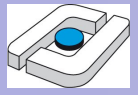
Zeiger

Layout: Mario Hauke / Oliver Henkel



II. Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

Zeiger in C++

C++ verwendet Wertesemantik als Standard: `T t;` erzeugt ein Objekt vom Typ `T` und jede Verwendung der Variable `t` operiert direkt auf dem Objekt.

Referenztypen `T& t_ref = t;` sind nicht so flexibel wie Objektreferenzen, wie man sie aus der Sprache Java kennt, da `t_ref` unwiderruflich stets an sein Referenzziel `t` gebunden ist.

Das Zeigerkonzept in C++ realisiert

- eine flexiblere Handhabung von Referenzen
- die dynamische Speicherverwaltung von Daten auf dem Heap (durch den Programmierer), anstelle des Stacks (automatische Verwaltung durch das System)
- Polymorphie

Dabei spielt es in C++ keine Rolle, ob der Typ `T` eine Klasse oder ein primitiver Datentyp ist.

II.1 Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.1 Adressen und Inhalte

Operatoren für die Referenzsemantik

- Der &-Operator referenziert den Operanden, d.h. liefert die Speicheradresse des Operanden. & nennt man deshalb auch Referenzierungs-Operator
- Der *-Operator dereferenziert den Operanden, d.h. liefert den zugehörigen Inhalt/Wert der Speicheradresse. * heißt deshalb auch Dereferenzierungs-Operator

Speicheradresse	$\xrightarrow{*}$ / $\xleftarrow{\&}$	Inhalt
adresse=&wert		wert
adresse		wert=*adresse

Im Beispiel steht wert für eine Variable beliebigen Typs (z.B. **int**) und adresse für einen Zeiger

II.1a Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

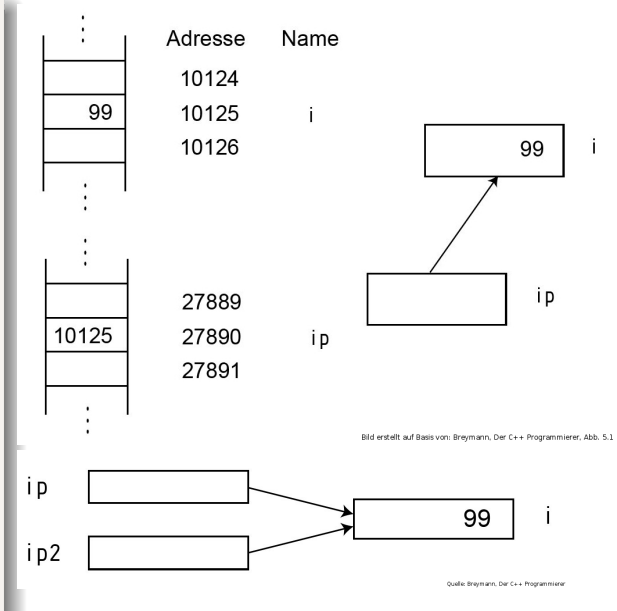
II.1a Zeigervariablen (Pointer)

Deklaration und Verwendung von Zeigern

- Da a priori nicht klar ist, welcher Art der Inhalt einer Speicheradresse ist (int-/float-Wert, Zeichenkette...), gehört zu einer Referenz stets ein Datentyp, der die Art der Dereferenzierung bestimmt.
- Eine Variable, die eine Adresse für einen bestimmten Datentyp aufnehmen kann, heißt allgemein Zeiger und wird wie folgt deklariert:

<Typ> *<Variablenname>;

- **int** *p; vereinbart einen Zeiger vom Grundtyp integer
- **int** i=99, *ip=&i; erzeugt Speicher für i (Wert 99) und ip; ip enthält als Wert die Adresse von i
- In Ausdrücken kann *ip überall dort stehen, wo auch i stehen kann
- **int** *ip2=&i; ist ein weiterer Zeiger auf i



II.1a Zeigervariablen

Beispiel

☞ ZEIGER_ERSTESBEISPIEL

```
1  // Deklaration eines (nicht initialisierten!)
2  // double-Zeigers und einer double-Variablen
3  double *a, b;
4
5  a=&b;      // a zeigt jetzt auf b
6  *a=5;      // b=5
7  (*a)++;    // b=6
8
9  // Ein Zeiger auf Zeiger
10 double **c = &a;
11 cout << *c << " " << **c;
12 // gibt die Adresse von b gefolgt von dem
13 // aktuellen Wert von b aus
14
15 //! Nicht initialisierte Zeiger sind gefaehrlich
16 // Im folgenden Beispiel wird ein zufaelliger
17 // Speicherbereich durch den int-Wert 5 ueberschrieben.
18 int *d; //erzeugt Speicher fuer die Zeigervariable d,
19 *d=5;    //aber nicht fuer die Adresse, auf die d zeigt!
20 //! Der weitere Verlauf dieses Programme ist jetzt
21 //! unvorhersehbar!
```

II.1a Zeigervariablen

Komponentenselektion in Objekten

☞ ZEIGER_KOMPONENTENSELEKTION

```
class K {
public:
 int x;
};

K k; K *pK=&k;
// Zugriff auf Komponente x im Zeigerziel:
(*pK).x;   /* oder kuerzer: */ pK->x;
```

II.1a Zeigervariablen

Der Nullzeiger

☞ Zeiger_Nullbeispiel

Jede Zeigervariable kann mit NULL (Speicheradresse 0) initialisiert werden. Dies kann im Programm abgefragt werden

```
int *p=NULL;
...
if (p) {   // p!=NULL
 <Mit p oder *p arbeiten>
} else {   // p==NULL
 <p zuweisen>
}
```

Warnung | undefinierte Zeiger vermeiden

Zeiger sollten NIEMALS undefiniert sein. Ist das Verweisziel bei der Deklaration eines Zeigers (noch) nicht bekannt, sollte der Zeiger mit NULL initialisiert werden

II.1a Zeigervariablen

typenlose Zeiger

☞ ZEIGER_VOIDZEIGER

Zur direkten byteweisen Speichermanipulation stehen typenlose Zeiger zur Verfügung. Der Grunddatentyp heißt **void** (leer) und entspricht einer Speicherstelle von 1 Byte Größe

`void`-Zeiger sind mit allen Datentypen kompatibel!

```
void *zeiger;
double d;
long int l;
zeiger=&d;   // geht
zeiger=&l;   // geht auch
```

II.1a Zeigervariablen

konstante Zeiger

☞ ZEIGER_KONSTANT

Es gibt auch Zeiger, die wie Referenzen unverrückbar an ihre Zieladresse gebunden sind

```
class K {};
K k;
K* const pK = &k; // pk wird unwiderruflich
                  // an das Objekt k gebunden
//! Beachte: Das Schluesselwort const steht
//! hinter K*
```

II.1a Zeigervariablen

Übung I

OK oder nicht OK?

```
int* x,y;
int  i=10;
x=y=&i;
```

II.1a Zeigervariablen

Übung II

OK oder nicht OK? / Ausgabe?

```
int i=7, *p, **q;
p=&i;
q=&p;
cout << *p << " " << *q << endl;
```

