Advanced Programming Tutorial
Worksheet 5: Build Time

Keywords: header files, Make, CMake, external libraries (Eigen)

Prepare!

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

- ☐ What are the different steps, going from a C++ code to an executable?
- ☐ What are header files and what problem do they solve? What are include guards?
- ☐ Why should we not use `using namespace std` in a header file?
- ☐ What is the difference between `const` and `constexpr`?
- ☐ How does a `Makefile` look like? Where does it help us?
- ☐ Remember exercise 2 from the previous worksheet and look at the skeleton `caseTrack.cpp`: Which functions are defined in there?
- ☐ Remember exercise 1 from the previous worksheet and look at the skeleton `mymatvec.cpp`: Which functions do you wish you would have preferred to get from a library?

Disclaimers & important notes:

- We tested this worksheet on Linux. If you are using anything else, you may need to modify the instructions for your system (but please try this out beforehand). We will be using the packages GNU Make, CMake, and Eigen.
- You also need to install Eigen, e.g., using the `libeigen3-dev` package, for CMake to find it. The Eigen source files are not enough for what we want to demonstrate in this worksheet.
- Try all the steps in your terminal. Setting up Visual Studio Code would need a few more, editor-specific steps, which we want to avoid for now to reduce complexity.

Exercise 1: Housekeeping

One evening while you are working on your pandemic tracking app, your friend Dmytro comes by. He is amused by your slowly growing `caseTrack.cpp` file and suggests that perhaps it is time to break down your code into multiple files.

As we saw in the lecture, we can break up our code into header and cpp files: `.h` and `.cpp`. The header file typically contains declarations of functions or types declared through, for example, a `struct` declaration. The corresponding `.cpp` file provides implementations of these functions. In order to use these functions in a different file, we can then just include the header.

1. In your project directory, create an `src` subfolder. This is where your source code, just `caseTrack.cpp` for now, will live from now on.

2. Create the files `helper.h` and `helper.cpp` and move all functions, apart from `main`, to `helper.cpp`. Provide delcarations for these function in `helper.h`. Don't forget the *include guards* and to *include* the appropriate headers, where necessary.

   - *Important:* originally we made the vector `std::vector<std::string> countries` and the variable `num_countries` global. In order to access them from `caseTrack.cpp` now, we can make them global variables: *generally a bad idea.* However, to keep the code working as is, we need to define them in `helper.h`. For this, declare the variables as `inline`. For the rationale, see the *Some new concepts* section.
   - *Important:* Some of our functions use default arguments for files. These default arguments should only be used in declaration of the function in the header file and not repeated in the `.cpp` file. Additionally, you should consider adding a data folder under your project directory, i.e. `project/data`, and moving your `.csv` files there.

   Idea: We can break down our `caseTrack.cpp` file into multiple, better separated logical units. Here, we only wanted to show the concept, but you can imagine, for example, putting functions that operate on data, e.g., `normalize_per_capita`, into a different header than Input/Output functions like `write_to_csv`. This would follow the single responsibility principle: each header file provides one kind of functionality for your project.

3. As a sanity check, make sure that our code still compiles:

   ```
   g++ -o caseTrack caseTrack.cpp helper.cpp
   ```

   Remember that this command internally creates the object files `caseTrack.o` and `helper.o`.

4. As we add more files to our project, we would not want to type everything out by hand. Following the example in the lecture, let's write a `Makefile` to compile our code with `make`. A `Makefile` usually consists of *rules* for building *targets*. Each *target* may have certain *dependencies*. For example, the target `helper.o` depends on `helper.cpp` and we can build it with `g++ -c`. In our `Makefile`, we can then write:

   ```
   helper.o : helper.cpp helper.h
       g++ -c -o helper.o helper.cpp
   ```

Note that you need to indent with *tabs* and not spaces. Run `make` to ensure your project still compiles.

Idea: If you program in C++ for any length of time, you will encounter Makefiles everywhere. The examples here and in the lecture are deliberately verbose but you can accomplish the same thing with fewer lines than this tutorial `Makefile` using make's "pattern rules". See, for example, this tutorial on Makefiles.

5. Having cut our teeth on writing a simple `Makefile`, let us see if we can automatically generate a `Makefile`. For this, we can use CMake. CMake requires a `CMakeLists.txt` file which we can place in the `project/src` directory. In principle, this can be just three lines:

```
cmake_minimum_required(VERSION 3.8)
project(caseTrack VERSION 0.3.0 LANGUAGES CXX)
add_executable(caseTrack caseTrack.cpp helper.cpp)
target_compile_features(caseTrack PUBLIC cxx_std_17)
```

   where all the magic happens in the `add_executable` command. This adds a target executable named `caseTrack` which depends on `caseTrack.cpp` and `helper.cpp`.

6. Create a build folder i.e. `project/build` and from within it run `cmake ../src/`. On a Linux system, this will usually generate a `Makefile` in the `build` folder that we can then run with `make` to ensure everything still works.

   Idea: If you want to start hacking together a CMake configuration for your project, you may want to look at this excellent introduction to cmake.

