Modellierung und Programmierung

Dr. Martin Riplinger

28.11.2012

UNIVERSITÄT DES SAARLANDES

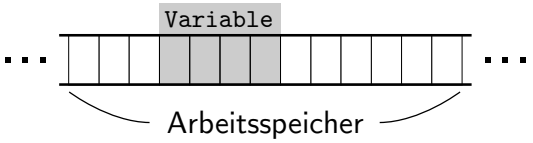
IAM Institut für Angewandte Mathematik

Zeiger: Definition und Prinzip

Definition

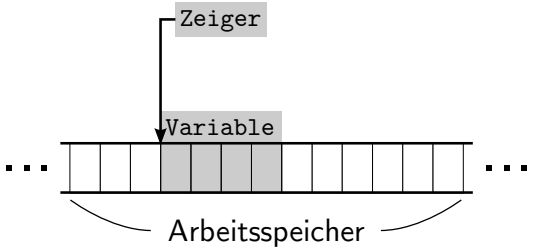
Ein Zeiger (engl. *pointer*) ist eine Variable, deren Inhalt eine Speicheradresse und der zugehörige Datentyp ist.

Funktionsprinzip:



Zugriff über Variablennamen

- Variablenname steht stellvertretend für einen Speicherbereich
- Verknüpfung ist mit Deklaration festgelegt und unveränderbar
- Bei Zugriff wird die Speicher-Referenz **intern** aufgelöst und der Wert anschließend gelesen bzw. geschrieben.



Zugriff über Zeiger

- Speicherbereich ist **Inhalt** der Zeigervariablen
- Dieser Inhalt ist selbstverständlich veränderbar → Zeiger kann „umgebogen“ werden
- Auflösung der Speicherreferenz geschieht **explizit**, d. h. durch den Programmierer

Zeiger

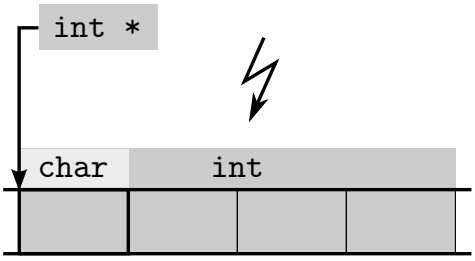
Deklaration: `Datentyp *Zeigername;` bzw. `Datentyp *z1,..., *zN;`

Achtung: vor jedem Zeigername muss ein „*“ stehen!

Beispiel: `int a, *p;` erzeugt eine `int`-Variable und einen **Zeiger auf** `int`.

Bedeutung des Datentyps: Wird über einen Zeiger auf eine Speicheradresse zugegriffen, so gibt der Datentyp an, wie viele Byte *von der (Start-)Adresse an* gelesen bzw. geschrieben werden sollen.
Achtung: Fehlinterpretationen, d. h. die Verwendung eines Zeigers auf einen anderen Datentyp als vorgesehen, kann zu ungewollten Ergebnissen führen.

Beispiel: Zeiger auf `int` statt Zeiger auf `char`.



- Eine `char`-Variable wurde deklariert
- Fälschlicherweise wurde ein *Zeiger auf* `int` verwendet, um über die Speicheradresse auf die `char`-Variable zugreifen zu können.
- Bei einem Zugriff werden statt 1 Byte 4 Byte angesprochen, von denen 3 nicht mehr zum „zulässigen“ Bereich gehören.
- Vor solchen Fällen **warnt** der Compiler bestenfalls!

Arbeiten mit Zeigern: Referenzen

Adress- oder Referenzoperator: `&`

- kann auf jedes beliebige Speicherobjekt (Variablen, Funktionen,...) angewandt werden.
- hat Vorrang vor Vergleichs- und arithmetischen Operatoren, nicht jedoch vor dem „Array/Index-Operator“ `[]`.
- liefert als Ergebnis einen **konstanten Zeiger auf die Adresse** (=Referenz) des Objekts.

