ModProg 15-16, Vorl. 7

Richard Grzibovski

Dec. 2, 2015

Übersicht

Übersicht

1. Variablen (Teil 4)
   - `const`
   - `volatile`
2. Zeiger
   - Motivation: Beispiele
   - Definition
   - Operationen

Variablen: const

`const Typ Name`

- Die Variable Name ist "schreib-geschützts", d.h. ihr darf nach der Initialisierung nichts mehr zugewiesen werden
- Daher i. Allg. Initialisierung direkt bei der Definition mit `const Typ Name = Wert;`
- Ergebnis bei Änderungsversuchen ist **compilerabhängig**

Beispiel:

```
double const PI = 4 * atan(1);
const int a = 37;
int const b = 25;
a = 5;    /* Fehler! */
b++;      /* Fehler! */
```

Variablen: volatile

`volatile Typ Name`

- Die Variable Name kann sich jederzeit ändern, insbesondere auch **ohne Zutun des Programms**.
- Hinweis an den Compiler, dass er keine Annahme über ihren Wert machen kann und dass damit bestimmte Optimierungen unmöglich werden.
- Einsatz bei Speicherbereichen, deren Wert von außen geändert wird, z.B.
  - System-Uhr, Messgeräte-Port, . . .

Beispiel:

```
volatile unsigned int systemUhr ;
volatile unsigned long geraeteStatus ;
```

Variablen: volatile

Mit der Schpeicherklasse `volatile` (engl. für flüchtig) bei der Variablendefinition wird dem C-Compiler mitgeteilt, dass der Wert dieser Variable auch von außerhalb des Programms – also von anderen Programmen bzw. von der Hardware – geändert werden kann. Solche Variablen werden bei der Optimierung des Quellcodes vom Compiler nicht berücksichtigt.

Beispiel:

Quellcode

```
int main() {
  long i;
  double a;
  for(i=0;i<1e6;i++) {
      a = 1.2;
  }
  printf("i= %ld \
    a= %le\n",i,a);
  return 0;
}
```

Compiler Optimierte Quellcode

```
int main() {
  long i;
  double a;

  a = 1.2;
  i = 1000000;
  printf("i= %ld \
    a= %le\n",i,a);
  return 0;
}
```

Zeiger: Motivation

```
1  #include<stdio.h>
2  void tausche(int a, int b) {
3       int hilf ;
4       hilf = a; a = b; b = hilf ;
5  }
6
7  int main(void) {
8       int x= 1 , y= 2;
9       printf(" x= %i , y= %i\n ", x, y);
10      tausche(x, y);
11      printf(" x= %i , y= %i\n ", x, y);
12      return 0;
13 }
```

Ergibt:

```
x= 1, y= 2
x= 1, y= 2
```

Zeiger: Motivation

Wichtig!

Funktionsaufrufe werden in C als "Call by Value" realisiert. Dabei nimmt die Funktion lediglich Kopien der übergebenen Variableninhalte entgegen.

Alle Manipulationen im Funktionsrumpf werden also an Duplikaten durchgeführt – die Inhalte der Variablen aus der aufrufenden Funktion bleiben davon unberührt.

Erklärung

Funktionsparameter sind lokale automatische Variablen. Sie werden beim Aufruf der Funktion mit den Inhalten der übergebenen Variablen initialisiert und existieren nach dem Verlassen der Funktion nicht mehr.

Zeiger: Motivation

```
1  #include<stdio.h>
2  void tausche(int a, int b) {
3       int hilf ;
4       printf("a= %i , b= %i\n ", a, b);
5       hilf = a; a = b; b = hilf ;
6       printf("a= %i , b= %i\n ", a, b);
7  }
8  int main(void) {
9       int x= 1 , y= 2;
10      printf("x= %i , y= %i\n ", x, y);
11      tausche(x, y);
12      printf("x= %i , y= %i\n ", x, y);
13      return 0;
14 }
```

Ergibt:

```
x= 1, y= 2
a= 1, b= 2
a= 2, b= 1
x= 1, y= 2
```

Zeiger: Beispiel mit Feldern

```
1  #include<stdio.h>
2  void transpos(int feld[], int idx1, int idx2) {
3      int hilf ;
4      hilf       = feld[idx1];
5      feld[idx1] = feld[idx2];
6      feld[idx2] = hilf;
7  }
8  int main(void) {
9    int fld[3]={1,2,3};
10   printf("fld[0]= %i , fld[2]= %i\n ", fld[0], fld[2]);
11   transpos(fld, 0, 2);
12   printf("fld[0]= %i , fld[2]= %i\n ", fld[0], fld[2]);
13   return 0;
14 }
```

Ergibt:

```
fld[0]= 1 , fld[2]= 3
fld[0]= 3 , fld[2]= 1
```

Die Funktion verändert die Feldeinträge tatsächlich!