II.1b Zeiger

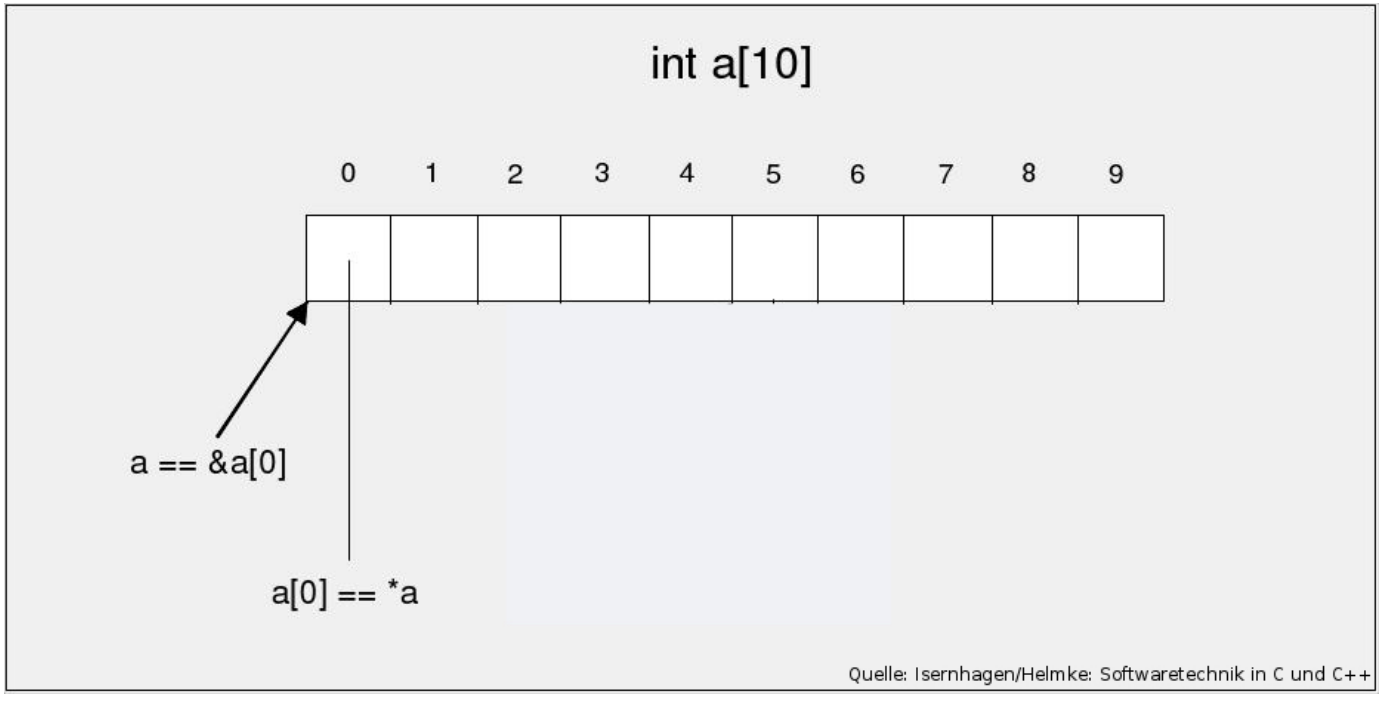
Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.1b Felder (Arrays)

Ein (sehr einfacher) Container

☞ FELDER_ALSZEIGER



Felder

```
int a[10];          // reserviert einen zusammenhaengenden
                    // Speicherbereich fuer 10 integer-Werte

// a ist ein (konstanter) Zeiger auf das erste Feldelement
// syntaktisch aequivalenter Typ:
int* const b {a};// hier erfolgt keine Speicherreservierung!

// Zugriff auf Feldelemente mittels des Indexoperators:
a[3] = 5; b[2] = 7;
```

II.1b Felder

Beispiel

☞ Felder_Beispiel

```
double quadratzahlen[10];
for (int i=0; i<6; i++)
 quadratzahlen[i]=i*i;
for (int i=0; i<10; i++)
 cout << "i=" << i << '\t'
      << "quadrat="
      << quadratzahlen[i]
      << endl;
```

```
i=0     quadrat=0
i=1     quadrat=1
i=2     quadrat=4
i=3     quadrat=9
i=4     quadrat=16
i=5     quadrat=25
i=6     quadrat=4.85832e-270
i=7     quadrat=-0.0289426
i=8     quadrat=-3.27494e-42
i=9     quadrat=1.52087e-314
```

Warnung | Unterschiede zu Vektoren

Die (maximale) Feldgröße muss zur Compile-Zeit bekannt sein.

- Felder lassen sich direkt bei der Vereinbarung mit Werten füllen
- Fehlende Initialwerte werden mit 0 vorbelegt
- Eine fehlende Dimensionierung wird aus der Initialisierungsliste festgelegt

```
/* vollstaendige Initialisierung */
int tage[6]={31,28,31,30,27,31};

/* unvollstaendige Initialisierung */
int n[5]={1,2,3};   /* n[3]=0, n[4]=0 */

/* Dimension durch Initialisierung festgelegt */
int tage[]={31,28,31,30,27,31};
```

II.1b Felder

Übergabe von Feldern an Funktionen/Methoden

☞ Felder_Uebergabe

Felder werden als (i.d. Regel nichtkonstante) Zeiger übergeben.
Die Länge des Feldes muss separat mit übergeben werden.

```
void zeigePalette(char *pFarbe, int anzFarben) {
  for(int i=0; i<anzFarben; i++) cout << pFarbe[i];
}

char farbe[3] = {'r','g','b'};
zeigePalette(farbe,3);
```

Schreibrecht auf Feldinhalte

```
void aendereFeld(int *p, int anzZahlen) {
 p[anzZahlen-1] = -1; // call-by-reference fuer die
                      // Feldelemente
}

int zahlen[10] =          {0,1,2,3,4,5,6,7,8,9};
aendereFeld(zahlen); // 0,1,2,3,4,5,6,7,8,-1
```

II.1b Felder

Zeichenkettenkonstanten sind char-Felder

☞ FELDER_CHARFELDER

Zeichenkettenkonstanten sind Felder vom Grundtyp **char**, die stets mit '\0' abschließen.

```
char text[8] = {'S','t','r','i','n','g','!','\0'};
char text[] = "String!";  //Dasselbe wie
// char* text= "String!"; //dieses hier
//! Beachte: text besteht wegen des abschließenden
//! '\0' aus 8 Zeichen

// "Hallo"[1] ist das Zeichen 'a'
// "Hallo"[5] ist das Element '\0'
```

Lesehilfe

Deklarationen beim Variablennamen beginnend
von rechts nach links lesen

```
// konstante Zeiger
char s[20];             // s ist konstanter Zeiger
char* const t = s;      // t ist konstanter Zeiger
s++; t++;               // geht beides nicht

// konstanter Inhalt
const char* u = "Halli";
u[3]=s[3];              // geht nicht
u = s;                  // geht!

// konstanter Zeiger auf konstanten Inhalt
const char* const v = "Hallo";
v[3]=s[3];              // geht nicht
v = s;                  // geht auch nicht mehr
```

Übergabe von Programmparametern

```
#include <iostream>
using namespace std;
int main(int nParameter, char* parameter[]){
 //            ^                   ^-- Feld von Zeigern auf char
 //            |                       Jeder Zeiger zeigt auf
 //            |                       den Anfang einer
 //            |                       Zeichenkettenkonstante
 //            |                       (=uebergebener Parameter)
 //           -- Anzahl Parameter
 //              (stets mindestens 1, da der Dateiname des
 //               ausfuehrenden Programms ebenfalls als
 //               Parameter gezaehlt wird)

 // Auslesen und Ausgabe der Parameter
 for(int i=0; i<nParameter; ++i) cout << parameter[i] << " ";
 cout << endl;

 //Variante mit Zeigerarithmetik --> spaeter
 do { cout << *parameter << " "; } while(NULL != *++parameter);
 cout << endl;

 return 0;
}
```

