1) How to setup a class

For each cpp class there should be made a separate header and source file.
The header files should look something like this:

cpp > include > C hello_world.h > HelloWorld

```
1   #ifndef HELLO_WORLD_H
2   #define HELLO_WORLD_H
3
4   #include <iostream>
5   #include <string>
6
7   class HelloWorld{
8       private:
9           std::string _hello;
10
11      public:
12          HelloWorld();
13          void print();
14
15  };
16
17  #endif
```

Here private variables are marked with a _ in front.
Class name is camel case which means each word has a capital letter, and the words are concatenated. Like this CamelCase. Each method name should be written with snake case, like this snake_case, where there are no capital letters, snake_case should also be used for variables.

The source files should look something like:

cpp > src > hello_world.cpp > print()

```
1   #include "hello_world.h"
2
3   HelloWorld::HelloWorld(): _hello("Hello World!"){}
4
5   void HelloWorld::print(){
6       std::cout << _hello << std::endl;
7   }
```

The only includes in the source file is the header file, and the header file has all other includes.

2) A method does one thing and one thing good

When written a method it should not do a lot of things, it should do one thing and that thing very well.

This is an example of a function that does two things:

```cpp
void HelloWorld::sort_and_print(std::vector<int>& nums) {
    std::sort(nums.begin(), nums.end());

    std::cout << "Sorted numbers: ";
    for (const auto& num : nums) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
```

This should be split up into two function each having one responsibility.

```cpp
void sort_numbers(std::vector<int>& nums) {
    std::sort(nums.begin(), nums.end());
}

void print_numbers(const std::vector<int>& nums) {
    std::cout << "Numbers: ";
    for (const auto& num : nums) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
```

Doing so separated the responsibility of each element of a class, and makes every element more reusable.

3) No magic numbers

There should never be found doubt for why certain numbers are used, this means that all magic numbers that needs to be there for reasons, these reasons should be clear. Therefore, no magic numbers should float around and therefore give them variables names or whole methods, so the code is more reader friendly.

Here is an example with magic number:

```
void HelloWorld::send_data(int data){
    delay(0.3);
    uart(data);
    delay(0.3);
    uart(data+99);
}
```

Here both the 0.3 in the delay and the +99 are numbers with no explanation for why they are there, this can be fixed in two ways, one way is the name the numbers:

```
void HelloWorld::send_data(int data){
    int wait_for_uart = 0.3;
    int rotate_gripper_back = 99;

    delay(wait_for_uart);
    uart(data);
    delay(wait_for_uart);
    uart(data+rotate_gripper_back);
}
```

They other is to make separate methods for each element, thereby splitting responsibility and making error handling easier.

```
void HelloWorld::send_data(int data){
    int wait_for_uart = 0.3;
    int rotate_gripper_back = 99;

    this-> wait_for_uart();
    uart(data);
    this-> wait_for_uart();
    this-> rotate_gripper_back(data);
}

void HelloWorld::wait_for_uart(){
    int data_bus_clear = 0.3;
    delay(data_bus_clear);
}
void HelloWorld::rotate_gripper_back(int data){
    int rotate_gripper_back = 99;
    uart(data+rotate_gripper_back);
}
```

4) NO ABBREVIATIONS!

I cannot stress this enough, there should never be abbreviations.

Something like this lPs should just be written as lego_position_space:

```
void lPs();                    void lego_position_space();
```

Abbreviations are never self-explanatory and can very easily make confusion about what the method or variable is doing.

Therefore, these needs to be eliminated before they are created, they are bad habits.

5) Too many comments mean code needs to be rewritten to make sense

There should never be a need for a lot of comments, comments do not need to explain obvious thing like, this is an example of how NOT to write comments.

```
void HelloWorld::print_numbers(const std::vector<int>& nums) {
    //Printing the sorted numbers
    std::cout << "Sorted numbers: ";
    //Range-based for loop for printing the numbers
    for (const auto& num : nums) {
        //Printing the number and adding spaces
        std::cout << num << " ";
    }
    //Printing a new line
    std::cout << std::endl;
}
```

Comments need to explain things that are not obvious, if some part of the code isn't very easily understood from the context of the code, comments can help understand the code. This is useful if the code is created from a deep understanding of a subject that is not easily understood if the knowledge of the subject is missing.

