Organizing code in C++

Lucas Frérot

Table of contents

C++ has flexible code organisation requirements that differ from some other languages. This note describes these requirements and the best practices for code organisation. We provide a skeleton repository for C++ projects and strongly recommend using it as a starting point for your code.

Compilation stages

Projects with multiple implementation files are compiled in two stages:

1. Creation of object files: each implementation file (`.cpp` or `.cc`) is compiled into an **object file**. Object files contain binary code that will be executed by the computer, as well as function call instructions. Each object file in the project is independently compiled.
2. Linking of object files: all object files are brought together into a single executable. At this point of the compilation the compiler checks that all the function calls in object files can be resolved.

These two stages of compilation make sense when thinking about avoiding unecessary compilation if only a single function is changed: only the `.cpp` file where the function is implemented will need to be recompiled.

It is very important to note that **each** `.cpp` file is **independently** compiled, meaning the compiler does not know about functions defined in *other* `.cpp` files. Giving that information to the compiler is the job of a **header** file (`.hh` or `.h`). Header files typically *do not contain implementation details*, they only give the **function signature**, or function definition. Each function that is called in a `.cpp` file needs to have been declared, in that file, either explicitly or in an included header file with `#include "header.hh"`. If a function signature is missing the compiler will raise an error.

In the second compilation stage (linking), the compiler needs to find *all the function implementations* (and needs to find exactly one implementation per function). If an object file is missing, the linker will raise an error about an undefined symbol.

Organizing Code

Knowing these requirements, one has a lot of freedom to organize code. For example, all the function definitions could be in a single header file, with each function implemented in a separate implementation file. However, the most common way to structure code is to divide into *logical units*. Functions and classes that logically belong together should be together in a single implementation file and single header file. For example, the file `atoms.hh` contains the class `Atoms`, and `atoms.cpp` contains the implementation of its member functions. To use the `Atoms` class in a `.cpp` file, one needs then to add `#include "atoms.hh"`.

Example with manual compilation

To get a feel of how this works, let us consider a function `square` that returns the square of a `double`. This function should be implemented in the file `square.cpp`:

```
// square.cpp

#include "square.hh"

double square(double x) {
    return x * x;
}
```

Next we need to provide the *function signature* in the header file `square.hh`:

```
// square.hh
#ifndef SQUARE_HH
#define SQUARE_HH

double square(double x);

#endif
```

Note the use of an *include guard* (the couple `#ifndef/#define`), which prevents the content of the file to be duplicated if it is included twice, which causes compile errors. Finally, we use the function in `main()`:

```
// main.cpp

#include "square.hh"

int main() {
    square(3);
    return 0;
}
```

To compile, we first generate the object files:

```
g++ -c square.cpp -o square.o
g++ -c main.cpp -o main.o
```

The `-c` flag indicates that we want to create an object file. Then we can link an executable:

```
g++ main.o square.o -o square_example
```

Try removing `square.o` in the above command and see what error is raised. Try removing `#include "square.hh"` and recompile `main.cpp` to see what error is raised.

You can see the effect of the `#include` directive by passing `-E` to the compiler, which only applies preprocessor directives like `#include`:

```
g++ -E main.cpp main.E
```

You'll see that the entire content of `square.hh` was duplicated into `main.E`. The whole content of `main.E` is called a **translation unit**.

Using CMake

CMake handles the two compilation stages transparently, so that the following `CMakeLists.txt` should create the correct executable:

```
cmake_minimum_required(VERSION 3.21)

project(
   compilation_example
   VERSION 0.1
   LANGUAGES CXX
)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(
   compilation_example
   main.cpp
)
```

Exceptions to the header/implementation rules

Two kinds of functions *must* be implemented in header files:

1. **Inline functions**: the `inline` keyword instructs the compiler to not create a function call but instead put the function implementation *in-place*. This is useful for performance reasons, but means the compiler must have access to the implementation, which must be in the header.
2. Functions with signatures that must be deduced by the compiler. These include function templates and functions with `auto` or `decltype(auto)` as a return type. The compiler needs the full knowledge of the function to deduce the information it is missing. Implementation must therefore be in the header.

Note that regular functions that would be normally implemented in `.cpp` files can be implemented in header files *if and only if* they are marked as `inline`, otherwise the compiler will create multiple symbols for the same function and linking will fail.

We will use both inline functions and template classes/functions, but the implementations will be provided in the right files.