Tutorial: Create 2D Game Engine using C++

URL: <https://www.youtube.com/playlist?list=PL-K0viiuJ2RctP5nlJlqmHGeh66-GOZR_>

# Purpose

These notes summarize the you tube video playlist “How to Make a 2D Game Engine with C++ & SDL”. We will try to follow the videos in order but will not hesitate to re-arrange the material to make it easier to read and follow and understand the concepts presented. I will make corrections to the code and provide additional material (without attribution) from the Internet. These are my notes in trying to follow and understand the videos.

I was going to use the latest version of SDL – SDL3 but found too many differences between SDL2 and SDL3 so decided to stick to the older version to match the video. I am using the latest version of SDL2 and Code::Blocks. Each section covers the material in the video faithfully. So I don’t feel like a code monkey I do add a lot of notes on the specific SDL functions so that I understand how the SDL functions work.

## Background

I am quite rusty with C++ and never worked with SDL2 library (I did start to write notes many years ago on SDL). I will describe things in a little bit more detail than what is covered in the video tutorial since I am using this tutorial to relearn C++ and SDL2.

# 1. Setup SDL, SDL\_image, SDL\_ttf in Code::Blocks

The first video walks you through obtaining and installing the application Code::Blocks. Code::Blocks will be used to create our C++ source files and compile and run our game program.

In addition this video creates the initial project to be used throughout the series, which of course starts as a “Hello, World” project. Lastly, we install the two main SDL libraries – SDL2 and SDL\_image.

## Why use Code::Blocks?

Code::Blocks is a free open-source, cross-platform Integrated Development Environment (IDE) primarily designed for C, C++ and Fortran programming. The IDE was originally released in 2005, and its development is driven by a team of contributors using wxWidgets library for its graphical user interface (GUI). The first official stable version was 8.02 released in 2008. The release marks the year and month of the Code::Blocks version. Like most modern IDEs it provides for extensions via a plugin system.

|  |
| --- |
| What is wxWidgets?  wxWidgets is a free and open-source C++ library used for creating cross-platform graphical user interfaces (GUIs). It allows developers to write applications that can run on multiple operating systems, such as Windows, macOS, Linux, and more, without needing significant code changes. This is achieved by using the native API of each platform, ensuring that applications have a native look and feel.  Originally created in 1992 by Julian Smart, wxWidgets was initially called wxWindows but was renamed in 2004 due to a trademark issue. It supports a wide range of features, including GUI components, event handling, multithreading, file handling, and more. Additionally, it has bindings for other programming languages like Python, Perl, and C#.  Developers often use wxWidgets for its portability and ability to create sophisticated applications with minimal effort. Let me know if you'd like to explore its features further! |

The key feature of Code::Blocks is that you will be able to find an installation package for all modern operating systems – Windows, Linux and Mac OS. This is why the emphasis on cross-platform. You should have no issue following the video tutorials using any operating system. In addition, we use all open source libraries.

## Install Code::Blocks

The official website to obtain Code::Blocks is <https://www.codeblocks.org/downloads/> . You have the option of:

* Downloading the setup file for Code::Blocks and executing it.
* Download the source code and build from scratch

The Binary release has a Windows, Linux and Mac OS X version. The fact that there is a version of Code::Blocks that runs on all three platforms is the primary reason it has been selected for this and other tutorials. At the time of this writing the latest version is 25.03, which means it was released in March of 2025.

My choices for Windows are:

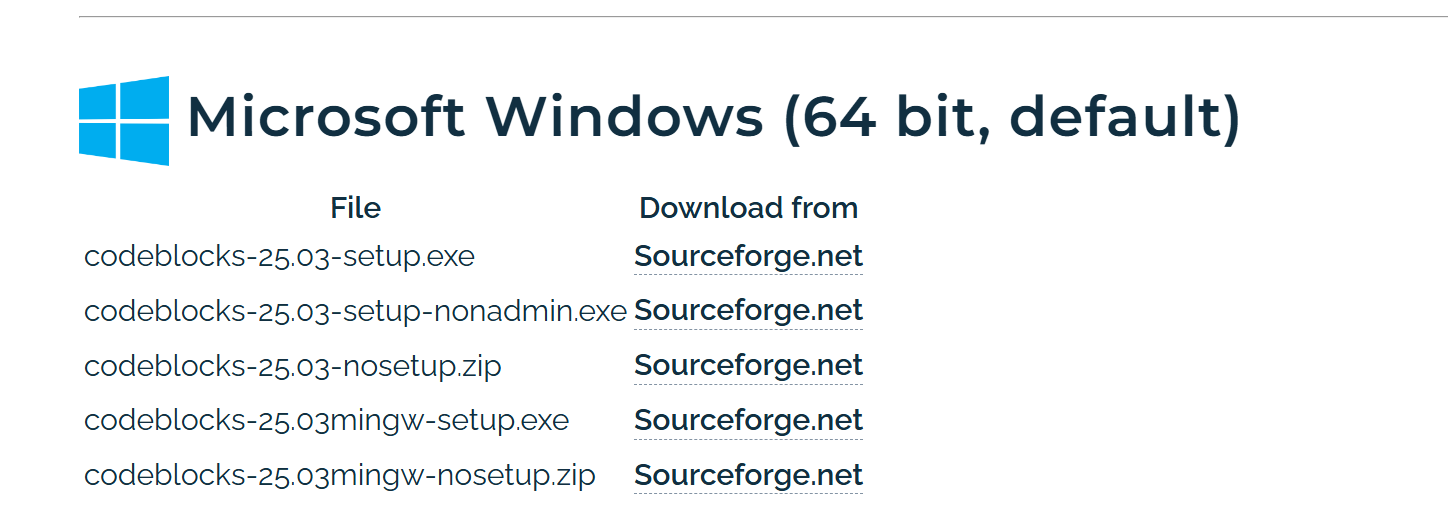


Figure 1 - Selecting the right setup.exe to download

I decided to download codeblock-25.03mingw-setup.exe for 64-bit.

Side note: I am surprised that the project has not moved to GitHub.



Figure 2 - Downloading file from Sourceforge

I then run the setup.exe file:



Figure 3 - Locating and executing the setup.exe file

### Running the Setup

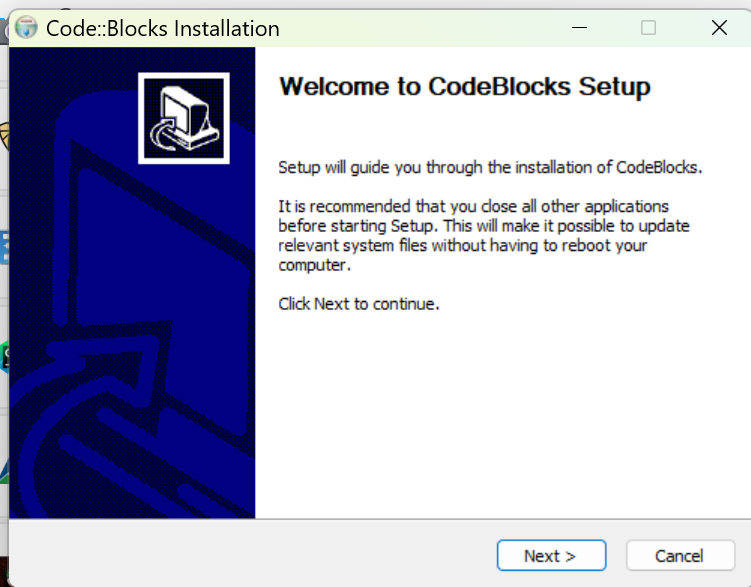


Figure 4 - Initial Code::Blocks Installation dialog

I did not find it necessary to close any other applications. I just pressed “Next >”.

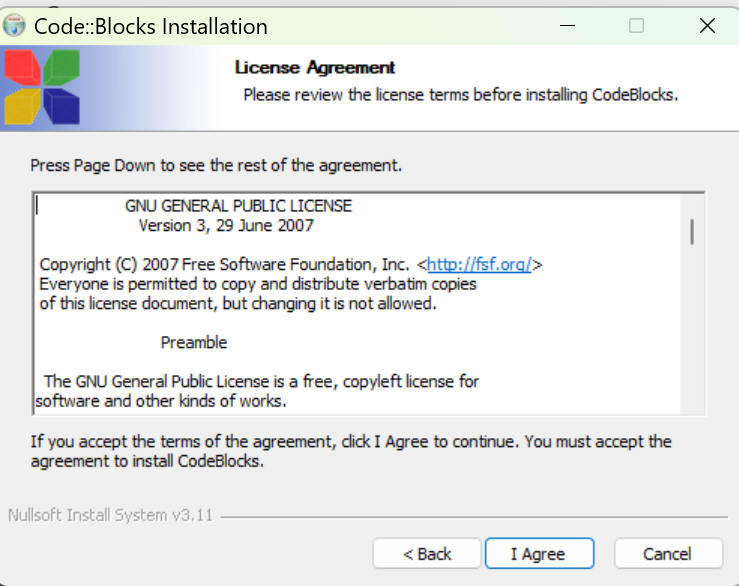


Figure 5 - The License Agreement screen

OK, I did not read the license agreement but knowing it is the GNU license assures me that it is open source and free. I just clicked on “I Agree”.

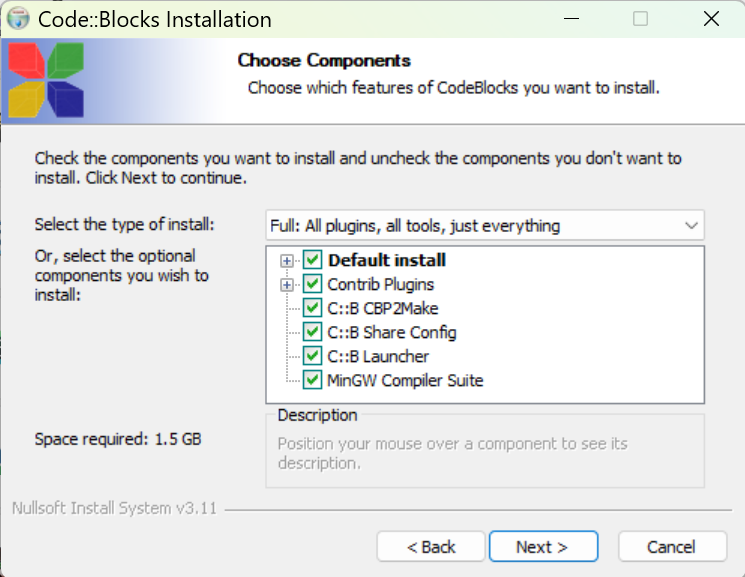


Figure 6 - The "Choose Components" screen

It appears by default all the components are selected. A quick description of each component:

* **Contrib Plugins** – these are additional plugins developed by the community to extend the functionality of the Code::Blocks IDE. These plugins are not part of the core set but have proven to be valuable enough to be included in the official Code::Blocks repository.
  + **Code Snippets Plugin**: Helps to manage and insert reusable code snippets.
  + **DoxyBlocks Plugin**: Integrates Doxygen for generating documentation from your code
  + **CppCheck Plugin**: Provides static code analysis to identify potential bugs or issues
  + **SpellChecker Plugin**: Checks spelling in comments and string literals
  + **Valgrind Plugin**: Integrates Valgrind for memory debugging and profiling
* **C::B CBP2Make** – is a tool designed to generate Makefiles from Code::Blocks project files (\*.cbp) or workspace files. Essentially, it allows you to convert your Code::Blocks projects into Makefiles that can be used with GNU Make or other build systems.
* **C::B Share Config** – this is a tool that allows you to import and export parts of your Code::Blocks configuration. It’s particularly useful when you want to transfer settings between different computers or configurations.
* **C::B Launcher** – is a utility that helps manage the launching of the Code::Blocks IDE. This tool is useful for advanced users.
* **MinGW Compiler Suite** – The MinGW (Minimalist GNU for Windows) is a development environment that provides a native Windows port of the GNU Compiler Collection (GCC). It allows you to build native Window applications without relying on third-party runtime libraries.

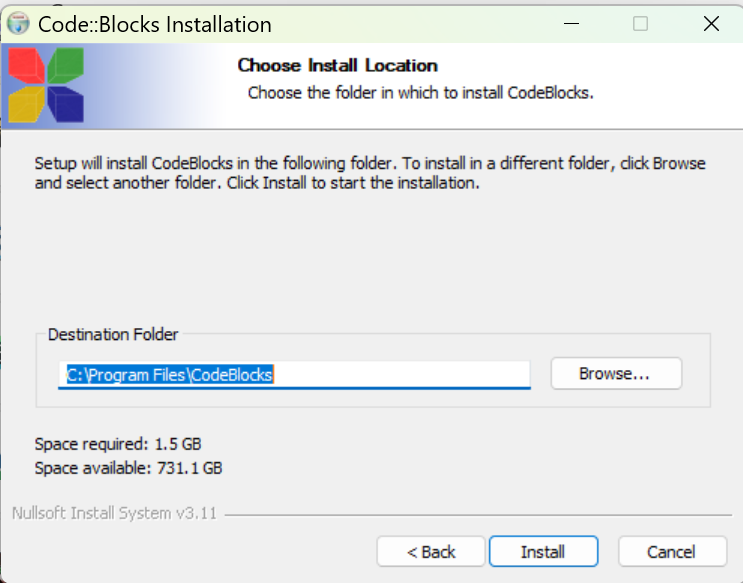


Figure 7 - Installation location on your PC

I usually take the default location.

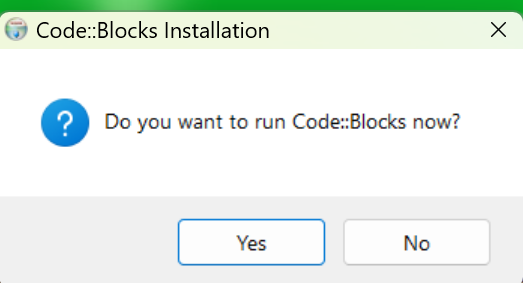


Figure 8 - Prompt to run Code::Blocks now?

I clicked on “Yes” so I can enter a simple “hello world” program to make sure everything works.

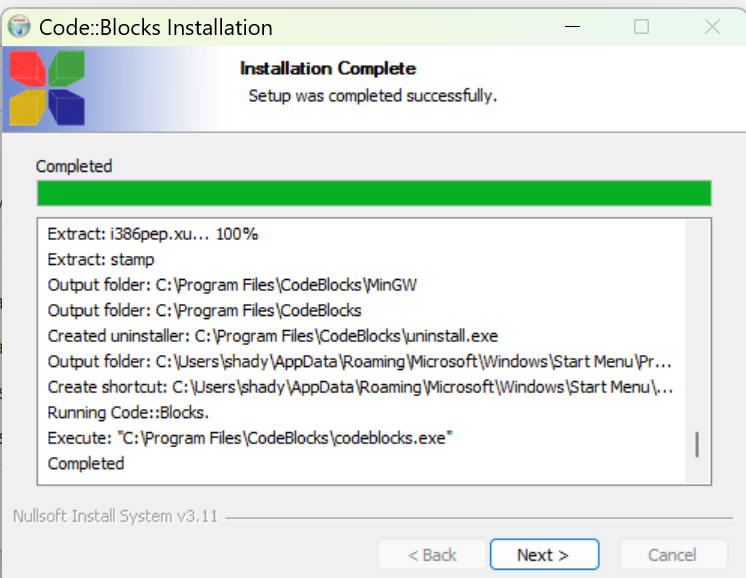


Figure 9 - Code::Blocks installation completed dialog

I clicked on “Next >”.