6) Using Doxygen instead of comments

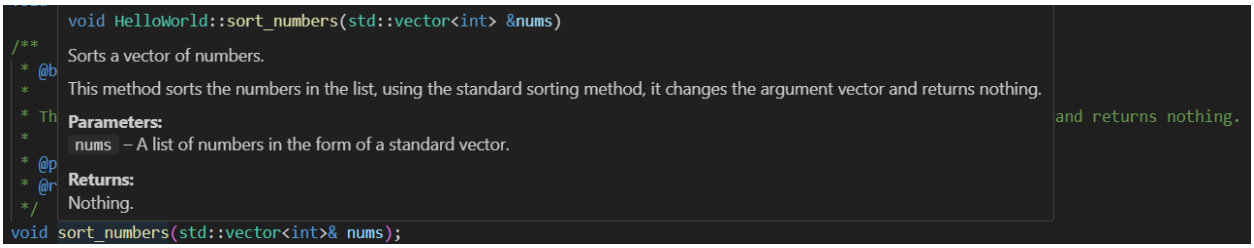
Using Doxygen is a very useful tool instead of using comments to explain your code, Doxygen makes possible to hover your methods and get a brief of what it does.

The syntax setup for Doxygen is like this:

```
/**
 * @brief Sorts a vector of numbers.
 *
 * This method sorts the numbers in the list, using the standard sorting method, it changes the argument vector and returns nothing.
 *
 * @param nums A list of numbers in the form of a standard vector.
 * @return Nothing.
 */
void sort_numbers(std::vector<int>& nums);
```

This should be done in the header file, right above the method for which it is made for.

When hovering the method is will display the information like this:



The QT-creator supports this by standard, if the VS-Code is used, an extension is needed.

7) How to use try catch error handling

When using error handling there should never be a time where you try to catch standard non expected errors, you should try to catch errors that you expect, this means only to catch errors you are aware exist.

This method tries to catch std::exception, this is the standard base class for all exceptions, this will then catch all exception that can be thrown by the uart method and rotate_gripper_back.

```
void HelloWorld::send_data(int data){
    this-> wait_for_uart();
    try{
        uart(data);
    }
    catch (std::exception& e){
        std::cout << "Error: " << e.what() << std::endl;
    }
    this-> wait_for_uart();
    try{
        this-> rotate_gripper_back(data);
    }
    catch (std::exception& e){
        std::cout << "Error: " << e.what() << std::endl;
    }
}
```

One way to fix this is overloading the exception class with your own exception and only try to catch that exception, if the code keeps giving other exceptions the code needs to be fixed because then its bad code.

The class now looks like this:

```
class HelloWorld{
    public:
        class uart_exception : public std::runtime_error {
            public:
                uart_exception(std::string error) : std::runtime_error(error) {}
                virtual const char* what() const throw() {
                    return "UART error";
                }
        };
        HelloWorld();
        void print();
        /**
         * @brief Sorts a vector of numbers.
         *
         * This method sorts the numbers in the list, using the standard sorting method, it changes the argument vector and returns nothing.
         *
         * @param nums A list of numbers in the form of a standard vector.
         * @return Nothing.
         */
        void sort_numbers(std::vector<int>& nums);
        void uart(const char* data);
```

Where we have made a public class inside, that contains the errors for that class:

Now we only throw our own exceptions where we expect the code to fail:

```cpp
void HelloWorld::uart(const char* data) {
    if (data == nullptr) {
        // Throw a runtime error if data is null
        throw uart_exception("Data cannot be null.");
    }

    for (size_t i = 0; i < strlen(data); i++) {
        if (printf("Transmitting: %c\n", data[i]) < 0) {
            // Simulate an error during transmission
            throw uart_exception("Failed to transmit data.");
        }
    }
}
```

Now we know where and what went wrong in the code and can handle it appropriately.