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'}
```

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

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

definiert ein Feld der Länge 4, das dem Text “bool” entspricht.

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"
```

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"
```

definiert ein Feld der Länge *5*, das dem Text “bool” entspricht und *null-terminiert* ist. Extrazeichen `'0'` wird am Ende angehängt – Der Text „speichert seine Länge”

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

Texte

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

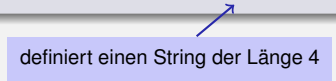
```
std::string text = "bool";
```

definiert einen String der Länge 4

Texte

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

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



Strings: gepimpte `char`-Felder

Ein `std::string`...

- kennt seine Länge

```
text.length()
```

Strings: gepimpte `char`-Felder

Ein `std::string`...

- kann mit variabler Länge initialisiert werden

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

Strings: gepimpte `char`-Felder

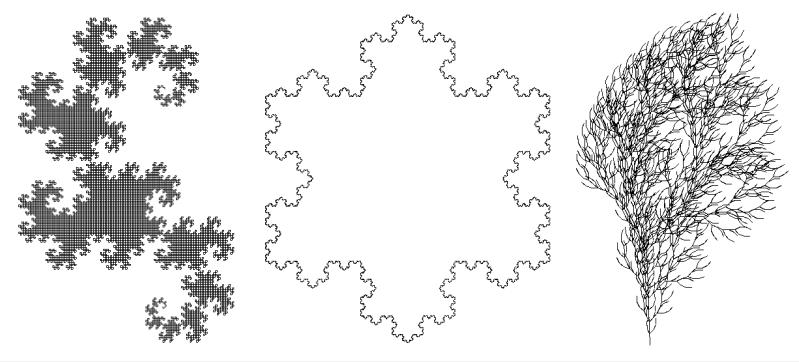
Ein `std::string`...

- „versteht” Vergleiche

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

Lindenmayer-Systeme (L-Systeme)

Fraktale aus Strings und Schildkröten



Definition und Beispiel

- Alphabet Σ
- $\{\mathrm{F}, +, -\}$

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ

- $\{\mathrm{F}, +, -\}$

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ
- Produktion $P : \Sigma \to \Sigma^*$

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

Definition und Beispiel

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

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

- F

Definition und Beispiel

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

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

- F

Definition

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

Die beschriebene Sprache

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

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$w_0 := s_0$

$w_0 := \mathrm{F}$

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$: $P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$w_0 := s_0$ $\qquad$ $w_0 := \mathrm{F}$

$w_1 := P(w_0)$ $\qquad$ $w_1 := \mathrm{F} + \mathrm{F} +$

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$: $P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$w_0 := s_0 \qquad w_0 := \mathrm{F}$$

$$w_1 := P(w_0) \qquad w_1 := \mathrm{F} + \mathrm{F} +$$

$$w_2 := P(w_1) \qquad w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$$

Definition

$P(c_1 c_2 \ldots c_n) := P(c_1) P(c_2) \ldots P(c_n)$

Die beschriebene Sprache

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

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$w_0 := s_0$ $\qquad$ $w_0 := \mathrm{F}$

$w_1 := P(w_0)$ $\qquad$ $w_1 := \mathrm{F} + \mathrm{F} +$

$w_2 := P(w_1)$ $\qquad$ $w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$

$P(\mathrm{F})\ P(+)\ P(\mathrm{F})\ P(+)$

Definition

$P(c_1 c_2 \ldots c_n) := P(c_1) P(c_2) \ldots P(c_n)$

Die beschriebene Sprache

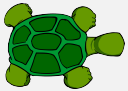
Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$
\begin{array}{lcl}
w_0 & := & s_0 \\
w_1 & := & P(w_0) \\
w_2 & := & P(w_1) \\
 & \vdots &
\end{array}
\qquad
\begin{array}{lcl}
w_0 & := & \mathrm{F} \\
w_1 & := & \mathrm{F} + \mathrm{F} + \\
w_2 & := & \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + + \\
 & \vdots &
\end{array}
$$

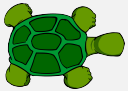
Turtle-Grafik

Schildkröte mit Position und Richtung



Turtle-Grafik

Schildkröte mit Position und Richtung

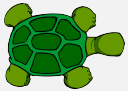


Schildkröte versteht 3 Befehle:

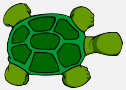
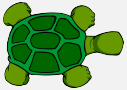
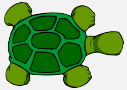
F: Gehe einen Schritt vorwärts	$+$: Drehe dich um 90 Grad	$-$: Drehe dich um -90 Grad

Turtle-Grafik

Schildkröte mit Position und Richtung

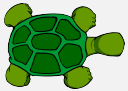


Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts 	$+$: Drehe dich um 90 Grad 	$-$: Drehe dich um -90 Grad 

Turtle-Grafik

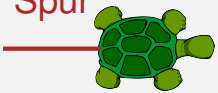
Schildkröte mit Position und Richtung



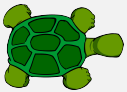
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓

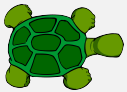
Spur



$+$: Drehe dich um 90 Grad

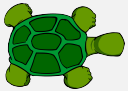


$-$: Drehe dich um -90 Grad



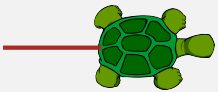
Turtle-Grafik

Schildkröte mit Position und Richtung



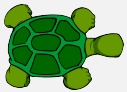
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓



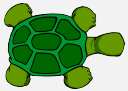
$+$: Drehe dich um 90 Grad ✓

$-$: Drehe dich um -90 Grad



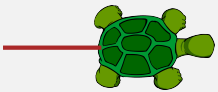
Turtle-Grafik

Schildkröte mit Position und Richtung



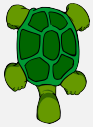
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓



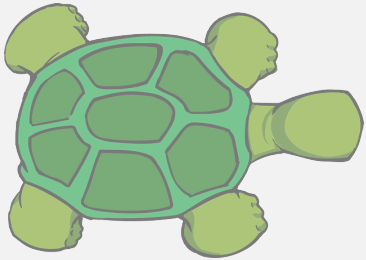
$+$: Drehe dich um 90 Grad ✓

$-$: Drehe dich um -90 Grad ✓



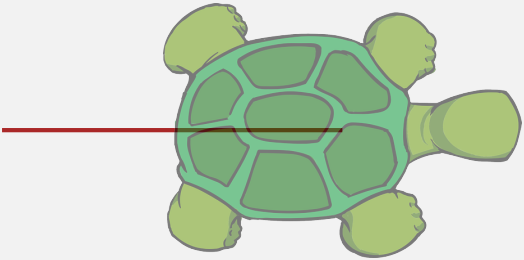
Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$



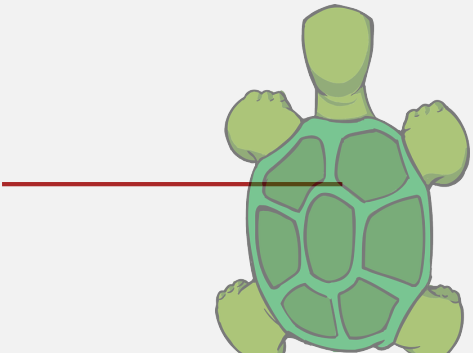
Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$



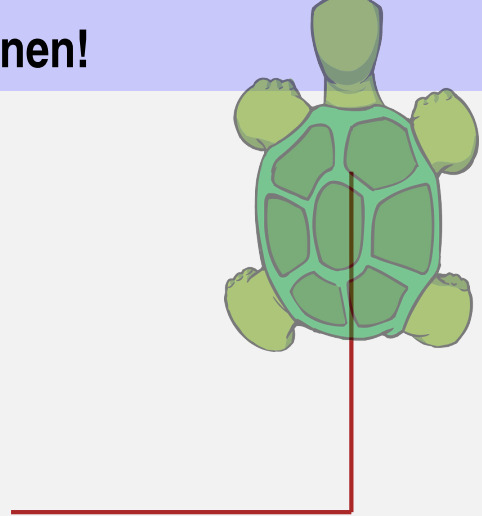
Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$

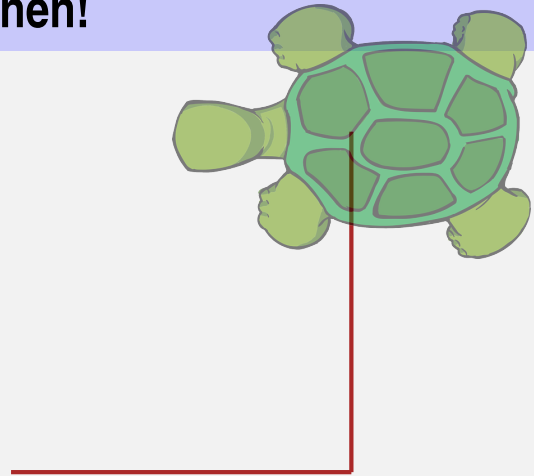


Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$



Wörter zeichnen!



$w_1 = \mathrm{F} + \mathrm{F} +$

Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} + \checkmark$

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#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);
```

