PB's Guide to Starting with wxWidgets on Microsoft Windows with MinGW and Code::Blocks

Guide Author: PB <pbfordev@gmail.com>

Guide Version: 2.4

Versions of Software Used in Guide

wxWidgets: 3.1.4

MinGW: MSYS2 mingw-w64-i686-toolchain (GCC 10.2 at the time of writing)

Code::Blocks: 20.03

The latest version of this guide is at https://github.com/PBfordev/wxpbguide.

Table of Contents

1 Introduction 4

2 Basic wxWidgets Information 5

2.1 wxWidgets Versions 5

2.2 Shared vs Static Linking 5

2.3 Multilib vs Monolithic Build 5

2.4 Debug and Release Configuration 6

2.5 32-bit vs 64-bit Build 6

2.6 To Build or Not to Build 6

3 Building wxWidgets with MinGW 7

3.1 Choosing MinGW Distribution 7

3.2 Installing MSYS2 7

3.3 Installing MSYS2 packages for developing C++ applications 7

3.4 Downloading wxWidgets Sources 9

3.5 Adding System Environment Variable WXWIN 11

3.6 The Build 13

3.6.1 Starting Simple 13

3.6.2 Improving Build Commands 14

3.6.3 Making wxWidgets Build Faster 15

3.6.4 Viewing Build Results 16

3.6.5 Using the Minimal Sample to Test the Build 17

3.6.6 Cleaning the Build 18

3.6.7 Building wxWidgets Statically 19

3.6.8 More Build Options 19

4 Setting Up wxWidgets Project with Code::Blocks and MinGW 20

4.1 Installing Code::Blocks 20

4.2 Configuring the Debugger 21

4.3 Configuring the Compiler 22

4.4 Creating Global Variable wx 23

4.5 Creating a wxWidgets Project with Project Wizard 24

4.5.1 Selecting wxWidgets Project Template 24

4.5.2 Wizard Page 1: Welcome 25

4.5.3 Wizard Page 2: wxWidgets Version 26

4.5.4 Wizard Page 3: Project Title and Paths 27

4.5.5 Wizard Page 4: Project Author 28

4.5.6 Wizard Page 5: GUI Builder and Application Type 29

4.5.7 Wizard Page 6: wxWidgets Location 30

4.5.8 Wizard Page 7: Compiler and Debug/Release Configurations 31

4.5.9 Wizard Page 8: Various Configuration Options 32

4.5.10 Wizard Page 9: Advanced Options 33

4.5.11 Wizard Page 10: Additional Libraries 34

4.6 Adjusting Project Settings 35

4.6.1 Setting C++ Standard Used in the Project 36

4.6.2 Setting Preprocessor Define *NDEBUG* for the Release Build Target 37

4.6.3 Setting Other Project Options 37

4.7 Finally Finished! *38*
4.8 Extra Reading: wxWidgets Code::Blocks Project Under the Hood *39*
4.8.1 Compiler Include Folders 39
4.8.2 Preprocessor Defines 39
4.8.3 Linker Settings 40
4.8.4 Using Static Instead of Shared wxWidgets Build 40
Document History **41**

1 Introduction

This Microsoft Windows specific guide hopes to assist the readers with decisions regarding which wxWidgets version and configuration to use and show how to build wxWidgets using MinGW with various parameters affecting the resulting build. It also describes in detail setting-up a wxWidgets project in popular IDE Code::Blocks.

This guide is not a reference manual, it is a tutorial for new users of wxWidgets, MinGW, and Code::Blocks; supposed to be followed step by step.

This guide does not cover actual wxWidgets programming nor using Code::Blocks (besides the steps necessary to set up the IDE to be able to build a wxWidgets project).

This guide assumes that the readers did their research and decided that for their applications, the most suitable choices as the programming language and the GUI toolkit are C++ and wxWidgets; therefore the pros and cons of these two choices are not discussed here.

This guide does not intend to replace the official documentation, it hopes to complement it. Its author is not affiliated with Microsoft, wxWidgets, MSYS2, mingw-w64, or Code::Blocks.

2 Basic wxWidgets Information

2.1 wxWidgets Versions

wxWidgets has two supported branches, called *stable* and *development*. The stable branch version has an even minor version number (e.g., 3.0.0 or 3.0.5) while the development branch has an odd number for its minor version (e.g., 3.1.0 or 3.1.4). The current development branch is also sometimes called “trunk” or “master”.

For wxWidgets, when the branch is called stable, the API and ABI of the Major.Minor version is sealed. This means that all the versions in the branch are API- and binary-wise fully compatible, i.e., one can switch to newer releases of the branch (e.g., from 3.0.0 to 3.0.5) without encountering any (adverse) compile- and run-time differences compared to the older releases. No such guarantees are made for the development branch and in fact, as new features are added, the API (and therefore ABI) does change. In other words, it does not mean that the stable branch is more stable as in being less prone to crashing your application than the development one. However, it means that no new features can be added to the stable branch after its initial release. The new releases in the stable branches are for bug fixes only, but even not all bug fixes can be backported from the development to the stable branch, for example when the code for the two branches has diverted too much.

