

# 12. Felder (Arrays) II

Caesar-Code, Strings, Lindenmayer-Systeme,  
Mehrdimensionale Felder, Vektoren von Vektoren,  
Kürzeste Wege

# Caesar-Code

Ersetze jedes druckbare Zeichen in einem Text durch seinen Vor-Vor-Vorgänger.

' ' (32) → '|' (124)

'!' (33) → '}' (125)

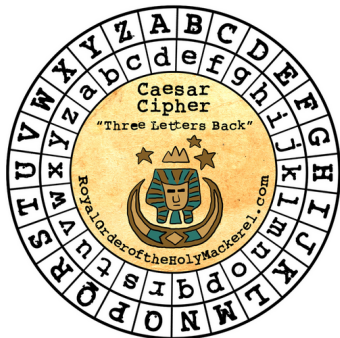
⋮

'D' (68) → 'A' (65)

'E' (69) → 'B' (66)

⋮

~ (126) → '{' (123)



# Caesar-Code:

# Hauptprogramm

```
// Program: caesar_encrypt.cpp
// encrypts a text by applying a cyclic shift of -3

#include<iostream>
#include<cassert>
#include<ios> // for std::noskipws ← Leerzeichen und Zeilenumbrüche
// sollen nicht ignoriert werden

// PRE: s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//        cyclically shifted s printable characters to the right
void shift (char& c, int s),

int main ()
{
    std::cin >> std::noskipws; // don't skip whitespaces!

    // encryption loop
    char next;
    while (std::cin >> next) ← Konversion nach bool: liefert false
        shift (next, -3); ← genau dann, wenn die Eingabe leer
        std::cout << next; // ist.

    }
    return 0;
}

// Behandelt nur die druckbaren Zeichen.
```

```
// PRE:  s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
    assert (s < 95 && s > -95);
    if (c >= 32 && c <= 126) {
        if (c + s > 126)
            c += (s - 95);
        else if (c + s < 32)
            c += (s + 95);
        else
            c += s;
    }
}
```

Call by reference!

Überlauf – 95 zurück!

Unterlauf – 95 vorwärts!

Normale Verschiebung

# ./caesar\_encrypt < power8.cpp

```
„|Moldo^j7|mltbo5+`mm  
„|O^fpb|^|krj_bo|ql|qeb|bfdeqe|mltbo+
```

Program = Moldo<sup>j</sup>

```
fk`irab|9flpqob^j;|
```

```
fkq|j^fk%&
```

```
x
```

```
||„|fkmrq
```

```
||pqa77`lrq|99|~@ljmrqb|^ [5|clo|^|: <|~8||
```

```
||fkq|^8
```

```
||pqa77`fk|;|^8
```

```
||„|`ljmrq^qflk
```

```
||fkq|_|:|^|'|^8|„|_|:|^|/
```

```
||_|:|_|'|_8| ||||„|_|:|^|1
```

```
||„|lrqmrq|_|'|_) |f+b+)^ [5
```

```
||pqa77`lrq|99|^|99|~ [5|:|~|99|_|'|_|99|~+Yk~8
```

```
||obqrok|-8
```

```
z
```

# Caesar-Code: Entschlüsselung

```
int main ()
{
    std::cin >> std::noskipws; // don't skip whitespaces!

    // decryption loop
    char next;
    while (std::cin >> next) {
        shift (next, 3);
        std::cout << next;
    }

    return 0;
}
```

Jetzt: Verschiebung um 3  
nach *rechts*

Interessante Art, `power8.cpp` auszugeben:

- `./caesar_encrypt < power8.cpp | ./caeser_decrypt`

# 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>`

# 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

- „versteht“ Vergleiche

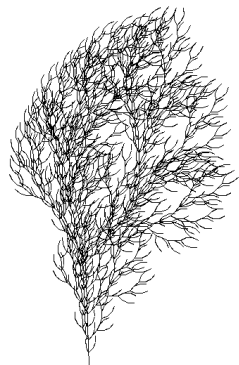
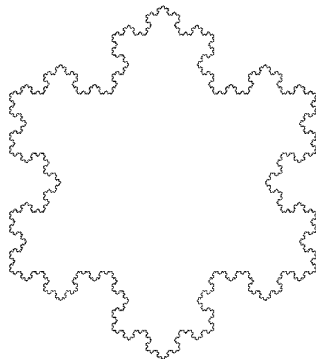
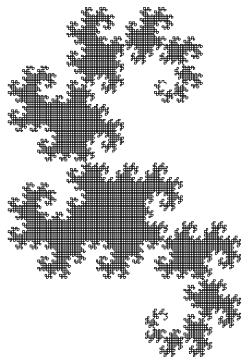
```
if (text1 == text2) ...
```

