API Programming Report

By: William Melander

Introduction:

This is a report reviewing the refactoring changes made to a Snake game code base we received as part of the final assignment during the API Programming course. The code provided is not clean code due to it being unclear in intent and design causing it to be hard to understand. Our task was to remedy this by refactoring the code that is simple, maintainable, understandable and easily modifiable code.

Dead Code:

Dead code is code that compiles but is unused. This was a major issue for the original code base. An example of dead code that needed to be removed is the sprite rendering and sprite management code as rendering was done through the use of colours and rectangles. The problem with leaving this code behind is that it cluttered and obscured the intent of the code to the reader. As argued in Clean Code's code smell G:9, "dead code is not completely updated when designs change. It still compiles, but it does not follow newer conventions or rules". This means that the code should be removed and implemented when/if sprite rendering is needed as it will not be bug prone and worth reading.

Comments:

Much like dead code, comments needed to be removed. The use of comments can be helpful to explain the intent of complicated pieces of code, however, clean code speaks for itself. This makes comments in code additional text that the reader has to look through. In the original code base, the original author writes comments that are either obsolete or redundant, as well as comments that include chunks of dead code. In Clean Code on pages 286-287, the code smells about comments are stated. To summarise this section, clean code should be expressive enough to describe what it is doing. Comments that are old and irrelevant only add more text that the reader has to parse through.

Naming Convention:

For code to be considered clean the names of functions and variables need to be clear and understandable. In the original code base, many pieces of code weren't named clearly enough so that the reader can make a reasonable distinction between certain pieces of code. A few examples can be found in the original code base's game header, but I want to focus on integers such as 'width' and 'height'. These are used for sizing the width and height of the window upon its creation. However, the names of these variables make their usage unclear. Without looking at the code the reader may assume that they are used to determining the dimension of a rendered rectangle. Clean code pages 309-313 describe the code smells regarding naming convention. The naming convention found in clean code is descriptive, unambiguous and describes any side effects that a function or variable has. This allows the reader to understand how the code functions without having to read too deeply into the code. I kept this in mind as I was refactoring to ensure that functions and variables are easily understood without the need for testing.

Functions should do one thing:

Functions are chunks of code that are called upon to perform specific actions. In clean code, each function should do one thing. In the original code base, multiple functions perform several actions that have been abstracted into separate functions. For example, the main() of the original code base notably handles the game loop, input handling, and rendering. That is several actions too many. Page 36 of Clean Code explains that "the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction". I solved this issue by abstracting these actions into the game class as separate functions. This allows the code reader to easily understand the actions performed step by step without the need for testing allowing for cleaner code to be achieved.

Const Correctness:

Const correctness is important not only for performance optimisations but also for showing the reader that a value will not mutate. In the original code base outside of some constant char pointers and a few function parameters passing by const reference, there is no const correctness. In Tour of C++ at the beginning of section 1.6 on page 9 Stroustrup states that having a const variable or function tells the reader that "I promise not to change this value". To uphold const correctness, I checked and make sure that variables such as colour and render size constexpr. For functions, I passed const-referenced variables into functions without the worry that they will mutate. Collectively this allowed for more robust code to be created. To the reader, this communicates that variables and functions marked as const/constexpr will not change during runtime allowing for better-understood code.

Usable Initialisations:

Useable initialisations need to be done within the class header. It is a persistent issue within the original code base as values are not initialised to useable values at runtime and instead have values become useable through the use of constructors and initialisation functions. According to the CPP Core Guidelines rule C.48 you should “Prefer in-class initializers to member initializers in constructors for constant initializers”. This properly communicates to the reader what the starting values of variables are so that they don’t need to parse through the rest of the code to find out what they are initialised to.

Rule of Zero:

The rule of zero is important to provide simple code. With some initialisations performed within constructors in the original code base, the rule of zero can not be upheld. The CPP Core Guidelines rule C.20 states that “C.20: If you can avoid defining default operations, do”. This is because by not doing so we would need to define all other default operations to uphold the rule of five. Not writing a constructor provides the cleanest semantics and results in cleaner code.

I fixed these issues by removing all initialisation functions and constructors that just initialise values by ensuring that all member variables have useable initialisation.

