EINFÜHRUNG IN C

S. BARTELS, 11.8.2014

I.A. **Struktur.** Die Programmiersprache C ist eine Compiler-basierte Sprache, das heißt mit einem Texteditor wie *emacs* oder *kate* erstellte Programme werden mit Hilfe des Compilers in einen maschinenlesbaren Code übersetzt. Damit dies fehlerfrei funktioniert, müssen Programme nach einem vorgegebenen Rahmen geschrieben werden. Ein C-Programm beginnt mit der Einbindung benötigter vordefinierter Routinen, die in Bibliotheken bereitgestellt werden, wie mathematischer Funktionen oder Ein- und Ausgabefunktionen. Es folgen optional selbstdefinierte Funktionen und am Ende steht das mit `main()` beginnende Hauptprogramm. Im Hauptprogram stehen Variablendefinitionen und Kommandos wie beispielsweise der Aufruf einer Funktion. Das linksseitig stehende Programm in Abbildung 1 zeigt ein einfaches Beispiel, in dem in einer Unterroutine das Quadrat einer Zahl berechnet wird. Diese wird vom Hauptprogramm aus mit einem Argument aufgerufen. Ist das Programm als Textdatei unter dem Namen `comp_square.c` abgespeichert, so kann es im selben Verzeichnis mit dem Unix-Kommando

```
gcc -lm comp_square.c -o comp_square.out
```

kompiliert werden. Mit dem Unix-Kommando

```
./comp_square.out
```

wird das Programm gestartet.

```c
// comp_squre.c
#include <stdio.h>
#include <math.h>
double square(double x){
  return pow(x,2.0);
}
main(){
  double x, y;
  x = 3.8;
  y = square(x);
  printf("square is %f\n",y);
}
```

```c
// simple_loop.c
#include <stdio.h>
main(){
  int i;
  for (i=0; i<5; i=i+1){
    printf("%d\n",i);
  }
  printf("\n");
  if (i==5){
    printf("i is 5 \n");
  }
}
```

ABBILDUNG 1. Elementare C-Programme.

I.B. **Bibliotheken.** Die Bibliothek `stdio.h` stellt Routinen zur Ein- und Ausgabe zur Verfügung, während in der Bibliothek `math.h` Implementationen elementarer mathematischer Funktionen realisiert sind. Die artihmetischen Grundoperationen `+,-,*,/` können ohne die Einbindung von Bibliotheken verwendet werden. Zur Darstellung von Gleitkommazahlen und ganzzahligen Maschinenzahlen wird das Kommando `printf("text %f\n",x)` beziehungsweise `printf("text %d\n",i)` verwendet, wobei `\n` einen Zeilenumbruch bewirkt. Das Einlesen von Werten für die Variablen erfolgt mit `scanf("%lf",&x)` beziehungsweise `scanf("%d",&i)`.

`printf`, `scanf`	Aus- und Eingabe von Text und Variablen
`/*...*/`, `//`	Kommentare
`cos`, `sin`, `tan`	trigonometrische Funktionen
`exp`, `log`, `log10`	Exponentialfunktion und Logarithmen
`pow`, `sqrt`	Potenz und Quadratwurzel
`floor`, `ceil`, `fabs`	Runden auf ganze Zahlen und Betragsfunktion

TABELLE 1. Ein- und Ausgabe, Kommentare sowie elementare mathematische Funktionen.

I.C. **Typen.** Jede Variable muss in C deklariert werden, das heißt ihr Typ wie *integer* oder *double* muss vor ihrer Verwendung in einem Unterprogramm oder im Hauptprogramm festgelegt werden. Auch die Verwendung von Zahlen und arithmetischen Operationen ist mit Typen verbunden, beispielsweise wird 2 als Variable vom Typ *integer* interpretiert, während 2. als Variable vom Typ *double* verwendet wird. Die Operation 2/3 wird in C als binäre Operation des höherwertigen Variablentyps ausgeführt, das heißt

$$2/3 = 0, \quad 2./3. \approx 0.\overline{6}, \quad 2/3. \approx 0.\overline{6}, \quad 2./3 \approx 0.\overline{6}.$$

Variablen können beispielsweise mit `x = (double)a` umgewandelt werden. Arrays fester Größe können über `double x[n]` oder `double A[m][n]` initialisiert werden. Die Indizierung der Einträge von Arrays beginnt mit 0. Eine falsche Indizierung von Arrays führt im Allgemeinen nicht zu einer Fehlermeldung und muss vom Programmierer ausgeschlossen werden. Bei der Deklaration einer Variablen kann bereits ein Wert zugewiesen werden, sofern es sich nicht um ein Array handelt, dessen Größe durch eine Variable definiert ist.

