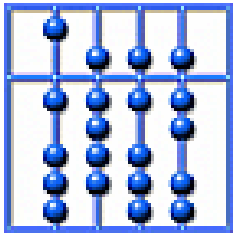
Einführung in die Programmierung II

5. Zeiger

Stefan Zimmer

24. 5. 2006



Bezüge als Objekte

- Bisher kennen wir als Bezüge (Lvalues) nur Variablennamen
- Jetzt kommt eine neue Sorte dazu, die *Zeiger* (Pointer), die es ermöglichen, Bezüge abzuspeichern
- Die Zeiger selber sind ihrerseits Objekte – wir werden sie abspeichern können, als Funktionsparameter übergeben und als Funktionsergebnis erhalten können und einige Operationen mit Zeigern ausführen können
- Wichtig wird daher sein, dass man den Überblick behält: wann reden wir über den Zeiger als eigenständiges Objekt (analog zum Variablennamen – mit dem man aber nicht viel anstellen konnte) und wann reden wir über das Objekt, auf das er verweist (analog zum Wert der Variable)

Datentypen für Zeiger

- Ein Zeiger hat als Objekt natürlich einen Datentyp
- Zu einem beliebigen Typ T gibt es dazu einen Typ „Zeiger auf T“ (strenggenommen: „Zeiger auf Objekte vom Typ T“), der als T * notiert wird
- Zu verschiedenen Datentypen gehören also verschiedene Typen für die zugehörigen Zeiger
- Außerdem gibt's noch Zeiger vom Typ `void *`, die (analog zu Funktionen, die als `void` deklariert sind) zu keinem Basistyp gehören (mehr dazu später)

Deklaration von Zeigervariablen

- Wie gesagt: da Zeiger Objekte sind, kann ich sie in einer Variable abspeichern, als Parameter einer Funktion übergeben oder als Funktionsergebnis bekommen
- Die Deklaration eines Zeigers vom Typ `T *` passiert dadurch, dass in einer Deklaration vom Typ `T` dem Bezeichner ein `*` vorangestellt wird:
  ```
  int *p; // ein Zeiger vom Typ int *
  ```
- Vorsicht: das sieht auf den ersten Blick wie eine Deklaration mit Typnamen `int *` aus – der `*` gehört aber zum Bezeichner, nicht zum `int`:
  ```
  int *p, x;
  ```
  definiert eine Zeigervariable `p` und eine `int`-Variable `x`!

Der Nullzeiger

- Ein Sonderfall sind Zeiger, die auf kein Objekt verweisen
- Um markieren zu können, gibt es einen speziellen Wert, den *Nullzeiger*, den man z.B. in einem Vergleich benutzen kann
- Um ihn aufzuschreiben, nutzt man aus, dass eine einzige Umwandlung von Zahlen in Zeiger und umgekehrt erlaubt ist (und bei Bedarf automatisch passiert): zwischen der Zahl 0 und dem Nullzeiger. Die Abfrage „wenn p nirgendwo hin zeigt“ könnte man also schreiben als `if (p==0) ...`
- Schöner: man verwendet die vordefinierte Konstante NULL (`#include <cstddef>` oder – C-Variante – `#include <stdio.h>`):

```
if (p==NULL) ...
```

Der Adressoperator &

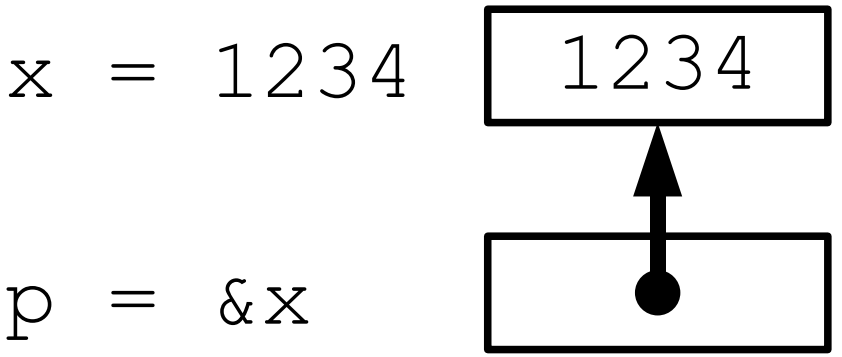
- Nun wollen wir aber Zeiger haben, die auf irgendwas verweisen, z.B. auf den Inhalt einer Variable

```
int x = 1234;
```

