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.

In addition, we will use the latest version of all software and libraries and note the changes.

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

The first video walks you through obtaining and installing Code::Blocks. In addition, it creates the initial project for this series, which of course starts as a “Hello, World” project. Lastly, it sets us up to use the SDL libraries.

## 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.

## 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 tutorial. 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. I am surprised that the project has not moved to GitHub.

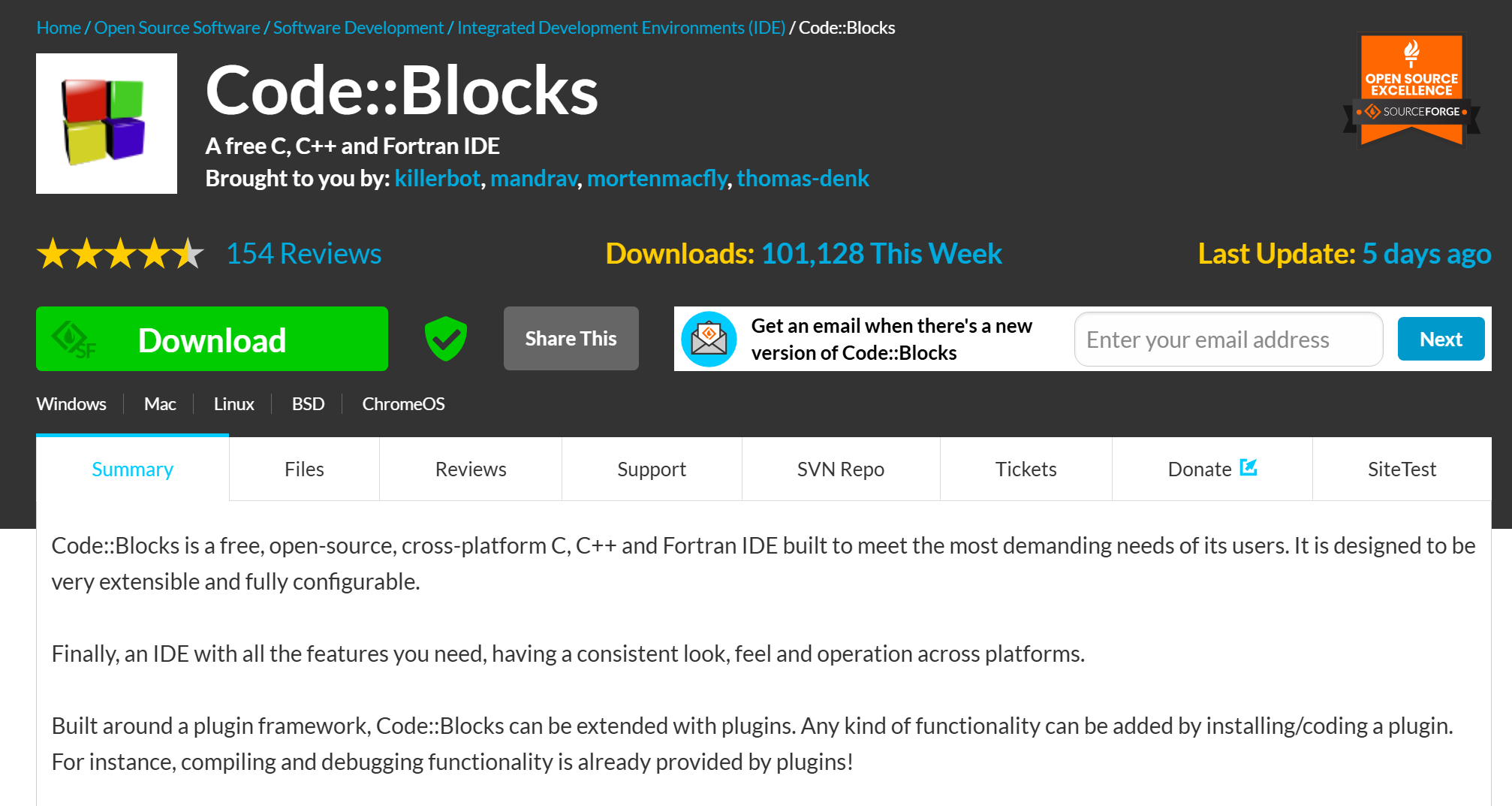


Figure 2 - Downloading file from Sourceforge

I then run the setup.exe file:



Figure 3 - Locating and executing the setup.exe file

### Running the Setup

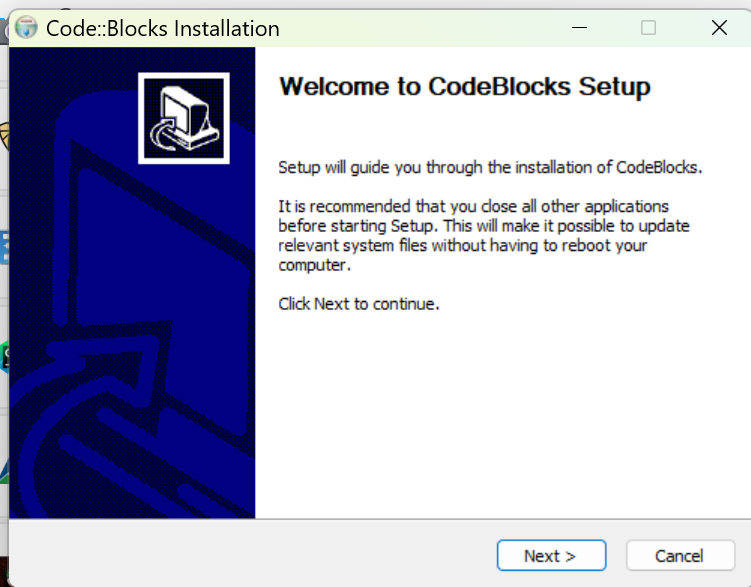


Figure 4 - Initial Code::Blocks Installation dialog

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

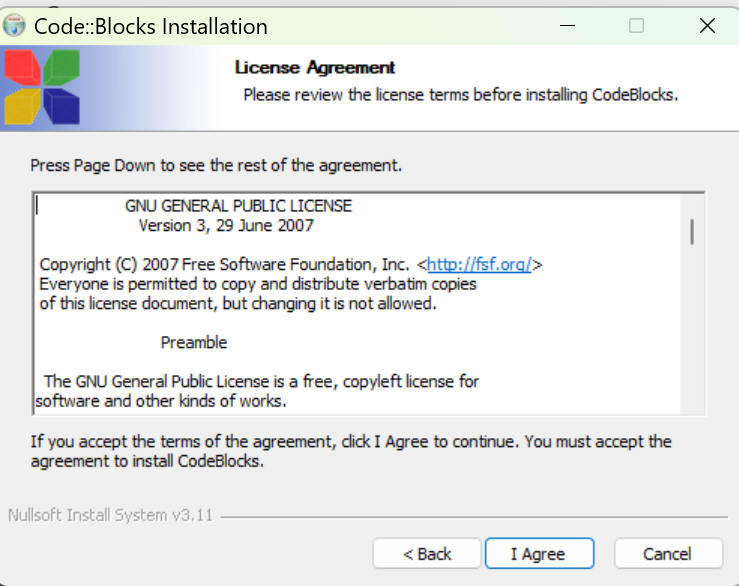


Figure 5 - The License Agreement screen

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”.

