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 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.

# 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 application.

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! |

## 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:

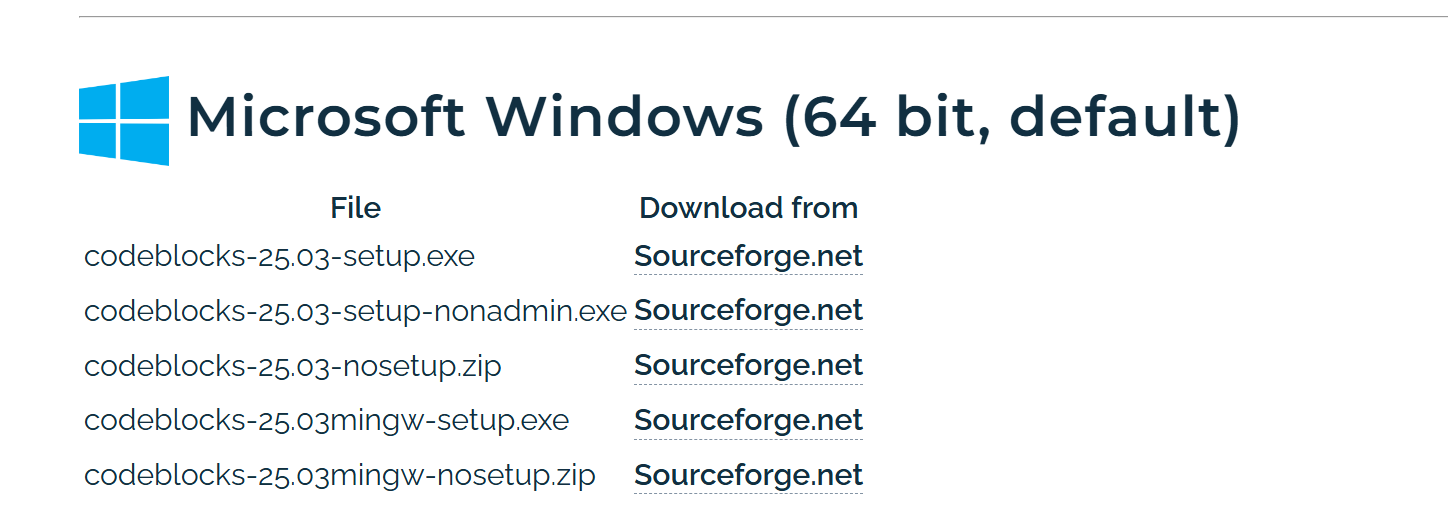


Figure - Selecting the right setup.exe to download

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 - Downloading file from Sourceforge

I then run the setup.exe file:



Figure - Locating and executing the setup.exe file

### Running the Setup

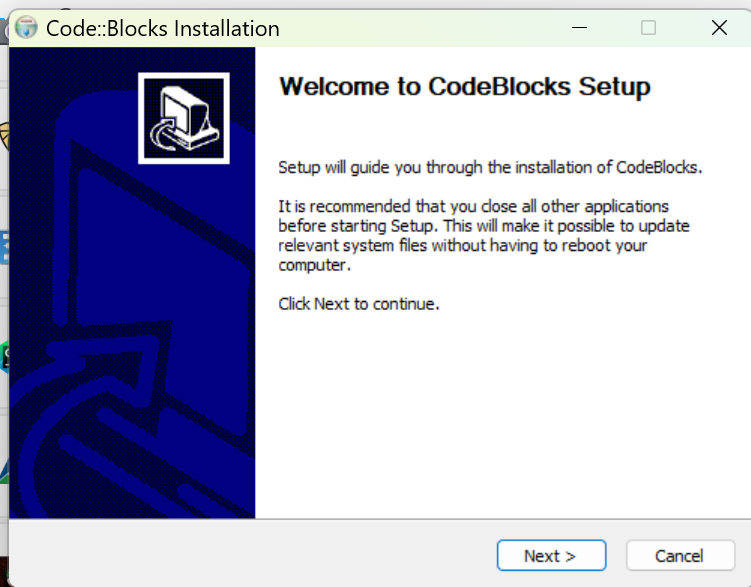


Figure - Initial Code::Blocks Installation dialog

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

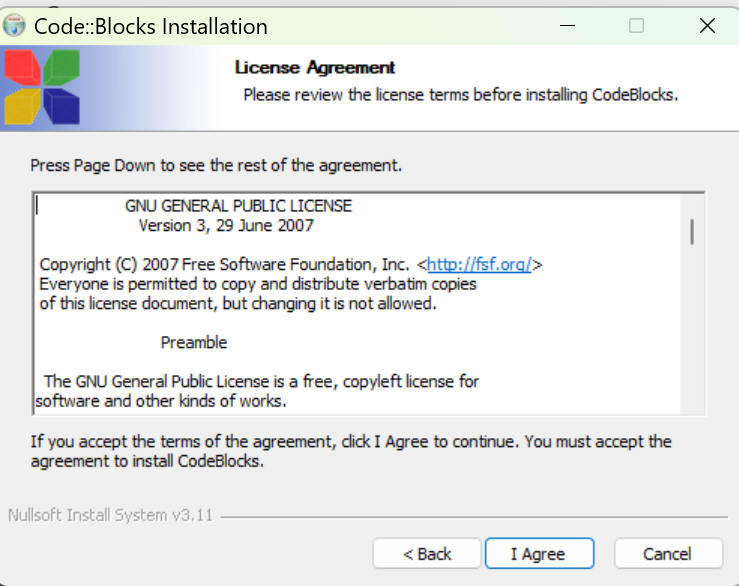


Figure - The License Agreement screen

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”.

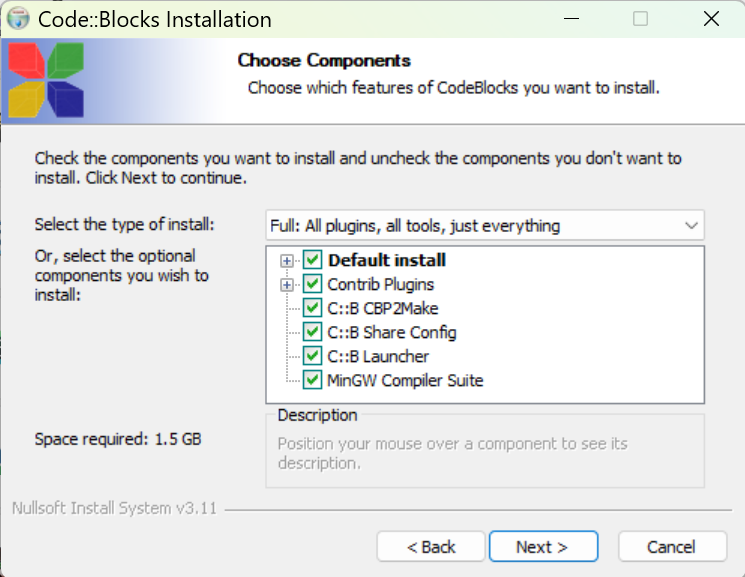


Figure - The "Choose Components" screen

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.

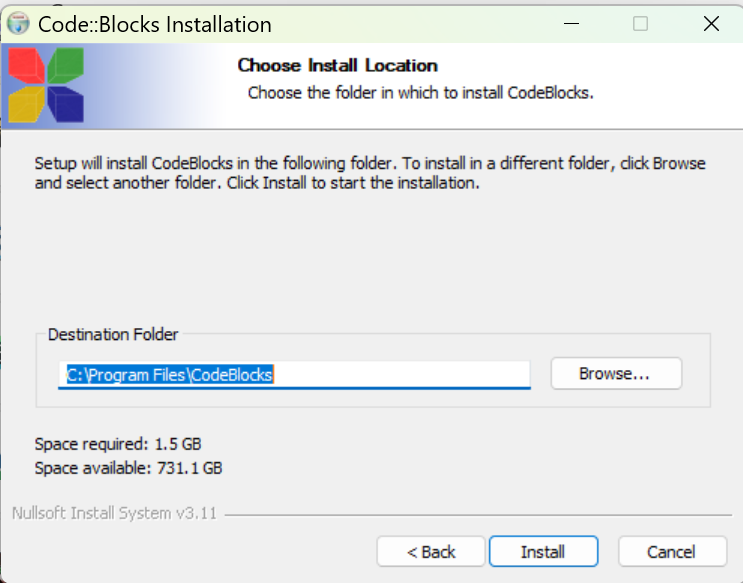


Figure - Installation location on your PC

I usually take the default location.

