Contents

CPP structure

How to write a class:

Each class should be properly divided into separate source and header file, the structure of the header file should be as follows:

```cpp
#ifndef FILE_NAME_H
#define FILE_NAME_H

#include <iostream>

class MyClass{
    private:
        int _private_member;

        void _private_func(std::string &argument);

    public:
        int public_member;

        void public_func(std::string &argument);
};

#endif // FILE_NAME_H
```

And the source file should be on the following format:

```cpp
#include "file_name.h"


void MyClass::_private_func(std::string &argument){
    /*
    a private function can only be accessed inside the class
    Therefore they should be used as helper function to a public function.
    */
    argument = "Something cool";
    _private_member = 10;
}

void MyClass::public_func(std::string &argument){
    /*
    A public function is something you call when using instances of the class.
    They can run multiple private helper functions
    */
    argument = "Something cool";
    public_member = 20;
}
```

Here the important takeaway is that every include happens in the header file, only the header files with relevant declarations should be included in the source file. a

How to store data:

Storing an arranging data is crucial for the readability of the code, therefore having a clear structure and organized data helps a ton.

Some helpful tools for this, are classes, structs and namespaces, these can all be used to store data in a readable format. Furthermore **Enums** can be used to give a large set of data, meaningful names, this is useful because if we have a logger that logs data, and there is levels to how critical the logged data is, giving them number like log levels 1, 2, 3, 4, 5 is very unclear and vague, WARNING, CRITICAL, STABIL, ERROR gives a lot more insight into the meaning.

Why not just have vectors with data and variables in a class? Well sometimes it can hard to remember where in the vector you stored something, or it can be a hard to get an overview or manage a lot of variables in a class.

Enums:

They make a concise large list of “definitions”, they would look like this:

```
enum InfoLevel {
    CRITICAL,
    WARNING,
    STABIL,
    ERROR
};
```

Here the default value is 0, 1, 2, 3 but it is possible to give them a value →

```
enum InfoLevel {
    CRITICAL =  -2,
    WARNING = -1,
    STABIL = 0,
    ERROR = 1
};
```

This is useful for making a bunch of definitions, for errors, package types, important to note is that you don’t need to ever call the **enum** name, the **enum** name InfoLevel is only there for readability, so everyone knows what the context is.

Structs:

Structs are a very good way of storing a lot of variables and information is a concise matter, **struct** should in general be very simple, don’t make them complex like a class. Structs are useful for storing data that’s not necessarily the same type:

```
struct Info {
    int level;
    std::string message;
};
```

Here we could have simple methods that make sense for the **struct**:

```
struct Info {
    int level;
    std::string message;

    void print_info(){
        std::cout << "Level: " << level << ", Message: " << message << std::endl;
    }
};
```

Here we give the information inside the **struct** names, and we can orderly store it in the same object.

A class could be use in the same way, but doing it with structs makes it clear that its not a complicated objects but instead a collection of data.

Here is an example of how to use enums and structs:

```
int main() {
    Info info;

    info.level = WARNING;
    info.message = "This is a warning message.";

    info.print_info();

    return 0;
}
```

We make a instance of our struct, then we set the info level inside the struct to warning, which is the same as setting it to -2, then we set the message and we can even call the primitive print function. This makes it very clear and easy what data we are dealing with.

Namespaces:

Namespaces are a way to group classes, functions e.g. under a shared collection name. This is partially to avoid confusing different libraries if they have some class or function names that are identical. They can also be used to group data like variables, functions and so on under the same namespace / collection.

```
namespace name {
    void displayMessage() {
        std::cout << "Hello from MyNamespace!" << std::endl;
    }

    int myVar = 10;

    class MyClass {
    public:
        void show() {
            std::cout << "This is MyClass inside MyNamespace." << std::endl;
        }
    };
}

int main(){
    name::displayMessage();
    name::myVar = 20;
    name::MyClass some_name;
    some_name.show();
}
```

Here is a namespace and how to use it, the namespace includes both classes, function and variables. Also a useful way of making a collection of variables and data.