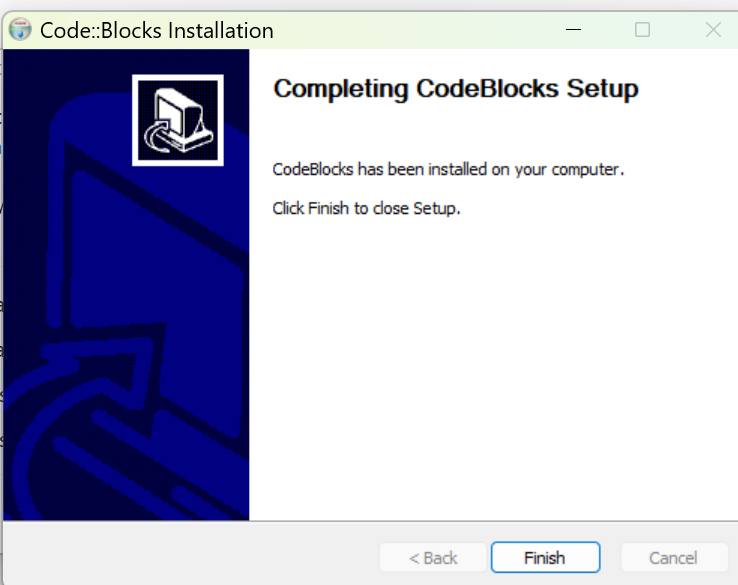


Figure 10 - The final Code::Blocks setup screen

Click on “Finish” and you will see Code::Blocks starting if you clicked “Yes” to start Code::Blocks now.

### Starting up Code::Blocks

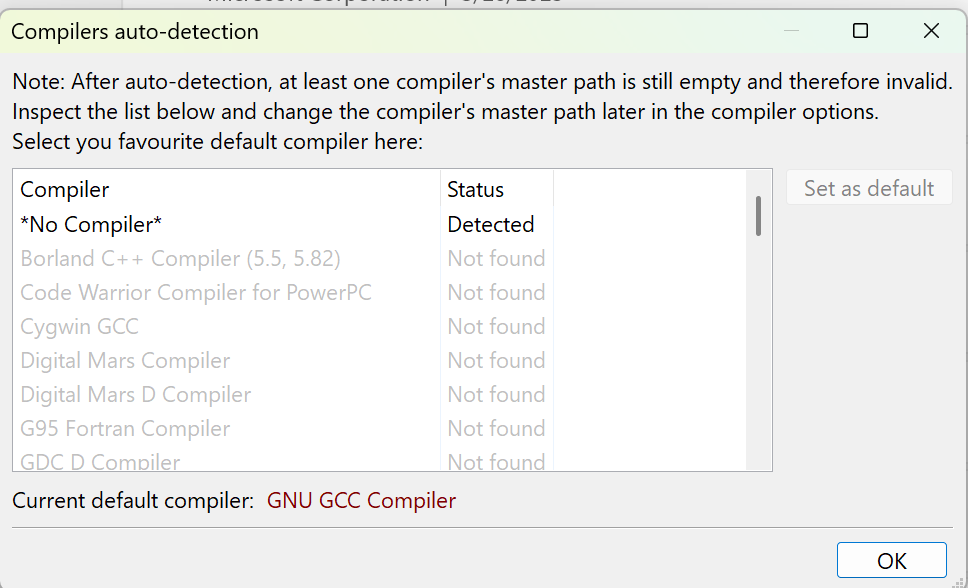


Figure 11 - Code::Blocks lists all the compilers it detected

As you can see from the list you have the option of using many different compilers in Code::Blocks. We do expect Code::Blocks to find MinGW compiler because that is the version we downloaded.

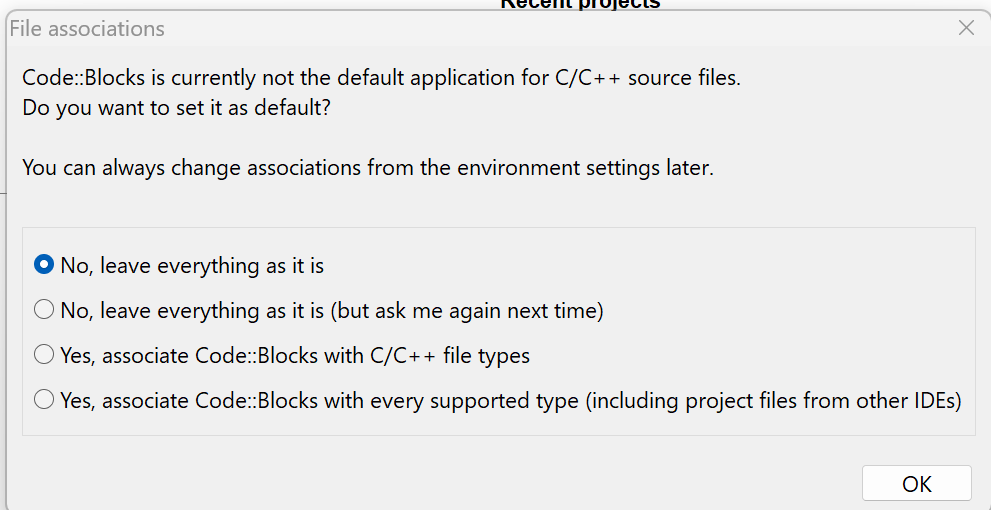


Figure 12 - Option to update file association

I opted for the default of “No, leave everything as it is” since I do use Visual Studio 2022 for other projects.

### Setting Code::Blocks to use K&R Style braces

When Code::Blocks creates the initial code for you it does not use K&R brace style (also known as Kernighan and Ritchie style), you can adjust the formatting settings in the editor since a. I prefer K&R and b. the video presenter uses that style.

* Go to Settings 🡺 Editor

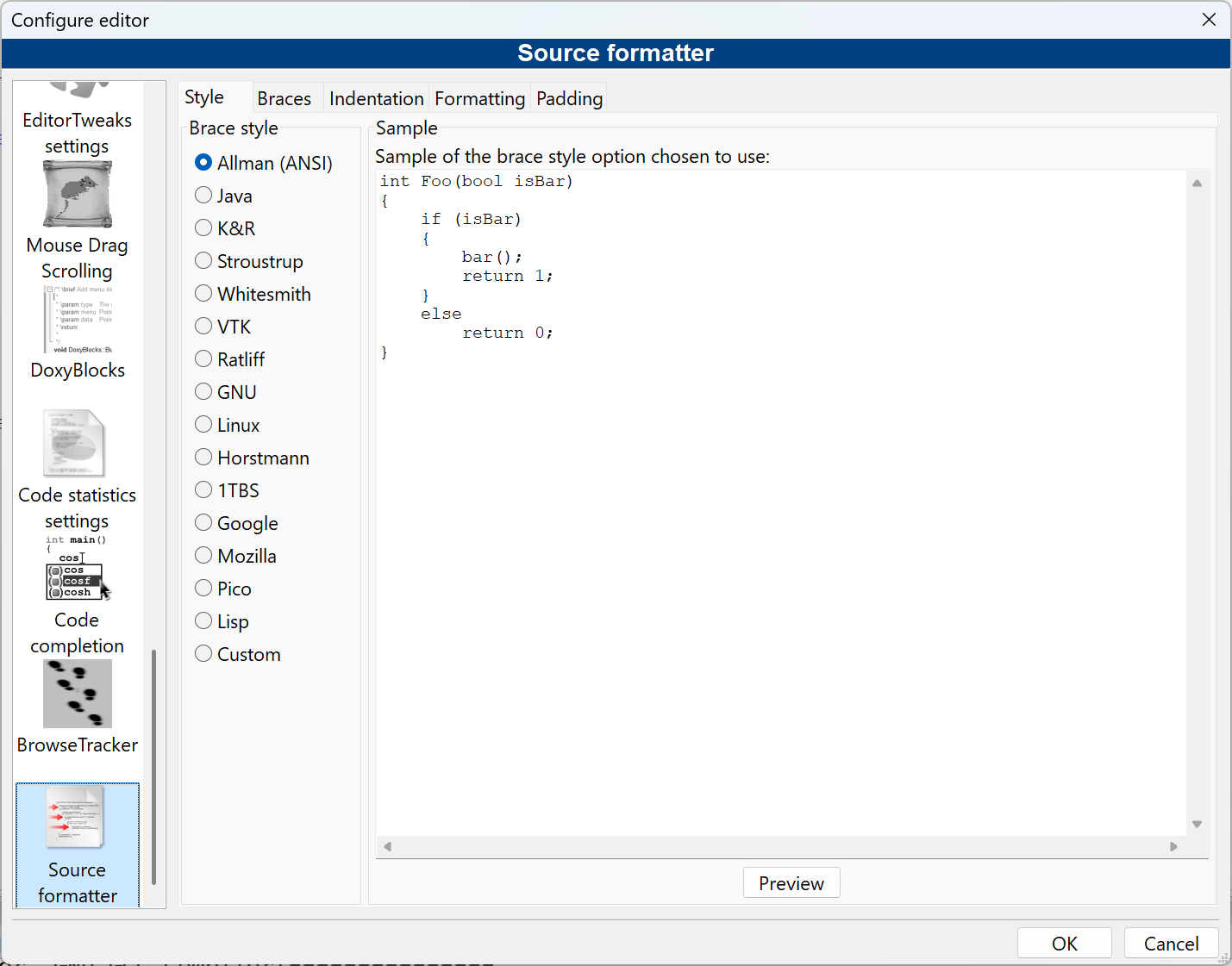


Figure 13 - Source formatter

* Select K&R

A screenshot of a computer

AI-generated content may be incorrect.

Figure 14 - Using K&R

I personally like Java style because I have mainly been a Java programmer for over 20 years (I know, I know, say no more!)

Note: I did not do this until after the end of video #3.

## Starting a Project with Code::Blocks

A screenshot of a computer

AI-generated content may be incorrect.

Figure 15 - The starting Code::Blocks screen

Note: I downloaded the Code::Blocks manual but the screenshots of the application appear to be dated!

### Making Adjustments to the screen

Another note: The screen icons and fonts appear to be small to me (it is probably due to my screen resolution). I made things larger by doing the following:

* Go to Settings 🡺 Environment 🡺 View
* Increase Message logs’ font size the Toolbar icon size

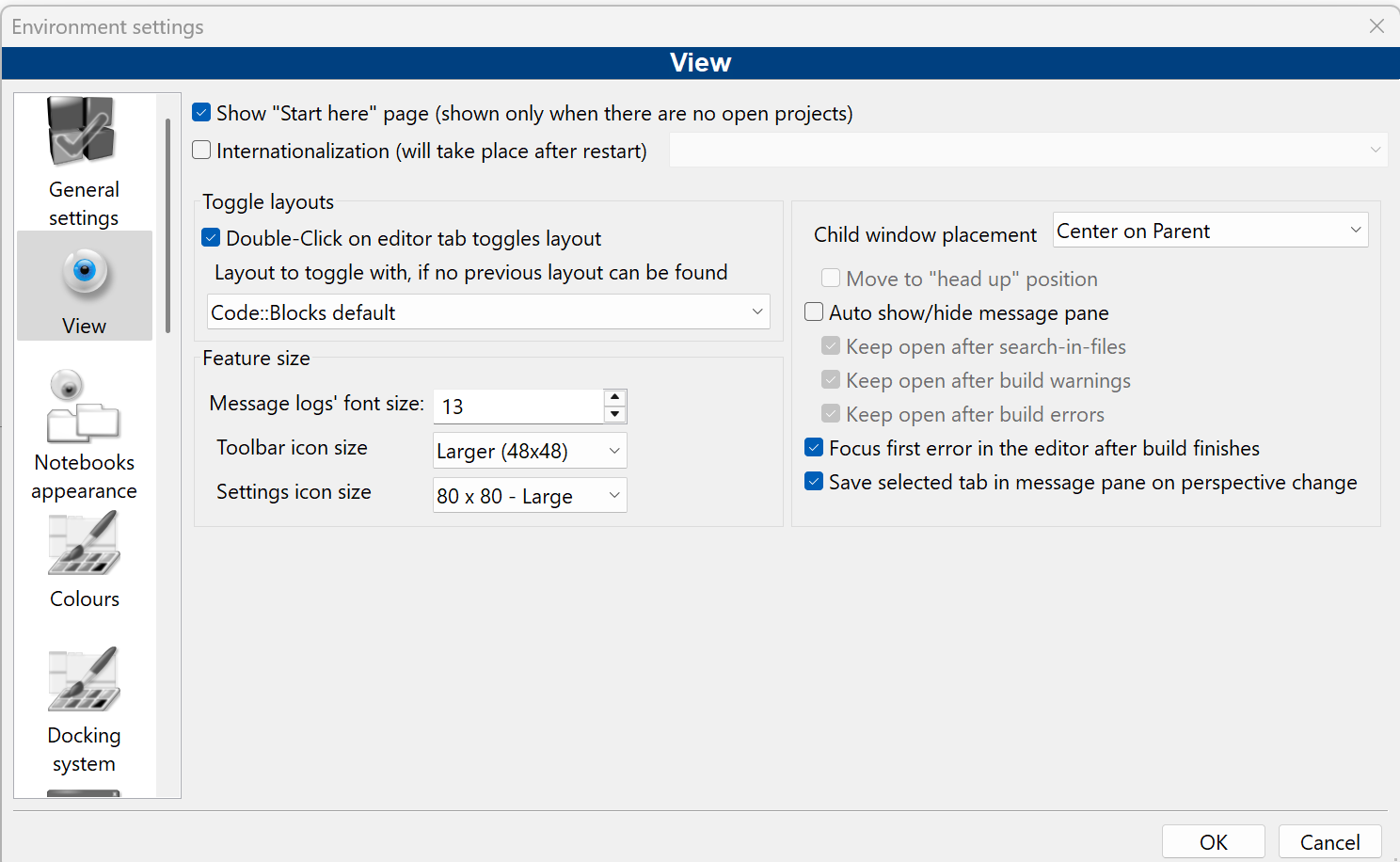
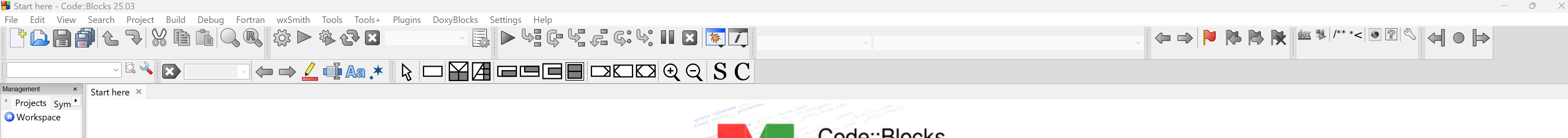


Figure 16 - Increasing the toolbar icon sizes

* Click “OK”
* Select File 🡺 Save Everything
* Restart Code::Blocks



I also increased the font size used by:

* Select Settings 🡺 Editor

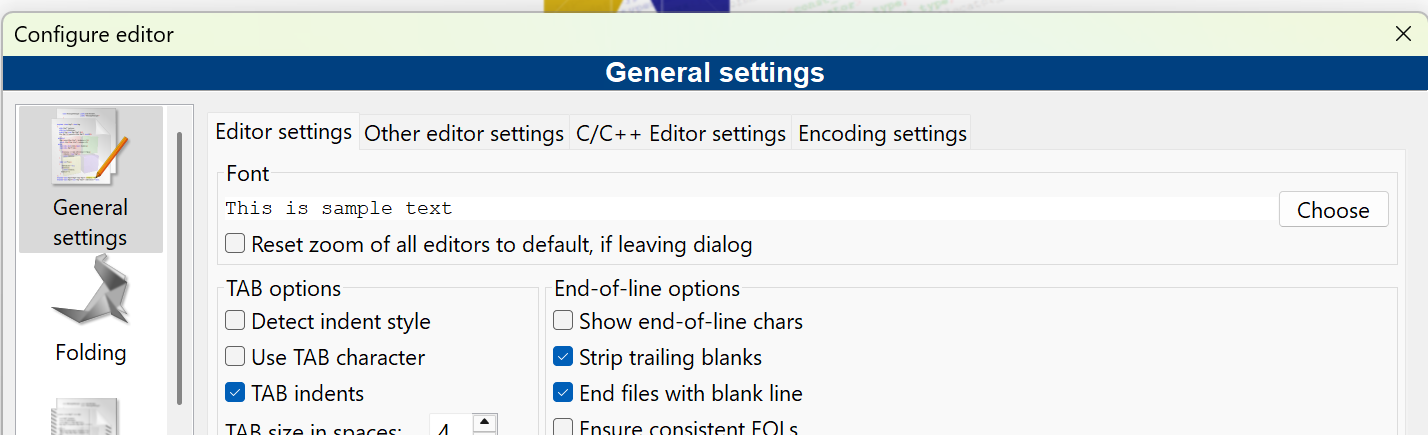


Figure 17 - Updating font-size

* Click on “Choose” button

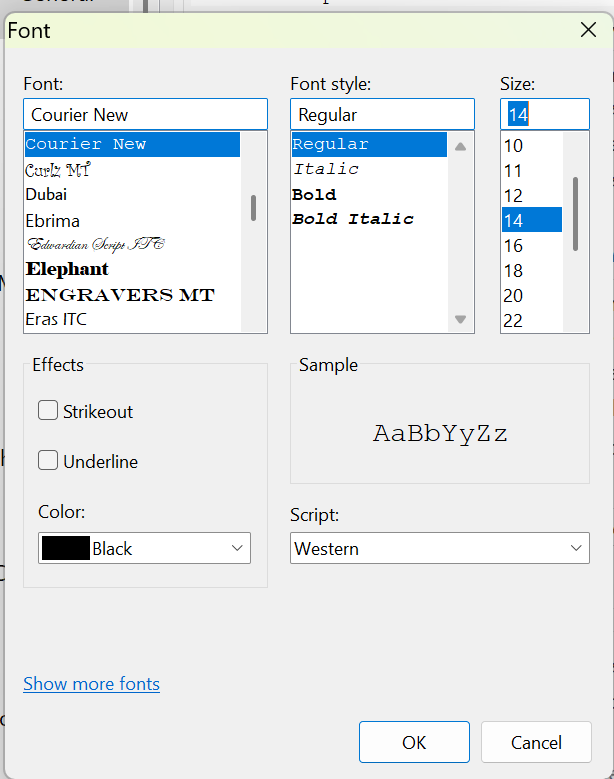


Figure 18 - Select desired font size

* I selected 14 and clicked on “OK”

