A Notizen zu C

A.1 System-Header

Oft braucht man

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

Wie findet man ggf. heraus, was fehlt? Wenn z.B. eine Standardfunktion wie `printf` zu einer Fehlermeldung führt, dann sieht man in der Manpage nach:
`man 3 printf`
Sie liefert einen Hinweis auf #include <stdio.h>

A.2 Numerische Datentypen

Bevorzugte Datentypen für ganze und Kommazahlen:
`int` – man sollte `long` *vermeiden*, weil es unter 64-Bit–Systemen als 64-Bit–Integer implementiert ist.
`double` – denn `float` ist in vielen Fällen zu ungenau.
Man beachte, dass beim Einlesen (z.B. mit `scanf`) das Format `%lf` (anstelle von `%f`) benötigt wird. Übrigens ist das Format `%d` *nicht* für `double`–Variablen zu verwenden, vielmehr ist das ein Synonym für `%i` (für Typ `int`).
Komplexe Zahlen sind in ANSI C (ISO C90) nicht vorgesehen, sie sind erst über das Paket "`complex.h`" in C99 implementiert. Beschreibungen findet man mit
`man complex`
oder in `http://wikipedia.org/wiki/Complex.h`.
C++ bringt eine andere (inkompatible!) Implementierung komplexer Arithmetik.

A.3 Prototypen

Wenn ein Programmteil eine Routine aufruft, die erst weiter unten oder sogar in einer anderen Datei definiert ist, dann braucht der Compiler vorweg eine Information über die *Deklaration* der Routine, das ist die *Definition* ohne den Programmblock {..}. Man setzt z.B.

```
void laplace(int n, double *phi_in, double *phi_out);
```

vor den Programmteil, der die Routine aufruft.

A.4 Funktionsname als Argument

Es kommt vor, dass man im Aufrufargument eines Unterprogramms eine *andere* Routine namentlich angeben will, die dann vom Unterprogramm aufgerufen werden soll. Wie formuliert man das?

Man setzt in die Deklaration/Definition als *dummy argument* wiederum eine Deklaration, die das Aufrufformat klarstellt, z.B.

```
void cg(int n, double *x, double *b,
      void (*matvec)(int n, double *phi_in, double *phi_out),
      double relerr, int maxiter, int istart){
      ...
      matvec(n, p, s);
      ...
}
```

Hiermit ist `matvec` als *function pointer* eingeführt, ansonsten entspricht die Deklaration dem Prototyp von `matvec`.
Nun kann man im tatsächlichen Aufruf von `cg` etwa das oben genannte `laplace` für `matvec` einsetzen.

A.5 Speicherklassen von Variablen

A.5.1 Begriffe

Eine **Deklaration** legt den Typ (`int, float, double, ..`) einer Variablen fest, reserviert aber i.a. noch keinen Speicher. Wenn auch das der Fall ist, spricht man von einer **Definition**.
Einer Deklaration/Definition kann die Angabe einer **Speicherklasse** (`static, extern, ..`) vorangestellt werden. Das hat Auswirkungen auf die **Sichtbarkeit** (wo bezeichnet derselbe Name dieselbe Variable?) und die **Lebensdauer** der Variablen (wie lange behält sie ihren Wert?).
Deklarationen/Definitionen können entweder am Anfang eines Programmblocks {..} stehen oder in der Datei außerhalb (meist oberhalb) der Programmteile. Die Speicherklasse hat dabei ganz verschiedene Auswirkungen.

A.5.2 Deklaration im Dateikopf

Speicherklasse	Sichtbarkeit	Lebensdauer	Definition
static	Datei	statisch	ja
extern	global	statisch	nein
k.A.	global	statisch	ja

Mit `static` macht man also eine Variable in der ganzen Datei bekannt, mit `extern` oder ohne Angabe(!) wird sie sogar von anderen Dateien aus zugänglich! Ihr Wert ist immer statisch, d.h. er bleibt während der gesamten Laufzeit des Programms erhalten. Im Fall von `extern` wird *kein* Speicherplatz angelegt (*Deklaration*), sondern nur auf eine Variable verwiesen, die in einer *anderen* Datei *definiert* ist. Damit ist auch eine Initialisierung an dieser Stelle ausgeschlossen.

A.5.3 Deklaration am Anfang eines Blocks {..}

Speicherklasse	Sichtbarkeit	Lebensdauer	Definition
static	Block	statisch	ja
auto	Block	Block	ja
k.A.	Block	Block	ja

In der Praxis macht man hier normalerweise keine Angabe und *definiert* damit lokale Variablen, deren Werte bei Verlassen des Blocks ungültig werden (und bei erneutem Eintritt in den Block auf den Initialisierungswert zurückgesetzt werden). Wenn man den zuletzt zugewiesenen Wert für das nächste Mal erhalten will, deklariert man die Variable als `static`.