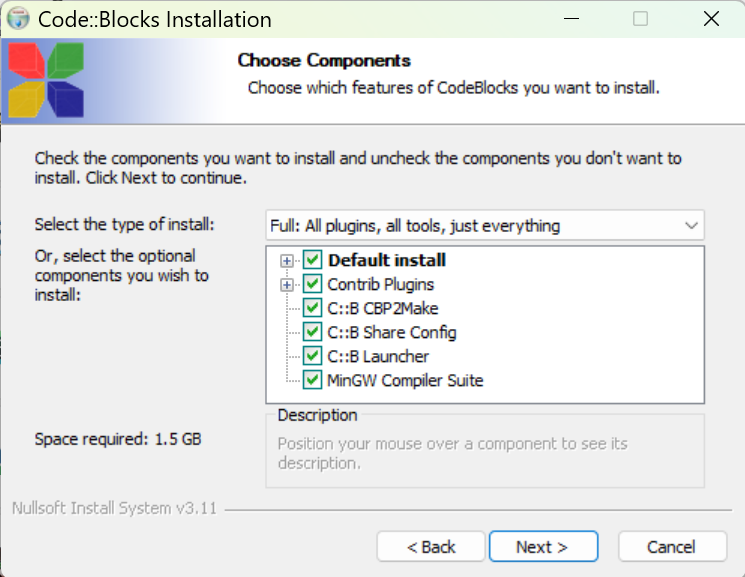


Figure 6 - The "Choose Components" screen

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.

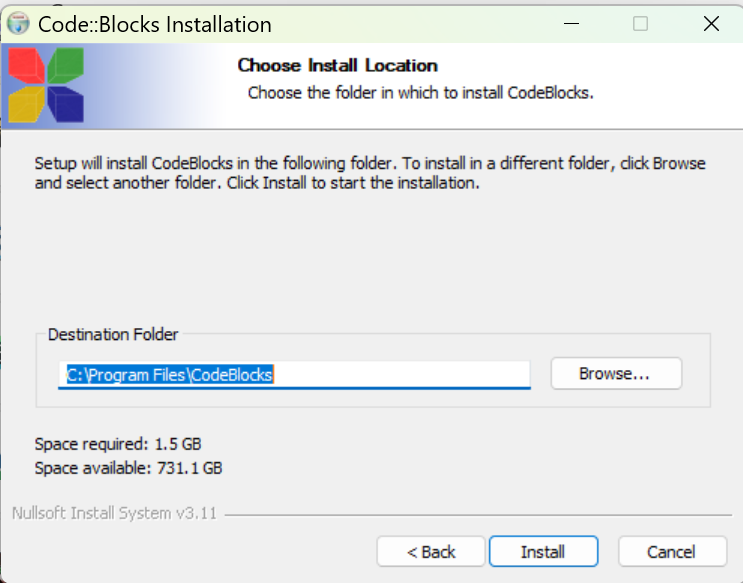


Figure 7 - Installation location on your PC

I usually take the default location.

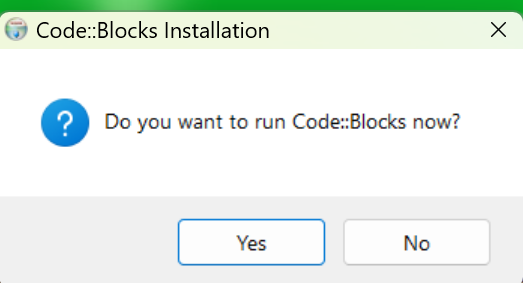


Figure 8 - Prompt to run Code::Blocks now?

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

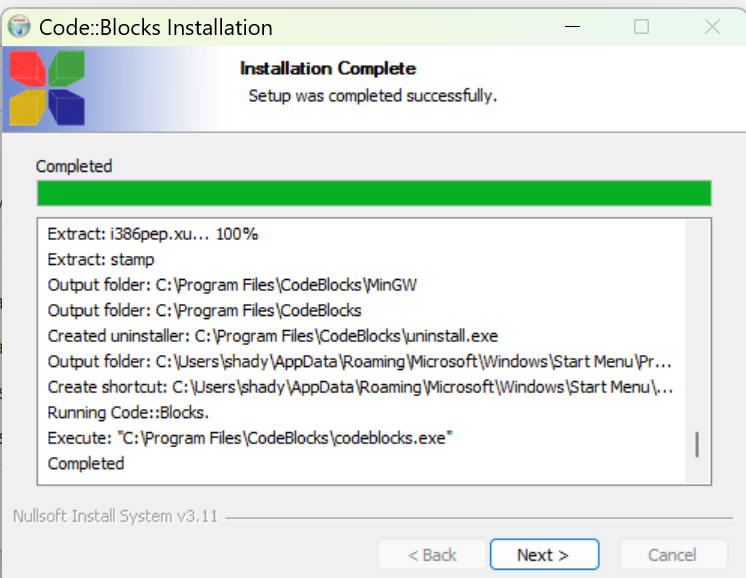


Figure 9 - Code::Blocks installation completed dialog

I clicked on “Next >”.