Template class

A **template class** in C++ is a blueprint for creating generic classes or functions. It allows you to write a single class or function that works with different data types.

Make the code very reusable and flexible.

```cpp
template <typename T>
class MyTempClass {
private:
    T _private_member;

public:
    MyTempClass(T val) : _private_member(val) {}
};
```

Here the T symbolizes any data type int, float, string, bool or even a custom data type like, MyClass or Info. The importance is that every function can use and operate on every data, type, or at least have good error handling. These types of classes are very useful for making stuff like Matrix objects or math vectors.

Not entirely sure they can be split into Header and Source files; therefore, they are accepted to have definition and declaration in the header file only. This is because when they are called in your code, the compiler needs their full implementation

Error handling:

There are a few ways to handle error, one is to return something from methods that you can conclude on whether it was successful, the other is to raise exceptions.

Make return statements

Here the general idea is to either create a bool, string or other type of variables, that you will modify and then in the end return so I will hold some value that you can cross check if it was successful or not. The no brainer here it to return a bool,

```cpp
bool error_function(){
    bool success = true;

    for(;;){
        // do something
        int oh_no = 1;
        if (oh_no == 1){
            success = false;
        }
        break;
    }
    return success;
}
```

Here we return a bool that gets altered if something goes wrong, then we can look at the output from the function and check if it was successful.

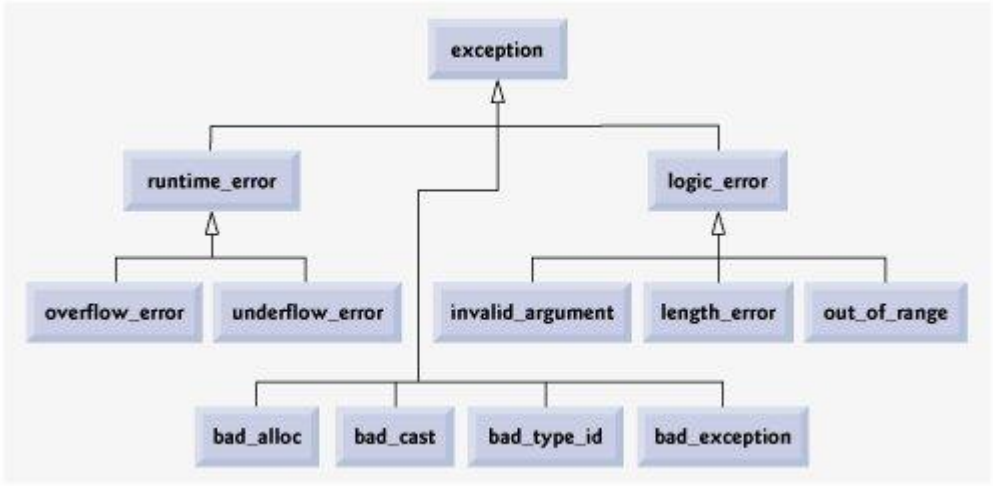
Raise exceptions

This is a more sophisticated one, where it is easy to do In a bad manner. The idea here is to throw an exception when an error occur then I will skip all the code that doesn't involve catching the error, and as soon as a catch is reached it will execute the code in the catch section. Here we should be careful with trying to catch errors and try to only catch errors we have thrown our self, so making custom exceptions are preferable.

```cpp
#include <exception>

class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "My custom exception occurred!";
    }
};
```

This is the general structure to make a custom exceptions, try to inherit from an exceptions that makes sense in the context of the custom error, so when making a custom division method, inherit from invalid_argument to make a dividing with zero exception would make sense.



Preprocessor directives:

The use of preprocessor directives:

These can be used to makes defines or even make defines based on conditions, this could be based on the system its building to or building on.

Common preprocessor directives:

#include <>	#include “”		
#if	#elif	#else	#endif
#ifndef	#endif		
#define	#undef		
#warning			

Usage of defines:

The more local the defines are made, the better, but defines should be things that have a unique name and should make sense to be a define.

