**Einführung in die Programmiersprachen C und C++**

Dr. Franz Gähler, Fakultät für Mathematik

Übungsblatt 3 (8 Seiten)

**Scope- oder Sichtbarkeitsregeln:**

Variablen in C werden zu Beginn einer zusammengesetzten Anweisung deklariert (oder definiert), also etwa innerhalb des `main`-Blocks oder innerhalb einer Funktion, oder aber sie werden außerhalb von `main` und außerhalb jeder Funktion definiert.
Im letzten Fall sind sie "sichtbar", also les- und (falls nicht `const`) schreibbar für alle Programmteile in der gleichen Datei, die der Deklaration folgen.
Im ersteren Fall sind sie sichtbar für die gesamte zusammengesetzte Anweisung, zu deren Anfang die Deklaration (Definition) erfolgte.
Von diesen Regeln gibt es folgende Abweichung: Wird in einer zusammengesetzten Anweisung eine Variable oder Konstante deklariert (definiert), die mit gleichem Namen schon außerhalb dieser Anweisung sichtbar war, so wird die Sichtbarkeit der letzteren aufgehoben und die neue Deklaration bekommt für die aktuelle zusammengesetzte Anweisung Gültigkeit. Beim Verlassen dieser zusammengesetzten Anweisung wird die Variable außerhalb wieder sichtbar.

Beispiel:

```
int function (int a, int b) {
    int i=0;
    ....
    ....
    if (...) {
       int i=1; /* Ab hier ist i eine neue "Instanz"
                   das i ausserhalb ist durch die neue Deklaration
                   "ueberschattet". */
       ....
       ....
    } /* Ab hier hat i wieder die Bedeutung, die es vor dem "if" hatte
    ....
    ....
}
```

Hier ein Programmbeispiel:
Das Programm `scope.c` auf der nächsten Seite hat folgende Ausgabe:

```
  i =   13 innerhalb von main

  i =  200 innerhalb der Funktion test(1)

       225 Rueckgabe von test(1)

  Jedoch: Innerhalb der eingeschlossenen zusammengesetzten Anweisung:
  i =    0
  i =    1
  i =    2

  i =   13 innerhalb von main und ausserhalb der eingeschlossenen
                  zusammengesetzten Anweisung

  i =  225 innerhalb der Funktion test(2)

       250 Rueckgabe von test(2)
```



```
/* scope.c, Programm zum Studieren/Demonstrieren von "Scope"- oder
   "Sichtbarkeits"-Regeln */

#include <stdio.h>

int i = 200;
int test(int);

int main() {
  /*---------------------------main-----------------------------------------------*/
  int i = 13;  /* Das i von ausserhalb wird hier durch das neue i "ueberschattet".*/

  printf("\n");
  printf("  i = %4d innerhalb von main\n\n", i);
  printf("      %4d Rueckgabe von test(1)\n\n", test(1));

  { /*---------eingeschlossene zusammengesetzte Anweisung-------------------------*/
    int i;
    /* Wieder eine neue "Instanz" von i, sie ueberschattet die beiden aeusseren.  */

    printf("  Jedoch:Innerhalb der eingeschlossenen zusammengesetzten Anweisung:\n");
    for (i=0; i < 3; i++) {
      printf("  i = %4d\n", i);
    }
  } /*---Ende der eingeschlossenen zusammengesetzten Anweisung---------------------*/
  printf("\n");
  printf("  i = %4d innerhalb von main und ausserhalb der eingeschlossenen\n\
                    zusammengesetzten Anweisung\n\n",i);

  printf("      %4d Rueckgabe von test(2) \n\n", test(2));
  /*-------------------------Ende von Main-----------------------------------------*/
}

/*----------- Ab hier wieder Variablen ausserhalb von Main sichtbar ---------------*/
int test(int a) {
  /* Diese Funktion inkrementiert den Wert von i aus der "aeusseren" Definition ---*/

  printf("  i = %4d innerhalb der Funktion test(%d)\n\n",i,a);

  i = i + 25;  /* Hier wird ein "Seiteneffekt" des Aufrufs von test() erzeugt -----*/

  return i;
}
```