### Creating the Project

* Click on “Create a new project” link



Figure 19 - Creating a new Project

* Select “Console Application”

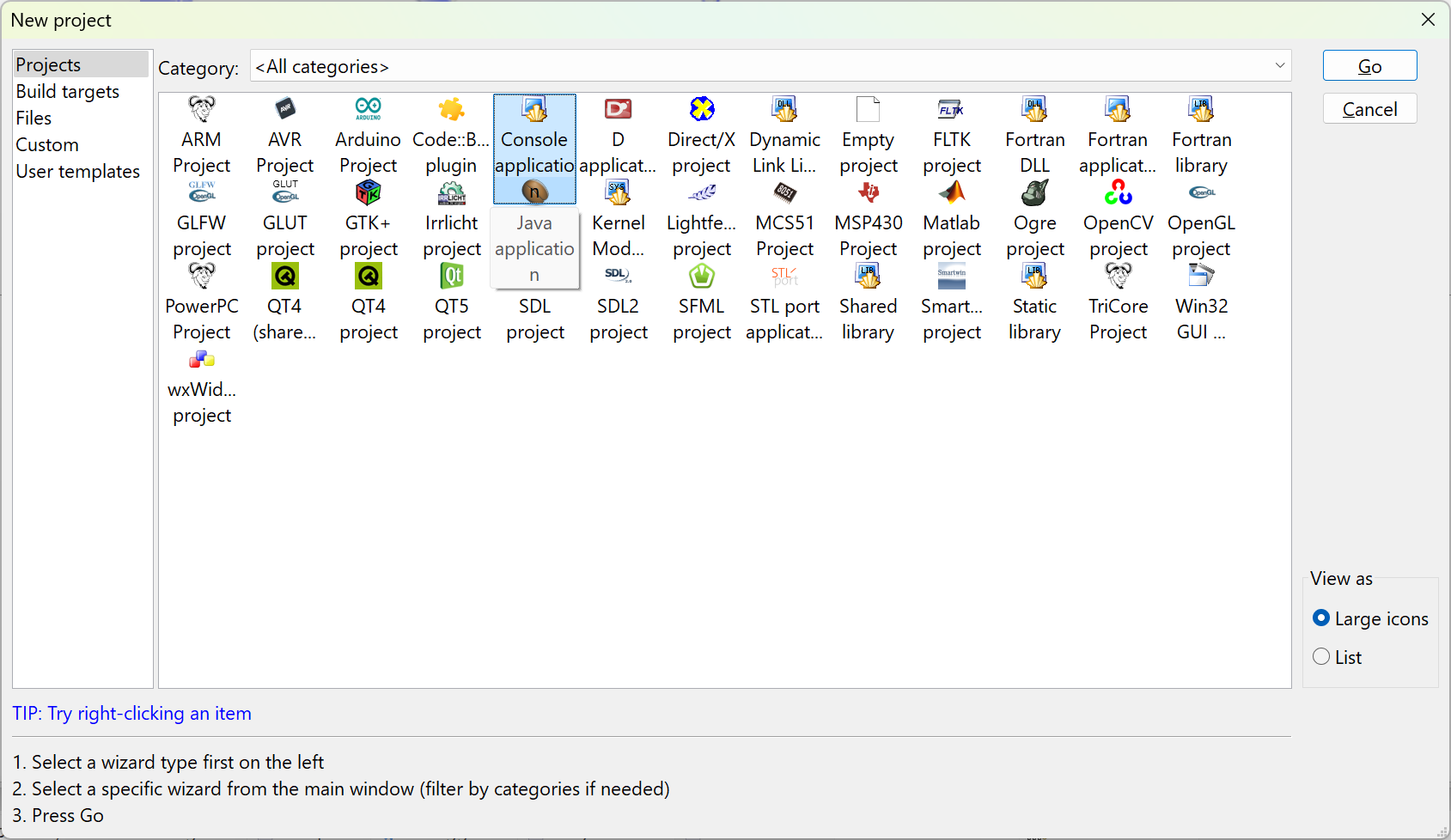


Figure 20 - Creating our first console application

* Click on “Go”, if this is your first time you will see the following dialog box appear:

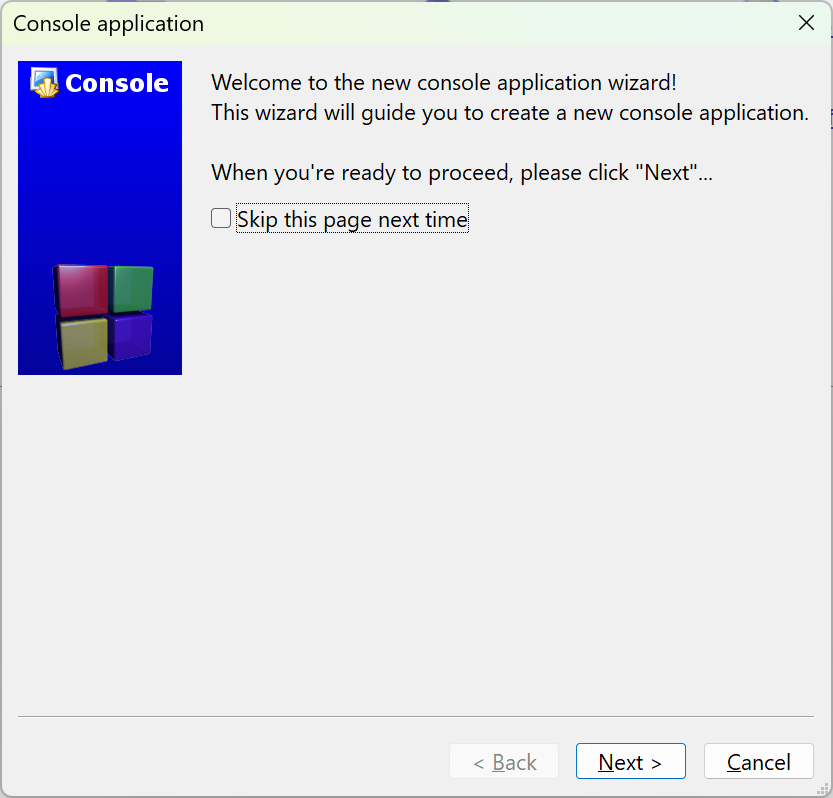
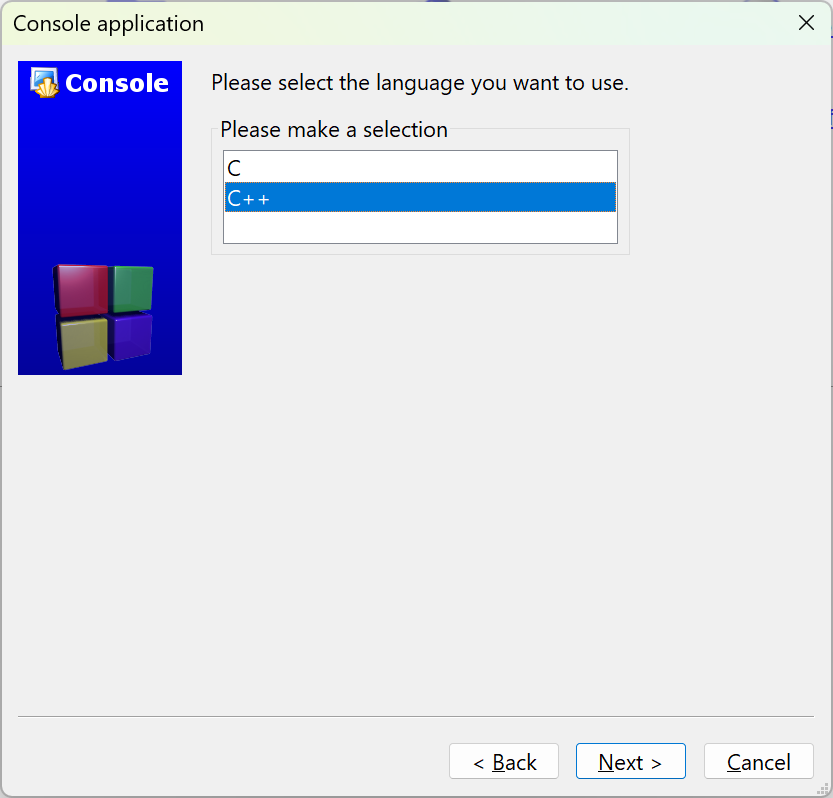


Figure 21 - The Console application wizard starting...

* Click on “Skip this page next time” and press on “Next >”



* Take the default C++ and click on “Next >”
* Fill in the Project information:

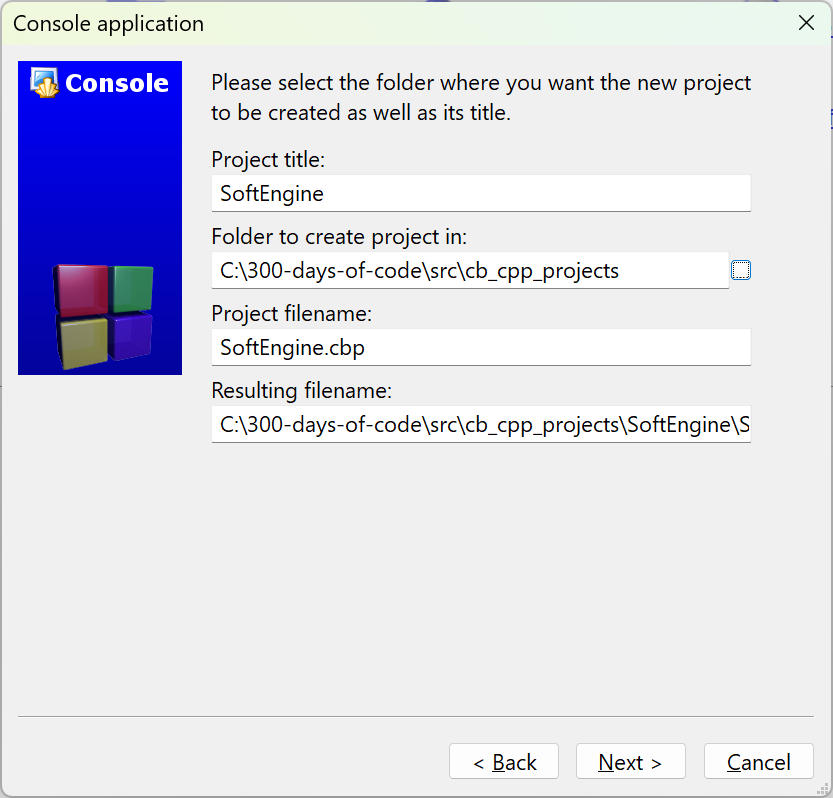


Figure 22 - Project information

The Project name will be SoftEngine. Since this is part of my 300-days-of-code effort I select a folder where I plan on placing all my Code::Blocks C++ projects. Enter a location that makes sense for your setup.

* Click on “Next >”

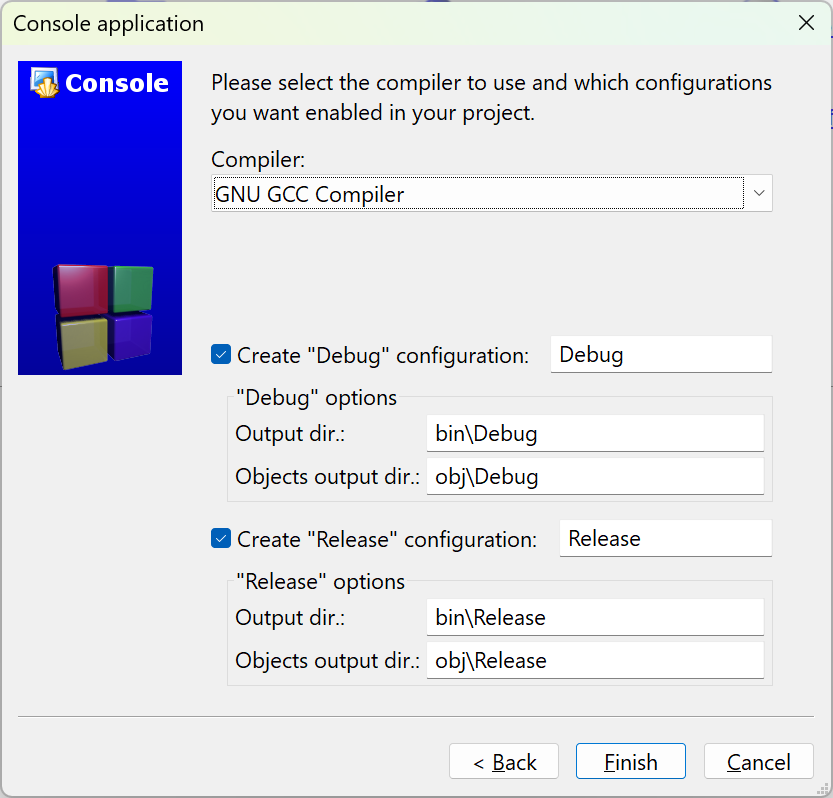


Figure 23 - Final step in creating our game engine project

* Make sure the “Debug” and “Release” configuration are selected and click on “Finish”

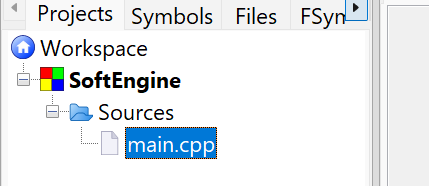


Figure 24 - Default main.cpp is created

* Following the video author’s convention, let’s rename the file main.cpp to Main.cpp
  + Right-click on the filename and select Rename file:

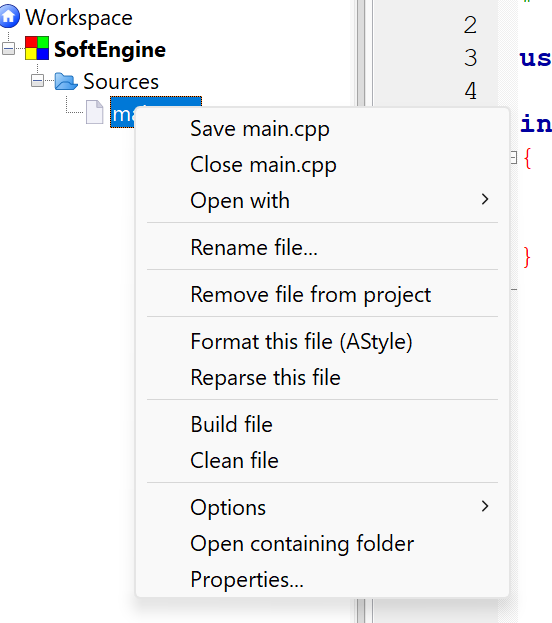


Figure 25 - Context menu for file

* Enter the name Main.cpp:

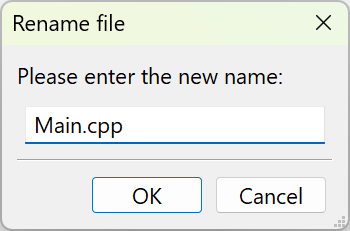


Figure 26 - Rename dialog box

* Click “OK”

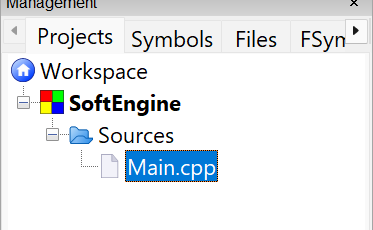


Figure 27 - Update name in Workspace

### Build and Run the Program

The ***Workspace*** contains one or more projects, in our case it shall only contain one project – SoftEngine. The workspace is the most top-level container. A ***project*** contains one or more build targets and the project’s files.

