§6 Zeiger und Referenzen - Allgemeines

Leitideen: *Zeiger sind Adressen mit zusätzlicher Typinformation über das dort gespeicherte Objekt.*

Die Vereinbarungssyntax soll der Ausdruckssyntax entsprechen und zugleich einfach interpretierbar sein.

Die Implementierung von Referenzparametern kann mit Zeigern erfolgen, die als Wertparameter übergeben werden. Die Analyse dieses Mechanismus führt zum Referenzdatentyp. Dasselbe gilt auch für konstante Referenzen.

Bei der Vereinbarungssyntax von Referenzen wird das Lesen von innen nach außen beibehalten, die Ähnlichkeit zur Ausdruckssyntax jedoch aufgegeben.

§6 Zeiger und Referenzen - Themenübersicht

- Eine einfache Idee und ihre Problematik
- Motivation und Operatoren
- Beispiel
- Veranschaulichung I,II
- Vereinbarungssyntax
- Explizite und implizite Verwendung von Zeigern I,II
- Referenzparameter und Zeiger I,II
- Konstanten I,II
- Referenzen auf Variablen I,II
- Referenzen auf Konstanten

Eine einfache Idee . . .

Voraussetzungen

- Kleinste adressierbare Speichereinheit (Byte) als Datentyp vorhanden, *hier* `byte` genannt. (Ganzzahltyp mit 8 Bit Länge)
- Ganzzahldatentyp, der Adresswerte speichern kann: *hier* `long` verwendet

Pseudocode (*Kein C++!)*

```
byte b = 37;
cout << &b;      // Adresse von b ausg., z.B. 2048
*(2048L) = 99; // Wert von b jetzt 99
```

Neue Operatoren

- `&` „Adressoperator“ und `*` „Inhaltsoperator“
- Es gilt: `*&b == b` und `&*a == a`, letzteres aber nur, wenn `a` die Adresse eines Objekts (Variable) ist.

. . . und ihre Problematik

- Nur einzelne Bytes adressierbar, nicht längere Objekte.
- Z.B. bei Zahlen vom Typ `int` oder `double` müsste der Programmierer die interne Darstellung kennen
- Die Abbildung Adresse $\rightarrow$ `long` ist maschinenabhängig.

Lösungsansatz in C++

- Einführung von Datentypen (Zeigertypen), die Adressen speichern können.
- Im jeweiligen Zeigertyp ist die Information enthalten, ob es sich z.B. um die Adresse eines `double`- oder `int`-Objekts handelt.
- Der Adressoperator liefert Zeiger (Adressen mit Typinformation).
- Der Inhaltsoperator liefert zu Zeigern das zugehörige Objekt, das an dieser Adresse gespeichert ist.
- Variablen vom Zeigertyp sind möglich und gebräuchlich.
- Der Datentyp `char` übernimmt in C++ die Rolle von `byte`.

Zeiger - Überblick

Motivation

(Speicherplatz)adresse und Typ des dort gespeicherten Datenobjekts (z.B. Wert einer Variablen) ermöglichen die korrekte Interpretation der dort gespeicherten Bits.

Entsprechender Datentyp: Zeiger (auf Typ)

Bsp.: Die Adresse einer `int`-Variable wird mit Hilfe des Datentyps „Zeiger auf `int`“ angegeben.

Operatoren

- Adressoperator `&`: Liefert Adresse+Datentyp (Zeiger) eines Datenobjekts (z.B einer Variablen)
- Inhaltsoperator `*`: Liefert zu einem Zeiger das an der zugehörigen Adresse gespeicherte Datenobjekt und interpretiert es gemäß der Typinformation.
- Zeigerwert (Adresse) `0` bedeutet, dass dort *nichts* gespeichert ist.
  Die Anwendung des Inhaltsoperators darauf ist unzulässig.
  Kann zu Programmabstürzen führen (segmentation fault)

