um switch zu verlassen, werden ab der Marke, deren "Wert" mit a übereinstimmt, alle folgenden Anweisungen ausgeführt. Sollen nur die Anweisungen hinter der zutreffenden case- bzw. default-Marke bis hin zur folgenden Marke ausgeführt werden, so muss switch mit einer entsprechenden Rücksprunganweisung abgebrochen werden. Im Beispiel wird das jeweils durch die Anweisung break erledigt. Es können daher mehrere case-Marken vor einer Anweisung stehen:

```
switch(a){
  case 0:
  case 1:
  case 2:
    printf("a hat den Wert 0, 1 oder 2.\n");
    break;
  default:
    printf("a hat weder den Wert 0 noch 1 noch 2.\n");
```

```
    break;
}
```

Die switch-Anweisung ist letztlich nichts anderes als eine - in manchen Fällen elegantere - Variante langverketteter if-else-goto-Anweisungen. Obiges Beispiel entspräche dann folgender if-else-Schleife:

```
if(a==0 || a==1 || a==2)
  printf("a hat den Wert 0, 1 oder 2.\n");
  else
    printf("a hat weder den Wert 0 noch 1 noch 2.\n");
```

5.6 Die while-Schleife

Die while-Schleife führt Anweisungen aus, solange ein bestimmter Ausdruck 'wahr' ist. Die Syntax hierzu lautet while(<Ausdruck>){ <Anweisungsblock> } oder als Variante auch do{ <Anweisungsblock> } while(<Ausdruck>). Zum Beispiel führt

```
int i;

...

while(i<10){
  printf("i ist kleiner als 10\n");
  i++;
}
```

solange aus was in den geschweiften Klammern der Schleife angewiesen ist, solange i<10 erfüllt ist. Man könnte obige while-Schleife auch genausogut in eine for-Schleife setzen. Nämlich so:

```
int i;

  ...

for(i;i<10;i++)
    printf("i ist kleiner als 10\n");
```

Ob man nun while- oder for-Schleifen bevorzugt ist Geschmackssache. Sind Anfang und Ende des Schleifendurchlaufes, wie im obigen Beispiel a priori gegeben, so bietet sich die for-Schleife an, da in der ersten Zeile sofort ersichtlich ist, wieviel Schleifendurchläufe erwartet werden. Ein Beispiel, bei dem es natürlicher erscheint, die while-Schleife zu verwenden, könnte so aussehen:

```
  /* TOL vom Typ double sei eine gegebene Toleranzgroesse */

while(err<TOL){

  /* Loese ein Problem approximativ (etwa ein lineares
     Gleichungssystem mittels iterativem LGS-Loeser) mit
     startwert uold. Die berechnete Loesung nennen wir u */

  loeser(uold,u);

  /* Berechne den Fehler zwischen der berechneten
     Loesung und der bekannten exakten Loesung, die
     wir hier uex nennen wollen. */

  err = fehler(u,uex);

  uold = u;
}
```

Hierbei liefere `loeser()` eine "bessere" Lösung je "besser" der Startwert `uold` gewählt wurde.

Bei der dritten Schleifenkonstruktion `do-while` wird im Gegensatz zur `for`- bzw. der `while`-Schleife das Abbruchkriterium am Ende der Schleife geprüft. Das bedeutet, dass der Anweisungsblock wenigstens einmal ausgeführt wird.

Achtung! Bei der `do-while`-Schleife muss bei nur einer Anweisung wie üblich die geschweifte Klammer nicht gesetzt werden. Der Übersicht halber ist es jedoch besser diese zu setzen, da sonst beim flüchtigen Lesen `while` für den Anfang einer `while`-Schleife gehalten werden könnte.

5.7 `break` und `continue`

Bereits in der `switch`-Anweisung haben wir `break` kennengelernt. Mit dieser Anweisung kann man auch Schleifen (`for`- `while`- und `do`-Schleifen) an einer beliebigen Stelle, nicht nur zu Beginn oder am Ende, abbrechen. Bei mehrfach geschachtelten Schleifen wird dann die direkt umgebende verlassen.

Außer in der `switch`-Anweisung kann die `continue`-Anweisung ähnlich wie die `break`-Anweisung verwendet werden. `continue` bricht die nächst äußere Schleife nicht komplett ab sondern unterbricht lediglich den aktuellen Schleifendurchlauf, um dann direkt zum nächsten überzugehen. Bei `while` und `do` wird sofort die Bedingung neu bewertet und bei der `for`-Schleife direkt die Reinitialisierung durchgeführt. `continue` hat innerhalb einer `switch`-Anweisung auf diese keine Auswirkung, sondern auf die `switch` umgebende nächste Schleife.

–6–
Zeiger

Zeiger ("Pointer") sind Variablen, welche die Speicheradresse einer Variablen enthalten. Das richtige Verständnis und die Benutzung von Zeigern ist für die gute C–Programmierung wichtig. Zum Teil, weil manchmal Zeiger die einzige Möglichkeit sind, um eine bestimmte Aufgabe umzusetzen (mittels dynamische Speicherverwaltung etwa), zum Teil, weil sie meist zu kompakteren und effizienteren Programmen führen. Zeiger sind eines der mächtigsten Werkzeuge in C, aber auch eines der gefährlichsten: Zeiger sind eine hervorragende Methode unverständliche Programme zu schreiben. Fehler bei der Verwendung von Zeigern sind oft sehr schwierig zu finden. Mit Sorgfalt eingesetzte Zeiger können aber durchaus zu klaren und leicht verständlichen Programmen führen.

6.1 Speicherorganisation

Bevor wir näher auf Zeigervariablen eingehen, befassen wir uns kurz mit einem vereinfachten Bild der Speicherorganisation. Ein typischer Rechner hat eine bestimmte Menge Speicher ("RAM") zur Verfügung. Die einzelnen Speicherzellen sind in einem langen Vektor angeordnet und fortlaufend numeriert. Siehe Abbildung 1. Die Nummer oder Adresse einer Speicherzelle (eines "Bytes") dient dazu, gezielt auf dieselbe zugreifen zu können. Abgesehen von Variablen des Typs `char` benötigt eine Variable mehrere hintereinanderliegende Bytes, um gespeichert zu werden. Ein Zeiger ist nun eine solche Gruppe von Speicherzellen, die eine eindeutige Speicheradresse aufnehmen kann. Ist z. B. `p` eine Zeigervariable, die auf eine Variable `c` verweist, dann enthält `p` die Adresse von `c`. Diese entspricht der ersten Speicherzelle, die für die Speicherung von `c` verwendet werden. Man sagt: "`p` zeigt auf `c`." Es erweist sich als äusserst nützlich, wenn man mit einem Zeiger nicht nur die Adresse der ersten Speicherzelle der Variable kennt, auf die der Pointer zeigt, sondern auch über den Variablentyp informiert ist. Damit ist dann klar, wieviele Speicherzellen "hinter" der Adresse im Zeiger zu einem festen Speicherbereich gehören, und ab welcher Adresse ein neuer Speicherbereich beginnt. Ausserdem ergibt sich mit dieser Information eine enge Verbindung von Zeigern und Feldern, wie wir weiter unten sehen werden.

6.2 Zeiger-Variablen

Wie oben gesagt, ist ein Zeiger eine Variable, die eine Speicheradresse (normalerweise die einer anderen Variablen) enthält. Die Deklaration eines Zeigers besteht aus einem der bekannten Basistypen (`char, int, float, double`), einem `*` und dem Namen der Variablen:

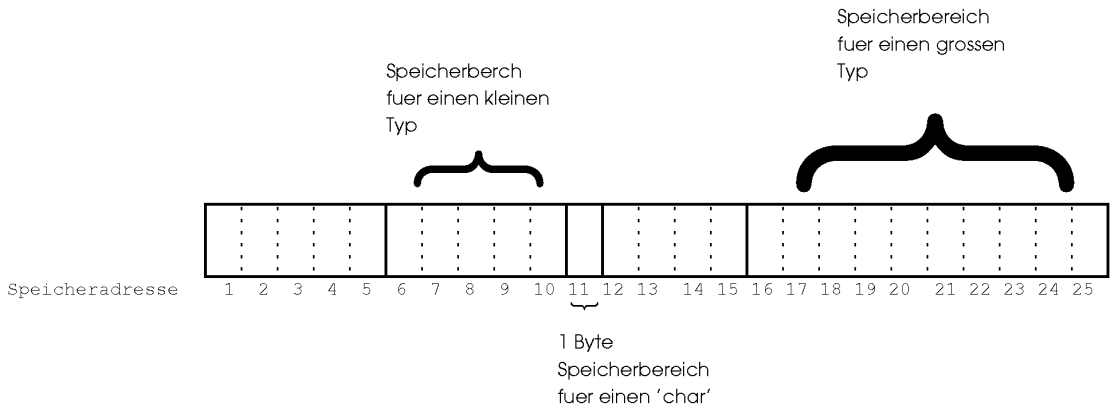


Abbildung 1: Sehr grobe Darstellung der Speicherorganisation im RAM

```
int *i;
float *x, *y;
```

Der Basistyp legt fest, auf welchen Typ von Variable der Zeiger zeigt. Ein Zeiger vom Typ `int*` darf immer nur auf Variablen vom Typ `int` zeigen. Ähnlich wie bei Zahlen die "0", gibt es für Zeiger eine Art "neutralen" Wert: Dann nämlich, wenn der betreffende Zeiger auf **keine** Speicheradresse zeigt. Man bezeichnet diesen Wert mit `NULL` oder `nil`. Damit die symbolische Konstante `NULL` zur Verfügung steht, muss die Zeile "`#include <stdlib.h>`" zu Beginn der Datei erscheinen. Zeiger und `int` Werte sind normalerweise nicht austauschbar. Einzige Ausnahme ist die Integerzahl "0", die man einem Zeiger zuweisen darf. Dieser Wert entspricht dann der (symbolischen) Konstanten `NULL`. Der Zeiger zeigt "nirgendwo" hin.

6.3 Die Zeiger–Operatoren

Es gibt zwei spezielle Zeiger–Operatoren: `*` und `&`. Beide sind unärer Operatoren, sie erwarten nur einen Operanden als Argument. Den zweiten, `&`, haben wir bislang schon bei den Argumenten der Funktion `scanf` benutzt. Dieser Operator liefert die Speicheradresse seines Operanden, z. B. legt

```
float *p;
float a, b;

p = &a;
```

die Speicheradresse der Variablen `a` in `p` ab. Man sagt auch: Der Operator `&` gibt die Adresse von `a` zurück.

Der andere Zeiger–Operator, `*`, ist die Umkehrung von `&`. Er liefert den Inhalt der Speicheradresse, auf die er angewendet wird. Wenn `p` die Speicheradresse der Variablen `a` enthält, legt

```
b = *p;
```

den Wert von a in b ab.

Beispiel: Die Variable a habe die Speicheradresse 7356 und den Wert 20.4. Nach der Zuweisung

```
p = &a;
```

hat p den Wert 7356 und nach

```
b = *p;
```

hat b den Wert 20.4, da dieser in der Adresse 7356 gespeichert ist. Vergleiche die Abbildung 2. **Achtung!** Die Speicheradresse von b taucht bei den obigen Zuweisungen nicht auf, im Bild ist b etwa an der Adresse 8012 gespeichert.

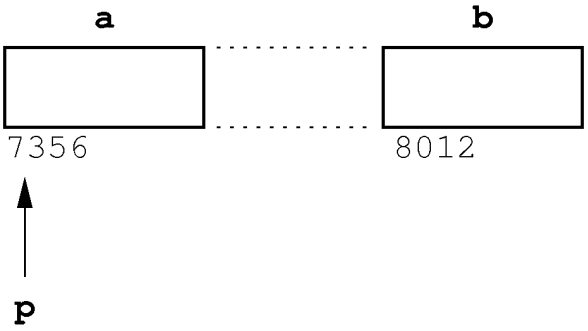


Abbildung 2: Der Zeiger p und die Variablen a und b im Speicher

6.4 Zuweisung von Zeigern

Wie bei anderen Variablen können auch Zeigern Werte zugewiesen werden. Betrachte folgendes Beispiel:

```
#include <stdio.h>
main()
{
    int i;
    int *p1, *p2;

    p1 = &i;
    p2 = p1;    /* Zuweisung der Adresse, die
                 * in p1 abgelegt ist, an p2  */
```

```
    i = 1524;

    printf("Die Adresse von i ist %p und der Wert von i ist %d\n", p2, *p1);
}
```

Die Formatangabe `%p` steht dabei für die Ausgabe einer Speicheradresse.

6.5 Zeiger und Felder, Adressarithmetik

Es gibt eine enge Verbindung zwischen Zeigern und Feldern. Jede Operation mit Feldindizes kann auch mit Zeigern formuliert werden. Dabei ist die Zeigerversion meist effizienter, allerdings schwieriger zu lesen. Die arithmetischen Operatoren `+`, `-`, `++` und `--` können auch auf Zeiger angewendet werden, solange ein Operand ein Zeiger, und der andere eine ganze Zahl ist. Wie die sogenannte "Adressarithmetik" zu verstehen ist, illustriert das Folgende.

Mit

```
    float a[3];
```

wird ein dreidimensionaler `float`–Vektor deklariert, d. h. einen zusammenhängenden Speicherblock mit Platz für drei Werte vom Typ `float`, auf die über die Namen `a[0]`, `a[1]` und `a[2]` zugegriffen werden kann. Durch die Zuweisung

```
    float *p;

    p = &a[0];
```

enthält der Zeiger `p` die Speicheradresse von `a[0]`. Zeigt `p` auf ein bestimmtes Element des Vektors `a`, so zeigt nach Definition `p + 1` auf das nachfolgende Element des Vektors `a`. Hat wie im Beispiel `p` den Wert `&a[0]`, dann hat `p + 1` den Wert `&a[1]`. Analog ergibt sich für `p + 2` der Wert `&a[2]`. Siehe Abbildung 3. **Vorsicht!** Der Ausdruck `p + 3` ist durchaus ein gültiger Ausdruck, allerdings entspricht der Wert dieses Ausdrucks einer Speicheradresse, die nicht mehr zum Vektor `a` gehört.

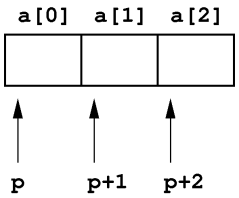


Abbildung 3: Vektor `a` und Zeiger `p` im Speicher

Ähnlich wie in diesem Beispiel arbeiten auch die anderen arithmetischen Operatoren. Damit dies funktioniert, ist bei der Deklaration von Zeigern der richtige Basistyp wichtig: Eine

Anweisung wie das obige `p + 2` wird als "gehe von der Speicheradresse, auf die `p` zeigt, um zwei `float`–Speicherbereiche weiter, `p--;` ändert den Wert von `p` dahingehend ab, dass `p` **nach** dieser Anweisung auf die Adresse "einen `float`–Speicherbereich vor der Adresse, auf die `p` bislang zeigte" ab.

