12. Arrays II

Strings, Lindenmayer Systems, Multidimensional Arrays, Vectors of Vectors, Shortest Paths, Arrays and Vectors as Function Arguments

---

**Texts**

- can be represented with the type `std::string` from the standard library.
  ```cpp
  std::string text = "bool";
  ```
  defines a string with length 4

- A string is conceptually an array with base type `char`, plus additional functionality
- Requires `#include <string>`

---

**Strings: pimped char-Arrays**

- A `std::string`...
  - knows its length
    ```cpp
    text.length()
    ```
    returns its length as `int` (call of a member function; will be explained later)
  - can be initialized with variable length
    ```cpp
    std::string text (n, 'a')
    ```
    text is filled with n 'a's
  - “understands” comparisons
    ```cpp
    if (text1 == text2) ...
    ```
    true if text1 and text2 match

---

**Lindenmayer-Systems (L-Systems)**

Fractals made from Strings and Turtles

L-Systems have been invented by the Hungarian biologist Aristid Lindenmayer (1925 – 1989) to model the growth of plants.
Definition and Example

- Alphabet $\Sigma$
- $\Sigma^*$: all finite words over $\Sigma$
- Production $P : \Sigma \to \Sigma^*$
- Initial word $s_0 \in \Sigma^*$

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

The Described Language

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

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

$$w_0 := s_0$$
$$w_1 := P(w_0)$$
$$w_2 := P(w_1)$$
$$\vdots$$

Definition
$$P(c_1c_2 \ldots c_n) := P(c_1)P(c_2) \ldots P(c_n)$$

Turtle-Graphics

Turtle with position and direction.

Turtle understands 3 commands:

- $F$: one step forward ✓
- $+$: turn by 90 degrees ✓
- $-$: turn by $-90$ degrees ✓

Draw Words!

$$w_1 = F + F + \checkmark$$
### lindenmayer.cpp: Main Program

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

```cpp
#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: next_word

// POST: replaces all symbols in word according to their production and returns the result
```cpp
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
```cpp
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
```cpp
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;
        }
}
```

### L-Systems: Extensions

- Additional symbols without graphical interpretation (dragon.cpp)
- Arbitrary angles (snowflake.cpp)
- Saving and restoring the turtle state → plants (bush.cpp)
Multidimensional Arrays

- are arrays of arrays
- can be used to store tables, matrices, ...

```
int a[2][3]
```

a contains two elements and each of them is an array of length 3 with base type int

Multidimensional Arrays

In memory: flat

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

in our head: matrix

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

Multidimensional Arrays

- are arrays of arrays of arrays ....

```
T a[expr_1] ... [expr_k]
```

a has expr_1 elements and each of them is an array with expr_2 elements each of which is an array of expr_3 elements and ...

constant expressions
Multidimensional Arrays

Initialization

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

First dimension can be omitted

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

Vectors of Vectors

- How do we get multidimensional arrays with variable dimensions?
- Solution: vectors of vectors

Example: vector of length $n$ of vectors with length $m$:

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

Application: Shortest Paths

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

Goal: find the shortest path of the robot from $S$ to $T$ via free cells.

Application: shortest paths

Solution
This problem appears to be different

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

This solves the original problem also: start in T; follow a path with decreasing lengths.

---

**Preparation: Input Format**

```
8 12
------X-----
-XXX--X-----
--SX--------
---X---XXX--
---X---X----
---X---X----
---X---X-T--
-------X----
```

⇒

```
S
T
rows columns
start position target position
obstacle
free cell
```

---

**Preparation: Sentinels**

Surrounding sentinels to avoid special cases.

```
row 0, column 0 row 0, column m+1
row n, column 0 row n+1, column m+1
S
T
```

---

This problem appears to be different

Find the *lengths* of the shortest paths to all possible targets.
The Shortest Path Program

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

```cpp
#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));
```

- Input the assignment of the hall and initialize the lengths

```cpp
    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;
        }
```

- Add the surrounding walls

```cpp
    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

![Diagram of a grid with cells marked and unmarked neighbors labeled]

**Main Loop**

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

```cpp
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.