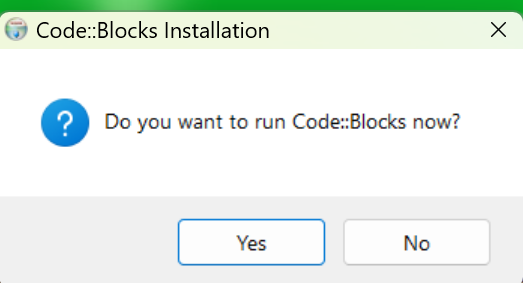


Figure - Prompt to run Code::Blocks now?

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

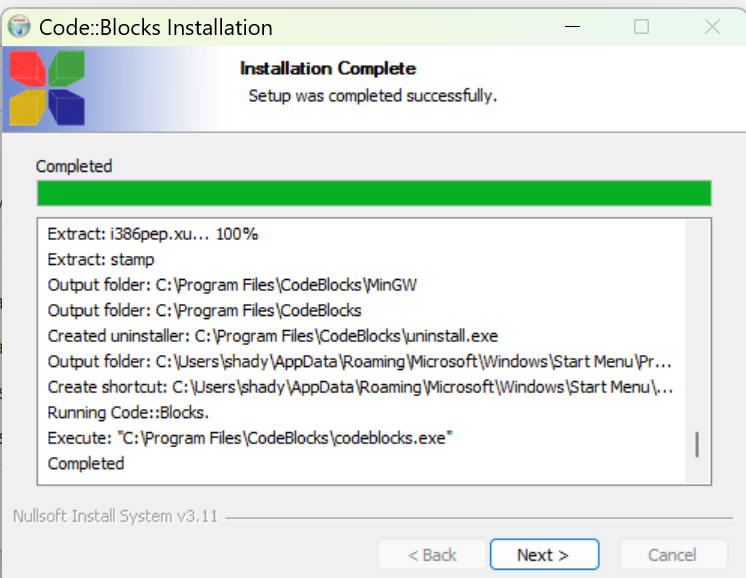


Figure - Code::Blocks installation completed dialog

I clicked on “Next >”.

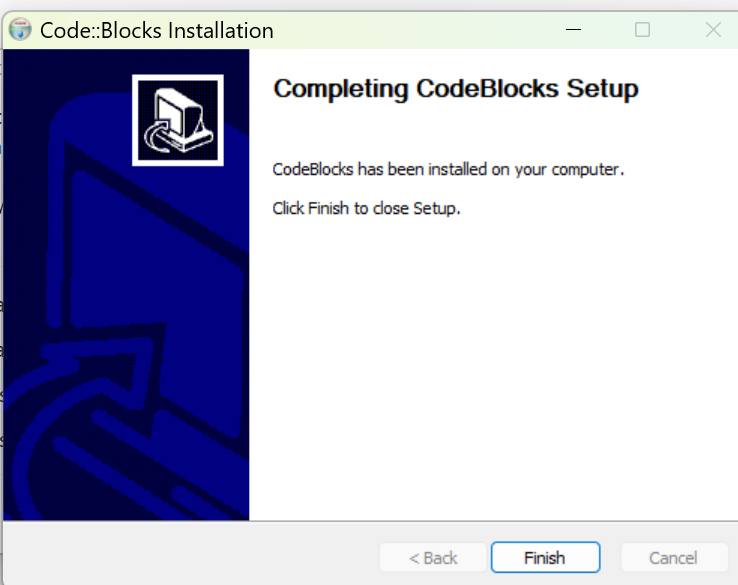


Figure - The final Code::Blocks setup screen

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

### Starting up Code::Blocks

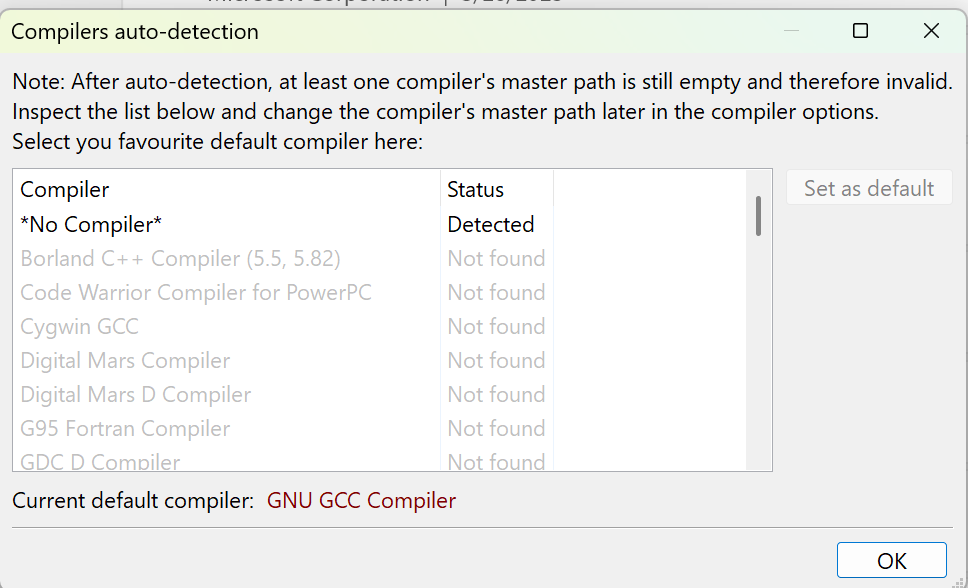


Figure - Code::Blocks lists all the compilers it detected

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.

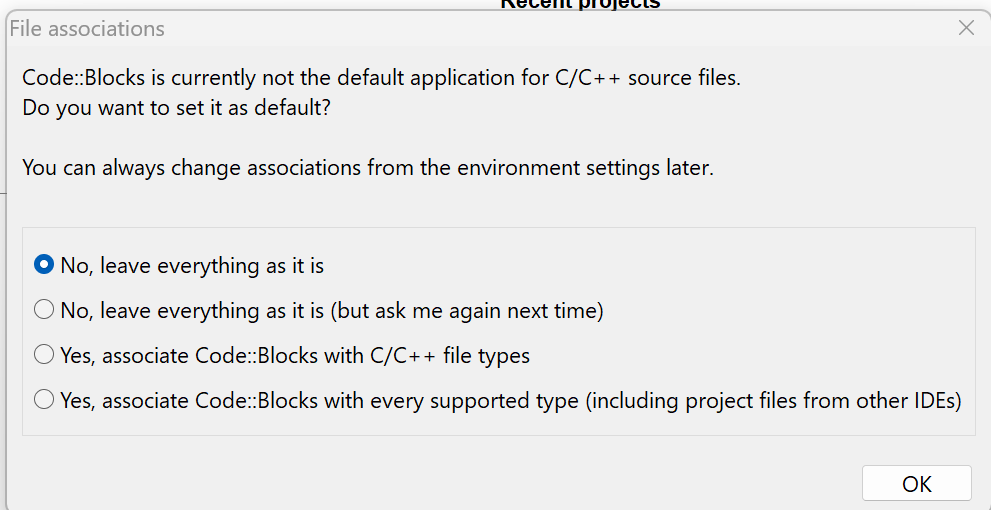


Figure - Option to update file association

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

## Starting a Project with Code::Blocks

A screenshot of a computer

AI-generated content may be incorrect.

