12. Arrays II

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

**Strings as Arrays**

- can be represented with underlying type `char`
  ```
  char text[] = {'b','o','o','l'}
  ```
- can also be defined as string-literals
  ```
  char text[] = "bool"
  ```
- can only be defined with constant size

**Strings: pimped char-Arrays**

A `std::string`

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

**Texts**

- can be represented with the type `std::string` from the standard library.
  ```
  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>`
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 \( P : \Sigma \rightarrow \Sigma^* \)
- Initial word \( s_0 \in \Sigma^* \)

<table>
<thead>
<tr>
<th>( c )</th>
<th>( P(c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( F + F + )</td>
</tr>
<tr>
<td>( + )</td>
<td>( + )</td>
</tr>
<tr>
<td>( - )</td>
<td>( - )</td>
</tr>
<tr>
<td>( F )</td>
<td></td>
</tr>
</tbody>
</table>

Definition

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

The Described Language

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

\[
P(F) = F + F +
\]

\[
w_0 := s_0
\]
\[
w_1 := P(w_0)
\]
\[
w_2 := P(w_1)
\]

\[
P(c_1c_2 \ldots c_n) := P(c_1)P(c_2) \ldots P(c_n)
\]

Turtle-Graphics

Turtle with position and direction.

Turtle understands 3 commands:

- \( F \): one step forward ✓
- \( + \): turn by 90 degrees ✓
- \( - \): turn by \(-90\) degrees ✓

trace
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: Main Program
```

```
lindenmayer.cpp: next_word
```
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, ....
  ```c
  int a[2][3]
  ```
  a contains two elements and each of them is an array of length 3 with base type int
Multidimensional Arrays

- are arrays of arrays of arrays ....

\[ T_{a[\text{expr}_1] \ldots [\text{expr}_k]} \]

\text{constant expressions}

\( a \) has \( \text{expr}_1 \) elements and each of them is an array with \( \text{expr}_2 \) elements each of which is an array of \( \text{expr}_3 \) elements and ...

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 \( (n \times m \text{ square cells}) \)

- Starting position of the robot
- Target position of the robot
- Obstacle
- Free cell

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

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

```
#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));
```

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
  
  ```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 showing cells with path length 2]

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)
}
```
### The Shortest Paths Program

- **Algorithm:** *Breadth First Search*
- 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 $i - 1$.

### 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.

Vectors as Function Arguments

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?

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

“Search for Vader, and Anakin find you will”

Pointers: Where is Anakin?

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

“Anakin’s address is 0x7fff6bdd1b54.”

Swap with Pointers

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

```cpp
... 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 $T$).

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

**Beispiele**

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

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

\[ \text{int value} \quad \text{adr} \quad \text{p = adr} \]
```

**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*.

**Example**

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

```c
\[ \text{ip = &i} \quad \text{i = 5} \]
```
**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.

---

**Address and Dereference Operators**

- **pointer (R-value)**
- **object (L-value)**

---

**Pointer Types**

Do not point with a double* to an int!

**Examples**

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

---

**Dereference Operator**

**Beispiel**

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

![Diagram](image-url)
Mnemonic Trick

The declaration

\[ T^* \ p; \quad \text{p is of the type “pointer to T”} \]

can be read as

\[ T \ *p; \quad \text{*p is of type T} \]

Although this is legal, we do not write it like this!

Pointer Arithmetics: Pointer plus \textit{int}

- \textit{ptr}: Pointer to element \(a[k]\) of the array \(a\) with length \(n\)
- Value of \textit{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 Arithmetics: Pointer minus \textit{int}

- If \textit{ptr} is a pointer to the element with index \(k\) in an array \(a\) with length \(n\)
- and the value of \textit{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 \(\Rightarrow\) 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

\begin{verbatim}
int a[5];
int* begin = a; // begin points to a[0]
\end{verbatim}

- Length information is lost („arrays are primitive“)
Iteration over an Array of Pointers

Example

```cpp
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.