Beispiel:

```
int i = 1;
double x = M_E,
       y[] = {3.0, 0.0, 42.0};

printf("Adresse von i: %p\n", &i);
printf("Wert von i: %d\n\n", i);

printf("Adresse von x: %p\n", &x);
printf("Wert von x: %lf\n\n", x);

printf("Adresse von y[2]: %p\n",
      &y[2]); // oder &(y[2])
printf("Wert von y[2]: %lf", y[2]);
```

Ausgabe:

```
Adresse von i: 0x7fffb476d89c
Wert von i: 1

Adresse von x: 0x7fff45786740
Wert von x: 2.718282

Adresse von y[2]: 0x7fff45786730
Wert von y[2]: 42.000000
```

Bemerkung: %p ist der Platzhalter für Adressen im Formatstring.

Arbeiten mit Zeigern: Referenzen

Inhalts- oder Dereferenzierungsoperator: *

- lässt sich auf Zeiger anwenden.
- steht auf der gleichen Prioritätsstufe wie der Adressoperator &.
- liefert den **Wert**, der an der Adresse gespeichert ist, auf die der Zeiger verweist.

Beispiel:

```
int i = 1;
double x = M_E, *px = &x,
       y[] = {3.0, 0.0, 42.0};

printf("Adresse von i: %p\n", &i);
printf("Wert von i: %d\n\n", *(&i));

printf("Adresse von x: %p\n", px);
printf("Wert von x: %lf\n\n", *px);

printf("Adresse von y: %p\n", y);
printf("\"Wert\" von y: %lf\n", *y);
```

Ausgabe:

```
Adresse von i: 0x7fffb476d89c
Wert von i: 1

Adresse von x: 0x7fff45786740
Wert von x: 2.718282

Adresse von y: 0x7fff45786720
"Wert" von y: 3.000000
```

Bemerkung: Der *-Operator kann auch auf Felder angewandt werden!?

Arbeiten mit Zeigern: indirekter Zugriff auf Variablen

Zeiger erlauben indirekten **lesenden und schreibenden** Zugriff auf zuvor deklarierte Variablen, indem man sie auf die entsprechenden Speicheradressen zeigen lässt.

Beispiel:

```
double x = M_PI, *p1 = &x, *p2;

printf("Wert von x: %lf\n\n", x);

printf("Zugriff via p1: x = %lf\n\n",*p1);

x = -M_1_PI;
printf("Nach Manipulation,\n");
printf("Zugriff via p1: x = %lf\n\n",*p1);

p2=p1;
*p2 = 0.1;
printf("Nach Manipulation via p2,\n");
printf("direkter Zugriff: x = %lf\n",x);
```

Ausgabe:

```
Wert von x: 3.141593

Zugriff via p1: x = 3.141593

Nach Manipulation,
Zugriff via p1: x = -0.318310

Nach Manipulation via p2,
direkter Zugriff: x = 0.100000
```

Achtung: In der Deklaration steht „*p1“ **nicht** für Dereferenzierung, sondern dafür, dass es sich bei der folgenden Variablen um einen Zeiger handelt!

Arbeiten mit Zeigern: „Umbiegen“

Einem Zeiger kann als Variable, deren Inhalt eine Adresse ist, (fast) jeder beliebige (Adress-)Wert zugewiesen werden. Man spricht dann vom „Umbiegen“ des Zeigers.

Beispiel:

```
int a = -5, b = 42,
    *p1 = &a, *p2 = &a;

printf("p1 zeigt auf a:\n");
printf("&a = %p\n", &a);
printf("p1 = %p\n\n", p1);

p1 = &b;
printf("Nach Umbiegen auf b:\n");
printf("&b = %p\n", &b);
printf("p1 = %p\n\n", p1);

p1 = p2;
printf("Zurueckbiegen mit p2:\n");
printf("p1 = %p\n", p1);
```

Ausgabe:

```
p1 zeigt auf a:
&a = 0x7fff72c3ecac
p1 = 0x7fff72c3ecac

Nach Umbiegen auf b:
&b = 0x7fff72c3eca8
p1 = 0x7fff72c3eca8

Zurueckbiegen mit p2:
p1 = 0x7fff72c3ecac
```

Typische Pointer-Fehler I

- Falsche Annahme, dass Name und Funktion automatisch zusammenhängen.

