Code Refactor Report

Daniel Burt, API Design

DISCLAIMER:
The code I reworked was my own snake from first year, not Teemu's. Features or aspects present in his version are not necessarily present in this one for this reason.

First I removed all redundant files, as per the instructions:
Collider
Image
Sprite
Rectangle

All of the resources and implicit dependencies were moved to where they were needed and their connections were made explicit.

Previously, the Window resource, Renderer, UpdateTimer and Game were separate objects. The renderer depends on the window to work. In main(), an instance of every object was created and connected through the implementation of the main() function itself. If a user wanted to launch this snake game from their own code without this main() function, they would have to figure out how to link everything together, which would be needlessly difficult.

To resolve this issue, every resource was moved into the game class and put in the necessary order. To run the game, the user calls the run() function. The game's default constructor acquires all the resources in the order they are needed, and run() runs the game until the game is over. This makes it trivial for the user to use this code as intended.

P.1: Express ideas directly in code

Everything possible was made into its own function with a name. This contributes massively to the readability of the entire project, since the intent of the code is expressed in the name of the function being called.

In some instances, functions only consist of a single line. However, we don't always know the context of this line. Making things functions, even just for the purpose of contextualizing single lines, is helpful for the reader.

In this example, intent is expressed in every line. If it is relevant, the user can peer into any of these functions to see exactly what is happening, though this is rarely as important or interesting as what is going on more broadly.

```
void Game::update()
{
    if (snake.is_alive() && snake.should_die())
    {
        snake.start_death_animation();
        print_result();
    }
    if (snake.head_coords() == apple_coords)
    {
        snake.add_part();
        apple_coords = get_random_free_space();
    }
    snake.update();
}
```

C.20: If you can avoid defining default operations, do

Several structs in the project contain nothing but basic types. All of these had several default operations, constructors and destructors which did nothing different what would have been default. I removed these redundant default operations, enforcing the rule of zero.

Several structs ended up looking like this:

```
struct Color
{
    unsigned char r{}, g{}, b{}, a{};
};
```

Applications:

Color
Vector2
Coords

P.8: Don’t leak any resources

I implemented RAII to all of the game’s resources. In their constructor, they acquire their relevant resource. In their destructor, they free this resource. This means I never have to do it manually and don’t risk forgetting.

Example: I need never worry about leaking my renderer resource.

```
Renderer(const Window& window)
{
    renderer = SDL_CreateRenderer(&window.get_window(), 0, SDL_RendererFlags::SDL_RENDERER_ACCELERATED);
    if (renderer == nullptr)
    {
        throw std::runtime_error(SDL_GetError());
    }
}
~Renderer() noexcept
{
    try
    {
        std::cout << "Destroying Renderer.." << '\n';
    } catch (...){}
    SDL_DestroyRenderer(renderer);
}
```

Applications:

SDL_Initializer
Window
Renderer

C.42: If a constructor cannot construct a valid object, throw an exception

---

Every constructor that acquires any resource makes sure said resource is valid before the end of the scope. If the resource is invalid, the constructor throws an exception. This exception is printed to the console and the program terminates immediately. This will make any fault in the code immediately apparent and clear. An invalid resource can lead to all kinds of bugs and problems further down the line and are much harder to diagnose. Throwing as soon as something goes wrong leads me straight to the source straight away.

Applications:

SDL_Initializer
Window
Renderer

C.21: If you define or `=delete` any copy, move, or destructor function, define or `=delete` them all

---

Destruction, moving and swapping are intrinsically related. If any of the operations needs to be explicitly defined, it is likely they all need defining. It is best to either define them or prevent them from being used if their default behaviour wouldn't work. This law of five was implemented for all of the resources.

Applications:

SDL_Initializer
Window
Renderer

```cpp
~Window() noexcept
{
    try
    {
        std::cout << "Destroying Window.." << '\n';
    }
    catch (...) {}
    SDL_DestroyWindow(window);
}
Window(const Window&) = delete;
Window& operator=(const Window&) = delete;
Window(Window&&) = delete;
Window& operator=(Window&&) = delete;
```

I.2: Avoid non-`const` global variables
Note: Global constants are useful.

Every level of the codebase needs to know some global value for some purpose.
Examples include:
Window needing to know what size it should create itself in.
The snake needing to know the size of the play area in order to know when it has left it
The UpdateTimer needs to know the target frame rate to know how long it should wait
etc.
A file of global constants made proved convenient and meant there was no need to pass any data back and forth. As an added bonus, this meant I could tweak the game in various different ways, all from the same place.

All global constants present in the game can be found here.

```cpp
#include "Color.h"
#include "string"
#include "Coords.h"

constexpr std::string_view WINDOW_NAME = "Snake";

constexpr int SQUARE_SIZE = 15;
constexpr bool FILLED_SQUARE = true;

constexpr int GRID_WIDTH  = 50;
constexpr int GRID_HEIGHT = 50;
constexpr Coords MIDDLE_COORDS = { .x = GRID_WIDTH / 2, .y = GRID_HEIGHT / 2 };
constexpr int SCREEN_WIDTH  = GRID_WIDTH * SQUARE_SIZE;
constexpr int SCREEN_HEIGHT = GRID_HEIGHT * SQUARE_SIZE;

constexpr float        TARGET_FPS = 20.0f;
constexpr unsigned int MS_BETWEEN_FRAMES = static_cast<unsigned int>(1000.0f / TARGET_FPS);

constexpr Color BACKGROUND_COLOR = Color(0, 0, 0, 0);
constexpr Color APPLE_COLOR      = Color(255, 0, 0, 0);
constexpr Color SNAKE_COLOR      = Color(0, 0, 255, 0);

constexpr int SNAKE_START_LENGTH = 3;
```

Application:
Constants.h

The Law of Demeter

The law of Demeter is a useful rule of thumb meant to simplify and condense code. It suggests that it is a good idea to implement interfaces for objects to communicate with their nearest friends.

Here's an example of how game program could find out various things about the snake:

```
If(snake.snake_parts.front() == apple_coords && snake.snake_parts.size() < 10)
.
.
.
```

Writing code in this way can quickly become very taxing for the person writing it and cumbersome to the person reading it, as they have to follow a daisy-chain of dots which often don't aid in understanding what's actually going on.

Moreover, if a line like this is found in many places and needs to change, it is an inconvenience to hunt every instance of it.

I made sure to implement interfaces such that more than one dot is never necessary in any operation. This makes the code easier to read, and easier to write in the long run.

These are the interfaces I implemented to communicate with the snake's friends.

```
inline Coords head_coords() const noexcept;
inline size_t length() const noexcept;

std::vector<Coords>::const_iterator begin() const noexcept;
std::vector<Coords>::const_iterator end() const noexcept;
```

Application:
Everywhere necessary

Final thoughts and reflections

In conclusion, this course has made me think more about how to write code that is convenient to the programmer. Alarm bells have started going off in my head whenever I see one too many dots in a row or a function with a bunch of no-context operations.

The biggest change I've noticed in my own programming is that I've started writing little functions for everything. This minor change makes the code more readable (because I'm directly expressing what I'm doing), more concise (Law of Demeter) and more bug-resistant, since I only have formulate the logic correctly in a single place. This also helps divide bigger problems that seem complicated when you have to look at ever aspect at a low level.

Another thing I've noticed is I'm more aware of all the work I could not be doing. Great algorithms exist so I don't need to create my own all the time. I don't need to implement a bunch of default operations that are already there. These kinds of things spare me a lot of effort.

These kinds of considerations and insights make programming more about what makes programming fun, and helps keep it simple. This course has gotten the ball rolling in a good direction.