Adressarithmetik beinhaltet noch mehr, als die Operationen wie die Addition (oder Subtraktion) einer ganzen Zahl zu (von) einem Zeiger. Zeigen die beiden Pointer `p` und `q` auf Elemente desselben Vektors, dann funktionieren die logischen Vergleiche `q < p`, `q == p`, etc. So ist etwa der Ausdruck `q < p` dann wahr, wenn `q` auf ein Element zeigt, welches **vor** dem Element kommt, auf welches `p` zeigt. Ist dies so, dann ist der Ausdruck `p - q + 1` gleich der Anzahl von Elementen von `q` bis `p` einschliesslich. Die Adressarithmetik ist dabei konsistent, sie berücksichtigt die tatsächliche Größe der zugrundeliegenden Basistypen.

Warnung! Ausdrücke wie `p - q` oder `p == q` sind formal korrekt, auch wenn die Zeiger `p` und `q` auf Elemente in verschiedenen Feldern zeigen. Dann allerdings sind die Ergebnisse unbestimmt...

Erlaubte Operationen mit Zeigern sind: Zuweisung von Zeigern des gleichen Basistyps, Addition oder Subtraktion einer ganzen Zahl von einem Zeiger, Subtraktion oder Vergleich zweier Zeiger auf Elemente des gleichen Vektors sowie Zuweisung von oder Vergleich mit `NULL`. Jede andere Zeigerarithmetik ist verboten, insbesondere dürfen zwei Zeiger nie addiert oder multipliziert werden (erst recht nicht dividiert...).

Eine äquivalente Schreibweise für `p+i` veranschaulicht den Zusammenhang zwischen Zeigern und Feldern noch deutlicher: `p[i]` entspricht dabei der Adresse `p+i`. Die Äquivalenz wird sogar noch weitergehen, vergleiche den Tip zur Programmieraufgabe weiter unten.

6.6 Dynamische Vergabe von Speicherplatz

Mit der Hilfe von Zeigern hat man die Möglichkeiten dynamischen Speicherplatz zu vergeben. Dies bedeutet, dass das Programm den Speicherplatz für Variablen zur Laufzeit zugewiesen bekommen und die Größe dieses Bereiches verändern kann. Damit ist es z.B. möglich, die Größe eines Feldes erst während der Laufzeit eines Programmes festzulegen. Wir erinnern uns an die Matrix–Vektor–Multiplikation und die Programme zum Gaussalgorithmus: es wäre also möglich, die Dimension über die Tastatur einzugeben und die Größe der Felder zur Laufzeit entsprechend festzusetzen. Ein deutlicher Vorteil zu unserer bisherigen Methode über die Präprozessorvariable `DIM` und ein zwingend notwendiges neues Übersetzen, wenn diese geändert wurde.

Die Funktion `malloc()` (**m**emory **alloc**ate) dient dazu, Speicherplatz für ein Objekt (etwa einen Vektor) während der Laufzeit anzufordern. Sie erwartet als Argument die Größe des angeforderten Speicherbereichs, und liefert als Ergebnis einen Zeiger auf den Beginn des Bereichs zurück. Im Allgemeinen hängt die erforderliche Größe vom speziellen Typ der Variablen (oder des Feldes) und dem benutzten Computersystem ab. Die Funktion `sizeof()` erleichtert es, auf die Größe eines bestimmten Types im betreffenden System zuzugreifen; `sizeof(int)` liefert etwa die Größe einer `int`–Variablen zurück. Zur Illustration ein Beispiel:

```
main()
{
    int i, dim;
    double *f;   /* Zeiger auf ein Variable vom Typ double */

    dim = 300;   /* lege die Groesse fest */


    /* dynamische Zuweisung von Speicherplatz */
    f = (double *) malloc(dim * sizeof(double));
    if (f == NULL)
        exit(1);

    for (i = 0; i < dim; i++)
        f[i] = 2.0;
    free(f);
}
```

Hier wird ein `dim`–dimensionaler `double`–Vektor `f` angelegt und mit `2.0` in jeder Komponente initialisiert.

Dazu deklarieren wir zunächst einen Zeiger auf eine Variable vom Typ `double`. Zu diesem Zeitpunkt hat dieser Zeiger noch keinen sinnvollen Inhalt. In der folgenden Zeile wird die Dimension des Vektors festgelegt, die z.B. auch von der Tastatur mit `scanf()` eingelesen werden könnte. Schließlich wird der Speicherplatz mit der Funktion `malloc()` reserviert.

In diesem Fall wollen wir für `dim double`– Werte Speicherplatz haben. Der Aufruf `sizeof(double)` liefert dabei die Speichergröße, die für einen `double`–Wert benötigt wird. `malloc()` gibt einen Wert zurück, nämlich die Startadresse des Speicherbereichs, wenn die Speicherreservierung erfolgreich war, ansonsten ein `NULL`-Zeiger. Der Basistyp des Rückgabewerts von `malloc()` ist unbestimmt (das wurde bei der Deklaration dieser Bibbliotheksfunktion durch `void * malloc(size_t size)` bewirkt. Allgemein gilt: Eine Variable vom Typ `void` ist von "unbestimmten" Typ...) Damit wir diese Adresse unserem (noch „leeren\) Zeiger `f` zuweisen können, muss daraus ein Zeiger vom Typ `double*` gemacht werden. Dieses geschieht durch Einfügen des "Cast" `(double*)` vor die Funktion `malloc()`. Nach dieser Zuweisung zeigt unser `f` auf einen Speicherbereich, der genau `dim double's` aufnehmen kann, oder er enthält einen `NULL`-Zeiger. Diesen Sonderfall sollte man sicherheitshalber immer abfragen, Fehler, die sonst später in Verbindung mit dem Pointer `f` auftreten können, sind sonst deutlich schwieriger zu finden sind. Mit `f` kann jetzt wie mit einem klassischem Vektor weiter gearbeitet werden, Zugriffe auf das `i`–te Element von `f` erfolgen mit `f[i]`. Die Adresse `&f[i]` entspricht dabei genau dem Wert von `f+i`.

Achtung! Nach der Nutzung muss der Speicherbereich, der mit `malloc()` reserviert wurde, mit dem entgegengesetzten Befehl `free()` wieder freigegeben werden. Vergisst man dies, so betrachtet das System diesen Speicherbereich mindestens bis zur Beendigung

des Programmes als belegt. "Speicherfresser"...

6.7 Matrizen als Zeiger

Eindimensionale dynamische Felder funktioniert nach dem Obigen also genau wie die bisher von uns benutzten statische eindimensionalen Feldern. Bei mehrdimensionalen Feldern, wie z.B. bei Matrizen wird es etwas komplizierter. Die Größe des Speicherbereichs zu bestimmen ist kein Problem, da man in der Regel die Dimension der Matrix kennt. Schwierig wird aber der Zugriff auf die Werte in der Matrix: Ausdrücke der Form `A[i][j]` sind nicht möglich, wenn der Speicherbereich dynamisch vergeben wurde.

Wenn man einen Speicherbereich mit

```
A = (double*) malloc( n*m*sizeof(double)
```

reserviert, kann dieser vom Platz her eine $n \times m$-Matrix von `double`-Werten aufnehmen. Das Problem ist, daß der Rechner diesen Speicherbereich nicht als eine Matrix sondern als ein eindimensionales Feld der Länge $n * m$ betrachtet. Deshalb ist ein Zugriff auf das Element a_{ij} mit `A[i][j]` nicht möglich. Um trotzdem auf die Elemente zugreifen zu können muß man sich eine Matrix nicht als Zeilen untereinander, sondern als alle Zeilen hintereinander vorstellen. Damit kann man über `A[(i*n) + j]` auf a_{ij} zugreifen. Genauso funktioniert es natürlich auch mit Spalten anstellen den Zeilen hintereinander geschrieben. Es muß problemabhängig entschieden werden welche Darstellung besser geeignet ist.

Eine Standardmethode um eine dynamische Matrix `A` zu erhalten, auf deren Einträge wie gewohnt mit `A[i][j]` zugegriffen werden kann, ist diese:

```
double **a;
double *a0;

int m, n, i;

a = malloc(m*sizeof(double*));
a0 = malloc(n*m*sizeof(double));
for (i=0; i<m; i++)
{
  a[i] = a0 + i*n;
}
```

Dabei ist `a` als ein Zeiger auf einen Zeiger auf `double` deklariert, (sowas geht auch!) der auf ein eindimensionales Feld mit den Zeiger auf die einzelnen Zeilen der Matrix zeigt. Mit der Anweisung `a = malloc(m*sizeof(double*));` wird Speicherplatz für alle benötigten

Zeilenzeiger allociert. Der Pointer a0 zweigt auf einen Speicherbereich, der die eigentliche Matrix (als eindimensionales Feld) enthält. Durch die `for`–Schleife werden die Zeilenzeiger aus dem Zeigerfeld a mit den "richtigen" Werten, nämlich den Adressen der Zeilenanfänge im Feld a0 initialisiert. Ab diesem Augenblick kann mit `a[i][j]` auf den Matrixeintrag A_{ij} zugegriffen werden.

Wie oft kommt wohl die Zeichenfolge "zeig" bzw. "Zeig" in diesem Kapitel vor? Hmm.

–7–
Strukturierte Programmierung

Funktionen sind die Bausteine, aus denen ein C–Programm zusammengesetzt ist. Sie führen bestimmte Aufgaben aus und besitzen i.A. Eingabe– und Ausgabewerte. Meist sind Funktionen für genau eine Aufgabe geschrieben, ein C–Programm wird eher aus vielen kleinen Spezialfunktionen als aus wenigen großen bestehen. Auch das Hauptprogramm selbst ist in C eine Funktion, die den Namen `main` trägt. Weitere Funktionen aus "externen" Bibliotheken sind uns auch schon begegnet, so etwa die Funktion `printf`.

7.1 Funktionen als Programmeinheiten

Funktionen ermöglichen es, Programme in kleinere, selbständige Einheiten zu zerlegen und so die Übersichtlichkeit zu steigern. Außerdem können einzelne Funktionen auch in anderen Programmen wiederverwendet werden. Genau wie die Variablen, welche in einem Programm benutzt werden, müssen prinzipiell alle benutzten Funktionen vor ihrem ersten Aufruf dem Compiler bekannt gemacht werden. Analog zu der Situation bei Variablen, geschieht dies i. A. durch die Deklaration der Funktion. Neben der Deklaration, die **vor** dem ersten Aufruf erfolgt sein muss, benötigt man selbstverständlich auch die Definition der Funktion, d. h. den eigentlichen Code, der bei einem Funktionsaufruf ausgeführt werden soll. Definition und Deklaration kann zusammen erfolgen (ähnlich wie Deklaration und Initialisierung von Variablen). Im folgenden Beispiel werden die Funktionen `min` und `max` deklariert und gleichzeitig definiert und schließlich im Hauptprogramm `main` aufgerufen.

```
#include <stdio.h>

double max(double x1, double x2)
{        /* Definition der Funktion max */
    if (x1 > x2)
        return (x1);
    else
        return (x2);
}

double min(double x1, double x2)
{        /* Definition der Funktion min */
    if (x1 > x2)
        return (x2);
    else
```

```
        return (x1);
}

int main( void )
{        /* Definition der Funktion main */
    float zahl1, zahl2;

    printf("Gebe zwei Zahlen ein: ");
    scanf("%f %f", &zahl1, &zahl2);

    printf("Das Maximum ist %f\n", max(zahl1, zahl2));
    printf("Das Minimum ist %f\n", min(zahl1, zahl2));

    return 0;
}
```

Der Nachteil der Kombination von Definition und Deklaration ist, dass die Funktionen vor ihrem ersten Aufruf im Programmcode definiert sein müssen. Bei größeren Programmen ist die dadurch bedingte Einhaltung der Reihenfolge nur noch schwierig zu überblicken. Die Übersicht über vorhandene Funktionen ist ebenfalls problematisch. Es bietet sich daher eine Trennung von Deklaration und Definition an:

Die Funktionsdeklaration wird dem Programm vorangestellt, dadurch wird die Funktion dem Compiler bekannt gemacht. Die Funktionsdefinition kann dann an beliebiger Stelle erfolgen. Die Deklaration beinhaltet den Typ des Ergebnisses der Funktion, den Namen der Funktion und die Liste der Funktionsargumente jeweils mit Typ und Namen. Damit sieht das obige Beispiel so aus:

```
#include <stdlib.h>

double max(double, double);      /* Deklaration der Funktion max */
double min(double, double);      /* Deklaration der Funktion min */

/* Jetzt das Hauptprogramm main: */

main()
{
    float zahl1, zahl2;

    printf("Gebe zwei Zahlen ein: ");
    scanf("%f %f", &zahl1, &zahl2);

    printf("Das Maximum ist %f\n", max(zahl1, zahl2));
    printf("Das Minimum ist %f\n", min(zahl1, zahl2));
```

```
}

/* Und hier nun die Definition der oben deklarierten Funktionen: */

double max(double x1, double x2)
{        /* Definition der Funktion max */
    if (x1 > x2)
        return (x1);
    else
        return (x2);
}

double min(double x1, double x2)
{        /* Definition der Funktion min */
    if (x1 > x2)
        return (x2);
    else
        return (x1);
}
```

7.2 Variablen in Funktionen

In jeder Funktion können Variablen genauso deklariert werden, wie in der Funktion `main`. Es stellt sich dabei die Frage über die "Sichtbarkeit" von Variablen: Ist eine Variable `x`, welche in einer Funktion deklariert wurde, auch in einer anderen Funktion bekannt? Die Antwort lautet: Nein. Es gilt die einfache Faustregel: Variablen gelten nur in dem Block, in dem sie deklariert sind (erkennbar an den geschweiften Klammern { }) . Außerhalb dieses Blockes sind sie nicht bekannt. Man nennt Variablen, die in einem Block definiert wurden, lokale Variablen. Sie unterstützen die Möglichkeit, Funktionen als unabhängige Unterprogramme zu schreiben. Es ist damit beispielsweise auch möglich, in verschiedenen Funktionen den selben Variablennamen zu verwenden, trotzdem werden damit verschiedene Variablen deklariert, d.h. die Variablen in der einen Funktion werden bei einer Änderung der Variablen in der anderen Funktion nicht mitverändert. Manchmal benötigt man aber Variablen, die in verschiedenen Funktionen gleichzeitig bekannt sind, und tatsächlich dieselbe Variable darstellen. Dies erreicht man etwa durch die Verwendung globaler Variablen. Solche globale Variablen werden außerhalb jedes Blockes, meistens gleich am Beginn der Datei, deklariert. Wie der Name schon sagt, sind diese global gültig, d. h. im Code ab der Deklaration überall ansprechbar, ohne innerhalb einer Funktion noch einmal extra deklariert werden zu müssen. Im Gegenteil, eine globale Variable **darf** nur einmal deklariert werden.