  Beispiele:
  ```
  int a, *a;
  ```

  ⇝ `Fehler: In Konflikt stehende Typen für »a«`

  ```
  double x = 3.14, *px;
  printf("%lf\n", *px);
  ```

  ⇝ `Warnung: »px« is used uninitialized in this function`

- Vergessenes „&“ oder „*“

  Beispiel:
  ```
  int a = 42, *pa;
  pa = a;
  printf("Wert von a: %d\n", pa);
  ```

  ⇝ Bei der Zuweisung `pa = a` wird der Wert von `a` (42) als Adresse aufgefasst.

  ⇝ In der `printf`-Anweisung wird die (vermeintliche) Referenz nicht aufgelöst, sondern die Adresse, die in `pa` gespeichert wurde, als `int` ausgegeben.

Warnung: Durch (schreibenden) Zugriff auf uninitialisierte oder wild verbogene Zeiger oder Zeiger auf einen inkompatiblen Datentyp lässt sich jedes Programm ins Chaos stürzen (z. B. wenn durch den falschen Zugriff ein anderer Zeiger verbogen wird → viel Spaß bei der Fehlersuche!).

Sonderfälle: `void *` und `NULL`

Der Datentyp `void *`

- Ein Zeiger vom Typ `void *` ist **kein** „Zeiger auf Nichts“, sondern ein (universeller) Zeiger, der wie üblich eine Adresse speichert, dessen Typ jedoch (noch) nicht festgelegt ist.
- Einem Zeiger vom Typ `void *` können Adressen von typisierten Speicherobjekten zugewiesen werden.
- Durch einen korrekten Cast kann über den Zeiger auf Inhalte der Speicherobjekte zugegriffen werden. **Dieser Cast sollte immer explizit sein.**

Der Nullzeiger `NULL`

- Formal ist `NULL` = `(void *)0`, d. h. die Zuweisungen `p = NULL` und `p = 0` sind für einen Zeiger äquivalent. Verwendung von `NULL` erhöht aber die Lesbarkeit.
- Der Versuch, `NULL` mittels „`*`“ zu dereferenzieren, führt unmittelbar zu einem Speicherzugriffsfehler.
- Bei der Deklaration eines Zeigers ist es ratsam, mit `NULL` (bzw. einem anderen Wert) zu initialisieren. Dadurch vermeidet man ungewollte Zugriffe auf fremde Speicherbereiche.
- Viele Funktionen mit einem Zeiger-Datentyp als Rückgabewert liefern bei Scheitern `NULL`. Dieser Fall kann zur Kontrolle abgefragt werden.

Typische Pointer-Fehler II

- Vergessener Cast

  Beispiel:

```
int a;
void *z = &a;
printf("a = %d\n", *z);
```

  ⇝ `Warnung: Dereferenzierung eines »void *«-Zeigers`
  ⇝ `Fehler: falsche Benutzung eines void-Ausdruckes`

- Cast eines `void *` auf einen falschen Datentyp (Fehlinterpretation des `void *`)

  Beispiel:

```
int a; void *z = &a;
double *p = (double *)z;
printf("a = %lf\n", *p);
```

  ⇝ `a = 5737915431541871325397352391378549122 73...` (undefinierter Wert)

- Zuweisung statt Vergleich mit `NULL`

  Beispiel:

```
int *p = NULL;

p = ...
if (p = NULL)
  return 1;
```

  ⇝ Der Fall, dass p den Wert `NULL` hat, wird **garantiert** nicht wie beabsichtigt abgefangen. (Wieso nicht?)

Call by reference

Definition

An eine Funktion werden Zeiger auf die Adressen von Variablen übergeben, mit deren Hilfe der Variableninhalt verändert werden kann.