Zeiger - Beispiel

- Oft werden Zeigervariablen mit einem p (“pointer“) am Ende gekennzeichnet, etwa im folgenden Beispiel ip und jp (Zeiger auf `int`)

Programm	*i*	*j*	*ip*	*jp*
`int i=5,j,*ip,*jp=0;`	5	undef	undef	`0`
`ip=&i;`			Adr. von *i*	
`j=*ip;`		5		
`jp=ip;`				Adr. von *i*
`*jp=6;`	6			

Caveat

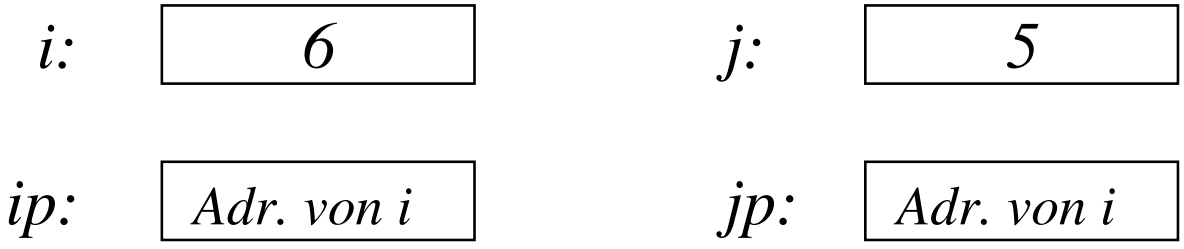
- Die Syntax für die Initialisierung und für die Zuweisung von Zeigern ist unterschiedlich:

  `int *jp=0;` wirkt wie `int *jp; jp=0;`

  und *nicht* wie `int *jp; *jp=0;`

Zeiger - Veranschaulichung I

Variablenbelegung

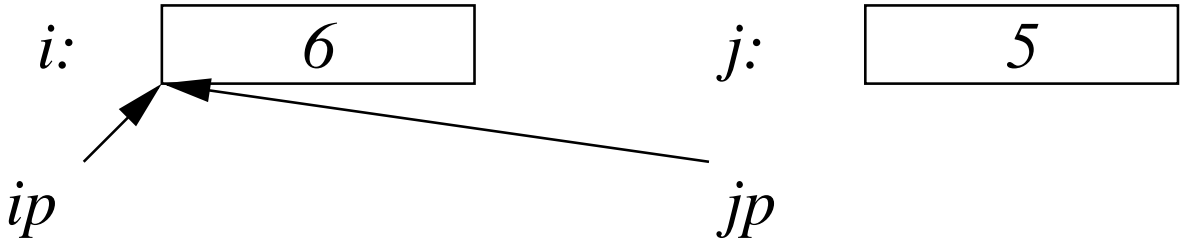


Programmcode *(ausgeführt)*

```
int i,j,*ip,*jp;
i=5;
jp=0;
ip=&i;
j=*ip;
jp=ip;
*jp=6;
```

Zeiger - Veranschaulichung II

Variablenbelegung



Programmcode *(ausgeführt)*

```
int i,j,*ip,*jp;
i=5;
jp=0;
ip=&i;
j=*ip;
jp=ip;
*jp=6;
```

Zeiger - Vereinbarungssyntax

`&i`	Adresse der Variable i
`*ip`	Inhalt des durch ip adressierten Speicherplatzes
`int *ip`	ip Zeiger auf int,
	d.h ip enthält Adresse eines int-Speicherplatzes

Ziel der Vereinbarungssyntax: Ähnlichkeit zu Ausdrücken

`*ip`	`int`-Zahl
`int *ip`	ip Zeiger auf int

`double *a[N]`	?	
`double *(a[N])`	?	(äquivalent zur vorigen Zeile!)
`double (*b)[N]`	?	

