

# Memory Management Chatbot

REVIEW

CODE REVIEW 17

HISTORY

## Meets Specifications

### 🌟 Congratulations 🌟

Dear fellow developer,

Please let me take the chance to express how much I appreciate the effort you have put into this project. 🙌🙌

**Memory management concepts are really challenging, and it takes time and practice to master them. You did a great job, and your code shows an understanding of the presented concepts. Well done!**

## The Big Picture:

To get a good grip on this topic's content, I gathered the points that constitute the big picture of the memory management course.

Every item in the list has its own related details. I want to let you know why the subjects' flow is how it is in the course. This flow is very rational and well built.

**Here is the list:**

1. Learn about dynamic allocation, malloc/free, and new/delete
2. Why we need new/delete after OOP was introduced and how to overload them for various reasons.

3. What are the memory problems that happen with new/delete? Problems like a memory leak, dangling pointers, and so on.
4. How data is being copied between objects, and what problems happen (like shallow copying).
5. The rule of three describes the necessity of implementing the constructor, copy constructor and destructor, and destructor. Copying ways have a strong relationship to the philosophy behind smart pointers (no

copying-> unique\_ptr, exclusive, shared copying -> shared\_ptr, and exclusive ownership also relates to the unique\_ptr after their resources are moved, they are owned by only one object).

6. Then, we come to discover that copying causes a lot of unnecessary creation, allocation, and deletion when objects are returned by value. We need to solve this problem for more memory efficiency.
7. The ultimate solution of the copying is moving through move semantics. To understand move semantics, we need to know the concepts of lvalue and rvalue.
8. Then, we learn about the rule of five that describes the structure of classes that can move resources rather than copying them.
9. Then comes the RAII concept and how it is good to eliminate the need to delete or free resources by wrapping them around an object that takes care of that.
10. The next topic is smart pointers. Please notice that they apply both move semantics and RAII to their internal resources (raw pointers) and eliminate the need to free them.
11. Smart pointers types, use cases, and conversion between them.
12. The topic of ownership and ways to pass smart pointers to function and vice-versa

Please revisit the optic later and use the above list to keep the material organized logically in your mind.

## C++ Standard Library:

Please develop your knowledge more about the C++ standard library. It provides helpful containers/data structures, as well as many functions to manipulate them.

Please have a look [here](#) for an overview of the contents of the library, and [here](#) for the detailed list of the header files in it.

## Coding Style:

I want to recommend this book [Clean Code: A Handbook of Agile Software Craftsmanship](#) and the following resources for code styling:

1. <https://google.github.io/styleguide/cppguide.html>
2. <https://web.cs.dal.ca/~jin/3132/assignments/style.html>

Please take a second to rate this review, and feel free to leave some feedback for me.  
All the best,

## Quality of Code

The code compiles and runs with `cmake` and `make` .

Great! The program successfully builds the chatbot executable.

## Resources on CMake and Make

In the beginning, it is usually not very clear what those build systems or build system generators are and how they can be beneficial to us.

I want to share with you some resources to help with this.

- Overview of [CMake](#) and how it manages cross-platform build processes.
- Introduction to CMake by [examples](#) applies to single file projects and multiple directory projects.

Please note that CMake isn't a build system. It's a *build system generator*. This is why we need to invoke make after running CMake. Running CMake generates Makefiles with the appropriate platform dependencies, and running make actually uses them.

However, we don't need to write Make files, as CMake does this for us, it is good to understand build systems.

- Introduction to [Make](#) and how it uses the Makefile generated by CMake to build the system.
- How to [write Makefiles](#). This is just for your information. It is already automated through CMake.

## Resource about GCC compiler warnings

I want to share with you this article to understand more about the different types of GCC compilers' warnings. Please have a look [here](#).

## Task 1: Exclusive Ownership 1

In file `chatgui.h` / `chatgui.cpp`, `_chatLogic` is made an exclusive resource to class `ChatbotPanelDialog` using an appropriate smart pointer. Where required, changes are made to the code such that data structures and function parameters reflect the new structure.

`_chatLogic` is made an exclusive resource of `ChatbotPanelDialog` using `unique_ptr`. Well done!

It is great that you used the following in `chatgui.cpp`

```
_chatLogic = std::make_unique();
```

The other way is as follows.

```
std::unique_ptr _chatLogic(new ChatLogic());
```

The above is not exception safe as it may cause memory leaks. Imagine when using `new` that an exception happens in the constructor of the resource. In such a case, the object would not be handled properly, and its destructor would never be called even if the managing object goes out of scope. Therefore, `make_unique` and `make_shared` should always be preferred.