Wird ein Feld von Objekten angelegt, muss die zugehörige Klasse einen parameterlosen (Standard-)Konstruktor besitzen

```
{
 class K {};
 K feldVonObjekten[5]; // 5 Konstruktoraufrufe
} // 5 Destruktoraufrufe
```

II.1b Felder

Zeiger auf Felder

Felder sind Datentypen C++
Deshalb kann man auch Zeiger vereinbaren, deren Grundtyp ein Feld ist

```
int feld[10];                    // Ein Feld von 10 int-Werten
int (*pfeld)[10] = &feld; // Zeiger auf feld
   //! Die runde Klammer ist hier notwendig
```

Auf die Klammern kommt es an

```
const int n=10;
T *p;        // Zeiger auf T-Objekte
T (*p)[n];   // Zeiger auf ein Feld von n T-Objekten
T *p[n];     // Feld von n Zeigern auf T-Objekten
```

Algorithmus zur Entschlüsselung von Zeigerdeklarationen

1. Beginn ist der Variablenname
2. Interpretation von links nach rechts
3. Bei Erreichen einer schließenden runden Klammer wird von rechts nach links interpretiert
4. Bei Erreichen einer öffnenden runden Klammer „von rechts“ wird wieder von links nach rechts interpretiert, beginnend nach der zugehörigen geschlossenen runden Klammer
5. Ist das Ende des Ausdrucks erreicht, werden die noch verbliebenen Teile von rechts nach links interpretiert

II.1c Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.1c Zeigerarithmetik

Zeiger sind flexibel

Zeigerarithmetik dient der typgerechten Verschiebung von Zeigerwerten und stellt ein effizientes Werkzeug zur Datenmanipulation im Speicher dar

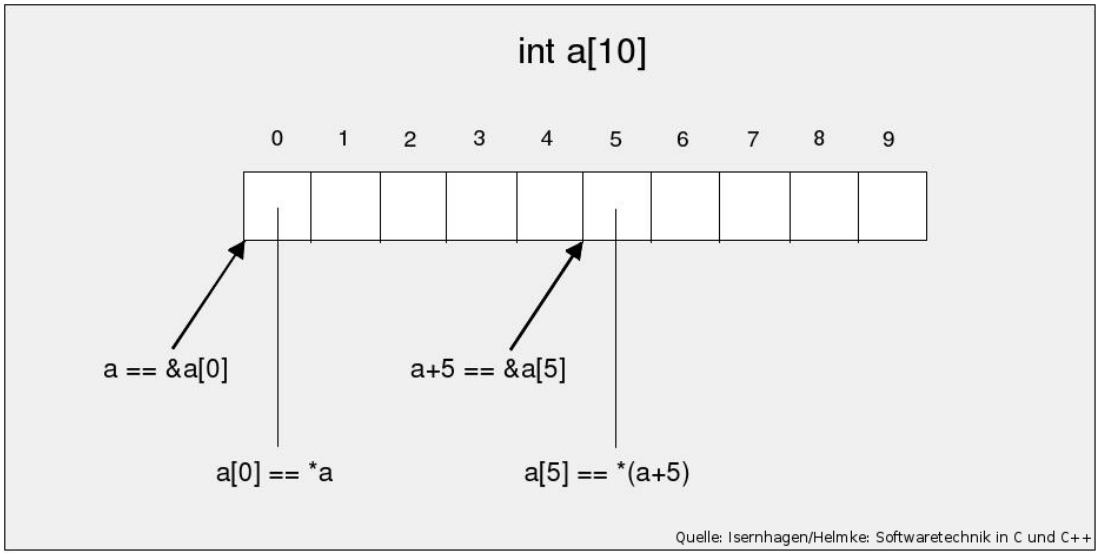
```
T *p; int n;
```

- p++, ++p, p−−, −−p erhöht/erniedrigt die Adresse, auf die p zeigt, um **sizeof**(T) Bytes.
- p+n, p−n erhöht/erniedrigt die Adresse, auf die p zeigt, um n∗**sizeof**(T) Bytes.
- Ist q ein weiterer Zeiger vom selben Grundtyp (T) wie p, so gilt: p−q ist die Anzahl von Speicherplätzen vom Typ T, die zwischen p und q liegen. Der Abstand wird also in Einheiten von **sizeof**(T) gemessen.

II.1c Zeigerarithmetik

Austauschbarkeit von Zeigern und Feldnamen beim Elementzugriff

Feldnamen sind ja selbst Zeiger...




```
int a[10]; int *pa=a;

// Feld als Zeiger
a[i];    // ist dasselbe wie *(a+i); oder *(pa+i);

// Zeiger als Feld
pa[i];   // ist dasselbe wie a[i]
```

ACHTUNG

In Ausdrücken sind a und pa austauschbar. Tatsächlich ist a jedoch ein konstanter Zeiger, d.h. unlösbar mit der Adresse &a[0] verbunden.

Elementzugriff mittels Zeigerarithmetik

```
int i, *pa,
    a[10]={0,1,2,3,4,5,6,7,8,9};

pa=a;
pa++;    /* zeigt nun auf a[1]          */
*pa=-1; /* Element geaendert           */
cout << a[5] << "␣" << *(a+5) << "␣"
     << *(pa+5) << "␣" << pa[5]) << endl;
        /* Ausgabe: 5 5 6 6            */

pa=a;    /* Zeiger zuruecksetzen        */
for (int i=0; i<10; i++)
 cout << *(pa++) << "␣";
       /* Ausgabe: 0 -1 2 3 .... 9 */
```