*Aufgabe 3.1: Wie sähe die Ausgabe des Programms aus, wenn zu Beginn des Programms* `test()`
*i) eine Deklaration* `int i;`
*ii) eine Definition* `int i = 10;`
*stehen würde?*
*(Erst überlegen, dann ausprobieren, dann das Ergebnis erklären!)*

**Arrays (= Felder), Strings und Pointer (=Zeiger)**

Beispiele zur Initialisierung von Arrays:

```
unsigned int prim[] = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

definiert ein Array `prim` von 8 Variablen `prim[0],..., prim[7]` vom Typ `unsigned int`. Der Größenoperator `sizeof(prim)` liefert den Wert `32 = 8 * sizeof(int)`. Der Ausdruck

```
char wort[] = "Hallo\n";                                    (1)
```

ist Abkürzung für

```
char wort[] = { 'H', 'a', 'l', 'l', 'o', '\n', '\0'};
```

Es wird bei der Notation mit `"..."` wie in (1) stets ein abschließendes `'\0'`-Zeichen eingebettet. Daher liefert `sizeof(wort)` den Wert 7, während `strlen(wort)` den Wert 6 zurückgibt. Es ist z. B. `wort[1] == 'a'`.

Ein **String** in C ist stets ein Array vom Typ `char`, dessen Elemente aus Zeichen (z.B. Buchstaben) bestehen, und deren letztes Zeichen stets das `'\0'`-Zeichen mit dem numerischen Wert 0 ist.

Die Syntax zur Definition eines Arrays von Strings wird klar aus folgendem Beispiel:

```
char *tage[] = { "Sonntag", "Montag", "Dienstag",
                 "Mittwoch", "Donnerstag", "Freitag", "Samstag" };
```

Jeder Tagesname wird seinerseits als Array von `char` angelegt, wobei jedes Array als letzten Eintrag den `'\0'`-Wert erhält.

Z. B. wird durch

```
printf("%c\n", tage[3][6]);
```

der siebte Buchstabe des vierten Tagesnamens ausgedruckt, also das `c`. Wenn das abschließende `'\0'`-Zeichen im String `"Mittwoch"` ausgegeben werden soll, muss beachtet werden, dass es ein nicht druckbares Zeichen ist, also nicht als `char`, sondern als Zahlenwert mit der Notation

```
printf("%d\n", tage[3][8]);
```

ausgegeben werden muss.

*Aufgabe 3.2: Erweitern Sie ein Programm aus Aufgabe 2.5, das auf die Eingabe eines Datums, etwa "29.2.2008", antwortet mit: "29. Februar 2008" oder analog, so dass unzulässige Daten wie 32.5.2008, 29.2.1900 usw. vom Programm abgelehnt werden. Noch einmal für Mutige: Erweitern Sie das Programm, so dass die Ausgabe "Freitag, 29. Februar 2008" oder analog ist.*

Variablen haben **Adressen**, das sind ganze Zahlen, die den Ort im (virtuellen) Speicher bezeichnen, der für den Variableninhalt vom Rechner vorgesehen ist. Der Speicher ist in fortlaufende Bytes unterteilt (ein Byte besteht in der Regel aus 8 Bit), die Bytes sind fortlaufend numeriert, die einem Byte zugeordnete Nummer ist die Adresse dieses Bytes und auch jeder mit diesem Byte beginnenden Folge von Bytes. In diesem Sinne lässt sich der ganze Speicher als ein Array von `char` auffassen.

Die Adresse einer Variablen ist somit die Nummer des ersten Bytes der Bytefolge, die den Inhalt dieser Variablen enthält.

Die Adresse der Variablen `a` ist `&a`. Zu jedem Variablentyp gibt es einen Zeiger- oder Pointertyp: Mit der Deklaration `double *pf` erklärt man eine Zeigervariable `pf` vom Typ "Zeiger auf `double`". Hier hinein kann man Adressen von `double`-Variablen zuweisen: Ist `double z` deklariert, so erhält man durch `pf = &z` die Adresse von `z` in `pf`. Der Dereferenzierungsoperator `*` erlaubt, auf den Wert zuzugreifen, auf den eine Zeigervariable zeigt: `x = *pf` weist der `double`-Variablen `x` den `double`-Wert zu, auf den `pf` zeigt, in diesem Fall also den Wert von `z`.

Ein kurzes Programmbeispiel hierzu:



**Beispiel 3.1**