Figure - 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

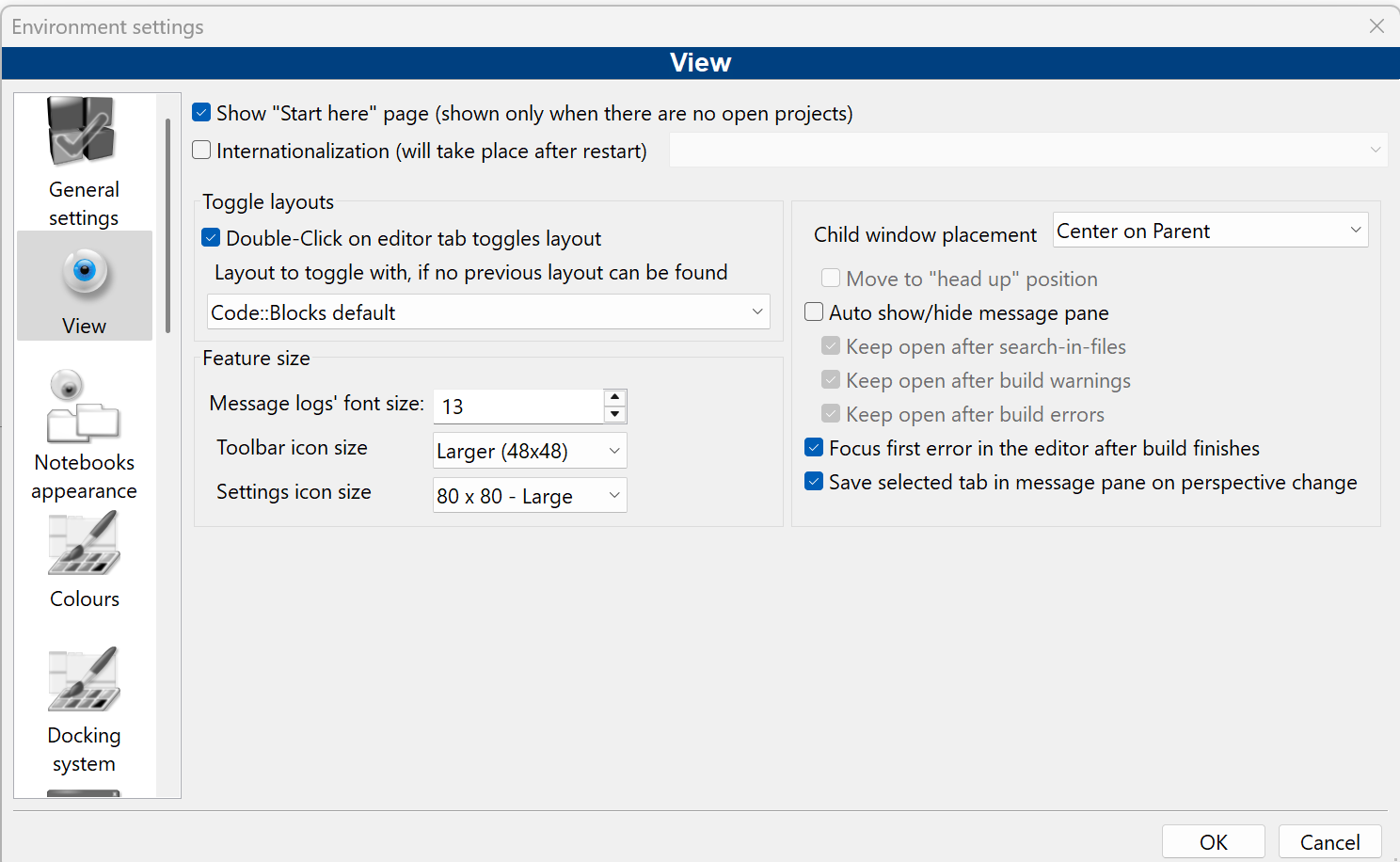
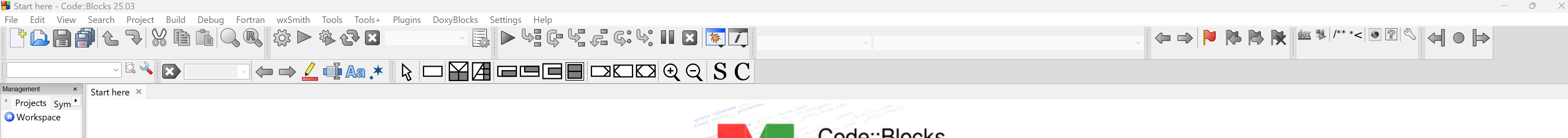


Figure - Increasing the toolbar icon sizes

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



I also increased the font size used by:

* Select Settings 🡺 Editor

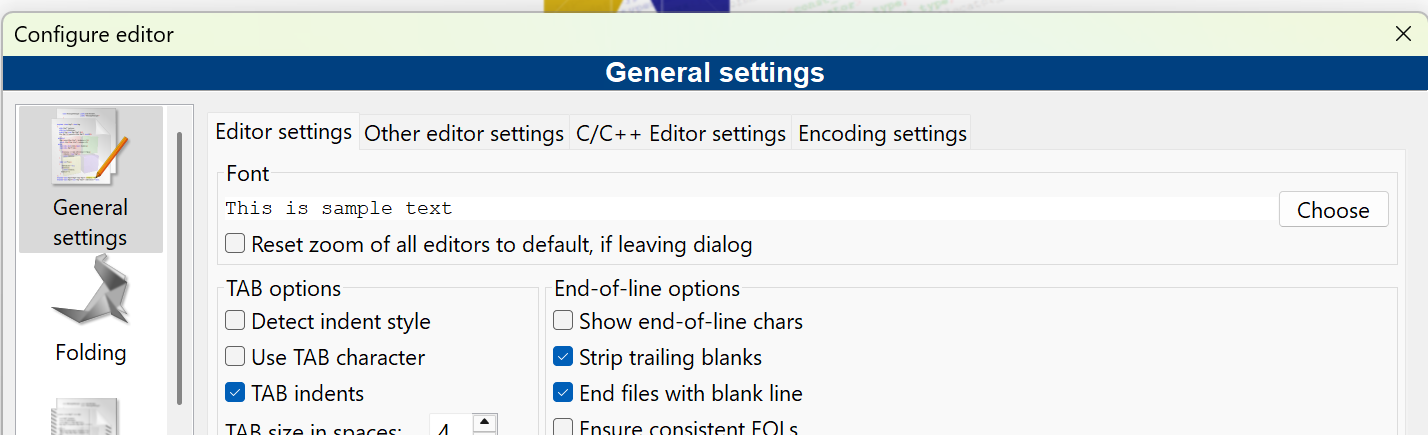


Figure - Updating font-size

* Click on “Choose” button

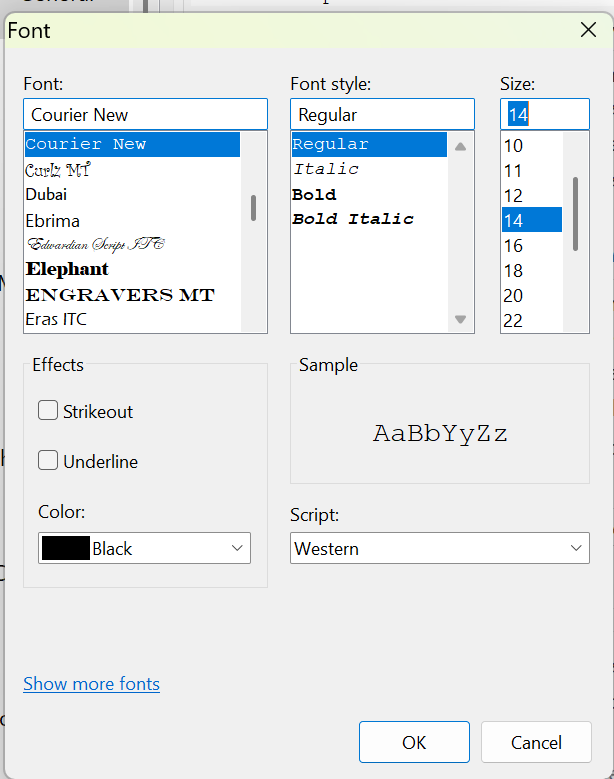


Figure - Select desired font size

* I selected 14 and clicked on “OK”

### Creating the Project

* Click on “Create a new project” link



Figure - Creating a new Project

* Select “Console Application”