A.6 Dynamischer Speicher

In ANSI C (C90) legt man (globale oder lokale) Felder so an:

```
double v[n];
double A[m][n];
```

Die Speicherbelegung ist **statisch**: `m` und `n` müssen (zur *Compilezeit* bekannte) Konstanten sein.
Sind die Feldgrößen erst zur Laufzeit bekannt, dann deklariert man sie zunächst als Pointer (global oder lokal) und reserviert den Speicher **dynamisch** mit `malloc`, z.B.

```
double *v;
int n;
...
v = (double *) malloc(n * sizeof(double));
...
free(v);
```

Für eine Matrix (Feld mit zwei Indizes) ist das komplizierter, denn man muss sie als "Feld von Pointern" anlegen. Dafür gibt es zwei Möglichkeiten:

```
double **A;
int m, n, i;
...
A = (double **) malloc(m * sizeof(double *));
for (i=0; i<m; i++)
   A[i] = (double *) malloc(n * sizeof(double));
...
for (i=0; i<m; i++) free(A[i]);
free(A);
```

Hier wird zunächst ein Vektor von "Zeilenpointern" angelegt und dann jede Zeile einzeln reserviert. Dabei kann es passieren, dass die Matrix nicht fortlaufend im Speicher steht. Deshalb muss man sie auch einzeln freigeben. Besser ist

```
...
A = (double **) malloc(m * sizeof(double *));
A[0] = (double *) malloc(m*n * sizeof(double));
for (i=1; i<m; i++) A[i] = A[0] + n*i;
...
free(A[0]);
free(A);
```

Der Speicherplatz für die ganze Matrix wird im zweiten `malloc` zusammenhängend angelegt und an den Beginn der ersten Zeile gelegt, danach werden die übrigen "Zeilenpointer" geeignet gesetzt.
Auf dem Sprachniveau von C99 gibt es zusätzlich die Möglichkeit, **automatische Arrays** anzulegen:

```
void sub(int n, ...){
   double v[n];
   ...
}
```

Das hat dieselbe Form wie eine statische Definition, aber hier ist die Feldgröße `n` erst zur *Laufzeit* bekannt. Der Speicher wird bei Eintritt in den Programmblock automatisch angelegt und beim Verlassen wieder frei gegeben. Automatische Arrays sind bequem für lokale Arbeitsvektoren, man sollte sich aber bewusst sein, dass sie weder in ANSI C (C90) noch in Standard C++ zulässig sind.
Globale Felder, die außerhalb von Programmblöcken deklariert werden, kann man so natürlich nicht anlegen.

A.7 Mehrere Quelldateien

Es ist oft sinnvoll, den Quellcode eines Projekts auf mehrere Dateien zu verteilen. Gleichzeitig möchte man natürlich vermeiden, Änderungen immer an vielen Stellen gleichzeitig vornehmen zu müssen, denn das provoziert Fehler.
Man teilt den Code in Programmdateien (schematisch: `prog1.c, prog2.c, ..`) auf. Eine enthält das Hauptprogram (`main`), die anderen jeweils einige wenige (zusammengehörige) Unterprogramme.
Zu jeder Programmdatei `progn.c` schreibt man eine Headerdatei `progn.h`, die nur die Prototypen der in `progn.c` definierten Routinen versammelt. Dann muss jede Datei, die eine Routine aus `progn.c` benutzt, ein

```
#include "progn.h"
```

bekommen, und der Compiler ist zufrieden.
Beispiel:

```
-------------------------------------------------------
/*
  quickdirty.c

  quick-and-dirty random numbers
  cf. Numerical Recipes, eq.(7.1.1) and Table, p.198
  irand = (ia*irand+ic) % mr
*/
#include <stdlib.h>

static int        mr=714025, ia=1366, ic=150889;
static double     qnorm=1./714025;
static int        irand = 0;

double qdirty(void){
      irand = (ia*irand+ic) % mr;
      return qnorm*irand;
}

void setdirty(int iseed){
      irand = abs(iseed) % mr;
}
-------------------------------------------------------
/*
  quickdirty.h
*/
double qdirty(void);
void setdirty(int iseed);
-------------------------------------------------------
```

A.8 Globale Variablen

Wenn globale Variablen vorkommen, die über mehrere Dateien bekannt gemacht werden sollen, dann verfasst man dazu eine besondere Headerdatei, z.B.

```
----------------------------------------------------------------------
/*
  global.h
*/
#define MAXDIM 5
#define MAXVOL 100000

extern int        ndim, nvol, lsize[MAXDIM+1], nn[MAXVOL][2*MAXDIM+1];
```

```
extern double     mass;
-----------------------------------------------------------------------
```