```
/* Demonstration des Verhaltens von Pointern = Zeigern */

#include <stdio.h>

int main() {
  double x, z;
  double *pf;          /* Zeigervariable vom Typ double */
  z = 3.1415926535;
  x = 2.7182818284;
  printf("sizeof(double) = %u\n", (unsigned int)sizeof(double));
  pf = &x;   /* pf zeigt auf x */
  printf("                        x       z           pf          *pf\n");
  printf("1. pf = &x     :   %6.4lf  %6.4lf  %10lu   %6.4lf\n",
         x,z,(unsigned long)pf,*pf);
  x += 4.2;  /* x inkrementiert */
  printf("2. x += 4.2    :   %6.4lf  %6.4lf  %10lu   %6.4lf\n",
         x,z,(unsigned long)pf,*pf);
  pf = &z;   /* pf zeigt auf z */
  printf("3. pf = &z     :   %6.4lf  %6.4lf  %10lu   %6.4lf\n",
         x,z,(unsigned long)pf,*pf);
  x = *pf;   /* Dereferenzierung von pf und Zuweisung nach x \*/
  printf("4. x  = *pf    :   %6.4lf  %6.4lf  %10lu   %6.4lf\n",
         x,z,(unsigned long)pf,*pf);
}
```

Der Programmlauf ergibt z. B. folgende Ausgabe:

```
sizeof(double) = 8
                          x       z           pf          *pf
1. pf = &x     :   2.7183  3.1416  140734052294200   2.7183
2. x += 4.2    :   6.9183  3.1416  140734052294200   6.9183
3. pf = &z     :   6.9183  3.1416  140734052294208   3.1416
4. x  = *pf    :   3.1416  3.1416  140734052294208   3.1416
```

Folgendes passiert hier:

Zeile 1: `pf` enthält die Adresse von `x`, also hat `*pf` den gleichen Wert wie `x`.
Zeile 2: `x` ist incrememtiert, also haben `x, *pf` den gleichen Wert.
Zeile 3: `pf` enthält nun die Adresse von `z` also haben `z` und `*pf` den gleichen Wert. Außerdem: `x, z` sind in der ersten Zeile von `main` unmittelbar hintereinander definiert, also unterscheiden sich ihre Adressen um `sizeof(double) = 8`
Zeile 4: `x` erhält den Wert von `*pf` zugewiesen, `x, z, *pf` sind nun identisch.

In einem Array– etwa `a[]`– eines gegebenen Typs `typ` haben die Arrayvariablen `a[i]` konsekutive Adressen. Dabei ist die Differenz zweier aufeinanderfolgender Variablen gleich `sizeof(typ)`, also der Anzahl der Bytes, die ein Vertreter dieses Typs im Speicher beansprucht.

Zeigervariable befolgen eine additive Arithmetik: Für die Deklaration `typ *p` bedeutet `p = p + 3` die Erhöhung von `p` um `3 * sizeof(typ)`, so dass also etwa innerhalb eines Arrays vom Typ `typ` um 3 Variable weitergezählt wird. Entsprechendes gilt für `-` ("minus").

Im obigen Beipiel zeigt etwa nach `p = &a[1]; p = p + 3` die Zeigervariable `p` auf `a[4]`, es ist also dann `p == &a[4]`.

**Beispiel 3.2**