Warnung: Manchmal stellen globale Variablen eine Vereinfachung dar, aber sie erschweren es, die genaue Funktionsweise einer Funktion zu durchschauen, welche eine globale Variable benutzt, ohne das gesamte Programm zu kennen. Deswegen sollte man globale Variablen

nur nach reiflicher Überlegung einsetzen.

7.3 Rückgabewerte

Bei der Deklaration und der Definition einer Funktion muss der Typ ihres Rückgabewertes angegeben werden, wie z.B. `char, int, float, double`. Hinzu kommt der nur bei Funktionen und Zeigern erlaubte Typ `void`, mit welchem eine Funktion gekennzeichnet wird, die **keinen** Rückgabewert hat.
Die Rückgabe erfolgt mit der `return()`–Anweisung. Fehlt diese, dann wird die Funktion nach der letzten Anweisung beendet und ein undefinierter Wert zurückgegeben. Fehlt der Ausdruck in der `return()`–Anweisung, wird ebenfalls ein undefinierter Wert zurückgegeben. Soll eine Funktion keinen Wert zurückgeben, muss der Typ `void` als Typ ihres Rückgabewertes vereinbart werden. Sollen mehrere Ergebnisse zurückgegeben werden, kann man das mittels der Parameterübergabe $call\ by\ reference$ tun, siehe übernächsten Abschnitt.

7.4 Parameterübergabe

7.4.1 Standardfall: $call\ by\ value$

Beim Aufrufen einer Funktion werden im Normalfall nur die Werte der Parameter übergeben. Im obigen Beispiel sind das die beiden Werte von `zahl1` und `zahl2`. Im Inneren der Funktionen sind die Namen der beiden Variablen `x1` und `x2`.

Beachte: Variablen, die im Inneren von Funktionen oder bei der Deklaration einer Funktion definiert werden, sind nur innerhalb dieser Funktion sichtbar. Es wäre auch möglich im obigen Beispiel die Namen im Hauptprogramm und in den beiden Funktionen gleich zu wählen.

Bei der Übergabe von Parametern kann die Funktion nur lesend, aber nicht schreibend auf diese zugreifen. Dies hat den Vorteil, dass keine unbeabsichtigten Änderungen von Variablenwerten durch Funktionsaufrufe auftreten können. In unserem Beispiel können die Werte von `zahl1` und `zahl2` innerhalb der Funktionen `min` und `max` nicht verändert werden.

7.4.2 Parameterübergabe per Verweis, $call\ by\ reference$

Mit Zeigern als Funktionsparameter können in C Adressen und damit Verweise auf die Variablen an Funktionen übergeben werden. Dieses Vorgehen ($call\ by\ reference$) ermöglicht es schreibend auf Variablen zuzugreifen, und so die Werte von Argumenten auch innerhalb von Funktionen zu verändern. Die Übergabeparameter müssen in diesem Fall explizit als Zeiger vereinbart werden und beim Funktionsaufruf dann die Adressen der Variablen angegeben werden. Ein Beispiel illustriert den Unterschied: Eine Funktion wie diese:

```
void swap(int i, int j)
{
```

```
  int temp;

  temp = i;
  i = j;
  j = temp;
}
```

die zwei Elemente miteinander vertauschen soll, würde dies nicht tun, sondern nur zwei Kopien der Argumente vertauschen. Möchte man tatsächlich mit den Variablen selbst arbeiten (und im obigen Fall ist dies wahrscheinlich...), so müssen Zeiger auf die Variablen übergeben werden. Damit würde die Funktion dann so aussehen:

```
void swap(int *ip, int *jp)
{
  int temp;

  temp = *ip;
  *ip = *jp;
  *jp = temp;
}
```

7.5 Ein– und Ausgabeparameter der Funktion `main()`

Oft ist es nützlich, Parameter direkt beim Start des Programmes in der Kommandozeile zu übergeben. In C gibt es zwei spezielle Parameter, mit deren Hilfe dies bewerkstelligt werden kann. Sie heißen `argv` und `argc`. Der Parameter `argc` enthält die Anzahl der Argumente, die von der Kommandozeile übernommen werden und ist vom Typ `int`. Der Wert von `argc` ist mindestens 1, da der Name des Programms bereits der erste Parameter ist. Der `argv`–Parameter ist ein Feld von Zeigern auf Zeichenfelder (auch strings genannt, Typ `char *`). Jedes Element dieses Feldes zeigt auf einen der Kommandozeilenparameter. Alle Parameter, auch Zahlwerte, werden als Zeichenketten abgelegt und müssen vom Programm in das entsprechende Format umgewandelt werden. `argv` muss richtig deklariert werden, normalerweise `char *argv[]`. Die leeren Klammern bedeuten, dass die Länge des Feldes unbestimmt ist. Damit kann nun jedes Element adressiert werden. So zeigt zum Beispiel `argv[0]` auf die erste Zeichenkette, den Namen des Programms selbst und `argv[1]` auf den ersten Parameter. Die Umwandlung von Zahlparametern in das richtige Format kann mit der Funktion `atoi()` bzw. `atof()` vorgenommen werden. `atoi()` wandelt eine Zeichenkette in einen Ganzzahlwert um (ASCII to Integer) und `atof()` in einen Gleitkommawert (`float`). Um diese Funktionen benutzen zu können, muss die Header-Datei `<stdlib.h>` eingebunden werden.
Betrachte folgendes Beispiel:

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

```
#include<stdio.h>

int main(int argc, char *argv[])
{
  int age;
  double heigth;
  char namen[80];

  if (argc == 4) {
    strcpy(namen, argv[1]);
    age = atoi(argv[2]);
    heigth = atof(argv[3]);
    printf("Hallo %s, Du bist %d Jahre alt und %f m gross \n",
           namen, age, heigth);
  } else
    printf("Bitte Namen, Alter und Groesse als Parameter uebergeben\n");

  return 0
}
```

Dieses Programm kann z.B. mit

```
name Stefan 28 1.83
```

gestartet werden. Die Parameter werden im Programm weiter verarbeitet. Dabei ist `Stefan` eine Zeichenkette (string), 28 wird in ein `int` umgewandelt und `1.83` in eine `double`. Zu der hier verwendeten Funktion `strcpy` lese man die `man`–Page mit dem Kommando `man strcpy`.

7.6 Zeiger auf Funktionen

Bislang haben wir Zeiger auf Variablen und Strukturen kennengelernt. In C ist es aber auch möglich, mit Zeigern auf Funktionen zu arbeiten. So könnte ein Unterprogramm `quadratur` zur numerischen Integration einer gegebenen Funktion als Parameter zum Beispiel die untere und obere Integrationsgrenze und einen Zeiger auf den Integranden haben und damit zur Integration unterschiedlicher Funktionen verwendet werden. Ein Beispiel für einen Zeiger auf eine Funktion:

```
double (*f)(double);
```

Dies deklariert einen Zeiger `f` auf eine Funktion, welche ein Argument vom Typ `double` hat, und einen `double`–Wert zurückliefert. Dabei ist dringend auf die Klammerung `(*f)` zu

achten! Ohne die Klammerung würde mit der obigen Deklaration eine Funktion `f` deklariert, mit einem `double` Argument und dem Rückgabewert "Zeiger auf `double`". Die Zuweisung eines Wertes an die Funktionszeigervariable `f` und ein Funktionsaufruf der durch diesen Zeiger bezeichneten Funktion geschieht wie folgt:

```
float u(float );           // Deklaration einer Funktion u

main()
{
    float y, z, x=2.;
    float (*f)(float );

    f = u;                  // f zeigt jetzt auf u

    y = u(x);               // y hat den Wert 4.
    z = f(x);               // z hat ebenfalls den Wert 4.
}
                                  // Definition der Funktion u=x^2
float u(float x){return pow(x,2.);}
```

Nach der Zuweisung unterscheidet sich der Aufruf einer "Zeigerfunktion" also nicht mehr von einem üblichen Funktionsaufruf.

Die Deklaration einer Funktion, die einen Zeiger auf eine weitere Funktion und einen float–Wert übergeben bekommt sieht so aus:

```
float g(float (*)(float ), float );
```

Und die Definition:

```
float g(float (*f)(float ), float x)
{
     <Anweisungen>
     // Der Aufruf von f erfolgt durch f(z)
}
```

Ein Beispiel: Eine Funktion g soll ein Feld mit den Werten einer Funktion belegen:

```
float f1(float );
float f2(float );
void fvec(float (*)(float ),float *, float *, int );

int main()
```

```
{
  int DIM;
  float *x, *y1, *y2;              // Laenge der Felder ist DIM

  <...>

  fvec(f1,x,y1,DIM);               // y1 wird mit werten von f1 belegt
  fvec(f2,x,y2,DIM);               // y2 wird mit werten von f2 belegt

  return 0;
}


void fvec(float (*f)(float ),float *x, float *y, int N)
{
  int i;

  for(i=0 ; i<N-1 ; i++)
    y[i] = f(x[i]);
}

float f1(float x){ return pow(x,2.);}
float f2(float x){ return pow(x,3.);}
```

7.7 Rekursion

C-Funktionen können rekursiv benutzt werden: Eine Funktion kann sich selbst aufrufen. Nützlich ist dies zum Beispiel bei der Arbeit mit "rekursiv" definierten Datensätzen, wie zum Beispiel Baumstrukturen. Ein kleines Beispiel soll den rekursiven Funktionsaufruf verdeutlichen.

```
int fakultaet(int n)
{
  if (n <= 0)
  {
    printf("Nur fuer positive ganze Zahlen!\n");
    return(-1);
  }
  else if (n == 1 || n == 2)
    return(n);
  else
    return(n*fakultaet(n-1));
}
```

Die ersten beiden `return` Statements arbeiten wie gewohnt und beenden die Funktion bei gleichzeitiger Rückgabe des Funktionswertes. Im letzten Fall dagegen sieht das etwas anders aus. Zur Berechnung des Funktionswertes ruft die Funktion `fakultaet` sich hier noch einmal selbst auf.

Generell ist es wichtig, beim rekursiven Aufruf von Funktionen eine endlose Rekursion zu verhindern. Damit die Rekursion nicht zwangsläufig zum Programmabbruch führen muss, sollte ein Ende der Rekursion (also ein `return` **ohne** erneuten rekursiven Funktionsaufruf) irgendwo in der betreffenden Funktion zu erreichen sein...

–8–
Dateizugriff

Bislang wurde jede Ein– und Ausgabe über die Tastatur bzw. den Bildschirm vorgenommen. Spätestens dann, wenn eine Berechnung am Computer längere Zeit in Anspruch nimmt, möchte man Zwischenergebnisse zum Beispiel für eine spätere Nachbearbeitung speichern. Wir benötigen daher die Fähigkeit Dateien zu lesen und zu schreiben.

Bevor ein Lese– oder Schreibzugriff auf eine Datei erfolgen kann, müssen die Datei und die gewünschte Zugriffsart (lesen oder schreiben, und Varianten davon) spezifiziert werden. Man spricht davon, dass eine Datei "geöffnet" wird. Dies geschieht mit der Funktion `fopen()`. Die Funktion erwartet als Argument einen String mit dem Namen der Datei, und einen weiteren String, welcher die Zugriffsart bezeichnet. Als Rückgabewert liefert `fopen()` einen sogenannten `FILE`– Zeiger. Der Typ `FILE` entspricht einer Struktur, welche die für das Betriebssystem notwendige Information zum Dateizugriff enthält. Die genaue Definition ist für uns nicht weiter wichtig, sie steht im Includefile `<stdio.h>`, welches prinzipiell bei jedem Programm mit Ein– und Ausgabeoperationen eingebunden werden muss. Ein Beispiel dazu:

```
#include <stdio.h>

main()
{

  FILE* fp;

...

  fp = fopen("meine.dat", "r+");

...

}
```

Damit wird die Datei `meine.dat` im aktuellen Verzeichnis zum Lesen und zum Schreiben geöffnet. Zugriffe auf diese Datei können jetzt über den `FILE`– Zeiger `fp` erfolgen.

Im ersten String kann der Name auch mit absolutem oder relativem Pfad stehen. In diesem Fall kann die Datei nur dann geöffnet werden, wenn die Pfadangabe gültig ist. Generell gilt bei einem Unixsystem, dass eine Datei nur dann geöffnet werden kann, wenn das Programm die Berechtigung dafür hat. Im Normalfall hat ein Programm dieselben Rechte, die der Benutzer hat, welcher das Programm startet.

Die vollständige Deklaration der Bibliotheksfunktion `fopen()` folgt hier:

```
FILE *fopen( const char *filename, const char *mode);
```

Erlaubte Werte für die Zugriffsmodi sind:

"r" Textdatei zum Lesen öffnen

"w" Textdatei zum Schreiben erzeugen; gegebenenfalls alten Inhalt wegwerfen

"a" anfügen; Textdatei zum Schreiben am Dateiende öffnen oder neu erzeugen

"$r+$" Textdatei zum Ändern öffnen

"$w+$" Textdatei zum Ändern erzeugen; gegebenenfalls alten Inhalt wegwerfen

"$a+$" anfügen; Textdatei zum Ändern eröffnen oder erzeugen; Schreiben am Ende

Eine mit `fopen()` geöffnete Datei sollte nach dem letzten Zugriff mit der Anweisung `fclose(fp)` wieder geschlossen werden, dabei steht `fp` für den `FILE`– Zeiger, welcher beim Öffnen von `fopen()` geliefert wurde.

8.1 Formatierte Ein– und Ausgabe

Ähnlich wie die Funktion `printf()` zur formatierten Bildschirmausgabe gibt es zur formatierten Ausgabe in eine (zuvor mit `fopen()` geöffnete) Datei die Funktion `fprintf()`:

```
int fprintf(FILE *filename, const char *format, ...);
```

Das erste Argument enthält einen `FILE`– Zeiger auf eine zum Schreiben geöffnete Datei, die folgenden Argumente entsprechen exakt den Argumenten der Funktion `printf()`.

Zum Lesen aus einer Datei verwendet man die Funktion `fscanf()`, welche analog zu `scanf()` (formatierte) "Eingaben" liest. Ein Beispiel:

```
#include <stdio.h>

main()
{

  FILE* fp;
  float x,y;
  int i,j;
...

  fp = fopen("meine.dat", "r");
```

```
  fscanf(fp,"%d", &i);

  fclose(fp);

  fp = fopen("meine.dat", "w");

  fprintf(fp,"x = %f, y = %f, i = %d und j = %d\n",x,y,i,j);

  fclose(fp);

...
  return;
}
```

Hier wird die Datei `meine.dat` zunächst zum Lesen geöffnet, eine Integerzahl eingelesen und in der Variablen `i` abgelegt. Dann wird die Datei geschlossen und zum Schreiben erzeugt. Dabei geht der alte Inhalt der Datei verloren. Schließlich werden die Variablen `x,y,i` und `j` mit zusätzlichem Text in die Datei geschrieben und diese wieder geschlossen.

