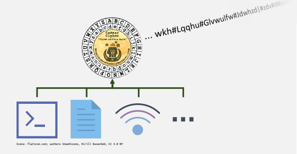
14. Characters and Texts II

Caesar Code with Streams, Text as Strings, String Operations

Caesar-Code: Generalisation

```
void caesar(int s) {
  std::cin >> std::noskipws;

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

- Currently only from `std::cin` to `std::cout`
- Better: from arbitrary character source (console, file, ...) to arbitrary character sink (console, ...)



Caesar-Code: Generalisation

```
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

  in >> std::noskipws;

  char next;
  while (in >> next) {
    out << shift(next, s);
  }
}
```

- `std::istream`/`std::ostream` is an *generic input/output stream* of `char`s
- Function is called with *specific streams*, e.g.: Console (`std::cin`/`cout`), Files (`std::i`/`ofstream`), Strings (`std::i`/`ostringstream`)

Caesar-Code: Generalisation, Example 1

```
#include <iostream>
...

// in void main():
caesar(std::cin, std::cout, s);
```

Calling the generalised `caesar` function: from `std::cin` to `std::cout`

Caesar-Code: Generalisation, Example 2

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string from_file_name = ...; // Name of file to read from
std::string to_file_name = ...; // Name of file to write to
std::ifstream from(from_file_name); // Input file stream
std::ofstream to(to_file_name); // Output file stream

caesar(from, to, s);
```

Calling the generalised `caesar` function: from file to file

Caesar-Code: Generalisation, Example 3

```
#include <iostream>
#include <sstream>
...

// in void main():
std::string plaintext = "My password is 1234";
std::istringstream from(plaintext);

caesar(from, std::cout, s);
```

Calling the generalised `caesar` function: from a string to `std::cout`

Texts

- Text "`to be or not to be`" could be represented as `vector<char>`
- Texts are ubiquitous, however, and thus have their own typ in the standard library: `std::string`
- Requires `#include <string>`

Using `std::string`

- Declaration, and initialisation with a literal:

```
std::string text = "Essen ist fertig!"
```

- Initialise with variable length:

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

`text` is filled with n `'a'`s

- Comparing texts:

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

`true` if character-wise equal

Using `std::string`

- Querying size:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

Size not equal to text length if multi-byte encoding is used, e.g. UTF-8

- Reading single characters:

```
if (text[0] == 'a') ... // or text.at(0)
```

`text[0]` does not check index bounds, whereas `text.at(0)` does

- Writing single characters:

```
text[0] = 'b'; // or text.at(0)
```


Using `std::string`

- Concatenate strings:

```
text = ":-";
text += ")";
assert(text == ":-)");
```

- Many more operations; if interested, see
  `https://en.cppreference.com/w/cpp/string`

15. Vectors II

Multidimensional Vector/Vectors of Vectors, Shortest Paths, Vectors as Function Arguments

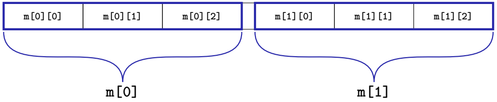
Multidimensional Vectors

- For storing multidimensional structures such as tables, matrices, ...
- ... *vectors of vectors* can be used:

```
std::vector<std::vector<int>> m; // An empty matrix
```

Multidimensional Vectors

In memory: flat



in our head: matrix

columns →, rows ↓

	0	1	2
0	m[0][0]	m[0][1]	m[0][2]
1	m[1][0]	m[1][1]	m[1][2]

Multidimensional Vectors: Initialisation Examples

Using literals[7]:

```
// A 3-by-5 matrix
std::vector<std::vector<std::string>> m = {
  {"ZH", "BE", "LU", "BS", "GE"},
  {"FR", "VD", "VS", "NE", "JU"},
  {"AR", "AI", "OW", "IW", "ZG"}
};

assert(m[1][2] == "VS");
```