```
/* Demonstration der Pointer- oder Zeigerarithmetik */

#include <stdio.h>
#define LEN 6
int main() {
  int i;  char *pc;  double *pd;
  char w[LEN]; double x[LEN];

  printf("Adressen in Arrays, Zeigerarithmetik:\n\n");
  printf("sizeof(char)   = %u, setze pc = w:\n", (unsigned int)sizeof(char));
  pc = w;
  for (i=0; i < LEN-1; i++) {
    printf("&w[%u] = %lu, w+%u = %lu, pc = %lu, pc++\n",
           i,(unsigned long)&w[i],i,(unsigned long)(w+i),(unsigned long)pc);
    pc++;
  }
  printf("sizeof(double) = %u, setze pd = x:\n",(unsigned int)sizeof(double));
  pd = x;
  for (i=0; i < LEN-1; i++) {
    printf("&x[%u] = %lu, x+%u = %lu, pd = %lu, pd++\n",
           i,(unsigned long)&x[i],i,(unsigned long)(x+i),(unsigned long)pd);
    pd++;
  }
  printf("Ein String:\n");
  w[0] = 'H'; w[1] = 'a'; w[2]=w[3]='l'; w[4]='o'; w[5]='\0';
  printf("%s", w);
  printf("\n");
  printf("...%s...\n",w);
  pc = w;
  for (i=0; i < LEN; i++)
    printf("%c ",*pc++);
  printf("\n");
}
```

Der Programmlauf ergibt:

```
Adressen in Arrays, Zeigerarithmetik:

sizeof(char)   = 1, setze pc = w:
&w[0] = 140735309176080, w+0 = 140735309176080, pc = 140735309176080, pc++
&w[1] = 140735309176081, w+1 = 140735309176081, pc = 140735309176081, pc++
&w[2] = 140735309176082, w+2 = 140735309176082, pc = 140735309176082, pc++
&w[3] = 140735309176083, w+3 = 140735309176083, pc = 140735309176083, pc++
&w[4] = 140735309176084, w+4 = 140735309176084, pc = 140735309176084, pc++
sizeof(double) = 8, setze pd = x:
&x[0] = 140735309176000, x+0 = 140735309176000, pd = 140735309176000, pd++
&x[1] = 140735309176008, x+1 = 140735309176008, pd = 140735309176008, pd++
&x[2] = 140735309176016, x+2 = 140735309176016, pd = 140735309176016, pd++
&x[3] = 140735309176024, x+3 = 140735309176024, pd = 140735309176024, pd++
&x[4] = 140735309176032, x+4 = 140735309176032, pd = 140735309176032, pd++
Ein String:
Hallo
...Hallo...
H a l l o
```



*Aufgabe 3.3: Interpretieren Sie die Ausgabe des letzten Programms in allen Einzelheiten im Sinne der Zeigerarithmetik.*

Zeiger gleichen Typs können subtrahiert werden, für `typ *p, *q` ist `p - q` die Ganzzahl `n`, für die im Sinne obiger Zeigerarithmetik gilt: `p = q + n`. Die Zahl `n` kann negativ sein.

*Aufgabe 3.4: Studieren Sie die Zeigerarithmetik, insbesondere den soeben beschriebenen Sachverhalt, anhand eigener Beispielprogramme; lassen Sie z. B. für zwei Zeiger gleichen Typs die Differenz ausgeben. Was geschieht, wenn Sie versuchen, zwei Zeiger verschiedenen Typs zu subtrahieren?*

**Beispiel 3.3: Kommandozeilenargumente, Dateienverwaltung**

Ein Programm kann Kommandozeilenargumente verarbeiten. Ist der Name des ausführbaren Programms etwa `progname`, so kann bei passender Vorbereitung mit dem Aufruf

`progname string1 string2 string3 ...`

vom Programm `progname` aus auf ein Array von Strings `*argv[]` mit den Werten

`argv[0] = progname, argv[1] = string1, argv[2] = string2, ...`

zugegriffen werden.