–9–
Strukturen

Manchmal gibt es eine Gruppe von Variablen verschiedenen Typs, die aus bestimmten Gründen eine Einheit bilden. Zum Beispiel könnten wir unter einem linearen Gleichungssystem eine Gruppe von drei Variablen verstehen, nämlich die Matrix `A` und die beiden Vektoren `x` und `b`. Eine Funktion wie `gauss()` erwartet ein lineares Gleichungssystem, welches innerhalb der Funktion zu lösen ist. Man kann nun die drei Variablen einzeln übergeben, aber es wäre eleganter, wenn wir eine Möglichkeit hätten, diese Variablen zu einem "Gleichungssystem" zusammen zu fassen und dieses als Ganzes zu übergeben. Das Mittel hierzu sind Strukturen. Eine Struktur ist in C eine solche Ansammlung mehrerer (oder auch nur einer einzigen) Variablen von möglicherweise unterschiedlichem Typ, die unter einem einzigen Namen zusammengefasst werden.

9.1 Grundlagen

Wir wollen zunächst einmal eine Struktur für unser lineares Gleichungssystem erzeugen. Dazu müssen wir zunächst überlegen, welche Größen wir tatsächlich zur Beschreibung des Systems benötigen: sicherlich die drei oben genannten Variablen Matrix, Lösungsvektor und rechte Seite. Aber falls wir die flexible Methode dynamischer Matrizen und Vektoren beibehalten wollen dann brauchen wir dazu auch noch eine Variable, welche die Dimension unseres Gleichungssystems enthält. Wir benötigen die folgenden Variablen:

```
  double *A;   /* das wird die Matrix werden */
  double *x;   /* der Loesungsvektor */
  double *b;   /* die rechte Seite */
  int    dim;  /* die Dimension des Systems */
```

Die Struktur `lgs` wird nun ganz einfach so angelegt:

```
struct lgs
{
  double *A;   /* das wird die Matrix werden */
  double *x;   /* der Loesungsvektor */
  double *b;   /* die rechte Seite */
  int    dim;  /* die Dimension des Systems */
};
```

Mit dem reservierten Wort `struct` beginnt eine Strukturvereinbarung. Als nächstes folgt hier das "Etikett", das ist der Name für die Struktur. Eingeschlossen in geschweiften Klammern werden schließlich die Variablen deklariert, welche in der Struktur zusammengefasst werden sollen. Man nennt diese Variablen die "Komponenten" der Struktur. Komponenten einer Struktur dürfen denselben Namen besitzen, wie gewöhnliche Variablen, ohne dass dadurch ein Konflikt entstünde. Genauso kann derselbe Name für Komponenten unterschiedlicher Strukturen auftreten. Mit der Vereinbarung der Struktur `lgs` haben wir allerdings noch keine solche Struktur explizit angelegt, sondern nur die Beschreibung für einen solche Struktur geliefert. Um tatsächlich Variablen vom Typ der neudefinierten Struktur zu erhalten, können deren Namen hinter der schließenden Klammer angegeben werden:

```
struct lgs
{
...
} erstesLGS, zweitesLGS;
```

Hier werden die beiden Variablen `erstesLGS` und `zweitesLGS` deklariert, welche vom Typ `struct lgs` sind. Bei einer Strukturvereinbarung darf das Etikett fehlen, dann ist diese Variante die einzige Möglichkeit Variablen eines solchen neuen Typs auch explizit im Speicher anzulegen. Mit Hilfe des Etiketts kann dies aber auch an einer anderen Stelle im Programm durch die Deklaration `struct lgs erstesLGS, zweitesLGS;` geschehen. Etikettenlose Strukturen sind folglich weniger flexibel als etikettierte Strukturen, weswegen letztere deutlich häufiger benutzt werden.

Hat man eine Struktur nun vereinbart und eine entsprechende Variable deklariert, dann kann mit einem Ausdruck der Form "Struktur–Variablenname.Komponente" auf die entsprechende Komponente der Struktur zugegriffen werden: `erstesLGS.a` steht für den Zeiger auf Zeiger auf `double` a in der Struktur `erstesLGS` vom Typ `struct lgs`.

Strukturen können ihrerseits wieder (zuvor vereinbarte) Strukturen als Komponenten enthalten, womit komplexen Datenstrukturen Tür und Tor geöffnet werden...

Variablen vom Typ einer Struktur dürfen kopiert oder als "Ganzes" einander zugewiesen werden: `zweitesLGS = erstesLGS;` Einer Funktion können als Argumente einzelne Komponenten einer Strukturvariablen oder eine ganze Strukturvariable übergeben werden:

```
float f(struct lgs mylgs)
{

...

}

struct lgs LGS;
```

```
float x;
...

x = f(LGS);
```

Mit dem Operator & kann die Adresse einer Struktur bestimmt werden. Bei der Deklaration einer Strukturvariable kann diese auch initialisiert werden, etwa so:

```
struct lgs{...};

struct lgs meinLGS = {nil,nil,nil,nil,0};
```

womit hier die Dimension `meinLGS.dim` auf den Wert 0 gesetzt wird und alle anderen Komponenten mit `nil` initialisiert werden.

Achtung! Strukturvariablen können **nicht** miteinander verglichen werden.

Ähnlich wie bei Variablen mit einem der bekannten Grundtypen, kann man auch Zeiger auf Variablen vom Typ einer Struktur deklarieren.

Achtung! Wenn man für eine dynamische Strukturvariable Speicherplatz allocieren möchte, dann muss dies analog zu diesem Beispiel geschehen:

```
...

struct lgs *ein_Zeiger_auf_mein_LGS;

...
ein_Zeiger_auf_mein_LGS = (struct lgs *) malloc(sizeof(struct lgs));
```

Zeiger auf Strukturen werden sehr oft verwendet. Um einfacher auf Komponenten einer Strukturvariablen über den Zeiger auf ihre Adresse zugreifen zu können, dient der Operator `->`. Damit können wir etwa auf die Komponente von `ein_Zeiger_auf_mein_LGS` wie folgt zugreifen:

```
double *meine_Matrix;
double *meineLoesung, *meineRHS;
int dim;

...

ein_Zeiger_auf_mein_LGS->dim = 17; /* Belegen */
```

```
meine_Matrix = ein_Zeiger_auf_mein_LGS->a; /* Zuweisen */
meineLoesung = ein_Zeiger_auf_mein_LGS->x;
meineRHS = ein_Zeiger_auf_mein_LGS->b;
dim = ein_Zeiger_auf_mein_LGS->dim;

...
```

9.2 Rekursive Strukturen

Manche Problemstellungen führen auf Listen, welche aus vielen Elementen desselben Typs bestehen. Ohne auf die verschiedenen Sorten von Listen hier näher eingehen zu wollen beschreiben wir hier nur kurz eine sogenannte "doppelt verkettete Liste". Bei einer solchen Liste kennt jedes Listenelement sowohl das vorhergehende, als auch das folgende Element der Liste. Ein Listenelement können wir also als eine Zusammenfassung zweier Zeiger auf Listenelemente und möglicherweise weiterer Variablen verstehen. (Die sollten schon noch dabei sein, denn wozu ist eine Liste gut, die nur sich selbst und sonst keine Daten enthält?) Zur Zusammenfassung von Variablen bieten sich Strukturen an. Im Gegensatz zu den bisherigen Strukturen benötigen wir hier aber eine Struktur, die Zeiger auf Strukturen vom selben Typ enthält: eine rekursive Struktur. Die Vereinbarung einer rekursiven Struktur unterscheidet sich in C nicht von der Vereinbarung sonstiger Strukturen:

```
struct Listeneintrag
{

  struct Listeneintrag *vorher;
  struct Listeneintrag *nachher;

...

};
```

Ähnlich ist auch eine andere Variante rekursiver Strukturen: zwei (oder mehrere) Strukturen, welche wechselseitig Zeiger auf einander enthalten:

```
struct a
{

...

 stuct b *z;
```

```
...
};

struct b
{

...

 stuct a *y;

...
};
```

9.3 typedef

Die Vereinbarung einer Struktur entspricht dem Einführen eines neuen Variablentyps. Zur Vereinfachung der Deklaration von Variablen eines neuen Typs können mit der Anweisung typedef Namen für neue Variablentypen deklariert werden. Dies sieht für unseren LGS–Typen zum Beispiel so aus:

```
typedef struct lgs
{
  double *a;   /* das wird die Matrix werden */
  double *x;   /* der Loesungsvektor */
  double *b;   /* die rechte Seite */
  int    dim;  /* die Dimension des Systems */
} LGS;
```

Damit existiert der neue Typ LGS und entspricht exakt dem Typen `struct lgs`. Die Anweisung typedef kann nicht nur bei Strukturvariablen verwendet werden, sondern eignet sich ebenfalls zur Vereinbarung neuer Typen, welche sich aus anderen Typen zusammensetzen:

```
typedef float vektor[2];  /* ein zweidimensionaler Vektortyp */
typedef int *ipointer;    /* ein Typ "Zeiger auf int" */
typedef double *dpointer; /* ein Typ "Zeiger auf double" */

typedef vektor *vpointer; /* ein Typ "Zeiger auf vektor" */
```

Mit dem geschickten Einsatz der typedef–Anweisung kann die Lesbarkeit eines Programmcodes insbesondere bei Verwendung komplexerer Strukturvariabler deutlich erhöht werden.

–10–
Zeichenketten

Zeichenketten, auch strings genannt, sind Felder vom Typ `char`.

10.1 Ein Zeichen

Die Deklaration eines Zeichens erfolgt durch

```
char c;
```

und die Zuweisung eines Zeichens durch

```
c = 's';
```

Deklaration und Definition kann man auch kombinieren durch

```
char c='s';
```

Die Formatangabe für `printf/scanf` erfolgt durch

```
%c
```

10.2 Eine Zeichenkette als Feld

Die Deklaration einer Zeichenkette mit 5 Zeichen erfolgt durch

```
char d[6];
```

Achtung: Felder (auch Zeiger) von Zeichenketten weisen im Gegensatz zu den bisher kennengelernten Feldern, eine Besonderheit auf: Es muss immer ein Feldeintrag mehr deklariert werden, da der letzte Eintrag, der den Abschluss einer Zeichenkette beschreibt, immer `'\0'` sein muß.

Die Zuweisung kann folgendermaßen aussehen:

```
d[0]='h';
d[1]='a';
d[2]='l';
d[3]='l';
d[4]='0';
d[5]='\0';
```

Möglich ist auch folgende Variante:

```
d[6]="hallo";
```

Bei Zeichenketten gibt es eine implizite Dimensionierung, dass heißt es ist auch möglich Deklaration und Definition zu kombinieren, ohne die Länge der Zeichenkette explizit anzugeben,

nämlich so:

```
char d[]="hallo";
```

Was man allerdings nicht machen kann ist das folgende:

X

```
char g[6];
g="hallo";
```

NEIN !!!

Möchte man ganze Zeichenketten zuweisen, so muß dies mittels Funktion `sprintf` aus der `string.h`–Bibliothek gemacht werden:

```
sprintf(g,"hallo");
```

`sprintf` funktioniert wie `printf`, nur dass die Ausgabe nicht nach STDOUT (Bildschirm) sondern in g geschrieben wird.

Demzufolge kann man auch Zeichenketten mit variablem Inhalt durch Formatangabe zuweisen:

```
int l=4;
char g[12];
sprintf(g,"Datei_%d.dat",l);
```

Die Formatangabe für Zeichenketten ist %s. Die Anweisung

```
printf("g enthaelt: %s",g);
```

liefert dann die Ausgabe am Bildschirm:

```
>g enthaelt: Datei_4.dat
```

10.3 Mehrdimensionale Felder mit Zeichenketten

Deklaration und Definition:

```
char name[2][11]={"Peter","Mustermann"};
```

`name[0]` enthält dann die Zeichenkette `"Peter"` und `name[1]` die Zeichenkette `"Mustermann"`.

Alternativ ginge auch

```
sprintf(name[0],"Peter");
sprintf(name[1],"Mustermann");
```

10.4 Zeiger auf Zeichenketten

Deklaration eines Zeigers auf eine Zeichenkette:

```
char *e;
```

Speicherreservierung erfolgt durch durch Angabe von gewünschter Länge der Zeichenkette und des Typs `char`:

```
e = (char *)malloc(10 * sizeof(char ));

sprintf(e,"Jackson");
```

Das sind 7 Zeichen und somit enthält `e[7]` gerade `'\0'`. Das das letzte Zeichen `'\0'` ist passiert bei Zeichenkettenbelegung automatisch.

Das folgende Beispiel `message` zeigt, wie man eine 60 Zeichen lange Zeichenkette aus zwei Zeichenketten `msg1`, `msg3` und Einzelzeichen `msg2` erzeugen kann. Das Resultat ist eine Zeichenkette `msg`, die mit `msg1` beginnt und mit `msg3` endet. Der Zwischenraum wird mit Zeichen `msg2` gefüllt.

```
void message(char *msg1, char msg2, char *msg3)
{
  int l1, l2, ll, i, maxl, cmd, add=0;
  char *msg;

  cmd = strcmp(msg1,"\n");
  if(!cmd) add=1;

  maxl = 60+add;
  msg  = (char *)malloc((maxl+1) * sizeof(char ));

  l1=strlen(msg1);
  l2=strlen(msg3);
  ll = maxl-(l2+l1);


  for(i=0 ; i<maxl ; i++){
    if(i<l1)
      msg[i] = msg1[i];
    else if(i<ll+l1)
msg[i] = msg2;
    else
      msg[i] = msg3[i-(ll+l1)];
  }
  msg[maxl] = '\0';
  printf("%s\n",msg);

  free(msg);
}
```

Der Aufruf

```
message("hello",'.',"world!");
```

liefert die Ausgabe

```
hello.....................................................world!
```

am Bildschirm.

–11–
make und makefile

11.1 Grundsätzliches

Ein Makefile ist sehr nützlich um den Übersetzungsvorgang zu vereinfachen. Zunächst kann man die Befehle die man in der Shell eingeben würde in eine bestimmte Format in das Makefile schreiben.

```
CC = gcc

# Eine minimales Makefile

hello: hello.c
    $(CC) -o hello hello.c
```

Dies hat schon den Vorteil, dass man anstatt der ganzen Zeile den Compileraufruf einfach mit `make` erledigen kann.