Offene Fragen

- Wie programmiert man eine Funktion, die zwei Gleitpunktzahle tauscht?
- Warum funktioniert die Funktion `scanf`?

```
double a=0.0;
scanf("%lf",&a);
printf("%lf\n",a);
```

- Felder
  - Was passiert mit Feldern, wenn man übergibt die als Funktionsargumente?
  - Wie ändert man die Feldlänge im Verlauf des Programms?

Offene Fragen

Nicht funktionierendes dynamisches Feld

```
1  #include<stdio.h>
2  int main() {
3      int i;
4      double fld[i]; // uninitialisierter Wert von i
5      scanf("%i",&i);
6      fld[i-1]=1.0;
7      printf("%le\n",fld[i-1]);
8      return 0;
9  }
```

Ergibt:

`Segmentation fault.` oder

`Speicherzugriffsfehler.`

Zeiger (Pointer)

Zu jeder Variablen gehören ihr

Wert: Eine Bitfolge im Speicher

Typ: Wie lang ist diese Bitfolge, und wie ist sie zu interpretieren ?

Name: Ermöglicht den Zugriff auf die Bitfolge: Wo im Speicher ist sie abgelegt?

→ *Name entspricht einer Adresse im Speicher*

Zeigervariablen

Zeigervariablen sind Variablen, deren Inhalt die Adresse eines Speicherbereiches (einer Variablen) ist.

Anwendung: Statt den Variableninhalt über den Variablennamen anzusprechen, greift man auf den Wert zu, der unter der betreffenden Adresse abgespeichert ist.

Deklaration

Damit die unter den Adressen hinterlegten Werte auch korrekt interpretiert werden, muss man bei der Deklaration auch den betreffenden Datentyp angeben.

```
Datentyp *Zeigername1, ... , *ZeigernameN;
```

Eine Zeigervariable ist bei der Deklaration also durch ein vorangestelltes `*` gekennzeichnet.

Man beachte, dass bei Listen von Zeigervariablen jedem Bezeichner ein `*` vorangestellt werden muss!

Beispiel: Mit

```
float *fzeiger, fzahl;
```

wird nämlich lediglich `fzeiger` als Zeigervariable deklariert, `fzahl` dagegen ist eine übliche Variable vom Typ `float`.

Sprachlich drückt man den Sachverhalt so aus:

"`fzeiger` ist ein Zeiger auf `float`."

Referenzieren und Dereferenzieren

Der Adressoperator

In C kann man die Adresse einer Variablen ermitteln, in dem man den **unären Adressoperator** `&` auf die Variable anwendet, so ist z.B. die Adresse der Variablen `fzahl` gegeben durch den Ausdruck `&fzahl`.

Ausdrücke dieser Art sind uns von Aufrufen der `scanf`-Funktion bereits bekannt.

Referenzierung

Die so gewonnene Adresse kann man nun einer Zeigervariablen vom gleichen Typ zuweisen, in unserem Beispiel: `fzeiger = &fzahl;` Hierdurch wird die Adresse der `float`-Variablen `fzahl` in der Zeigervariablen `fzeiger` gespeichert.

Man sagt:
"`fzeiger` zeigt auf (den Inhalt von) `fzahl`."
"`fzeiger` referenziert den Inhalt von `fzahl`."

Referenzieren und Dereferenzieren

Dereferenzierung

Wie die eckigen Klammern bei den Feldern hat auch * eine Doppelfunktion:

- Bei der Deklaration werden Zeigervariablen durch * gekennzeichnet.
- Mit dem **unären Inhaltsoperator** * greift man auf den Inhalt zu, der durch die Zeigervariable referenziert wird. Dieser Zugriff ist sowohl lesend als auch schreibend.

