

12. Felder (Arrays) II

Strings, Lindenmayer-Systeme, Mehrdimensionale Felder, Vektoren von Vektoren, Kürzeste Wege, Felder und Vektoren als Funktionsargumente

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

```
char text[] = {'b','o','o','l'}
```

- können auch durch String-Literale definiert werden

```
char text[] = "bool"
```

- können nur mit konstanter Grösse definiert werden

416

417

Texte

- können mit dem Typ `std::string` aus der Standardbibliothek repräsentiert werden.

```
std::string text = "bool";
```

definiert einen String der Länge 4

- Ein String ist im Prinzip ein Feld mit zugrundeliegendem Typ `char`, plus Zusatzfunktionalität
- Benutzung benötigt `#include <string>`

418

Strings: gepimpte char-Felder

Ein `std::string...`

- kennt seine Länge

```
text.length()
```

gibt Länge als `int` zurück (Aufruf einer Mitglieds-Funktion; später in der Vorlesung)

- kann mit variabler Länge initialisiert werden

```
std::string text (n, 'a')
```

text wird mit `n` 'a's gefüllt

- „verstehen“ Vergleiche

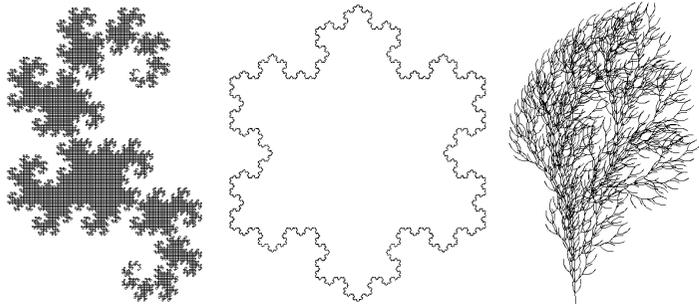
```
if (text1 == text2) ...
```

true wenn `text1` und `text2` übereinstimmen

419

Lindenmayer-Systeme (L-Systeme)

Fraktale aus Strings und Schildkröten



L-Systeme wurden vom ungarischen Biologen Aristid Lindenmayer (1925–1989) zur Modellierung von Pflanzenwachstum erfunden.

Definition und Beispiel

- Alphabet Σ
- Σ^* : alle endlichen Wörter über Σ
- Produktion $P : \Sigma \rightarrow \Sigma^*$
- Startwort $s_0 \in \Sigma^*$

	$\{F, +, -\}$
c	$P(c)$
F	F + F +
+	+
-	-
F	

Definition

Das Tripel $\mathcal{L} = (\Sigma, P, s_0)$ ist ein L-System.

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \dots \in \Sigma^*$:

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

$$w_1 := P(w_0)$$

$$w_1 := F + F +$$

$$w_2 := P(w_1)$$

$$w_2 := F + F + + F + F + +$$

$$P(F)P(+)P(F)P(+)$$

⋮

⋮

Definition

$$P(c_1c_2 \dots c_n) := P(c_1)P(c_2) \dots P(c_n)$$

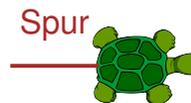
Turtle-Grafik

Schildkröte mit Position und Richtung



Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓



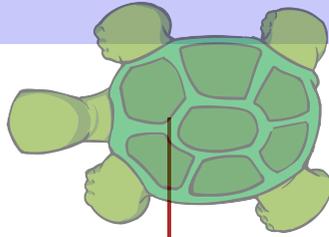
+ : Drehe dich um 90 Grad ✓



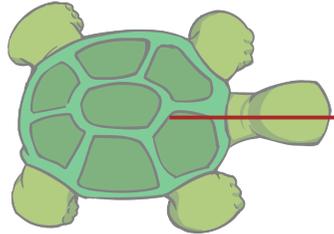
- : Drehe dich um -90 Grad ✓



Wörter zeichnen!



$$w_1 = F + F + \checkmark$$



lindenmayer.cpp:

Wörter $w_0, w_1, w_2, \dots, w_n \in \Sigma^*$:

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;
```

```
std::string w = "F";
```

```
for (unsigned int i = 0; i < n; ++i)
    w = next_word (w);
```

```
draw_word (w);
```

Hauptprogramm

```
std::string
```

$w = w_0 = F$

$w = w_i \rightarrow w = w_{i+1}$

Zeichne $w = w_n$!

424

425

lindenmayer.cpp:

next_word

```
// POST: replaces all symbols in word according to their
//       production and returns the result
std::string next_word (std::string word) {
    std::string next;
    for (unsigned int k = 0; k < word.length(); ++k)
        next += production (word[k]);
    return next;
}

// POST: returns the production of c
std::string production (char c) {
    switch (c) {
        case 'F': return "F+F+";
        default: return std::string (1, c); // trivial production c -> c
    }
}
```

426

lindenmayer.cpp:

draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
    for (unsigned int k = 0; k < word.length(); ++k)
        switch (word[k]) {
            case 'F':
                turtle::forward();
                break;
            case '+':
                turtle::left(90);
                break;
            case '-':
                turtle::right(90);
                break;
        }
}
```

Springe zum case, der word[k] entspricht.

Vorwärts! (Funktion aus unserer Schildkröten-Bibliothek)

Überspringe die folgenden cases

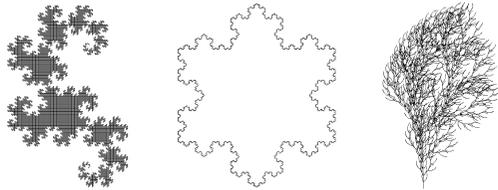
Drehe dich um 90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

Drehe dich um -90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

427

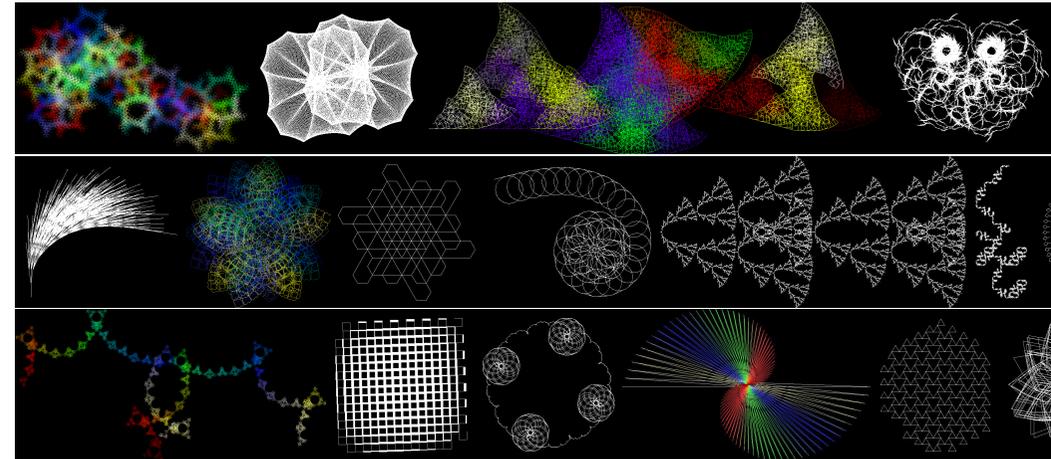
L-Systeme: Erweiterungen

- Beliebige Symbole ohne grafische Interpretation (`dragon.cpp`)
- Beliebige Drehwinkel (`snowflake.cpp`)
- Sichern und Wiederherstellen des Schildkröten-Zustandes → Pflanzen (`bush.cpp`)



L-System-Challenge:

`amazing.cpp!`



428

429

Mehrdimensionale Felder

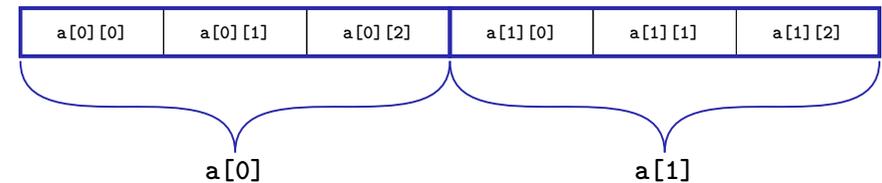
- sind Felder von Feldern
- dienen zum Speichern von *Tabellen, Matrizen,...*