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#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);
```

$w = w_0 = \mathrm{F}$

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#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);
```

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

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#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);
```

Zeichne $w = w_n$!

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;
}
```

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
  }
}
```

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);
    }
}
```

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);
    }
}
```

Springe zum `case`, der `word[k]` entspricht.

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);
    }
}
```

Vorwärts! (Funktion aus unserer Schildkröten-Bibliothek)

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);
    }
}
```

Überspringe die folgenden cases

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);
    }
}
```

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

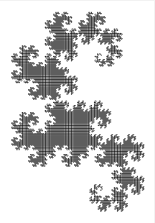
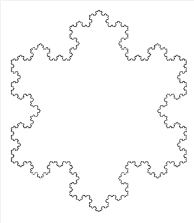
lindenmayer.cpp: draw_word

```
// 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);
    }
}
```

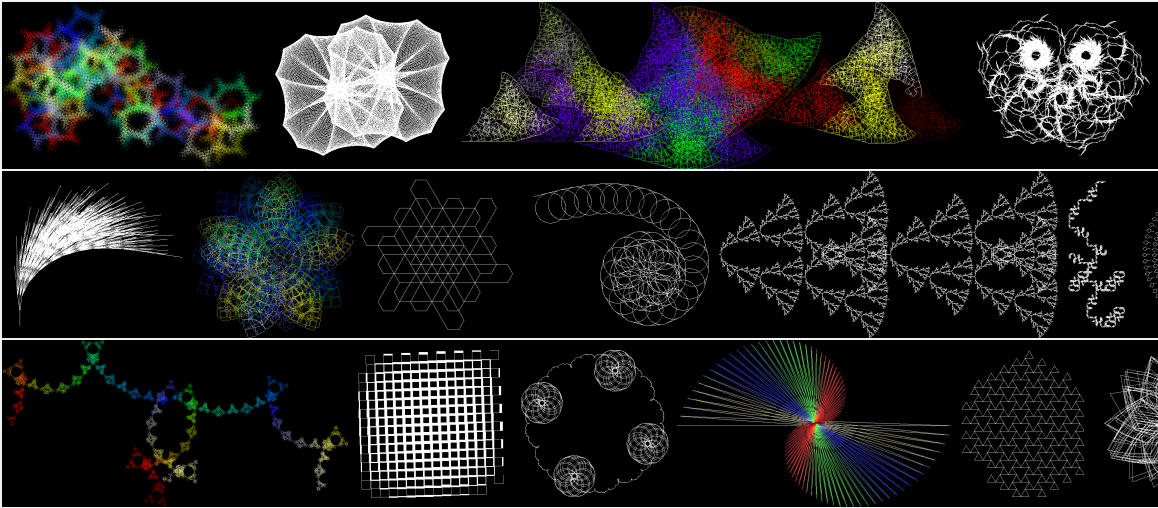
Drehe dich um -90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

L-Systeme: Erweiterungen

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


L-System-Challenge: `amazing.cpp`!