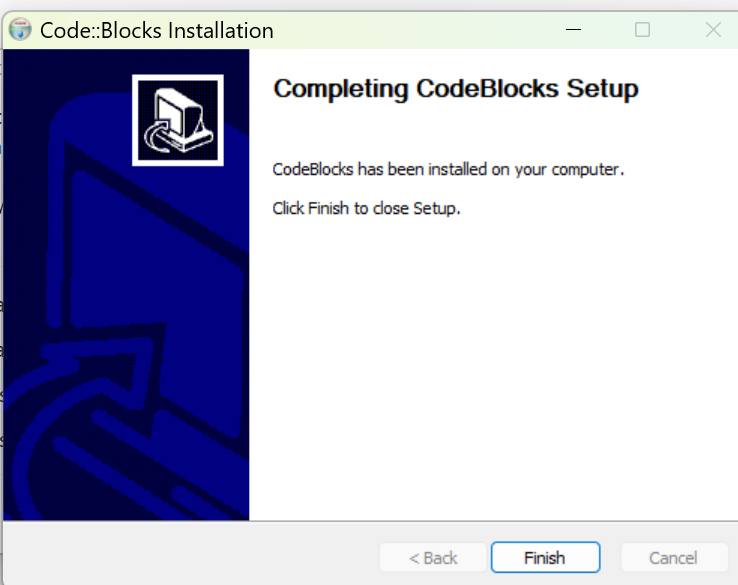


Figure 10 - The final Code::Blocks setup screen

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

### Starting up Code::Blocks

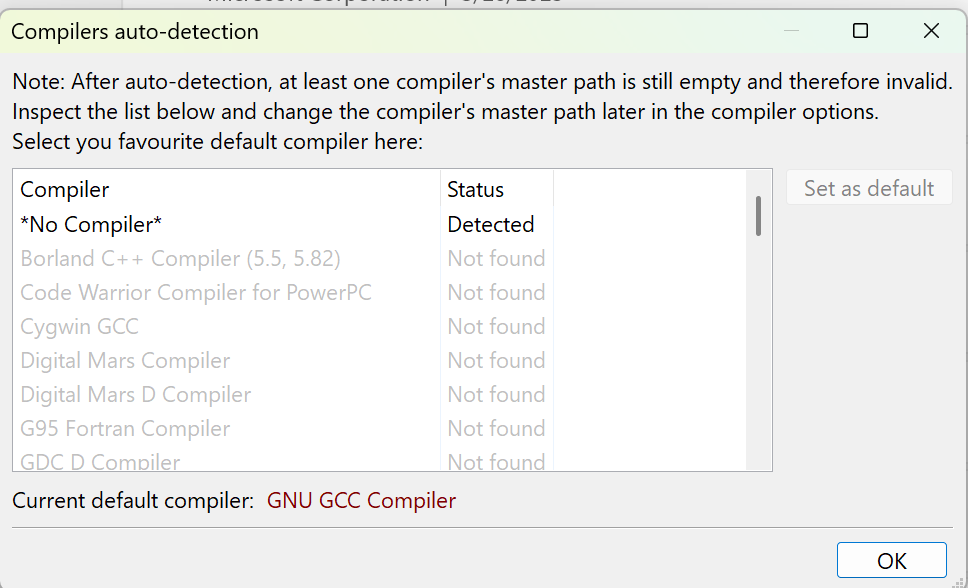


Figure 11 - Code::Blocks lists all the compilers it detected

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.

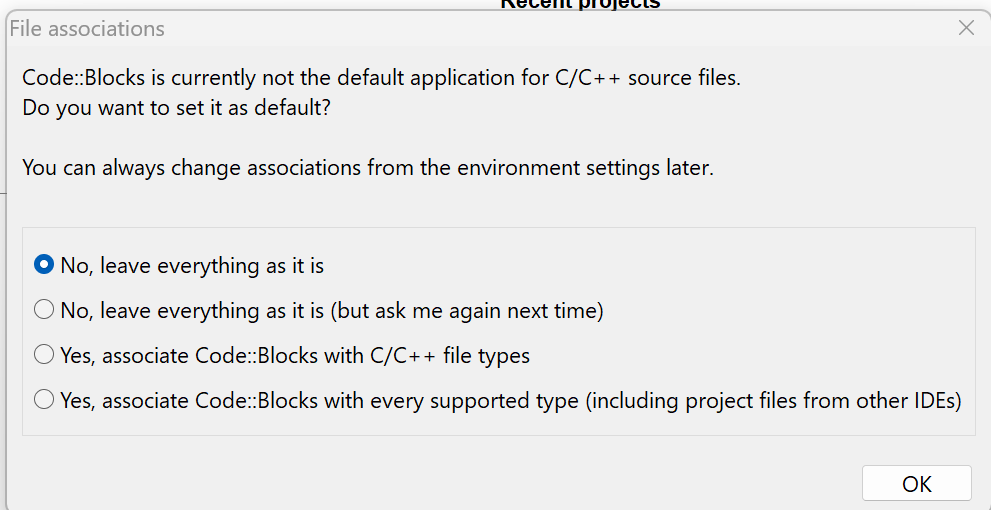


Figure 12 - Option to update file association

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

## Starting a Project with Code::Blocks

A screenshot of a computer

AI-generated content may be incorrect.

Figure 13 - 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

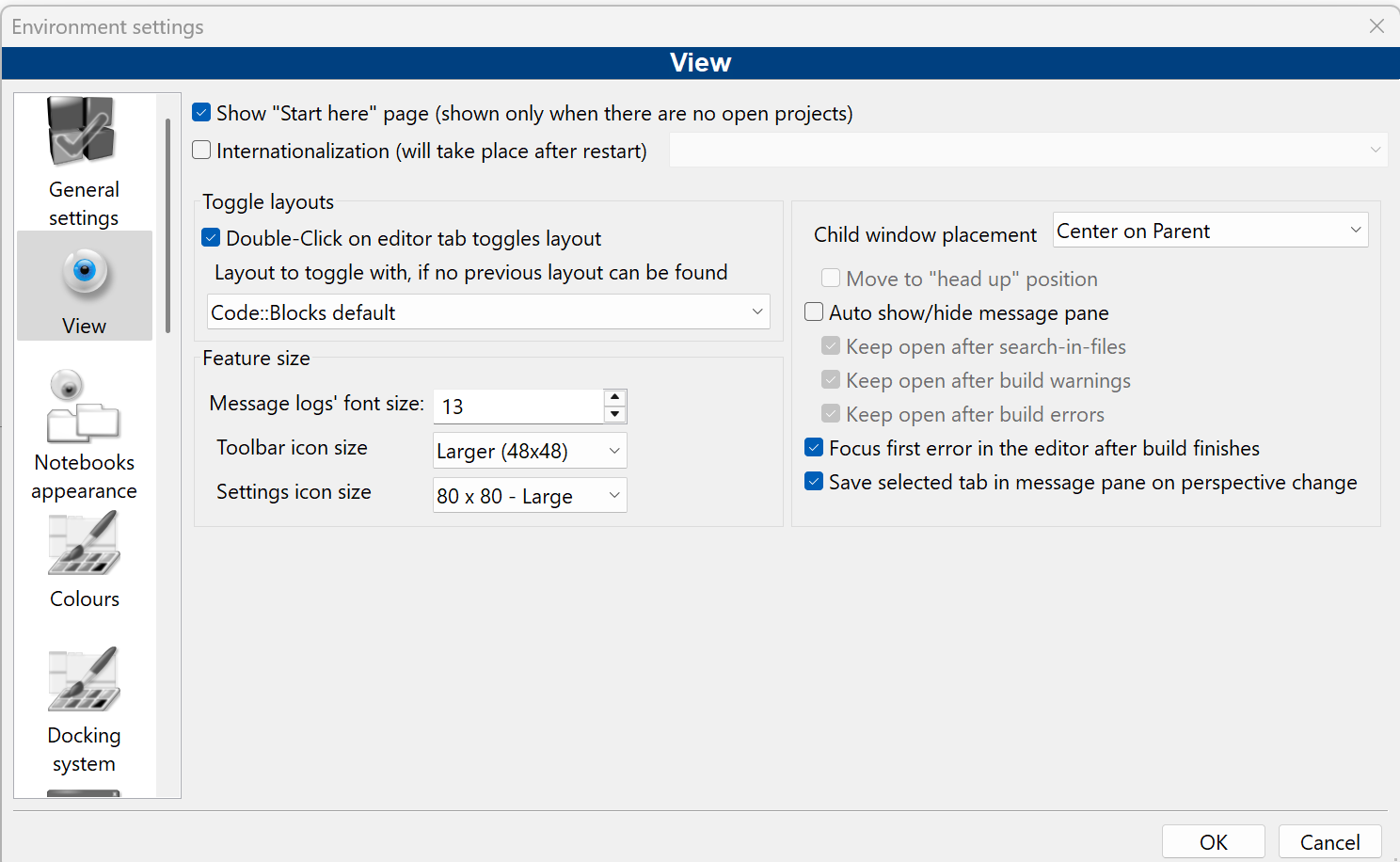
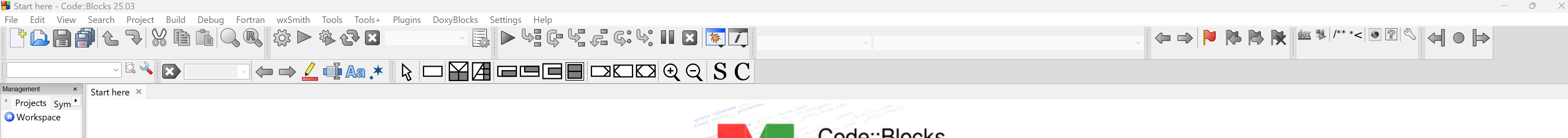