`int`	ganzzahlige Maschinenzahlen
`float`, `double`	Gleitpunktzahlen einfacher und doppelter Genauigkeit

TABELLE 2. Variablentypen in C.

I.D. **Kontrollanweisungen.** In C können Fallunterscheidungen und Schleifen in den in Abbildung 2 dargestellten Formaten realisiert werden. Dabei steht `cond` für eine logische Bedingung, die über eine logische Operation wie `a<b` definiert sein kann, während `statement` für eine Liste von Kommandos steht, die von geschwungenen Klammern eingefasst sind. Die Ausdrücke `init` und `step` stehen für eine Initialisierung wie `i=0` und eine Anweisung der Art `i=i+1`, deren Ausführung so lange wiederholt wird, wie die Bedingung `cond` beispielsweise `i<5` als wahr ausgewertet wird. Dabei wird zunächst `init` ausgeführt, dann `cond` überprüft, anschließend der Kommandoblock `statement` abgearbeitet und schließlich `step` ausgeführt, bevor wiederum die Bedingung `cond` ausgewertet wird und dieser Vorgang wiederholt wird, bis `cond` falsch ist. Gelegentlich ist auch die Verwendung der `do while`-Schleife sinnvoll, bei der die Bedingung im Unterschied zur `while`-Schleife nach statt vor der Ausführung der Kommandos überprüft wird.

```
if (condA) statementA else if (condB) statementB else statementC
while (cond) statement
for (init; cond; step) statement
```

Abbildung 2. Kontrollstrukturen in C.

I.E. **Logische Ausdrücke und Inkremente.** Zur Formulierung logischer Bedingungen stehen binäre Operationen zum Vergleich von Maschinenzahlen, die logischen Verknüpfungen *und*, *oder* sowie die Negation zur Verfügung. Vergleiche stehen dabei in Klammern also beispielsweise `(a<b)`. Gleitkommazahlen werden nur bis auf Maschinengenauigkeit verglichen und aufgrund möglicher Störungen ist ein Test auf exakte Gleichheit zweier Gleitkommazahlen wenig sinnvoll. Das Kommando `i=i+1` kann in C durch `i++` oder `++i` ersetzt werden und in arithmetischen oder logischen Ausdrücken verwendet werden. Im Fall `i++` wird die Variable zunächst erhöht und anschließend der Ausdruck ausgewertet, während bei Verwendung von `++i` zunächst die Variable erhöht wird. Die logischen Ausdrücke `(i++==1)` or `(++i==1)` führen also zu unterschiedlichen Ergebnissen.

`==`, `!=`, `>`, `>=`, `<`, `<=`	logischer Vergleich von Maschinenzahlen
`&&`, `||`, `!`	logische Verknüpfungen *und*, *oder* sowie Negation
`i++`, `++i`, `i--`, `--i`	Prä- und Postinkrement sowie -dekrement
`b+=x`	Kurzform für `b=b+x`

Tabelle 3. Logische Operationen sowie Inkrement- und Dekrementfunktionen.