Some New Concepts

Compilation Unit

While reading documentation, you may come across the term "Compilation Unit" or "Translation Unit". This generally means a preprocessed `.cpp` file: A `.cpp` file usually has many `#include` files. During the preprocessing step, these files are just *pasted* into the `.cpp` file. This preprocessed file, which is then passed on to the compiler, constitutes a Compilation Unit.

Inline variables

C++17 introduced a new concept called *inline variables*. With the `inline` keyword, we can define variables in multiple files without violating the one-definition rule. For more details see this article.

As an alternative we could also use `extern`. This is usually applied to a global variable (there are other use cases too). In case of a global variable, it specifies that the variable is defined in an "external" compilation unit and will have to be resolved by the linker. For more details, see this article from MSDN.

Exercise 2: Why did I spend my life writing my own linear algebra library?

Here is how research often goes... After spending a few nights developing yet another code that solves a classical problem, you find out that there are already more complete, robust, fast, and flexible libraries that offer the same functionality. One prominent example is Eigen[1].

Eigen lets us define matrices of fixed or dynamic size and of different types. For example:

```
// A dynamic size (X) matrix of doubles (d), which starts as 3x3
Eigen::MatrixXd mymatrix(3, 3);
// A fixed size 2x2 (2) matrix of integers (i)
Eigen::Matrix2i mymatrix;
// A fixed size (3) vector of floats (f)
Eigen::Vector3f myvector;
```

It also lets as set or print values very easily, using streams:

```
mymatrix << 1, 2, 3,
            4, 5, 6,
            7, 8, 9;

std::cout << mymatrix << std::endl;
```

Matrix operations then are as simple as `mymatrix * myvector`.

Since Eigen is an external library, we need to download it and then tell our compiler where to find it:

```
g++ -I path/to/eigen-version/ mycode.cpp -o mycode
```

Fortunately, Eigen is a header-only library, so we don't need to also tell the linker where to find its compiled binaries (usually with the `-l` and `-L` flags). In order to use it, we need to include the header files we need. In this case:

```
#include <Eigen/Dense>
```

This header file happens to not have a file extension (`.h`), but it is still a header file.

Learn more about Eigen following the very nice official tutorial.

Tasks

Let's adapt our solution for Worksheet 3 (or 4) to use Eigen.

1. Compute the matrix-vector product, this time using only Eigen types and operations:
   (a) Include `<Eigen/Dense>`
   (b) Replace the `std::vector<std::vector<double>>` types with `Eigen::MatrixXd` and give some elements.

[1] Eigen: `http://eigen.tuxfamily.org/`

(c) Replace the `std::vector<double>` types with `Eigen::VectorXd` and give some elements.

(d) Print the matrix-vector product.

(e) Compile as above, run, and make sure that the tests keep passing.

2. Remove all the code you don't need anymore: the custom operations and print functions. `:'-(`

3. To make our lives easier in the following, let's now install Eigen to our system, e.g., installing the `libeigen3-dev` package. Then we can focus on CMake:

   (a) Use the included[2] `CMakeLists.txt` to generate a Makefile and build your code.

   (b) We often want to generate our tests as a separate binary. For that, move your tests into a separate C++ file, `testmymatvec.cpp`. Compile it by normally calling the compiler and run it.

   (c) Look into the `CMakeLists.txt`: we have left a part of it commented-out. What does this do? Enable it, configure and build your project again[3]. Run your tests with `make test`.

Advice / C++ Core Guidelines

- SF.1: Use a `.cpp` suffix for code files and `.h` for interface files if your project doesn't already follow another convention
- SF.2: A `.h` file must not contain object definitions or non-inline function definitions
- SF.6: Use using namespace directives for transition, for foundation libraries (such as std), or within a local scope (only)
- SF.7: Don't write `using namespace` at global scope in a header file
- SF.8: Use `#include` guards for all `.h` files
- SF.10: Avoid dependencies on implicitly `#included` names
- SF.11: Header files should be self-contained
- SF.12: Prefer the quoted form of `#include` for files relative to the including file and the angle bracket form everywhere else
- An additional advice: Organize your project files to separate at least the sources from the build artifacts, e.g., in `src/` and `build/`. This will make some common operations easier, such as version control and cleaning up. You can use a consistent layout such as the Pitchfork layout to organize your files.

[2] See Using Eigen in CMake Projects

[3] Since we modified `CMakeLists.txt`, we first need to delete the file `CMakeCache.txt`.

Epilogue

Almost every C++ project we use or develop is structured into several hundreds of source files and relies on multiple dependencies. In previous years, writing a `Makefile` from scratch and modifying it for a new system was a common practice. Nowadays, we rely on build configuration tools such as CMake. On many projects we will find a `CMakeLists.txt` that might be several hundreds of lines long and sources external files as well, but the main concept is the same as with `Makefile`: CMake tries to find and configure the dependencies, source files, and build targets that we specify.

Converting from the one system to the other is typically a multiple weeks long project and needs experience that not every developer has. Fortunately, once this is set up, it is much easier for new developers to join the project. This is not yet the case for many legacy projects, however, which typically results in long periods of trying to just build the code for the first time on a new system. Companies now have full-time "DevOps" positions, with the role of making the developers' lives easier (for example by simplifying the building, testing, packaging, and distribution/deployment of software). Similarly, in academia, Research Software Engineering positions aim to help researchers write software using modern best practices.