You should familiarize yourself with the following icons/operations:

A diagram of a program

AI-generated content may be incorrect.

Figure 28 - Common operations

* **Build** – this function compiles your source code into an executable program. It processes the code files in your project, checks for errors, and links them to create a standalone application. This is where you written code is transformed into something that can run on your machine.
* **Run** – this function executes the program that you’ve built. If the program is a console application (which is true in our case), it will open a terminal or console window to display the output.
* **Build and Run** – it combines the two steps above
* **Rebuild** – This is a more thorough version of Build. While “Build” compiles *only* the files that have been modified since the last build, “Rebuild” forces the *entire* project to be recompiled from scratch, regardless of whether files have changed.
* **Abort** – this command is used to stop an ongoing build or compilation process. It is specifically for interrupting the build process.

#### Let’s Build and run!

Our Main.cpp code is:

1. #include <iostream>

2.

3. using namespace std;

4.

5. int main()

6. {

7. cout << "Hello world!" << endl;

8. return 0;

9. }

10.

The code above is our simple “Hello, world!” program.

* Click on the “Build and Run” icon

A screenshot of a computer

AI-generated content may be incorrect.

Figure 29 - Result of building and running our Main.cpp

* Press any key on the keyboard to dismiss

## Install and Setup SDL

### What is SDL?

SDL, or ***Simple DirectMedia Layer***, is a cross-platform software development library designed to provide a hardware abstraction layer for multimedia hardware components. It was originally created by Sam Lantinga in 1998. SDL is widely used for developing high-performance computer games and multimedia applications across various operating systems, including Android, iOS, Linux, macOS, and Windows.

The library is written in C and provides an application programming interface (API) in C, with bindings available for other programming languages. Over the years, SDL has evolved significantly, with major updates like SDL 2.0 in 2013, which introduced better support for 3D hardware acceleration. SDL 3.0, released in January 2025, brought further enhancements and new features.

SDL is free and open-source software, licensed under the zlib License since version 2.0, allowing developers to use it in both open-source and closed-source projects. It has been extensively used in the industry, with hundreds of games, applications, and demos built using the library.

The website to obtain more information is: <https://www.libsdl.org/>

### Installing SDL2 and SDL2\_image

* Create a folder that will hold both SDL2 and SDL2\_image download files. I will create the folder D:\SDL2\_dev folder.
* Download the latest SDL2
  + Go to: <https://github.com/libsdl-org/SDL>

A screenshot of a computer

AI-generated content may be incorrect.

Figure 30 - Versions of sdl on Github

The sdl2 link displays a list of projects that uses sdl2.

* Go to <https://github.com/libsdl-org/SDL/releases/tag/release-2.32.4> to get the latest SDL2 version.

Note: We will be using the latest version of SDL – SDL2.32.4. The video series uses SDL2-2.0.10.

Another note: I prefer to learn and use SDL3 but there are too many differences between SDL2 and SDL3 that will introduce another set of issues that will get in the way of the video series!

* Navigate to the release github url (for me it is: <https://github.com/libsdl-org/SDL/releases/tag/release-3.2.10>)

#### What version of SDL2 should I use?

A screenshot of a computer

AI-generated content may be incorrect.

Figure 31 - Versions of SDL3 to choose from

Since I am using Code::Blocks with mingw I will utilize SDL2-devel-2.32.4-mingw.zip version. If you are using Visual Studio you should download and install SDL2-devel-2.32.4-VC.zip

Note: If you want to utilize SDL2 you can find the latest SDL2 release on the same github website: <https://github.com/libsdl-org/SDL/releases>

* Unzip the file[[1]](#footnote-1) to D:\SDL2\_dev:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 32 - Unzipping the SDL2 version

You will see:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 33 - Unzipped SDL2 file

* Open the INSTALL.md file to determine which of the folders you will use:

A number with numbers and lines

AI-generated content may be incorrect.

Figure 34 - Contents of INSTALL.txt file

I will be using the 64-bit architecture of the files in x86\_64-w64-mingw32. Select the folder that makes sense for your machine and setup.

* Go to [https://github.com/libsdl-org/SDL\_image/releases to obtain the 2.8.9](https://github.com/libsdl-org/SDL_image/releases%20to%20obtain%20the%202.8.9) release of SDL2\_image

A screenshot of a computer

AI-generated content may be incorrect.

Figure 35 - SDL2\_image choices

* I downloaded SDL2\_image-devel-2.8.8-mingw.zip to be consistent with the version of SDL2 I downloaded.
* Unzip in the D:\SDL2\_dev folder

You will see the following two top-level folders in D:\SDL2\_dev:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 36 - Top-level folders

Note: Unlike the video – I will leave the files in the location I unzipped them.

### Setting up Code::Blocks to access SDL folders

* Open the project (if not open) we created SoftEngine. To re-open the project
  + Open Code::Blocks
  + Click on “Open an existing project”
  + Navigate to the folder you placed your project and select SoftEngine.cbp and click “Open”

A screenshot of a computer

AI-generated content may be incorrect.

Figure 37 - Opening an existing project

* Go to Settings 🡺 Compiler…

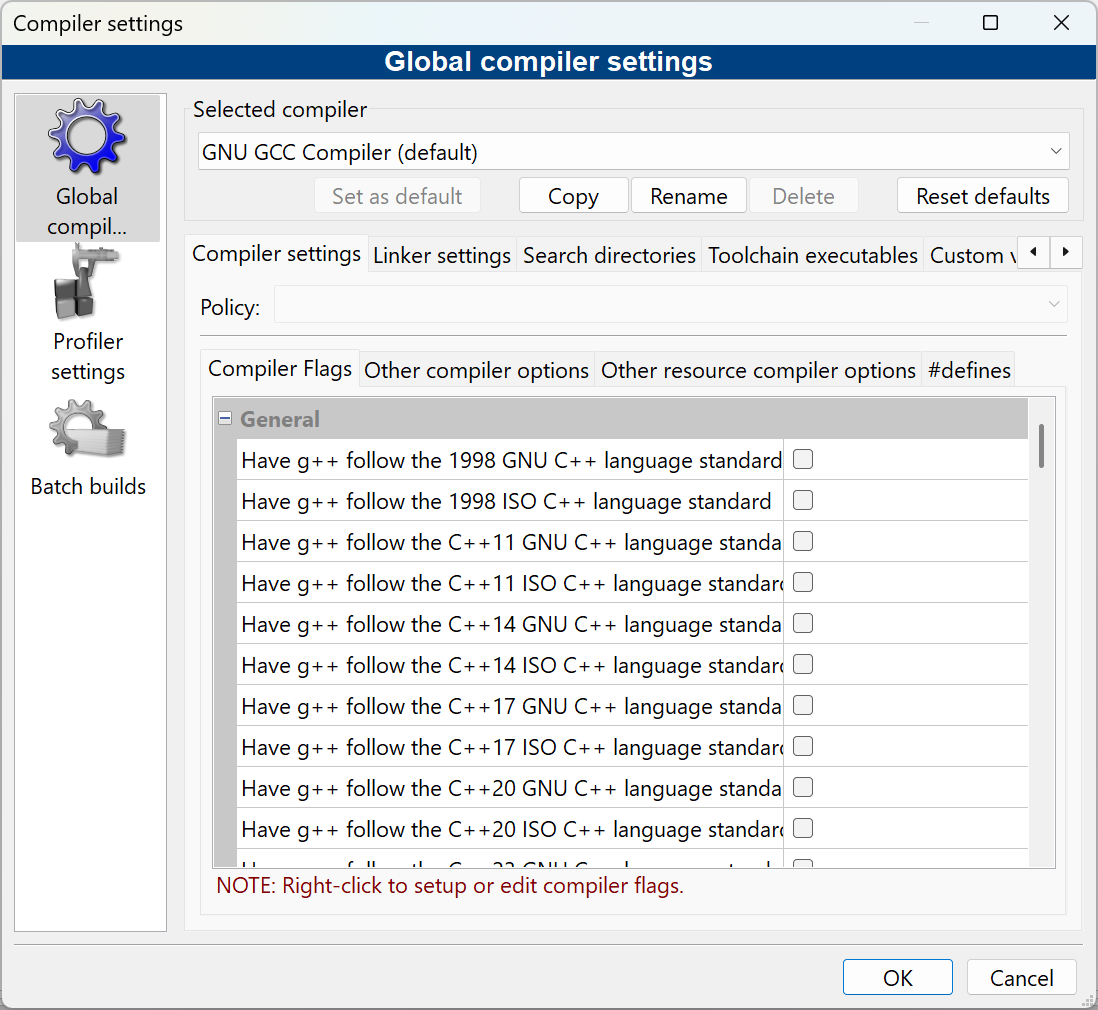


Figure 38 - Global compiler settings

* Click on the “Linker settings” tab

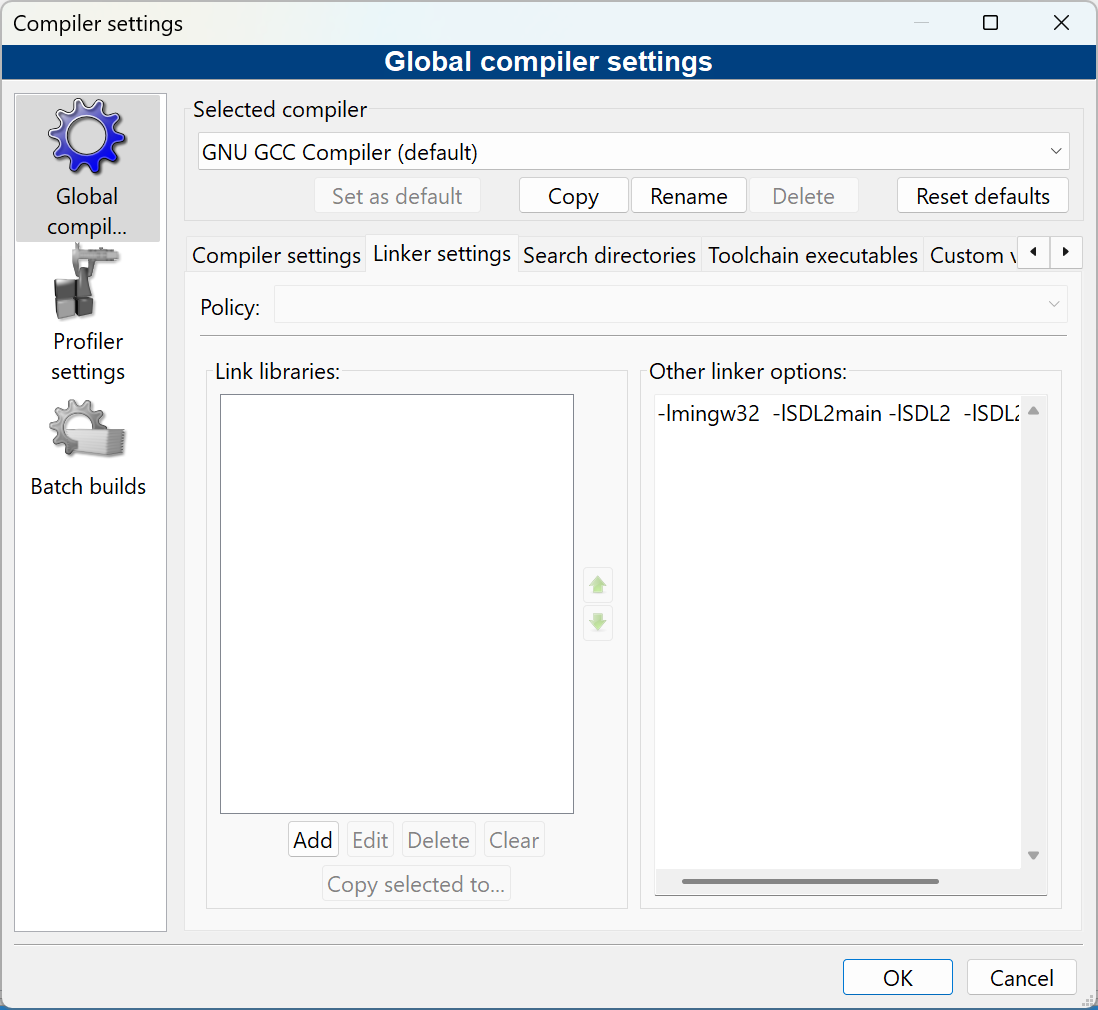
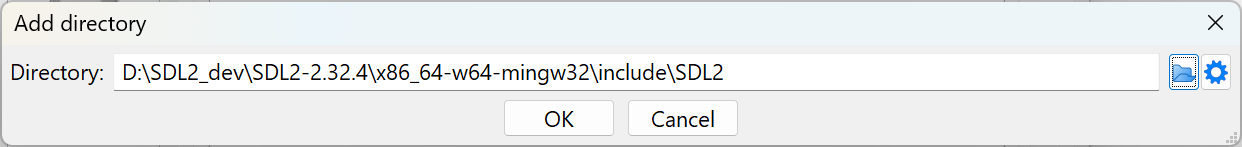


Figure 39 - Adding linker options

* In the “Other linker options:” input box enter:

-lmingw32 -lSDL2main -lSDL2 -lSDL2\_image

* Click on the “Search directories” tab
* Make sure the “Compiler” tab is selected
* Click on “Add” and navigate to the include folder for SDL



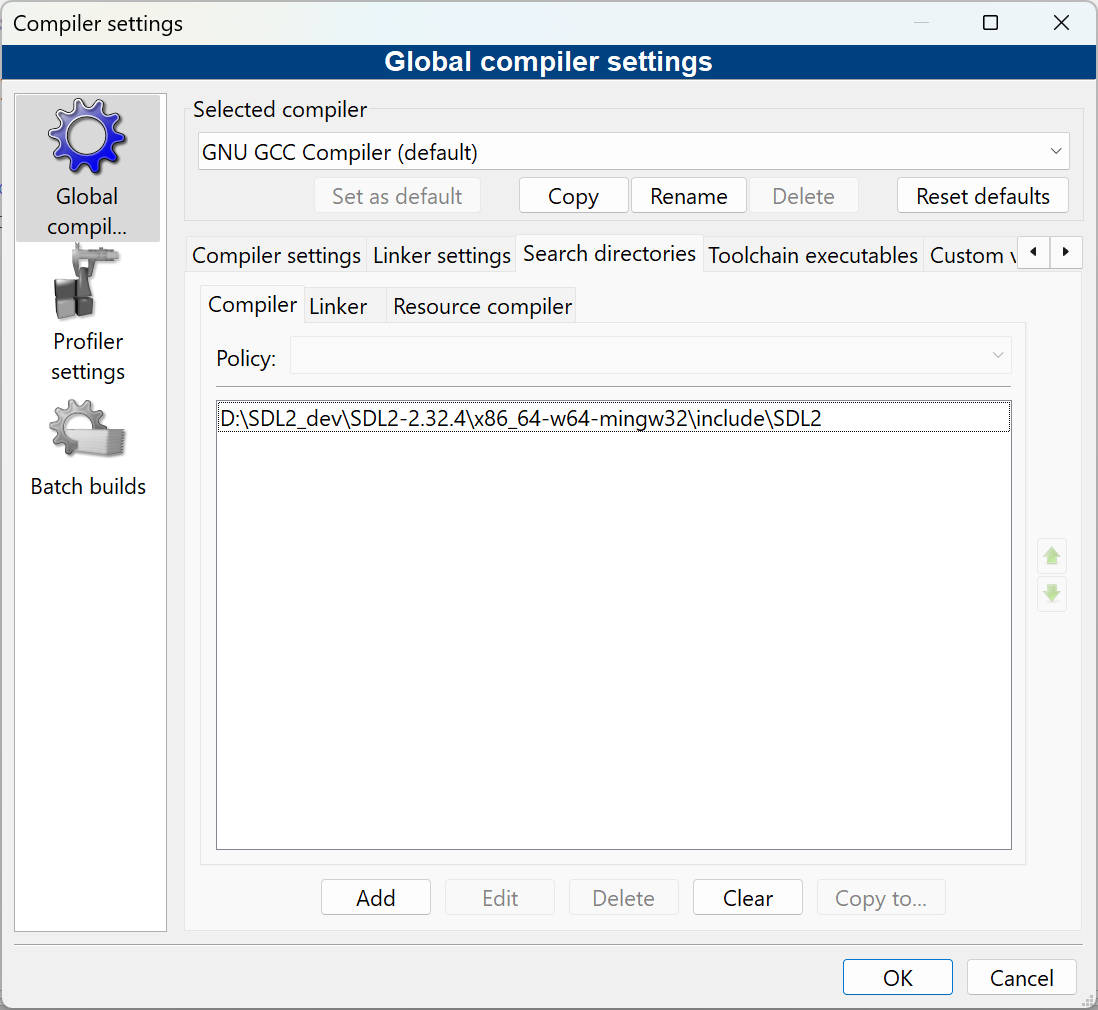


Figure 40 - Adding SDL3 include folder for the compiler

* Add the SDL2\_image include as well

A screenshot of a computer

AI-generated content may be incorrect.