I.F. **Funktionen.** Funktionen können entweder eine oder keine Variable zurückgeben. Wird ein Wert zurückgegeben, so steht der Typ des Funktionswerts vor dem Funktionsnamen, andernfalls wird `void` verwendet. Auf den Funktionsnamen folgt in Klammern eine Liste von Argumenten. Argumente einfachen Typs wie `double` und `int` werden mittels *call by value* behandelt, das heißt sie werden in eine lokale Variable kopiert. Die entsprechende Variable im Hauptprogramm bleibt dabei unverändert. Arrays sind nicht als Rückgabewert einer Funktion zugelassen. Gegebenenfalls werden daher Arrays an Funktionen als Argumente mittels *call by reference* übergeben und dabei als globale Variablen von dem Unterprogramm verändert. In dem in Abbildung 3 links gezeigten Programm wird das Array `x` an die Funktion `mod_vector` übergeben und dort unter dem Namen `vec` verwendet. Nach dem Durchlauf der Funktion sind die Werte des Arrays `x` verändert. Wichtig ist hierbei die Verwendung des Sterns bei der Deklarierung des Arguments der Funktion.

```c
// static_array.c
#include <stdio.h>
void mod_vector(double* vec){
  vec[0] = 2.0;
  vec[1] = 1.0;
}
main(){
  int n = 2;
  double x[n];
  x[0] = 1.0;
  x[1] = 2.0;
  mod_vector(x);
  printf("x[0] = %f\n",x[0]);
  printf("x[1] = %f\n",x[1]);
}
```

```c
// functions.c
#include <stdio.h>
void fun_1(double z){
  z = z+1.0;
}
void fun_2(double* z){
  *z = *z+1.0;
}
main(){
  double x = 1.0;
  fun_1(x);
  printf("x = %f\n",x);
  fun_2(&x);
  printf("x = %f\n",x);
}
```

ABBILDUNG 3. Übergabe von Arrays sowie einfachen Variablen und Pointern an Funktionen.

I.G. **Pointer.** Ein Pointer ist eine Variable, die die Adresse eines Abschnitts im Speicher enthält. Der Pointer erlaubt es, den Inhalt des entsprechenden Speicherabschnitts auszulesen oder zu verändern. Die Größe des Speicherabschnitts hängt davon ab, ob dort eine Gleitkommazahl oder ganzzahlige Maschinenzahl abgelegt werden soll. Ist `ptr` ein Pointer, so ist der Wert der Variable, die unter der in `ptr` enthaltenen Adresse zu finden ist, gegeben durch `*ptr`. Umgekehrt wird für eine gewöhnliche Variable `var` durch `&var` ein Pointer definiert, der die Adresse des entsprechenden Speicherplatzes enthält. Initialisiert wird ein Pointer beispielsweise mittels `double* ptr`. Übergibt man die Adresse einer Variable, das heißt den entsprechenden

Pointer, an eine Funktion, dann wird diese Variable mittels *call by reference* behandelt, so dass der Inhalt der Variablen mit globalen Auswirkungen manipuliert werden kann. Die oben beschriebene Übergabe von Arrays an Funktionen folgt gerade diesem Prinzip. In dem in Abbildung 3 rechts gezeigten Programm verändert die Funktion `fun_1` den Wert der Variablen `x` des Hauptprogramms nicht, während die Funktion `fun_2` deren Wert erhöht. Ein in einer Funktion definierter Pointer kann als Rückgabewert der Funktion verwendet werden. In diesem Fall muss die Funktion mittels `double*` deklariert werden.

I.H. **Dynamische Arrays.** Die oben beschriebene Verwendung von Arrays stößt an Grenzen, wenn die Dimension der Arrays erst im Laufe des Programmdurchlaufs bestimmt wird. In diesem Fall können dynamische Arrays verwendet werden, für deren Deklaration und Manipulation von der Bibliothek `stdlib.h` Routinen bereitgestellt werden. Ein Array kann als Kette von Pointern angesehen werden und es muss Speicherplatz für die entsprechenden Variablen reserviert werden. Ein Array kann damit einem Pointer `ptr` auf eine entsprechende Variable zugewiesen werden und mit `ptr[j]` erhält man die entsprechenden Variablen. Reservierter Speicher sollte stets wieder freigegeben werden.

`malloc`	Reservieren von Speicherplatz
`realloc`	neues Reservieren von Speicherplatz
`free`	Freigeben von Speicherplatz
`NULL`	Wert für Pointer ohne reservierten Speicherplatz

TABELLE 4. Kommandos zur Initialisierung und De-Initialisierung dynamischer Arrays.