In der ersten Zeile wird der Variable `CC` der Wert `gcc` zugewiesen (Anders als bei C müssen Variablen nicht zuerst deklariert werden). Die mit # beginnende Zeile ist ein Kommentar. Oder genauer ausgedrückt, alles was hinter # steht wird ignoriert. Jetzt aber zum interessanten Teil. Die zwei Zeilen

```
hello: hello.c
    $(CC) -o hello hello.c
```

müssen als eine Einheit betrachtet werden. Die erst Zeile beschreibt das Ziel (`hello` das vor `:` steht) und die Abhängigkeiten (alles was hinter `:` steht, hier nur `hello.c`). Die nächste Zeile, die mit einem Tab anfangen muss beschreibt mit welchen Befehlen das Ziel aus den Abhängigkeiten erzeugt werden soll. Über `$(CC)` wird auf den Wert `gcc` der vorher definierten Variable `CC` zugegriffen. Der Befehl der hier also ausgeführt wird um `hello` zu erzeugen ist

`gcc -o hello hello.c`.

Der grundsätzliche Aufbau sieht also formal so aus:

```
Ziel: Abhaengigkeiten
<tab>  Befehl
<tab>  ...
```

Ein Ziel hängt von den Abhängigkeiten ab und wird umgesetzt, wenn die folgenden Befehle ausgeführt werden. Wenn ein Ziel gefunden wird, werden die Befehle nicht automatisch umgesetzt. `make` überprüft zuerst ob es überhaupt nötig ist und zwar einfach in dem es die Änderungszeit des Ziels, im obigen Beispiel wäre das die Entstehungszeit der Datei `hello` (Zieldatei), mit den Änderungszeiten der Abhängigkeiten vergleicht. Wenn alle Abhängigkeiten als Dateien existieren und älter sind als das Ziel (die Zieldatei) geht make davon aus, dass das Ziel nicht mehr neu erzeugt werden muss. Wenn nur eine Datei in den Abhängigkeiten jünger ist als die Zieldatei oder die Zieldatei noch nicht existiert, wird es nach den Regeln in der nächsten Zeile neu erzeugt.
Der Vergleich des Zieles und der Abhängigkeiten muss sich aber nicht nur auf einer Ebenen abspielen, denn jede Abhängigkeit wird als neues Ziel betrachtet und es wird solange weiter geprüft bis ein Ziel ohne Abhängigkeiten oder eines bei dem alle Abhängigkeiten älter sind gefunden wurde. (Deshalb muss man aufpassen das man keine zirkulären Abhängigkeiten einbaut) Ein Beispiel dafür, dass eine Abhängigkeit selbst Ziel ist kommt später noch.

Wenn `make` ausgeführt wird, versucht es das erste Ziel im `makefile` zu erzeugen. Es ist nicht auf eine Reihenfolge der Zielangabe zu achten, aber darauf, welches das erste ist. Wir gehen darauf später noch mal ein.

11.2 Aufteilen in mehrere Quelldateien

Ab einer bestimmten Größe werden Programme schnell unübersichtlich. Um dem ein wenig entgegen zu wirken, kann man den Quellcode auf verschiedene Dateien verteilen. Dabei sollten natürlich logisch zusammenhängende Einheiten entstehen. Etwa eine Zusammenfassung aller Funktionen und lokalen Variablen, welche zu einem Löser für lineare Gleichungssysteme notwendig sind. Eine solche Strukturierung fördert nicht nur die Lesbarkeit eines Programmes, sondern erlaubt auch die einfache Weiterverwendung einzelner "Module" in einem anderen Programm.

Betrachten wir einmal folgendes Beispiel:

Ein Programm erzeugt den äußerst spannenden Ausdruck `hello`. Dies allerdings ausgeführt von einer Funktion `schreibwas()`. Unser Programm könnte dann, aufgeteilt auf mehrere Dateien so aussehen:

hello.c:
```
#include ‘‘func.h’’

int main()
{

 schreibwas();

 return 0;

}
```

func.c:
```
#include ‘‘func.h’’

void schreibwas()
{

 printf(‘‘hello’’);

}
```

func.h:
```
#include <stdio>

void schreibwas();
```

Im Headerfile `func.h` wird die Funktion `schreibwas` deklariert, in der Quelldatei `func.c` wird sie definiert und in `hello.c` aufgerufen. Überall dort, wo die Funktion definiert oder benutzt wird, muss sie auch deklariert werden. Deshalb wird das Headerfile `func.h` in beide Dateien eingebunden. Headerfiles, oder auch Includedateien genannt, die sich in einem Systembekannten Verzeichnis befinden (etwa `/usr/include` oder `/usr/local/include` zum Beispiel) werden mit eckigen Klammern gekennzeichnet. Wir kenne das schon: `#include <stdio.h>`. Alle selbstgeschriebenen, beziehungsweise alle Headerfiles, die keinen systembekannten Pfad besitzen müssen mit Anführungsstrichen versehen werden und werden vom Compiler im aktuellen Verzeichnis gesucht. Dort sollten sie zunächst auch liegen.

Unser Makefile könnte (muß es nicht) dann so aussehen

makefile (Variante 1):
```
CC=gcc

hello: hello.c func.c
<tab> $(CC) -o hello hello.c func.c
```

und in unserem aktuellen Verzeichnis sieht befinden sich nach dem Aufruf von `make` diese Dateien:

```
> ls
makefile hello.c func.c func.h hello
```

Nun können wir das Ganze noch ein wenig verbessern. Es ist nämlich möglich, dass jede Quelldatei zunächst separat zu einer sogenannten Objektdatei übersetzt wird. Sie trägt dann die Endung `.o`. Aus `hello.c` wird `hello.o` und aus `func.` wird `func.o`. Anschließend werden die erzeugten Objektdateien zu einem ausführbaren Programm `hello` zusammengefügt, beziehungsweise verlinkt. Das entsprechende Makefile sieht dann so aus:

makefile (Variante 2):

```
CC=gcc

hello: hello.o func.o
<tab> $(CC) -o hello hello.o func.o

hello.o: hello.c
<tab> $(CC) -c hello.c
func.o: func.c
<tab> $(CC) -c func.c
```

Das Ziel `hello` hängt nun von den Objektdateien `hello.o` und `func.o` ab. Diese wiederrum von den jeweiligen Quelldateien. make wird also, um das Ziel `hello` zu erreichen zunächst die Ziele `hello.o` und `func.o` in Angriff nehmen. Dabei entstehen die entsprechenden Objektdateien. Nach dem Ausführen von `make` sieht es dann so in unserem Verzeichnis aus:

```
> ls
makefile hello.c func.c func.h hello.o func.o hello
```

Der Vorteil dieser Variante liegt darin, dass bei einem erneuten Ausführen von `make` nur diejenigen Quelldateien neu übersetzt werden, an denen eine Änderung stattgefunden hat, das heißt, die ein neueres Datum (Uhrzeit) haben als das ausführbare Programm. Bei sehr großen Programmen kann das ein enormer Vorteil sein, wenn man bedenkt, dass es Programme gibt, die einige Minuten benötigen, um übersetzt zu werden!

Man kann Ziele auch direkt ansteuern, indem man `> make <Ziel>` aufruft. Es wird dann nur das entsprechende Ziel verarbeitet. Durch Aufruf von

```
> make hello.o
```

wir also nur `hello.o` erzeugt.

Es gäbe kaum ein eigenes Kapitel für make und Makefile, wenn wir nun schon am Ende wären! Wir können Variante 2, ohne inhaltlich etwas zu verändern noch etwas kompakter, schöner und handlicher schreiben. Nämlich so:

makefile (Variante 3):

```
CC=gcc

ofiles=hello.o func.o

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

hello.o: hello.c
<tab> $(CC) -c hello.c
func.o: func.c
<tab> $(CC) -c func.c
```

In der dritten variante haben wir lediglich eine neue Variable `ofiles` eingeführt, der wir alle Objektdateien zuweisen, die das Ziel `hello` als Abhängigkeiten benötigt. Das sieht dann etwas übersichtlicher aus und ist auch handlicher, weil wir jede Datei, die während des Programmierens dazu kommt nur einmal aufschreiben müssen. Lästig ist nun noch, dass wir jede zu erzeugende Objaktdatei als Ziel vereinbaren müssen. Diese Schreibarbeit können wir uns auch sparen, indem wir zu Variante 4 des Makefiles übergehen:

makefile (Variante 4):

```
CC=gcc

ofiles=hello.o func.o

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -c $<
```

Die letzten beiden Zeilen sind so zu verstehen, dass alle Quelldateien (`.c`) zu Objektdateien (`.o`) übersetzt werden sollen. "Dodemit isch de Kiddel dann g'flickt!".

Bevor wir weitere Makefile-features betrachten kommen wir zunächst zu ein paar Dingen, die auf der C–Programmier–Seite noch zu beachten sind:

11.3 Externe Variablen

Bei einer Aufteilung des Codes auf verschiedene Dateien sind mehrere Dinge zu beachten: manche Variablen müssen in verschiedenen Quelldateien gültig sein, und doch jeweils dasselbe Objekt bezeichnen. Solche globale Variablen erschweren jedoch die Aufteilung in unterschiedliche Dateien und die Verwendung einzelner Teile des Codes in anderen Programmen. Daher sollten möglichst wenige globale Variablen verwendet werden. Es ist statt dessen oft besser, Variablen als Parameter an Funktionen weiterzureichen. Manche Funktionen sind nur im lokalen Zusammenhang sinnvoll, diese sollten dann auch als lokale Funktionen deklariert werden. Um die global verwendbaren Funktionen auch tatsächlich in unterschiedlichen Dateien zur Verfügung zu haben, müssen diese in jeder Datei vor ihrer Verwendung deklariert werden. Dies geschieht am einfachsten durch den Einsatz von eigenen Headerfiles. Schließlich kommen beim Übersetzen eines Programmes, dessen Quellcode aus mehreren Dateien besteht, die Vorteile eines Makefiles deutlich zur Geltung.

Der Gültigkeitsbereich einer Variablen, die außerhalb jeder Funktion deklariert wurde, umfasst den gesamten Bereich ab der Stelle ihrer Deklaration bis zum Ende der betreffenden Quelldatei. Im Gegensatz zu den internen Variablen, welche innerhalb einer Funktion deklariert werden, und nur dort gültig sind, können alle Funktionen der Datei, die nach der Variablendeklaration deklariert wurden, auf diese globale Variable zugreifen. Allerdings endet der Gültigkeitsbereich mit der Quelldatei. Um eine globale Variable auch den Funktionen aus

einer weiteren Quelldatei zur Verfügung zu stellen, muss die betreffende Variable "extern" deklariert werden.

Die notwendige `extern`–Deklaration von Funktionen und Variablen, die in mehreren Quelldateien gemeinsam benutzt werden sollen, lässt sich mit Hilfe von Deklarations– oder Headerfiles elegant bewerkstelligen. Bereits von der ersten Lektion an haben wir mit bestimmten Deklarationsdateien gearbeitet: dem Headerfile `stdio.h` zum Beispiel. Mit der Verwendung von Headerfiles vereinfacht sich die `extern`–Deklaration in unterschiedlichen Quelldateien deutlich: Es gibt nur noch eine Datei, in der alle benötigten Deklarationen externer Variablen und Funktionen eingetragen werden müssen. In den Quelldateien, in welchen bestimmte externe Variable oder Funktionen benutzt werden, muss zu Beginn der Datei das betreffende Headerfile per `include`–Anweisung eingebunden werden. Generell muss man abwägen zwischen dem Wunsch, dass jede Datei nur Zugriff auf die tatsächlich benötigten externen Variablen und Funktionen hat, und dem steigenden Aufwand bei der Verwaltung von immer mehr Deklarationsdateien.

Zurück zu unserem Beispiel:

hello.c:

```
#include ‘‘func.h’’

int flag=1;


int main(int argc, int *argv[])
{

 if(argc>1) flag=atoi(argv[1]);

 schreibwas();

 return 0;

}
```

func.c:

```
#include ‘‘func.h’’

extern int flag;


void schreibwas()
{

 if(flag) printf(‘‘hello’’);

}
```

Das Headerfile `func.h` bleibt dabei unverändert.

Für globale und externe Variablen gilt: in allen Quelldateien für ein Programm darf es insgesamt nur eine Deklaration für eine bestimmte (globale) Variable geben. Dagegen dürfen in beliebig vielen Quelldateien `extern`–Deklarationen einer definierten Variable stehen.

11.4 Makefile für Fortgeschrittene

Nun ist es vielleicht sinnvoll sowohl die Headerfiles als auch die Quelldateien in verschiedenen Verzeichnissen zu sammeln. Das können wir auch tun, wobei wir dann `make` die entsprechenden Pfade der Verzeichnisse mitteilen müssen. Wir erweitern unser hello–Beispiel folgender-

massen: Alle Headerfiles (hier nur eins) sollen in das Verzeichnis `INC` und alle Quelldateien bis auf die Haputdatei `hello.c` sollen in das Verzeichnis `SRC` verschoben werden. Fangen wir mit dem Headerfile an. Unser Verzeichnis sieht so aus:

```
> ls -R

.:
func.c hello.c INC makefile


./INC:
func.h
```

Das Makefile variieren wir dahingehend, dass dem Compiler als Option `-I<Pfad der Includedateien>`, also hier `-I./INC` übergeben. `./INC` bedeutet "in diesem Verzeichnis (`./`) das Unterverzeichnis `INC`.

makefile (Variante 5):

```
CC=gcc

ofiles=hello.o func.o

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -I./INC -c $<
```

In Variante 6 machen wir das gleich noch ein wenig schöner, was inhaltlich aber das gleiche ist wie Variante 5, und definieren eine Variable namens `includes`, die den Pfad der Headerfiles beinhaltet, also `includes=./INC`.

makefile (Variante 6):

```
CC=gcc

ofiles=hello.o func.o

includes=./INC

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -I$(includes) -c $<
```

Mehrere Includeverzeichnisse können durch mehrmaliges Übergeben der Option `-I` bekannt gegeben werden. Etwa so:

```
-I./INC1 -I./INC2
```

und wo seiter.

Da jetzt die Includedatei in ihrem Unterverzeichnis bekannt ist können wir auch getrost die Funktion `func.c` in ein weiteres Unterverzeichnis, sagen wir `SRC` verschieben. Die Verzeichnisstruktur sieht nun so aus:

```
> ls -R
.:
hello.c INC makefile SRC

./INC:
func.h

./SRC:
func.c
```

Damit `make` weiß wo es nach Quelldateien zu suchen hat belegen wir die Variable `VPATH` mit den Pfaden der entsprechenden Verzeichnisse. In unserem Beispiel ist es nur ein Verzeichnis und mit dem Makefile Variante 7 können wir problemlos unser Programm übersetzen lassen.

makefile (Variante 7):

```
VPATH=./SRC

CC=gcc

ofiles=hello.o func.o

includes=./INC

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -I$(includes) -c $<
```

`VPATH` ist ein feststehender Name für `make` und kann nicht durch einen anderen ersetzt werden. Mehrere Verzeichnissangaben werden durch `:` voneinander getrennt. Etwa

```
VPATH=./:./SRC1:./SRC2
```

und so weiter.