Figure 14 - Increasing the toolbar icon sizes

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



I also increased the font size used by:

* Select Settings 🡺 Editor

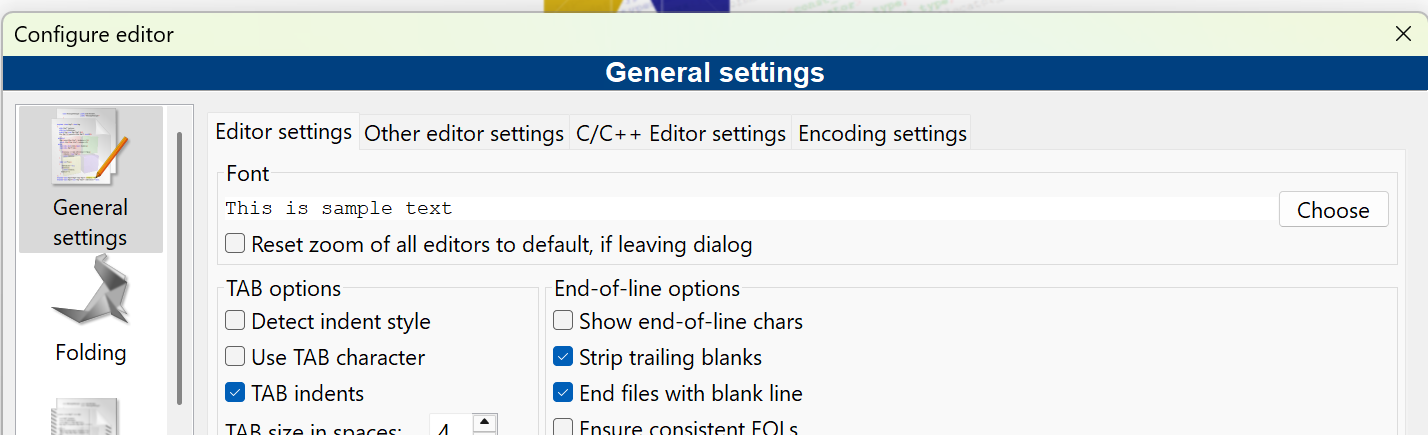


Figure 15 - Updating font-size

* Click on “Choose” button

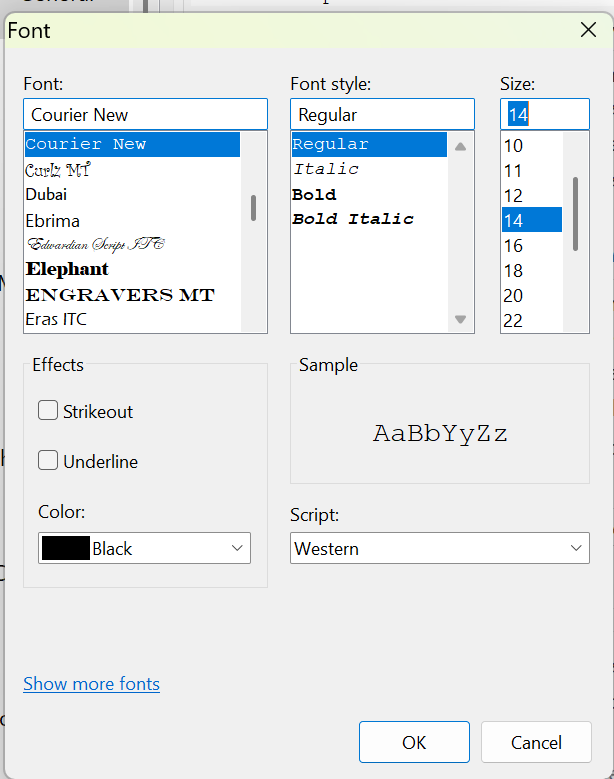


Figure 16 - Select desired font size

* I selected 14 and clicked on “OK”

### Creating the Project

* Click on “Create a new project” link



Figure 17 - Creating a new Project

* Select “Console Application”