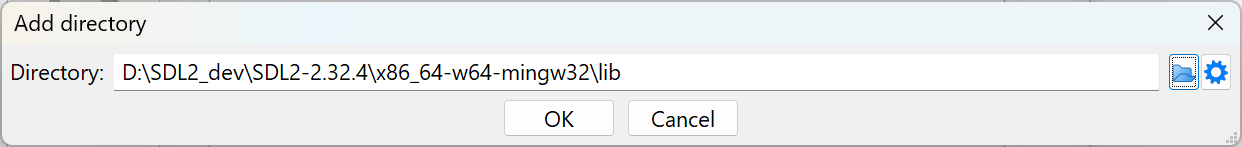
* Select the “Linker” sub-tab and add the location of the lib folder:

A screenshot of a search box

AI-generated content may be incorrect.

Figure 41 - Linker sub-tab

* Navigate to the SDL2 lib folder



* Click on OK
* Add lib folder under SDL\_Image directory

A screenshot of a computer

AI-generated content may be incorrect.

Figure 42 - SDL2 and SDL2\_image lib folders for the linker

### Testing the setup

* Update the code as follows:

1. #include <iostream>

2. #include "SDL.h"

3.

4. using namespace std;

5.

6. int main(int argc, char\*\* argv)

7. {

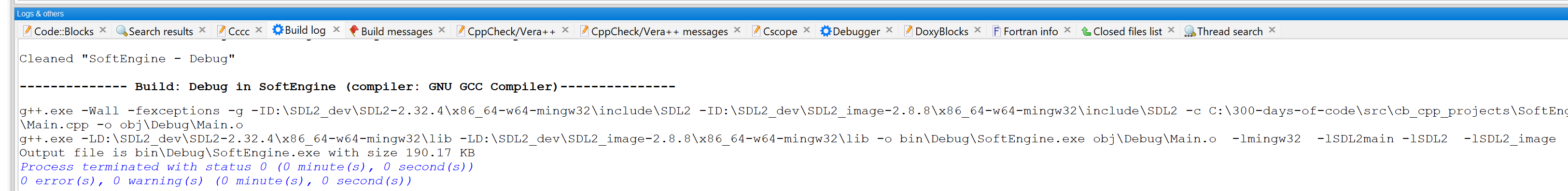
8. cout << "Hello world!" << endl;

9. return 0;

10. }

11.

* Click on the “Build” or “Rebuild”



You should see no errors.

The video uses SDL\_INIT\_EVERYTHING to initialize all the SDL subsystems at the same time. This is considered a bad practice and the flag SDL\_INIT\_EVERYTHING no longer exists in SDL3. We will follow the video but note that it does not exist in SDL3.

* Let’s initialize SDL to make sure everything runs correctly. Update the Main.cpp:

1. #include <iostream>

2. #include "SDL.h"

3.

4. using namespace std;

5.

6. int main(int argc, char\*\* argv)

7. {

8. if (SDL\_Init(SDL\_INIT\_EVERYTHING) == 0) {

9. cout << "SDL\_Init worked!" << endl;

10. }

11. SDL\_Quit(); // clean up resources

12. return 0;

13. }

* Try to build it again. It should build.
* Now, try to run the application. It fails:

A screen shot of a computer error

AI-generated content may be incorrect.

Figure 43 - Application failed to find SDL3.dll file

The problem is that the application could not find the SDL2.dll file that is located in the SDL2 \bin folder:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 44 - The location of the SDL2.dll file

* The video presenter suggests putting the dll files for SDL2 and SDL2\_image in the project debug directory:

A screenshot of a computer

AI-generated content may be incorrect.

I usually have a D:\bin or a C:\bin directory where I put in all my generic tools and common dlls that will be utilized across many projects:

A screenshot of a computer

AI-generated content may be incorrect.I recommend that a more general location be created similar to the above and the folder be placed in the environment path.

Which ever you choose, once the \*.dll files of SDL2.dll and SDL2\_image.dll are made accessible the program should now execute successfully:

A screen shot of a computer

AI-generated content may be incorrect.

Figure 45 - Program with SDL component running ran successfully

# 2. Game Loop for SDL Game

In this video we implement the fundamental structure in every game and game engine – the **Game Loop**. We will add a new class – Engine.cpp (and Engine.h) that will have all the key operations that a game loop goes through. Our Main.cpp class will be modified to invoke the key game loop functions.

## Opening the Project

If you are opening the project anew for the next video a fast way is to select File 🡺 Recent projects and select the project you are working on .

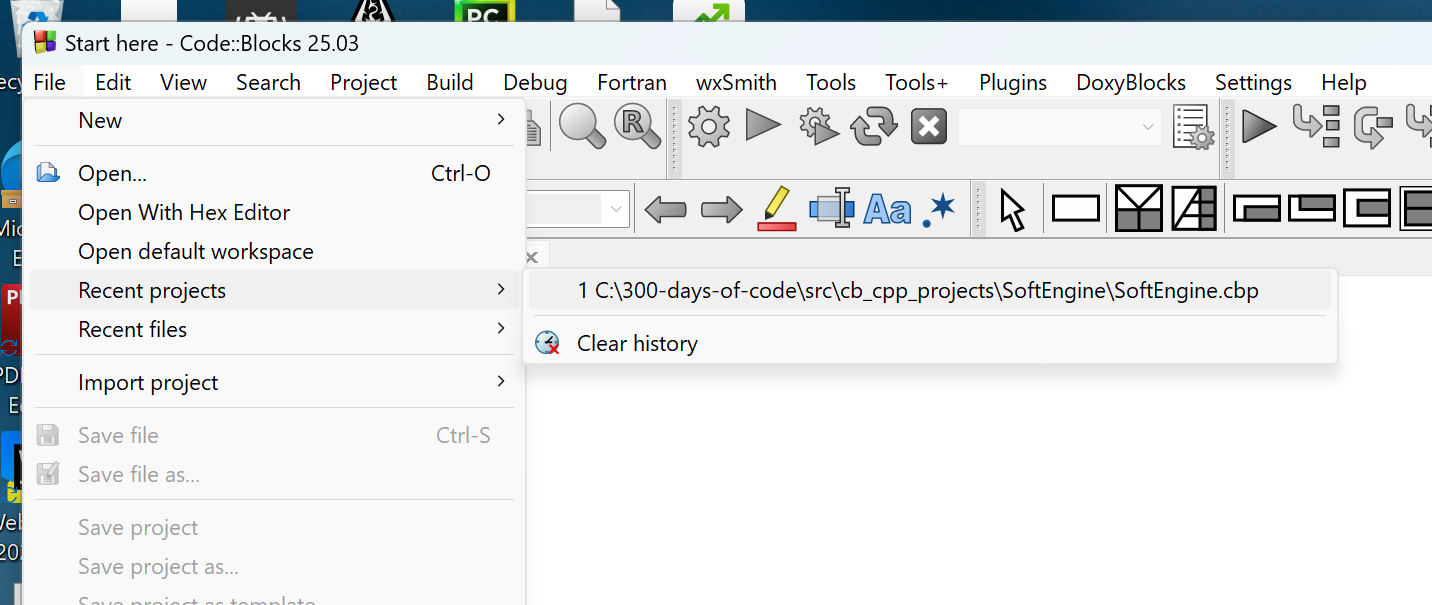


Figure 46 - Selecting a recent project

The goal of these set of videos is to create a 2D game engine that appears as follows:



Figure 47 - The final look of our 2D Game Engine

TBD: Replace with an unmarked image

## What is a game loop?

A game loop is the fundamental, repeating process that updates and renders a game’s state. It’s the core of how a game runs, constantly updating the game world, handling player inputs (via events) and drawing the graphics to the screen.

The core components of a Game Loop:

* Input: Processes player input from various sources like keyboard, mouse, or controller
* Update: Update the game’s state, including physics, AI, and other game logic
* Render: Draws the updated game world to the screen

**Initialization:**

The game loop starts with an initialization phase where the game sets up its resources, including game states, graphics, and input systems.

**Loop Execution:**

The loop then repeatedly executes these phases:

* Process Input – The game detects player input and updates the corresponding entities.
* Update Game World – The game logic is applied, updating the state of the game world based on player input and game rules
* Render Graphics – The game renders the updated game world to the screen, creating the visual display for the player

**Loop Continues:**

This process continues until the game is closed or the loop’s condition is no longer met (player won or player died).

Importance of the Game Loop:

* Smooth Gameplay: The game loop ensures the game runs smoothly and consistently, providing a fluid and responsive experience for the player
* State Management: The loop is responsible for managing the game’s state, ensuring it remains consistent and up-to-date
* Foundation of Game Development: It’s fundamental structure upon which most game development engines are built.

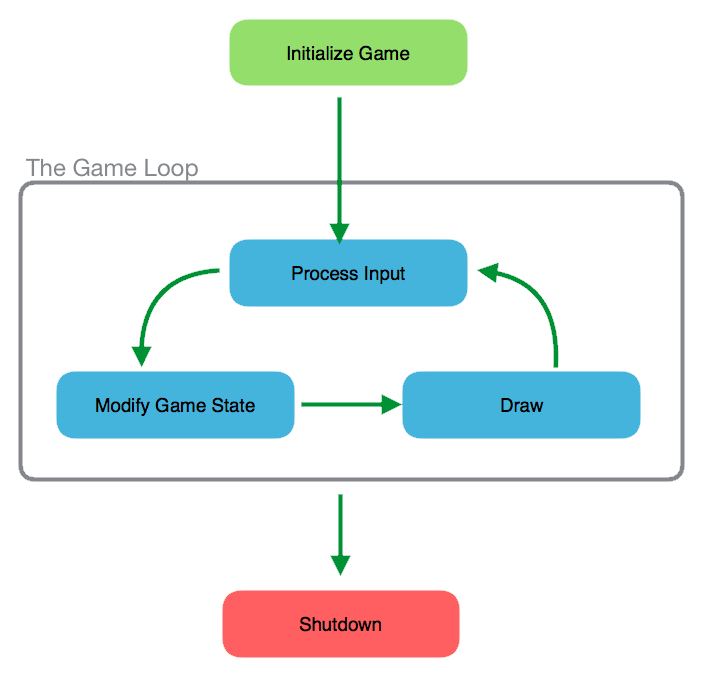


Figure 48 - A diagram of the "Game Loop"

## Create an Engine class file

We will create an Engine.cpp and Engine.h C++ class that will capture the actions we want to implement a game loop.

* Select File 🡺 New 🡺 Class…

A screenshot of a computer

AI-generated content may be incorrect.

Figure 49 - Adding a new class to the project

* Fill in the “Create new class” dialog as shown:

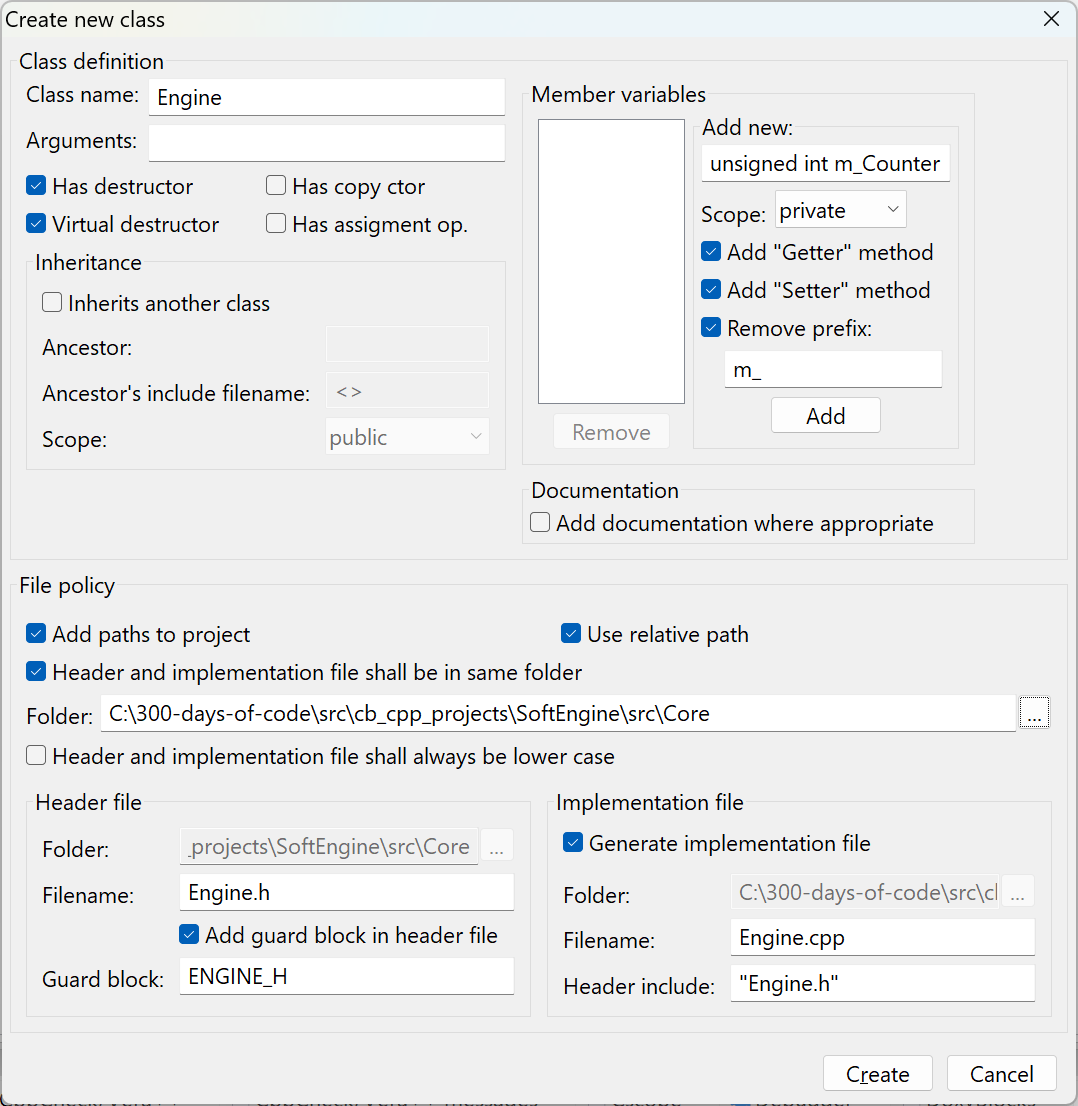


Figure 50 - Creating our Engine.cpp and Engine.h files

Note, the Folder for our new class is \src\Core. All the classes we will create for this project will be in their own folder under the src directory.