11.5 Noch ein wenig Schnick–Schnack

Gelegentlich ist es sinnvoll alles bisher Übersetzte neu zu übersetzen. Wenn zum Beispiel nur eine Headerdatei verändert wurde wird `make` das nicht registrieren, da es nur Ziele und

Abhängigkeiten nach dem Datum und der Uhrzeit überprüft. Um kurzerhand alles zu löschen und neu zu übersetzen, bauen wir in unser Makefile die Ziele `clean` und `new` ein:

makefile (Variante 8):

```
VPATH=./SRC

CC=gcc

ofiles=hello.o func.o

includes=./INC

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -I$(includes) -c $<

clean:
<tab> rm -f *.o hello

clean: clean hello
```

`clean` hat keine Abhängigkeiten aber den Befehl alles zu löschen, was mit .o endet und das ausführbare Programm `hello`.

`new` hat keinen direkten Befehl aber Abhängigkeiten, die wiederrum zu verarbeitende Ziele sind, nämlich erst `clean` und dann `hello`.

`make` arbeitet von oben nach unten das Makefile ab und verfolgt das erste Ziel, das ihm begegnet und diejenigen Ziele, die das erste als Abhängigkeiten enthält. In unserem Beispiel wird `make` lediglich das Ziel `hello` ansteuern und infolge dessen auch das Ziel `.c.o`. Sonst nix. Stünde das Ziel `clean` an erster Stelle, so würde beim Aufruf von `> make` das selbe wie bei `> make clean` passieren. Damit man nicht ständig die Ziele umrangieren muss kann man sich der Variable mit feststehemdem Namen `default` bedienen. Diese als erstes Ziel gesetzt bedeutet, dass das Ziel, welches die Abhängigkeit von `default` ist angesteuert wird, egal wo es geschrieben steht, falls `make` ohne genaue Zielangabe aufgerufen wird.

makefile (Variante 9):

```
VPATH=./SRC

CC=gcc

ofiles=hello.o func.o

includes=./INC

default: hello

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -I$(includes) -c $<

clean:
<tab> rm -f *.o hello

clean: clean hello
```

Noch eine kleine, hübsche Kleinigkeit. Wir können dem Compiler diverse Optionen übergeben. Je nachdem, ob das Programm möglichst schnell also optimiert laufen soll oder ob es sowohl beim compilieren als auch beim Rechnen anschließend verstärkt Informationen, wie zum Beispiel Allokationsfehler, sammeln soll. Dies kann man bequem durch einen Übergabeparameter, nennen wir ihn C an `make` realisieren. Wir erweitern unser Makefile durch folgende if–else–Anweisung:

```
ifeq ($C,1)
    CC += -g -Wall -pedantic
else
    CC += -O2
endif
```

Bei Aufruf von `> make C=1` werden dem Compiler noch die Optionen `-g -Wall -pedantic` zum debuggen, andernfalls die Option `-O2` zum optimieren übergeben. Nun sieht unser Makefiles so aus:

makefile (Variante 10):

```
VPATH=./SRC

CC=gcc

ifeq ($C,1)
CC += -g -Wall -pedantic
else
CC += -O2
endif

ofiles=hello.o func.o

includes=./INC

default: hello

hello: $(ofiles)
<tab> $(CC) -o $(ofiles)

.c.o:
<tab> $(CC) -I$(includes) -c $<

clean:
<tab> rm -f *.o hello

clean: clean hello
```

Und damit können wir schon ganz zufrieden sein.

Welche Optionen es gibt und was sie genau bedeuten hängt vom Compiler und der Compilerversion ab. Informationen darüber erhält man mittels

```
> man gcc
```

11.6 Der Nutzen

`make` ist durch seine Eigenschaften eine sehr verbreitetes und auch vielseitig einsetzbares Programmierwerkzeug.

- Statt langer Kommandozeilen für viele Programme einzugeben genügt ein einfacher Aufruf von `make`.
- Es hilft große Projekte, die aus hunderten Einzelprogrammen und vielleicht aus tausenden C-Quelldateien bestehen zu übersetzen.

- Es verhindert das bei einer kleinen Änderung alles neu übersetzt werden muss.
- Gleichzeitig trägt es dafür Sorge, dass die Änderungen überall dort Auswirkungen haben wo es sein muss.
- Durch ein `makefile` können die Änderungen die zum Portieren auf ein anderes System nötig sind, zentralisiert und minimierte werden. (Das beste Beispiel ist wohl die Variable `CC`, falls der Compiler anders heißt muss man nur `CC` ändern.

Leider hat das natürlich auch Nachteile. Zum einen ist `make` eine weitere Programmiersprache die man lernen muss. Zum anderen kann ein `Makefile` durch viele Abhängigkeiten auch sehr komplex werden und Fehler in der logischen Struktur können auch zu Fehlern in den Programmen führen.

11.7 `static`

Manchmal ist es sinnvoll, den Gültigkeitsbereich einer gobalen Variablen oder einer Funktion auf eine einzige Datei einzuschränken. Dies kann mittels einer `static`–Vereinbarung geschehen:

```
static int dim;
static double mymax(double x, double y);
```

Erfolgt eine `static`–Vereinbarung einer Variablen innerhalb einer Funktion, so ist die betreffende (lokale) Variable "statisch". Das bedeutet, dass die Variable nicht bei jedem Funktionsaufruf im Speicher neu erzeugt wird, sondern nach dem ersten Aufruf erhalten bleiben und bei weiteren Aufrufen derselben Funktion mit dem **alten** Inhalt zur Verfügung stehen.

A Eine kleine Einführung in Unix

Das Compilieren und Linken von C Programmen führen wir über die Konsole aus. Die wichtigsten Befehle in der Konsole sind:

- `cd <Verzeichnisname>`:(**c**hange **d**irectory)
  Wechselt in das Verzeichnis `<Verzeichnisname>`
- `cd ..`:
  Wechselt in das Verzeichnis eine Ebene höher.
- `ls`: (**l**i**s**t)
  Listet den Inhalt des aktuellen Verzeichnisses auf.
- `ls -l`: (**l**ist **l**ong)
  Listet den Inhalt des aktuellen Verzeichnisses mit Zusatzinformationen auf.
- `rm <Dateiname>`:(re**m**ove)
  Datei `<Dateiname>` (unwiderruflich) löschen.
- `cp <Quelldatei> <Ziel /Verzeichnis>`: (**c**o**p**y)
  Datei `<Quelldatei>` kopieren.
- `mv <Quelldatei> <Ziel /Verzeichnis>`: (**m**o**v**e)
  Datei `<Quelldatei>` verschieben.
- `pwd`: (**p**rint **w**orking **d**irectory)
  Gibt den Namen des aktuellen Verzeichnisses aus.
- `mkdir <Verzeichnisname>`: (**m**a**k**e **dir**ectory)
  Verzeichnis `<Verzeichnisname>` anlegen.
- `rmdir <Verzeichnisname>`: (re**m**ove **dir**ectory)
  Verzeichnis `<Verzeichnisname>` löschen.
- `man <Befehlsname>`:(**man**ual)
  Manual-pages des Komandos `<Befehlsname>` anschauen. `man ls` liefert den Handbucheintrag zum Befehl `ls`.

Wenn man die ↑-Taste drückt, steht in der Konsole wieder der Befehl, den man zuletzt eingegeben hatte.

Ganz wichtig ist es, sich bewusst zu sein, dass unter Unix immer Groß- und Kleinschreibung unterschieden werden. So erzeugt `mkdir Programm` ein anderes Verzeichnis als `mkdir programm`, und `MKDIR Programm` wird nicht funktionieren, weil es das Programm `MKDIR` nicht gibt.

Unixprogramme fragen meistens keine Bestätigung ab. Das heißt, alle eingegebenen Befehle werden nach `return` ausgeführt. Dies wird manchmal im Zusammenhang mit `rm` und Tippfehlern schmerzlich in Erinnerung gerufen. Bei den meisten Befehlen kann man eine Bestätigung `-i` (**i**nteractiv) einschalten, also z.B.: `rm -i`, `mv -i`, `cp -i`.

B portable maps

Portable Bit/Grey/Pixmaps sind ein unkomprimiertes Format zur Bilddatenspeicherung. Der Vorteil bei dieser Art Bilddaten zu speichern, besteht darin, dass durch das Fehlen der Kompression keine Datenverluste entstehen. Außerdem ist es möglich, auf einfachste Weise auf die Werte einzelner Pixel zuzugreifen. Der Nachteil besteht dann darin, dass die Dateien sehr groß sind. Es gint zwei Formate der Pixelwertspeicherung nämlich ascii oder binär.

Der Grundsätzliche Aufbau eines Bildes, sagen wir mal eines Schwar–Weiß Bildes sieht so aus:

```
  1. Zeile: Identifikator
  2. Zeile: Kommentar (optional)
  3. Zeile: Breite Hoehe des Bildes
  4. Zeile: maximaler Farbwert (nicht bei Schwarz--Wei"s Bildern)
> 5. Zeile: Bilddaten
```

Betrachten wir zur Verdeutlichung das Bild mit dem Dateiinhalt in Abbildung 4. Die dargestellten Nullen stellen weiße und die Einsen schwarze Pixel dar. Auf der rechten Seite ist das entsprechende Bild zu sehen. P1 in der ersten Zeile ist der sogenannte Identifikator. Er "identifiziert" das Bildformat, was bedeutet, dass er darüber Aufschluß gibt ob es sich bei dem vorhandenen Bild um ein Schwarz–Weiß (P1/P4), Grau– (P2/P5) oder Farbbild (P3/P6) handelt. Des weiteren enhält er die Information, ob die Bilddaten im Ascii– (P1/P2/P3) oder Binär– (P4/P5/P6) Format gespeichert sind. Der Kommentar in der zweiten Zeile ist optional und kann auch entfallen. Es muss dort vor allen Dingen nicht der Name des Bildes stehen. In der dritten Zeile stehen Bildbreite und Bildhöhe, dass heisst im Beispiel aus Abbildung 4 besteht das Bild aus 12 Spalten und 18 Zeilen.

```
P1
# hallo.pbm
12 18
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 1 1 0 0 1 1 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0
0 0 1 0 1 0 0 1 0 1 0 0
0 1 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 1 0
0 0 1 0 1 0 0 1 0 1 0 0
0 0 1 0 0 1 1 0 0 1 0 0
0 0 0 1 1 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0
```

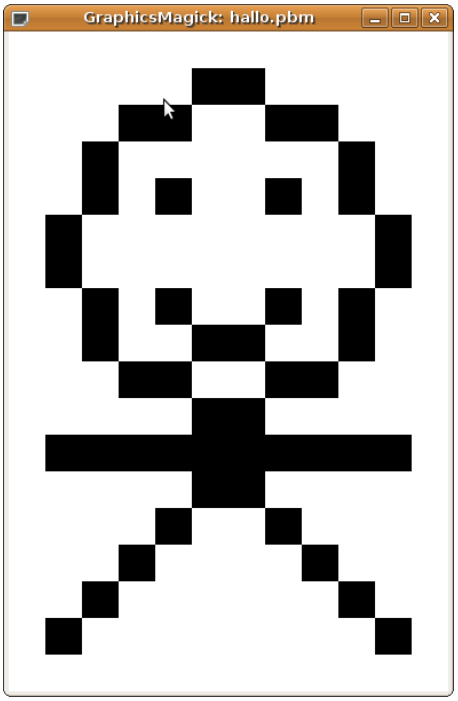


Abbildung 4: Beispiel für ein Portable Bitmap

Abbildung 5 zeigt ein Beispiel für ein grauwertiges Bild, dass heisst ein Bild, das nicht nur aus schwarzen und weißen Pixeln besteht sondern auch aus Grautönen dazwischen. Hier wird nun, im Gegensatz zum pbm–Format, Schwarz mit Nullen dargestellt. Der Identifikator in der ersten Zeile ist P2, der Kommentar in der zweiten Zeile ist auch hier, sowie in allen Varianten optional und kann genauso gut entfallen. Die dritte Zeile enthält wieder Bildbreite und Bildhöhe in Pixeln. Welcher Pixelwert Weiß entspricht hängt von der in Zeile vier festegelegten Farbtiefe oder dem maximalen Farbwert ab. Im Beispiel in Abbildung 5 ist der maximale Farbwert mit 15 festgelegt. Damit wird jeder Pixel mit dem Wert 15 in weiß dargestellt. Alle anderen Werte müssen dazwischen liegen und liefern einen entsprechenden Grauwert. Je kleiner der Wert desto dunkler ist der Pixel.

Bei einer Pixelwertdarstellung, wie sie für `feep.pgm` (Abbildung 5) gewählt wurde kann eine maximale Farbtiefe von 8 bit verwendet werden, das heisst, dass die Bildwerte zwischen 0 und 255 liegen müssen. Will man größere Werte darstellen so müssen die Werte in einer besonderen Weise gespeichert werden. Werte, die mehr als 8 bit benötigen werden als zwei 8 bit Zahlen beschrieben, die entstehen wenn man die entsprechende Darstellung in 16 bit auf zwei 8 bit Zahlen zerlegt. Wir verdeutlichen uns dies an einem Beispiel:

Wir wollen für einen Pixel die Zahl 1234 als Bildwert in die Bilddatei schreiben. Die 16 bit Darstellung für 1234 lautet:

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Abbildung 5: Beispiel für ein Portable Greymap

$$\underbrace{0\,0\,0\,0\,0\,1\,0\,0}_{2^2=4}\ \underbrace{1\,1\,0\,1\,0\,0\,1\,0}_{2^7+2^6+2^4+2^1=210} = 2^{10}+2^7+2^6+2^4+2^1 = 1234$$

An Stelle von 1234 wird dann 4 210 in die Datei gespeichert (mit Leerzeichen dazwischen!). Aber Vorsicht: In der Regel können Monitore nur 8 bit Farben darstellen, so dass man über den Monitor keinen Unterschied feststellen wird. 16 bit Farbtiefe ist zum Beispiel dann sinnvoll, wenn das Bild bereits – in diesem Sinne – so hoch aufgelöst aufgenommen wurde. Man hat dann bessere Chancen, bei der Nachbearbeitung Details aus über– oder unterbelichteten Bereichen herauszumodellieren.

Schlussendlich wollen wir aber mit Farbbildern arbeiten. Bei Farbbildern erhält jeder Pixel einen Rot–, einen Grün– und einen Blauwert. Alle weiteren Farben ergen sich aus der additiven Farbmischung aus diesen drei Grundfarben. Ein paar Beispiele sind in Abbildung 6 dargestellt.