`true` wenn `text1` und `text2` übereinstimmen



# 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  $\Sigma$
- $\Sigma^*$ : alle endlichen Wörter über  $\Sigma$
- Produktion  
 $P : \Sigma \rightarrow \Sigma^*$
- Startwort  $s \in \Sigma^*$

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

| $c$ | $P(c)$  |
|-----|---------|
| F   | F + F + |
| +   | +       |
| -   | -       |

- F

## Definition

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

# Die beschriebene Sprache

Wörter  $w_0, w_1, w_2, \dots \in \Sigma^*$ :  $P(F) = F + F +$

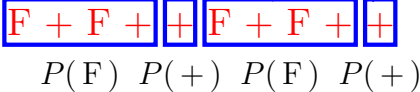
$w_0 := s$

$w_0 := F$

$w_1 := P(w_0)$

$w_1 :=$  

$w_2 := P(w_1)$

$w_2 :=$  

$\vdots$

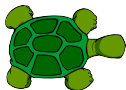
$\vdots$

## Definition

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

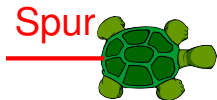
# Turtle-Grafik

Schildkröte mit Position und Richtung

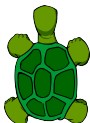


Schildkröte versteht 3 Befehle:

**F**: Gehe  
einen Schritt  
vorwärts ✓



**+**: Drehe  
dich um 90  
Grad ✓

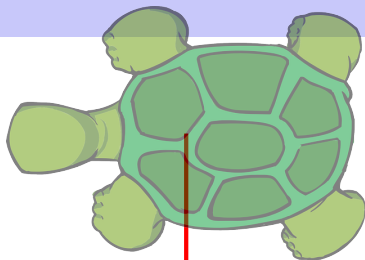
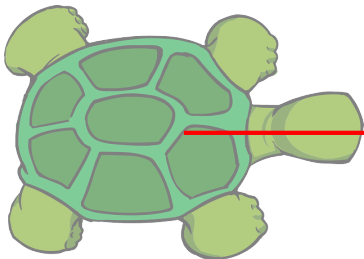


**-**: Drehe  
dich um -90  
Grad ✓



# Wörter zeichnen!

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



# lindenmayer.cpp: Hauptprogramm

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

std::string

```
int main () {
    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);

    return 0;
}
```

$$w = w_0 = F$$

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

Zeichne  $w = w_n!$

```
// POST: replaces all symbols in word according to their
//       production and returns the result
std::string next_word (const 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 (const char c)
{
    switch (c) {
        case 'F':
            return "F+F+";
        default:
            return std::string (1, c); // trivial production c -> c
    }
}
```

```
// POST: draws the turtle graphic interpretation of word
```

```
void draw_word (const std::string& word)
```

```
{  
    for (unsigned int k = 0; k < word.length(); ++k)
```

```
        switch (word[k]) {  
            case 'F' :
```

```
                ifmp::forward();
```

```
                break;
```

```
            case '+' :
```

```
                ifmp::left(90);
```

```
                break;
```

```
            case '-' :
```

```
                ifmp::right(90);
```

```
        }  
}
```

springe zum case, der dem Wert von word[k] entspricht

Vorwärts! (Funktion aus unserer Schildkröten-Bibliothek)

Überspringe die folgenden cases

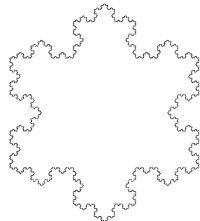
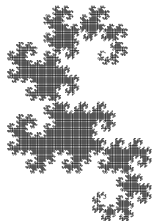
Drehe dich um 90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