The number of differences in new features introduced and bugs fixed between the two branches increases with time passed since the initial stable version release. I advise checking the changelog describing the differences between the two versions, such as this one for 3.0 vs 3.1.4 and deciding for yourself whether the difference is worth using the development branch. The core developers suggest that the development branch is generally suitable for production use.

Lastly, you can also embrace the “Live at Head” concept and use the current GIT head. If you want to do this, make sure to read the GIT instructions.

2.2 Shared vs Static Linking

wxWidgets can be built either as a static or shared (DLL) library. Both have their pros and cons, so which one to choose depends on use case. When the shared configuration is used, all the necessary DLLs must be shipped with the application and the install package is usually larger then when using the static libraries where only the used code is linked into the final executable. On the other hand, if one has a suite of applications using wxWidgets, having them all using the same wxWidgets DLLs is beneficial. Similarly, for applications with plugins, linking to wxWidgets dynamically ensures that the both the application and the plugin are using the same library.

In practice for most cases the differences from both the user and the programmer’s point of view are usually negligible, with static linking perhaps being a bit more convenient regarding the application distribution but also being slower when linking the application during its each and every build.

GCC allows linking the run-time libraries statically, so if all the libraries the application uses can be linked statically and support being built with the statically-linked compiler run-time libraries, the whole application code can be contained in a single executable which does not require any DLLs at all (see chapters 3.6.7 and 4.6.3).

I would use the static build only if the application was small, used just wxWidgets as the compiled library and I was concerned about how much bandwidth and disk space the installer and the installed application take. Otherwise, I would prefer dynamic linking, i.e., the shared build and this build will be preferably used in this guide.

2.3 Multilib vs Monolithic Build

By default, wxWidgets is built as a set of several libraries, which is called the Multilib build. wxWidgets can also be built in the so-called Monolithic build, where all the wxWidgets libraries are merged into just two DLLs instead of fifteen. I cannot think of even one significant advantage of using the Monolithic build, so I recommend sticking to the default Multilib one and will use this build in this guide.

2.4 Debug and Release Configuration

wxWidgets on Windows is supposed to be built in both Debug and Release configurations, where the Debug configuration is used for development (contains some useful runtime checks helping to find bugs) and the Release configuration for distribution to the end user. See also the Debugging programmer guide for more information.

2.5 32-bit vs 64-bit Build

These days, if we are speaking about Windows 7 and newer, 64-bit versions seem to be largely prevalent. Nevertheless, some users may still be using a 32-bit version of their Windows. Unless an application needs 2 or more GB of address space, uses a library available only as 64-bit, or measurably benefits from the 64-bit instruction set; building the application (and hence using the 32-bit wxWidgets build) as 32-bit may generally still be a better choice. One minor advantage of 32-bit applications may also be that Windows 10X at the time of writing can run only x86 and not x64 applications.

2.6 To Build or Not to Build

wxWidgets website provides prebuilt binaries for several popular compilers so one does not have to build wxWidgets on their own. However, I suggest building wxWidgets on your own, it is simple, relatively quick, and you may tailor the build to your needs.

You may also want to use other MinGW configuration than provided (e.g., use build with posix instead of win32 threads). Moreover, if you want to link wxWidgets statically, you have no choice, as the prebuilt wxWidgets binaries are provided only for the shared (DLL) configuration.

3 Building wxWidgets with MinGW

3.1 Choosing MinGW Distribution

MinGW is a free and open source software that can be used to create Microsoft Windows applications. It uses GNU Compiler Collection (GCC) and ships with Microsoft Windows header files and import libraries. There are several different "distributions" of MinGW with differences being the bundled compiler version and completeness of headers and libraries needed for Microsoft Windows.

One should pick a distribution that (a) produces correct code and (b) supports required C++ features (such as C++17). wxWidgets tries hard to address possible issues with missing declarations but it is still best to use a distribution with a good header files set. I believe that at the time of writing, the best MinGW is mingw-w64 and this will be the toolchain used in this guide, in its variant for the 32-bit architecture.

Project mingw-w64 used to provide its own full compiler toolchain distribution, which allowed to choose the GCC version and configuration for threads and exception handling. The older versions of the guide used this distribution; however, the last version is with GCC 8.1 released in 2018. Therefore, this guide now uses a distribution from the MSYS2 project, which seems to be actively maintained.

3.2 Installing MSYS2

Go to https://repo.msys2.org/distrib/x86_64/ and download the latest version of the installer executable. Launch the installer, leave the options at their defaults and let it finish the installation. After the installation, the packages must be updated, so launch the MSYS environment, either from the last page of the installer or via Start Menu, where it should be listed as **MSYS2 MSYS**. On MSYS command prompt enter **`pacman -Syuu`** and let MSYS update all packages. You can use this command to update all installed packages whenever needed.