```cpp
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**

![Diagram of a grid with cells marked]
Arrays as Function Arguments

Arrays can also be passed as reference arguments to a function. (here: const because v is read-only)

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

Arrays as Function Arguments

This also works for multidimensional arrays.

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

Vectors can be passed by value or by reference

```cpp
void print_vector(const std::vector<int>& v) {
    for (int i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
}
Here: call by reference is more efficient because the vector could be very long.
```

This also works for multidimensional vectors.

```cpp
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. Pointers, Algorithms, Iterators and Containers I

Pointers, Address operator, Dereference operator, Array-to-Pointer Conversion

Strange Things...

```cpp
#include<iostream>
#include<algorithm>
int main(){
    int a[] = {3, 2, 1, 5, 4, 6, 7};
    // output the smallest element of a
    std::cout << *std::min_element(a, a + 7);
    return 0;
}
```

We have to understand pointers first!
References: Where is Anakin?

“Search for Vader, and Anakin find you will”

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

// anakin_skywalker = 22

Pointers: Where is Anakin?

“Anakin’s address is 0x7fff6bdd1b54.”

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

// anakin_skywalker = 22

Swap with Pointers

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

Pointer Types

T* Pointer type to base type T.
An expression of type T* is called pointer (to T).
**Pointer Types**

*Value* of a pointer to T is the address of an object of type T.

**Beispiele**

int* p; Variable p is pointer to an int.
float* q; Variable q is pointer to a float.

**Address Operator**

The expression

& lval

provides, as R-value, a *pointer* of type T* to an object at the address of lval

The operator & is called *Address-Operator*.

**Dereference Operator**

The expression

*rval

returns as L-value the *value* of the object at the address represented by rval.

The operator * is called *Dereference Operator*. 
### Dereference Operator

**Beispiel**

```c
int i = 5;
int* ip = &i; // ip initialized
    // with address of i.
int j = *ip; // j == 5
```

### Address and Dereference Operators

- **Address (R-value)**: `&`
- **Object (L-value)**: `*`

### Pointer Types

- Do not point with a `double*` to an `int`!

**Examples**

```c
int* i = ...; // at address i “lives” an int...
double* j = i; //...and at j lives a double: error!
```

### Mnemonic Trick

- **The declaration**
  ```c
  T* p; p is of the type “pointer to T”
  ```
  
- **can be read as**
  ```c
  T *p; *p is of type T
  ```
  
  Although this is legal, we do not write it like this!
**Pointer Arithemtics: Pointer plus int**

- **ptr**: Pointer to element \( a[k] \) of the array \( a \) with length \( n \)
- **Value of expr**: integer \( i \) with \( 0 \leq k + i \leq n \)

\[ \text{ptr} + \text{expr} \]

is a pointer to \( a[k + i] \).

For \( k + i = n \) we get a past-the-end-pointer that must not be dereferenced.

**Pointer Arithemtics: Pointer minus int**

- If \( \text{ptr} \) is a pointer to the element with index \( k \) in an array \( a \) with length \( n \)
- and the value of \( \text{expr} \) is an integer \( i \), \( 0 \leq k - i \leq n \),

then the expression

\[ \text{ptr} - \text{expr} \]

provides a pointer to an element of \( a \) with index \( k - i \).

**Conversion Array ⇒ Pointer**

How do we get a pointer to the first element of an array?

- Static array of type \( T[n] \) is convertible to \( T* \)

**Example**

```c
int a[5] = {3, 4, 6, 1, 2};
int* begin = a; // begin points to a[0]
```

- Length information is lost ("arrays are primitive")

**Iteration over an Array of Pointers**

**Example**

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

- \( a+5 \) is a pointer behind the end of the array (past-the-end) that must not be dereferenced.
- The pointer comparison (\( p < a+5 \)) refers to the order of the two addresses in memory.