Exception Handling:

Exception handling is important to have a proper flow of code that can throw exceptions. The original code base has slight exception handling upon the initialisation of the window and renderer however they don’t throw exceptions and just exit the code if they are nullptr’s. Tour of C++ recommends exception handling, on page 36 the “use of the exception-handling mechanisms can make error handling simpler, more systematic, and more readable”. I solved this lack of exception handling by having the game run within a try block as well as having an SDL_InitCheck to see if the exception was caused by SDL. For all other exceptions I use the standard library std::exception, if that doesn't catch it I use the catch-all clause. Exception handling tells the reader that everything within the try block can throw an exception allowing for better-understood and robust code.

Algorithms:

As algorithms are part of the standard library we should use them. In the original code base raw for loops are mainly used in the player class to modify and look through the snake_body array. Algorithms can solve this as said in Tour of C++ on page 157 “standard-library algorithms tend to be more carefully designed, specified, and implemented

than the average hand-crafted loop". For the body part shifting raw for loop that makes snake parts stay together, I used std::shift_right. For checking if the snake has collided with itself I used std::any_of. Since algorithms have documentation that describes what they do and provides examples of how they work the reader can read and understand the code without having to figure out a hand-crafted and error-prone piece of code.

RAII:

RAII allows for code to clean up after itself removing all possibility of memory leaks. In the original code base, this is not used as even if an exception occurs with the window or renderer they do not get cleaned up causing memory leaks to occur. I solved this with the use of unique pointers. In Tour of C++ section on RAII, Stroustrup describes the use of unique pointers as a way to prevent careless memory leakage and to ensure proper interaction with exception handling. This allows for robust code to be written as memory leaks will not occur due to the proper resource handling.

Law of Demeter:

The Law of Demeter is used to minimise coupling between classes. In the original code base, this coupling can be seen within the player class when updating the positions of the snake. Clean Code rule G36 states that we should "avoid transitive navigation" as it creates rigid architecture. With rigid architecture, it is much harder for the design and architecture of the code base to change. I solved this issue by creating public getter functions for the variables that needed to be accessed outside of a class. As well as ensuring that essential variables such as position are easily accessed without needing to navigate transitively. Clean code is maintainable and easily changeable code.

Complexity:

A defining part of the original code base was its complexity with some notable overengineer taking place with input handling and rendering being the most notable of them. Input handling was translated SDL_Keycodes over to a KeyEnum. This enum contained all possible keys the player could press but only the up, down, left and right arrow keys were used. I solved this by having the player use the SDL_Keycodes that then set the movement direction of the snake. This movement direction was a Vector2Int was the movement speed of the specified direction, since the position and movement were both Vector2Int's I was able to reduce the movement code down to a single line by adding the movement speed and player position. For the rendering, it had a render queue, a redundant piece of code that would fill up with rects and then empty after rendering all in one frame. This was fixed by removing the render_queue and just directly calling on a render function that would take a colour and position and draw a rect. This drastically reduced the complexity of simply rendering a square on the window. A quote from Clean Code by Grady Booch says "Clean code is simple

and direct.”. With these few examples of the reduction in complexity, the code reader can better understand the intent and function of the code due to it being simple and expressive with its intent.

Conclusion:

In conclusion, with the usage of modern c++ libraries and concepts, we can achieve clean code. With proper naming conventions and functions that do one thing, we can provide expressive pieces of code without clutter. Through the use of const correctness, algorithms, RAII and exception handling we can write code that is efficient and robust. With the Law of Demeter we can protect member variables and reduce coupling between different classes allowing us to easily change the code architecture. With all of these combined we create straightforward clean code.

Personal Reflection:

I think the second time around I learned a lot more due to your feedback. This and doing significantly more research allowed me to understand modern C++ coding concepts and libraries to a larger degree. Now I can say that I understand C++ much better now especially when it comes to clean code. With these refactorings and redone code review of Oliver's code, I felt much better equipped knowledge-wise even though learning the concepts felt like a slog. I learned about myself that I have a slight tendency to overengineer my code and your feedback pointed out flaws that I didn't even realise existed. As stated with my last personal reflection I feel like I have become a better programmer because of this course.