```
/* Komandozeilen-Argumente: */
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  printf("argc = %d\n", argc);
  for (i=0; i < argc; i++)
    printf("argv[%d] = %s\n", i, argv[i] );
}
```

*Aufgabe 3.5: Angenommen, die ausführbare Version dieses Programms heiße* `a.out`. *Was ergibt dann der Aufruf*

`./a.out dies ist nur ein test`

*für eine Ausgabe? Interpretieren Sie die gesamte Ausgabe!*

Da die Anzahl der Argumente in der Kommandozeile nicht a priori feststeht, benötigt man die Variable `argc`, in der nach Aufruf die aktuelle Argumentezahl enthalten ist.

Dies kann man zum Beispiel verwenden, um ein `copy`-Programm zu schreiben, das mit dem Befehlsaufruf `copy quelle ziel` die Datei `quelle` auf die Datei `ziel` kopiert, ohne die Umleitungstechnik des Betriebssystems mit den Zeichen `>,<` zu verwenden. Dazu muss man Dateien öffnen, schließen, lesen und schreiben können. Der hierbei verwendete Datentyp ist ein "File pointer" `FILE *`, der von der IO-Bibliothek standardmäßig zur Verfügung gestellt wird, ebenso wie die Funktionen

`FILE *fopen(char *name, char *mode), int fclose(FILE *fp),`
`int getc(FILE *fp), int putc(int c, FILE *fp)`

und andere.

**Beispiel 3.4:**

```
/* Kopierprogramm mit Komandozeilen-Argumenten */
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[] ) {
  int c; FILE *quelle, *ziel;
  if (argc != 3) {
    printf("Aufruf: programmname quelldatei zieldatei\n");
    exit(1);
  }
  quelle = fopen(argv[1], "r");
  ziel   = fopen(argv[2], "w");
  while ( (c = getc(quelle)) != EOF ) putc(c, ziel);
  fclose(quelle);
  fclose(ziel);
}
```

Das folgende Programm mit praktischem Nutzen wendet verschiedene Aspekte von Arrays und von Zeigerarithmetik an und illustriert Ein- und Ausgabe von einzelnen Zeichen.

```
/* titel: ascii.c; gibt Ascii-Zeichen aus. Eingabe oktal, dezimal, hex */

/* ASCII := American Standard Code for Information Interchange   */

#include <stdio.h>
#include <stdlib.h>  /* wegen exit             */
#include <ctype.h>   /* wegen iscntrl, isdigit */
#include <string.h>  /* wegen strcmp, strlen   */

/* Definition eines Arrays von Strings: jedes Array-Element ist ein
   String und als solcher ebenfalls ein Array, allerdings von char.
   Die Strings selbst werden daher als Pointer auf char deklariert und
   durch ihre explizite Angabe initialisiert.
*/
char *name[] = { "NUL","SOH","STX","ETX","EOT","ENQ","ACK","BEL",
                  "BS", "HT", "LF", "VT", "FF", "CR", "SO", "SI",
                 "DLE","DC1","DC2","DC3","DC4","NAK","SYN","ETB",
                 "CAN", "EM","SUB","ESC", "FS", "GS", "RS", "US",
                 "SP"
               };

char *eingabe = "\
  Eingabe eines Einzelzeichens liefert dessen Asciiwert \n\
  Eingabe von ^Zeichen liefert Asciiwert des Control-Zeichens\n\
  Eingabe einer mehrstelligen Ziffer liefert zugehoeriges Asciizeichen\n\
  Eingabe einer Ziffer mit vorangestelltem\n\
                o, O oder 0:      Ziffer wird oktal gelesen\n\
                d oder D:         Ziffer wird dezimal gelesen\n\
                x oder X:         Ziffer wird hexadezimal gelesen\n\
                tab:      Eine Ascii-Tabelle wird ausgegeben\n\
                TAB:      Eine Tabelle der Zeichen > 127 wird ausgegeben\n\
  Abbruch durch Eingabe von \"ex\" oder mit Ctrl-C\n";