CPP convention

Naming convention

Defines should always be defines in the beginning of the file, or at the start of the section in which it is being used. Defines should always be all caps:

#define MAX_VAL = 100

Classes should always be pascal case where the words are concatenated but every word start with a capital letter:

MyClass

Function and variables should always be snake case:

My_func my_variable

When creating global variables (which you ideally never did), they should be expressed with a g_ in front

g_my_var

private variables should have a _ in front of them.

```
#define MAX 200             // Define all caps
int _private_member;        // private member with an _
int my_variable;            // normal variable
MyClass instance;           // class with capital beginning
int g_global_variable;      // global with a g_ in front
```

Loops

Do not write loops without a clear body:

```
while(true);

for(int i = 0; i < MAX; i++);

// Giving them a clear body

while(true)
{
    // body
}

for(int i = 0; i < MAX; i++)
{
    // body
}
```

Here the for loop is running with no body, but its betty to explicitly say that the loop should do nothing than to leave out the body of the loop.

Variables

Use of auto:

Never use auto, auto makes it very hard to understand the meaning of a function when its hidden what return parameter is has. Auto can be used when the type is exceptionally long and very clear what type it is.

```
std::chrono::high_resolution_clock::time_point start = std::chrono::high_resolution_clock::now();
// here the std::chrono::high_resolution_clock::time_point is not that important of a type
// and the use of start is very clear from the context
auto end = std::chrono::high_resolution_clock::now();
```

When used instead of simple vectors, it would be nicer to know what the vector consist of.

Use of abbreviations:

Never use abbreviations, or at least only use them for very common abbreviations, don't expect someone to look it up or to go into your header file to see the abbreviation in a comment.

Use of global variables:

Avoid using global variables since these can be changes everywhere in the code and the more complex the code gets, the harder it gets to track where they are changed and when. Makes debugging a lot harder, and can introduce some unwanted errors, when variables have a different value than expected. Use of defines, static variables or structs and reference passing is all

Using Doxygen

Using doxygen is very useful for documentation and readability, it makes it possible to make a description of a method so it is easy to figure out what the method does, what parameters it takes and what it eventually return.

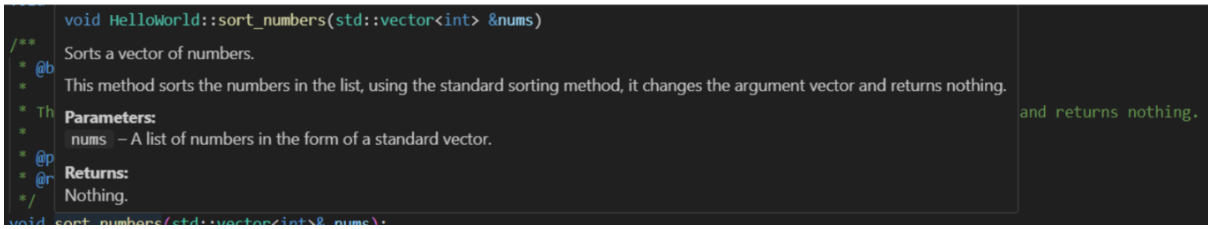
There are two methods to making them, they are made using comments and should be hoovering over a method/function in their respective header file.

```
/**
 * @brief Method description, shortly about what it do
 * @param argument Type: argument type - short description of what it do
 * @return Type: return type - short description of what the return represents
 */
void displayMessage() {
    std::cout << "Hello from MyNamespace!" << std::endl;
}
```

Both are valid, and will make it so when ever the function is hoovered in the code it will display the brief, param and return like in the third picture.

```
/// @brief Method description, shortly about what it do
/// @param argument Type: argument type - short description of what it do
/// @return Type: return type - short description of what the return represents
bool error_function(){
    bool success = true;

    for(;;){
        // do something
        int oh_no = 1;
        if (oh_no == 1){
            success = false;
        }
        break;
    }
    return success;
}
```



For it to be used in VSCode like the examples, an extension is needed, either doxygen or a IntelliSense which comes with C/C++ extensions.

Furthermore, if Doxygen is used properly it can make a HTML based page which can be used af documentation for the code.