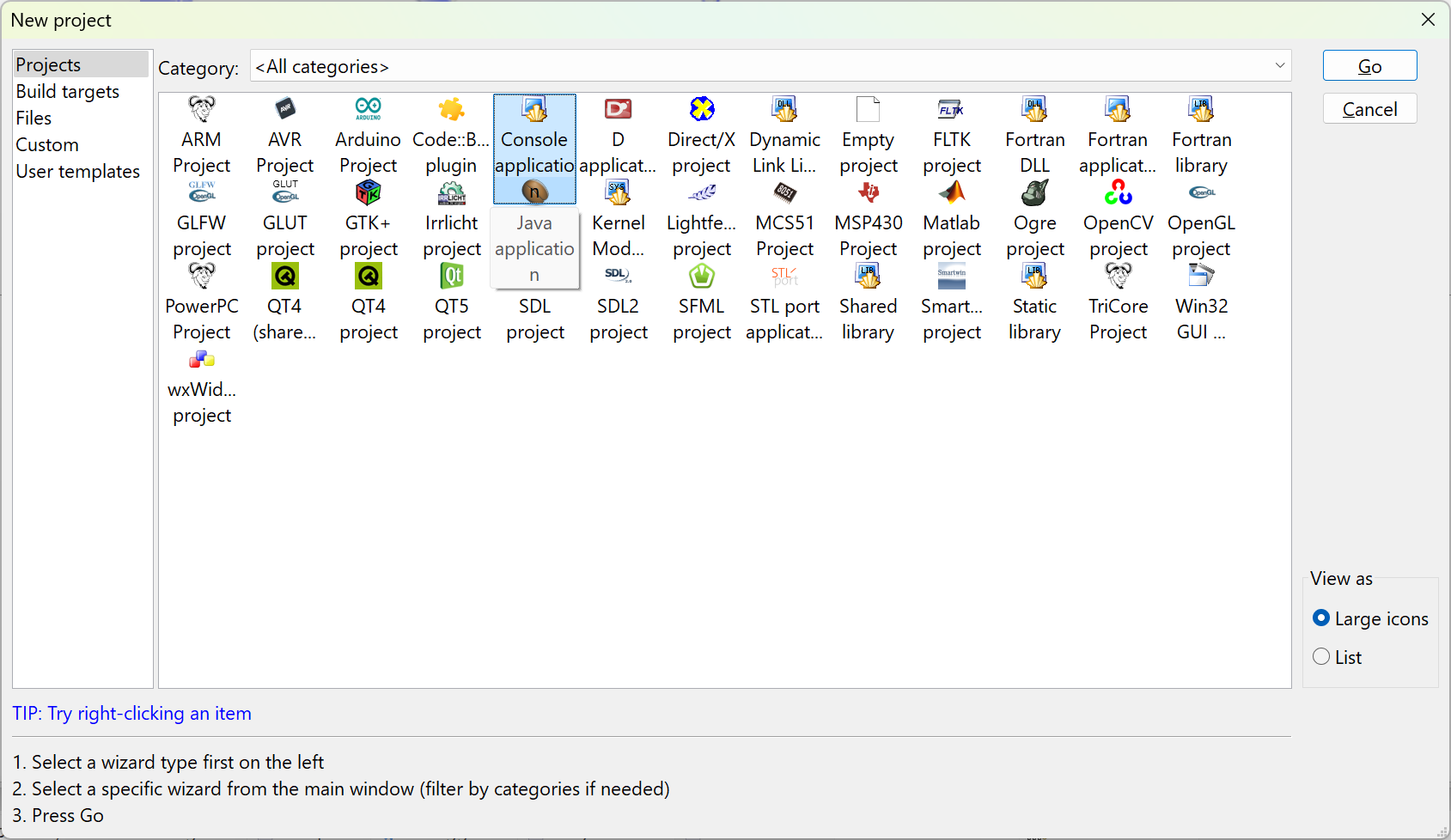


Figure 18 - Creating our first console application

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

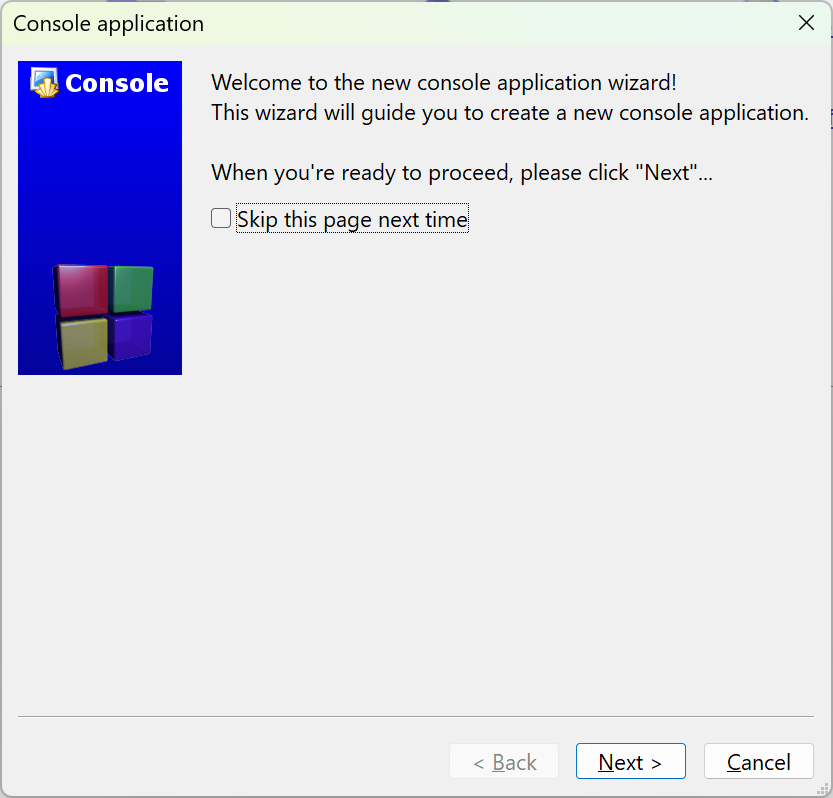
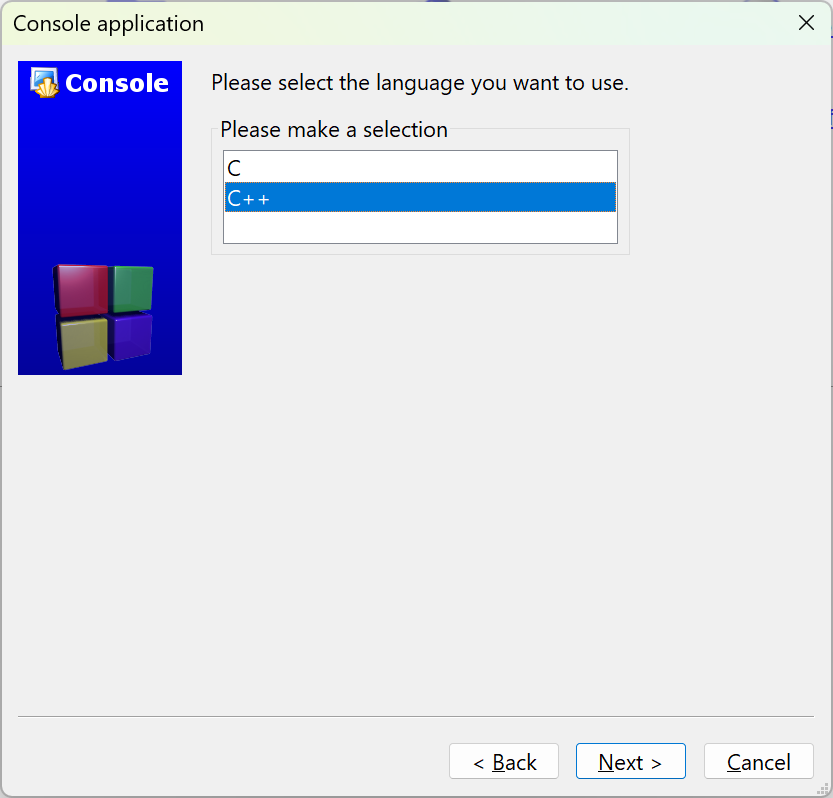


Figure 19 - The Console application wizard starting...

