Exercise Sheet #3

Deadline: 06.05.2024, 12:00h

Problem 1 (*Working with Arrays: Connect Four*) (10 points)

In this problem we will implement the famous game *Connect Four*. In this game two players start by choosing a color and then take turns dropping colored tokens into a six-row, seven-column vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own tokens. Your goal is to implement Connect Four in `C++`. The two colors will be represented by symbols, namely `o` and `x`. The playing field is represented as a 2D array with 6 rows (`ROWS`) and 7 columns (`COLS`). The following steps will guide you through the implementation. You can either implement your own version from scratch, or use the program skeleton given below or online https://itp.uni-frankfurt.de/~nevermann/teaching/connect_four.cpp.

(a) Implement a function

```
void print_board(char board[ROWS][COLS])
```

that takes the playing field array (`board`) as input and prints it to the console. Represent the tokens of the two players with `o` and `x`. Empty cells should be represented by a single space. Include boundaries of the playing field in your visualization. Also add column indices to simplify the gameplay.

(② points)

(b) Next, implement a function

```
void drop_piece(char board[ROWS][COLS], int col, char token)
```

that takes the `board`, a column index `col` and a `token` (`o` or `x`) as input. The function should simulate a drop of a `token` into the column with index `col`.

Make sure that the token correctly drops to the lowest available space within the column. If the column is full, display a warning to the user and simply exit out of the function.

(③ points)

(c) Finally, implement a function to check if the game is over. The function signature should be

```
bool is_game_over(char board[ROWS][COLS])
```

where the function returns a `bool` that indicates if the game is over or not. The game is over once one player forms a horizontal, vertical, or diagonal line of four of its own tokens, or if the playing field is full.

(⑤ points)

Hint: If you are using the code skeleton and did not complete part (c), comment out lines `31 - 34` (that is, prepend `//` to the beginning of the line) in the given sample code.

```
#include <iostream>

#define ROWS 6
#define COLS 7

void print_board(char board[ROWS][COLS]) {
    // Insert code for part (a)
}

void drop_piece(char board[ROWS][COLS], int col_number, char token) {
    // Insert code for part (b)
}

bool is_game_over(char board[ROWS][COLS]) {
    // Insert code for part (c)
}

int main() {
    char board[ROWS][COLS] = {
        {' ', ' ', ' ', ' ', ' ', ' ', ' '},
        {' ', ' ', ' ', ' ', ' ', ' ', ' '},
        {' ', ' ', ' ', ' ', ' ', ' ', ' '},
        {' ', ' ', ' ', ' ', ' ', ' ', ' '},
        {' ', ' ', ' ', ' ', ' ', ' ', ' '},
        {' ', ' ', ' ', ' ', ' ', ' ', ' '}
    };
    // Game loop
    bool next_is_player1 = true;
    while (true) {
        print_board(board);
        if (is_game_over(board)) {
            std::cout << "Game is over!" << std::endl;
            break;
        }
        int next_col;
        char next_token;
        if (next_is_player1) {
            std::cout << "NEXT UP: o" << std::endl;
            next_token = 'o';
        }
        else {
            std::cout << "NEXT UP: x" << std::endl;
            next_token = 'x';
        }
        std::cout << "-> Enter a column number to insert a token: ";
        std::cin >> next_col;
        drop_piece(board, next_col, next_token);
        // Alter who is next
        next_is_player1 = !next_is_player1;
    }
    return 0;
}
```

Problem 2 (*Pointers*) (10 points)

(a) Define three pointers of the type `int`, `double`, `long double` that each point to an array of the corresponding type. Print the addresses of the arrays (i.e. the values of the pointers) and the value that the pointer represents. Then increment the pointers (e.g. by `++p` or `p=p+1`) and print both the value of the pointer and the element it points. What do you find for the different data types?

(① point)

(b) With `reinterpret_cast<intptr_t>(p)`, you can convert the hexadecimal representation of a pointer `p` into decimal (for more information, see https://en.cppreference.com/w/cpp/language/reinterpret_cast and https://en.cppreference.com/w/cpp/types/integer). For the data types from part (a), reprint the decimal addresses of two consecutive elements in each array. The jumps in the address numbers should correspond to the memory needed to store these data types.

(① point)

(c) We now want to work with `C`-style strings. These can be thought of as `char` arrays and are thus declared by a pointer to the first character in the string. `C`-style strings are ended by a *null-terminator*, a special character `'\0'`. Write a program that asks the user to input a string and then stores that string as a `C`-style string. Determine its length (number of characters) using pointer arithmetics and output the result to the user.

(① point)

For the following parts provide the function definition as well as a small example and print the result to the console.

(d) Write a function

```
int myfunc(int &x, int &y)
```

that takes two integers by reference and returns their product. Rewrite the function such that it takes two pointers as arguments instead.

(② points)

(e) Write a function

```
double scalarProduct(double *arr1, double *arr2)
```

that returns the scalar product of two vectors stored in `arr1`, `arr2`.

(② points)

(f) Now implement the vector product of two vectors, therefore implement the function

```
void outerProduct(double *arr1, double *arr2, double *result)
```

where the two vectors are stored in `arr1`, `arr2` and the result is to be stored in `result`. This structure is the typical way to return arrays from functions.

(③ points)

Problem 3 (Advanced: *Sorting Algorithm*) (10 points)

A sorting algorithm is an algorithm that puts elements of a list (or an array) in a certain order, we will consider numerical order of lists of integers. The goal of this exercise is to compare different sorting algorithms for their performance.
Implement the following sorting algorithms:

- Radix Sort https://en.wikipedia.org/wiki/Radix_sort
- Bubble Sort https://en.wikipedia.org/wiki/Bubble_sort
- Another sorting algorithm of your choice https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

These algorithms can be classified by their average performance on randomly generated lists. This is important because many algorithms need a different number of steps depending on the particular set of numbers in the list. Analyze the algorithms' average performance:

- Sort $N_l = 1000$ different lists, containing n random integer numbers having up to $n_d = 4$ digits, recording the average run-time.
- Analyse the scaling of the average run-time as a function of $n \in [100, ..., 10000]$.
- Compare the scaling that you find for the different algorithms with each other and with the theoretical scaling.
- Plot and discuss these results.

Hint: You can measure the run-time within `C++` with the `chrono` library. To compile the example on the next page, you need to set the `-std=c++11` option in the `g++` compiler.

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

void f() {
        // Do some work
}

int main() {
        auto start = high_resolution_clock::now();

        f();

        auto stop = high_resolution_clock::now();
        auto dt = duration_cast<microseconds>(stop - start);

        cout << "Run-time: " << dt.count() << " microseconds"
            << endl;

        return 0;
}
```