Mehrdimensionale Felder

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

Mehrdimensionale Felder

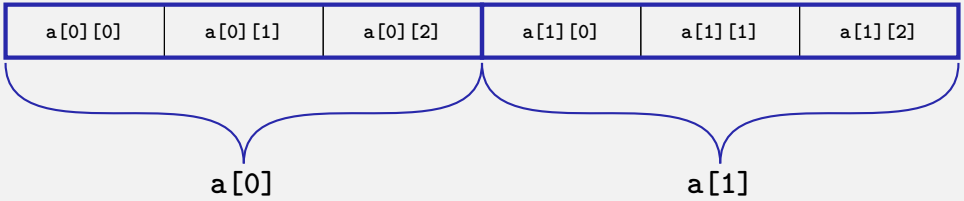
- sind Felder von Feldern

```
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



Mehrdimensionale Felder

Im Speicher: flach

Im Kopf: Matrix

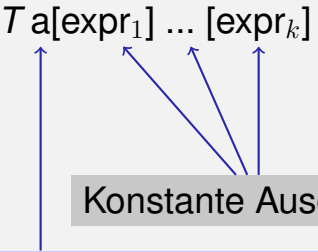
Spalten →

Zeilen ↓

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]

Mehrdimensionale Felder

- sind Felder von Feldern von Feldern ....



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

2	4	6	1	3	5

Mehrdimensionale Felder

Initialisierung:

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

Erste Dimension kann weggelassen werden

Vektoren von Vektoren

- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?

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<?> a (n, ?);
```

Zugrundeliegender Typ des ersten Vektors?

Vektoren von Vektoren

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

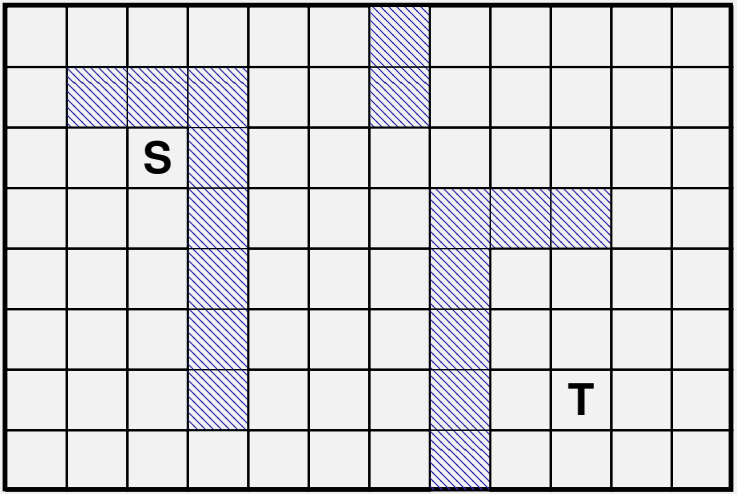
Beispiel: Vektor der Länge n von Vektoren der Länge m:

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

Zugrundeliegender Typ des ersten Vektors?
`std::vector<int>`, initialisiert mit Länge m:
`std::vector<int>(m)`

