1 Die Programmiersprache C

Dieser Abschnitt gibt eine kleine Einführung in die Programmiersprache C. Die dargestellten Information umfassen bei weitem nicht alle Möglichkeiten, die C bietet. Sie dienen lediglich dazu, soviel Information zu vermitteln, dass die in der Vorlesung auftretenden Algorithmen in C programmiert werden können. Es wird empfohlen, weiterführende Literatur dazu zu lesen, wie zum Beispiel das Standardwerk von Kernighan und Ritchie [1].

1.1 Der C-Zeichensatz

Folgende Zeichen sind in C erlaubt:

```
A-Z a-z 0-9 , . ; : ? ' " () [] {} <> !
| / \ ~ _ # % & ^ * - = +
```

In Kommentaren, Strings und Charakters sind hingegen alle darstellbaren Zeichen erlaubt.

White-Space und Escape-Folgen

Vereinfacht ausgedrückt sind das Zeichen, die nichts Schwarzes erzeugen. Sie werden ignoriert, sind aber dazu da sog. **tokens** (deutsch Zeichen) zu trennen. Hierzu zählen das Leerzeichen (engl. blank), das Tabulator-Zeichen sowie die Return bzw. Enter-Taste. In C werden diese Zeichen wie folgt ausgedrückt, vgl. dazu den Abschnitt über Strings.

```
        /* Leerzeichen */
\t      /* horizontal tab */
\n      /* new line */
\v      /* vertical tab */
```

Charakter-Konstanten

Charakterkonstanten werden innerhalb von einfachen Anführungszeichen geschrieben und belegen 1 Byte Speicher. Beispiele

```
'a' '0' '\0' '\017' '\x61'
```

Unterscheide den Charakter '0' von der Zahl 0.

String-Konstanten

String-Konstanten sind eine Folge von Charakters in doppelten Anführungszeichen eingeschlossen. Beispiele:

```
"Folge von Zeichen"        /* besteht aus 17 Zeichen */
"adam und eva"             /* besteht aus 12 Zeichen */
"adam und eva\n"           /* besteht aus 13 Zeichen */
printf ("adam und eva\n")  /* druckt adam und eva plus
                              new line */
```

Beachte, eine String-Konstante benötigt (Anzahl Charakters + 1) Bytes an Speicher, denn jeder String wird mit '\0' abgeschlossen, um das Ende des Strings anzugeben.

Ganze Zahlen

```
0 1 15 -17
```

Ganze Zahlen belegen in der Regel 4 Bytes und können im Bereich von -2147483647 und $+2147483647$ auftreten.

Gleitpunktzahlen

```
3.14
0.314e1
[-]d.d...d[e[+-]d...d]
```

Gleitpunktzahlen belegen in der Regel 8 Bytes. Es kann sowohl 'e' als auch 'E' als Symbol für den Exponenten verwendet werden. Beachte, 'e' ist nicht die Eulerzahl, sondern Symbol für die Potenz zur Basis 10, also zum Beispiel $0.314e1 = 0.314 \cdot 10^1 = 3.14$.

Variablennamen

Variablennamen beginnen mit einem Buchstaben oder Unterstreichungszeichen (engl. underscore) '_' und enthalten ansonsten Buchstaben, '_' und Ziffern. Beispiele:

```
_XYZabc
Adam_und_Eva
x01
```

Beachte, dass in `C` zwischen Groß- und Kleinbuchstaben unterschieden wird, also `x` und `X` sind zwei verschieden Variablen. In der Regel werden in `C` Variablennamen mit Kleinbuchstaben und Konstanten mit Großbuchstaben geschrieben.

Schlüsselwörter

Schlüsselwörter sind Wörter, die in `C` eine feste Bedeutung haben und die nicht als Variablennamen verwendet werden dürfen. Dazu zählen:

```
if else
for switch goto
while do extern
break continue
```

Typbezeichnungen

Jede Variable, die man in `C` verwenden will, erhält einen festen Typ, der vor Verwendung der Variablen festgelegt werden muss. Die Wichtigsten sind:

```
char          /* Buchstaben */
int           /* ganze Zahlen */
double        /* Gleitpunktzahlen */
unsigned int  /* nat"urliche Zahlen */
void          /* kein oder unbestimmter Typ */
```

Neben obigen Standardtypen kann man auch eigene Typen definieren. Wie man das macht, werden wir später sehen.