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



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

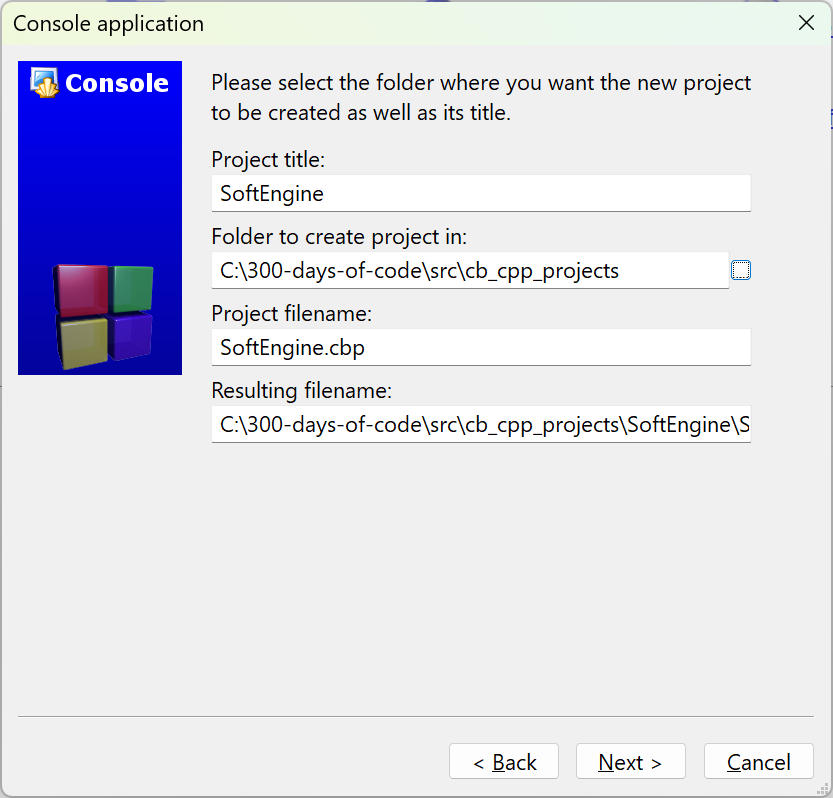


Figure 20 - Project information

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 >”

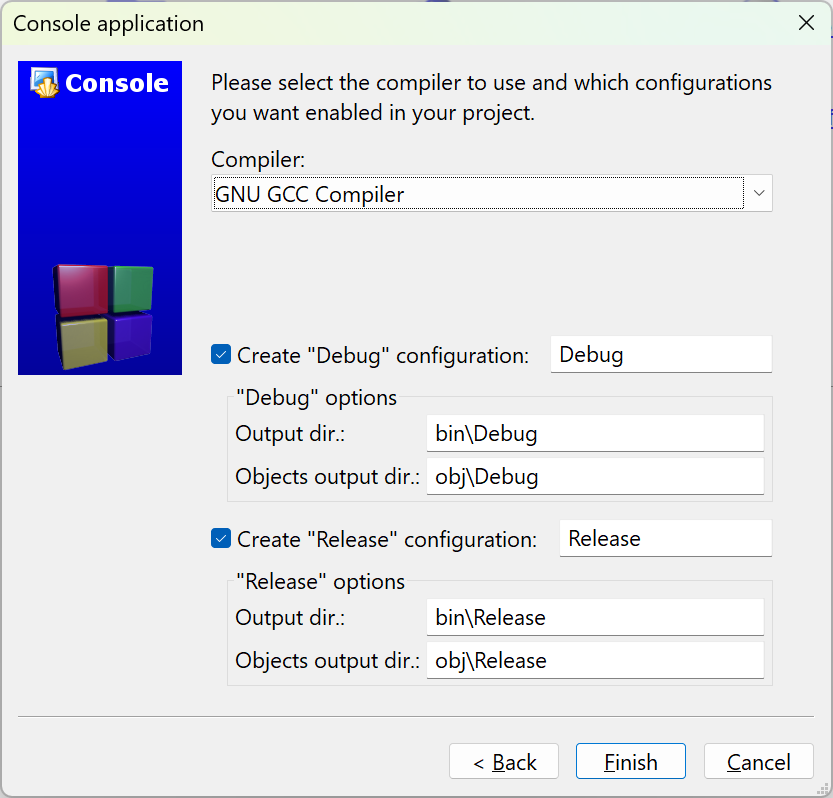


Figure 21 - Final step in creating our game engine project

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

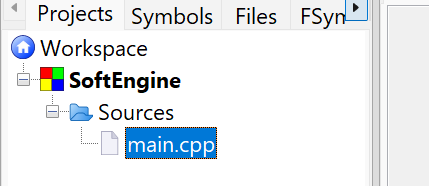


Figure 22 - Default main.cpp is created

* 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:

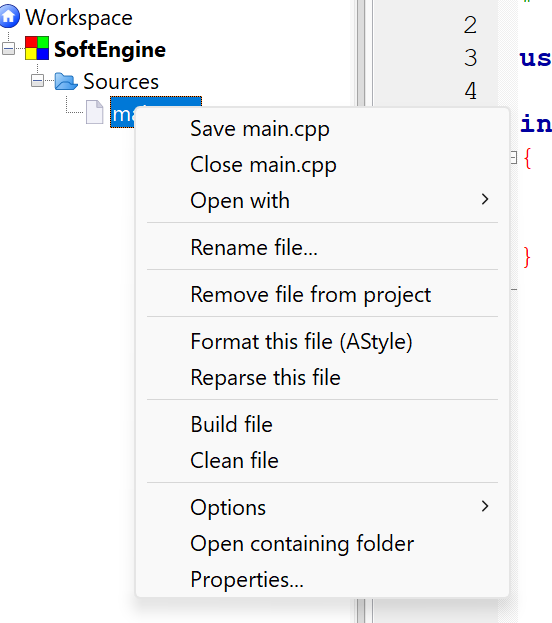


Figure 23 - Context menu for file

* Enter the name Main.cpp:

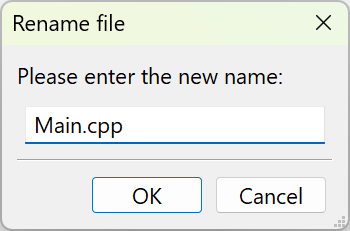


Figure 24 - Rename dialog box

* Click “OK”

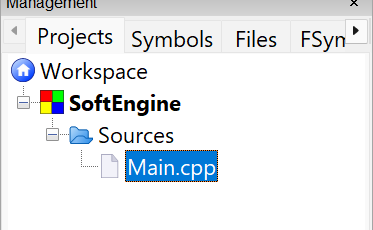


Figure 25 - Update name in Workspace

### 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 26 - 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 27 - 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 SDL

* Create a folder somewhere to install SDL files. I created a folder on the D:\ drive named SDL3.
* Navigate to the release github url (for me it is: <https://github.com/libsdl-org/SDL/releases/tag/release-3.2.10>)

Note: We will be using the latest version of SDL – SDL3.2.10. The video series uses SDL2-2.0.10. I prefer to explore the latest version in this series.