Mechanismus: Lesen von innen nach außen

`double * (a [N] )` N-Vektor aus Zeigern auf double

(3) (2) (1) (1) (2) (3)

`double ( * b) [N]` Zeiger auf N-Vektor von double

(3) (1) (2) (1) (2) (3)

Explizite und implizite Verwendung von Zeigern

- Zeiger können als Ersatz für Referenzparameter eingesetzt werden. *(→ Inf.bl. 11, S.4 und nächste Folienseiten)*
- Referenzen und konstante Referenzen können mit Hilfe von Zeigern bzw. Zeigern auf Konstante implementiert werden. *(→ Folgeseiten)*
- Die Übergabe eines C-Vektor (a) erfolgt durch Übergabe des Zeigers auf die Komponente 0 des Vektors ($\&a[0]$). Der Zugriff auf eine Vektorkomponente kann dann durch Bestimmung des Offsets durchgeführt werden. Die Analyse dieses Mechanismus führt zur Zeigerarithmetik. *(→ Folie Zeigerarithmetik)*
- Bei der Übergabe von C-Matrizen wird sinnvollerweise ein Hilfsvektor aus Zeigern auf die Zeilenanfänge übergeben. Ähnliches gilt für die Übergabe von Kommandoargumenten. *(→ Folie Zeigerarithmetik)*

Explizite und implizite Verwendung von Zeigern II

- Dynamisch allozierte Größen werden über Zeiger angesprochen. *(→ Folie Zeigerarithm.: Dyn. Speich.reserv.)*
- Mit Zeigern können kompliziertere Datenstrukturen wie verkettete Listen und Bäume aufgebaut werden. Diese werden praktischerweise in eigenen Behälterklassen gekapselt, z.B in der STL. *(→ Programmieren II)*
- Zum Durchwandern von STL-Behältern gibt es Iteratoren, die als verallgemeinerte Zeiger betrachtet werden können. *(→ Programmieren II)*
- Funktionszeiger stehen in einem ähnlichen Verhältnis zu Funktionen wie Zeiger zu Vektoren. Intern werden sie bei der Übergabe von Funktionen und bei der Implementierung von virtuellen Funktionen verwendet. *(→ Programmieren II)*
- Polymorphes Objektverhalten ist in C++ nur über Zeiger verfügbar. *(→ Programmieren II)*

Referenzparameter und Zeiger

- **Referenzparameter**

```
void verdopple(double& x)    // x Referenzparam.
{ x = x * 2; }

int main()
{ double x=4;
  verdopple(x);     // Wert von x: 8
}
```

- **Zeiger**

```
void verdopple(double *xp)   // xp Zeiger
{ *xp = *xp * 2; }

int main()
{ double x=4; double *xp=&x;
  verdopple(xp);    // Wert von x: 8
}
```

Referenzparameter und Zeiger II

- *Implementierung von Referenzparametern mit Zeigern:*

Parameterliste:	`T& x`	$\rightarrow$	`T *xp`
Parameter in Fkt.:	`x`	$\rightarrow$	`*xp`
Vereinb. in aufruf. Fkt.:	*zusätzl.:*		`T *xp = &x;`
Argument in Fkt.aufruf:	`x`	$\rightarrow$	`xp`

- *Anmerkung 1:*

  Die Adresse der Variable `x` wird auf zwei unterschiedliche Variablen kopiert:

  - Variable `xp` in der aufrufenden Funktion
  - Parameter `xp` in der aufgerufenen Funktion

  Die aufgerufene Funktion kann zwar *nicht* den Wert der Variable `xp` in der aufrufenden Funktion ändern, *aber* den Wert der Variablen `x`, auf den beide `xp` zeigen.