Beispiel:

```
1  #include<stdio.h>
2  int main() {
3    int a=42, b, *p;
4    p = &a;
5    b = *p;
6    printf("%i\n",a==b); // druckt 1 (wahr)
7    return 0;
8  }
```

Zusammenfassung

Ein Zeiger (engl. pointer) markiert die Position eines Datenobjekts im Speicher oder alternativ ausgedrückt:
Eine Zeigervariable speichert die Adresse des betreffenden Datenobjekts und gibt an, wie die unter der Adresse abgelegten Bytes zu interpretieren sind.

Typ-Bindung von Pointern

Bei

```
int *pi;
double *pd;
```

gilt:

- Der Typ von `pi` ist: "Zeiger auf eine Variable vom Typ `int`"
- Der Typ von `pd` ist: "Zeiger auf eine Variable vom Typ `double`"

Deshalb wäre die Zuweisung

```
pd = pi;
```

nicht zulässig (verschiedene Typen !).

Typ-Bindung von Pointern

(Initialisierte) Zeiger zeigen immer auf eine Größe eines bestimmten Typs.

Am Typ eines Pointers `p` kann der Compiler erkennen

- wie großder Speicherbereich ist, auf den `p` zeigt und
- wie die dort abgelegte Bitfolge zu interpretieren ist.

Call by Reference - Konzept

Konzept

Beim Funktionsaufruf *Call by Reference* werden Zeiger auf die zu verändern den Datenobjekte als Argumente übergeben.

tausche-Funktion

```
1  #include<stdio.h>
2  void tausche(int *a, int *b) {
3       int hilf ;
4       hilf = *a; *a = *b; *b = hilf ;
5  }
6  int main(void) {
7       int x= 1, y= 2;
8       printf("x= %i , y= %i\n ", x, y);
9       tausche(&x, &y);
10      printf("x= %i , y= %i\n ", x, y);
11      return 0;
12 }
```

Ergibt:

```
x= 1, y= 2
x= 2, y= 1
```

Zeiger und Felder: ein Experiment

```
1  #include<stdio.h>
2  int main(void)
3  {
4      double feld[] = {1.0, 2.0, 3.0};
5      printf("feld \t\t\t = %p\n ",           feld    );
6      printf("Adresse von feld[0]\t = %p\n", &feld[0]);
7      printf("feld[0]\t\t = %.1f\n",          feld[0]);
8      printf("Inhalt von feld\t = %.1f\n" ,  *feld    );
9      return 0;
10 }
```

Ergibt:

```
feld                  = 0x7fff1a5803e0
Adresse von feld[0]   = 0x7fff1a5803e0
feld[0]               = 1.0
Inhalt von feld       = 1.0
```

Zeiger und Felder: die Regel

Es gilt die folgende wichtige Regel in C:

Wichtig

Der Name eines Feldes ist zugleich ein Zeiger auf das erste Feldelement.

Zeigerarithmetik

Wir wissen aus den vorangegangenen Betrachtungen:

- Adressen sind im prinzipiell ganzzahlige Größen.
- Feldnamen sind Zeiger auf das erste Feldelement.

Die Fragen

- Im Hinblick auf den ersten Punkt kann man sich fragen, ob man an den Inhalten von Zeigervariablen arithmetische Operationen durchführen kann.
- Hinsichtlich des zweiten Punkts kann man sich ferner fragen, ob man solche Operationen in Verbindung mit dem Inhaltsoperator dazu verwenden kann, auf Feldkomponenten zuzugreifen.

Zeiger und Felder: ein Experiment

```
1  #include<stdio.h>
2  int main(void) {
3        float f[]={-1.0 f, 0.0f, 1.0f};
4        float *zgr ;
5        printf ("       f[2] = %f\n",  f[2]);
6        printf ("    *(f+2) = %f\n",*(f+2));
7        zgr = f ;
8        printf("        *zgr = %f\n",   *zgr    );
9        printf("   *(zgr++) = %f\n",   *(zgr++));
10       printf("        *zgr = %f\n",   *zgr    );
11       printf("   *(++zgr) = %f\n",*(++zgr)   );
12       return 0;
13 }
```

Ergibt:

```
      f[2] = 1.000000
   *(f+2) = 1.000000
     *zgr = -1.000000
*(zgr++) = -1.000000
     *zgr = 0.000000
*(++zgr) = 1.000000
```

