

Einführung in die Programmiersprachen C und C++

Markus Kirschmer, Fakultät für Mathematik

Übungsblatt 3

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 erstenen 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 ist ein komplettes 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 globale i wird hier durch das lokale 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 globalen 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 der Funktion `test()`

1. eine Deklaration `int i;`
2. eine Definition `int i = 10;`

stehen würde?

Erst überlegen, dann ausprobieren und anschließend das Ergebnis erklären!

Arrays, 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 eine Abkürzung für

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

Auch bei der ersten Variante wird ein abschließendes `\0`-Zeichen angefügt. 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 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 das `'\0'`-Zeichen erhält.

Z.B. wird durch

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

der siebte Buchstabe des vierten Tagesnamens ausgedruckt, also ein `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 das 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.

Wer will erweitert 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 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 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 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.

Zeigervariablen erlauben eine 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 Variablen weitergezählt wird. Entsprechendes gilt für `p-3`.

Im obigen Beispiel zeigt etwa nach `p = &a[1]`; `p = p + 3` die Zeigervariable `p` auf `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);
    }
    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");
}

```

Aufgabe 3.3. Interpretieren Sie die folgende Ausgabe des obigen Programms im Sinne der Zeigerarithmetik:

```

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

```

Zeiger gleichen Typs können subtrahiert werden für `typ *p,*q` ist `p - q` die ganze Zahl `n` mit `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.

Da die Anzahl der Argumente in der Kommandozeile nicht a priori feststeht, benötigt man die Variable `argc`, in der nach Aufruf die aktuelle Argumentanzahl enthalten ist. Hier ist ein Beispiel:

```

/* Kommandozeilen-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 `./a.out` dies ist nur ein test für eine Ausgabe? Interpretieren Sie die gesamte Ausgabe!

Aufgabe 3.6. Ändern Sie die `main` Funktion des obigen Beispiels ab in

```
int main(int argc, char *argv[], char *env[])
```

Die Variable `env` ist ein NULL-terminiertes Array von Strings, das die Umgebungsvariablen des Programms enthält. Geben Sie diese nacheinander aus.

Das folgende Programm `copy` kopiert Dateien. Der Aufruf `copy quelle ziel` kopiert die Datei `quelle` auf die Datei `ziel`, ohne die Umleitungstechnik des Betriebssystems mit `>`, `<` zu verwenden. Dazu muss man Dateien öffnen, schließen, lesen und schreiben. Der hierbei verwendete Datentyp ist ein „File pointer“ `FILE *`, der von der Standard-IO Bibliothek 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 Kommandozeilen-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");

```