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

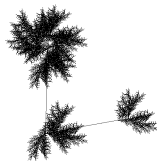
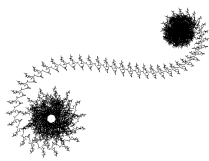
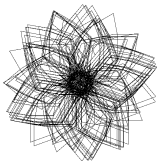
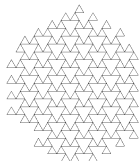
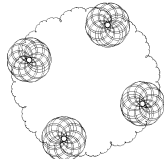
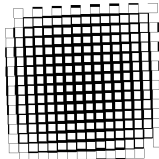
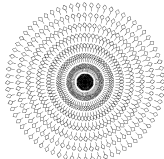
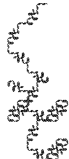
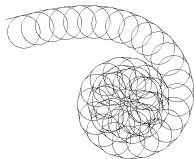
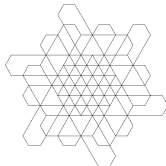
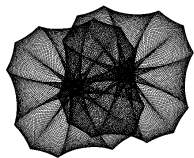


# L-Systeme: Erweiterungen

- 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`!



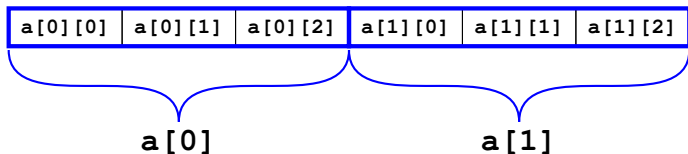
# Mehrdimensionale Felder

- 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`

Im Speicher: flach



# Mehrdimensionale Felder

- sind Felder von Feldern von Feldern ....

$T a[\text{expr}_1] \dots [\text{expr}_k]$

Konstante Ausdrücke!

$\mathbf{a}$  hat  $\text{expr}_1$  Elemente und jedes von ihnen ist ein Feld mit  $\text{expr}_2$  Elementen, von denen jedes ein Feld mit  $\text{expr}_3$  Elementen ist, ...

# Mehrdimensionale Felder

Initialisierung:

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

Erste Dimension kann wegge-  
lassen werden

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

# Vektoren von Vektoren

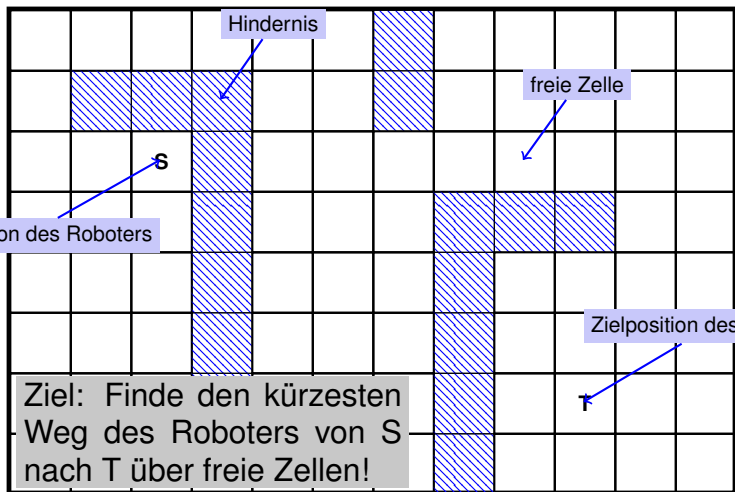
- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?
- Lösung: Vektoren von Vektoren

