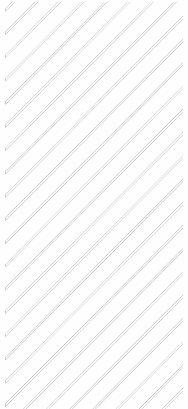
Lecture 12

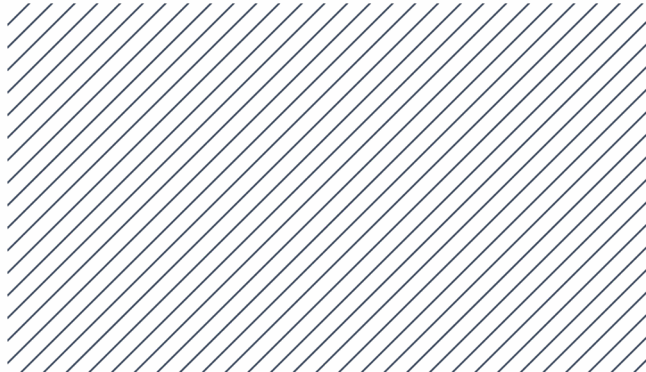
Classes and structures, pt.4

Konstantin Leladze

OOP in C++

DIHT MIPT 2021





Splitting the solution into multiple files

C++ has a common project structure. For now, I'll just teach you how to split your solution into several files and organize the structure correctly without going into details.

Let's take the **vector** class we wrote in the previous lesson.

So far, our entire solution is written in one file - **main.cpp**. So far, this does not cause problems, since there is little code, but on an industrial scale, the amount of code in the repository can be several million lines of code, so it is important to be able to split the solution into separate components, and the files corresponding to them.

For example, our code now has three parts:

- The double_cmp function, which compares doubles, separate from the **vector** class
- The **vector** class, which is a vector in three-dimensional space, and the operators redefined to work with it
- The main function

Splitting the solution into multiple files

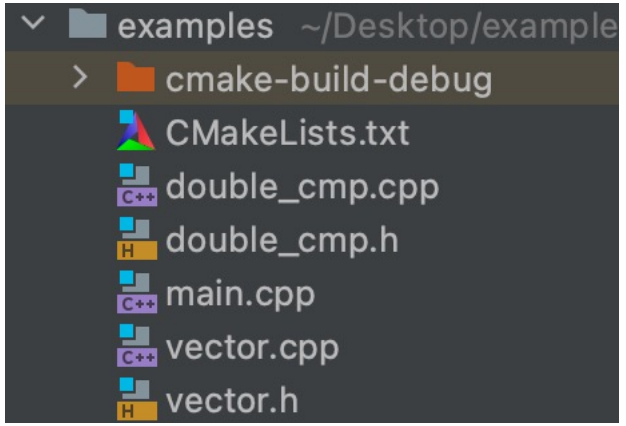
Each component, which is, for example, a class (just like in our case) must have two files: .h and .cpp

- In the .h files, we **declare** functions, classes and their methods, but we don't **define** them!
- In the .cpp files, we **define** functions and methods of the class

Both files must have the same name, but have different extensions. The name of the files must coincide with the name of the class, if the component is a class, or, otherwise, when the component contains several classes, generalize their essence.

For example, our project can be divided into 2 components:
- vector: will contain class vector
- double_cmp: will contain function double_cmp

The project structure is shown on the right:



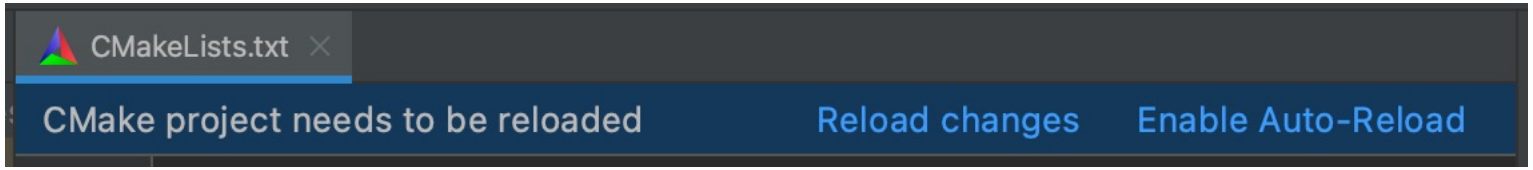
Splitting the solution into multiple files

After adding the files to the project, you will also have to add the file names to **CMakeLists.txt**

CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.19)
2  project(examples)
3
4  set(CMAKE_CXX_STANDARD 20)
5
6  add_executable(examples main.cpp vector.cpp vector.h double_cmp.cpp double_cmp.h)
7
```

Also, if you are working in CLion, after changing CMakeLists, you will be prompted to reload project. Obviously, this must be done in order for the compiler to update the project structure.



Splitting the solution into multiple files

double_cmp.h

```
1   #ifndef EXAMPLES_DOUBLE_CMP_H
2   #define EXAMPLES_DOUBLE_CMP_H
3
4
5
6   bool double_cmp (double first, double second, double eps = 1e-6);
7
8
9   #endif /// EXAMPLES_DOUBLE_CMP_H.
10
```

- In double_cmp.h file, we declare a function double_cmp.
- Note that the arguments in the function declaration are not const-qualified, because const-qualifier is pointless for unmodified type in declarations. In other words, const qualifier of unmodified type (non-pointer, non-array and non-reference type) is only valid in definitions.
- **EXAMPLES_DOUBLE_CMP_H** – is a “full name” of the file, which compiler will use in order to avoid including one file multiple times.
  - This name should be unique for every header file in your project.
  - Usually, this name is built according to the following principle: **{project_name}_{path_to_file}**.
  - For example, if my project is called geometry, and the file path is **lib/figures/polygon.h**, then define should be: **GEOMETRY_LIB_FIGURES_POLYGON_H**

Splitting the solution into multiple files

double_cmp.cpp