- Dazu gibt es den Adressoperator **&**, den man auf einen Lvalue (z.B. `x`) anwenden kann, man erhält einen Zeiger, der auf den Inhalt des Lvalues verweist:

```
int  *p;
p = &x;
```

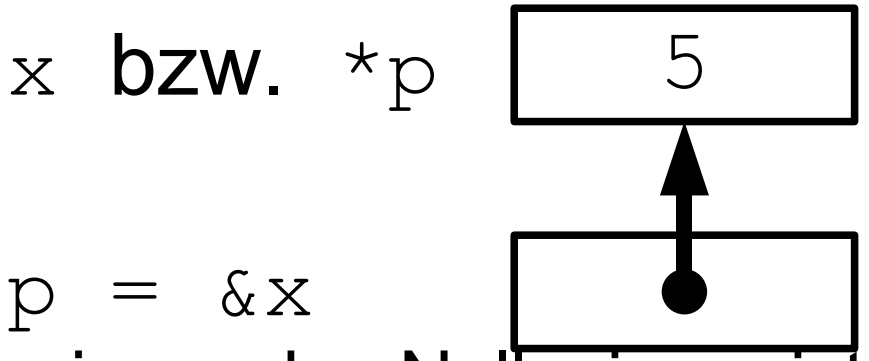
- Diesen Prozess nennt man auch *Referenzieren*



Dereferenzieren mit *

- Der eben erzeugte (und in `p` gespeicherte) Zeiger ist selber kein Lvalue
- Um wieder auf das Objekt (hier: den Inhalt von `x`) zugreifen zu können, müssen wir *dereferenzieren* (das Referenzieren umkehren) – dazu gibt es den Operator `*`
- Auf einen Zeiger angewendet, ergibt der einen Lvalue, der auf das Objekt verweist. Das kann also z.B. auf der linken Seite einer Zuweisung stehen:

```
*p = 5; // gleiche Wirkung wie x = 5;
```



- Das Dereferenzieren des Nullzeigers ist ein Fehler (vergleichbar mit der Division durch 0)