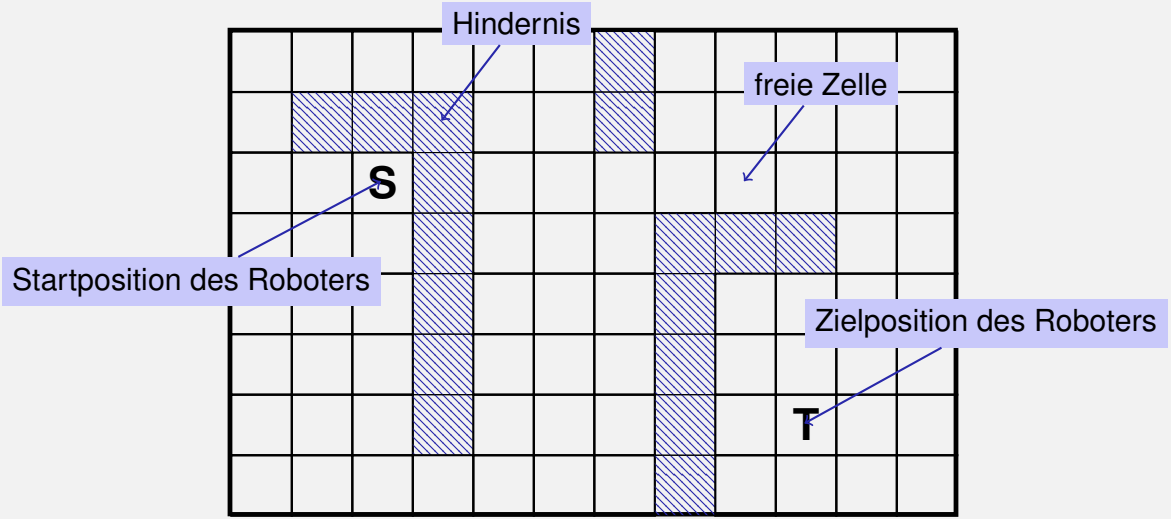
Anwendung: Kürzeste Wege

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



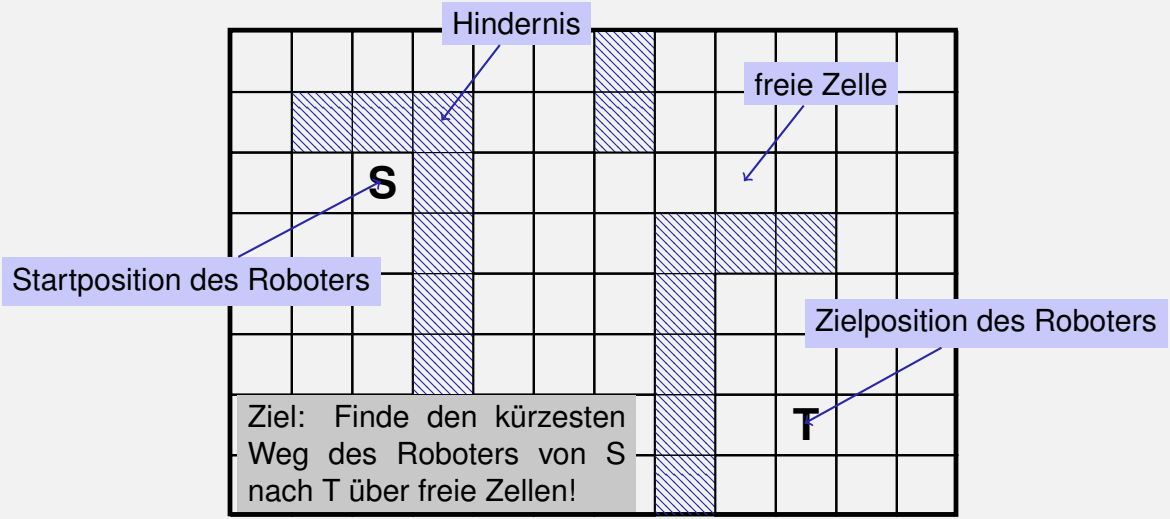
Anwendung: Kürzeste Wege

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



Anwendung: Kürzeste Wege

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



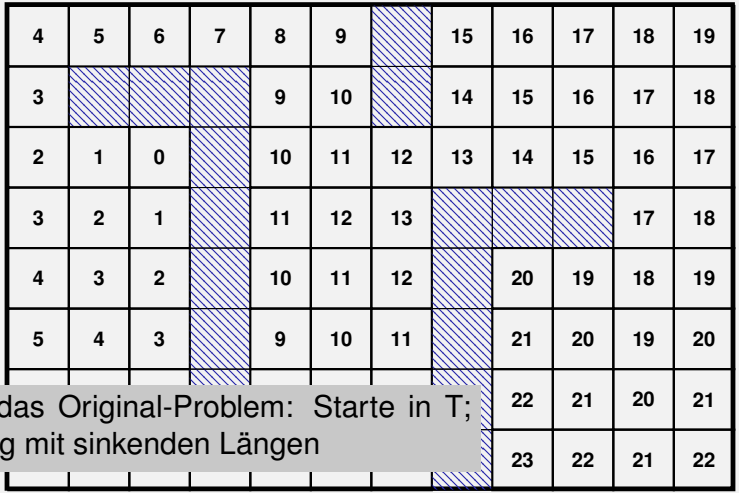
Ein (scheinbar) anderes Problem

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

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1		11	12	13				17	18
4	3	2		10	11	12		20	19	18	19
5	4	3		9	10	11		21	20	19	20
6	5	4		8	9	10		22	21	20	21
7	6	5	6	7	8	9		23	22	21	22

Ein (scheinbar) anderes Problem

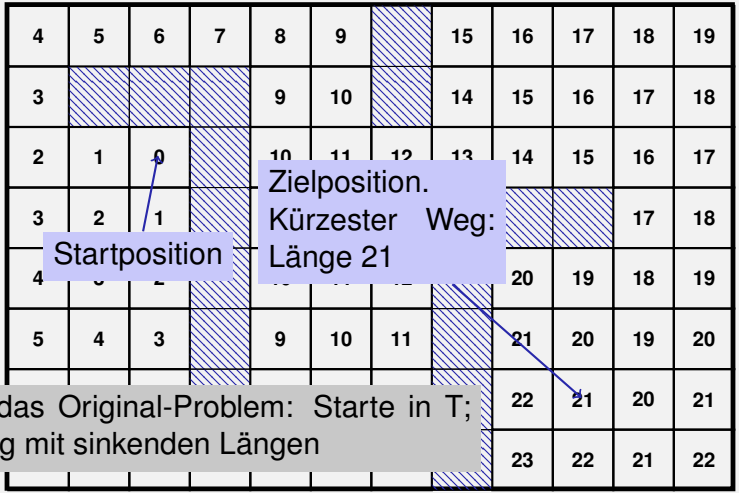
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



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

Ein (scheinbar) anderes Problem

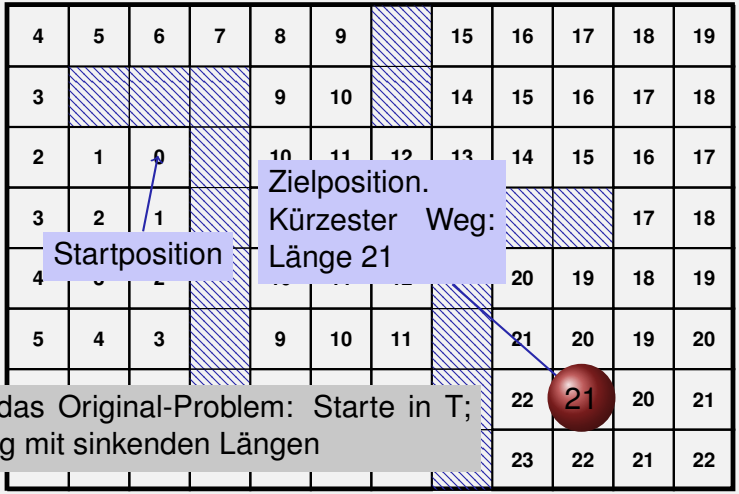
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



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

Ein (scheinbar) anderes Problem

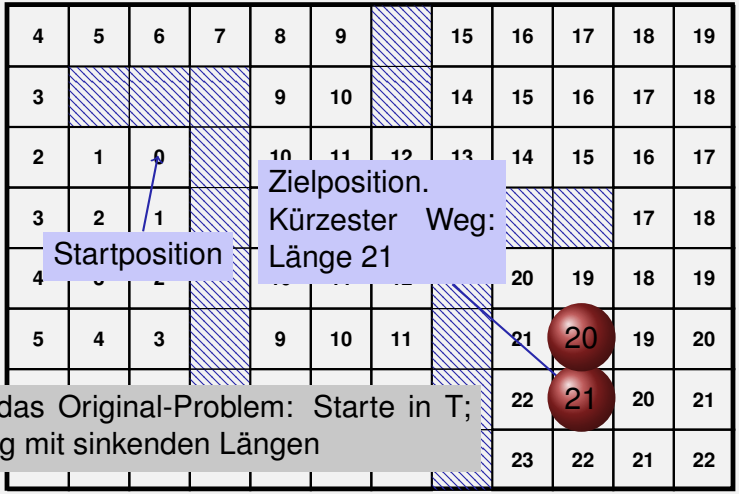
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