Zeigerarithmetik

Wir fassen zusammen:

- Mit Adressen / Zeigern kann man arithmetische Operationen ausführen, mit deren Hilfe man z.B. auf Feldeinträge zugreifen kann.
- Anhand des Datentyps und der damit bekannten Größe der Datenobjekte bestimmt der Compiler, wie die Additionen, In- und Dekrementierungen im Speicher umzusetzen sind ("wie weit er jeweils springen muss").
- Bei der Kombination von Zeigerarithmetik und Inhaltsoperator ist auf die korrekte Klammerung zu achten, da sonst semantisch bzw. syntaktisch problematische Ausdrücke entstehen können. Z.B. ist bei `int`-Variablen der Ausdruck `*zgr++` von `*(zgr++)` zu unterscheiden: Ersteres bedeutet, dass der Inhalt inkrementiert wird, letzteres, dass erst der Inhalt ausgelesen und danach der Zeiger inkrementiert wird.

Unveränderbare Zeiger

Ein Zeiger kann unveränderbar sein (d.h. er kann auf keine andere Adresse zeigen) oder auf eine unveränderbare Variable zeigen – oder beides; je nachdem, an welcher Stelle das Schlüsselwort `const` verwendet wird.

Unveränderbare Zeiger

Beispiel:

```
int          i1 = 5;
int const i2 = 3;
```

veränderbarer Zeiger auf veränderbare Variable:

```
int          *          ip1 = &i1;
```

veränderbarer Zeiger auf unveränderbare Variable:

```
int const *             ip2 = &i2;
```

unveränderbarer Zeiger auf veränderbare Variable:

```
int          * const ip3 = &i1;
```

unveränderbarer Zeiger auf unveränderbare Variable:

```
int const * const ip4 = &i2;
```

`*(ip1++); (*ip1)++; *(ip2++);` – erlaubt!

`(*ip2)++;` – Fehler - unveränderbare Variable!

`*(ip3++);` – Fehler - unveränderbarer Zeiger!

`(*ip3)++;` – erlaubt!

`*(ip4++);` – Fehler - unveränderbarer Zeiger!

`(*ip4)++;` – Fehler - unveränderbare Variable!

Die besondere Zeigerwert NULL

- NULL ist in stdlib.h vordefinierte Zeigerwert.
- Eine Zeigervariable mit Wert NULL (heißt NULL-Zeiger) referenziert sozusagen einen nicht existierenden Speicherbereich.
- Versuche ins diese Bereich zu Schreiben oder von diese Bereich zu lesen führen zu der Speicherzugriffsfehler.

Beispiel:

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main(void) {
4      int *p; /* p ist eine  zufaelige Adresse */
5      p=NULL; /* p ist eine ungueltige Adresse */
6      if(p==NULL) // <-- wahr
7          printf("p ist NULL\n");
8      printf("%d\n",*p); //<-- Speicherzugriffsfehler
9      return 0 ;
10 }
```

Zeiger auf Zeiger

Beispiel:

```
1  #include <stdio.h>
2  int main(void) {
3      int a = 3, b = 5 ;
4      int *p = &a ;
5      int **pp;
6      /* pp ist ein Zeiger auf einen Zeiger, der auf eine
7         int-Variable zeigen muss; pp enthaelt noch keine
8         gueltige Adresse */
9      pp = &p ; /* pp zeigt jetzt auf den int-Zeiger p */
10     printf("%d\n", **pp); // druckt 3
11      *pp = &b; // bedeutet, dass p = &b, deswegen
12     printf("%d\n",   *p); // druckt 5
13     **pp = 17; // bedeutet, dass b = 17, deswegen
14     printf("%d\n",    b); // druckt 17
15     return 0 ;
16 }
```

Ein Feld von Zeigern

Beispiel:

```
1  #include <stdio.h>
2  int main(void) {
3      char satz[]="123 Hallo, Welt!";
4      char *p[2]; // p ist ein Feld mit 2 Zeigern
5      p[0] = satz +  4; // *p[0] == 'H'
6      p[1] = satz + 10; // *p[1] == 'W'
7      printf("%s\n", satz); // druckt "123 Hallo, Welt!"
8      printf("%s\n", p[0]); // druckt "Hallo, Welt!"
9      printf("%s\n", p[1]); // druckt "Welt!"
10     return 0 ;
11 }
```