* Click on “Create”

A screenshot of a computer

AI-generated content may be incorrect.

Figure 51 - Add to project prompt?

* Click “Yes” to the above prompt.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 52 - Add to Debug and Release prompt

* Click “OK” to the above prompt.

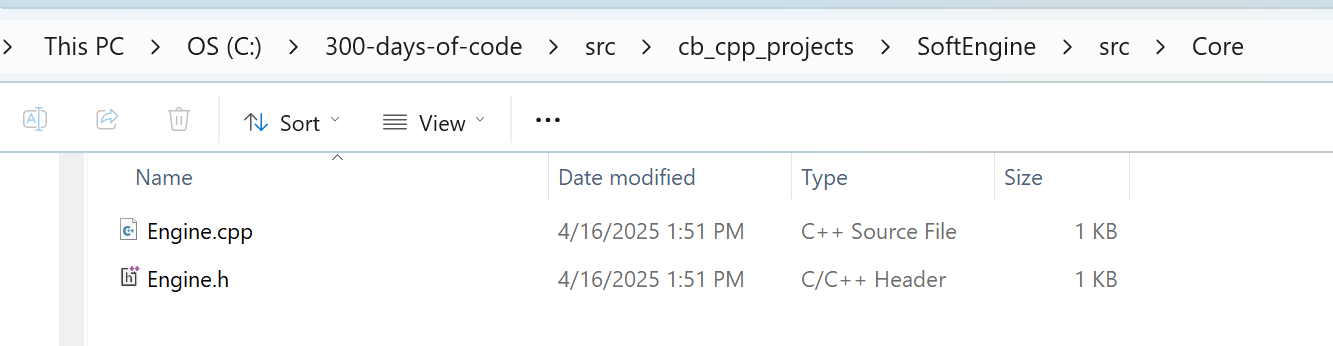
The project will appear as:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 53 - Project view of the files

In fact, if you examine your folders:



You see that the Engine.cpp and Engine.h are actually in the same folder. The presenter likes to see the files together in the project view. Right-click on the SoftEngine project and select “Project tree” 🡺 “Categorize by file types”.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 54 - Viewing the files as in actual folders

* Check “Categorize by file types” off

The result is:

A screenshot of a computer

AI-generated content may be incorrect.

**Engine.cpp** (initial):

1. #include "Engine.h"

2.

3. Engine::Engine()

4. {

5. //ctor

6. }

7.

8. Engine::~Engine()

9. {

10. //dtor

11. }

The Engine.cpp file created has an empty constructor and empty destructor defined. This is the implementation class.

**Engine.h** (initial):

1. Engine.h (initial):

2. #ifndef ENGINE\_H

3. #define ENGINE\_H

4.

5.

6. class Engine

7. {

8. public:

9. Engine();

10. virtual ~Engine();

11.

12. protected:

13.

14. private:

15. };

16.

17. #endif // ENGINE\_H

18.

We want our Engine class to be a Singleton class.

|  |
| --- |
| **What is a Singleton class?**  A Singleton class in C++ is a design pattern that ensures a class has only ***one instance*** throughout the program and provides a global point of access to that instance. This is useful in cases where you need centralized management of a resource, like a configuration manager or a logging system.  The Singleton pattern is typically implemented by:   1. Using a private static pointer to the single instance of the class. 2. Making the constructor private to prevent the creation of multiple instances. 3. Providing a public static method that returns the single instance (creating it if it doesn't exist).   Here's a simple example:  1. #include <iostream>  2. #include <memory>  3.  4. class Singleton {  5. private:  6. static std::unique\_ptr<Singleton> instance; // Static pointer to the instance  7. Singleton() {} // Private constructor  8.  9. public:  10. Singleton(const Singleton&) = delete; // Prevent copying  11. Singleton& operator=(const Singleton&) = delete; // Prevent assignment  12.  13. static Singleton& getInstance() {  14. if (!instance) {  15. instance = std::make\_unique<Singleton>();  16. }  17. return \*instance;  18. }  19.  20. void displayMessage() {  21. std::cout << "Singleton instance accessed!\n";  22. }  23. };  24.  25. std::unique\_ptr<Singleton> Singleton::instance = nullptr; // Initialize the static pointer  26.  27. int main() {  28. Singleton& singleton = Singleton::getInstance();  29. singleton.displayMessage();  30.  31. return 0;  32. }  33.  The example provided is compatible with **C++11** and later versions. Features like std::unique\_ptr for memory management and the explicit use of delete to prevent copying were introduced in C11. If you use this code with compilers supporting C11 or newer (e.g., C14, C17, C++20), it should work seamlessly. |

We only want to have one instance of the Engine class.

### Setting the compiler to use C++ 17

We probably should have done this earlier (in the first video) but we need to ensure that we use C++ 17 to match the presenter’s version.

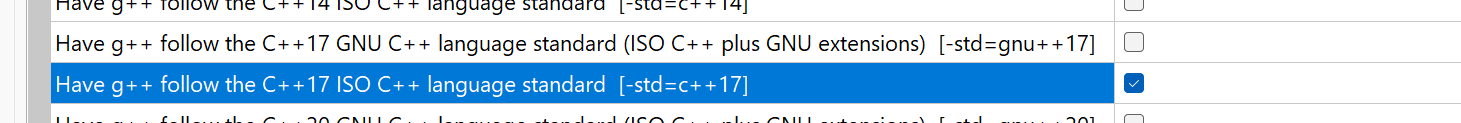
* Click on Settings 🡺 Compiler…

We have several choices here:

A screenshot of a computer

AI-generated content may be incorrect.

The one that matches is:



### Making Engine a Singleton

**Engine.h:**

1. class Engine

2. {

3. public:

4. Engine();

5.

6. static Engine\* GetInstance() {

7. return s\_Instance = (s\_Instance != nullpter) ? s\_Instance : new Engine();

8. }

9. virtual ~Engine();

10.

11. protected:

12.

13. private:

14. static Engine\* s\_Instance;

15.

16. };

17.

We create a static class method that either returns an Engine\* or creates it. The Engine instance is saved as a pointer in s\_Instance.

We will only ever have on instance of our game Engine, therefore we will move the constructor into the private section.

1. class Engine

2. {

3. public:

4. static Engine\* GetInstance() {

5. return s\_Instance = (s\_Instance != nullpter) ? s\_Instance : new Engine();

6. }

7. virtual ~Engine();

8.

9. protected:

10.

11. private:

12. Engine() {};

13. static Engine\* s\_Instance;

14.

15. };

16.

### Adding key game loop functions to our Engine

The key game loop functions are Init() for initialization of our game, Events() to obtain input events (e.g. mouse move, keyboard entry, etc.), Update() to update the entities according to the game logic, Render() to update the graphical screen. The Clean() function is used to clean up all resources and finally Quit() to terminate the game.

We will also add an inline isRunning() function to be used for our game loop, as long as the member variable m\_IsRunning is true, we execute a cycle of the game loop.

#### Adding to Engine.h:

1. #ifndef ENGINE\_H

2. #define ENGINE\_H

3.

4.

5. class Engine

6. {

7. public:

8. static Engine\* GetInstance() {

9. return s\_Instance = (s\_Instance != nullptr) ? s\_Instance : new Engine();

10. }

11.

12. bool Init();

13. bool Clean();

14. void Quit();

15.

16. void Update();

17. void Render();

18. void Events();

19.

20. inline bool isRunning() {

21. return m\_IsRunning;

22. }

23.

24. protected:

25.

26. private:

27. Engine();

28. static Engine\* s\_Instance;

29. bool m\_IsRunning;

30.

31. };

32.

33. #endif // ENGINE\_H

34.

There are two class methods that are defined in Engine.h:

* GetInstance()
* IsRunning()

### Adding to Engine.cpp

To get Code::Blocks to automatically add implementation functions for all the missing functions do the following:

* Right-click on the Engine.cpp page
* Select Insert/Refactor
* Select “All class method without implementation…”

A screenshot of a computer

AI-generated content may be incorrect.

Figure 55 - Adding missing class methods

* Select all the methods you want the code to seed your class with:

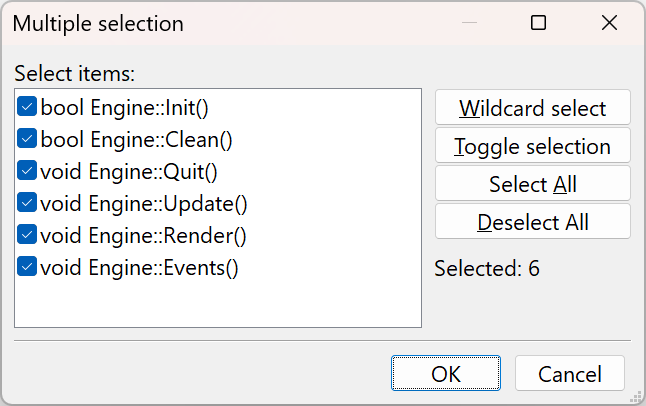


Figure 56 - Selecting the class methods to insert

* Click “OK”

Engine.cpp:

1. #include "Engine.h"

2.

3. Engine::Engine()

4. {

5. //ctor

6. }

7.

8. bool Engine::Init()

9. {

10.

11. }

12.

13. bool Engine::Clean()

14. {

15.

16. }

17.

18. void Engine::Quit()

19. {

20.

21. }

22.

23. void Engine::Update()

24. {

25.

26. }

27.

28. void Engine::Render()

29. {

30.

31. }

32.

33. void Engine::Events()

34. {

35.

36. }

Updated Main.cpp:

1. #include "Engine.h"

2. #include <iostream>

3.

4. bool Engine::Init()

5. {

6. }

7.

8. bool Engine::Clean()

9. {

10. }

11.

12. void Engine::Quit()

13. {

14.

15. }

16.

17. void Engine::Update()

18. {

19.

20. }

21.

22. void Engine::Render()

23. {

24. }

25.

26. void Engine::Events()

27. {

28. }

29.

Let’s now add minimal code to ensure that the above will work.

We will:

* Set s\_Instance to nullptr;
* Insert std::cout statements to each function
* Add logic to Init() to set m\_IsRunning

**Engine.cpp:**

1. #include "Engine.h"

2. #include <iostream>

3.

4. Engine\* Engine::s\_Instance = nullptr;

5.

6. bool Engine::Init()

7. {

8. std::cout << "Initializing..." << std::endl;

9. m\_IsRunning = true;

10. return true;

11. }

12.

13. bool Engine::Clean()

14. {

15. std::cout << "Clean..." << std::endl;

16. return true;

17. }

18.

19. void Engine::Quit()

20. {

21.

22. }

23.

24. void Engine::Update()

25. {

26. std::cout << "Updating..." << std::endl;

27. }

28.

29. void Engine::Render()

30. {

31. std::cout << "Render..." << std::endl;

32. }

33.

34. void Engine::Events()

35. {

36. std::cout << "Events..." << std::endl;

37. }

38.

Now we implement the game loop in Main.cpp by invoking the methods in our Engine:

Main.cpp

1. #include "Engine.h"

2.

3. int main(int argc, char\*\* argv)

4. {

5. Engine::GetInstance()->Init();

6.

7.

8. while (Engine::GetInstance()->isRunning()) {

9. // Get all current events (e.g. mouse clicks, etc.)

10. Engine::GetInstance()->Events();

11.

12. // Update all objects/entities

13. Engine::GetInstance()->Update();

14.

15. // Render/update the game graphics

16. Engine::GetInstance()->Render();

17. }

18.

19. // Clean everything up

20. Engine::GetInstance()->Clean();

21.

22. return 0;

23. }

24.

The code above implements the game loop! It does not do anything right now but print over and over again the std::cout messages associated with each Engine method.

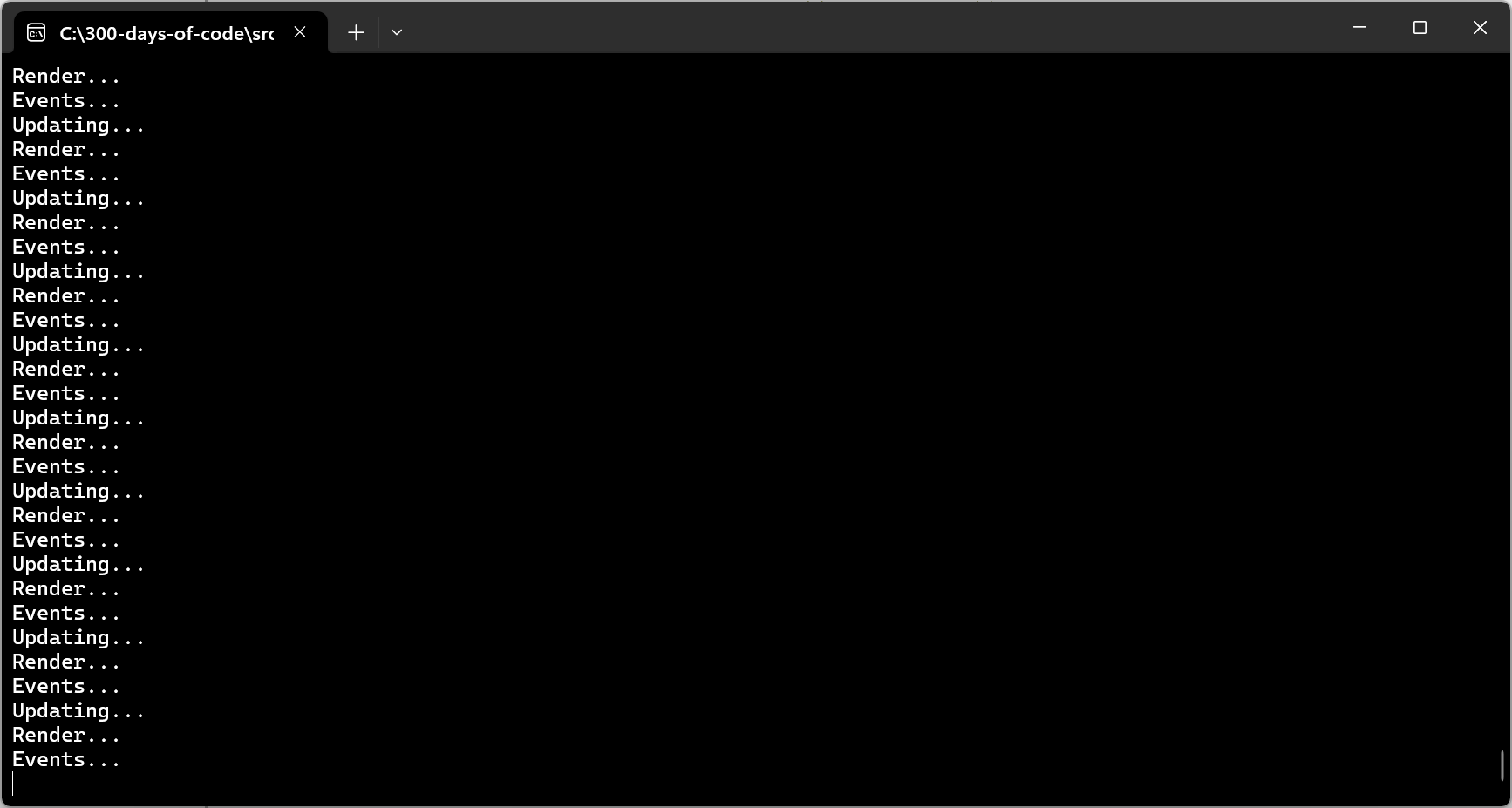


Figure 57 - Running our game loop

Note: If you followed the video you may have encountered an issue when you tried to run the program. The problem was due to the missing bool return values from the methods Init() and Clean().

# 3. Create SDL Window and Renderer

This video starts the journey of filling in the components of game loop by

* Initializing SDL
* Creating our game window with SDL
* Creating the ability to draw in our window
* How to end our game

We cover/edit the key Engine.cpp methods in this video.

The first file we will edit is Engine.h. We will add statements to the page to allow the Engine to access SDL functions:

1. #ifndef ENGINE\_H

2. #define ENGINE\_H

3. #include "SDL.h"

4. #include "SDL\_image.h"

The above #include will allow our Engine class to reference SDL and SDL\_image elements.

In the private section we will add two new member class variables for an SDL\_Window and an SDL\_Renderer.