Bei Farbbildern werden genau wie bei portable greymaps Farbwerte zwischen Null und dem angegebenen maximalen Farbwert definiert. Der einzige Unterschied besteht nun darin, dass für jeden Pixel drei statt einem Wert vergeben werden müssen. Das Beispiel in Abbildung 7 diene der Verdeutlichung. Der erste Pixel in der ersten Zeile hat die RGB–Werte (0,0,15). Das

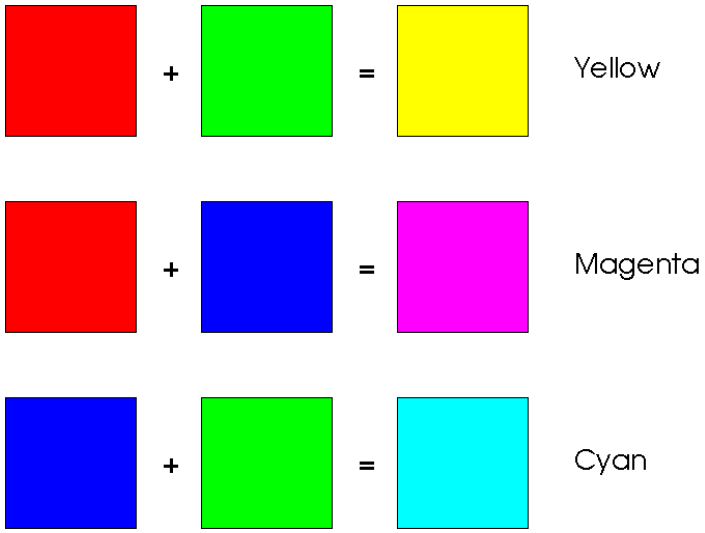


Abbildung 6: Additive Farbmischung

bedeutet kein Rot, kein Grün, voll Blau (da 15 die maximale Farbtiefe ist). Also ist der Pixel links oben in Abbildung 7 blau.

```
P3
# farben.ppm
3 3
15
0  0 15 15 15  0 15  0  0
0 15  0 15 15 15  0 15  0
15 0  0  0  0  0  0  0 15
```

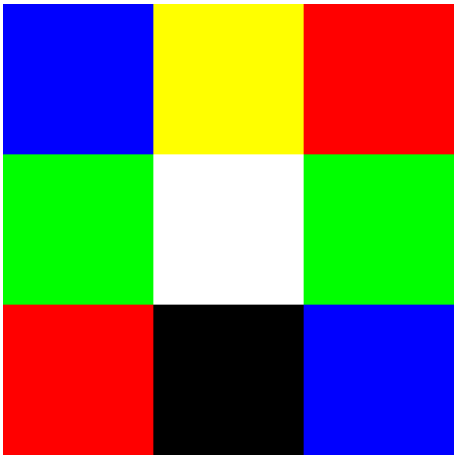


Abbildung 7: Beispiel für ein Portable Pixmap

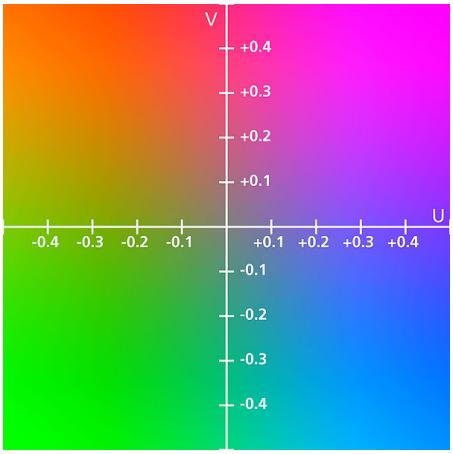
Abbildung 8 gibt noch einmal einen Überblick über die verschiedenen Formate.

Name	Portable Bitmap		Portable Greymap		Portable Pixmap	
Endung	.pbm		.pgm		.ppm	
Format	ascii	binär	ascii	binär	ascii	binär
Identifikation	P1	P4	P2	P5	P3	P6
max. Farbtiefe (mF)	–	–	$\leq$ 16 bit	$\leq$ 8 bit	$\leq$ 48 bit (=3x16 bit)	$\leq$ 24 bit (=3x8 bit)
Bilddaten	0,1	0,1	0,..,mF	0,..,mF	RGB: 0,..,mF	RGB: 0,..,mF

Abbildung 8: Überblick über die Bildformate

C Farbraumkonvertierung: RGB – YUV

Wir haben bereits die Pixeldarstellung durch RGB–Werte kennengelernt. In diesem Kapitel betrachten wir eine andere Möglichkeit von Pixeldarstellungen: Die YUV–Darstellung. Es handelt sich hierbei wiederum um eine Darstellung durch drei Werte Y, U und V. Y beinhaltet die Information über die Helligkeit und U und V enthalten Informationen über die eigentliche Farbe. Man kann sich grob, ganz grob vorstellen, Y beinhalte den Grün–Wert, U die Differenz zwischen Grün und Blau und V die Differenz zwischen Grün und Rot. So kann man sich vorstellen, dass man eineindeutig zwischen RGB– und YUV–Werten abbilden kann. Bei RGB– und YUV–Darstellungen sprechen wir von verschiedenen Farbräumen, in denen ein Pixel repräsentiert wird. Es gibt noch weitere Farbräume, wie zum Beispiel der HSV–Farbraum, der besonders dafür geeignet ist, die Sättigung eines Farbwertes zu variieren oder einfach nur in Erfahrung zu bringen. Auf Wikipedia befindet sich eine schöne Beschreibung dazu. Aber zurück zum YUV–Farbraum. Die oben genannte Transformation ist nicht ganz korrekt. Dennoch hat der Grün–Wert eines Pixels den größten Einfluss auf den Helligkeitswert. Das hat man so festgelegt, weil aus biologischen Grnden das Auge im Grünbereich am besten auflösen kann, das heisst, dass das Auge mehr unterschiedliche Töne in Grün wahrnehmen kann als in allen anderen Farbbereichen.



(r)0.4 Abbildung 9: UV–Ebene Die YUV–Darstellung findet ihren Ursprung im Schwarz–Weiss–Fernsehen. Zunächst wurden farblose Signale gesendet, denen dann später, bei der Entwicklung des Farbfernsehens, noch zwei Farbkanäle hinzugefügt wurden, nämlich U und V (siehe auch Abbildung 9). Man hat die Struktur ein graubwertiges Signal und zwei farbige Kanäle aus Kompatibilitätsgründen zu den farblosen Fernsehgeräten beibehalten. Sowohl bei PAL (Europa) als auch NTSC (USA) Formaten wird dieser Farbraum noch heute verwendet, denn er bietet noch weitere Vorzüge. Oft genügt es Berechnungen auf dem Y–Kanal, dem sogenannten Luma–Kanal, durchzuführen. Dazu gehört zum Beispiel das Entrauschen oder Schärfen von Bildern. Die Farb–Kanäle U und V, auch Chroma–Kanäle genannt, können dabei völlig außer Acht gelassen werden. Auch die Kantendetektion wird in der Regel nur auf den Luma–Werten durchgeführt.

Es gibt einige Varianten des YUV–Farbraumes und DIN normierte Transformationen dazu. Wir wollen es einfach halten und begnügen uns mit folgenden Farbraumtransformationen:

Es sei $P = (P_R, P_G, P_B)$ ein Punkt im RGB–Farbraum und $Q := (Q_Y, Q_U, Q_V)$ ein Punkt im YUV–Farbraum. Dann beschreibt die Abbildung

$$\Phi(P) := AP$$

mit

$$A := \begin{pmatrix} 19 & 38 & 7 \\ -11 & -21 & 32 \\ 32 & -27 & -5 \end{pmatrix} \frac{1}{64}$$

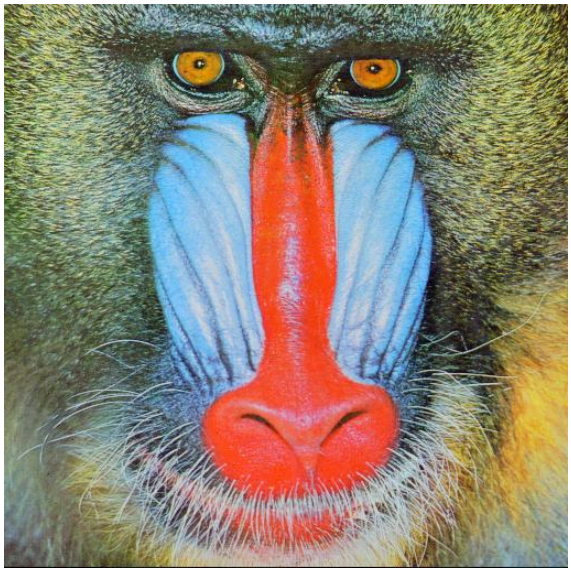
die Abbildung von RGB nach YUV und

$$\Psi(Q) := A^{-1}Q$$

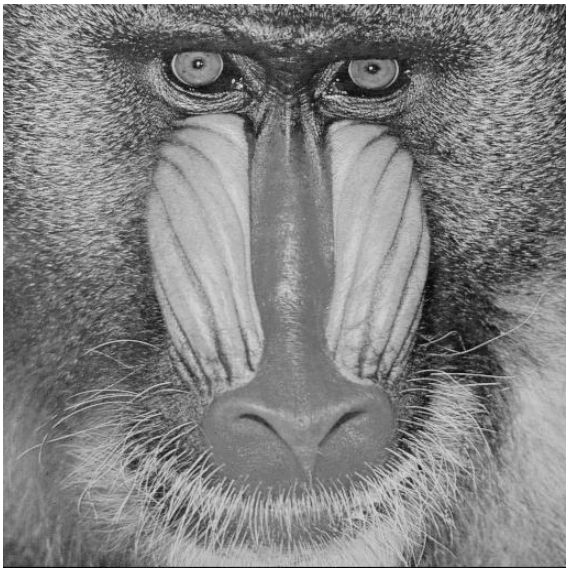
mit

$$A^{-1} := \begin{pmatrix} 969 & 1 & 1363 \\ 969 & -319 & -685 \\ 969 & 1729 & 19 \end{pmatrix} \frac{1}{969}$$

die Abbildung von YUV nach RGB. Als Beispiel siehe Abbildung 10.



Original



Y

U

V

Abbildung 10: Ein Beispiel für diese YUV–Darstellung

D Dokumentation mit doxygen

Doxygen ist ein Tool zum automatischen Erstellen von Dokumentationen aus speziellen Quelltext-Kommentaren. Nach verschiedenen Erweiterungen ist doxygen inzwischen in der Lage solche Code-Referenzen in verschiedenen Formaten, wie z.B. HTML und LaTeX, zu erstellen. Für gewöhnlich wird doxygen per Kommandozeile bedient, allerdings steht für Windows ein sehr praktisches GUI zur Verfügung.

D.1 Doxygen-konforme Kommentare

Eine simple Art der doxygen-konformen Kommentierung erreicht man schlicht durch Erweiterung der eigenen Kommentare um führende Anführungszeichen.

Grundlegend sind zwei Arten von Kommentaren vorgesehen: Kurz- und Detailbeschreibungen. Kurzbeschreibungen werden dabei mit `//!` formatiert und Detailbeschreibungen, die aus mehreren Zeilen bestehen können, mit `/*! Text */`.

So eine Kurzbeschreibung eignet sich besonders um sie bei Deklarationen stehen zu haben, z.B. in einer Header-Datei. Sie muss vor dem Quelltext stehen zu dem Bezug genommen wird.

Diese Art der Kommentare eignen sich besonders um vor Definitionen aufgeführt zu werden. Man kann auch beide Kommentartypen aufeinander folgend benutzen, wie z.B. bei Java-Code angebracht. Zusätzliche Funktionalität bietet doxygen durch Tags, die mit einem Backslash angeführt und mit einem Leerzeichen vom nachfolgenden Text getrennt, innerhalb von Kommentaren eingefügt werden können. Zwei solcher Tags sind `\param` (um Bemerkungen zu Parametern besonders hervorzuheben) und `\sa` (steht für "see also").

Dazu Beispiele:

1. ```
   /*! All needed UI components (and containers to hold these) for
   this medical data type are being instantiated using the Java Swing
   Foundation Classes. Component properties are being set and layout
   managers created. Action Listeners are realized as needed. \param
   name is used to indicate affiliation of the data displayed via the
   UI through labels */
       public BloodPressure(String name)...
   ```

2. ```
   /*! Creates the user interface and network communication
   by calling the respective methods \sa createAndShowUI() \sa
   createNetworkCommunication() */
       public static void main(String[] args) ...
   ```

Doxygen ist erhältlich unter

http://www.doxygen.org

für Windows, Linux und Mac.

D.2 Installation und Vorbereitung

Zur Installation der Software, die denkbar einfach ist, verweise ich auf die oben genannte Internetadresse. Wir behandeln hier lediglich die Handhabung.

Doxygen verwendet eine Konfigurationsdatei, um all seine Einstellungen festzusetzen. Das bedeutet, man muss zunächst eine Datei erstellen (keine Sorge auch das macht doxygen), in der man bestimmte Einstellungen wählt. Zu den Einstellungen gehört zum Beispiel welche Dateien in welchen Verzeichnissen dokumentiert werden sollen, mit oder ohne die Headerdateien, welche Dateien nicht dokumentiert werden sollen (zum Beispiel all die .dat-Dateien, die bei Programmabläufen unter Umständen erstellt werden), um welchen Quellcode es sich handelt (C, C++, Fortran, Java, etc.), in welches Verzeichnis die Dokumentation erstellt werden soll und einiges mehr. Wir wollen hier gar nicht auf jedes Detail eingehen.

Jedes Projekt sollte seine eigene doxygen-Konfigurationsdatei haben.

Um die Erstellung einer Konfigurationsdatei zu vereinfachen kann man sich von doxygen eine Vorlage dieser Datei erstellen lassen. Dies geschieht von der Kommandozeile aus mit dem Befehl

```
> doxygen -g <config-file>
```

Sagen wir mal so:

```
> doxygen -g doxyconfig
```

Nach Eingabe dieses Befehls wird eine Konfigurationsdatei `doxygenconfig` erstellt, die nachfolgend noch editiert werden muss. Es handelt sich dabei um eine Textdatei, die mit dem gewohnten Editor geöffnet werden kann.

Alle Variablen und Parameter, die in dieser Datei gesetzt werden können sind gut dokumentiert (sollte es auch, denn um's Dokumentieren geht's ja auch, oder?), so dass sie im Grunde genommen selbsterklärend ist.

Wir betrachten mal einige wenige, aber genügende Einstellungen für unser LGS–Programmpaket:

```
PROJECT_NAME           = LGS                    # optional (Titel)
OUTPUT_DIRECTORY       = doc                    # hier drin wird dann
                                                # die Doku erstellt
OUTPUT_LANGUAGE        = German                 # wenn man deutsch
                                                # dokumentiert
OPTIMIZE_OUTPUT_FOR_C  = YES                    # das ist schoener
INPUT                  = ./ LGS inc Others      # Verzeichnisse in
                                                # nach Dateien mit
                                                # doxygen-Kommentaren
                                                # gesucht werden soll
```

```
FILE_PATTERNS          = *.c *.h                 # diese Dateien
                                                 # dokumentieren
EXCLUDE                = *.dat *.ppm             # diese Dateien
                                                 # nicht dokumentieren
GENERATE_HTML          = YES                     # html Dokumentation
HTML_OUTPUT            = html                    # Verzeichnis dafuer
GENERATE_LATEX         = YES                     # latex Dokumentation
LATEX_OUTPUT           = latex                   # Verzeichnis dafuer
```

Ein Muss sind die Tags `OUTPUT_DIRECTORY`, `INPUT`, `FILE_PATTERNS`, `GENERATE_HTML`, `HTML_OUTPUT` und wenn gewünscht andere Ausgaben.