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

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

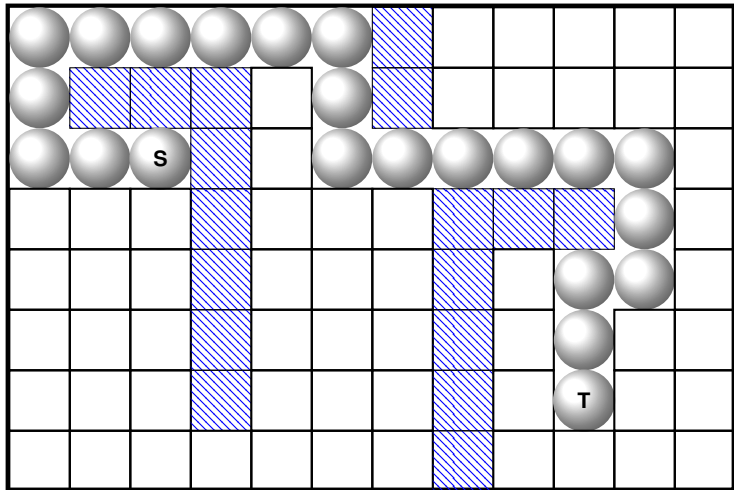
# Anwendung: Kürzeste Wege

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



# Anwendung: Kürzeste Wege

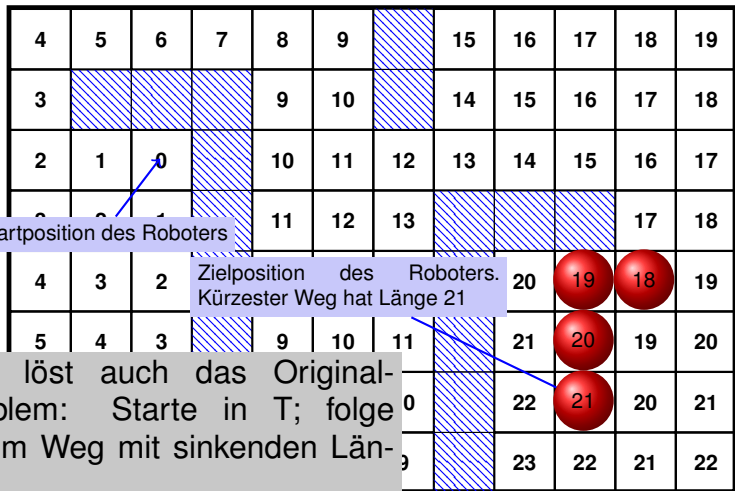
Lösung





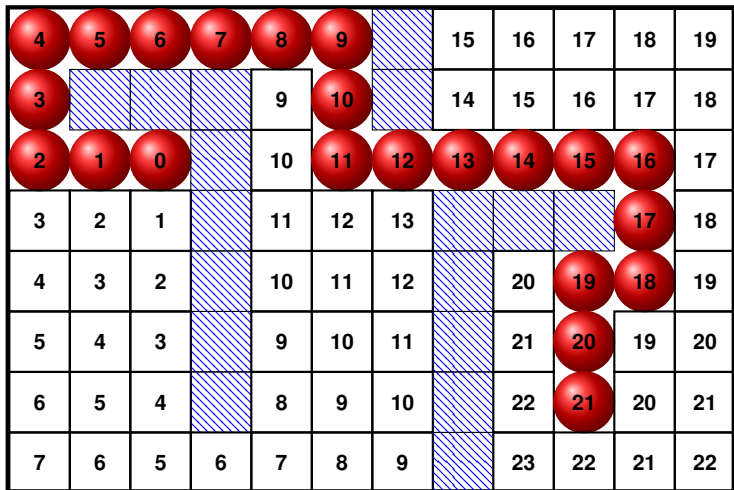
# Ein (scheinbar) anderes Problem

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

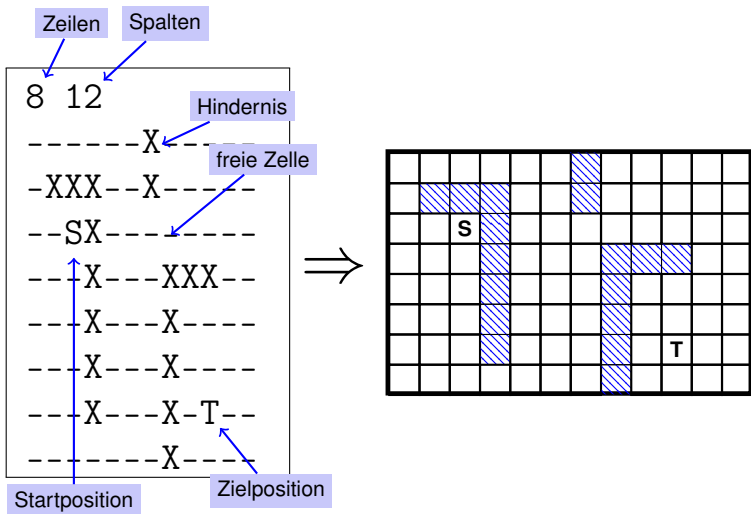


# Ein (scheinbar) anderes Problem

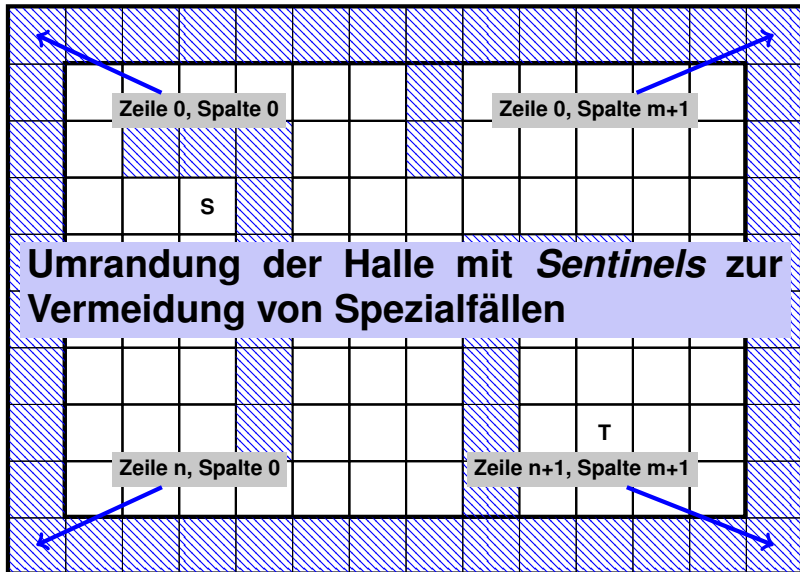
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



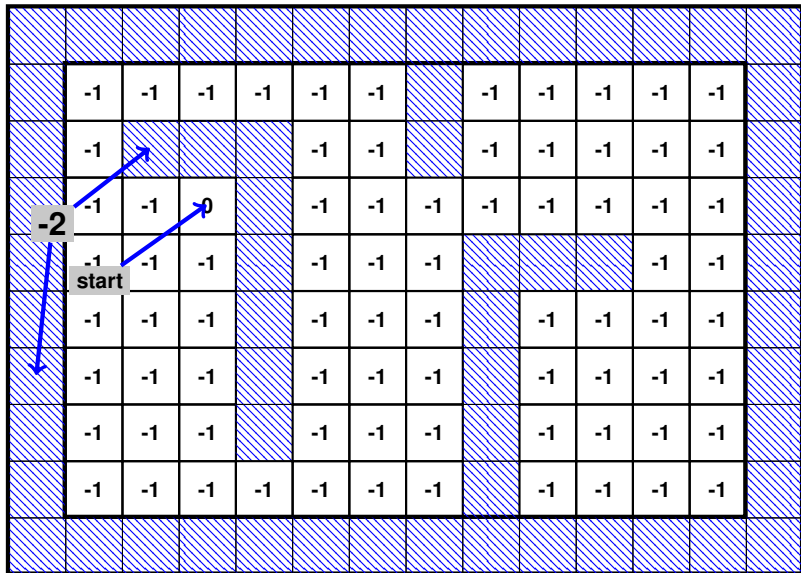