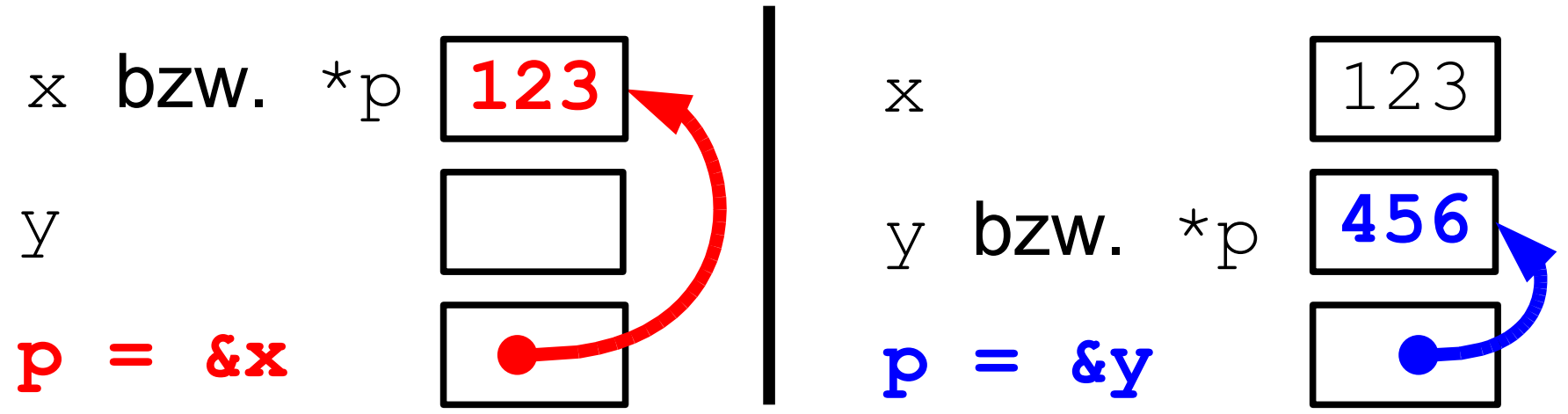
Präzedenzregeln

- Bevor wir uns näher mit Anwendungen von Zeigern befassen, bringen wir noch schnell die Überlegungen zur Syntax zu ende und fügen die neuen Operatoren in die Tabelle der Vorrangregeln (Kap.2, Folie 33) ein
- Beide Operatoren `*` und `&` (als einstellige vorangestellte Operatoren – im Gegensatz zum Multiplikations- bzw „Bitweise-Logisches-Und“-Operator) werden zu den anderen einstelligen vorangestellten Operatoren eingefügt, also in die zweite Zeile
- Beispiel: `*p++` ist zu lesen als `*(p++)` und nicht als `(*p)++`, weil `++` höhere Priorität hat als `*`.
  (ich würde hier übrigens lieber Klammern setzen als mich an die Regeln zu erinnern)
- Die Semantik dieses Ausdrucks ist hier noch egal

Beispiel: Referenzieren, Dereferenzieren

- Ein Beispiel mit zwei Variablen und einem Zeiger:

```
int x, y;
int *p;
p = &x;       // *p und x sind synonym
x = 123;
std::cout << *p << '\n';
p = &y;       // nun sind *p und y synonym
*p = 456;
std::cout << y << '\n';
```



Operationen mit Zeigern

- Neben dem Referenzieren und Dereferenzieren sind weitere Operationen mit Zeigern erlaubt
- Es ist die Addition `p+x` eines Zeigers `p` mit einer ganzen Zahl `x` definiert – die Semantik dieses Ausdrucks kommt noch
- Wo eine Bedingung verlangt ist (`if` oder im Kontrollteil eines `for` oder eines `while`,...), darf auch ein Zeiger sein. Die Bedingung gilt als erfüllt, wenn der Zeiger nicht der Nullzeiger ist
- Mit einem Zeiger `p` lesen wir z.B.

```
if (p) ...
```

als

```
if ((p) != NULL) ...
```

Zeiger als Parameter und Fkt.-Ergebnis

- Zeiger können problemlos als Parameter und Ergebnis einer Funktion auftreten. Die wenig sinnvolle Funktion

```
int *zweites(int *p1, int *p2) {
    return p2;
}
```

könnte man z.B. so verwenden:

```
int i1=111, i2=222;
int *p1, *p2, *p3;
p1 = &i1; p2 = &i2;
p3 = zweites(p1, p2); // p3 ist nun &i2
std::cout << *p3;
```

- Übergeben wird natürlich das Zeiger-Objekt, nicht das Objekt, auf das es verweist
- Die Konsequenzen von Zeigern als Parametern werden schwerwiegend sein, wir sehen uns jetzt an, warum das so ist

Lebensdauer des Objekts

- Unverständiges Hantieren mit Zeigern ist saugefährlich, weil wir dadurch einen Bezug auf eine Variable bekommen können, dessen Lebensdauer die vom Objekt übersteigt (und von diesem Ende nichts „merkt“!):

```
{ int *p;
  {  int i;
     p = &i;
  }
  *p = 5;  // Autsch!!
}
```

- Vielleicht sieht man hier besser, wie schön der Automatismus automatischer Variablen ist: hätten wir `i=5` statt `*p=5` geschrieben, hätte der Übersetzer uns das nicht durchgehen lassen (da außerhalb des Gültigkeitsbereichs des Bezeichners `i`)

Verlassen des Gültigkeitsbereichs

- Anders als bei einem Zugriff nach Ende der Lebensdauer eines Objektes kann eine Verwendung einer Variable außerhalb des Gültigkeitsbereichs ihres Bezeichners – mittels eines Zeigers – sinnvoll sein!
- Ein Beispiel:

```
void fuenf(int *p) {
   *p = 5;
}
void fuenf_test() {
   int x;
   fuenf(&x); // Verändert der Wert von x!
   std::cout << x;
}
```