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

Ein (scheinbar) anderes Problem

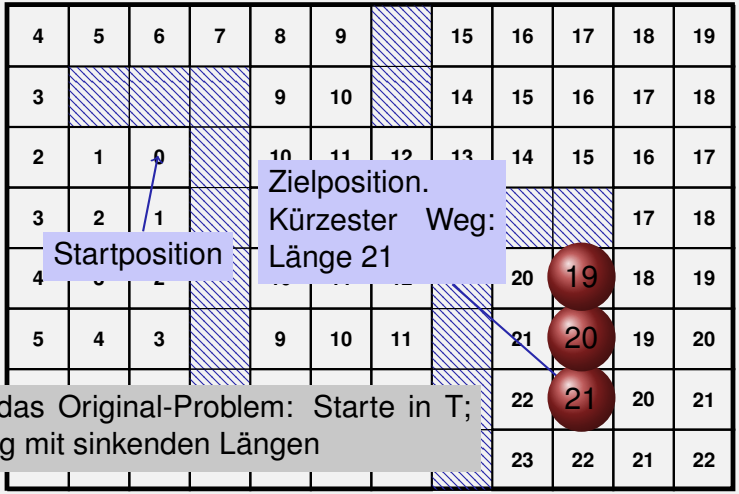
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



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

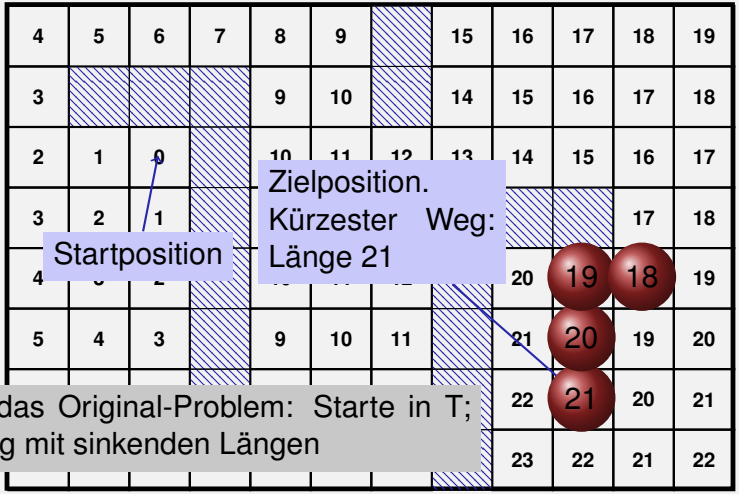
Ein (scheinbar) anderes Problem

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



Ein (scheinbar) anderes Problem

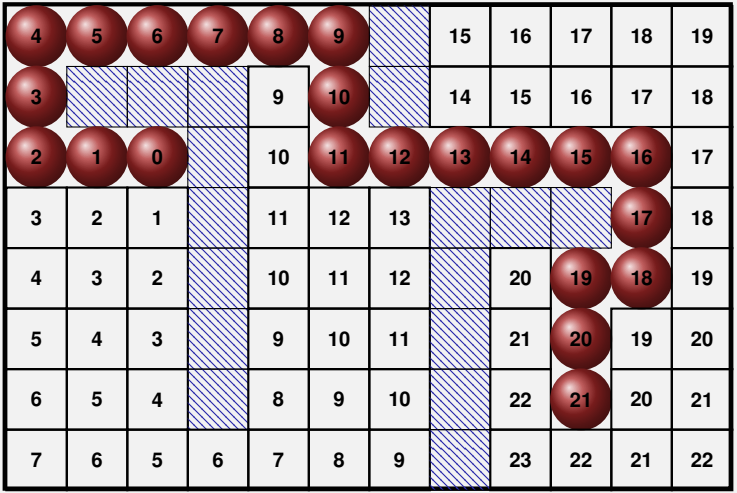
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



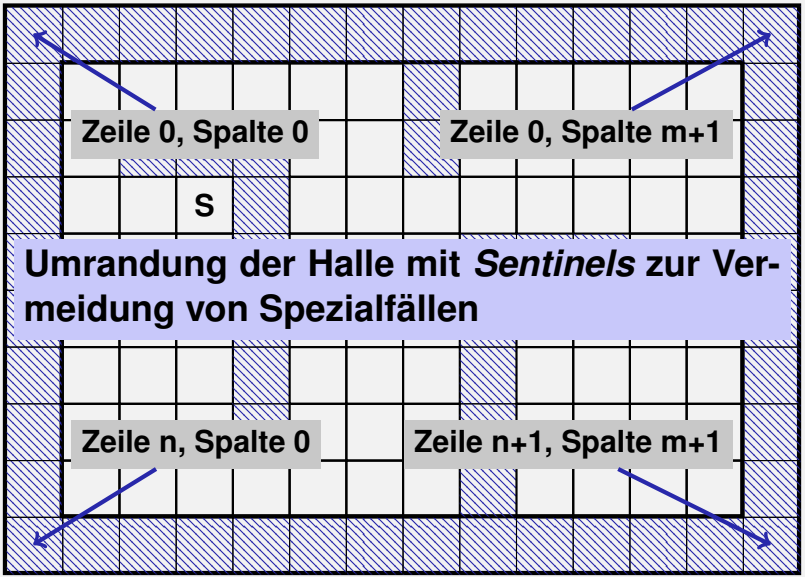
Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

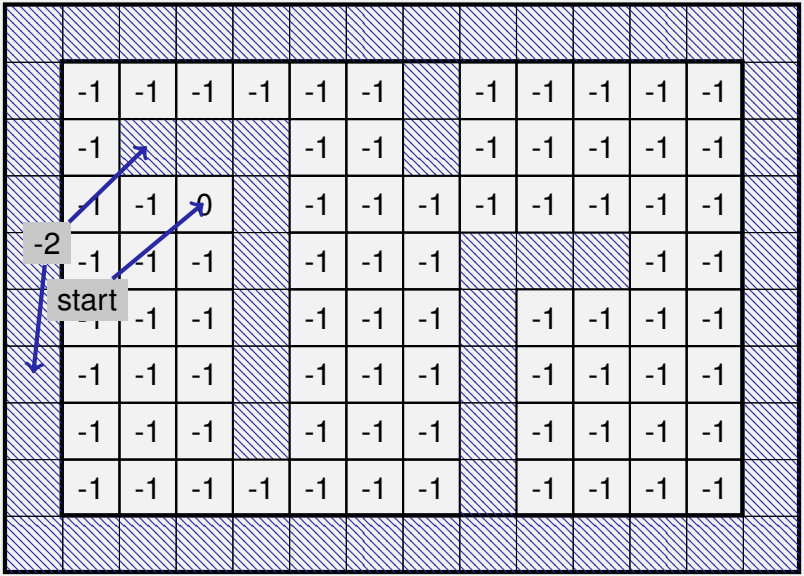
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Vorbereitung: Wächter (*Sentinels*)