Bestimmung der Länge einer Zeichenkettenkonstante

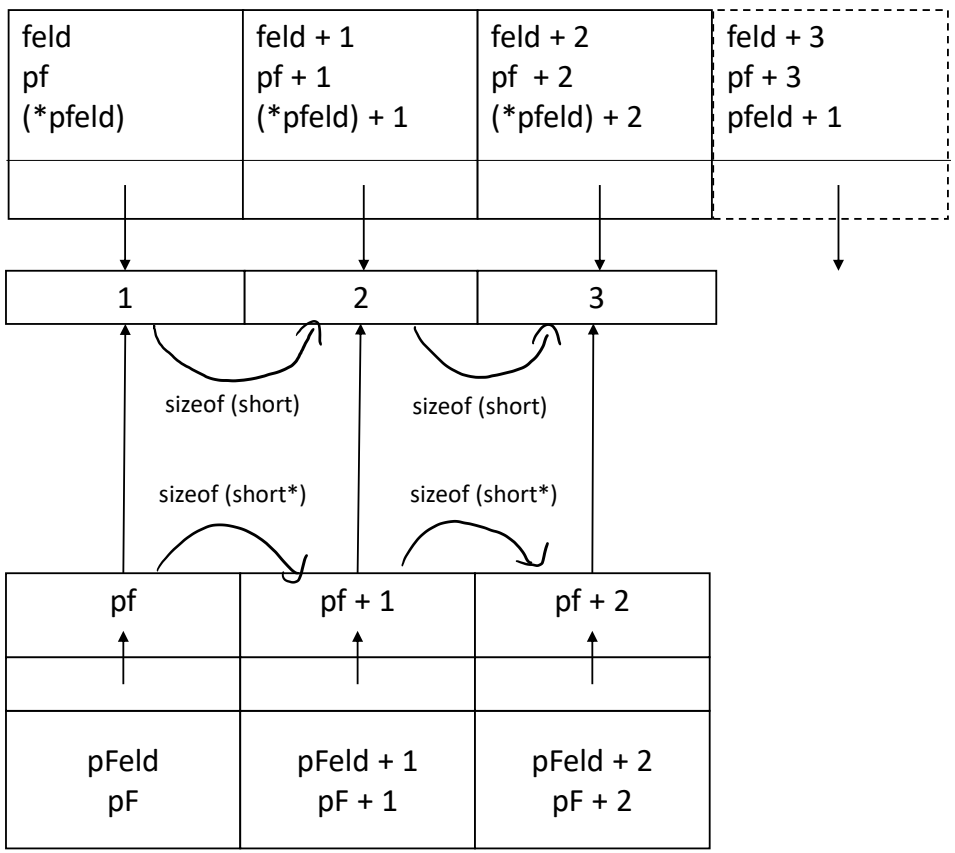
```
#include <iostream>
using namespace std;

int laenge(char *str){ //str zeigt auf 'H' im Beispiel
 char *p_str=str;                    //p_str jetzt auch
 while(*p_str!='\0') {
  p_str++;
 }              //p_str zeigt nun auf '\0' im Beispiel
 return p_str-str;            //Differenz zweier Zeiger
}

int main() {
 char* text="Hallo";
 int textLaenge=laenge(text);
 cout << textLaenge << endl;

 return 0;
}
```

II.1c Zeigerarithmetik

Übung: Zeiger auf Felder

☞ ZEIGERARITHMETIK_ZEIGER-AUF-FELDER

```
const int n=3;
//
short feld[n] = {1,2,3};     // Feld von n short-Werten
short *pf = feld;            // Zeiger auf ersten short-Wert
short (*pfeld)[n] = &feld;   // Zeiger auf feld
//
short *pFeld[n] = {pf, pf+1, pf+2};// Feld von short-Zeigern
short **pF = pFeld;          // Zeiger auf ersten short-Zeiger
```



II.1c Zeigerarithmetik

kleine Übung zur Auswertungsrangfolge

Zeigerverschiebung und Dereferenzierung

Prio	Operator	Ass.	Bemerkung
1	++ −− () [] −> .	l	postfix
2	++ −− ! ~ + − ∗ & **sizeof** (Typ)	r	präfix, unär

Wenn p eine Zeigervariable bezeichnet, dann ist...

- ∗p++=∗(p++), d.h p wird inkrementiert, nachdem (postfix!) der dereferenzierte Inhalt von p geliefert wurde.
- (∗p)++der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.
- ∗++p=∗(++p) (rechts-assoziativ), d.h. der Zeiger p wird inkrementiert und der Inhalt der neuen Adresse wird ausgelesen.
- ++∗p=++(∗p) (rechts-assoziativ), d.h. der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.

II.1c Zeigerarithmetik

kleine Übung zur Auswertungsrangfolge

Zeigerverschiebung und Dereferenzierung

Prio	Operator	Ass.	Bemerkung
1	++ −− () [] −> .	l	postfix
2	++ −− ! ~ + − ∗ & **sizeof** (Typ)	r	präfix, unär

Wenn p eine Zeigervariable bezeichnet, dann ist...

- ∗p++=∗(p++), d.h p wird inkrementiert, nachdem (postfix!) der dereferenzierte Inhalt von p geliefert wurde.
- (∗p)++der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.
- ∗++p=∗(++p) (rechts-assoziativ), d.h. der Zeiger p wird inkrementiert und der Inhalt der neuen Adresse wird ausgelesen.
- ++∗p=++(∗p) (rechts-assoziativ), d.h. der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.

II.1c Zeigerarithmetik

kleine Übung zur Auswertungsrangfolge

Zeigerverschiebung und Dereferenzierung

Prio	Operator	Ass.	Bemerkung
1	++ −− () [] −> .	l	postfix
2	++ −− ! ~ + − ∗ & **sizeof** (Typ)	r	präfix, unär

Wenn p eine Zeigervariable bezeichnet, dann ist...

- ∗p++=∗(p++), d.h p wird inkrementiert, nachdem (postfix!) der dereferenzierte Inhalt von p geliefert wurde.
- (∗p)++der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.
- ∗++p=∗(++p) (rechts-assoziativ), d.h. der Zeiger p wird inkrementiert und der Inhalt der neuen Adresse wird ausgelesen.
- ++∗p=++(∗p) (rechts-assoziativ), d.h. der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.

II.1c Zeigerarithmetik

kleine Übung zur Auswertungsrangfolge

Zeigerverschiebung und Dereferenzierung

Prio	Operator	Ass.	Bemerkung
1	++ −− () [] −> .	l	postfix
2	++ −− ! ~ + − ∗ & **sizeof** (Typ)	r	präfix, unär

Wenn p eine Zeigervariable bezeichnet, dann ist...

- ∗p++=∗(p++), d.h p wird inkrementiert, nachdem (postfix!) der dereferenzierte Inhalt von p geliefert wurde.
- (∗p)++der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.
- ∗++p=∗(++p) (rechts-assoziativ), d.h. der Zeiger p wird inkrementiert und der Inhalt der neuen Adresse wird ausgelesen.
- ++∗p=++(∗p) (rechts-assoziativ), d.h. der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.

II.1c Zeigerarithmetik

kleine Übung zur Auswertungsrangfolge

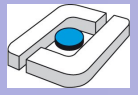
Zeigerverschiebung und Dereferenzierung

Prio	Operator	Ass.	Bemerkung
1	++ −− () [] −> .	l	postfix
2	++ −− ! ~ + − ∗ & **sizeof** (Typ)	r	präfix, unär

Wenn p eine Zeigervariable bezeichnet, dann ist...

- ∗p++=∗(p++), d.h p wird inkrementiert, nachdem (postfix!) der dereferenzierte Inhalt von p geliefert wurde.
- (∗p)++der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.
- ∗++p=∗(++p) (rechts-assoziativ), d.h. der Zeiger p wird inkrementiert und der Inhalt der neuen Adresse wird ausgelesen.
- ++∗p=++(∗p) (rechts-assoziativ), d.h. der Inhalt von p wird ausgelesen und inkrementiert. Der Zeiger p bleibt unverändert.