- `fuenf(&x)` verändert den Wert von x außerhalb des Gültigkeitsbereichs (aber innerhalb der Lebensdauer)

Variablenparameter

- Das ist der Weg, um in C *Variablenparameter* zu verwenden, d.h. der aufgerufenen Funktion zu ermöglichen, Variablen der aufrufenden Funktion zu verändern: statt des Wertes einen Zeiger auf das Objekt übergeben!
- Das ist keine Ausnahme des Werteparameter-Prinzips: Parameter ist der Zeiger und der wird genau so kopiert wie Parameter in C das immer werden
- Dadurch werden neue Seiteneffekte möglich (das Verändern von Variablen mittels eines Funktionsaufrufs) – das kann nützlich sein, kann die Lesbarkeit des Programms aber auch völlig zerstören
- Am `&` vor dem Aktualparameter sieht man, dass Gefahr droht – in anderen Sprachen (schon in C++) können Variablenparameter beim Aufruf ganz „normal“ aussehen

Variablenparameter: Beispiel

- Hier nochmal der erfolglose Versuch, den ++-Operator nachzubilden (Kap. 4, Folie 17) und wie man's besser macht:

```
void plusplus1(int x) {
  x = x + 1;
}
void plusplus2(int *p) {
  *p = *p + 1;
}
void test() {
  int a = 5;
  plusplus1(a);     // a ist immer noch 5
  std::cout << a << '\n';
  plusplus2(&a);    // jetzt ist a==6
  std::cout << a << '\n';
}
```

Zeiger auf Zeiger

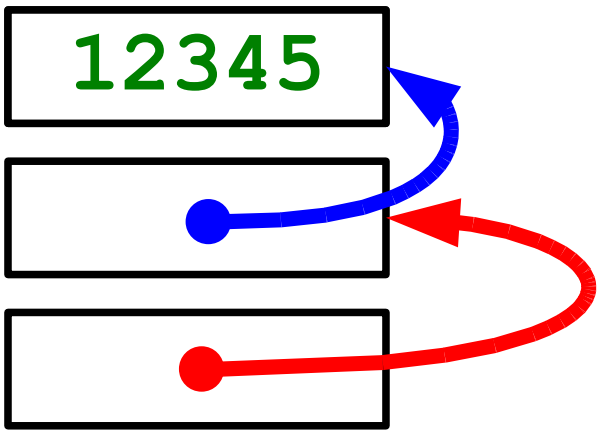
- Da Zeiger selber Objekte sind, können wir auch Zeiger auf Zeiger erzeugen – das mag verwirren, ist aber nichts prinzipiell Neues:

```
int x;          // int
int *p;         // Zeiger auf int
int **pp;       // Zeiger auf Zeiger auf int
pp = &p;        // *pp entspricht nun p
*pp = &x;       // äquivalent: p = &x;
**pp = 12345;// äquivalent: x = 12345;
std::cout << x << '\n'; // 12345
```

`x` bzw. `*p` bzw. `**pp`

`p` bzw. `*pp`

`pp = &p`



Zeigervariablen deklarieren, nochmal

- Für die, die's genau wissen wollen: die Deklaration `int **pp` passt gar nicht in das Schema von Folie 4
- Folie 4 war auch die Variante für Kinder, die wirkliche Regel für die bisher vorkommenden Konstruktionen (später wird's noch komplizierter) geht so:
  - Hat in einer Deklaration mit Typnamen **T** ein Element D der Liste die Form ***D1**
  - und in einer Deklaration **T D1** wäre der Name von **D1** vom Typ **M T**,
  - dann hat der Name von D den Typ **M** „**Zeiger auf**“ **T**
- Alles klar? Vielleicht besser ein Beispiel: `int **pp;`
  Hier ist **T=„`int`“**, D=„`**pp`“, mithin **D1=„`*pp`“** vom Typ **„Zeiger auf `int`“**, also **M = „Zeiger auf“**
  Gibt für `pp`: **„Zeiger auf Zeiger auf `int`“**