Hier werden einige Preprozessor–Konstanten definiert, die zum Anlegen globaler Felder gebraucht werden (ANSI C / ISO C90 lässt sonst nur noch Speicherplatzvergabe über `malloc` zu). Die Variablen selber sind mit der Speicherklasse `extern` versehen, um mehrfache Definition zu vermeiden. Man fügt also (fast) überall ein:
`#include "global.h"`
An *einer* Stelle (z.B. oberhalb von `main`) muss aber wirklich Speicherplatz reserviert werden, deshalb ist dort das `extern` wegzugelassen. Bevor man nun eine zweite Version von `global.h` schreibt, hier eine Neufassung mit einem Preprozessor–Trick:

```
-----------------------------------------------------------------------
/*
  global.h  -- mit Trick
*/
#define MAXDIM 5
#define MAXVOL 100000

#ifdef DEFINE_GLOBAL
#define EXTERN
#else
#define EXTERN extern
#endif

EXTERN int        ndim, nvol, lsize[MAXDIM+1], nn[MAXVOL][2*MAXDIM+1];
EXTERN double     mass;
-----------------------------------------------------------------------
```

Dort, wo die globalen Variablen *definiert* werden sollen, schreibt man nun

```
#define DEFINE_GLOBAL
#include "global.h"
```

Damit wird in `global.h` die Präprozessor–Variable `EXTERN` auf den leeren String gesetzt, und die folgenden Deklarationen werden zu Definitionen. Andernfalls wird `EXTERN -> extern` substituiert, und es werden externe Variablen deklariert.

A.9 Compilieren und Linken

Im einfachsten Fall compiliert man eine Quelldatei `prog.c` mit

```
gcc prog.c -lm
```

und erzeugt so (wenn alles gut geht) ein lauffähiges Binärfile `a.out`
Tatsächlich besteht die Erzeugung des Binärfiles aus mehreren Schritten, die man auch einzeln aufrufen kann:

```
gcc -c prog.c              # compile prog.c -> prog.o
gcc -o prog.x prog.o -lm   # link prog.o -> prog.x
```

Das wird deutlicher, wenn man den Quellcode in mehreren Dateien `prog1.c prog2.c ..` vorliegen hat:

```
gcc -o prog.x *.c
```

erzeugt wiederum ein lauffähiges Binärfile `prog.x` in einem Schritt, während

```
gcc -c *.c
gcc -o prog.x *.o -lm
```

erstmal jedes Quellfile `progn.c` zu einem Objectfile `progn.o` compiliert und diese dann im zweiten Schritt zum Binärfile `prog.x` verlinkt.
Die Optionen sind jeweils entweder dem Compilier- der dem Linkschritt zuzuordnen.
Compileroptionen sind u.a.

```
-O            Optimierung einschalten (GNU!)
-Wall         alle Warnungen anzeigen (Fehlersuche!)
-c            nur compilieren, nicht linken
```

Einige Linkeroptionen:

```
-o <name>     Binaerfile umbenennen (statt a.out)
-lm           Math.-Bibl. einbinden (libm.a / libm.so)
```

A.10 Fehlersuche

C–Compiler unterstützen die Fehlersuche durch einige nützliche Optionen.

A.10.1 Undeklarierte Funktionen

Eine nicht deklarierte Funktion (fehlender Prototyp) erzeugt meistens eine Fehlermeldung beim Übersetzen. Manchmal reicht auch die (implizite) Deklaration als `int` aus, und das Programm zeigt seltsame Resultate. Das kann man mit einer Compileroption unterbinden.

A.10.2 Nicht initialisierte Variablen

Der Zugriff auf nicht gesetzte Variablen führt zu obskurem Verhalten, denn es wird der zufällig vorgefundene Speicherinhalt als Wert genommen. Der Compiler kann eine entsprechende Warnung geben.

A.10.3 Überschreiten der Feldgrenzen

Wenn ein Index die deklarierten Feldgrenzen überschreitet, wird auf einen willkürlichen benachbarten Speicherplatz zugegriffen (oder es gibt einen *segmentation fault*). Hier hilft nur eine genaue Untersuchung mit Debugger...

A.10.4 Übersicht

	gcc	icc	pgcc
undeklarierte Funktionen	-Wimplicit -Wall		
uninitialisierte Variablen	-Wuninitialized -Wall	-check-uninit	

A.11 Einhalten des Standards

So lässt man den Compiler kontrollieren, ob das Quellprogramm dem Standard (ANSI C = ISO C90; ISO C99) entspricht:

```
gcc -ansi -pedantic ..               # ANSI C (ISO C90)
gcc -std=c89 -pedantic ..            # dasselbe
gcc -std=c99 -pedantic ..            # ISO C99

pgcc -c89 ..                         # ANSI C
pgcc -c99 ..                         # ISO C99
```

Der `icc` kennt auch Optionen zum Sprachstandard:
`-ansi, -strict-ansi, -std=c89, -std=c99`,
benutzt sie aber nicht vorrangig zur Prüfung des Programms, sondern zur `gcc`–Kompatibilität beim Kompilieren.