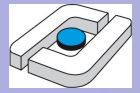
Fazit: Klammern setzen



II.1d Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder



II.1d Zeiger vs Felder

Vergleich

- Felder sind konstante Zeiger auf das erste Feldelement und der Compiler reserviert bei der Vereinbarung automatisch einen zusammenhängenden Speicherbereich für das gesamte Feld
- Der Zugriff auf Feldelemente erfolgt entweder über den []-Operator oder nach den Regeln der Zeigerarithmetik (und anschließender Dereferenzierung)
- Die Zeigerarithmetik ist syntaktisch identisch mit der Nutzung von Random-Access Iteratoren für Container
- Bei der Übergabe von Feldern an Funktionen, ist der Formalparameter i.d.R. ein nicht-konstanter Zeiger vom kompatiblen Grundtyp. Damit kann mittels Zeigerarithmetik an beliebige Stellen im Feld gesprungen werden.
  Die Information über die Länge geht dabei verloren
  (und muss ggf. als weiterer Parameter übergeben werden)

II.1d Zeiger vs Referenzen

Vergleich

- Eine Referenz ist ein Zeiger, der an einen existierenden Wert gebunden ist und diese Verbindung ist nicht lösbar
- Zeiger können überall hin zeigen und sind nicht an einen festen Wert gebunden (Iteratorenfunktionalität durch Zeigerarithmetik)
- Zeiger sind flexibler als Referenzen, aber auch fehleranfälliger, da der Programmierer verantwortlich dafür ist, dass die Zeigervariable auf einen definierten Wert zeigt
- Referenzen sind sicherer als Zeiger und syntaktisch angenehmer, da eine Referenz automatisch auf ihren Wert dereferenziert wird (kein Voranstellen von $*$ notwendig)

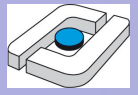
Referenzen sind Zeigern stets vorzuziehen, wenn die zusätzliche Funktionalität der Zeigern nicht benötigt wird

II.1d Zeiger vs Referenzen

Beispiel: Call-by-reference

☞ Zeiger_vs_Referenz

```
void tauschMitZeigern(int* links, int* rechts) {
 int hilf=*links;      // ^-- Call-by-reference
 *links=*rechts;       // bei jedem Wertzugriff muss
 *rechts=hilf;         // explizit dereferenziert werden
}

void tauschMitReferenzen(int& links, int& rechts) {
 int hilf=links;       //     ^-- Call-by-reference
 links=rechts;         // automatische Dereferenzierung
 rechts=hilf;
}

int main() {
 int a=5, b=6;
 tauschMitZeigern(&a, &b); // Es muessen zwingend die
                           // Adressen uebergeben werden
                           // a=6 b=5

 tauschMitReferenzen(a,b); // Werte werden automatisch
                           // an Formalparameter gebunden
                           //    ^--Referenzen, s.o.
                           // a=5 b=6
 return 0;
}
```



II.1e Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

Anwendung | Zeiger als Elemente von Listen/Containern

Ein Onlineshop verwaltet seine Artikel, sowie Listen mit Suchergebnissen von Nutzern (in der Praxis: viele Listen). Die Suchergebnisse sollen intern nach Lieferzeit sortiert werden. Um nicht alle gefundenen Artikel (von sämtlichen Listen) gleichzeitig im Speicher vorhalten zu müssen, werden in den Suchergebnissen nur Verweise (Zeiger) auf die Artikel gespeichert
Nutzen und Gefahren:

- ursprüngliche Objekte müssen nur einmal im Speicher gehalten werden
- Über die Zeiger lassen sich die originalen Objekte verändern

Die Artikelklasse im Online-Shop

```
typedef unsigned int uint;
class Artikel {
private:
 uint nr {0};
 string beschreibung {""};
 float preis {0.};
 uint lieferzeit {0};
public:
 Artikel() {}
 Artikel(uint _nr, const string& text, float _preis,
         uint _lieferzeit);

 unsigned int gibNr()                              { return nr;}
 void setzeNr(unsigned int val)                     { nr = val;}
 string gibBeschreibung()                  { return beschreibung;}
 void setzeBeschreibung(string val)     { beschreibung = val;}
 float gibPreis()                                  { return preis;}
 void setzePreis(float val)                         { preis = val;}
 unsigned int gibLieferzeit()              { return lieferzeit;}
 void setzeLieferzeit(unsigned int val) { lieferzeit = val;}

 const string toString();
};
```

```
const string Artikel::toString() {
  ostringstream strom;
  strom << nr << ": " << beschreibung
  << ", " << preis << "Euro, Lieferzeit "
  << lieferzeit << "Tage";
  return strom.str();
}
```

OnlineShop I

```
// Feld mit Artikeln aufbauen
const int anzGesamt=10;     // ein sehr kleiner OnlineShop...
Artikel alleArtikel[anzGesamt] = {
 Artikel(1,"Scherzkeks", 5.99, 3),
 Artikel(2,"Kruemelmonsters Kekse", 3.99, 4),
 Artikel(3,"Kruemelmonster, klein", 7.99, 1),
 Artikel(4,"\'Junge\', von den Aerzten", 9.99, 2),
 Artikel(5,"Lah-Le-Lu-CD", 12.99, 1),
 Artikel(6,"Quietsche-Entchen", 2.99, 5),
 Artikel(7,"Quietsche-Entchen, Sonderedition rot-metallic", 3.99,
 Artikel(8,"\'Junge\' von Heino", 5.99, 3),
 Artikel(9,"Plagiate leicht gemacht", 10.99, 2),
 Artikel(10,"Ernie-Puppe", 11.99, 1)
};
```

OnlineShop II

```
// zufaellige Suchergebnisse generieren, dabei sollen die
// Artikel nicht kopiert werden
cout << "Suchergebnisse:" << endl;
int anzGefunden=7;                    // (Beispiel)
srand(static_cast<unsigned int>(time(0)));
list<Artikel*> suchergebnisse; // Liste speichert nur Verweise!
for (int i=0; i<anzGefunden; i++)
 suchergebnisse.push_back(&alleArtikel[rand()%anzGesamt]);
                          //  ^-- Speicheradresse des Artikels

// Ausgabe der Suchergebnisse
list<Artikel*>::iterator it;  // "Zeiger" auf Listenelemente
for (it=suchergebnisse.begin(); it!=suchergebnisse.end(); it++)
 cout << (*(*it)).toString() << endl;
 //        ^ ^-- Elementzugriff im Container mittels Iterator
 //        |     Ergebnis ist vom Typ Artikel*
 //        |
 //       -- Dereferenziert den Zeiger;
 //          Ergebnis ist vom Typ Artikel
```

OnlineShop III

```
// binaeres Praedikat zum Vergleich zweier
// Artikel-Zeiger
class LieferzeitKleiner {
public:
 bool operator()(Artikel* links, Artikel* rechts) {
  return links->gibLieferzeit() < (*rechts).gibLieferzeit();
  //           ^-- Abkuerzung fuer --^
 }
};

// Sortieren der Ergebnisliste
// --> Es werden nur die Zeiger sortiert, nicht die
//     Artikelobjekte selbst!
cout << endl << "Nach Lieferzeit sortiert:" << endl;
suchergebnisse.sort(LieferzeitKleiner());
for (it=suchergebnisse.begin(); it!=suchergebnisse.end(); it++)
 cout << (*it)->toString() << endl;
 //       ^   ^-- auch bei Zeigern gibt es
 //       |       die Komponentenselektion mittels ->
 //       -- Beachte: *it hat den Typ Artikel*
 //          Artikel* ist ein Zeiger, kein Klassenobjekt,
 //          daher ist it-> hier syntaktisch falsch
```

II.2 Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.2a Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.2a new und delete

Wie bei Feldern kann man für beliebige Datentypen in einem Schritt Speicher anfordern, initialisieren und einen Zeiger auf die Anfangsadresse erhalten

Anforderung von Speicher (statt `int` kann ein beliebiger Datentyp eingesetzt werden)	
`int* pi;` `pi=new int;`	legt Speicher für eine `int`-Variable an und liefert dessen Speicheradresse zurück
`int* pi;` `pi=new int(5);`	wie oben; Wert wird mit 5 initialisiert / Konstruktoraufruf
`int* pfeld;` `pfeld=new int[10];`	zusammenhängender Speicher für 10 int-Variablen; 10 Standardkonstruktoraufrufe; liefert Anfangsadresse zurück