`int a[2][3]`

a hat zwei Elemente, und jedes von ihnen ist ein Feld der Länge 3 mit zugrundeliegendem Typ `int`

Mehrdimensionale Felder

Im Speicher: flach



Im Kopf: Matrix

		Spalten →		
		0	1	2
Zeilen ↓	0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>
	1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>

430

431

Mehrdimensionale Felder

- sind Felder von Feldern von Feldern

$T a[expr_1] \dots [expr_k]$

Konstante Ausdrücke!

a hat $expr_1$ Elemente und jedes von ihnen ist ein Feld mit $expr_2$ Elementen, von denen jedes ein Feld mit $expr_3$ Elementen ist, ...

Mehrdimensionale Felder

Initialisierung:

```
int a[] [3] =
{
    {2, 4, 6}, {1, 3, 5}
}
```

Erste Dimension kann weggelassen werden

2	4	6	1	3	5
---	---	---	---	---	---

432

433

Vektoren von Vektoren

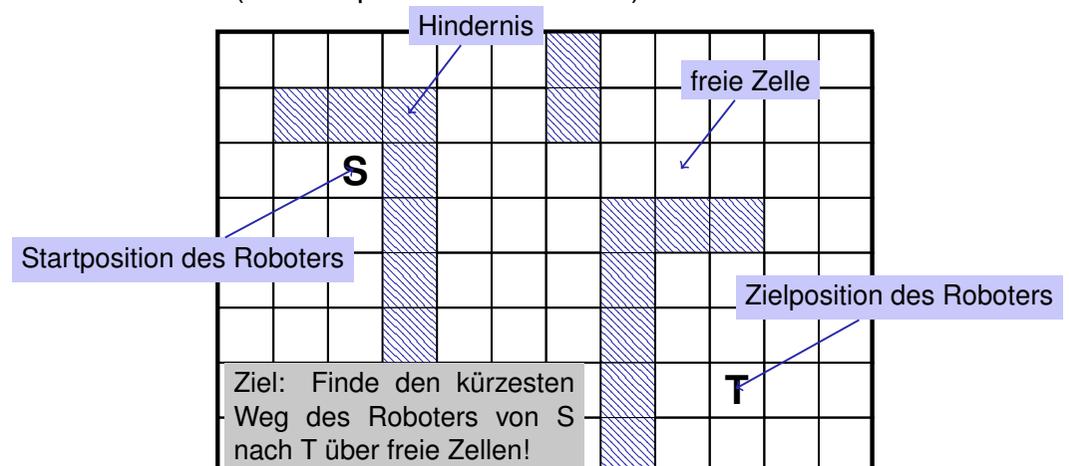
- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?
- Lösung: Vektoren von Vektoren

Beispiel: Vektor der Länge n von Vektoren der Länge m :

```
std::vector<std::vector<int> > a (n,
    std::vector<int>(m));
```

Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)

