Linux and Open Source

C++ good practises

Use smart pointers

- Avoid using *new* without RAII wrapper like smart pointer
  - Prefer std::unique_ptr over std::shared_ptr
  - **Do not overuse std::shared_ptr**
    - Most of the cases can be solved with std::unique_ptr, typical use-case for std::shared_ptr is multithreading

Avoid using *new* at all

- When working on desktop applications, where stack size is not an important limitation, prefer allocating objects on stack over heap

Do not return pointers to locally created objects

```
struct User
{
	std::string name;
	std::string id;
};

User* getUser() {
	User u;
	return &u;
}
```

Avoid dependency hell

- Avoid circular dependencies

a.h

```
#include “b.h”

class A {
public:
    A(B* b) : m_b(b){}
private:
    B* m_b = nullptr;
}
```

b.h

```
#include “a.h”

class B {
public:
    B(A* a) : m_a(a){}
private:
    A* m_a = nullptr;
}
```

Avoid dependency hell

- Avoid circular dependencies
  - Consider dependency via abstraction
  - Use forward declaration and pImpl idiom

Avoid dependency hell

- Avoid circular dependencies
    - Consider dependency via abstraction
    - Use forward declaration and pImpl idiom

Avoid dependency hell

- Avoid circular dependencies
  - Consider dependency via abstraction
  - Use forward declaration and pImpl idiom
- Define interfaces *(abstractions)* in separate header file

Avoid dependency hell

- Avoid circular dependencies
    - Consider dependency via abstraction
    - Use forward declaration and pImpl idiom
- Define interfaces *(abstractions)* in separate header file
- Limit includes in header files to bare minimum, headers needed for implementation include in *cpp* file

Avoid dependency hell

- Avoid circular dependencies
  - Consider dependency via abstraction
  - Use forward declaration and pImpl idiom
- Define interfaces *(abstractions)* in separate header file
- Limit includes in header files to bare minimum, headers needed for implementation include in *cpp* file
- When working on modules boundaries, prefer copies of DTO’s over incorrect dependencies

Avoid dependency hell

- Avoid circular dependencies
  - Consider dependency via abstraction
  - Use forward declaration and pImpl idiom
- Define interfaces *(abstractions)* in separate header file
- Limit includes in header files to bare minimum, headers needed for implementation include in *cpp* file
- When working on modules boundaries, prefer copies of DTO's over incorrect dependencies
- Avoid depending on IO/OS modules directly, introduce reasonable abstractions instead

Avoid dependency hell

- Avoid circular dependencies
  - Consider dependency via abstraction
  - Use forward declaration and pImpl idiom
- Define interfaces *(abstractions)* in separate header file
- Limit includes in header files to bare minimum, headers needed for implementation include in *cpp* file
- When working on modules boundaries, prefer copies of DTO’s over incorrect dependencies
- Avoid depending on IO/OS modules directly, introduce reasonable abstractions instead
- Do not link DLL-s (shared objects on Linux) statically

Do not join *die-hard template team*

- With many advantages of *templates* in c++, there are several problems:
  - Increased compilation time
  - Need to implement whole template in header file *(see: Avoid dependency hell)*
  - Misused templates leads to vast specializations which makes things hard to maintain in the long run

Do not overcomplicate things

- I know you are smart, there is no need to manifest it in code

Do not oversimplify things

- If there are advantages of increased complexity, then do not cut corners
  - Temporary solutions and hacky workarounds tends to become long-run solutions increased tech debt significantly

Think about testing

- Test behavior, not an implementation (BDD vs TDD)
  - Unit test on interface level, there is no need to test implementation details
    - Too specific tests needs to be updated each time implementation change which is error prone
- Stick to test pyramid:
  - most of the tests should be unit tests
  - another big, lesser though, chunk of tests are integration tests
  - write some regression tests to have overall testing and whole system condition

There is (almost) always a trade-off

- Think about pros & cons of alternative ways to implement things, just pick whatever seems to come with best result even if it means minor problems

Deal with imperfection

- Technical perfection is a process, not a goal