## What is an SDL\_Window?

An **SDL\_Window** is a structure used in the **Simple DirectMedia Layer (SDL)** library to represent a window in your application. It serves as the foundation for rendering graphics, handling events, and interacting with the display. You can create an SDL\_Window using the SDL\_CreateWindow function, specifying parameters like the title, dimensions, position, and flags.

The window can have various properties, such as being resizable, fullscreen, or supporting OpenGL/Vulkan contexts. It also interacts with high-DPI displays and can handle input focus, mouse grabbing, and more.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 58 - An example of a Window (SDL\_Window) from: <https://jakob.space/blog/sdl-tutorial-part-0x00.html>

Now that you have a window you want to draw or render objects on our window for that we create SDL\_Renderer.

## What is an SDL\_Renderer?

An **SDL\_Renderer** is a structure in the **Simple DirectMedia Layer (SDL)** library that handles rendering operations. It is tied to an **SDL\_Window** and is used to draw graphics, such as textures, shapes, and other visual elements, onto the window.

Here are some key points about SDL\_Renderer:

* **Creation**: You can create an SDL\_Renderer using the SDL\_CreateRenderer function, which links it to a specific SDL\_Window.
* **Drawing**: It provides functions like SDL\_RenderCopy to render textures, SDL\_SetRenderDrawColor to set drawing colors, and SDL\_RenderClear to clear the screen.
* **Hardware Acceleration**: SDL\_Renderer can use hardware acceleration if supported by the system, making rendering faster and more efficient.
* **Presentation**: After drawing, you use SDL\_RenderPresent to update the window with the rendered content.

## Adding an SDL\_Window and SDL\_Renderer to Engine.h

We add two new member variables:

1. private:

2. Engine() {}

3. static Engine\* s\_Instance;

4. SDL\_Window\* m\_Window;

5. SDL\_Renderer\* m\_Renderer;

6. bool m\_IsRunning;

Since writing to the screen will be done across many other classes we will add an inline function to to Engine.h to return the m\_Renderer in the public section:

1. inline SDL\_Renderer\* GetRenderer() { return m\_Renderer;}

### Adding SCREEN\_WIDTH and SCREEN\_HEIGHT

A major consideration is what your game screen width and height will be, that is, the aspect ratio to use for your window.

Aspect ratios for your **SDL\_Window** depend on the type of application you're developing and the devices you want to support. Here are some considerations:

1. **Common Aspect Ratios**:
   * **16:9**: Widely used for modern displays, including HD and 4K screens.
   * **4:3**: Suitable for older monitors or retro-style applications.
   * **21:9**: Ideal for ultrawide monitors, often used in gaming.
   * **1:1**: Square aspect ratio, useful for specific design or artistic purposes.
2. **Maintaining Aspect Ratio**:
   * If your application allows window resizing, you can use functions like SDL\_SetWindowAspectRatio to enforce a specific aspect ratio.
   * When rendering, ensure your content scales proportionally to avoid distortion.
3. **Application Context**:
   * For games, 16:9 is a safe choice as it's standard for most displays.
   * For productivity tools or creative software, consider supporting multiple aspect ratios to accommodate different user setups.

The presenter decides to use 960x640:

1. #define SCREEN\_WIDTH 960

2. #define SCREEN\_HEIGHT 640

The resolution **960x640** corresponds to an aspect ratio of **3:2**. This aspect ratio is less common for modern displays, which often favor **16:9** or **4:3**, but it has its own merits depending on your use case:

* **Photography**: The 3:2 aspect ratio is widely used in photography, as it matches the native aspect ratio of many DSLR and mirrorless cameras.
* **Retro or Niche Applications**: If you're designing something with a retro aesthetic or targeting specific devices, 3:2 might be a good fit.
* **Content Scaling**: Keep in mind that using a less common aspect ratio might require additional effort to scale or crop content for modern screens.

If you're developing an SDL application, consider your target audience and devices. For general-purpose apps or games, **16:9** might be a safer choice for compatibility.

## Editing Engine.cpp

* Remove all the std::cout statements in the file Engine.cpp
* Update the Engine::Init() to initialize SDL

1. bool Engine::Init()

2. {

3. if (SDL\_Init(SDL\_INIT\_VIDEO) != 0 && IMG\_Init(IMG\_INIT\_JPG | IMG\_INIT\_PNG) != 0) {

4. SDL\_Log("Failed to initialize SDL: %s", SDL\_GetError());

5. return false;

6. }

7.

8. return m\_IsRunning = true;

9. }

### What is SDL\_Init?

The SDL\_Init() function is used to initialize the **Simple DirectMedia Layer (SDL)** library and its subsystems. It's one of the first functions you call when starting an SDL application. Here's a breakdown of its details:

**Syntax:**

1. int SDL\_Init(Uint32 flags);

**Parameters:**

* **flags**: Specifies which SDL subsystems to initialize. You can combine multiple flags using the bitwise OR operator (|). Common flags include:
  + **SDL\_INIT\_VIDEO**: Initializes the video subsystem.
  + **SDL\_INIT\_AUDIO**: Initializes the audio subsystem.
  + **SDL\_INIT\_TIMER**: Initializes the timer subsystem.
  + **SDL\_INIT\_EVENTS**: Initializes the event handling subsystem.
  + **SDL\_INIT\_JOYSTICK**: Initializes the joystick subsystem.
  + **SDL\_INIT\_GAMEPAD**: Initializes the gamepad subsystem.
  + **SDL\_INIT\_HAPTIC**: Initializes the haptic (force feedback) subsystem.
  + **SDL\_INIT\_EVERYTHING**: Initializes all available subsystems.

**Return Value**

* Returns 0 on success.
* Returns a negative error code on failure. You can use SDL\_GetError() to retrieve a human-readable error message.

The video does not discuss the fact that for every subsystem you initialize you should also close it down in order to release resources SDL\_Quit();

### What is IMG\_Init?

IMG\_Init is a function from the **SDL\_image** library, an extension of SDL that provides support for loading and handling various image formats. This function initializes the SDL\_image library and prepares it to work with specific image formats.

**Syntax**

1. int IMG\_Init(int flags);

**Parameters**

* **flags**: Specifies the image formats to initialize. You can combine multiple flags using the bitwise OR operator (|). Common flags include:
  + **IMG\_INIT\_JPG**: Support for JPEG images.
  + **IMG\_INIT\_PNG**: Support for PNG images.
  + **IMG\_INIT\_TIF**: Support for TIFF images.
  + **IMG\_INIT\_WEBP**: Support for WebP images.

**Return Value**

* Returns a bitmask of the successfully initialized formats.
* If the return value doesn't match the requested flags, it means some formats failed to initialize.

**Notes**

* You must call IMG\_Init before using any SDL\_image functions that depend on specific image formats.
* Always call IMG\_Quit to clean up when you're done using SDL\_image.

### What is SDL\_GetError()?

SDL\_GetError() is a function in the SDL library that retrieves a human-readable error message describing the last error that occurred in the current thread. It's incredibly useful for debugging, as it provides insight into why an SDL function might have failed.

**Syntax**

1. const char\* SDL\_GetError(void);

**Return Value**

* Returns a string containing the error message.
* If no error has occurred, it returns an empty string ("").

**Key Points**

* **Thread-Specific**: The error message is thread-local, meaning errors in other threads won't interfere with the current thread's error state.
* **Last Error Only**: It only retrieves the most recent error. If multiple errors occur, earlier ones are overwritten.
* **Doesn't Clear Errors**: Calling SDL\_GetError() doesn't reset the error state. To clear the error, use SDL\_ClearError().

**Notes**

* Always check the return values of SDL functions to determine when to call SDL\_GetError().
* Use it alongside SDL\_SetError() if you want to set custom error messages for debugging.

### What is SDL\_Log?

SDL\_Log() is a function in the SDL library used for logging messages to the console or other output streams. It provides a simple way to debug and track the behavior of your application.

**Syntax**

1. void SDL\_Log(const char \*fmt, ...);

**Parameters**

* **fmt**: A printf()-style format string.
* **...**: Additional parameters that match the format specifiers in fmt.

**Features**

* **Thread-Safe**: You can safely call SDL\_Log() from any thread.
* **Categories and Priorities**: SDL provides logging categories (e.g., SDL\_LOG\_CATEGORY\_APPLICATION) and priorities (e.g., SDL\_LOG\_PRIORITY\_INFO) for more structured logging. You can use functions like SDL\_LogMessage() for categorized logging.

**Notes**

* By default, logs are quiet, but you can adjust the logging priority using SDL\_LogSetPriority() or SDL\_LogSetAllPriority().
* On different platforms, logs are directed to different outputs (e.g., debug output stream on Windows, log output on Android, or stderr on others).

## Resume Editing Engine.cpp

* Edit Engine::Update()

1. void Engine::Update()

2. {

3. SDL\_Log("Updating in the Game Loop...");

4. }

At this point we have our Engine::Init initializing SDL subsystems.

* Test everything by running the application

A screenshot of a computer

AI-generated content may be incorrect.

Figure 59 - Running the program with SDL Initialization

## Creating SDL\_Window and SDL\_Renderer

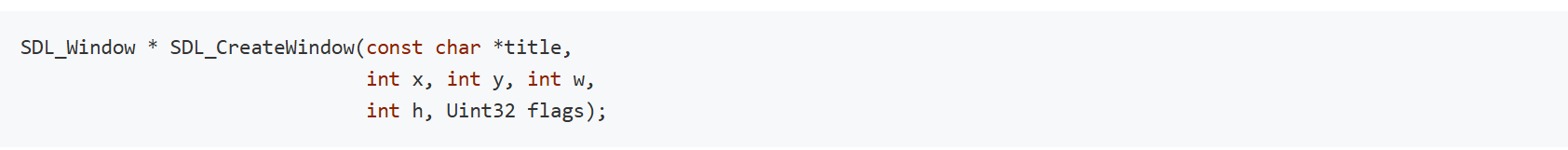
We will now add the code to create our SDL\_Window and allow ourselves to draw on it with SDL\_Renderer.

We start to use <https://wiki.libsdl.org/SDL2/FrontPage> functions.

### SDL\_CreateWindow

The SDL\_CreateWindow is used to create a window with specified position, dimensions, and flags.

**Syntax:**



**Function Parameters:**

A screenshot of a computer

AI-generated content may be incorrect.

SDL\_WINDOWPOS\_CENTERED is a predefined constant in the SDL library that specifies the position of a window to be **centered** on the screen. When you use this flag while creating an SDL window, SDL automatically calculates the coordinates needed to place the window in the center of the display. It is used to simplify window positioning, especially when you want your application to look polished by centering the window.

Note: If you have multiple monitors, SDL\_WINDOPOS\_CENTERED will center the window on the **primary display**. For centering on a specific display, you can use SDL\_WINDOWPOS\_CENTERED\_DISPLAY(n) where n is the display index.

SDL\_WINDOWPOS\_UNDEFINED is a predefined constant in the SDL library that indicates you don't care about the initial position of a window when creating it. When you use this flag, SDL will allow the operating system to decide where to place the window on the screen.

Note: If you want to specify the position later, you can use SDL\_SetWindowPosition().

The flags parameter can be one or more SDL\_WindowFlags:

1. typedef enum SDL\_WindowFlags

2. {

3. SDL\_WINDOW\_FULLSCREEN = 0x00000001, /\*\*< fullscreen window \*/

4. SDL\_WINDOW\_OPENGL = 0x00000002, /\*\*< window usable with OpenGL context \*/

5. SDL\_WINDOW\_SHOWN = 0x00000004, /\*\*< window is visible \*/

6. SDL\_WINDOW\_HIDDEN = 0x00000008, /\*\*< window is not visible \*/

7. SDL\_WINDOW\_BORDERLESS = 0x00000010, /\*\*< no window decoration \*/

8. SDL\_WINDOW\_RESIZABLE = 0x00000020, /\*\*< window can be resized \*/

9. SDL\_WINDOW\_MINIMIZED = 0x00000040, /\*\*< window is minimized \*/

10. SDL\_WINDOW\_MAXIMIZED = 0x00000080, /\*\*< window is maximized \*/

11. SDL\_WINDOW\_MOUSE\_GRABBED = 0x00000100, /\*\*< window has grabbed mouse input \*/

12. SDL\_WINDOW\_INPUT\_FOCUS = 0x00000200, /\*\*< window has input focus \*/

13. SDL\_WINDOW\_MOUSE\_FOCUS = 0x00000400, /\*\*< window has mouse focus \*/

14. SDL\_WINDOW\_FULLSCREEN\_DESKTOP = ( SDL\_WINDOW\_FULLSCREEN | 0x00001000 ),

15. SDL\_WINDOW\_FOREIGN = 0x00000800, /\*\*< window not created by SDL \*/

16. SDL\_WINDOW\_ALLOW\_HIGHDPI = 0x00002000, /\*\*< window should be created in high-DPI mode if supported.

17. On macOS NSHighResolutionCapable must be set true in the

18. application's Info.plist for this to have any effect. \*/

19. SDL\_WINDOW\_MOUSE\_CAPTURE = 0x00004000, /\*\*< window has mouse captured (unrelated to MOUSE\_GRABBED) \*/

20. SDL\_WINDOW\_ALWAYS\_ON\_TOP = 0x00008000, /\*\*< window should always be above others \*/

21. SDL\_WINDOW\_SKIP\_TASKBAR = 0x00010000, /\*\*< window should not be added to the taskbar \*/

22. SDL\_WINDOW\_UTILITY = 0x00020000, /\*\*< window should be treated as a utility window \*/

23. SDL\_WINDOW\_TOOLTIP = 0x00040000, /\*\*< window should be treated as a tooltip \*/

24. SDL\_WINDOW\_POPUP\_MENU = 0x00080000, /\*\*< window should be treated as a popup menu \*/

25. SDL\_WINDOW\_KEYBOARD\_GRABBED = 0x00100000, /\*\*< window has grabbed keyboard input \*/

26. SDL\_WINDOW\_VULKAN = 0x10000000, /\*\*< window usable for Vulkan surface \*/

27. SDL\_WINDOW\_METAL = 0x20000000, /\*\*< window usable for Metal view \*/

28.

29. SDL\_WINDOW\_INPUT\_GRABBED = SDL\_WINDOW\_MOUSE\_GRABBED /\*\*< equivalent to SDL\_WINDOW\_MOUSE\_GRABBED for compatibility \*/

30. } SDL\_WindowFlags;

31.

Setting the **flags** parameter of SDL\_CreateWindow to **0** means that no special properties or behaviors are applied to the window. The window will be created with default settings, and it won't have features like fullscreen mode, OpenGL/Vulkan support, or resizable borders.

**Default Behavior**

When the flags parameter is set to 0:

* The window is **visible** by default.
* It has **standard decorations** (title bar, borders, etc.).
* It is **not resizable**, unless explicitly specified later.

**Notes**

If you want to add specific features to the window, you can use flags like:

* **SDL\_WINDOW\_FULLSCREEN**: Makes the window fullscreen.
* **SDL\_WINDOW\_RESIZABLE**: Allows the window to be resized.
* **SDL\_WINDOW\_OPENGL**: Enables OpenGL rendering.

Note: You should use SDL\_DestroyWindow to close or return resources created from a SDL\_CreateWindow.

### SDL\_CreateRenderer

The SDL\_CreateRenderer is used to create a 2D rendering context for a window.

**Syntax:**

1. SDL\_Renderer \* SDL\_CreateRenderer(SDL\_Window \* window,

2. int index, Uint32 flags);

**Function Parameters:**

A screenshot of a computer

AI-generated content may be incorrect.

**Return Value:**

([SDL\_Renderer](https://wiki.libsdl.org/SDL2/SDL_Renderer) \*) Returns a valid rendering context or NULL if there was an error; call [SDL\_GetError](https://wiki.libsdl.org/SDL2/SDL_GetError)() for more information.

The SDL\_RendererFlags are used to create a rendering context. Here are the possible values:

1. typedef enum SDL\_RendererFlags

2. {

3. SDL\_RENDERER\_SOFTWARE = 0x00000001, /\*\*< The renderer is a software fallback \*/

4. SDL\_RENDERER\_ACCELERATED = 0x00000002, /\*\*< The renderer uses hardware

5. acceleration \*/

6. SDL\_RENDERER\_PRESENTVSYNC = 0x00000004, /\*\*< Present is synchronized

7. with the refresh rate \*/

8. SDL\_RENDERER\_TARGETTEXTURE = 0x00000008 /\*\*< The renderer supports

9. rendering to texture \*/

10. } SDL\_RendererFlags;

Note: You should return or close an SDL\_Renderer with SDL\_DestroyRenderer().