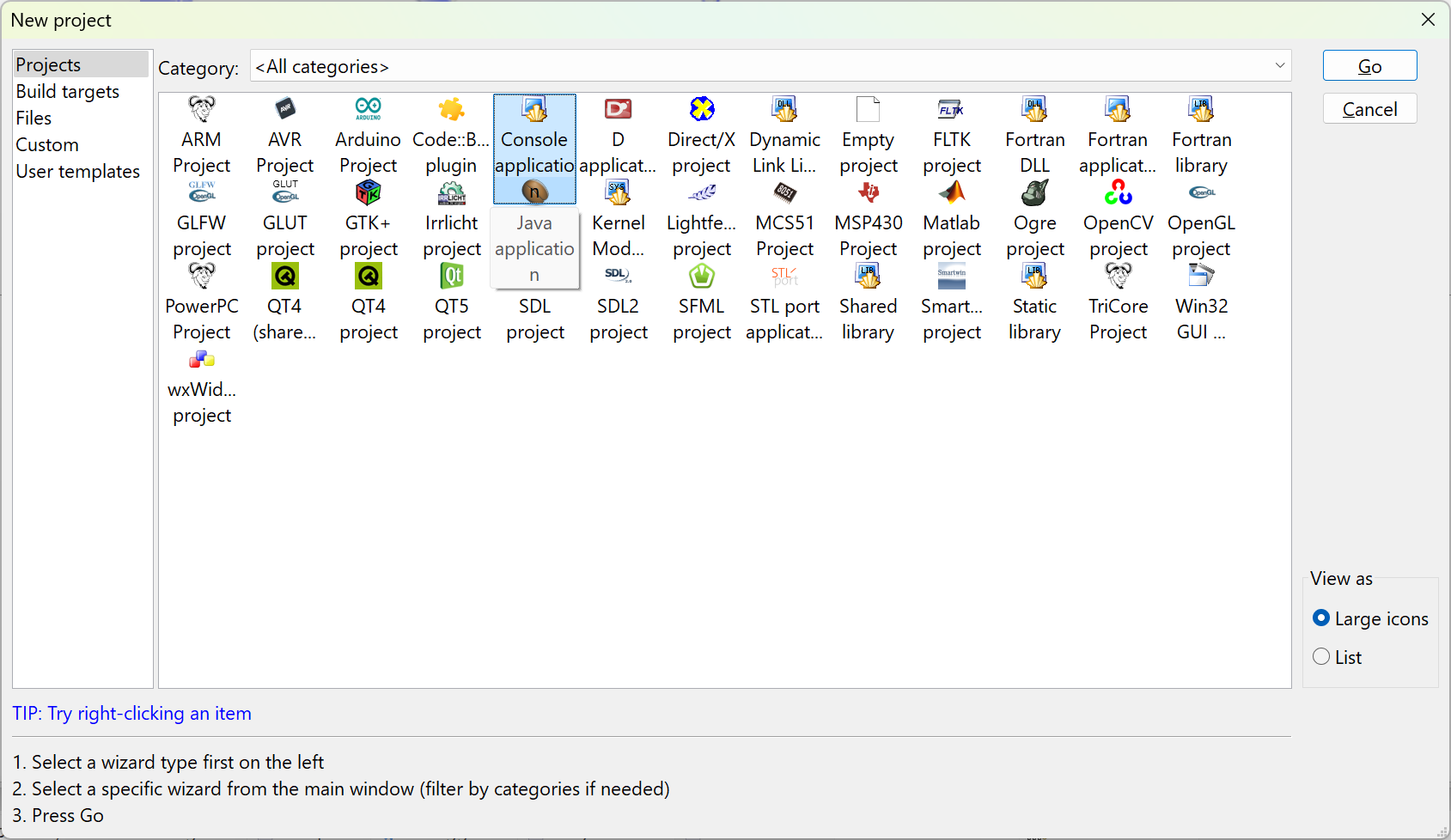


Figure - Creating our first console application

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

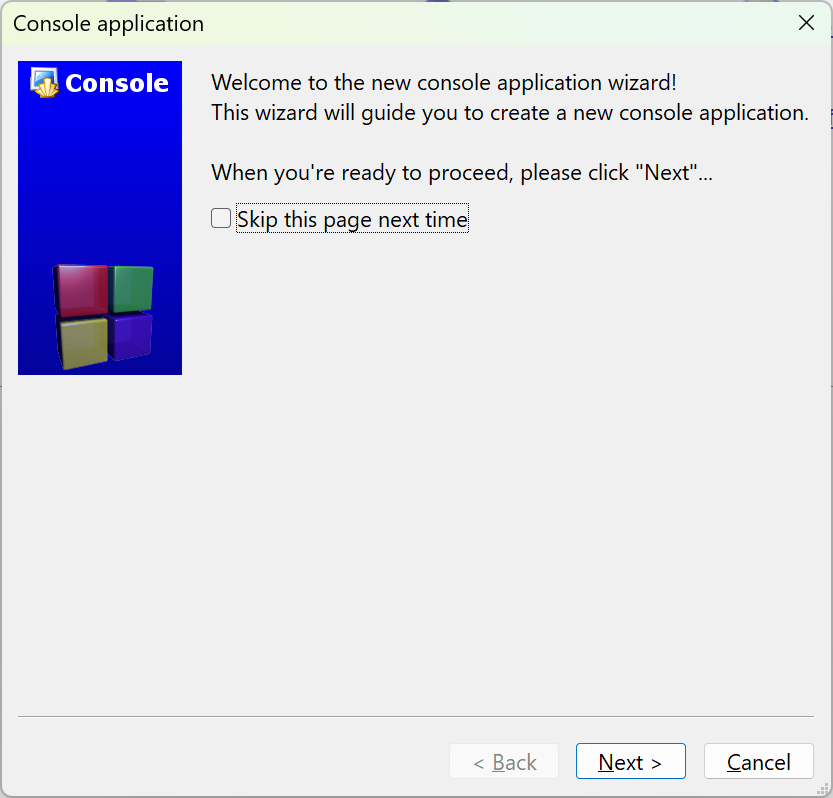
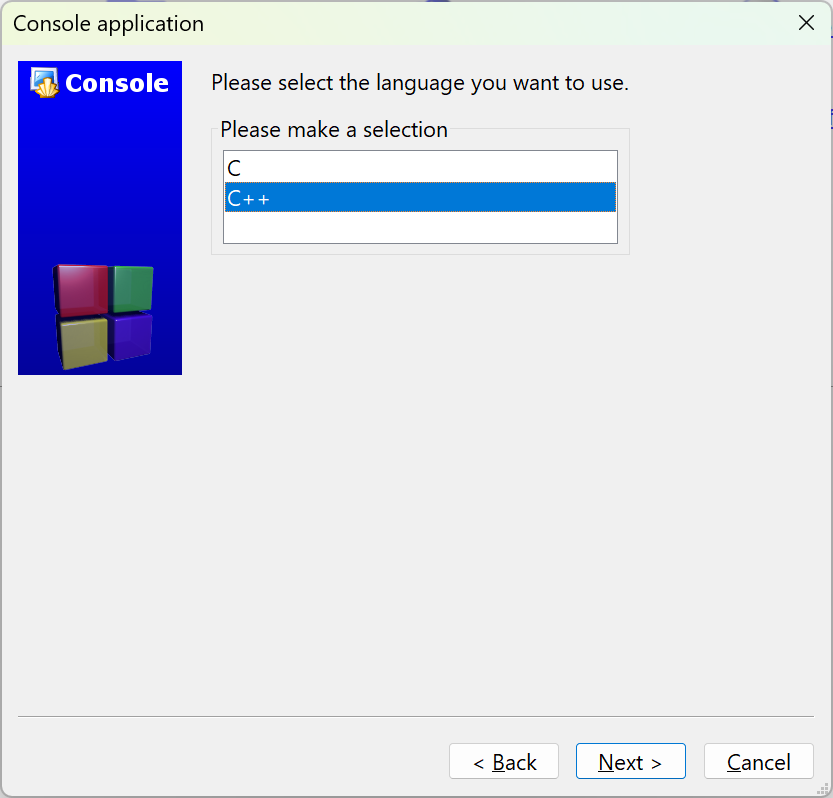


Figure - The Console application wizard starting...

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



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

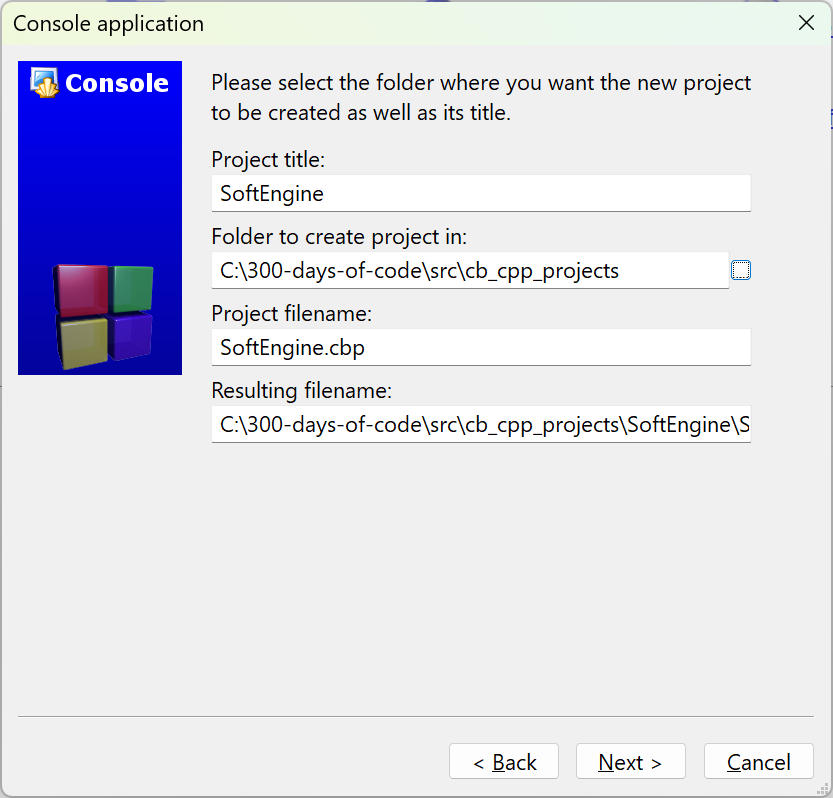


Figure - Project information

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 >”