#### What version of SDL3 should I use?

A screenshot of a computer

AI-generated content may be incorrect.

Figure 28 - Versions of SDL3 to choose from

Since I am using Code::Blocks with mingw I will utilize SDL3-devel-3.2.10-mingw.zip version.

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 to D:\SDL3 and you will see:

A screenshot of a computer

AI-generated content may be incorrect.

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

A black screen with white text

AI-generated content may be incorrect.

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.

A screenshot of a computer

AI-generated content may be incorrect.

We will be using the **include** and **lib** folders.

To make my life easier I will copy the folders in x86\_64-w64-mingw32 to the top-level folder:

A screenshot of a computer

AI-generated content may be incorrect.

This will allow me to easily navigate to the two key folders: D:\SDL3\include and D:\SDL3\lib.

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

* Open the project (if not open) we created SoftEngine
* Go to Settings 🡺 Compiler…

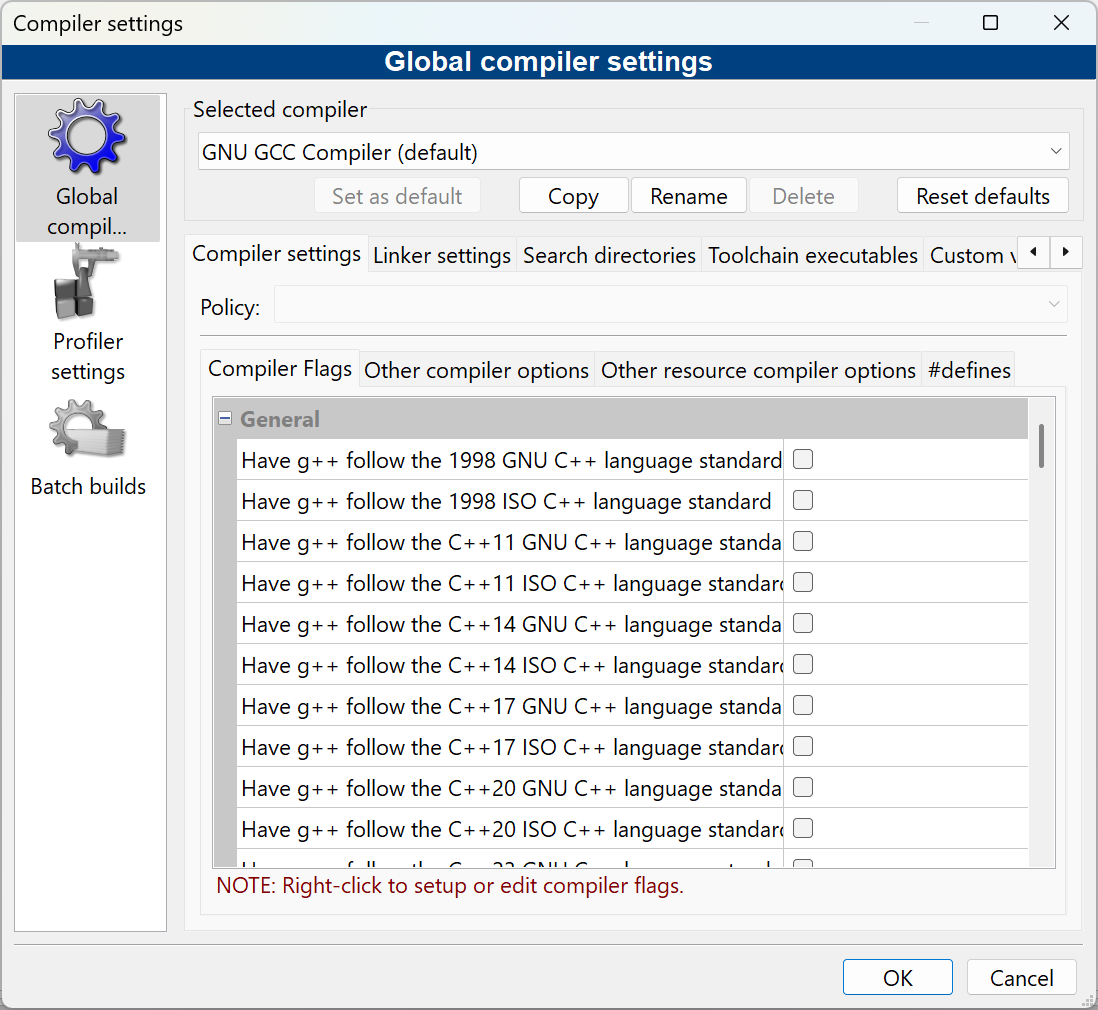


Figure 29 - Global compiler settings

* Click on the “Linker settings” tab

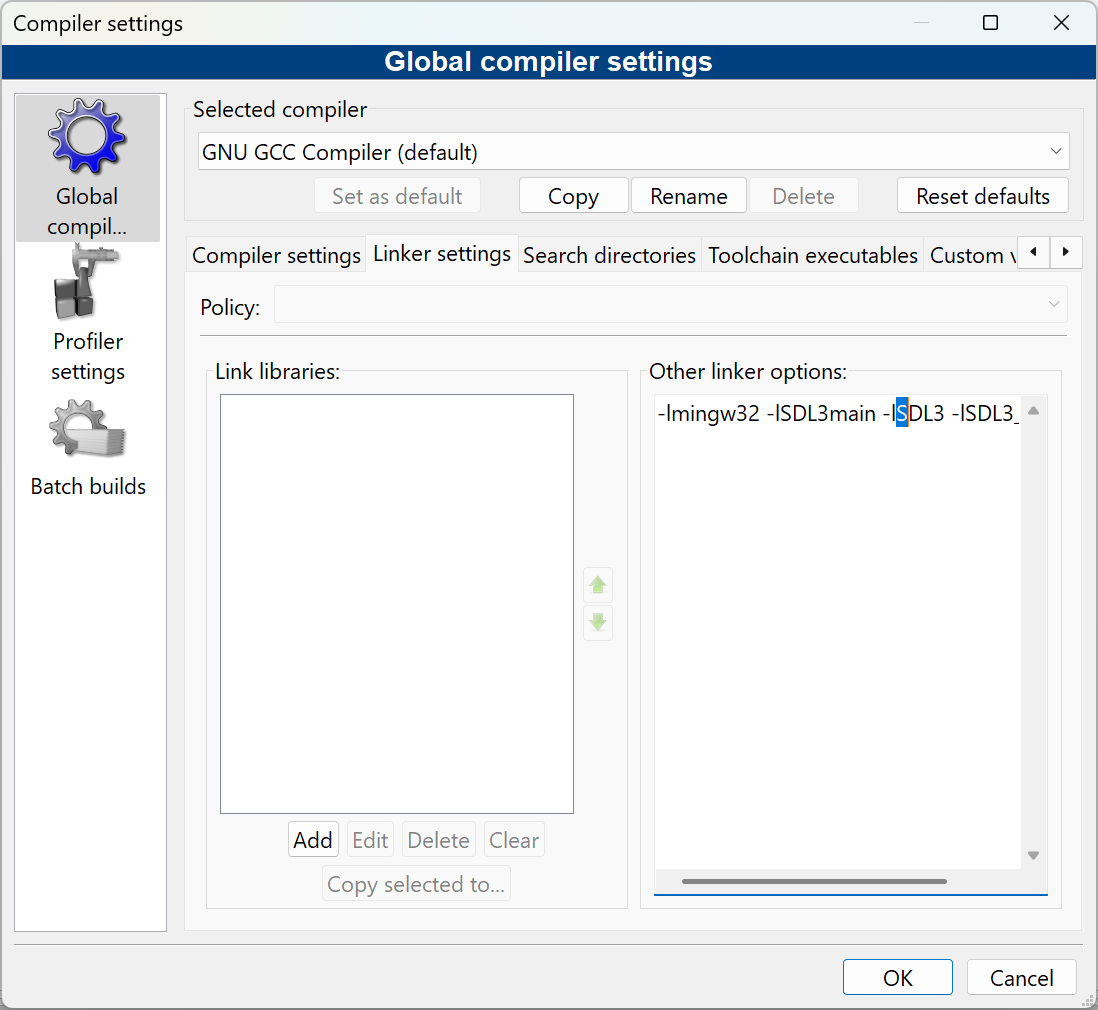


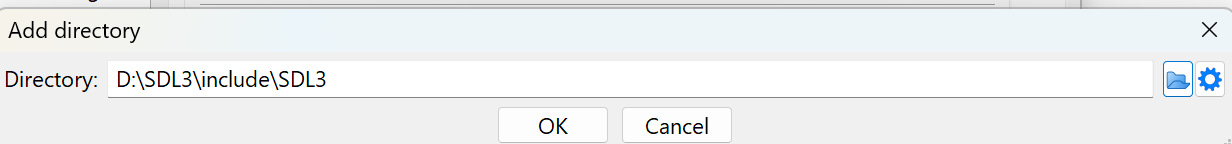
Figure 30 - Adding linker options

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

-lmingw32 ~~-lSDL3main~~ -lSDL3 -lSDL3\_image

**Note:** Remove -lSDL3main, it is NO LONGER required and will generate an error

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



Note: We will need to do the same thing for SDL3\_image later.

A screenshot of a computer

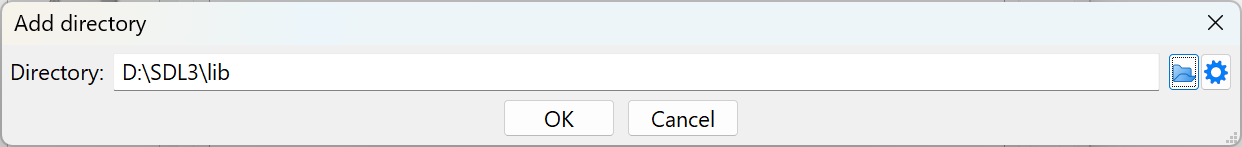
AI-generated content may be incorrect.

Figure 31 - Adding SDL3 include folder for the compiler

Note: The above differs from the video in that it adds D:\SDL3\include\SLD3 but that created an error. The above will require that when we include SDL.h we do it as:

1. #include "SDL3/SDL.h"

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



* Click on OK:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 32 - Adding the lib folder to Linker

### Getting SDL\_Image setup

* Locate the Github repository SDL\_Image that matches the version of SDL you are using. In my case it is, <https://github.com/libsdl-org/SDL_image> and <https://github.com/libsdl-org/SDL_image/releases>