/* Die folgende Funktion verwendet Zeigerarithmetik fuer Arrays
 s ist eine Zeigervariable fuer char, die anfangs auf den Beginn eines
 char-Arrays zeigt, der fuer die Speicherung einer Zeile verwendet werden soll.
 Nach jedem empfangenen Zeiche wird s inkrementiert, also erreicht, dass s
 auf eine neue Position zeigt, um das naechste Zeichen aufnehmen zu koennen.
*/

void mygetline(char *s) { /* Prueft keinen Speicher-Ueberlauf!!! */
  int c, cc = 0;
  while ( (c=getchar()) != '\n' && c != EOF) {
    *s++ = c; cc++;
  }
  if ( cc == 0 && c == '\n')  /* newline nur eintragen, wenn es einziges
   Zeichen der Zeile ist, damit es als Ascii-Zeichen erkannt werden kann.*/
    *s++ = c;
  *s = '\0';
}
```



```
int main() {
  char s[10]; int c; void tabelle(int);
  printf( "%s", eingabe );
  while (1) {
    printf(": ");
    mygetline(s);      /* scanf("%s",s)  geht nicht wegen \n */
    if ( strcmp(s, "ex" ) == 0 )   /* Abbruch */
      exit(0);
    if ( strcmp(s, "tab") == 0 ) {
      tabelle(0); continue;
    }
    if ( strcmp(s, "TAB") == 0) {
      tabelle(1); continue;
    }
    if ( strlen(s) == 1 ) c = s[0];
    else if ( s[0] == '^' ) c = s[1] & 0x1f ; /* bitweise &: nur die letzten 5 bit */
    else if ( s[0] == 'o' || s[0] == 'O' || s[0] == '0')
      sscanf(&s[1],"%o",&c); /* sscanf liest aus dem bei &s[1] beginnenden String */
    else if ( s[0] == 'd' || s[0] == 'D')
      sscanf(&s[1],"%d",&c);
    else if     ( isdigit( s[0] ) )
      sscanf(s, "%d", &c);
    else if ( s[0] == 'x' || s[0] == 'X' )
      sscanf(&s[1],"%x",&c);
    c &= 0xff;
    if ( !iscntrl(c) && c != ' ')
      printf("ASCII(%c) = o%o  d%d  x%x\n",c,c,c,c);
    else if ( c == ' ' )
      printf("ASCII( ) = %s = o%o  d%d  x%x\n",name[c],c,c,c);
    else if ( c != 0x7f )
      printf("ASCII(^%c) = %s = o%o  d%d  x%x\n",  c+0x40,name[c],c,c,c);
    else
      printf("ASCII(DEL) = o%o  d%d  x%x\n",c,c,c);
  }
}
void tabelle(int x) {
    int i;
    printf("\n");
    if (x==0) {
      for (i=0; i < 31; i++)
        printf("\t%3d %2x %s\t%3d %2x %c\t%3d %2x %c\t%3d %2x %c\n",\
               i,i,name[i],i+32,i+32,i+32,i+64,i+64,i+64,i+96,i+96,i+96);
      printf("\t%3d %2x %s\t%3d %2x %c\t%3d %2x %c\t%3d %2x DEL\n",\
             i,i,name[i],i+32,i+32,i+32,i+64,i+64,i+64,i+96,i+96);
    }
    else
      for (i=128; i < 128 + 32; i++)
        printf("\t%3d %2x ??? \t%3d %2x %c\t%3d %2x %c\t%3d %2x %c\n",\
               i,i,  i+32,i+32,i+32,i+64,i+64,i+64,i+96,i+96,i+96);
    printf("\n");
}
```