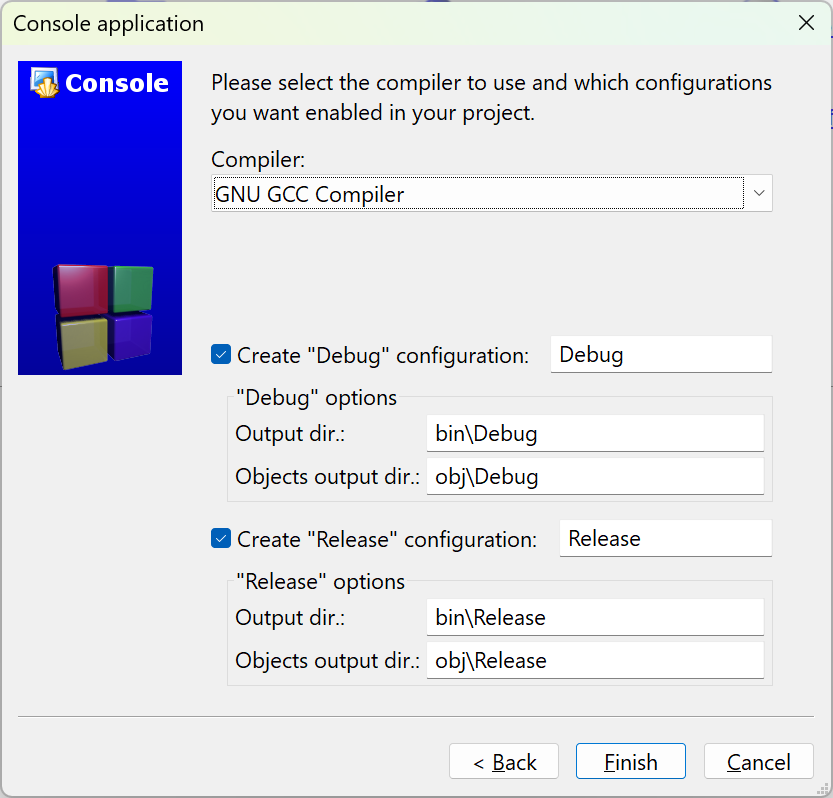


Figure - Final step in creating our game engine project

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

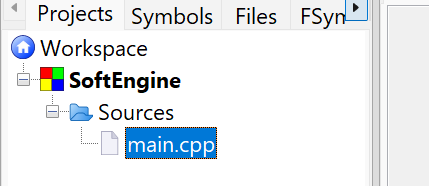


Figure - Default main.cpp is created

* 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:

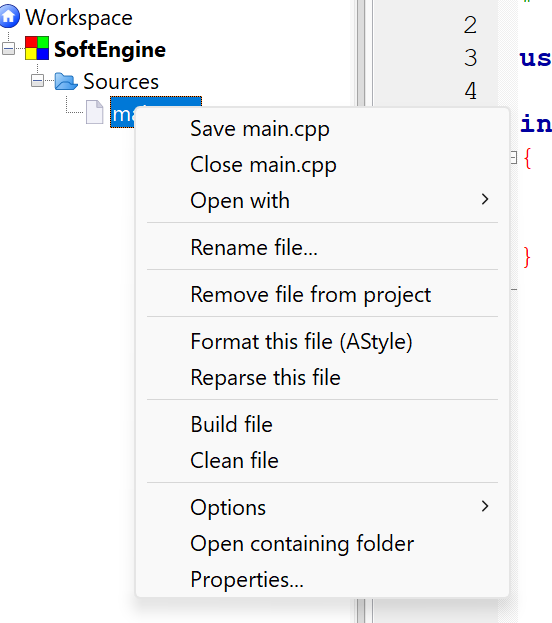


Figure - Context menu for file

* Enter the name Main.cpp:

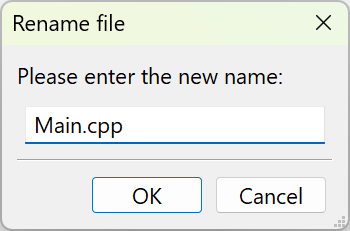


Figure - Rename dialog box

* Click “OK”

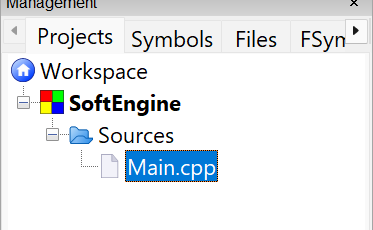


Figure - Update name in Workspace

### 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 - 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 - 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 - 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 - 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 - Unzipping the SDL2 version

You will see:

A screenshot of a computer

AI-generated content may be incorrect.

Figure - 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 - 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 - 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 - 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 - Opening an existing project