Vorbereitung: Initiale Markierung



Das Kürzeste-Wege-Programm

```
// 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));


// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

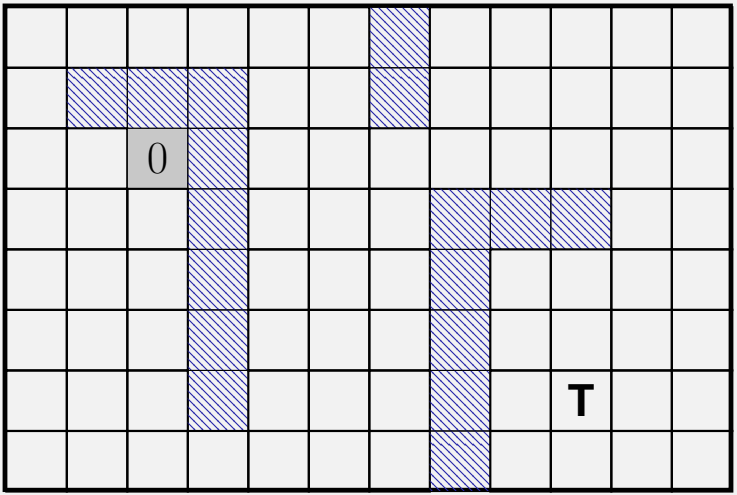
Das Kürzeste-Wege-Programm

```
// 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));
```


```
// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

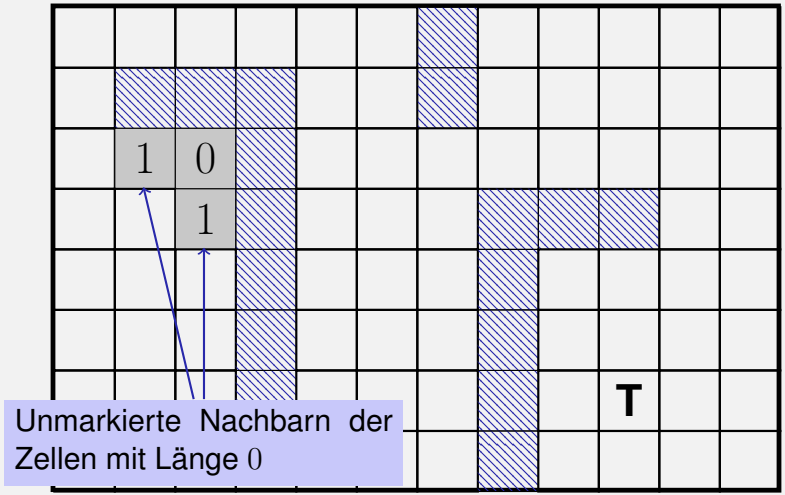
Markierung aller Zellen mit ihren Weglängen

Schritt 0: Alle Zellen mit Weglänge 0



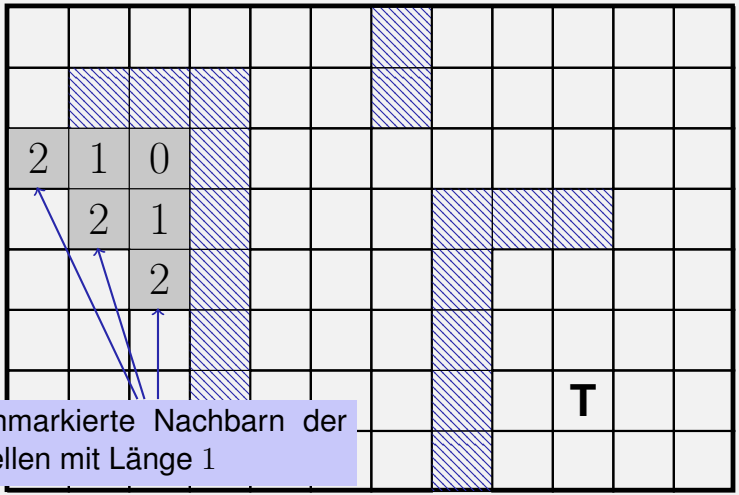
Markierung aller Zellen mit ihren Weglängen

Schritt 1: Alle Zellen mit Weglänge 1



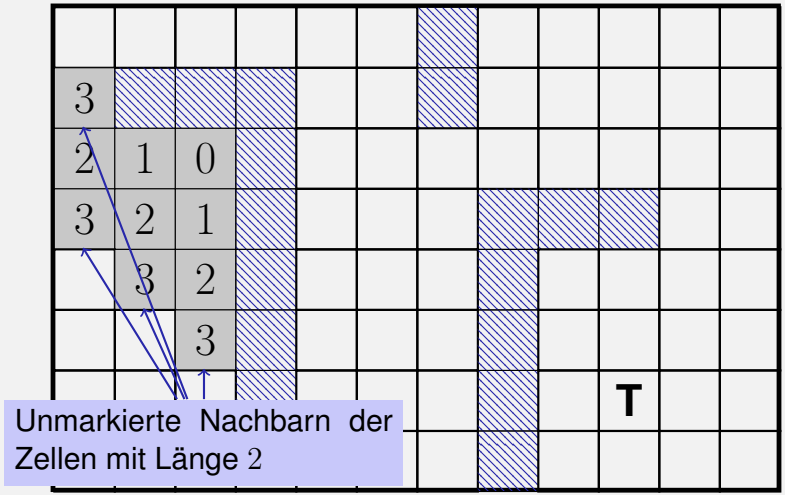
Markierung aller Zellen mit ihren Weglängen

Schritt 2: Alle Zellen mit Weglänge 2



Markierung aller Zellen mit ihren Weglängen

Schritt 3: Alle Zellen mit Weglänge 3



Hauptschleife

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

```
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;
}
```

Hauptschleife

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

```
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;
}
```

zeigt an, ob in einem Durchlauf durch alle Zellen Fortschritt gemacht wurde

Hauptschleife

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

```
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;
}
```

Gehe über alle Zellen

Hauptschleife

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

```
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;
}
```

Zelle schon markiert oder Hindernis

Hauptschleife

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

```
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;
}
```

Ein Nachbar hat Weglänge $i - 1$. Die Wächter garantieren immer 4 Nachbarn.