Kommentare

Kommentare haben wir oben bereits vereinzelt verwendet. Sie beginnen mit der Zeichenfolge '/*' und enden mit der Zeichenfolge '*/'.

```
/* ..... */
```

Ein Kommentar verhält sich wie ein White-Space.

1.2 Aufbau eines C-Programms

Ein C-Programm ist aus einzelnen *Funktionen* aufgebaut. Der Aufbau einer Funktion ist wie folgt:

```
Returnparametertyp Funktionsname (Parameterliste)
{
   Variablendeklaration;
   Anweisungen;
   return (R"uckgabewert);
}
```

Der `Returnparametertyp` muss einer der oben genannten Typen sein. Innerhalb einer Funktion darf man weitere Funktionen aufrufen. Eine Verschachtelung von Funktionen ist jedoch nicht erlaubt. Funktionen können auf mehrere Dateien verteilt sein. Genau eine Funktion muss den Funktionsnamen `main` haben. Zunächst ein einfaches Beispiel:

```
int main (void)
{
   printf ("Hello World\n");
   return (0);
}
```

Dateien, die C-Funktionen enthalten, nennt man *Quelldateien* (engl. *Source-Files*). C-Dateien müssen mit der Endung `.c` enden. Schreiben wir obigen C-Code in eine Datei namens `hello.c` und führen das Shell-Kommando

```
$ make hello
$
```

aus, so erhalten wir ein ausführbares Programm namens `hello`. Rufen wir dieses Programm auf, so ergibt sich:

```
$ hello
Hello World
$
```

Bevor wir uns komplexere Beispiele anschauen können, müssen wir noch einige Begriffe klären.

Anweisungen

Anweisungen haben immer folgende Form:

```
<statement> := <expression>;
```

das heißt eine Anweisung ist ein Ausdruck, der mit einem Strichpunkt abgeschlossen wird. Beispiele für Ausdrücke sind:

```
i = 5
a = b + c
a < b
```

Operatoren

(a) Arithmetische Operatoren

```
+  -  *  /
%              /* modulo */
```

(b) Vergleichende und logische Operatoren

```
>  >=  <  <=
==  !=          /* gleich ungleich */
&&              /* AND */
||              /* OR */
```

(c) Zähloperatoren

```
++          /* addiert 1 auf eine Variable */
--          /* subtrahiert 1 von einer Variablen */
```

Beachte, dass diese Operatoren sowohl einer Variablen vorangestellt als auch nachgestellt sein können. Im ersten Fall wird zuerst die Variable erhöht bzw. erniedrigt und dann die Anweisung ausgeführt, im zweiten Fall ist es umgekehrt. Im folgenden Beispiel

```
i = 5;
j = ++i;
k = i++;
```

ist $j = 6$ und $k = 5$.

(d) Bitweise Operatoren

```
<<  >>          /* left and right shift */
&               /* AND */
^               /* XOR */
|               /* OR */
~               /* Komplenent */
```

Zuweisungen

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

Blöcke

Ein *Block* wird durch die Geschweifteklammer Auf und Zu definiert.

```
{
   ...
}
```

Jede Funktion kann als Block aufgefasst werden, dem ein Name gegeben wurde und dem Werte durch Parameter übergeben werden können, siehe obiges Beispiel `main`. Blöcke können beliebig geschachtelt werden:

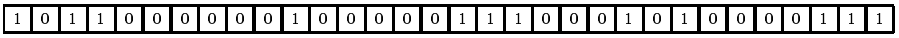
```
{
   {
      {
         ...
      }
   }
}
```

Variablendeklaration

Um Variablen in einem `C`-Programm zu verwenden, müssen sie vorher deklariert werden (im Gegensatz zum Beispiel zu `Matlab`). Dies geschieht auf folgende Art und Weise:

```
int n;
int i, j;
double x;
double y = 1.0;
char a = 'a';
```

Beachte, den Variablen `y` und `a` wurde gleich ein Wert zugeordnet, alle anderen obigen Variablen sind nicht initialisiert. In dem für die jeweilige Variable reservierten Speicherplatz kann eine beliebige Bitfolge stehen, zum Beispiel

n → 

In anderen Worten, n hat den Wert

$$\begin{aligned} n &= -(2^{29} + 2^{28} + 2^{21} + 2^{15} + 2^{14} + 2^{13} + 2^9 + 2^7 + 2^2 + 2^1 + 2^0) \\ &= -807461511 \end{aligned}$$

Die Variablendeklaration erfolgt immer am Anfang eines Blocks. Variablen sind nur gültig innerhalb dieses Blocks. Ausnahmen bilden Variablen, die außerhalb von Funktionen deklariert werden. Sie sind gültig vom Zeitpunkt der Definition bis ans Ende der jeweiligen Datei.