Freigabe von *zuvor reserviertem* Speicher	
`delete pi;`	Destruktoraufruf für `*pi`
`delete[] pfeld;`	Destruktoraufruf für alle Feldelemente

- Ohne die (korrekte) Freigabe entstehen Speicherlecks!
- Die Lebensdauer von mit `new` erzeugten Objekten besteht bis zur Speicherfreigabe mit `delete`. Es ist darauf zu achten, dass der zugehörige Zeiger noch bis zu diesem Zeitpunkt 'lebt'
- `delete` auf einen Nullzeiger bewirkt nichts

II.2a new und delete

☞ HEAP_NEWDELETE

```
class K {
 int n;
public:
 K()             : n(0)    { cout << "K " ; }
 K(int _n)       : n(_n)   { cout << "K(" << n <<") " ; }
 K(const K& k) : n(k.n) { cout << "KK(" << n <<") " ; }
 ~K()                      { cout << "D(" << n <<") " ; }
};

{
 // Speicher anfordern...
 K* pK_standard=new K;              // -->K
 {
  K* pK_anweisungsblock=new K;// -->K
 }
 K* pK_zwei=new K(2);               // -->K(2)
 K* pK_feld=new K[5];               // -->K K K K K
 vector<K> vK(5);                   // -->K K K K K

 // Speicher freigeben
 delete pK_standard;                // -->D(0)
 //delete pK_anweisungsblock; // -->Speicherleck!!!
 delete pK_zwei;                    // -->D(2)
 delete [] pK_feld;                 // -->D(0) D(0) D(0) D(0) D(0)
 //      ^--  !!!
}                                   // -->D(0) D(0) D(0) D(0) D(0)
```

II.2a new und delete

Übung: Zeiger-Spielerei

☞ HEAP_ZEIGERSPIELEREI

Was passiert hier? / Ausgabe?

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main() {
6   int *x=new int; int *y=new int; int *z=new int;
7
8   *x=1; *y=2; *z=3;
9   cout << setw(2) << *x << *y << *z << endl;
10
11  *z=*x;
12  cout << setw(2) << *x << *y << *z << endl;
13
14  y=x;
15  cout << setw(2) << *x << *y << *z << endl;
16
17  *x=4; *y=5; *z=6;
18  cout << setw(2) << *x << *y << *z << endl;
19
20  delete x; delete y; delete z;
21  return 0;
22 }
```

II.2a new und delete

Übung: dynamische Felderzeugung

Korrigieren Sie das Programm

```
1  //
2  // Erzeugung eines dynamischen Feld mit 10 Elementen
3  //
4
5  void newArray(int *p, int length) {
6   p=new int[length];
7  }
8
9  int main() {
10  int *p {0};
11  newArray(p,10);
12  p[0]=2; p[1]=5; //...und so weiter
13
14  return 0;
15 }
```

II.2b Zeiger

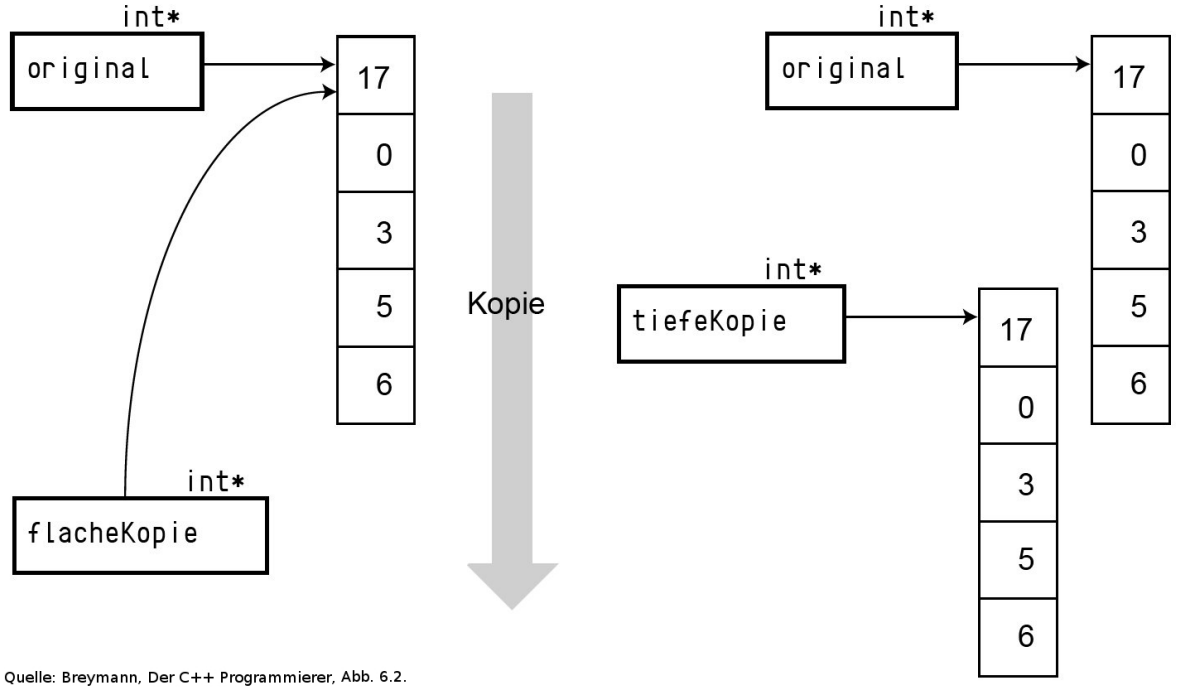
Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.2b flache und tiefe Kopie

☞ LIVE_TIEFE-KOPIE

- Eine flache Kopie kopiert sämtliche Datenkomponenten einer Klasse. Enthält die Klasse Zeiger, so werden lediglich der Zeiger kopiert
- Eine tiefe Kopie legt eine echte Inhaltskopie an, d.h. bei Bedarf wird passender Speicher angefordert und auch die Zeigerziele werden kopiert



Quelle: Breymann, Der C++ Programmierer, Abb. 6.2.

Beispiel: flache und tiefe Kopie eines durch einen Zeiger referenzierten Feldes

II.2c Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.2c Anwendung: Favoritenliste

☞ Favoriten(Liste)

Anwendung | tiefe Kopie einer Liste von Verweisen

Gespeicherte Artikel werden vom Nutzer lokal in einer Klasse FavoritenListe als Verweise gespeichert.

- Alle Speichervorgänge darin sollen tief sein
- die Liste soll kopiert und zugewiesen werden können

Favoriten

```
// Feld mit Artikeln aufbauen
const int anzGesamt=10; //! Feld von Artikel-Zeigern
Artikel* alleArtikel[anzGesamt] = {
 new Artikel(1,"Scherzkeks", 5.99, 3),
 new Artikel(2,"Suesse Schokolade", 3.99, 4),
 new Artikel(3,"Alles wird gut", 7.99, 1),
 new Artikel(4,"\'Junge\', von den Aerzten", 9.99, 2),
 new Artikel(5,"Lah-Le-Lu-CD", 12.99, 1),
 new Artikel(6,"Quietsche-Entchen", 2.99, 5),
 new Artikel(7,"Quietsche-Entchen, Sonderedition rot-metallic", 3.
 new Artikel(8,"\'Junge\' von Heino", 5.99, 3),
 new Artikel(9,"Plagiate leicht gemacht", 10.99, 2),
 new Artikel(10,"Ernie-Puppe", 11.99, 1)
};
// Favoritenliste generieren und ausgeben
int anzFav=4;
FavoritenListe favListe;      // erzeugt leere Favoritenliste
for (int i=0; i<anzFav; i++) {
 favListe << alleArtikel[rand()%anzGesamt];
 //!      ^-- Anforderung: Artikelobjekte sollen kopiert
 //!          und als Verweis gespeichert werden
 cout << favListe[i]->toString() << endl;
 //!              |   ^-- Methode der Klasse Artikel(!)
}//!             |-- Selektion eines Artikel-Zeigers
```

FavoritenListe.h — Anforderungskatalog