Hauptschleife

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

```
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;
}
```

Kein Fortschritt, alle erreichbaren Zellen markiert; fertig.

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche*

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche*
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche*
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden
- Verbesserung: Für Markierung i, durchlaufe nur die Nachbarn der Zellen mit Markierung $i - 1$

Felder als Funktionsargumente

Felder können auch als *Referenz*-Argumente an eine Funktion übergeben werden. (Hier const, weil nur Lesezugriff nötig).

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

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";
  }
}
```

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] << " ";
  }
}
```

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.

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";
  }
}
```

13. Zeiger, Algorithmen, Iteratoren und Container I

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

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;
}
```

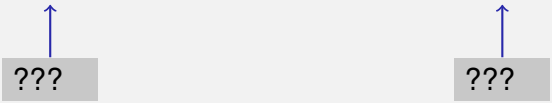
Komische Dinge...

```
#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;
}
```



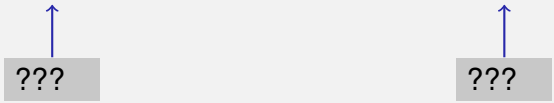
Komische Dinge...

```
#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!

Referenzen: Wo ist Anakin?

```
int anakin_skywalker = 9;
```

Referenzen: Wo ist Anakin?


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

Referenzen: Wo ist Anakin? Zuweisung

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

// anakin_skywalker = 22
```

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
```

Zeiger: Wo ist Anakin?

```
int anakin_skywalker = 9;
```

Zeiger: Wo ist Anakin?

```
int anakin_skywalker = 9;
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Zeiger: Wo ist Anakin?

Initialisierung

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Zeiger: Wo ist Anakin?

Ausgabe

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

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Zeiger: Wo ist Anakin?

Zuweisung

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

// anakin_skywalker = 22
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

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
```

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`.

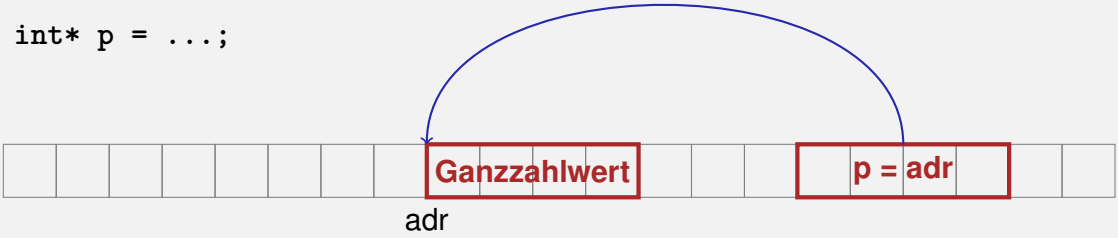
Beispiele

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

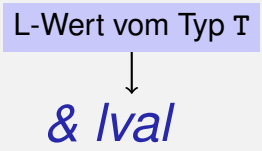
Zeiger Typen

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

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



Adress-Operator



Typ: *T**

Wert: Adresse von *lval*

Adress-Operator

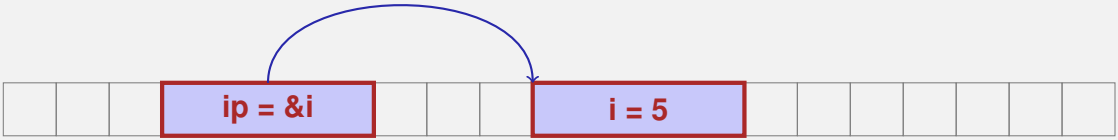
Beispiel

```
int i = 5;
```

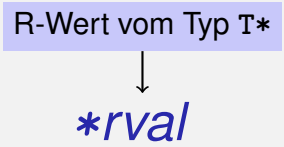
Adress-Operator

Beispiel

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



Dereferenz-Operator



Typ: *T*

Wert: *Wert* des Objekts an Addresse *rval*

Dereferenz-Operator

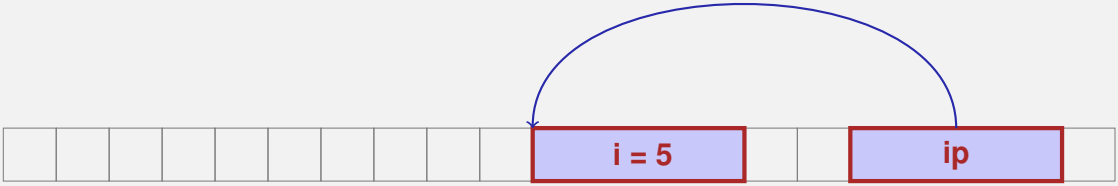
Beispiel

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

Dereferenz-Operator

Beispiel

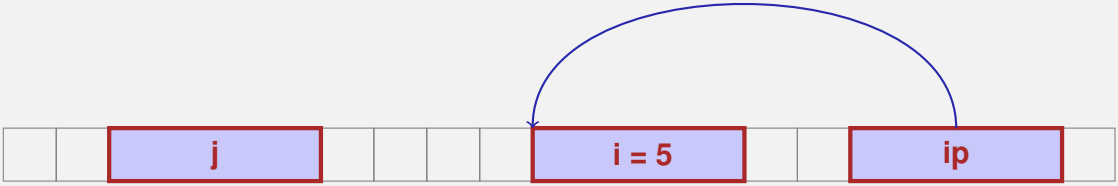
```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