Kontrollkonstrukte

Alternativen

```
if ( expression ) statement 1
else              statement 2


if      ( expression ) statement 1
else if ( expression ) statement 2
....
else                   statement 3


switch ( expression )
{
   case const-expr: statements
   ....
```

```
   case const-expr: statements
   default:         statements
}
```

Beachte, dass jedes einzelne Statement auch durch einen Block ersetzt werden kann. Beachte im `switch`-Statement, dass, wenn der Ausdruck den Wert von `case` i hat, alle Statements von i bis zum Ende ausgeführt werden. Um dies zu vermeiden, muss an entsprecheder Stelle eine `break`-Anweisung erfolgen. Zum Beispiel wird mit

```
switch ( i % 2 )
{
   case 0: break;
   case 1: i++;
           break;
}
```

jede ganze Zahl auf die nächste gerade Zahl aufgerundet, während mit

```
switch ( i % 2 )
{
   case 0:
   case 1: i++;
}
```

jede Zahl um eins erhöht wird.
Beachte auch, dass es eine `default`-Anweisung nicht geben muss. Ist sie vorhanden, so werden die `default`-Anweisungen immer dann ausgeführt, wenn keiner der vorher explizit aufgeführten Ausdrücke zutrifft.

Schleifen

```
for ( initial; expression; incremental )
   statement


while ( expression )
   statement
```

```
do
   statement
while ( expression );
```

Mit `break` kann eine Schleife verlassen werden. Mit `continue` wird der nächste Schleifendurchlauf begonnen.

Arrays

```
int  a[10];         /* legt Speicher f"ur 10 Integer an */
int  b[];           /* Zeigerdeklaration,
                       kein Speicherplatz angelegt */
double A[12][10];   /* eine 12x10 Matrix wird angelegt */
```

Beachte, dass in `C` die Indizierung von 0 bis $n-1$ erfolgt. Ein kleines Beispiel:

```
/* count_loops counts the number of loops in a graph.
   The graph contains m edges represented in the
   arrays head[] (heads) and tail[] (tails of the edges) */

int count_loops (int head[], int tail[], int m)
{
   int e;      /* Index-Variable */
   int loops;  /* Anzahl Schleifen */

   loops = 0;
   for (e = 0; e < m; e++) {
      if ( tail[e] == head[e] ) loops++;
   }

   return (loops);
}
```

Übung. Schreiben Sie ein Programm, dass die Anzahl paralleler Kanten (Bögen) in einem Graphen zählt. Welche Laufzeit hat Ihr Programm?

Pointer

Pointer (zu deutsch: Zeiger) sind Variablen, die eine Speicheradresse auf eine bestimmte Variable enthalten, nicht jedoch den Wert der Variablen selbst. Pointer werden durch einen '*' deklariert. Beispiele:

```
int    *p;
double *q;
```

`p` ist also eine Variable, die die Speicheradresse einer `int`-Variable enthält. Genauso ist `q` eine Variable, die die Speicheradresse einer `double`-Variablen enthält. Beachte, dass jede deklarierte Variable Speicher in Anspruch nimmt, vgl. obiges Kapitel. Mit dem Konzept von Pointern erfährt man, wo die Variable im Speicher steht. Mit dem Operator '&' erhält man die Adresse zu einer Variablen. Mit '*' bekommt man zu einem Pointer den Wert der Variablen, auf die der Pointer zeigt. Beispiel:

```
int  i, j;
int  *p;

i = 5;
p = &i;   /* p enth"alt nun die Adresse der Variablen i */
j = *p;   /* j erh"alt den Wert der Variablen, auf die
             p zeigt, also in diesem Fall den Wert 5    */
```

