Tutorial 5 - Refactoring

Before starting this tutorial, you must complete the last one and have it working. Failure to do so will result in not being able to complete these steps. It is your choice but is recommended to crate a new branch each tutorial.

This tutorial is going to walk through the steps of refactoring. Refactoring is a term you will soon become familiar with as a programmer. The term means to restructure your code as to improve operations without altering its functionality. We will be refactoring our texture code into its own class. All loading, positioning, rendering and more will be encapsulated in what we call a Texture2D class.

Rather than continually passing around separate values for the X and Y coordinates for images, we are going to combine them into a single struct. This we will call Vector2D and will enable us to pass a single object between functions. We will put all the new types we create in a single header file.

1. Firstly, add a new file to the Header folder called "Commons.h".
2. If it hasn't automatically done so, add #pragma once to the top of the file to ensure it get processed only once.
3. Next, declare a struct for a Vector2D type:

   ```
   struct Vector2D
   {

   };
   ```

   a. In the body of the struct add two floats, one for x and one for y.

   ```
   float x;
   float y;
   ```

   b. To enable us to create an object of this type in function calls, we need to add default constructors. Do this with the following after your float creation:

   ```
   Vector2D()
   {
           x = 0.0f;
           y = 0.0f;
   }
   ```

   c. Now, add one more constructor, this will be just as the one above, however, this constructor will take parameters. Set the parameters as two floats, initial_x and initial_y. This will allow for values to be passed and set their corresponding internal values. Then, in the body, instead of x = 0.0f; etc have x = initial_x and the same for y.

We can now turn our attention to the new Texture2D class. A portion of the code required was implemented in previous tutorials and will simply require moving. If you have not completed the previous tutorials, I suggest you do this before continuing.

1. To create a new class right the solution explorer -> Add -> Class and call it Texture2D.
2. As mentioned in semester 1 when creating classes, the header file needs a macro declaring to prevent the compiler trying to create multiple versions of the class

```
#pragma once
#ifndef _TEXTURE2D_H
#define _TEXTURE2D_H
class Texture2D
{

};

#endif //_TEXTURE2D_H
```

   for every class you create the header should have a macro added in this manner. The name will change obviously, for example the character class would use #ifndef CHARACTER_H. If you do not do this, you will have redefinition errors. It is important that you remember to do this, and you are expected to add this without prompt going forward.
3. Within the body of the class in header add two access types like so:

```
public:

private:
```

   You'll be instructed to add code to one of these sections as we go along.
4. Next, include SDL.h before the class and after the macro so we can continue to work with the framework. We will also need to include string and the Commons header we created.

```
#include <SDL.h>
#include <string>
#include "Commons.h"
```

5. Let's start with adding the private variables (typically, class variables should always be private):

```
SDL_Renderer* m_renderer;
SDL_Texture* m_texture;

int m_width;
int m_height;
```

6. Now for the public function prototypes. We need a constructor which takes in a pointer to the renderer and a destructor. We will also create/move functions to load textures, free them and render them. Finally, we will add two inline functions to simply return the height

and widths.

```
Texture2D(SDL_Renderer* renderer);
~Texture2D();

bool LoadFromFile(std::string path);
void Free();
void Render(Vector2D new_position, SDL_RendererFlip flip, double angle = 0.0);

int GetWidth() { return m_width; }
int GetHeight() { return  m_height; }
```

7. Move across to the Texture2D cpp file now. This should have the include for the header file already but if not, include it (using double quotations). Now is a good time to include SDL_image.h and iostream (these are included using the chevrons <>) feel free to add the using namespace for std if you wish too.
8. We need to implement the functionality for each of the function prototypes detailed in the header. I would recommend copying all the definitions across from the header. Remove the semi colons and add Texture2D:: between the return value and the function name. Also add the required parenthesis. Here is an example of the constructor:

   Copy and paste the prototype

   ```
   Texture2D(SDL_Renderer* renderer);
   ```

   remove the semi colon, add braces and Texture2D::

   ```
   Texture2D::Texture2D(SDL_Renderer* renderer)
   {

   }
   ```

   Constructors do not have return values, which is why there is not one shown in the above snippet. For functions that do have return values remember to put the Texture2D:: between that and the function name.