* Go to Settings 🡺 Compiler…

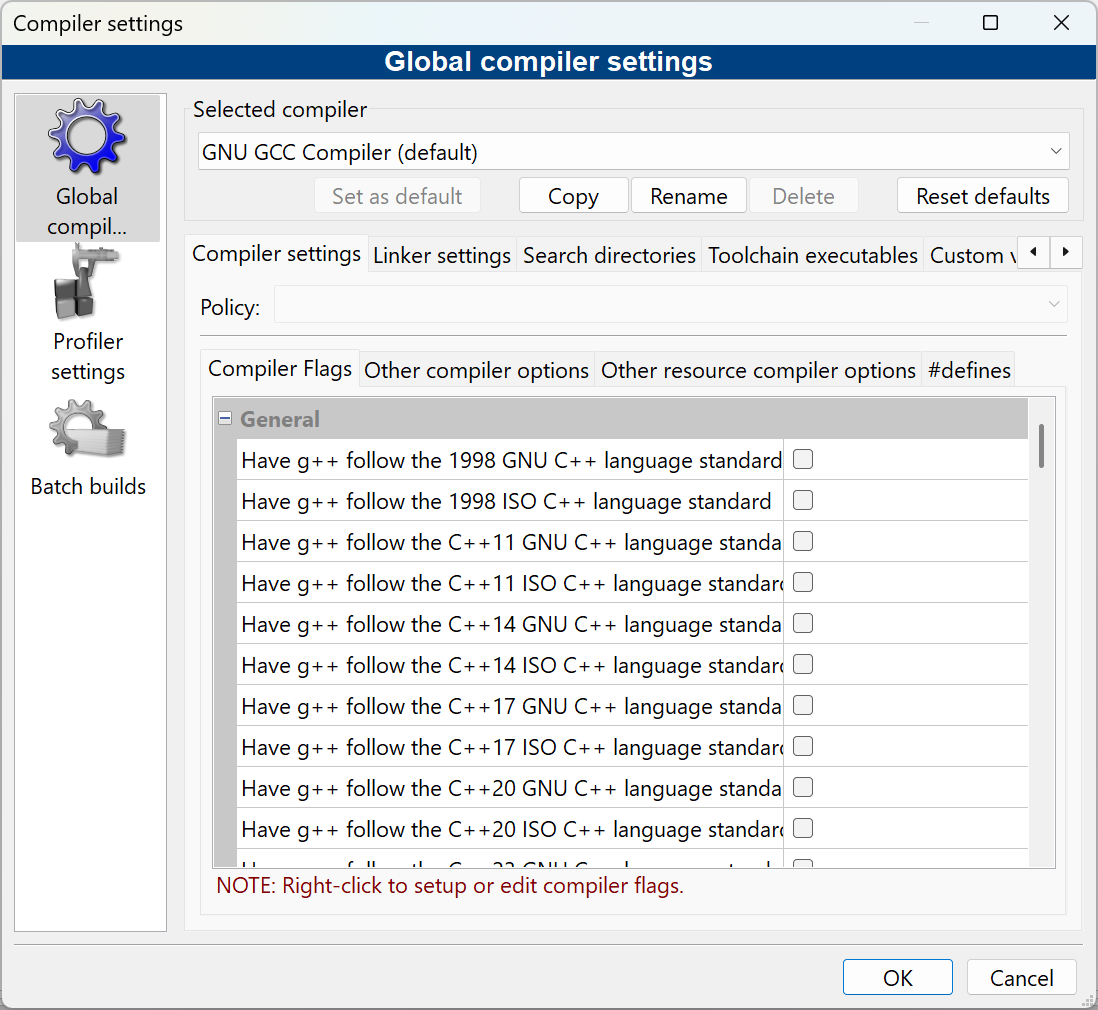


Figure - Global compiler settings

* Click on the “Linker settings” tab

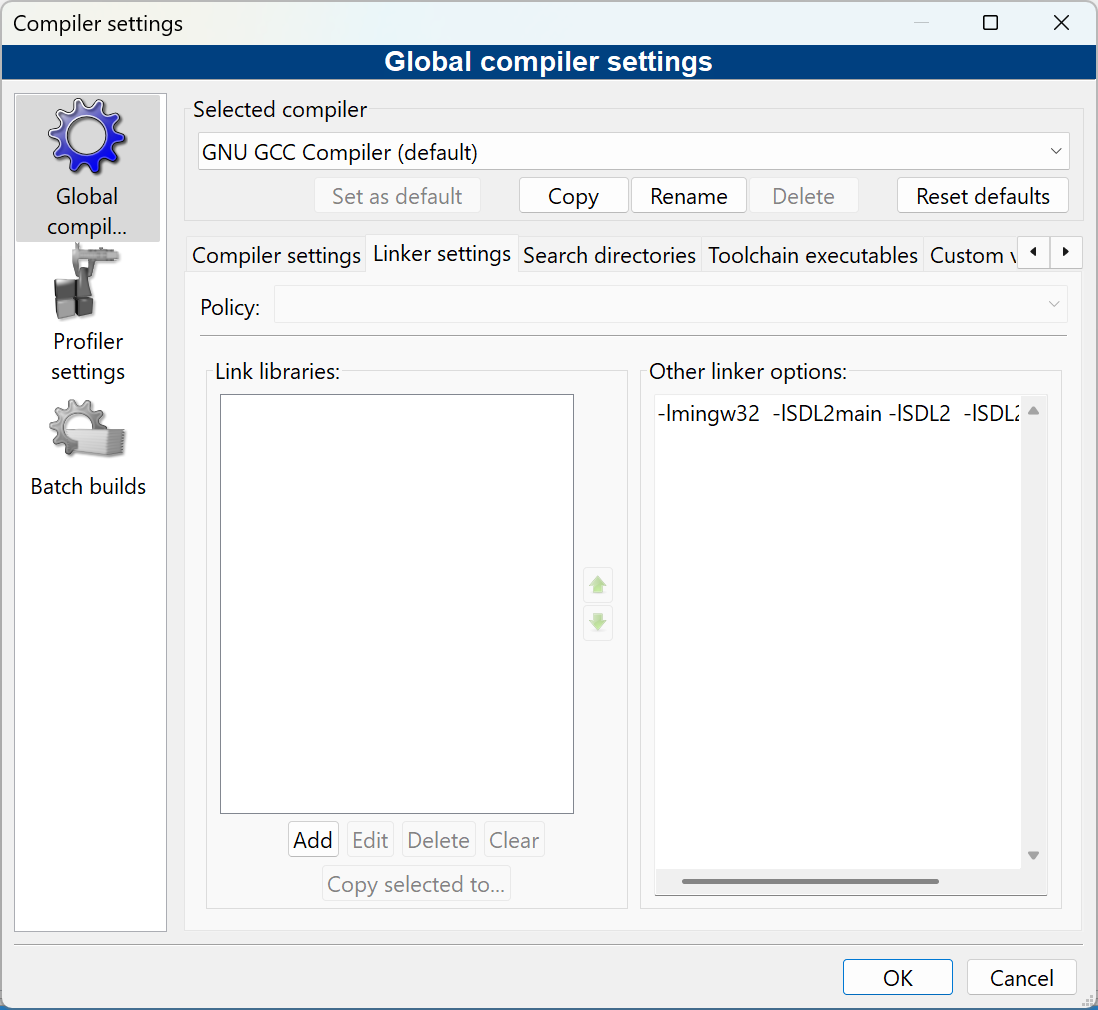
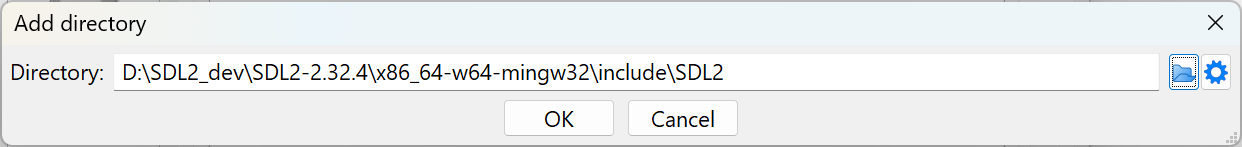


Figure - Adding linker options

* 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



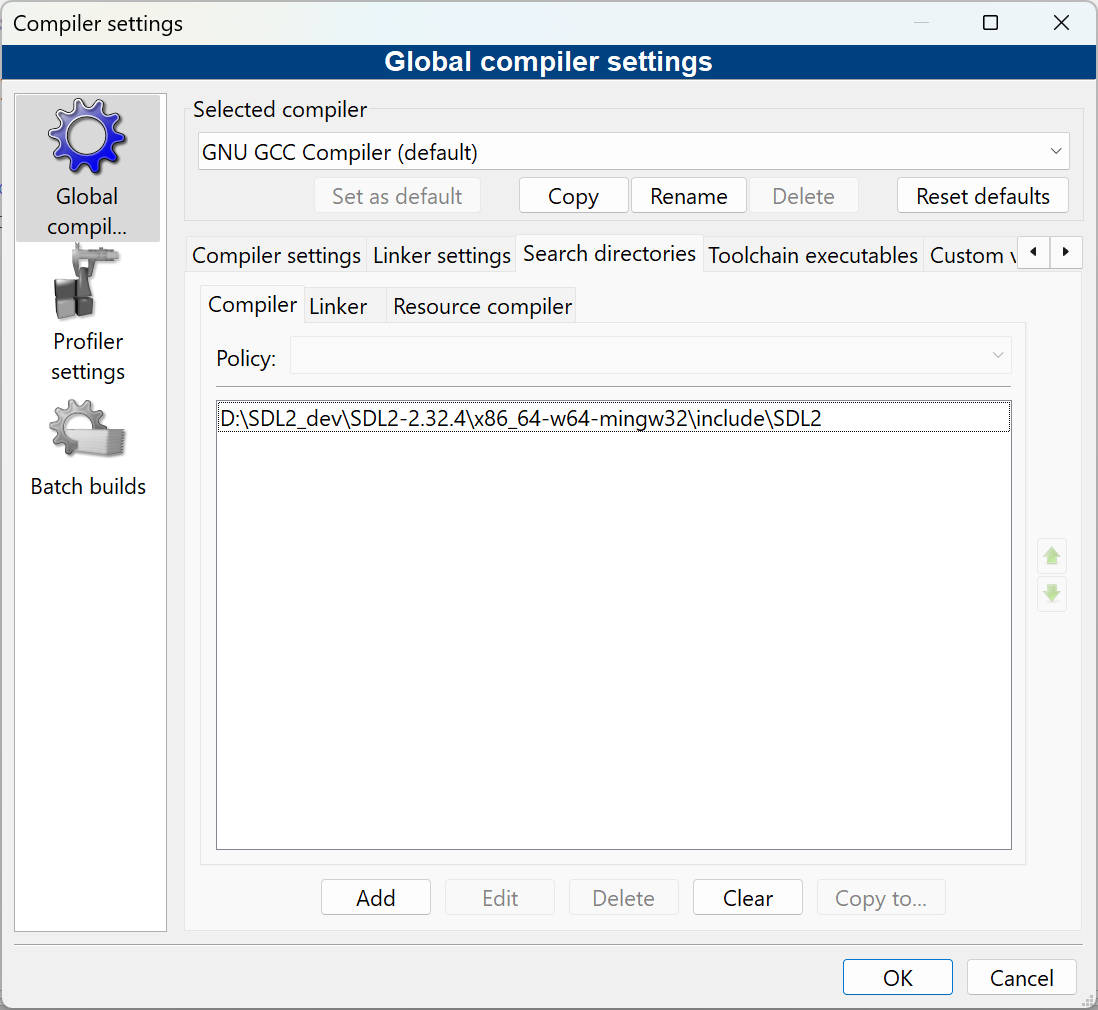


Figure - Adding SDL3 include folder for the compiler

* Add the SDL2\_image include as well

A screenshot of a computer

AI-generated content may be incorrect.

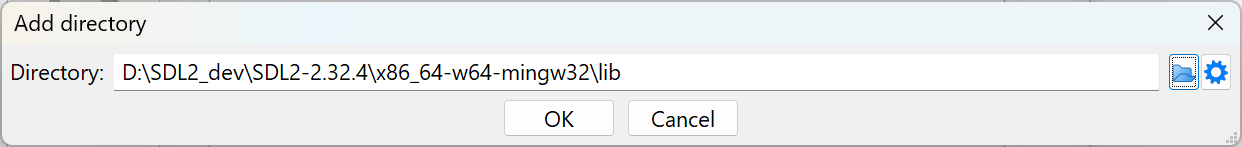
* Select the “Linker” sub-tab and add the location of the lib folder:

A screenshot of a search box

AI-generated content may be incorrect.