Zur Erinnerung: Call by value

```
void vertausche(int a, int b)
{
  int hilf = b;
  b = a; a = hilf;
  return;
}
```

Ergebnis: Beim Aufruf von `vertausche` in `main` passiert effektiv nichts, da nur die Werte der Funktionsparameter übergeben werden.

Call by reference

```
void vertausche(int *a, int *b)
{
  int hilf = *b;
  *b = *a; *a = hilf;
  return;
}
```

In `main`:

```
int x = 1, y = -2;
vertausche(&x, &y);
```

Ergebnis: Die Werte von `x` und `y` werden tatsächlich vertauscht!

Wie funktioniert Call by reference?

Beispiel: Eine Funktion `fkt` soll zwei Gleitkommazahlen als Ergebnis einer Berechnung liefern (z. B. die Koordinaten eines berechneten Punktes in $\mathbb{R}^2$). Mit Hilfe von `return` kann jedoch nur **ein** Wert zurückgegeben werden.

Idee: Es werden zwei `double`-Variablen `erg1` und `erg2` an die Funktion übergeben, in welche die Ergebnisse gespeichert werden sollen.

Der Programmcode sieht wie folgt aus:

- **Deklaration** von `fkt` als `void fkt(double *e1, double *e2);`
- **Aufruf in** `main` in der Form `fkt(&erg1, &erg2);`
- **Definition** von `fkt` enthält Zuweisungen `*e1 = ...; *e2 = ...;`

Folgendes passiert bei der Ausführung:

- Durch Anwendung des `&`-Operators entstehen zwei *Zeiger auf* `double`.
- Diese beiden Zeiger werden als Parameter (in Übereinstimmung mit der Deklaration) an `fkt` übergeben.
- Die Funktion `fkt` „sieht“ zwar die beiden Variablen nicht, verfügt aber mit den Zeigern über ihre Adressen.
- Mit dem `*`-Operator wird die Referenz aufgelöst, und die Zuweisung von Werten an die betreffenden Speicherbereiche wird möglich.

Fazit: Call by reference überwindet die Grenze des *scope* von Variablen.

Typische Pointer-Fehler III

- Call by reference überwindet zwar die *scope*-Grenze, jedoch nicht die *lifetime*-Grenze!

Beispiel:

```
int *neuer_zeiger(void)
{
  int a;
  return &a;
}
```

⇝ Nach Beendigung der Funktion ist die Lebenszeit der Variablen a vorbei.

⇝ Der (gültige!) erzeugte Zeiger verweist auf **irgendeine** Speicherstelle, die längst anderweitig vergeben sein kann.

- Call by value mit Pointern statt Call by reference

Beispiel:

```
void biege(int *a, int *b)
{
  int *hilf = b;
  b = a; a = hilf;
  return;
}
```

⇝ Die Speicherreferenzen werden nicht aufgelöst. Stattdessen finden (ausschließlich lokale!) Zuweisungen von Adressen an Zeigervariablen statt.

⇝ Pointer, die als Parameter übergeben werden, bleiben unverändert. (Call by value!!)

Zeigerarithmetik

Erlaubte Operationen mit Zeigervariablen

- Addition und Subtraktion von Integer-Werten.
- Arithmetische Zuweisungen += und -= .
- Inkrement ++ und Dekrement -- .
- Differenzbildung zweier Zeiger.

Beispiel:

```
double x1=5.5, x2=7, *p1=&x1, *p2=&x2;

printf("&x1 = %p \n" ,&x1);
printf("&x2 = %p \n\n",&x2);

printf("&p1 = %p, p1 = %p \n*p1 = %lf\n",
       &p1,p1,*p1);
printf("&p2 = %p, p2 = %p \n*p2 = %lf\n\n",
       &p2,p2,*p2);
p2 = p1 + 2;
printf("&p2=%p, p2=%p \n*p2=%lf\n\n",
       &p2,p2,*p2);
p2--;p2--;
printf("&p2=%p, p2=%p \n*p2=%lf\n",
       &p2,p2,*p2);
```