   ```
   void Texture2D::Free()
   {

   }
   ```

   Note: when copying over the Render function, remove the value (0.0) for angle as leaving it in will cause issues, we have already initialised it so it is not needed again. This is a good point to remember.
9. Starting with the constructor, we will initialise the renderer private variable, we are passing the renderer to this function so make m_renderer equal the passed renderer.

a. Next the destructor, call the Free function so everything is cleaned up and set the m_renderer to nullptr.

```
//Free up memory
Free();

m_renderer = nullptr;
```

b. Since we just called the Free function let's work on that next. This is for you to complete. The function should first check if the m_texture variable is nullptr or not. This code was written in the previous tutorial, you can just move it over and swap g_texture for m_texture. You will also want to set m_width and m_height to equal 0 if the texture is destroyed.

   i. Once the code has been moved or added, you may delete FreeTexture from the source file. For now, just remove the function body, prototype, and function call in CloseSDL, don't worry about the error this creates in LoadTextureFromFile, we'll sort that after.

c. In the render function, take the following code from the previous weeks Render() function and adapt it to use the parameters passed in.

```
//set where to render the texture
SDL_Rect renderLocation = { 0,0,m_width, m_height };

//Render to screen
SDL_RenderCopyEx(m_renderer, m_texture, nullptr, &renderLocation, 0, nullptr,
SDL_FLIP_NONE);
```

You need only take these two lines!

d. Finally, we have the LoadFromFile(string path) function. The majority of this code has already been implemented in a previous tutorial. Copy this code across and adapt it as follows.

   i. You should have the error first of FreeTexture() as this no longer exists relace with a call to Free().

   ii. Next, remove the pointer for p_texture, this will no longer be needed.

   iii. Keep the surface pointer as it is and the if statement that checks if p_surface is nullptr or not. In the if statement, first add the following:

```
//colour key the image to be transparent
SDL_SetColorKey(p_surface, SDL_TRUE, SDL_MapRGB(p_surface->format, 0, 0XFF, 0XFF));
```

   iv. Now change the p_texture and g_renderer to the correct m variables for the SDL_CreateTextureFromSurface.

   v. After this you should be faced with an if statement for p_texture == nullptr. Obviously change the p_texture to the member variable but then add an else condition to accompany this if. In this else statement set the height and width values based off the surface size.

```
m_width = p_surface->w;
m_height = p_surface->h;
```

vi. Everything else can stay the same bar the return. Rather than returning the texture we will be returning whether the process was successful or not.

```
//Return whether the process was successful
return m_texture != nullptr;
```

10. Back over the main/source file we need to make some changes to use our new class. If you haven't already removed the function body and prototype for the old LoadTextureFromFile do so now. And include the “Texture2D.h” and “Commons.h” files.

    a. Now, remove the SDL_Texture g_texture pointer and replace with a new one using our class.

    ```
    Texture2D* g_texture = nullptr;
    ```

    b. Let's adapt the InitSDL first. You should have an error from the loading of the background image from path. Change it to the following:

    ```
    //Load the background texture
    g_texture = new Texture2D(g_renderer);

    if(!g_texture->LoadFromFile("Images/test.bmp"))
    {
            return false;
    }
    ```

    c. Next up is the Render function. We now need to call the render function of the Texture2D class, it should look like this:

    ```
    //Clear the screen
    SDL_SetRenderDrawColor(g_renderer, 0xFF, 0xFF, 0xFF, 0xFF);
    SDL_RenderClear(g_renderer);

    g_texture->Render(Vector2D(), SDL_FLIP_NONE);

    //update the screen
    SDL_RenderPresent(g_renderer);
    ```

    d. Finally, in the CloseSDL add the following to clean up:

    ```
    //release the texture
    delete g_texture;
    g_texture = nullptr;
    ```

11. Build and run your project. It should have the same output as before. Should have any errors, go back through the steps to ensure you have done everything right.

The reason for this refactoring is so we can now use our LoadFromFile function in other classes such as the character class (which we will make later) without having to replicate the process each time.

The next tutorial will focus on creating a screen manager class.