Advanced Programming Tutorial
Worksheet 4: Resource Management

Keywords: raw pointers, `new`, `delete`, address sanitizer, `unique_ptr`, `std::map`, `struct`, `->`

Prepare!

Before coming to the tutorial, and so that the time is enough to solve the exercise, revise the slides of the lecture unit "Resource management" and answer the following questions:

- ☐ What is the difference between references and pointers?
- ☐ What do the keywords `new` and `delete` do? What is a memory leak?
- ☐ What is RAII? How do smart pointers implement RAII?
- ☐ What are some ways to define arrays of fixed and of dynamic size?
- ☐ What is an iterator and where is it used?
- ☐ What is an lvalue and what is an rvalue? What does `&&` mean?
- ☐ Check that your compiler provides address sanitizers. On Linux with GCC, `g++ -fsanitize=address` should not complain about an unrecognized option.
- ☐ Remember what you did in Worksheet 2, exercise 1 (`vis_app`). Open the skeleton file `vis_app2.cpp`: Which parts of the code can you ignore, and where will you be adding your changes?
- ☐ Read through the "some new concepts" section at the end.

Exercise 1: Matrix-vector product - A Space Odyssey

Sometimes you can't escape your legacy. There will be times that you have to deal with lower-level code. Let's explore how we could have implemented the same matrix-vector product from Worksheet 3 in 2001.

In older C++, we used to represent 2D matrices as pointers to 1D arrays, like this:

```
1 double * matrix = new double[matrix_rows * matrix_columns];
2 matrix[i*matrix_columns+j]
```

Doesn't this code smell like the old wooden desk and red leather chair it was written at?

The file `matvec.cpp` containts the solution from Worksheet 3 and an alternative, legacy-style implementation `mat_times_vec()` of the matrix-vector product we implemented in `operator*`.

1. Read through the code: what differences do you observe? Which additional keywords, types, and variables do you see?
2. Important to remember is that we always need to `delete` everything that we allocate with `new`. However, it is easy to forget something. Compile your code with GCC and use the options `-fsanitize=address` and `-g` (to enable debug symbols):
   `g++ matvec.cpp -fsanitize=address -g -o matvec`
   What additional information do you get when you run the program?

3. There is a memory leak in the program and the sanitizer tell you where. Fix it!

4. **Idea:** Can you think of a better way to organize your code so that you can more easily avoid the potential memory leaks? Think of a principle we discussed in the lecture.

Note: You are of course using Git to keep track of your exercises, right? `;-)` Since we will not go too far in this direction, you may want to make a separate git branch for this.

Exercise 2: The pandemic (tracker) gets real (data)

In Worksheet 2, you already created your first pandemic tracker. For the purpose of quick development, you used pseudo-randomly generated data. Now you want to also process real data from a CSV file[1] However, you want to keep the dummy data to be able to easily test your implementation, without changing the code.

In this exercise, we will use a `unique_ptr` to select between the dummy and the real dataset at runtime. We will also learn how to pass the `unique_ptr` to a function and we will work with `std::map` to create views of the dataset.

Open the skeleton file `caseTrack.cpp` and:

1. Take 5-10min to read through the code and the section after this exercise. There are code parts you don't need to read right now (see guiding comments).

2. Fill the `TODO` parts in `main()`:

   (a) Make the unique pointer `data_frame` so that it points to dummy or real data, depending on the `if` branch.

   (b) Print the cases for each country, independent of the kind of data (dummy or real).

3. Implement `normalize_per_capita(std::unique_ptr<DataFrame>& data_frame)`. This should return a map named `cases_normalized` with the cases per 100,000 people for each country and day.

4. Call the function `normalize_per_capita` on the given `data_frame` and print the results.

[1] European Centre for Disease Prevention and Control, file formatted for our needs.

Some new concepts

Structures: `struct`

You may already know classes: that's good! You can think of structures as simplified classes (with public members by default). You don't already know what a class is? Then you can think of a struct as a collection of data, such as:

```
struct Color {
    int red;
    int green;
    int blue;
};
```

We can then create a specific color and access the underlying elements as:

```
Color TUMBlue = {0, 101, 189};
std::cout << TUMBlue.red << "\n";
```

Arrow operator: `->`

If we access our color through a pointer (raw or smart), we first need to dereference it to access the members of it:

```
Color TUMBlue = {0, 101, 189}; // ok, normally no temporary object needed
unique_ptr<Color> myFavoriteColor = make_unique<Color>(TUMBlue);
std::cout << (*myFavoriteColor).red << "\n";
```

To make our lives easier, we can alternatively write the equivalent:

```
std::cout << myFavoriteColor->red << "\n";
```

Maps: `std::map`

A map is a data structure that maps *keys* to *values*. For example:

```
Color TUMBlue = {0, 101, 189};
Color TUMGreen = {162, 173, 0};
Color TUMLightBlue = {100, 160, 200};
std::map<std::string, Color> TUMColorPalette {
    {"color-logo", TUMBlue},
    {"color-highlight", TUMGreen},
    {"color-notes", TUMLightBlue}
}
```

To iterate over a map, we can use a ranged-for loop:

```cpp
for( const auto & [name, color] : TUMColorPalette ) {
    std::cout << "Name: " << name << ", So much red: " << color.red << "\n";
}
```

We can also directly access a map element like this:

```cpp
TUMColorPalette["color-logo"] = TUMLightBlue;
```

Important: If `"color-logo"` does not already exist in the map, then attempting to access `TUMColorPalette["color-logo"]` (without assigning anything to it) would add an empty value to this key. To check if key exists, use `map.contains(key)` (`C++20`).

Advice / C++ Core Guidelines

- R.1: Manage resources automatically using resource handles and RAII
- R.3: A raw pointer (a T*) is non-owning
- R.5: Prefer scoped objects, don't heap-allocate unnecessarily
- R.10: Avoid `malloc()` and `free()`
- R.11: Avoid calling `new` and `delete` explicitly

Epilogue

In research projects, you will still find raw pointers and explicit calls to `new` and `delete` everywhere. However, actively developed projects typically try to at least implement RAII. The easiet way to do that is using STL containers, such as `std::vector`. Note that raw pointers still have their place in your codebase, and they are often used to pass data to libraries with C-compatible API. By the way, code sanitizers are very useful, but not that widely known and used. Let's change that!