### Updating Engine.cpp

We now update Engine::Init to create our m\_Window and m\_Renderer.

1. bool Engine::Init()

2. {

3. if (SDL\_Init(SDL\_INIT\_VIDEO) != 0 && IMG\_Init(IMG\_INIT\_JPG | IMG\_INIT\_PNG) != 0) {

4. SDL\_Log("Failed to initialize SDL: %s", SDL\_GetError());

5. return false;

6. }

7.

8. // Create our SDL window

9. m\_Window = SDL\_CreateWindow("Soft Engine", SDL\_WINDOWPOS\_CENTERED, SDL\_WINDOWPOS\_CENTERED, SCREEN\_WIDTH, SCREEN\_HEIGHT, 0);

10. if (m\_Window == nullptr) {

11. SDL\_Log("Failed to create window: %s", SDL\_GetError());

12. return false;

13. }

14.

15. m\_Renderer = SDL\_CreateRenderer(m\_Window, -1, SDL\_RENDERER\_ACCELERATED | SDL\_RENDERER\_PRESENTVSYNC);

16. if (m\_Renderer == nullptr) {

17. SDL\_Log("Failed to create Renderer: %s", SDL\_GetError());

18. return false;

19. }

20.

21. return m\_IsRunning = true;

22. }

23.

### Running and Testing the Application

If you run the application now you will notice that you cannot close the SDL\_Window, this is because we did not add code to be responsive to users actions (via Events). So for now close the background window.

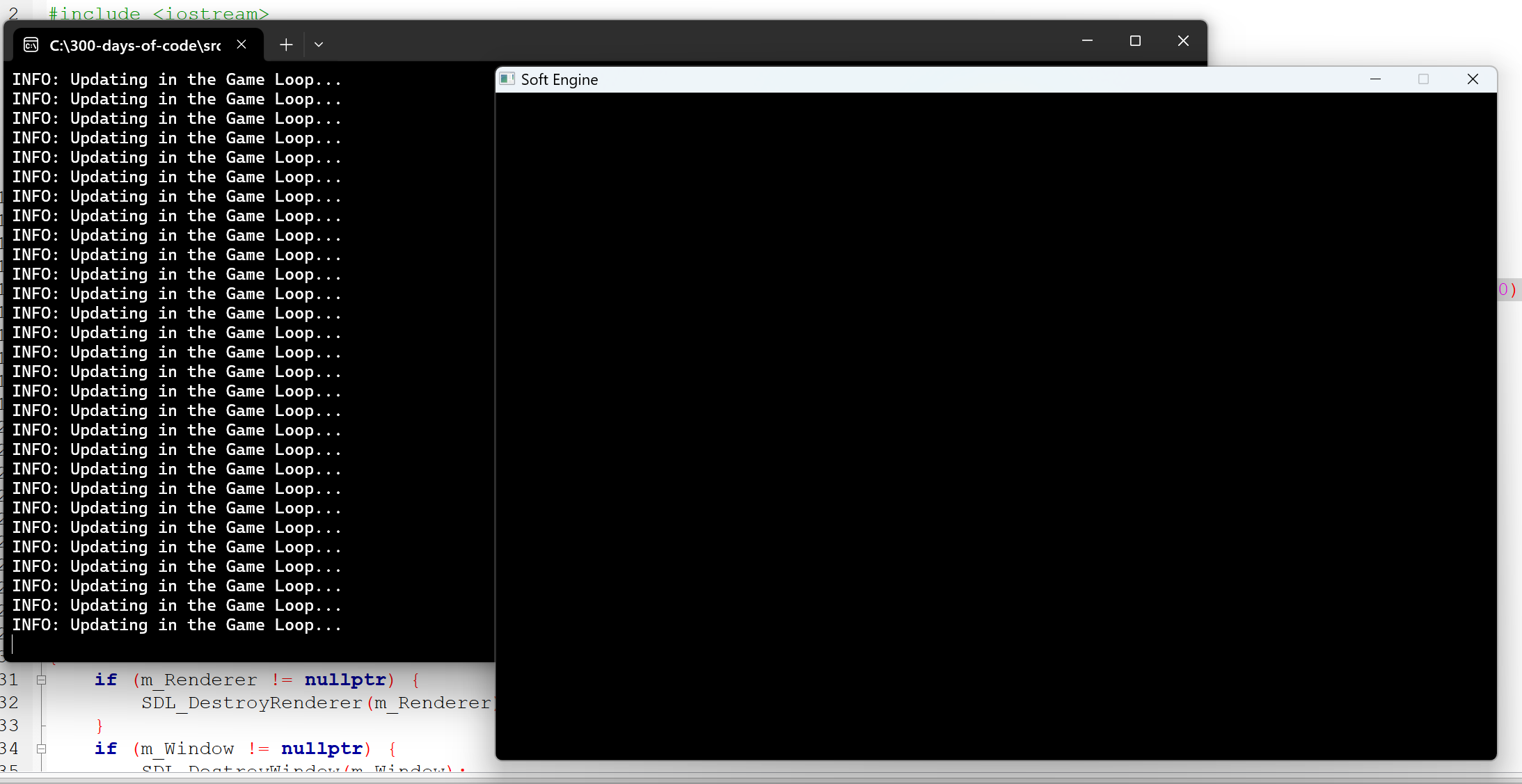


Figure 60 - Running the application at this stage of development

## Handling Events

In order to provide user control of the SDL screen we need to be able to handle events like close window, etc. SDL manages an event queue for us. The event queue holds all the events that have taken place like mouse move, keyboard press, mouse click, etc. Every time we do something on our SDL window an event is placed on a queue.

A diagram of a number

AI-generated content may be incorrect.

Figure 61 - How an event queue works (from https://circuitcellar.com/resources/event-queues/)

### What is an SDL\_Event?

The data structure SDL\_Event holds an event structure. An event can span various things like Window event (closing the window), Keyboard event (pressing a key), Mouse event (pressing a button). So SDL\_Event is a union of other structures.

Syntax:

1. typedef union SDL\_Event

2. {

3. Uint32 type; /\*\*< Event type, shared with all events \*/

4. SDL\_CommonEvent common; /\*\*< Common event data \*/

5. SDL\_DisplayEvent display; /\*\*< Display event data \*/

6. SDL\_WindowEvent window; /\*\*< Window event data \*/

7. SDL\_KeyboardEvent key; /\*\*< Keyboard event data \*/

8. SDL\_TextEditingEvent edit; /\*\*< Text editing event data \*/

9. SDL\_TextEditingExtEvent editExt; /\*\*< Extended text editing event data \*/

10. SDL\_TextInputEvent text; /\*\*< Text input event data \*/

11. SDL\_MouseMotionEvent motion; /\*\*< Mouse motion event data \*/

12. SDL\_MouseButtonEvent button; /\*\*< Mouse button event data \*/

13. SDL\_MouseWheelEvent wheel; /\*\*< Mouse wheel event data \*/

14. SDL\_JoyAxisEvent jaxis; /\*\*< Joystick axis event data \*/

15. SDL\_JoyBallEvent jball; /\*\*< Joystick ball event data \*/

16. SDL\_JoyHatEvent jhat; /\*\*< Joystick hat event data \*/

17. SDL\_JoyButtonEvent jbutton; /\*\*< Joystick button event data \*/

18. SDL\_JoyDeviceEvent jdevice; /\*\*< Joystick device change event data \*/

19. SDL\_JoyBatteryEvent jbattery; /\*\*< Joystick battery event data \*/

20. SDL\_ControllerAxisEvent caxis; /\*\*< Game Controller axis event data \*/

21. SDL\_ControllerButtonEvent cbutton; /\*\*< Game Controller button event data \*/

22. SDL\_ControllerDeviceEvent cdevice; /\*\*< Game Controller device event data \*/

23. SDL\_ControllerTouchpadEvent ctouchpad; /\*\*< Game Controller touchpad event data \*/

24. SDL\_ControllerSensorEvent csensor; /\*\*< Game Controller sensor event data \*/

25. SDL\_AudioDeviceEvent adevice; /\*\*< Audio device event data \*/

26. SDL\_SensorEvent sensor; /\*\*< Sensor event data \*/

27. SDL\_QuitEvent quit; /\*\*< Quit request event data \*/

28. SDL\_UserEvent user; /\*\*< Custom event data \*/

29. SDL\_SysWMEvent syswm; /\*\*< System dependent window event data \*/

30. SDL\_TouchFingerEvent tfinger; /\*\*< Touch finger event data \*/

31. SDL\_MultiGestureEvent mgesture; /\*\*< Gesture event data \*/

32. SDL\_DollarGestureEvent dgesture; /\*\*< Gesture event data \*/

33. SDL\_DropEvent drop; /\*\*< Drag and drop event data \*/

34.

35. /\* This is necessary for ABI compatibility between Visual C++ and GCC.

36. Visual C++ will respect the push pack pragma and use 52 bytes (size of

37. SDL\_TextEditingEvent, the largest structure for 32-bit and 64-bit

38. architectures) for this union, and GCC will use the alignment of the

39. largest datatype within the union, which is 8 bytes on 64-bit

40. architectures.

41.

42. So... we'll add padding to force the size to be 56 bytes for both.

43.

44. On architectures where pointers are 16 bytes, this needs rounding up to

45. the next multiple of 16, 64, and on architectures where pointers are

46. even larger the size of SDL\_UserEvent will dominate as being 3 pointers.

47. \*/

48. Uint8 padding[sizeof(void \*) <= 8 ? 56 : sizeof(void \*) == 16 ? 64 : 3 \* sizeof(void \*)];

49. } SDL\_Event;

50.

**Using events**

The [SDL\_Event](https://wiki.libsdl.org/SDL2/SDL_Event) structure has two uses:

* Reading events from the event queue
* Placing events on the event queue

**Reading events from the event queue**

Reading events from the event queue is done with either [SDL\_PollEvent](https://wiki.libsdl.org/SDL2/SDL_PollEvent) or [SDL\_PeepEvents](https://wiki.libsdl.org/SDL2/SDL_PeepEvents).

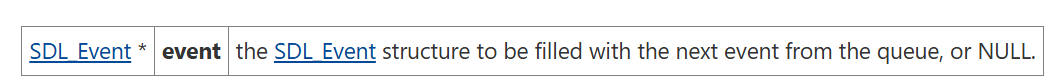
### SDL\_PollEvent

Poll for currently pending events.

**Syntax:**

1. int SDL\_PollEvent(SDL\_Event \* event);

**Function Parameters**



The [SDL\_PollEvent](https://wiki.libsdl.org/SDL2/SDL_PollEvent) function takes a pointer to an [SDL\_Event](https://wiki.libsdl.org/SDL2/SDL_Event) structure that is to be filled with event information. We know that if [SDL\_PollEvent](https://wiki.libsdl.org/SDL2/SDL_PollEvent) removes an event from the queue then the event information will be placed in our event structure, but we also know that the type of event will be placed in the **type** member of event. So to handle each event type separately we use a switch statement.

1. switch (event.type) {

We need to know what kind of events we're looking for and the event types of those events.

The SDL\_Event.type field represents the type of event in SDL's event system. Here are some of the possible values:

**Application Events**

* **SDL\_QUIT**: Indicates a user-requested quit (e.g., closing the window).
* **SDL\_APP\_TERMINATING**: The application is being terminated by the operating system.
* **SDL\_APP\_LOWMEMORY**: The application is low on memory.
* **SDL\_APP\_WILLENTERBACKGROUND**: The application is about to enter the background.
* **SDL\_APP\_DIDENTERBACKGROUND**: The application has entered the background.
* **SDL\_APP\_WILLENTERFOREGROUND**: The application is about to enter the foreground.
* **SDL\_APP\_DIDENTERFOREGROUND**: The application has entered the foreground.

**Window Events**

* **SDL\_WINDOWEVENT**: Represents various window-related events (e.g., resizing, moving, focus changes).
* **SDL\_SYSWMEVENT**: System-specific window events.

**Keyboard Events**

* **SDL\_KEYDOWN**: A key was pressed.
* **SDL\_KEYUP**: A key was released.
* **SDL\_TEXTEDITING**: Text editing (composition).
* **SDL\_TEXTINPUT**: Text input.

**Mouse Events**

* **SDL\_MOUSEMOTION**: The mouse moved.
* **SDL\_MOUSEBUTTONDOWN**: A mouse button was pressed.
* **SDL\_MOUSEBUTTONUP**: A mouse button was released.
* **SDL\_MOUSEWHEEL**: Mouse wheel motion.

**Joystick Events**

* **SDL\_JOYAXISMOTION**: Joystick axis motion.
* **SDL\_JOYBALLMOTION**: Joystick trackball motion.
* **SDL\_JOYHATMOTION**: Joystick hat position change.
* **SDL\_JOYBUTTONDOWN**: Joystick button pressed.
* **SDL\_JOYBUTTONUP**: Joystick button released.
* **SDL\_JOYDEVICEADDED**: A joystick was connected.
* **SDL\_JOYDEVICEREMOVED**: A joystick was disconnected.

**Controller Events**

* **SDL\_CONTROLLERAXISMOTION**: Game controller axis motion.
* **SDL\_CONTROLLERBUTTONDOWN**: Game controller button pressed.
* **SDL\_CONTROLLERBUTTONUP**: Game controller button released.
* **SDL\_CONTROLLERDEVICEADDED**: A game controller was connected.
* **SDL\_CONTROLLERDEVICEREMOVED**: A game controller was disconnected.

**Other Events**

* **SDL\_AUDIODEVICEADDED**: An audio device was added.
* **SDL\_AUDIODEVICEREMOVED**: An audio device was removed.
* **SDL\_SENSORUPDATE**: Sensor data update.
* **SDL\_DROPFILE**: A file was dropped onto the window.

This is just a subset of the possible event types. You can find a comprehensive list in the SDL documentation [here](https://wiki.libsdl.org/SDL3/SDL_EventType). Let me know if you'd like help handling specific events!

Each event type is associated with a different union member that holds more details about the event.

### SDL\_Quit

The **SDL\_Quit** event is triggered when the user requests to terminate the application. This typically happens when the user closes the window or selects a quit option from the operating system. The event is part of SDL's event system and is represented by the SDL\_QUIT constant.

Key Details:

* **Event Type**: SDL\_QUIT
* **Purpose**: Indicates that the application should shut down gracefully.
* **Usage**: You can handle this event in your main event loop to perform cleanup tasks before exiting.

## Update to Engine.cpp

In this section we add code to Engine.cpp to allow the user to “quit” or “close” the window application.

### Add code the Engine::Events()

1. void Engine::Events()

2. {

3. SDL\_Event event;

4. SDL\_PollEvent(&event);

5. switch(event.type) {

6. case SDL\_QUIT:

7. Quit();

8. break;

9. }

10. }

11.

On line 3 we create an SDL\_Event and then obtain the next item in the event queue by invoking SDL\_PollEvent on line 4. The details of the next event is stored in the event variable.

The switch statement on 5 examines a value all events have type. If the event.type is SDL\_QUIT we invoke an internal class method called Quit().

### Add code the Engine::Quit()

If we examine our game loop in Main.cpp:

1. while (Engine::GetInstance()->isRunning()) {

2. // Get all current events (e.g. mouse clicks, etc.)

3. Engine::GetInstance()->Events();

4.

5. // Update all objects/entities

6. Engine::GetInstance()->Update();

7.

8. // Render/update the game graphics

9. Engine::GetInstance()->Render();

10. }

The way to exit the application is to have Engine::GetInstance()->isRunning() to return false.

1. inline bool isRunning() {

2. return m\_IsRunning;

3. }

The value returned by isRunning() is just the current value of m\_IsRunning. So, if the user closes the application we should just set the m\_IsRunning class member variable to false. This should occur in Quit().

1. void Engine::Quit()

2. {

3. m\_IsRunning = false;

4. }

Let’s test and run the program again. You will see that you can move the window around (wondering how since we did not add an event handler for that?) and when we close the window the SDL window closes.

1. I use 7-zip to manage my zip files [↑](#footnote-ref-1)