Pointer sind ein sehr wichtiges Konzept in `C`. Zum Beispiel kann damit dynamisch Speicher allociert werden. In den (kleinen) Programmen, die wir in der Vorlesung behandeln, werden wir ohne dynamische Speicherallocierung auskommen. Für große Programmpakete und für Programme, die an die Grenzen der Speicherkapazität des Computers kommen, ist dieses Konzept jedoch unabdingbar.
Pointer sind auch noch in einem weiteren Zusammenhang von Bedeutung, nämlich beim Aufruf von Funktionen. Beim Aufruf einer Funktion wird von den Übergabeparametern jeweils eine Kopie angelegt und nicht die „Originalparameter“ übergeben. Das heißt, selbst wenn in einer Funktion ein Parameter verändert wird, behält er in dem die Funktion aufrufenden Programm den ursprünglichen Wert bei. Betrachten wir dazu noch einmal unser Beispiel `count_loops` in leicht modifizierter Form:

```
int main ()
{
   int head[5] = { 0, 1, 3, 4, 4};
   int tail[5] = { 0, 1, 1, 4, 2};
   int loops = 0;
```

```
   count_loops (head, tail, 5, loops);
   printf ("Nr of loops %d\n", loops);

   return 0;
}

void count_loops (int head[], int tail[], int m, int loops)
{
   int e;      /* Index-Variable */

   loops = 0;
   for (e = 0; e < m; e++) {
      if ( tail[e] == head[e] ) loops++;
   }

   return;
}
```

Obiges Programm wird immer den Wert 0 ausgeben (Überprüfen Sie das). Um dies zu umgehen, muss mit Pointern gearbeitet werden. Man übergibt nicht die Variable selbst, deren Inhalt man ändern möchte, sondern die Adresse dieser Variable. In der Funktion selbst, die eine Kopie der Adresse der Variablen bekommt, arbeitet man nun mit dem Inhalt dieser Zeigervariablen:

```
int main ()
{
   int head[5] = { 0, 1, 3, 4, 4};
   int tail[5] = { 0, 1, 1, 4, 2};
   int loops = 0;

   count_loops (head, tail, 5, &loops);
   printf ("Nr of loops %d\n", loops);

   return 0;
}

void count_loops (int head[], int tail[], int m, int *loops)
{
   int e;      /* Index-Variable */
```

```
    *loops = 0;
    for (e = 0; e < m; e++) {
       if ( tail[e] == head[e] ) (*loops)++;
    }

    return;
}
```

In diesem Fall wird nun der richtige Wert, nämlich 3 ausgegeben.

1.3 Präprozessor

Ein Präprozessor führt Substitutionen von Makros durch, ermöglicht eine konditionale Compilierung und ermöglicht die Einbeziehung anderer Dateien. Alle Präprocessor-Anweisungen beginnen mit einem '#'. Hier die für uns wichtigsten Anweisungen:

```
#include <stdio.h>
#include "./my.h"

#define n  100
#define m  300
#define PI 3.14159

#if condition
C-Code
#else
C-Code
#endif
```

Mit der `#include`-Anweisung können weitere `C`-Funktionen eingebunden werden, z.B. enthält `stdio.h` Funktionen, die das Bearbeiten von Dateien ermöglichen und die von `C` standardmäßig zur Verfügung gestellt werden. Entsprechend kann man eigene Dateien einbinden, wie mit `my.h` angedeutet wird. `#define` bietet eine weitere Möglichkeiten Konstanten zu definieren, die von der Stelle der Deklaration bis ans Ende des Files bzw. Blocks gelten. Damit lassen sich auch einfach bedingte Übersetzungen durchführen, z.B. kann es sein, wenn man in der Entwicklungsphase eines Programms ist, das man

deutlich mehr Ausgabe haben möchte, die ansonsten nur störend wirkt. In diesem Fall kann man z.B. schreiben

```
#ifdef DEBUG
printf ("Blablabla ...\n");
#endif
```

Durch vorherige Definition bzw. Nichtdefinition der Konstanten `DEBUG` wird dann die Print-Anweisung ausgeführt oder nicht.

```
#define DEBUG   /* oder alternativ */
#undef  DEBUG
```

1.4 Ein- und Ausgabe

in diesem Abschnitt beschäftigen wir uns noch mit den Möglichkeiten, wie in `C` Daten aus Dateien (engl. Files) eingelesen und ausgeschrieben werden können. Die wichtigsten Funktionen hierzu sind, wie man ein File öffnet und schließt und wie man daraus Daten liest bzw. Daten in ein File schreibt und an eines anhängt. Für uns von Interesse sind nur Textfiles, die ASCII-Zeichen enthalten (darüber hinaus gibt es noch Binärfiles wie zum Beispiel ausführbare Programme).