[7] *initialisation lists*, actually

Multidimensional Vectors: Initialisation Examples

Fill to specific size:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
  m(a, std::vector<int>(b, 1));
```

`m` (type `std::vector<std::vector<int>>`) is a vector of length `a`, whose elements (type `std::vector<int>`) are vectors of length `b`, whose Elements (type `int`) are all ones

(Many further ways of initialising a vector exist)

Multidimensional Vectors and Type Aliases

- Also possible: vectors of vectors of vectors of ...:
  `std::vector<std::vector<std::vector<...>>>`
- Type names can obviously become looooooong
- The declaration of a *type alias* helps here:

`using` *Name* = *Typ*;

Name that can now be used to access the type

existing type

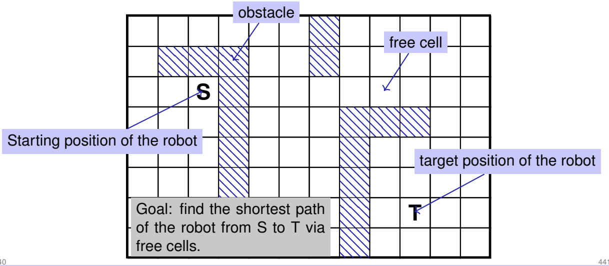
Type Aliases: Example

```
#include <iostream>
#include <vector>
using imatrix = std::vector<std::vector<int>>;

// POST: Matrix 'm' was output to stream 'out'
void print(const imatrix& m, std::ostream& out);

int main() {
  imatrix m = ...;
  print(m, std::cout);
}
```

Note: `const` reference for enfficiency (no copy) and safety (immutable)

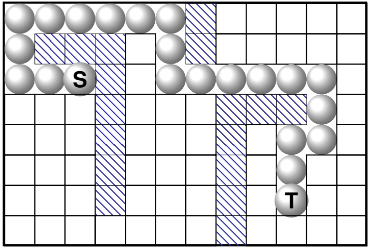
Application: Shortest Paths

Factory hall ($n \times m$ square cells)



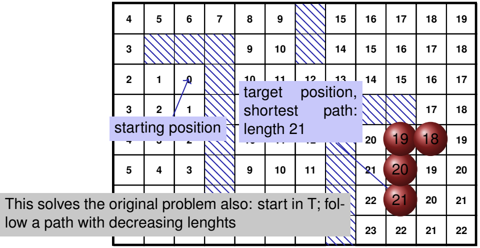
Application: shortest paths

Solution



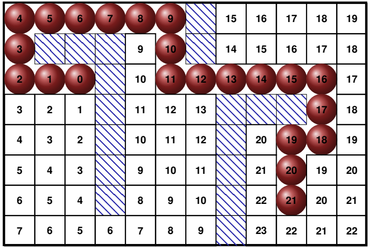
This problem appears to be different

Find the *lengths* of the shortest paths to *all* possible targets.

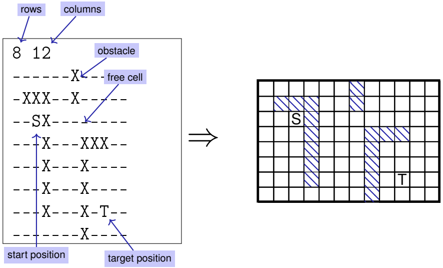


This problem appears to be different

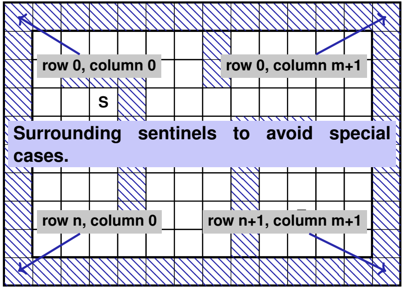
Find the *lengths* of the shortest paths to *all* possible targets.



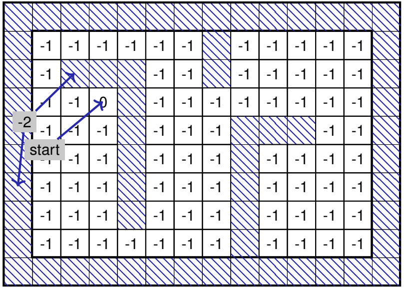
Preparation: Input Format



Preparation: Sentinels



Preparation: Initial Marking



The Shortest Path Program

- Read in dimensions and provide a two dimensional array for the path lengths