```cpp
#include "double_cmp.h"

#include <cmath>



bool double_cmp (const double first, const double second, const double eps) {
    return std::fabs(first - second) < eps;
}
```

- In double_cmp.cpp file, we define a function double_cmp.
- In the definition we add const-qualifier, but we do not assign the default value for the **eps** argument (default value should be assigned in the declaration)
- It's mandatory to include the corresponding **header file** in the **source file**.
- But the opposite should never be done! In other words, never do an include of a .cpp file in your code!

Splitting the solution into multiple files

vector.h

```
#ifndef EXAMPLES_VECTOR_H
#define EXAMPLES_VECTOR_H

#include <iostream>


struct vector {
    vector ();
    vector (double x, double y, double z);
    vector (const vector &other);
    vector &operator= (const vector &other);
    ~vector ();

    vector& operator+= (const vector& other);
    vector& operator-= (const vector& other);
    vector& operator*= (double coefficient);
    vector& operator*= (const vector& other);

    const vector operator- () const;
    const vector operator+ () const;

    double x, y, z;
};

const vector operator+ (const vector& first, const vector& second);
const vector operator- (const vector& first, const vector& second);
const vector operator* (const vector& first, const vector& second);
const vector operator* (const vector& first, double second);
const vector operator* (double first, const vector& second);

bool operator== (const vector& first, const vector& second);
bool operator!= (const vector& first, const vector& second);

std::istream& operator>> (std::istream& is, vector& instance);
std::ostream& operator<< (std::ostream& os, const vector& instance);


#endif /// EXAMPLES_VECTOR_H.
```

- In the vector.h file, we only provide function declarations, even if they are methods (defined inside the class)!
- Also, don’t forget about #define
- Once again, const qualifiers are absent in functions signatures for the arguments, provided by values

Splitting the solution into multiple files

vector.cpp

```cpp
#include "double_cmp.h"
#include "vector.h"

#include <iostream>



vector::vector () : vector(0, 0, 0) {}
vector::vector (const double x, const double y, const double z) : x(x), y(y), z(z) {}
vector::vector (const vector &other) = default;
vector& vector::operator= (const vector &other) = default;
vector::~vector () = default;
```

- In vector.cpp we define methods and functions which are declared in vector.h
- Don't forget **#include "vector.h"**”
- Using the **operator ::**, we can define methods
  - For this, before the method name, you need to add the class name and the **operator ::**
  - In the case of non-method **functions**, we don't need to do this!
  - For example, if the class is called **my_class**, and the method in it is **my_method**, then its definition might look like this: **void my_class::my_method () {}**

Splitting the solution into multiple files

Here’s some more definitions of several methods:

vector.cpp

```
14  vector& vector::operator+= (const vector& other) {
15      x += other.x;
16      y += other.y;
17      z += other.z;
18      return *this;
19  }
20
21  vector& vector::operator-= (const vector& other) {
22      x -= other.x;
23      y -= other.y;
24      z -= other.z;
25      return *this;
26  }
```

vector.cpp

```
28  vector& vector::operator*= (const double coefficient) {
29      x *= coefficient;
30      y *= coefficient;
31      z *= coefficient;
32      return *this;
33  }
34
35  vector& vector::operator*= (const vector& other) {
36      const double new_x = y * other.z - z * other.y;
37      const double new_y = z * other.x - x * other.z;
38      const double new_z = x * other.y - y * other.x;
39      x = new_x;
40      y = new_y;
41      z = new_z;
42      return *this;
43  }
44
45  const vector vector::operator- () const { return vector(-x, -y, -z); }
46  const vector vector::operator+ () const { return vector(-x, y, z); }
```

Splitting the solution into multiple files

- And finally, the definition of ordinary functions.
- By the way, note that in order to use the **double_cmp** function, we included file **double_cmp.h** in the **vector.cpp** file (but not in **vector.h**!)

```
48  const vector operator+ (const vector& first, const vector& second) { return vector(first) += second; }
49  const vector operator- (const vector& first, const vector& second) { return vector(first) -= second; }
50  const vector operator* (const vector& first, const vector& second) { return vector(first) *= second; }
51  const vector operator* (const vector& first, const double second) { return vector(first) *= second; }
52  const vector operator* (const double first, const vector& second) { return vector(second) *= first; }
53
54  bool operator== (const vector& first, const vector& second) {
55      return
56          double_cmp(first.x, second.x) &&
57          double_cmp(first.y, second.y) &&
58          double_cmp(first.z, second.z);
59  }
60
61  bool operator!= (const vector& first, const vector& second) {
62      return !(first == second);
63  }
64
65  std::istream& operator>> (std::istream& is, vector& instance) {
66      return is >> instance.x >> instance.y >> instance.z;
67  }
68
69  std::ostream& operator<< (std::ostream& os, const vector& instance) {
70      return os << "(" << instance.x << ", " << instance.y << ", " << instance.z << ")";
71  }
```

Splitting the solution into multiple files

To use our **vector** class in the **main** function, you need to include the corresponding header in the **main.cpp** file.

main.cpp

```cpp
#include <iostream>

#include "vector.h"



int main () {
    vector first, second;
    std::cin >> first;
    std::cin >> second;
    std::cout << (first + second) << std::endl;
}

```

Splitting the solution into multiple files

Links to the full code:

CMakeLists.txt
double_cmp.h
double_cmp.cpp
vector.h
vector.cpp
main.cpp

Lecture 12

Classes and structures, pt.4

Konstantin Leladze

OOP in C++

DIHT MIPT 2021