Eine gute Anleitung kann man unter

http://www.stack.nl/ dimitri/doxygen

finden. Dort gibt es eine Auflistung und Beschreibung spezieller Tag–Befehle:

http://www.stack.nl/ dimitri/doxygen/commands.html

Der Aufruf erfolgt dann in der Kommandozeile durch

```
doxygen doxyconfig
```

Es werden im Verzeichnis `doc` die Unterverzeichnisse `html` und `latex` erstellt und darin die entsprechnden Dokumentationen.

D.3 Dokumentationssyntax

Im Moment steht in unserer Dokumentation nichts darin, da wir auch noch nicht dokumentiert haben. Das machen wir jetzt mal in unserem LGS–Paket.

Merkregeln:

1. Jede Datei, die dokumentiert werden soll beginne mit

   ```
   /*!
     \file <dateiname>
   */
   ```

   Das ist zwingend. Man kann aber auch noch mehr reinschreiben. Etwa in `lgs.c`:

   ```
   /*!
     \file lgs.c
     \author Rebekka Axthelm
     \date Juli 2009
     \brief Dieses Programm loest ein lineares Gleichungssystem,
    wahlweise ...
   */
   ```

2. Zur Dokumentation gibt's zwei Möglichkeiten:

   (a) Kurzbeschreibung bei Funktionsdeklaration und ausführliche Beschreibung Bei Funktionsdefinition. Etwa so:

   ```
   //! gauss(A,x,y,N) loest Ax=y fuer ein NxN-System
   void gauss(float *A, float *x, float *y, int N); //Deklaration
   /*!
     Der Gauss-Algorithums arbeitet mit einer Pivotisierung,
     falls Nulldiagonalelemente auftreten. Er bricht mit einer
     Fehlermeldung ab, falls die uebergebende Matrix singulaer ist.
   */
   ```

   (b) Man kann aber auch beide Beschreibungsvarianten kombinieren. Analog waere folgende Variante:

   ```
   /*!
     \brief gauss(A,x,y,N) loest Ax=y fuer ein NxN-System

     Der Gauss-Algorithums arbeitet mit einer Pivotisierung,
     falls Nulldiagonalelemente auftreten. Er bricht mit einer
     Fehlermeldung ab, falls die uebergebende Matrix singulaer ist.
   */
   ```

3. Strukturdokumentationen funktionieren genauso wie Funktionsdokumentationen. Das könnte in der Datei `inc/lgs.h` so aussehen (`/*! \file lgs.h */` nicht vergessen!):

```
/*!
  \brief Struktur fuer lineare Gleichungssysteme

  LGS mylgs; liefert die Komponenten

  mylgs.A: Ein Feld der Groesse mylgs.DIM^2, das zeilenweise
           die Matrix des Gleichungssystems beinhaltet.
  mylgs.x: Der Loesungsvektor (Laenge mylgs.DIM)
  mylgs.y: Die rechte Seite des Gelichungssystems (Laenge mylgs.DIM)
*/
typedef struct{
  REAL *A, *x, *y;
  int DIM;
} LGS;
```

D.4 Doxygen für Fortgeschrittene

Nun kann man sowohl für html als auch für Latex auch schön formatierte dokuemntationen erstellen:

D.4.1 Formatieren von Dokumentationen in html

In den Kommentaren für doxygen kann man mit html–Syntax schreiben, und somit die Dokumentation noch schöner und übersichtlicher gestalten.

Nehmen wir als Beispiel die Funktion `ReadOptions`, die die Eingabe der Optionen auf Kommandozeilenebene organisiert:

Wir wollen die Ausgabetabelle, die bei der `-h` ausgegeben wird erstellen:

```
/*!
  \brief Verwaltet \"Ubergabe der Kommandozeilenoptionen

  Die option -h erstellt folgende Ausgabe


<table border="0">
    <tr>
      <td align="left">option</td>
      <td align="center">:</td>
      <td align="left">description</td>
      <td align="right">[standard]</td>
    </tr>
    <tr>
      <td colspan="4"><hr></td>
    </tr>
    <tr>
      <td align="left">-h</td>
      <td align="center">:</td>
      <td align="left">help</td>
      <td align="right"></td>
    </tr>
    <tr>
      <td align="left">-a</td>
      <td align="center">:</td>
      <td align="left">choose kind of algorithm</td>
      <td align="right">[2]</td>
    </tr>
    <tr>
      <td align="left"></td>
      <td align="center"></td>
      <td align="left">1 Gauss</td>
      <td align="right"></td>
    </tr>
    <tr>
      <td align="left"></td>
      <td align="center"></td>
      <td align="left">2 Jakobi</td>
      <td align="right"></td>
    </tr>
    <tr>
      <td align="left"></td>
      <td align="center"></td>
      <td align="left">3 Gauss-Seidel</td>
      <td align="right"></td>
    </tr>
</table>

*/
```

D.4.2 Formatieren von Dokumenten für Latex

Kommentare, die Latex-Formatierungen enthalten sind mit `\f[ \f]` geklammert.

```
\f[
   <Text>
\f]
```

Ein Beispiel: In `Others/tools.c` könnte der Kommentar für `diffquad` folgendermaßen aussehen:

```
/*!
  \brief Differenz zweier Felder bezueglich der Euklidischen Norm

  \f[
        \|y-x\|_2 = \left(\sum_{i=1}^N(y_i-x_i)^2\right)^{\frac{1}{2}}
  \f]

*/
```

Die Latex-Dateien werden im Verzeichnis `doc/latex` erstellt, so wie wir es in `doxycongif` gewünscht haben und muss dort (es gibt ein Makefile) mit

```
make
```

noch übersetzt werden. Schlussendlich erhalten wir eine pdf–datei, die dann die fertige Dokumentation enthält. (Man beachte: Auch zum texen kann man `make` und `makefile` verwenden. Ein kleiner Hinweis für die letzten Skeptiker).

D.4.3 Einbinden von Bildern

Die Syntax zum Einbinden von Bildern (hier werden Standardformate erwartet, die Latex, html, etc. üblicherweise kennt) erfolgt mit dem Tag `\image` und folgender Syntax (im doxygen–Kommentar! Wie immer.):

Für html:

```
\image html <Bilddatei> ‘‘Überschrift’’ width=...
```

Für latex:

```
\image latex <Bilddatei> ‘‘Überschrift’’ width=...
```

Wenn der Pfad von Bilddateien in `doxyconfig` gesetzt wurde, sagen wir durch

```
IMAGE_PATH = doc/pics
```

dann genügt die Angabe der Bilddatei. Andernfalls muss der ganze Pfad eingesetzt werden. In unserem Fall wären also die Kommentare

```
\image latex doc/pics/sol_gauss.png ‘‘Lösung von Gauss’’ width=10cm
```

und

`\image latex sol_gauss.png ‘‘Lösung von Gauss’’ width=10cm`

gleichbedeutend.

`\image latex` setzt das Bild nur in die Latex–Dokumentation und `\image html` nur in die html–Dokumentation. Will man beides haben muss man auch beides angeben.

Nun noch ein letzte Kleinigkeit:

D.4.4 Aufräumen

Was nutzt uns, dass wir mühevoll ordentlich programmiert haben, Funktionen in verschiedene Dateien zerlegt haben, graue Haare beim Erstellen eines Makefiles erhalten haben, wenn wir nun vor lauter Dokumentiererei die Ordnung und übersicht in unseren Dateien verlieren?

Na?

Nichts natürlich!

Wir hätten das nicht alles gemacht, wenn es nicht eine Lösung für dieses Problem gäbe. Es ist Letztenendes nicht so, dass die Kommentare für die Dokumentation direkt bei den Funktionsdefinitionen stehen müssen. Mit Hilfe des Tags `\fn` kann man einer Kommentarumgebung klar machen zu welcher Funktion sie gehört und sie dann irgendwohin setzen. Irgendwohin beschränkt sich natürlich auf Dateien, die auch von doxygen gelesen werden. Aber das bestimmen wir ja in `doxyconfig`. Zum Beispiel so:

Die umfangreichen Dokumentationen wollen wir auslagern in ein Verzeichnis

`doc/des`

und eine Datei, die den Namen der Quellcode–Datei enthält und die Endung .des. Also die aufwändige Dokkumentation von `ReadOptions` zum Beispiel soll in

`doc/des/tools.des`

beschrieben werden, um die Übersicht im Programm wieder zu erhöhen.

Wir setzen in `doxyconfig`:

```
INPUT                  = ./ LGS inc Others doc/des
FILE_PATTERNS          = *.c *.h *.des
```

Jetzt werden in den angegebenen Verzeichnissen – so auch in `doc/des` – nach den Dateien mit Endung .des gesucht. Gut so.

Dann schreiben wir in die Datei `tools.des`

```
/*!
     \fn void ReadOptions(int argc,char *argv[],int *DIM,int *SOLVER,int
*LOOPMAX)
     \brief Verwaltet Übergabe der Kommandozeilenoptionen
     Die option -h erstellt folgende Ausgabe
     <table border="0">
          <tr>
            <td align="left">option</td>
            <td align="center">:</td>
            <td align="left">description</td>
            <td align="right">[standard]</td>
          </tr>
          <tr>
            <td colspan="4"><hr></td>
          </tr>
          <tr>
            <td align="left">-h</td>
            <td align="center">:</td>
            <td align="left">help</td>
            <td align="right"></td>
          </tr>
          <tr>
            <td align="left">-a</td>
            <td align="center">:</td>
            <td align="left">choose kind of algorithm</td>
            <td align="right">[2]</td>
          </tr>
          <tr>
            <td align="left"></td>
            <td align="center"></td>
            <td align="left">1 Gauss</td>
            <td align="right"></td>
          </tr>
          <tr>
            <td align="left"></td>
            <td align="center"></td>
            <td align="left">2 Jakobi</td>
            <td align="right"></td>
          </tr>
          <tr>
            <td align="left"></td>
            <td align="center"></td>
            <td align="left">3 Gauss-Seidel</td>
            <td align="right"></td>
          </tr>
     </table>
*/
```

Viel Spass damit!

E Zahlendarstellung – Bitumrechnung

1. Positive ganze Zahlen (unsigned integers)

Darstellung durch N Binärziffern x_i: $X = \sum_{i=0}^{N-1} x_i 2^i$

Wertigkeit	2^{N-1}	2^{N-2}	...	...	...	...	...	...	...	2^0
Stelle	x_{N-1}	x_{N-2}	...	...	...	...	...	...	...	x_0

Bereich: $X \in \left[0, 2^N - 1\right]$

Addition: Ziffernweise, beginnend bei der niederwertigsten Stelle wie dezimal (Strichrechnung), d. h. zur Berechnung von X + Y = Z wird $z_i = x_i + y_i + ü_i$ berechnet mit $ü_i$ Übertrag aus vorhergehender Stelle i – 1 auf Stelle i ($ü_0 = 0$).

Multiplikation: Wie dezimal durch Rückführung auf Addition.

Überlauferkennung: Übertrag aus MSB; OV = $ü_N$ (OV = Overflow).

2. Ganze Zahlen mit Vorzeichen (signed integers)

Darstellung im Zweierkomplement: $X = -x_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} x_i \cdot 2^i$

Wertigkeit	-2^{N-1}	2^{N-2}	...	...	...	...	...	...	...	2^0
Stelle	x_{N-1}	x_{N-2}	...	...	...	...	...	...	...	x_0

Bereich: $X \in \left[-2^{N-1}, 2^{N-1} - 1\right]$

Negation: $-X = \overline{X} + 1 - 2^N$

Sign extension: Bei Vergrößerung der Stellenzahl muss Vorzeichenbit in alle links hinzugekommenen Bits übernommen werden!

Addition: 1. sign extension, 2. Addition wie unsigned int. Achtung: Überlauf möglich!

Subtraktion: Addition eines negierten Operanden.

Überlauferkennung: Achtung: Überträge können Vorzeichen verfälschen!
OV = $ü_N \oplus ü_{N-1}$ ($\oplus$: Exklusiv-Oder)

F Weiterführende Literatur

Literatur

F.1 Mathematik

(1) Josef Stoer. *Numerische Mathematik 1*. Springer, 7. Auflage

(2) Josef Stoer, Roland Bulirsch. *Numerische Mathematik 2*. Springer, 3. Auflage

(3) William H. Press, Saul A. Teuklosky, William T. Vetterling, Brian P. Flannery . *Numerical Recipies in C*. Cambridge University Prees, Second Edition, 1992
Das Standardwerk zur numerischen Programmierung (in C). Es wird wohl nicht sehr viele nicht sehr speziellen Algorithmen geben, die hier nicht aufgeführt sind. Gibt es auch für FORTRAN und in der ersten Ausgabe auch noch für BASIC und Pascal.

F.2 C–Programmierung

(4) Brian Kernighan, Dennis Ritchie. *Programmieren in C*. Hanser, 2. Ausgaben ANSI C, 1990
Das Standardwerk. Kernighan und Ritchie sind die beiden Entwickler von C und damit kommt die Information komprimiert aus erster Hand. Es wird auch von fortgeschrittenen Programmiere als Nachschlagewerk benutzt, für absolute Programmier-Anfänger aber wohl nur bedingt geeignet.

(5) Peter van der Linden *Expert – C – Programmierung*. Heise, 1995
Laut eigener Aussage das zweite Buch was man über C lesen sollte. Es gibt locker und unterhaltsam geschrieben, Einblicke in C Interna. Es führt die Ecken und Kanten von C vor und bietet einen reichen Fundus an interessanten Geschichten.

F.3 UNIX und seine Werkzeuge

(6) Matt Welsh, Lar Kaufman. *Linux - Wegweiser zur Installation & Konfiguration*. O'Reilly, 2.Auflage, 1997
Linux ist eines der frei verfügbaren UNIX (artigen) Betriebssysteme. Wenn man mehr über Linux (UNIX) und seine Werkzeuge lernen will hilft dieses Buch, es leistet nicht nur Hilfe bei der Installation und Administration sondern gibt auch erste Einführungen in das Programmieren mit verschiedenen Programmiersprachen.

(7) compiled by Eric S. Raymond *The New Hackers Dictionary*. The MIT Press, 1996
Das Wörterbuch des Computer-Slangs aus 40 Jahren Computer-Geschichte.