Figure - 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 - 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. {

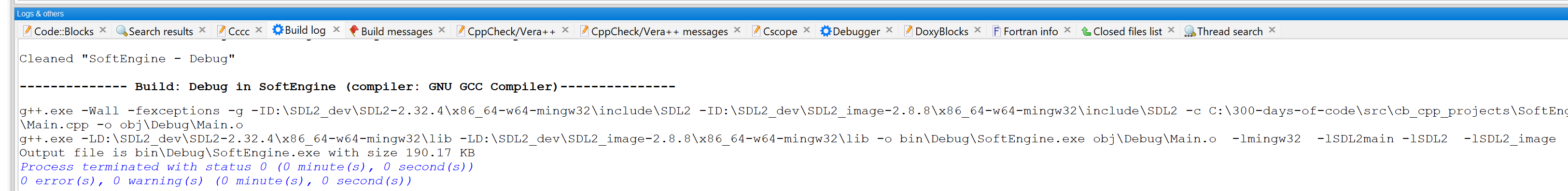
8. cout << "Hello world!" << endl;

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 - 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 - 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 - Program with SDL component running ran successfully

# 2. Game Loop for SDL Game

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 .

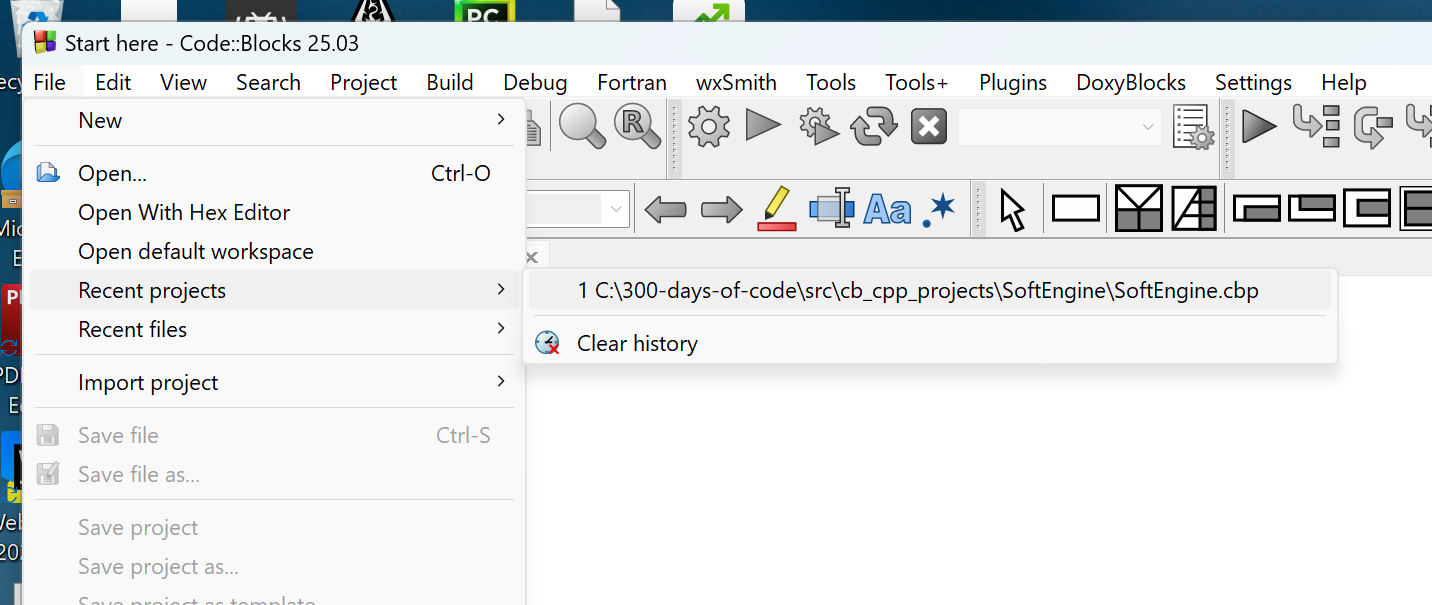


Figure - Selecting a recent project

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



Figure - 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.

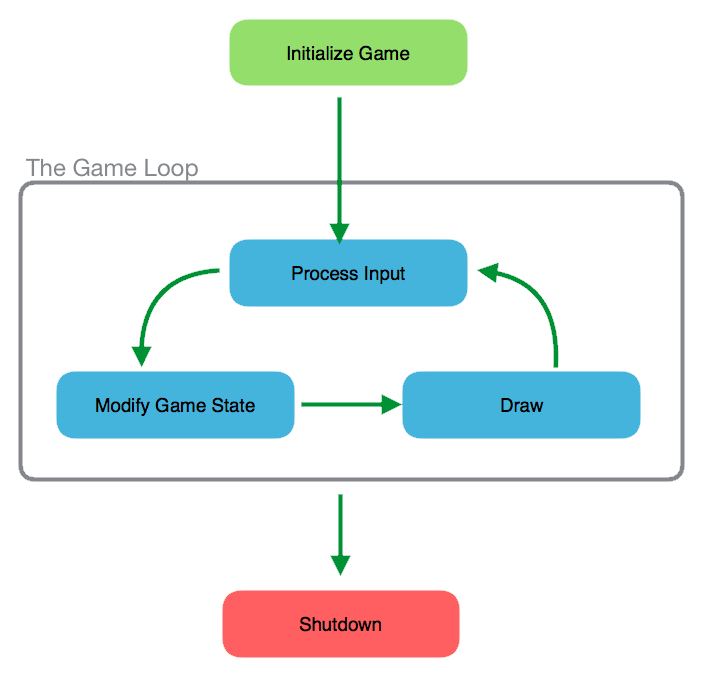


Figure - A diagram of the "Game Loop"

## 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 - Adding a new class to the project

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

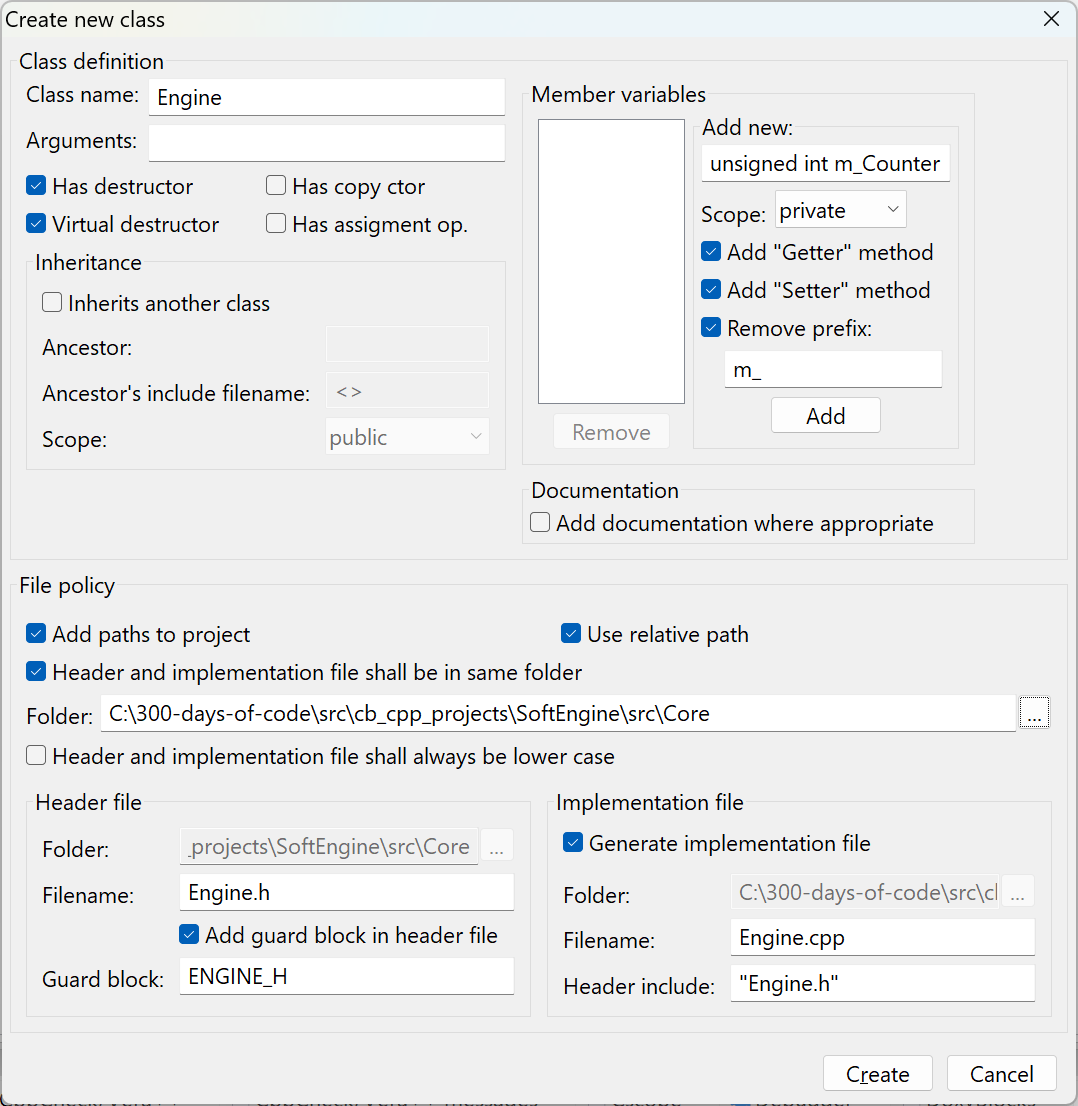


Figure - Creating our Engine.cpp and Engine.h files

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 - Add to project prompt?