Öffnen und Schließen von Dateien

```
FILE *fopen (pathname, type);
```

`fopen` liefert einen Pointer auf die eingerichtete File-Struktur. `pathname` bezeichnet dabei den Pfadnamen der Datei, die man bearbeiten möchte. Der `type` gibt an, in welcher Form die Datei bearbeitet werden soll. Möglich sind

```
"r"   /* "Offnen zum Lesen */
"w"   /* "Offnen zum Schreiben, l"oscht alten Inhalt */
"a"   /* "Offnen zum Anh"angen */
"r+"  /* "Offnen zum Aktualiseren, Schreiben und Lesen */
"w+"  /* wie "w" */
"a+"  /* wie "a" */
```

Die Funktion `fopen` gibt `NULL` zurück, falls die Datei nicht in dem gewollten Sinne geöffnet werden konnte. Dateien, die immer offen sind, sind

```
stdout  /* Standardausgabe, i.d.R. die Konsole */
stdin   /* Standardeingabe, i.d.R. die Konsole */
stderr  /* Standardfehlerausgabe, i.d.R. die Konsole */
```

Mit der Funktion

```
int  fclose (FILE *file);
```

schließt man ein File wieder. Dies sollte spätestens am Ende eines Programms geschehen. Die wichtigste Funktion zum Schreiben von Daten in ein File ist die Funktion

```
int fprintf (file, formated string);
```

Hier ein kleines Beispiel:

```
#include <stdio.h>

int main (void)
{
   FILE  *outfile = NULL;
   int   i = 5;
   double x = 3.14;

   outfile = (FILE *)  fopen ("Ausgabe", "w");
   if ( outfile == NULL ) {
     fprintf (stderr, "Could not open file 'Ausgabe'\n");
     return 1;
   }

   fprintf (outfile, "Dies ist meine Ausgabedatei.\n");
   fprintf (outfile, "Zum Test schreibe ich");
   fprintf (outfile, " folgende Zahlen rein:\n");
   fprintf (outfile, "%d ist eine ganze Zahl\n", i);
   fprintf (outfile, "%lf ist eine Gleitkommazahl\n", x);
   fprintf (outfile, "'%s' ist ein String\n", "Hallo");

   fclose (outfile);
}
```

Obiges Beispiel generiert eine Datei namens 'Ausgabe' in dem Verzeichnis, in dem das Programm aufgerufen wurde, mit folgenden Inhalt.

```
Dies ist meine Ausgabedatei.
Zum Test schreibe ich folgende Zahlen rein:
5 ist eine ganze Zahl
3.140000 ist eine Gleitkommazahl
'Hallo' ist ein String
```

Beachte, eine Datei ist eine lineare Anordnung von Bytes, numeriert von 0 bis Dateilänge - 1. Mit dem Öffnen der Datei erhält man einen Filepointer, der zu Beginn am Anfang des Files steht, also auf 0. Mit den `fprintf`-Anweisungen wird immer an der Stelle angehängt, an der der File-Pointer gerade steht, also in obigen Fällen immer am Ende des Files.
Möchte man eine Datei zum Lesen öffnen, so geschieht dasselbe. Man erhält einen File-Pointer, der am Anfang auf 0 steht und man kann immer nur von der Stelle lesen, an der man gerade steht. Die wichtigste Funktion zum Lesen aus Dateien ist

```
int  fscanf (file, formated string);
```

Um zum Beispiel die ganze Zahl zu lesen, die in der Datei 'Ausgabe' steht, muss folgendes getan werden.

```
#include <stdio.h>

#define  STRING_SIZE  80

int main ()
{
   char s[STRING_SIZE];
   FILE *fin = NULL;
   int i;

   fin = (FILE *) fopen ("Ausgabe", "r");
   if ( fin == NULL )
   {
      printf ("Cannot open file 'Ausgabe'\n");
      return 1;
   }
```

```
    do
    {
       fscanf (fin, "%s", s);
    }
    while ( s[0] != 'r' );

    if ( strcmp (s, "rein:") )
    {
       printf ("Could not find word 'rein:'\n");
       return 1;
    }

    fscanf (fin, "%d", &i);

    printf ("Die ganze Zahl %d wurde ", i);
    printf ("in der Datei 'Ausgabe' gefunden.\n");

    return 0;
}
```

Literatur

[1] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.