

Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 4 (4 Seiten)

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";
```

ist Abkürzung für

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

Es wird bei der Notation mit `"..."` stets ein abschließendes `'\0'`-Zeichen eingebettet. Daher liefert `sizeof(wort)` den Wert 7, während `strlen(wort)` 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", "Sonntagabend" };
```

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 4.1: Schreiben Sie ein Programm, das auf die Eingabe eines Datums, etwa "30.3.2000", antwortet mit: "30. März 2000" oder analog. (Siehe Aufgabe 2.5.) Sorgen Sie dafür, dass unzulässige Daten wie 32.5.2000, 29.2.1900 usw. vom Programm abgelehnt werden. Für Mutige: Erweitern Sie das Programm, so dass die Ausgabe "Donnerstag, 30. März 2000" 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 nummeriert, 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 4.1

```
#include <stdio.h>

main() {
    double x, z;
    double *pf;          /* Zeigervariable vom Typ double */
    z = 3.1415926535;
    x = 2.7182818284;
    printf("sizeof(double) = %d\n\n", sizeof(double));

    pf = &x; /* pf zeigt auf x */
    printf("      x      z      pf      *pf\n");
    printf("1. pf = &x      :   %6.4lf %6.4lf %10u   %6.4lf\n", x,z,pf,*pf);
    x += 4.2; /* z inkrementiert */
    printf("2. x += 4.2    :   %6.4lf %6.4lf %10u   %6.4lf\n", x,z,pf,*pf);
    pf = &z; /* pf zeigt auf z */
    printf("3. pf = &z      :   %6.4lf %6.4lf %10u   %6.4lf\n", x,z,pf,*pf);
    x = *pf; /* Dereferenzierung von pf und Zuweisung nach x */
    printf("4. x = *pf      :   %6.4lf %6.4lf %10u   %6.4lf\n", x,z,pf,*pf);
}
```

Der Programmlauf ergibt z. B. folgende Ausgabe:

```
sizeof(double) = 8

      x      z      pf      *pf
1. pf = &x      :   2.7183  3.1416 3221223128  2.7183
2. x += 4.2     :   6.9183  3.1416 3221223128  6.9183
3. pf = &z      :   6.9183  3.1416 3221223120  3.1416
4. x = *pf      :   3.1416  3.1416 3221223120  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 inkrementiert, 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 Beispiel zeigt etwa nach `p = &a[1]`; `p = p + 3` die Zeigervariable `p` auf `a[4]`, es ist also dann `p == &a[4]`.

Beispiel 4.2

```

#include <stdio.h>
#define LEN 5
main() {
    int i; char *pc; double *pd;
    char w[LEN]; double x[LEN];

    printf("Adressen in Arrays, Zeigerarithmetik:\n");
    printf("sizeof(char)   = %u, setze pc = s:\n", sizeof(char) );
    pc = w;
    for (i=0; i < LEN; i++) {
        printf("&w[%u] = %u, s+%u = %u, pc = %u, pc++\n", i,&w[i],i,w+i,pc);
        pc++;
    }
    pd = x;
    printf("sizeof(double) = %u, setze pd = x:\n", sizeof(double) );
    pd = x;
    for (i=0; i < LEN; i++) {
        printf("&x[%u] = %u, x+%u = %u, pd = %u, pd++\n", i,&x[i],i,x+i,pd);
        pd++;
    }
    printf("Ein String:\n");
    w[0] = 'H'; w[1] = 'a'; w[2]=w[3]='l'; w[4]='o'; w[5]='\0';
    printf(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 = s:
&w[0] = 3221223128, s+0 = 3221223128, pc = 3221223128, pc++
&w[1] = 3221223129, s+1 = 3221223129, pc = 3221223129, pc++
&w[2] = 3221223130, s+2 = 3221223130, pc = 3221223130, pc++
&w[3] = 3221223131, s+3 = 3221223131, pc = 3221223131, pc++
&w[4] = 3221223132, s+4 = 3221223132, pc = 3221223132, pc++
sizeof(double) = 8, setze pd = x:
&x[0] = 3221223088, x+0 = 3221223088, pd = 3221223088, pd++
&x[1] = 3221223096, x+1 = 3221223096, pd = 3221223096, pd++
&x[2] = 3221223104, x+2 = 3221223104, pd = 3221223104, pd++
&x[3] = 3221223112, x+3 = 3221223112, pd = 3221223112, pd++
&x[4] = 3221223120, x+4 = 3221223120, pd = 3221223120, pd++
Ein String:
Hallo
...Hallo...
H a l l o

```

Aufgabe 4.2: 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 4.3: 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 4.3: Kommandozeilenargumente, Dateienverwaltung

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

```
programe string1 string2 string3 ...
```

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

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

zugegriffen werden. (Bemerkung: Unter MS-DOS und kompatiblen Betriebssystemen ist bei den älteren Versionen der Zugriff auf *argv[0]* nicht möglich.)

```
#include <stdio.h>
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 4.4: 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 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 4.4:

```
/* titel: copy.c, kopiert Dateien */
#include <stdio.h>
main( int argc, char *argv[] ) {
    int c; FILE *quelle, *ziel, *fopen();
    quelle = fopen(argv[1], "r");
    ziel    = fopen(argv[2], "w");
    while ( (c = getc(quelle)) != EOF )
        putc(c, ziel);
    fclose(quelle);
    fclose(ziel);
}
```