- *Anmerkung 2:*

  Besser wäre es, statt der Variablen `xp` eine gleichnamige Konstante zu benutzen ($\rightarrow$ nächste Seite)

Konstanten

- Deklarationsangabe `const`: Konstante (schreibgeschützt)
  Bsp.:
  ```
  const int n   // Integerkonstante
  int const n   // gleichbedeutend
                // wie vorige Zeile
  ```
- Bedeutung bei C-Vektoren: Komponenten konstant
  Bsp.: `const char s[5] // Datentyp von "Text"`
- Bedeutung bei Zeigern: positionsabhängig
  Gespeicherte Adresse oder Zielobjekt schreibgeschützt oder beides
  Bsp.:
  ```
  const char *p  // Zeiger auf eine Konst.
  char * const q // Konst. Zeiger
  const char * const r // Konst. Zeiger
                       // auf Konst.
  ```
- Deklarationen von innen nach außen lesbar
  Bsp.: p – Zeiger auf konstantes `char`
  q – Konstanter Zeiger auf `char`
  r – Konstanter Zeiger auf konstantes `char`

Konstanten - Fortsetzung

- Schreibschutz bei Konstanten als Unterstützung gedacht – lässt sich umgehen
- Initialisierung sogar syntaktisch notwendig – wird vom Compiler überprüft
- Bei Initialisierung: `const`-Attribute auf linker und rechter Seite irrelevant

  Ausnahme: Initialisierung von Zeiger auf Nichtkonstante mit Zeiger auf Konstante *verboten*
- Initialisierung $\neq$ Zuweisung
  Zuweisung an Konstante natürlich *unzulässig*
- Parameterübergabe und Rückgabe in Funktionen: Initialisierung, *keine* Zuweisung
- Eingebaute Datentypen: Initialisierung $\approx$ Zuweisung
- In Klassen: Zuweisung durch überlad. Gleichheitsop. Initialisierung durch Konstruktoren *(später!)*

Referenzen auf Variablen

Begriff

Referenz auf Variable: weiterer Name für eine Variable
(Diese muss bei der Vereinbarung mit angegeben werden)

- *Bsp.:*
```
int  i=5;
int& j=i; // j anderer Name fuer i
          // j Referenz auf i
          // Wert von j: 5
i++;      // Wert von i,j: 6
j++;      // Wert von i,j: 7
```
- Zulässige Schreibweisen: `int& i`, `int &i`, `int & i`
  Üblich: Erste Schreibweise
- Lesen von innen nach außen: `&` steht für Referenz
  Bsp.:
```
char *p    ; // p Zeiger auf char
char*& cp=p; // Ref. auf Zeiger auf char
             // cp anderer Name fuer p
```

Referenzen auf Variablen - Fortsetzung

- Bisher: Vereinbarungssyntax ähnlich zu Ausdruckssyntax

  Bsp.:
  ```
  int *ip   // Zeiger auf int
  *ip       // int-Wert
  ```

  Referenzen: Vereinb.syntax $\neq$ Ausdruckssyntax

  Bsp.:
  ```
  int& j    // Referenz auf int
  j         // int-Wert
            // nicht mehr: &j int-Wert
  ```

- Realisierung von Referenzen auf Variablen mittels konstanter Zeiger denkbar:

  ```
  int& j = i;
  ```

  Umsetzung durch Compiler:

  ```
  int * const jp = &i
  ```
  Anlegen eines konst. Zeigers

  ```
  j → (*jp)
  ```
  Überall ersetzen

- Referenzen auch als Rückgabewerte von Funktionen möglich *(später!)*

Referenzen auf Konstanten („konstante Referenzen“)

Begriff

Referenz auf Konstante:

1. weiterer Name für eine Variable, deren Wert über die Referenz *nicht* geändert werden darf
2. weiterer Name für eine Konstante,
   ggf. wird eine Temporärvariable erzeugt

Der erste Name muss bei der Definition angegeben werden.

- *Beispiel:*

```
int i=5;
const int& j=i; // j ander. Name f. i
const int& k=4; // k ander. Name f. 4
i++;            // Wert von i und j: 6
j++;            // verboten
k++;            // verboten
```

- Realisierung mittels konstanter Zeiger auf Konstanten denkbar.