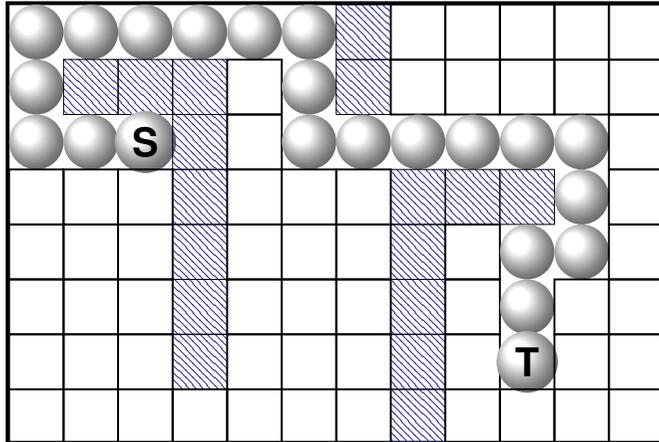


434

435

Anwendung: Kürzeste Wege

Lösung



Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



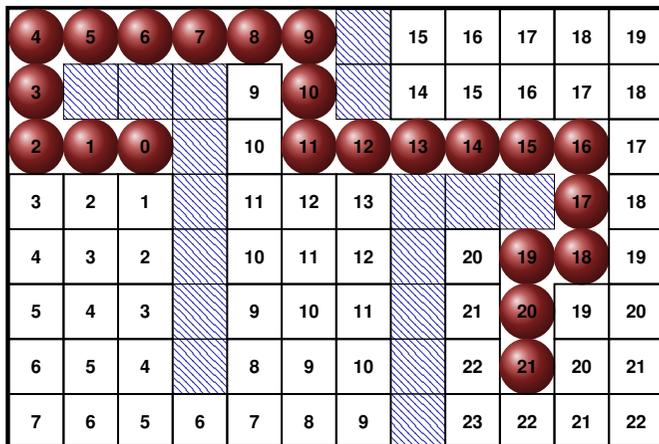
Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

436

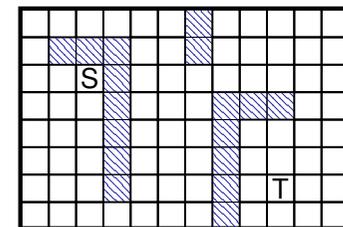
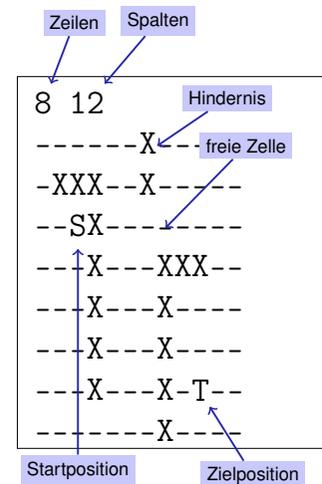
437

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



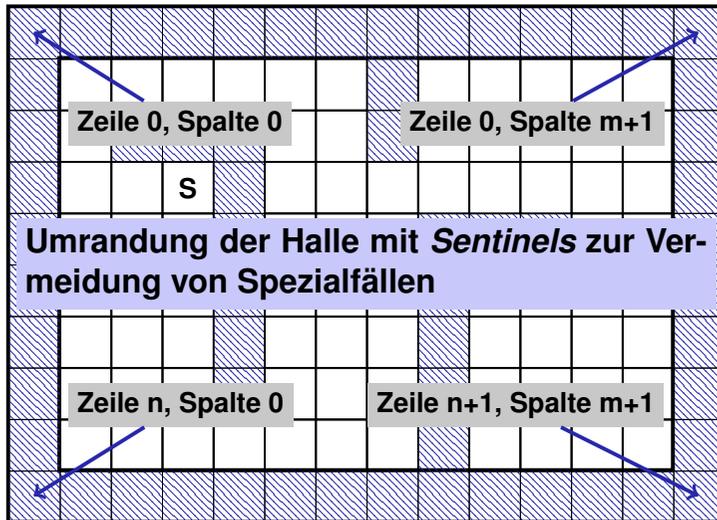
Vorbereitung: Eingabeformat



438

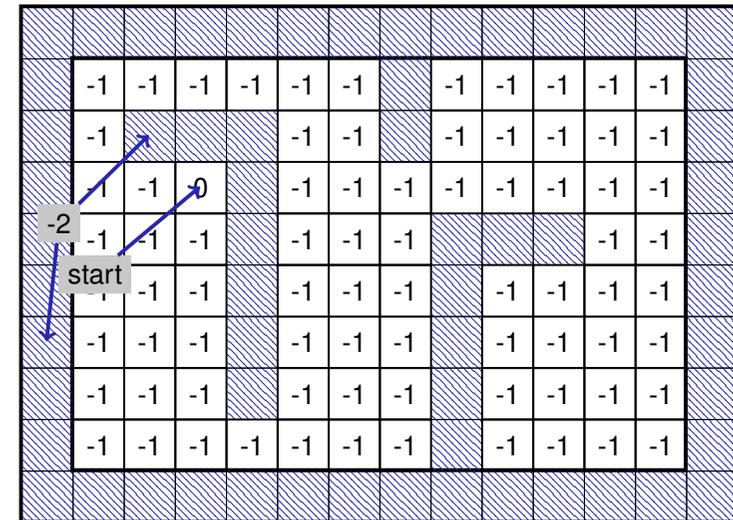
439

Vorbereitung: Wächter (Sentinels)