```
#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));
```

Sentinel (annotation pointing to "plus extra walls around")

The Shortest Path Program

- Input the assignment of the hall and intialize the lengths

```
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;
  }
```

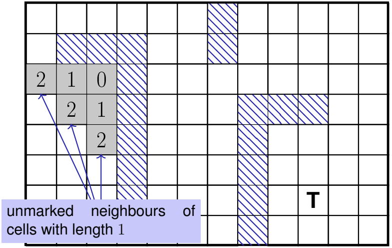

Das Kürzeste-Wege-Programm

- Add the surrounding walls

```
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;
```


Mark all Cells with their Path Lengths

Step 2: all cells with path length 2



Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

```
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;
}
```


The Shortest Paths Program

Mark the shortest path by walking backwards from target to 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)
}
```


Finish

-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

The Shortest Path Program: output

Output

```
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";
}
```

$\Rightarrow$

```
ooooooX-----
oXXX-oX-----
ooSX-oooooo-
---X---XXXo-
---X---X-oo-
---X---X-o--
---X---X-T--
-------X----
```

The Shortest Paths Program

- Algorithm: *Breadth First Search* (Breadth-first vs. depth-first is typically discussed in lectures on algorithms)
- The program can become pretty slow because for each i all cells are traversed
- Improvement: for marking with i, traverse only the neighbours of the cells marked with $i-1$.
- Improvement: stop once the goal has been reached

16. Recursion 1

Mathematical Recursion, Termination, Call Stack, Examples, Recursion vs. Iteration, n-Queen Problem, Lindenmayer Systems

Mathematical Recursion

- Many mathematical functions can be naturally defined recursively.
- This means, the function appears in its own definition

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

Recursion in $C++$: In the same Way!

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

```
// POST: return value is n!
unsigned int fac(unsigned int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```

Infinite Recursion

- is as bad as an infinite loop ...
- ...but even worse: it burns time and memory

```
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```


Recursive Functions: Termination

As with loops we need *guaranteed progress towards an exit condition (≈ base case)*

Example `fac(n)`:

- Recursion ends if $n \leq 1$
- Recursive call with new argument $< n$
- Exit condition will thus be reached eventually

```
unsigned int fac(
    unsigned int n) {

  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```


Recursive Functions: Evaluation

Example: `fac(4)`

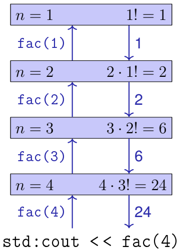
```
// POST: return value is n!
unsigned int fac(unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialization of the formal argument: $n = 4$
recursive call with argument $n - 1 == 3$

The Call Stack

For each function call:

- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



Euclidean Algorithm

- finds the greatest common divisor $\gcd(a, b)$ of two natural numbers a and b
- is based on the following mathematical recursion (proof in the lecture notes):

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Euclidean Algorithm in $\mathrm{C{+}{+}}$

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

```
unsigned int gcd(unsigned int a, unsigned int b) {
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
```

Termination: $a \bmod b < b$, thus b gets smaller in each recursive call.

Fibonacci Numbers

$$F_n := \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

Fibonacci Numbers in $\mathrm{C{+}{+}}$

Laufzeit

`fib(50)` takes “forever” because it computes F_{48} two times, F_{47} 3 times, F_{46} 5 times, F_{45} 8 times, F_{44} 13 times, F_{43} 21 times ... F_1 ca. 10^9 times (!)