# Vorbereitung: Eingabeformat



# Vorbereitung: Wächter (*Sentinels*)



# Vorbereitung: Initiale Markierung



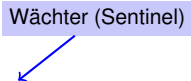
# Das Kürzeste-Wege-Programm

- 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));
```



# 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;
    }
```

# Das Kürzeste-Wege-Programm

Hinzufügen der umschliessenden „Wände“

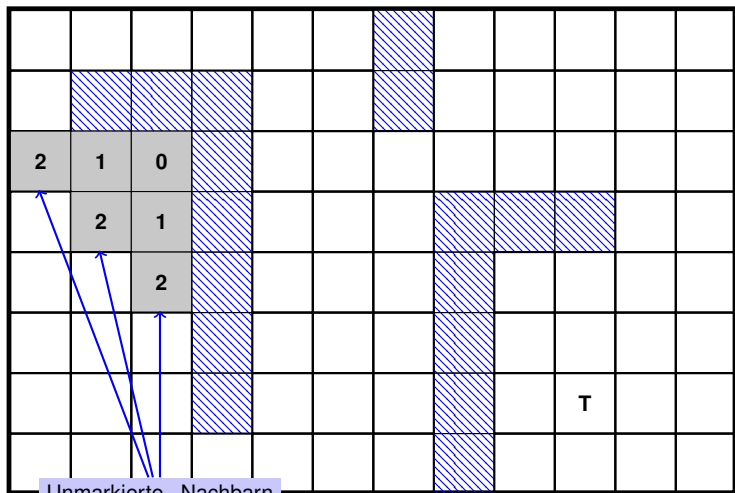
```
for (int r=0; r<n+2; ++r)
    floor[r][0] = floor[r][m+1] = -2;

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



Unmarkierte Nachbarn  
der Zellen mit Länge 1

# Das Kürzeste-Wege-Programm

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;
}
```