Dereferenz-Operator

Beispiel

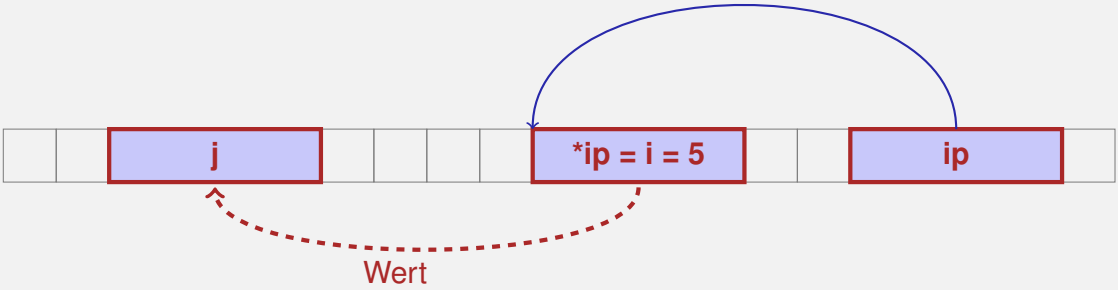
```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



Dereferenz-Operator

Beispiel

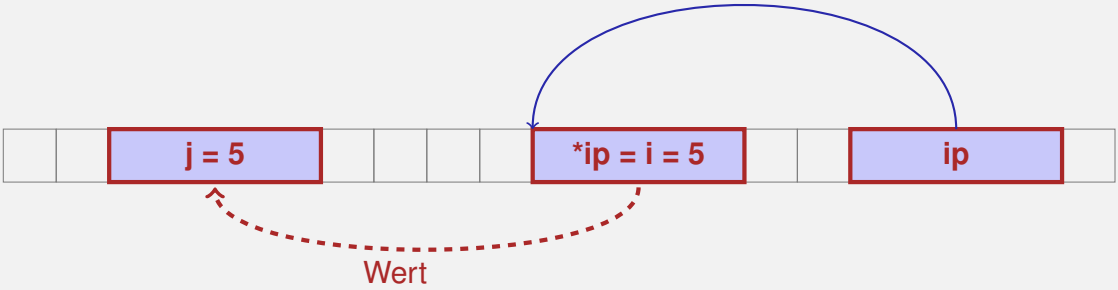
```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



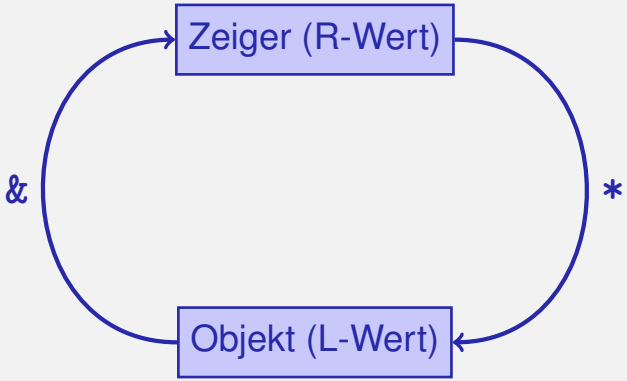
Dereferenz-Operator

Beispiel

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



Adress- und Dereferenzoperator



Zeiger-Typen

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

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!
```

Eselsbrücke

Die Deklaration

```
T* p;
```

`p` ist vom Typ “Zeiger auf `T`”

Eselsbrücke

Die Deklaration

```
T* p;        p ist vom Typ “Zeiger auf T”
```

kann gelesen werden als

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

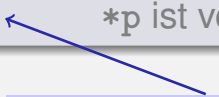
Eselsbrücke

Die Deklaration

```
T* p;        p ist vom Typ “Zeiger auf T”
```

kann gelesen werden als

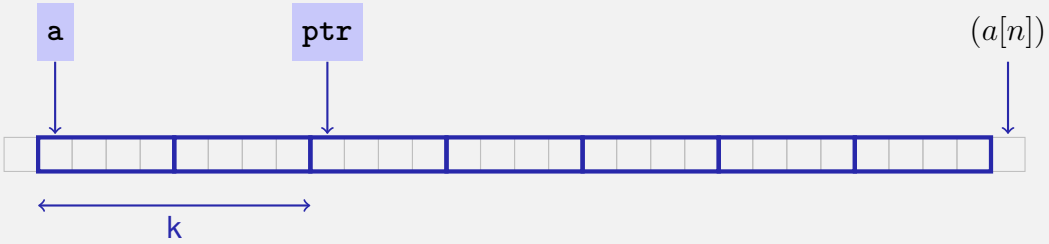
```
T *p;        *p ist vom Typ T
```



Obwohl das legal ist, schreiben wir es nicht so!

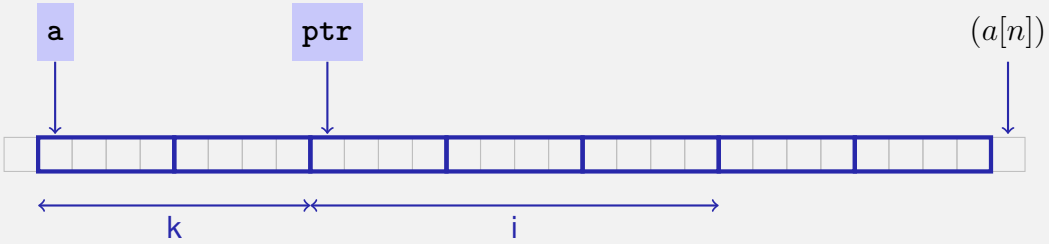
Zeiger-Arithmetik: Zeiger plus `int`

- *ptr*: Zeiger auf Element $a[k]$ des Arrays a mit Länge n



Zeiger-Arithmetik: Zeiger plus `int`

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

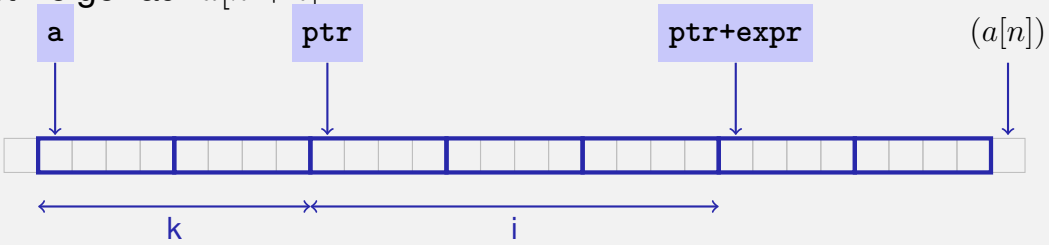


Zeiger-Arithmetik: Zeiger plus `int`

- *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 $a[k + i]$.



Konversion Feld $\Rightarrow$ Zeiger

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

Konversion Feld $\Rightarrow$ Zeiger

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

- Statisches Feld vom Typ $T[\mathrm{n}]$ ist konvertierbar nach $T*$

Beispiel

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

Konversion Feld $\Rightarrow$ Zeiger

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

- Statisches Feld vom Typ $T[\mathrm{n}]$ ist konvertierbar nach $T*$

Beispiel

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

- Längeninformation geht verloren („Felder sind primitiv”).

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
```

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.

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.