**Resource:** For more details about the member functions of the unique pointers, please have a look at the following link

renaming min

[https://en.cppreference.com/w/cpp/memory/unique\\_ptr](https://en.cppreference.com/w/cpp/memory/unique_ptr)

**Note:** Unique pointers should be the default choice unless you know for certain that sharing is required at a later stage. You will see examples of using `shared_ptr` in the next course.

## Task 2: The Rule of Five

In file `chatbot.h` / `chatbot.cpp`, changes are made to the class `ChatBot` such that it complies with the Rule of Five. Memory resources are properly allocated / deallocated on the heap and member data is copied where it makes sense. In each of the methods (e.g. the copy constructor), a string of the type "ChatBot Copy Constructor" is printed to the console so that it is possible to see which method is called in later examples.

The ChatBot class design correctly implements the rule of five. 🙌

**Note:** I would like to share with you the following summary of the Rule of Five:

The Rule of Five states that if you have to write one of the functions listed below, you should consider implementing all of them with a proper resource management policy. If you forget to implement one or more, the compiler will usually generate the missing ones (without warning), but the default versions might not be suitable for the purpose you have in mind. The five functions are:

- The destructor: Responsible for freeing the resource once the object belongs to goes out of scope.
- The copy assignment operator: The default assignment operator performs a member-wise shallow copy, which does not copy the content behind the resource handle. If a deep copy is needed, it has to be implemented by the programmer.
- The copy constructor: As with the assignment operator, the default copy constructor performs a shallow copy of the data members. If something else is needed, the programmer has to implement it accordingly.
- The move constructor: Because copying objects can be an expensive operation that involves creating, copying, and destroying temporary objects, rvalue references are used to bind to an rvalue. Using this mechanism, the move constructor transfers the ownership of a resource from a (temporary) rvalue object to a permanent lvalue object.
- The move assignment operator: With this operator, ownership of a resource can be transferred from one object to another. The internal behavior is very similar to the move constructor.

## Task 3: Exclusive Ownership 2

In file `chatlogic.h` / `chatlogic.cpp`, the vector `_nodes` are adapted in a way that the instances of `GraphNode` to which the vector elements refer are exclusively owned by the class `ChatLogic`. An appropriate type of smart pointer is used to achieve this.

The required change is well implemented, and the class `ChatLogic` exclusively owns `GraphNode` in the vector `_nodes`. Great work!

When passing the `GraphNode` instances to functions, ownership is not transferred.

The instances in your code are passed without ownership transfer utilizing references. 👍

You did a good job with the following

```
std::unique_ptr& node
```

Using the reference here is the proper practice to make sure the ownership is not transferred and the `unique_ptr` lifetime is controlled by the lifetime of the `ChatLogic` class where it is created.

## Task 4: Moving Smart Pointers

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp` all instances of `GraphEdge` are changed in a way such that each instance of `GraphNode` exclusively owns the outgoing `GraphEdges` and holds non-owning references to incoming `GraphEdges`. Appropriate smart pointers are used to do this. Where required, changes are made to the code such that data structures and function parameters reflect the changes.

The exclusive ownership here of the incoming and outgoing instances of `GraphEdges` is working as expected.

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp`, move semantics are used when transferring ownership from class `ChatLogic`, where all instances of `GraphEdge` are created, into instances of `GraphNode`.

Well done using move semantics to move the edges to the graph nodes!

**Resources** on smart pointers:

I found this series on smart pointers very useful. I would like to share it with you

Understanding Move Semantics and Perfect Forwarding: [part 1](#), [part 2](#), [part 3](#).

## Task 5: Moving the ChatBot

In file `chatlogic.cpp`, a local `ChatBot` instance is created on the stack at the bottom of function `LoadAnswerGraphFromFile` and move semantics are used to pass the `ChatBot` instance into the root node.

The local `ChatBot` variable is created in the right place and is properly moved!

It is equally important to understand when moving is not performed and done in the wrong way. I want to share the following **resources** for that:

- [When C++ doesn't move.](#)

This article explores situations in which the move constructor is not called despite being there. The compiler defaults to the copy constructor instead in these cases. It is important to know these situations as we aim to realize performance. In such cases, we write the code that calls move constructors, but this doesn't happen.

- [Understanding when not to std::move in C++](#)

This article gives a deeper view of std::move and its role in moving objects. It is actually a preparatory step that turns objects into rvalues to prepare them to be moved. The article discusses two incorrect usages of std::move.

`ChatLogic` has no ownership relation to the `ChatBot` instance and thus is no longer responsible for memory allocation and deallocation.

The chatlogic class no longer creates or destroys the instance of the chatbot.

When the program is executed, messages are printed to the console indicating which Rule of Five component of `ChatBot` is being called.

The shown output now is as described in the project rubric. great job! 🙌

 [DOWNLOAD PROJECT](#)

17

[CODE REVIEW COMMENTS](#)



[RETURN TO PATH](#)

[Rate this review](#)