# 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)
}
```

# Markierung am Ende

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

# Das Kürzeste-Wege-Programm: Ausgabe

## Ausgabe

```
for (int r=1; r<n+1; ++r) {  
    for (int c=1; c<m+1; ++c)  
        if (floor[r][c] == 0)  
            std::cout << 'S';  
        else if (r == tr && c == tc)  
            std::cout << 'T';  
        else if (floor[r][c] == -3)  
            std::cout << 'o';  
        else if (floor[r][c] == -2)  
            std::cout << 'X';  
        else  
            std::cout << '-';  
    std::cout << "\n";  
}
```



```
○○○○○○X-----  
○XXX-○X-----  
○○SX-○○○○○○-  
---X---XXX○-  
---X---X-○○-  
---X---X-○--  
---X---X-T---  
-----X-----
```

# Das Kürzeste-Wege-Programm

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

# 13. Zeiger, Algorithmen, Iteratoren und Container I

Zeiger, Felder als Funktionsargumente


# 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+ 5);

    return 0;
}
```



The diagram consists of two grey rectangular boxes, each containing three question marks '???' in black text. A blue arrow points from the first box to the asterisk '\*' in the code line 'std::cout << \*std::min\_element (a, a+ 5);'. A second blue arrow points from the second box to the 'a+ 5' part of the same code line.

Dafür müssen wir zuerst *Zeiger* verstehen!



# Rückblick Referenzen

```
float ms = 1e7;  
float& panta_rhei = ms;  
panta_rhei = 1.1e7;
```

`panta_rhei`  $\approx$  intern 0xD1

`ms` Wert: 11 Mio.  
Adresse: 0xD1

|   | 1               | 2 | 3 | 4 | 5 | 6 |
|---|-----------------|---|---|---|---|---|
| A |                 |   |   |   |   |   |
| B |                 |   |   |   |   |   |
| C |                 |   |   |   |   |   |
| D | <code>ms</code> |   |   |   |   |   |
| E |                 |   |   |   |   |   |
| F |                 |   |   |   |   |   |



`panta_rhei`

# Ausblick: Zeiger

```
float ms = 1e7;  
float* link_to_ms = &ms;  
*link_to_ms = 1.1e7;
```

```
link_to_ms == 0xD1
```

|                       |                                |
|-----------------------|--------------------------------|
| <code>&amp;x</code> : | Link auf <code>x</code>        |
| <code>*l</code> :     | darauf verweist <code>l</code> |

`ms` Wert: 11 Mio.  
Adresse: 0xD1

|   | 1               | 2 | 3 | 4 | 5 | 6 |
|---|-----------------|---|---|---|---|---|
| A |                 |   |   |   |   |   |
| B |                 |   |   |   |   |   |
| C |                 |   |   |   |   |   |
| D | <code>ms</code> |   |   |   |   |   |
| E |                 |   |   |   |   |   |
| F |                 |   |   |   |   |   |

`link_to_ms` Wert: 0xD1

# Swap mit Zeigern

```
void swap(int* l1, int* l2) {  
    int t = *l1;  
    *l1 = *l2;  
    *l2 = t;  
}
```

```
...  
int x = 2;  
int y = 3;  
swap(&x, &y);  
std::cout << "x = " << x << "\n"; // 3  
std::cout << "y = " << y << "\n"; // 2
```

# Zeiger Typen

$\mathbf{T}^*$

Zeiger-Typ zum zugrunde liegenden  
Typ  $\mathbf{T}$ .

Ein Ausdruck vom Typ  $\mathbf{T}^*$  heisst *Zeiger* (auf  $\mathbf{T}$ ).

# Zeiger Typen

Wert eines Zeigers auf  $\mathbf{T}$  ist die Adresse eines Objektes vom Typ  $\mathbf{T}$ .

## Beispiele

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

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

`int* p = ...;`

`p`: Adresse (der ersten Speicherzelle) eines `int`