440

Vorbereitung: Initiale Markierung



441

Das Kürzeste-Wege-Programm

- Einlesen der Dimensionen und Bereitstellung eines zweidimensionalen Feldes für die Weglängen

```
#include<iostream>
#include<vector>

int main()
{
    // read floor dimensions
    int n; std::cin >> n; // number of rows
    int m; std::cin >> m; // number of columns

    // define a two-dimensional
    // array of dimensions
    // (n+2) x (m+2) to hold the floor plus extra walls around
    std::vector<std::vector<int>> > floor (n+2, std::vector<int>(m+2));
```

Wächter (Sentinel)

442

Das Kürzeste-Wege-Programm

- Einlesen der Hallenbelegung und Initialisierung der Längen

```
int tr = 0;
int tc = 0;
for (int r=1; r<n+1; ++r)
    for (int c=1; c<m+1; ++c) {
        char entry = '-';
        std::cin >> entry;
        if (entry == 'S') floor[r][c] = 0;
        else if (entry == 'T') floor[tr = r][tc = c] = -1;
        else if (entry == 'X') floor[r][c] = -2;
        else if (entry == '-') floor[r][c] = -1;
    }
```

444

Das Kürzeste-Wege-Programm

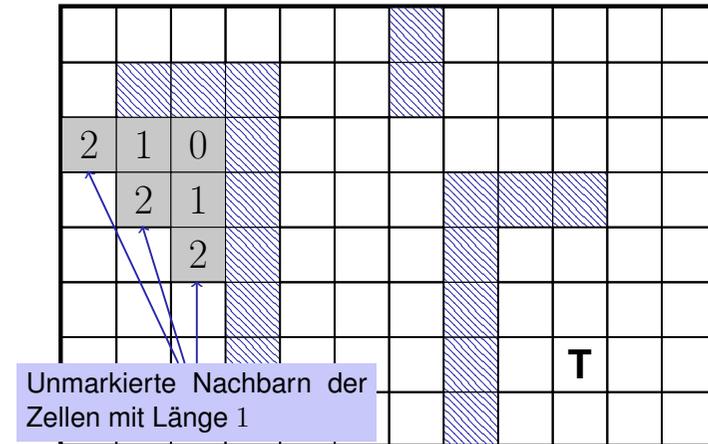
■ Hinzufügen der umschliessenden „Wände“

```
for (int r=0; r<n+2; ++r)
    floor[r][0] = floor[r][m+1] = -2;

for (int c=0; c<m+2; ++c)
    floor[0][c] = floor[n+1][c] = -2;
```

Markierung aller Zellen mit ihren Weglängen

Schritt 2: Alle Zellen mit Weglänge 2



445

446

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {
    bool progress = false;
    for (int r=1; r<n+1; ++r)
        for (int c=1; c<m+1; ++c) {
            if (floor[r][c] != -1) continue;
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1) {
                floor[r][c] = i; // label cell with i
                progress = true;
            }
        }
    if (!progress) break;
}
```

447

Das Kürzeste-Wege-Programm

Markieren des kürzesten Weges durch „Rückwärtslaufen“ vom Ziel zum Start

```
int r = tr; int c = tc;
while (floor[r][c] > 0) {
    const int d = floor[r][c] - 1;
    floor[r][c] = -3;
    if (floor[r-1][c] == d) --r;
    else if (floor[r+1][c] == d) ++r;
    else if (floor[r][c-1] == d) --c;
    else ++c; // (floor[r][c+1] == d)
}
```

448

Felder als Funktionsargumente

Das geht auch für mehrdimensionale Felder.

```
void print_matrix(const int (&m)[3][3]) {
    for (int i = 0; i < 3; ++i) {
        print_vector(m[i]);
        std::cout << "\n";
    }
}
```

453

Vektoren als Funktionsargumente

Vektoren können *per value*, aber auch als *Referenz*-Argumente an eine Funktion übergeben werden.