I.I. **Arbeiten mit Matrizen.** Eine Matrix $A \in \mathbb{R}^{m\times n}$ kann mit einem Vektor $\widehat{A} \in \mathbb{R}^{mn}$ identifiziert werden, indem man die Spalten von A untereinander in einen Vektor schreibt. Es gilt, bei Numerierung der Einträge mit den Indizes $i = 0, 1, \ldots, m-1$ und $j = 0, 1, \ldots, n-1$, dass

$$A_{ij} = \widehat{A}_{i+jm}.$$

Mit dieser Identifikation lassen sich Matrizen in C wie Vektoren behandeln. Gelegentlich ist es übersichtlicher, Matrizen als zweidimensionale Arrays zu behandeln. Ist die Größe bekannnt, so kann man sie als Arrays verwenden wie beispielsweise in dem in Abbildung 5 gezeigten Programm.

I.J. **Zeitmessung, Speichern und Pakete.** (i) Die Bibliothek `time.h` stellt den Typen `clock_t`, das Kommando `clock()`, und die Konstante `CLOCKS_PER_SEC` zur Verfügung, mit denen Laufzeitmessungen durchgeführt werden können. Deren Verwendung ist in Abbildung 6 illustriert.
(ii) Zum Speichern von Daten können Befehle aus der Bibliothek `stdio.h`

```
// dynamic_vectors.c
#include <stdio.h>
#include <stdlib.h>
double* scan_vector(int n){
  int i = 0;
  double* vec = malloc(n*sizeof(double));
  for (i=0; i<n; ++i){
    printf("x[%d] = ",i);
    scanf("%lf",&vec[i]);
  }
  return vec;
}
void print_vector(double* vec, int n){
  int i = 0;
  for (i=0; i<n; ++i){
    printf("x[%d] = %f\n",i,vec[i]);
  }
}
main(){
  double* x = NULL;
  int dim = 0;
  printf("dim = ");
  scanf("%d",&dim);
  x = scan_vector(dim);
  print_vector(x,dim);
  free(x);
  x = NULL;
}
```

ABBILDUNG 4. Ein- und Ausgabe von Vektoren beliebiger Länge.

verwendet werden. Mit dem Variablentypen `FILE` und dem Kommando `fopen` kann ein Pointer auf eine Datei definiert werden, in die dann mit `fprintf` geschrieben werden. Ist das Schreiben beendet, so muss die Datei mit `fclose` geschlossen werden. Das in Abbildung 6 rechts gezeigte Beispielprogramm speichert einen Vektor in der Datei `var.dat` ab. Diese kann von MATLAB mit dem Kommando `load var.dat` ausgelesen werden und weist die Werte des Vektors der Variablen `var` zu.
(iii) Die Pakete `BLAS` und `LAPACK` stellen Implementationen numerischer Verfahren beispielsweise zur Lösung linearer Gleichungssysteme und Eigenwertaufgaben bereit.

```c
// matrix.c
#include <stdio.h>
const int m = 3;
const int n = 2;
void mod_matrix(double mat[m][n]){
  mat[0][0] = 7.0;
  mat[2][1] = 8.0;
}
main(){
  double A[m][n];
  A[0][0] = 1.0; A[0][1] = 2.0;
  A[1][0] = 3.0; A[1][1] = 4.0;
  A[2][0] = 5.0; A[2][1] = 6.0;
  mod_matrix(A);
  printf("A[0][0] = %f\n",A[0][0]);
  printf("A[2][1] = %f\n",A[2][1]);
}
```

ABBILDUNG 5. Verwendung zweidimensionaler Arrays.

```c
// runtime.c
#include <stdio.h>
#include <time.h>
main(){
  int i;
  double t, diff, x = 0.33;
  clock_t t1, t2;
  t1 = clock();
  for (i=0; i<10000000; i++){
    x*x;
  }
  t2 = clock();
  diff = (double)(t2-t1);
  t = diff/CLOCKS_PER_SEC;
  printf("runtime = %fs\n",t);
}
```

```c
// save_data.c
#include <stdio.h>
main(){
  FILE* file;
  int i;
  double x[3] = {0.0,1.0,2.0};
  file = fopen("var.dat","w");
  if (file==NULL){
    printf("file error");
  }
  for (i=0; i<3; i++){
    fprintf(file,"%f\n",x[i]);
  }
  fclose(file);
}
//
```

ABBILDUNG 6. Laufzeitmessung und Speichern von Daten.