A screenshot of a computer

AI-generated content may be incorrect.

Figure 33 - SDL\_image choices

* I will download the version similar to the SDL3 version I downloaded SDL3\_image-devel-3.2.4-mingw.zip
* I unzipped into the folder D:\SDL3\_image

A screenshot of a computer

AI-generated content may be incorrect.

Figure 34 - SDL3\_image folders

* Determine the right folder you will use by reading the INSTALL.md and copy the folders/directories to the top level

A screenshot of a computer

AI-generated content may be incorrect.

Figure 35 - SDL3\_image files accessible from the top-level

* Update the Global compiler settings, compiler tab, Search Directories to find the D:\SDL3\_image\include folder. You should see:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 36 - Adding SDL3\_image

* Add SDL3\_image \lib folder on the Linker sub-tab:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 37 - Adding SDL3\_image to Linker "Search directories"

### Testing the setup

* Update the code as follows:

1. #include <iostream>

2. #include "SDL3/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”

A screenshot of a computer

AI-generated content may be incorrect.

You should see no errors.

The video uses SDL\_INIT\_EVERYTHING to initialize all the SDL subsystems at the same time. This was considered a bad practice and changed so that in SDL3 you need to specify the subsystem individually.

The list of subsystems you can initialize:

* **SDL\_INIT\_AUDIO**: Initializes the audio subsystem.
* **SDL\_INIT\_VIDEO**: Initializes the video subsystem (should be done on the main thread).
* **SDL\_INIT\_JOYSTICK**: Initializes the joystick subsystem.
* **SDL\_INIT\_HAPTIC**: Initializes the haptic (force feedback) subsystem.
* **SDL\_INIT\_GAMEPAD**: Initializes the gamepad subsystem (also initializes the joystick subsystem).
* **SDL\_INIT\_EVENTS**: Initializes the events subsystem.
* **SDL\_INIT\_SENSOR**: Initializes the sensor subsystem.
* **SDL\_INIT\_CAMERA**: Initializes the camera subsystem.

You can use SDL\_Init() or SDL\_InitSubSystem(). For our example we will initialize the SDL\_INIT\_VIDEO and SDL\_INIT\_AUDIO.

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