```
#include <vector>
#include "Artikel.h"
typedef unsigned int uint;
typedef unsigned long ulong;

class FavoritenListe {
 vector<Artikel*> favorit;    // Artikel werden als Verweise
public:                       // im Vektor gespeichert
 FavoritenListe();
 ~FavoritenListe();
                              // tiefe Kopie/Zuweisung
 FavoritenListe(const FavoritenListe& other);
 FavoritenListe& operator=(const FavoritenListe& other);

 // getter
 ulong getAnzahl()              { return favorit.size(); }
 Artikel*& operator[](ulong i) { return favorit[i]; }
 //       ^-- Schreibrecht fuer zurueckgebenen Zeiger,
 //           falls benoetigt

 // Artikel kopieren und als Verweis zur Liste hinzufuegen
 FavoritenListe& operator<< (Artikel* art);
};
```

FavoritenListe.cpp I — Kon/Des

```
// Standardkonstruktor
FavoritenListe::FavoritenListe() {}

// Destruktor --> Speicherfreigabe alle Containerelemente!
FavoritenListe::~FavoritenListe() {
 for (int i=0; i<favorit.size(); i++) delete favorit[i];
 favorit.clear();
}

// Kopierkonstruktor --> tief!
FavoritenListe::FavoritenListe(const FavoritenListe& orig) {
 for (int i=0; i<orig.favorit.size(); i++)
  favorit.push_back(new Artikel(*orig.favorit[i]));
}    // beachte Operatorprio-/assoziativitaeten von *, ., []
```

Prio	Operator	Ass.	Bemerkung
0	::		Bereichsauflösung
1	++ -- () [] -> . x_cast<Typ>() x=static, const, reinterpret, dynamic	l	postfix, unär
2	++ -- ! ~ + - * & sizeof() new delete	r	präfix, unär

FavoritenListe.cpp II — Zuweisung/<<

```
FavoritenListe&
FavoritenListe::operator=(const FavoritenListe& rhs)
{
 if (this == &rhs) return *this;          // Selbstzuweisung
                                          // verhindern

 for (int i=0; i<favorit.size(); i++)     // alten Speicher
  delete favorit[i];                      // freigeben
 favorit.clear();
                                          // tiefe Kopie wie
 for (int i=0; i<rhs.favorit.size(); i++) // im KK
  favorit.push_back(new Artikel(*rhs.favorit[i]));

 return *this;
}

// Artikel zur Liste hinzufuegen
FavoritenListe& FavoritenListe::operator<< (Artikel* art)  {
 favorit.push_back(new Artikel(*art));    // tiefe Kopie

 return *this;
```

II.3 Zeiger

Inhalt

1. Zeiger und Felder
   - Zeigervariablen
   - Felder
   - Zeigerarithmetik
   - Vergleich: Zeiger vs Felder & Referenzen
   - Anwendung: Online-Shop
2. dynamische Speicherverwaltung
   - Speicheranforderung und Speicherfreigabe
   - flache und tiefe Kopie
   - Anwendung: Favoritenliste
3. Mehrdimensionale Felder

II.3 (mehrdimensionale) Felder und Zeiger

	Typ	syntaktisch äquivalenter Zeiger	Parameterliste in Funktionen
1D	T zeile[n]	T * const pZeile T pZeile[]	(T *pZeile, int n) (T pZeile[], int n)
	1D-Felder sind konstante Zeiger auf das erste Element		
2D	T matrix[m][n]	T (* const pMatrix)[n] T pMatrix[][n]	(T (*pMatrix)[n], int m) (T pMatrix[][n], int m)
	2D-Felder sind konstante Zeiger auf 1D-Felder vorgegebener Länge n		
3D	T quader[k][m][n]	T (* const pQuader)[m][n] T pQuader[][m][n]	(T (*pQuader)[m][n], int k) (T pQuader[][m][n], int k)
	3D-Felder sind konstante Zeiger auf $m \times n$ 2D-Felder		

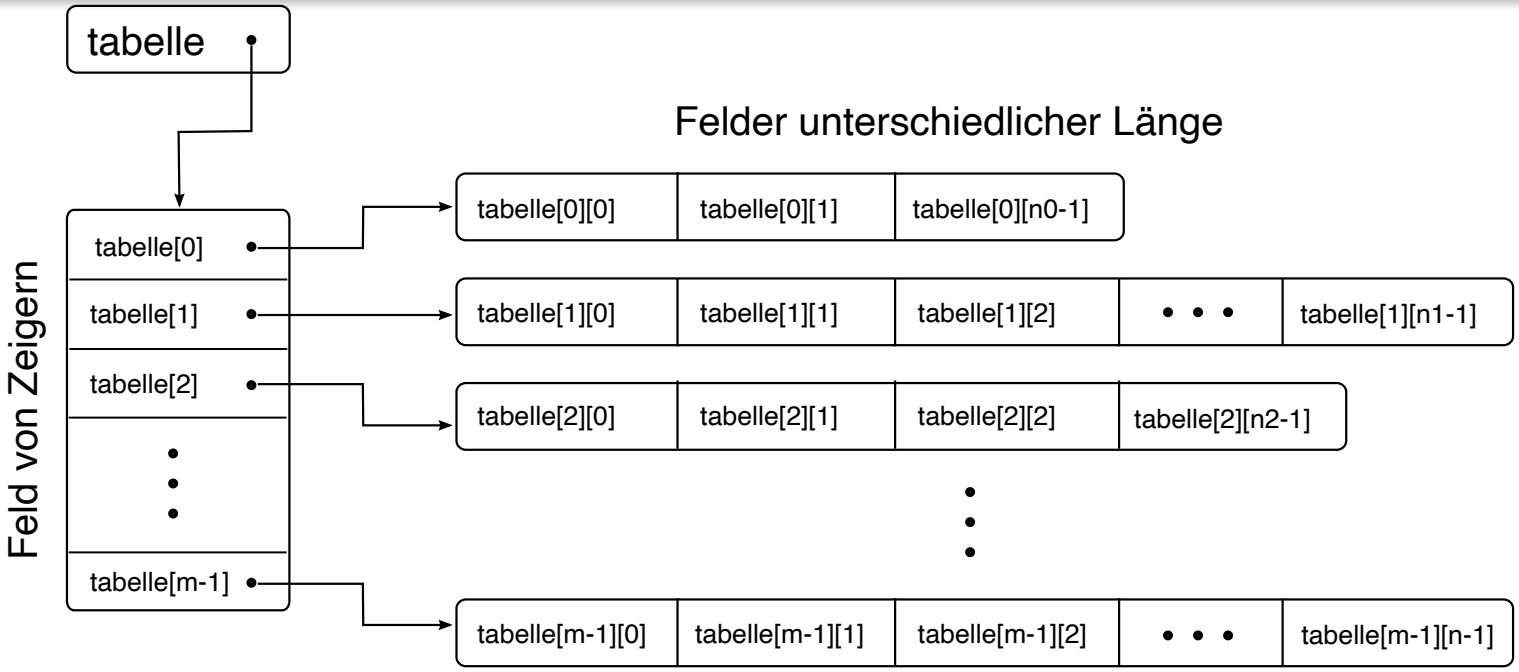
Erinnerung: Klammersetzung		
T *p	ist ein Zeiger auf T-Objekte	Inkrement: **sizeof**(T)
T (*p)[n]	ist ein Zeiger auf ein Feld von n T-Objekten	Inkrement: n***sizeof**(T)
T *p[n]	ist ein Feld von n Zeigern auf T-Objekten	Inkrement: **sizeof**(T*)