Ausgabe:

```
&x1 = 0x28ac08
&x2 = 0x28ac00

&p1 = 0x28abfc p1 = 0x28ac08
*p1 = 5.500000
&p2 = 0x28abf8 p2 = 0x28ac00
*p2 = 7.000000

&p2 = 0x28abf8 p2 = 0x28ac18
*p2 = 1807675861650890483

&p2 = 0x28abf8 p2 = 0x28ac08
*p2 = 5.500000
```

Zeiger und Felder

Erinnerung: Der *-Operator lässt sich auf statische Felder anwenden und liefert das erste Feldelement.

Frage: Wieso?

Antwort:

```
int main(void)
{
  double x[] = {-3.14, 0.0, 4};

  printf("Werte:\n");
  printf(" *x = %lf\n", *x);
  printf("x[0] = %lf\n", x[0]);

  printf("\nAdressen:\n");
  printf("   x = %p\n", x);
  printf("  &x = %p\n", &x);
  printf("&x[0] = %p\n", &x[0]);

  return 0;
}
```

```
Werte:
  *x = -3.140000
x[0] = -3.140000

Adressen:
    x = 0x7fffc291a0b0
   &x = 0x7fffc291a0b0
&x[0] = 0x7fffc291a0b0
```

Fazit: Der Feldname ist zugleich ein **konstanter** Zeiger auf das erste Feldelement!

↪ Manipulationsversuche von `x` der Form `x=x+2;` `x++;` usw. scheitern.

Folgerung: Mit einem Zeiger auf den Datentyp der Feldelemente kann man ebenfalls auf das Feld zugreifen!

Zeiger und Felder

Frage: Das erste Feldelement ist schon mal ein Anfang, aber wie erhält man die restlichen mit Hilfe eines anderen Zeigers?

Antwort: Genau wie beim Original-Feld mit „`[]`“.

```
int main(void)
{
  double x[] = {-3.14, 0.1, 4};
  double *p = x; // oder &x
                 // oder &x[0]

  printf("x[2] = %lf\n", x[2]);
  printf("*(x+2) = %lf\n", *(x+2));

  printf("\n*p++ = %lf\n",*p++);
  // dauerhafte Veraenderung von p!
  printf("*p = %lf\n", *p);
  printf(" p[0] = %lf\n", p[0]);
  printf(" p[1] = %lf\n", p[1]);

  return 0;
}
```

```
x[2] = 4.000000
*(x+2) = 4.000000

*p++ = -3.140000
*p = 0.100000
p[0] = 0.100000
p[1] = 4.000000
```

Da die Elemente **hintereinander** im Speicher angeordnet sind, wird „`x[n]`“ intern interpretiert als „springe um `n` Speicherpositionen weiter und gib den Wert an dieser Stelle zurück“.

In Pointer-Sprache ausgedrückt: `*(x+n)`

Zeigerarithmetik: Zeiger und Felder

Bemerkungen

- Klammer bei `*(x+2)` ist nötig, da der Inhaltsoperator `*` eine höhere Priorität als der Additionsoperator besitzt.
- Inkrement- und Inhaltsoperator sind beide von der selben Prioritätsstufe. Wichtig: der Inkrementoperator bezieht sich auf den Zeiger und nicht auf den dereferenzierten Inhalt. Der Zeiger wird dauerhaft umgesetzt, d.h. er zeigt nun auf eine andere Adresse im Speicher.

Merke
Sei A ein Feld von einem beliebigen Typ. Dann sind die jeweiligen Ausdrücke in den folgenden Boxen äquivalent:

```
A[i]   ,   *(A+i)
```

Der zweite Ausdruck ist jedoch schwerer lesbar und wird als stilistisch schlecht angesehen.

```
&A[i]   ,   A+i
```

```
A   ,   &A   ,   &A[0]   ,   A+0
```

Zeigerarithmetik: Zeiger und Felder