```
void print_vector(const std::vector<int>& v) {
    for (int i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
}
```

Hier: *Call by Reference* ist effizienter, weil der Vektor sehr lang sein kann.

454

Vektoren als Funktionsargumente

Das geht auch für mehrdimensionale Vektoren.

```
void print_matrix(const std::vector<std::vector<int>>& m) {
    for (int i = 0; i < m.size(); ++i) {
        print_vector(m[i]);
        std::cout << "\n";
    }
}
```

455

13. Zeiger, Algorithmen, Iteratoren und Container I

Zeiger, Address- und Dereferenzenoperator,
Feld-nach-Zeiger-Konversion

456

Komische Dinge...

```
#include<iostream>
#include<algorithm>

int main(){
    int a[] = {3, 2, 1, 5, 4, 6, 7};

    // gib das kleinste Element in a aus
    std::cout << *std::min_element (a, a + 7);

    return 0;
}
```

Dafür müssen wir zuerst *Zeiger* verstehen!

457

Referenzen: Wo ist Anakin?

“Suche nach Vader, und Anakin finden du wirst.”

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker;
darth_vader = 22;

// anakin_skywalker = 22
```



458

Zeiger: Wo ist Anakin?

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
std::cout << here; // Adresse
*here = 22;

// anakin_skywalker = 22
```

“Anakins Adresse ist 0x7fff6bdd1b54.”



459

Swap mit Zeigern

```
void swap(int* x, int* y){
    int t = *x;
    *x = *y;
    *y = t;
}

...
int a = 2;
int b = 1;
swap(&a, &b);
std::cout << "a= " << a << "\n"; // 1
std::cout << "b = " << b << "\n"; // 2
```

460

Zeiger Typen

T* Zeiger-Typ zum zugrunde liegenden Typ T.

Ein Ausdruck vom Typ T* heisst *Zeiger* (auf T).

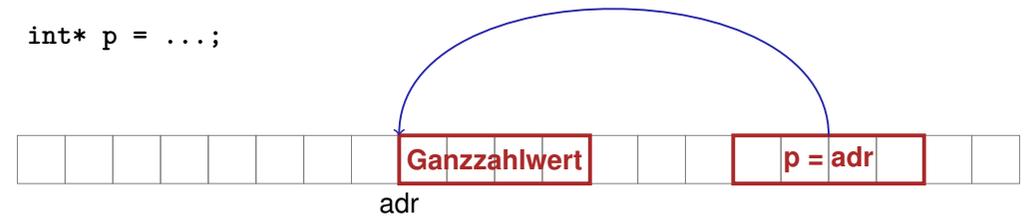
Zeiger Typen

Wert eines Zeigers auf T ist die Adresse eines Objektes vom Typ T.

Beispiele

```
int* p; Variable p ist Zeiger auf ein int.  
float* q; Variable q ist Zeiger auf ein float.
```

```
int* p = ...;
```



461

462

Adress-Operator

Der Ausdruck

L-Wert vom Typ T



& lval

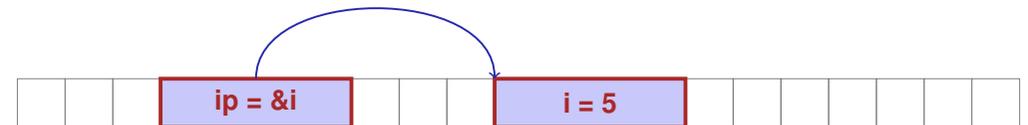
liefert als R-Wert einen *Zeiger* vom Typ T* auf das Objekt an der Adresse von *lval*

Der Operator *&* heisst *Adress-Operator*.

Adress-Operator

Beispiel