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$
- Memorize the most recent two Fibonacci numbers (variables `a` and `b`)
- Compute the next number as a sum of `a` and `b`

Can be implemented recursively and iteratively, the latter is easier/more direct

Fast Fibonacci Numbers in C++

```
unsigned int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  unsigned int a = 0;  // F_0
  unsigned int b = 1;  // F_1
  for (unsigned int i = 2; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

very fast, also for `fib(50)`

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

Recursion and Iteration

Recursion can *always* be simulated by

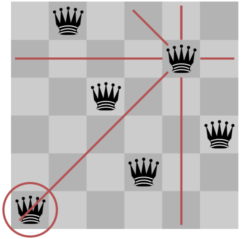
- Iteration (loops)
- explicit “call stack” (e.g. array)

Often recursive formulations are simpler, but sometimes also less efficient.

The Power of Recursion

- Some problems appear to be hard to solve without recursion. With recursion they become significantly simpler.
- Examples: *The n-Queens-Problem*, The towers of Hanoi, *Sudoku-Solver*, Expression Parsers, Reversing In- or Output, Searching in Trees, Divide-And-Conquer (e.g. sorting)

The n-Queens Problem

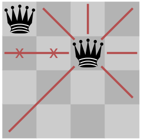


- Provided is a $n\ times n$ chessboard
- For example $n = 6$
- Question: is it possiblt to position n queens such that no two queens threaten each other?
- If yes, how many solutions are there?

Solution?

- Try all possible placements?
- $\binom{n^2}{n}$ possibilities. Too many!
- Only ne queen per row: n^n possibilities. Better – but still too many.
- Idea: don't proceed with futile attempts, retract incorrect moves instead $\Rightarrow$ *Backtracking*

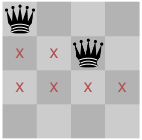
Solution with Backtracking



Second Queen in next row (no collision)

queens
0
2
0
0

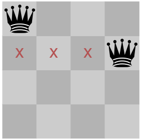
Solution with Backtracking



All squares in next row forbiden. Track back !

queens
0
2
4
0

Solution with Backtracking

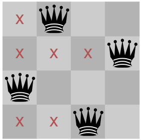


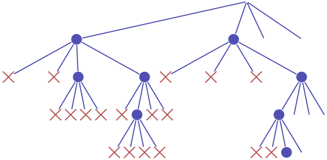
Move queen one step further and try again

queens

0
3
0
0

Search Strategy Visualized as a Tree





Check Queen

```
using Queens = std::vector<unsigned int>;

// post: returns if queen in the given row is valid, i.e.
//      does not share a common row, column or diagonal
//      with any of the queens on rows 0 to row-1
bool valid(const Queens& queens, unsigned int row) {
  unsigned int col = queens[row];
  for (unsigned int r = 0; r != row; ++r) {
    unsigned int c = queens[r];
    if (col == c || col - row == c - r || col + row == c + r)
      return false; // same column or diagonal
  }
  return true; // no shared column or diagonal
}
```


Recursion: Find a Solution

```
// pre: all queens from row 0 to row-1 are valid,
//      i.e. do not share any common row, column or diagonal
// post: returns if there is a valid position for queens on
//      row .. queens.size(). if true is returned then the
//      queens vector contains a valid configuration.
bool solve(Queens& queens, unsigned int row) {
  if (row == queens.size())
    return true;
  for (unsigned int col = 0; col != queens.size(); ++col) {
    queens[row] = col;
    if (valid(queens, row) && solve(queens,row+1))
        return true; // (else check next position)
  }
  return false; // no valid configuration found
}
```

Recursion: Count all Solutions

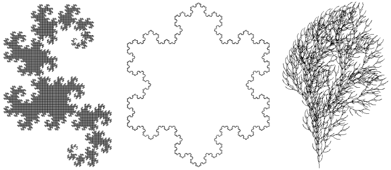
```
// pre: all queens from row 0 to row−1 are valid,
//   i.e. do not share any common row, column or diagonal
// post: returns the number of valid configurations of the
//   remaining queens on rows row ... queens.size()
int nSolutions(Queens& queens, unsigned int row) {
  if (row == queens.size())
    return 1;
  int count = 0;
  for (unsigned int col = 0; col != queens.size(); ++col) {
    queens[row] = col;
    if (valid(queens, row))
      count += nSolutions(queens,row+1);
  }
  return count;
}
```


Main Program