```
void gibMatrixAus(short (*pFeld)[N], int m) { ... }
// Matrix auf dem Stack
const int M=4;  // Anzahl Zeilen (zur Compilezeit bekannt!)
const int N=3;  // Anzahl Spalten (zur Compilezeit bekannt!)
short matrix[M][N]; // Vereinbarung des 2D-Feldes
matrix[2][3] = 5;   // Elementzugriff
gibMatrixAus(matrix, M); // Uebergabe an Funktion

// Matrix auf dem Heap
int m=4;  // Anzahl Zeilen (reicht: zur Laufzeit bekannt)
short (*const pMatrix)[N] = new short[m][N]; // dynamisch
pMatrix[2][3] = 5;  // Elementzugriff
gibMatrixAus(pMatrix, m); // Uebergabe an Funktion
...
delete[] pMatrix;       // Speicherfreigabe!
```

Beachte

Die "Zeiger" matrix[i]=*(matrix+i) bzw. pMatrix[i]=*(pMatrix+i) existieren nur logisch und nicht physikalisch im Speicher.
Sie repräsentieren Adressen, die sich mittels Zeigerarithmetik ergeben.

II.3 2D-Felder: variable Zeilenlänge

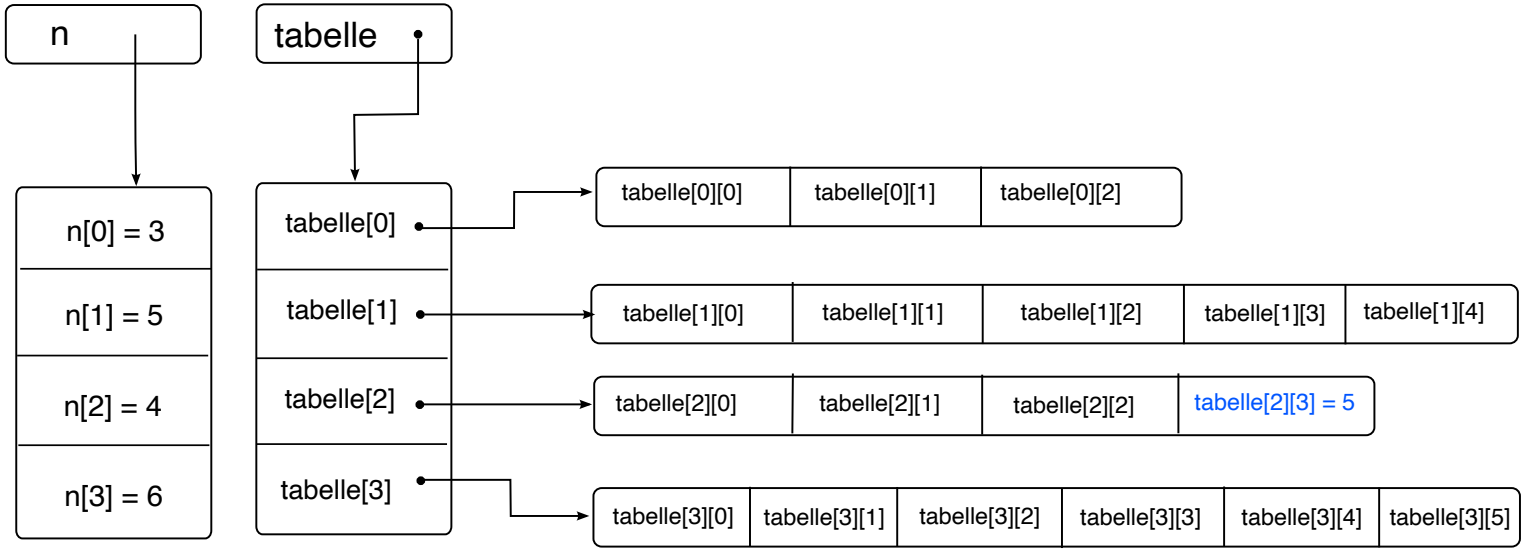


Typ	Parameterliste in Funktionen
T **tabelle	(T** tabelle, **int** m, **int** *n)

Beachte

Die Zeiger tabelle[i] existieren physikalisch im Speicher.
Sie enthalten die Startadressen, für die einzelnen Tabellenzeilen.
Aufeinanderfolgende Tabellenzeilen müssen nicht notwendig hintereinander im Speicher abgelegt sein.

II.3 2D-Felder: variable Zeilenlänge ☞ 2DFELD_VARIABLEZEILEN




```
void gibTabelleAus(short** tabelle, int m,   int* n) { ... }
int m=4;                                  // Anzahl Zeilen
int *n=new int[m];                        // Zur Aufnahme der
n[0]=3; n[1]=5; n[2]=4; n[3]=6;           // Zeilenlaengen
short **const tabelle=new short*[m]; // Speicher fuer Zeigerfeld
for(int i=0;i<m;i++)                      // Fuer jeden short-Zeiger ein
  tabelle[i]=new short[n[i]];             // Feld erzeugen
tabelle[2][3] = 5;                        // Elementzugriff
gibTabelleAus(tabelle,m,n);               // Uebergabe an Funktion
...
// Speicher wieder freigeben in UMGEKEHRTER Reihenfolge
for(int i=0;i<m;i++) delete[] tabelle[i];//short-Felder
delete[] tabelle;                         //Zeigerfeld
delete[] n;                               //Zeilenlaengen
```

II.3 2D-Felder: Zeigerarithmetik

Obwohl bei Feldern manche "Zeiger" nur logisch existieren, unterstützen sie alle die gleiche Zeigerarithmetik, wie für Tabellen.
Lediglich die Adressen der nur logisch vorhandenen Zeiger stimmen in diesem Fall alle mit der Adresse des ersten Matrixeintrages überein

Zeiger	Grundtyp	Bedeutung des Grundtyps
T **A	T*	Feld von Zeigern
T *A[i]	T	i-tes Feld von T-Werten

Ausdruck	Wert	Beschreibung
A	&A[0]	Zeiger auf den Anfang des Zeigerfeldes (A=matrix,pMatrix: Zeiger auf Anfangselement)
A+i	&A[i]	Zeiger auf den i-ten Zeiger im Zeigerfeld (A=matrix,pMatrix: Zeiger auf Anf.elem. i-te Zeile)
*(A+i)	A[i]=&A[i][0]	Zeiger auf den Anfang der i-ten Tabellenzeile
*(A+i)+j	A[i]+j=&A[i][j]	Zeiger auf Element j der i-ten Tabellenzeile
((A+i)+j)	A[i][j]	j-tes Element der i-ten Tabellenzeile