* Click “Yes” to the above prompt.

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Add to Debug and Release prompt

* Click “OK” to the above prompt.

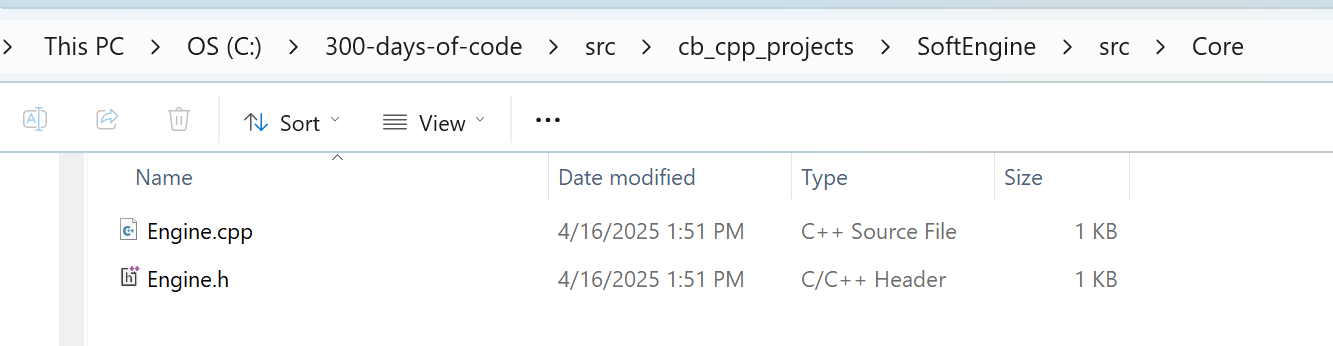
The project will appear as:

A screenshot of a computer

AI-generated content may be incorrect.

Figure - 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 - 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.

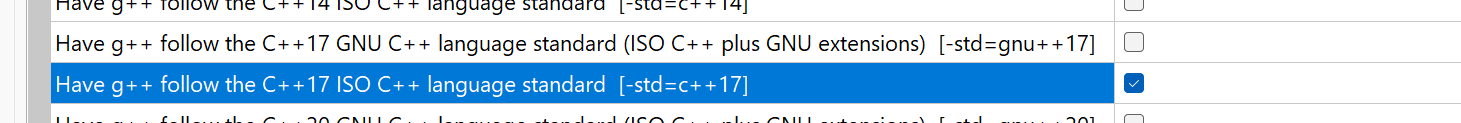
* Click on Settings 🡺 Compiler…

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 - Adding missing class methods

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

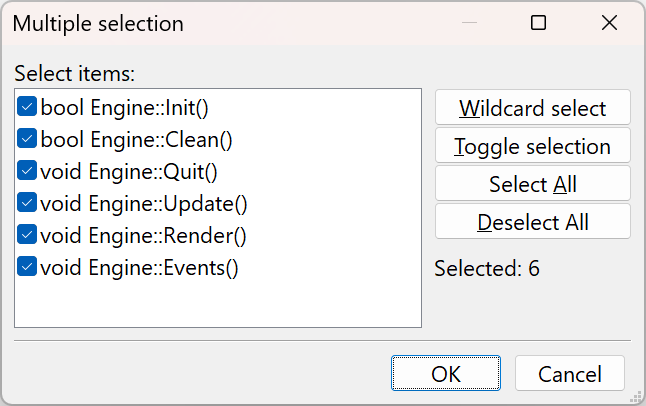


Figure - Selecting the class methods to insert

* 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.

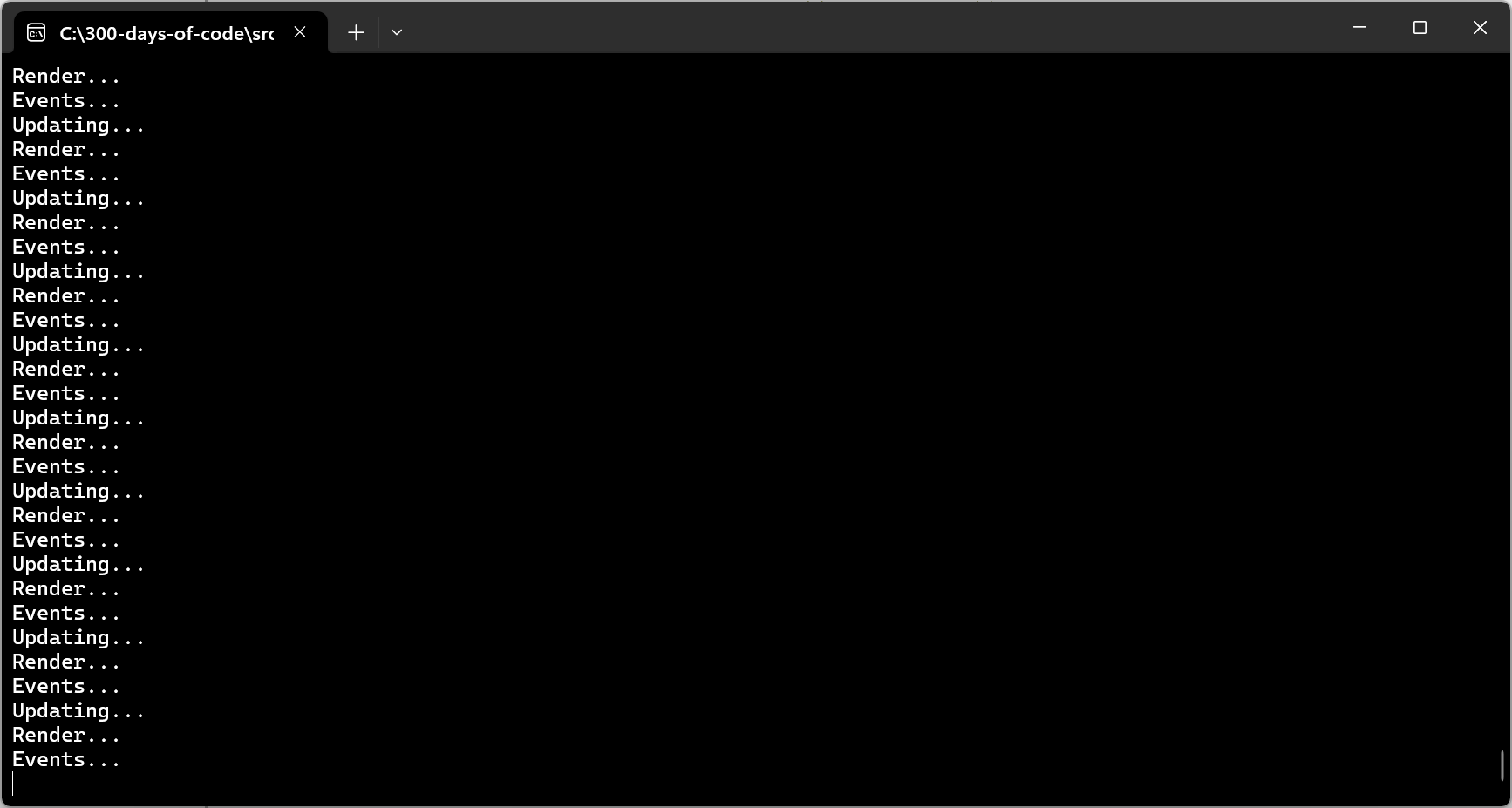


Figure - Running our game loop

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().

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