```
// pre: positions of the queens in vector queens
// post: output of the positions of the queens in a graphical way
void print(const Queens& queens);

int main() {
  int n;
  std::cin >> n;
  Queens queens(n);
  if (solve(queens,0)) {
    print(queens);
    std::cout << "# solutions:" << nSolutions(queens,0) << std::endl;
  } else
    std::cout << "no solution" << std::endl;
  return 0;
}
```


Lindenmayer-Systems (L-Systems)

Fractals from Strings and Turtles



- L-Systems have been invented by the Hungarian biologist Aristid Lindenmayer (1925–1989) to model the growth of plants.
- Recursion is of course relevant for the exam, but L-Systems themselves are not

Definition and Example

- alphabet Σ
- Σ^*: finite words over Σ
- production $P : \Sigma \to \Sigma^*$
- initial word $s_0 \in \Sigma^*$

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

c	$P(c)$
F	F + F +
+	+
−	−

- F

Definition

The triple $\mathcal{L} = (\Sigma, P, s_0)$ is an L-System.

The Language Described

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

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

$$\mathrm{F} + \mathrm{F} +$$

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

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

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

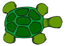
$$\vdots \qquad \vdots$$

Definition

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

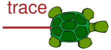
Turtle Graphics

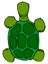
Turtle with position and direction

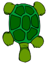


Turtle understands 3 commands:

F: move one step forwards ✓	$+$: rotate by 90 degrees ✓	$-$: rotate by -90 degrees ✓

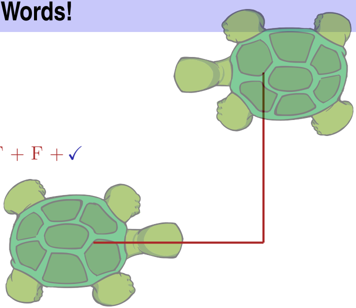






Draw Words!

$w_1 = \mathrm{F} + \mathrm{F} +$ ✓



lindenmayer: Main Program

word $w_0 \in \Sigma^*$:

```
int main() {
  std::cout << "Maximal Recursion Depth =? ";
  unsigned int n;
  std::cin >> n;

  std::string w = "F"; // w_0
  produce(w,n);

  return 0;
}
```

$w = w_0 = \mathrm{F}$

lindenmayer: production

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth) {
  if (depth > 0) {
    for (unsigned int k = 0; k < word.length(); ++k)
      produce(replace(word[k]), depth-1);
  } else {
    draw_word(word);
  }
}
```

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

draw $w = w_n$

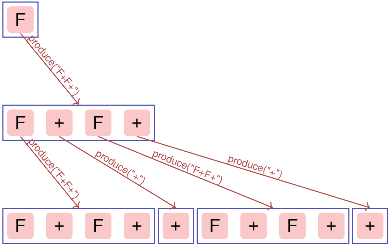
lindenmayer: replace

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


lindenmayer: draw

```
// 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':
        turtle::forward(); // move one step forward
        break;
      case '+':
        turtle::left(90); // turn counterclockwise by 90 degrees
        break;
      case '-':
        turtle::right(90); // turn clockwise by 90 degrees
    }
}
```


The Recursion



(Implementation above proceeds *depth-first*)

L-Systeme: Erweiterungen

- arbitrary symbols without graphical interpetation
- arbitrary angles (snowflake)
- saving and restoring the state of the turtle $\rightarrow$ plants (bush)