```
int i = 5;  
int* ip = &i; // ip initialisiert  
// mit Adresse von i.
```

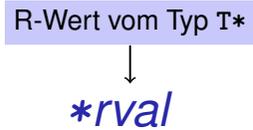


463

464

Dereferenz-Operator

Der Ausdruck



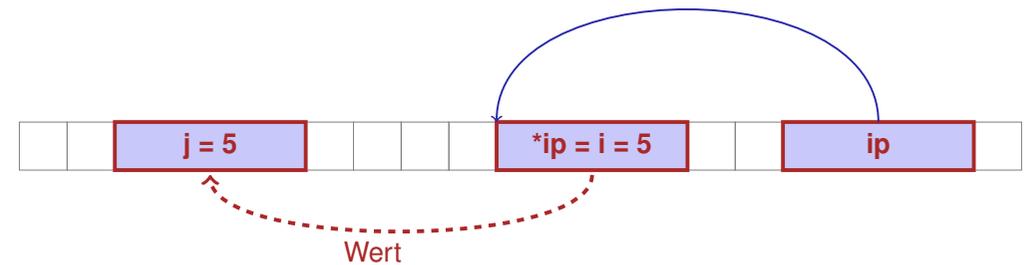
liefert als L-Wert den Wert des Objekts an der durch *rval* repräsentierten Adresse

Der Operator *** heisst Dereferenz-Operator.

Dereferenz-Operator

Beispiel

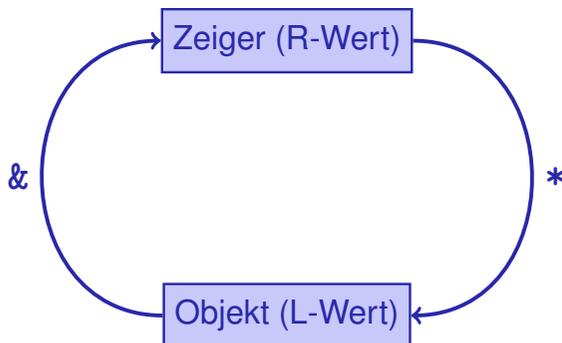
```
int i = 5;  
int* ip = &i; // ip initialisiert  
           // mit Adresse von i.  
int j = *ip; // j == 5
```



465

466

Adress- und Dereferenzoperator



467

Zeiger-Typen

Man zeigt nicht mit einem `double*` auf einen `int`!

Beispiele

```
int* i = ...; // an Adresse i "wohnt" ein int...  
double* j = i; //...und an j ein double: Fehler!
```

468

Eselsbrücke

Die Deklaration

```
T* p;    p ist vom Typ "Zeiger auf T"
```

kann gelesen werden als

```
T *p;    *p ist vom Typ T
```

Obwohl das legal ist,
schreiben wir es nicht so!

Zeiger-Arithmetik: Zeiger plus int

- ptr : Zeiger auf Element $a[k]$ des Arrays a mit Länge n
- Wert von $expr$: ganze Zahl i mit $0 \leq k + i \leq n$

$ptr + expr$

ist Zeiger auf $a[k + i]$.

Für $k + i = n$ erhalten wir einen *past-the-end*-Zeiger, der nicht dereferenziert werden darf.

469

470

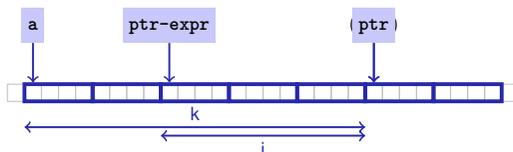
Zeiger-Arithmetik: Zeiger minus int

- Wenn ptr ein Zeiger auf das Element mit Index k in einem Array a der Länge n ist
- und der Wert von $expr$ eine ganze Zahl i ist, $0 \leq k - i \leq n$,

dann liefert der Ausdruck

$ptr - expr$

einen Zeiger zum Element von a mit Index $k - i$.



471

Konversion Feld \Rightarrow Zeiger

Wie bekommen wir einen Zeiger auf das erste Element eines Feldes?

- Statisches Feld vom Typ $T[n]$ ist konvertierbar nach $T*$

Beispiel

```
int a[5];  
int* begin = a; // begin zeigt auf a[0]
```

- Längenangabe geht verloren („Felder sind primitiv“).

472

Iteration über ein Feld mit Zeigern

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

- `a+5` ist ein Zeiger direkt hinter das Ende des Feldes (past-the-end), **der nicht dereferenziert werden darf**.
- Zeigervergleich (`p < a+5`) bezieht sich auf die Reihenfolge der beiden Adressen im Speicher.