3.3 Installing MSYS2 packages for developing C++ applications

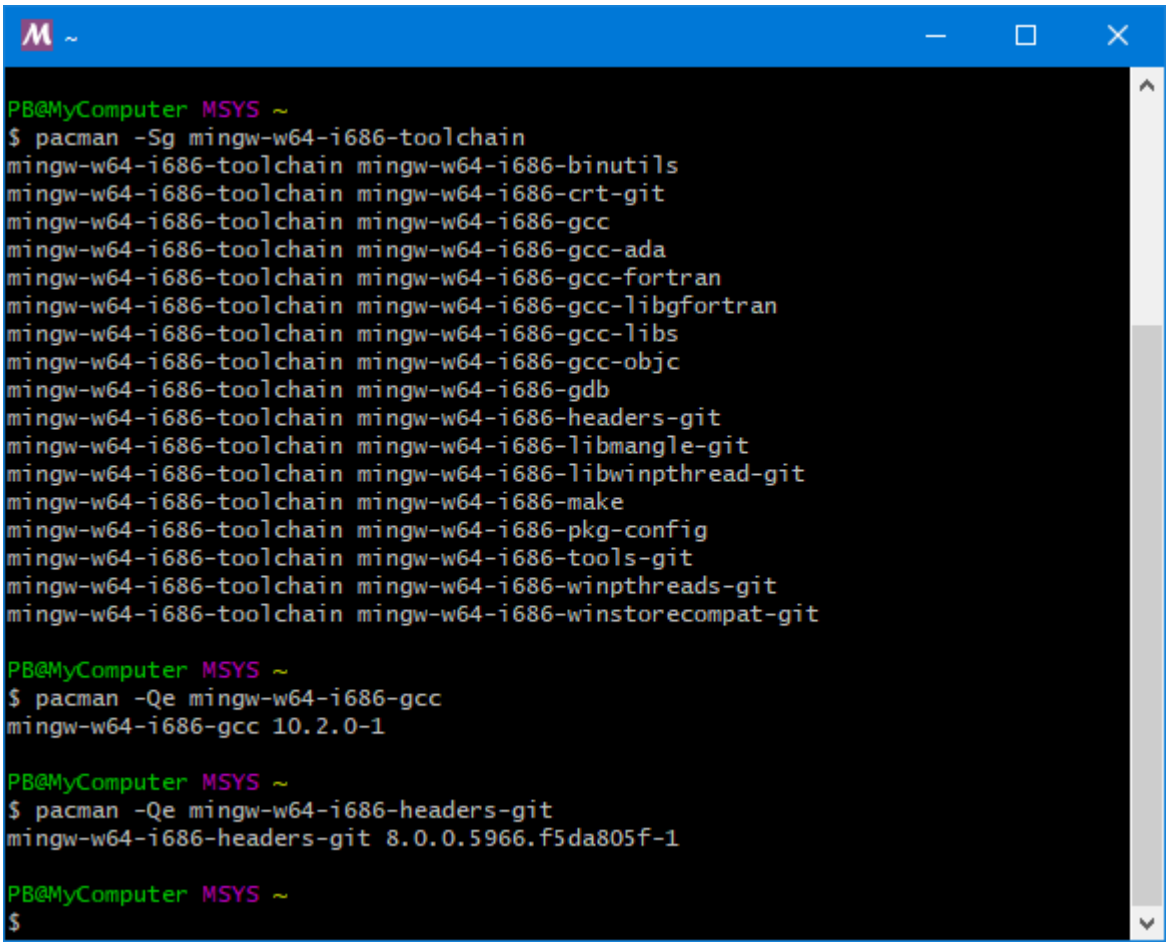
Now we need to install all the tools required for 32-bit GCC MinGW toolchain. They are easiest to install as a group named *mingw-w64-i686-toolchain*, so in the MSYS command prompt enter **`pacman -S mingw-w64-i686-toolchain`**; when asked to select the packages just press **`<Enter>`** to select them all and then confirm that you want to continue with installation. Please notice that the toolchain executables are installed in *mingw32\bin* subfolder of your MSYS2 root folder, e.g., *c:\msys64\mingw32\bin*.

Do not add the folder where you installed MinGW to your *PATH* system environment variable. It may seem convenient, but it can (and sometimes does) create problems. Moreover, make sure that no other GCC-based compiler toolchain is there, as this could cause some odd issues when building the applications as well as when running them. Use the command prompt and either **`set`** command to inspect the content of *PATH* or better yet, use **`where`** command (**`where $PATH:gcc.exe`**) to confirm that no GCC-based compiler is installed in the folders added to *PATH* (see also included *test-gcc-not-in-path.bat*).

You can list which packages are included in *mingw-w64-i686-toolchain* group by using **`pacman -Sg mingw-w64-i686-toolchain`** or in your web browser at https://packages.msys2.org/group/mingw-w64-i686-toolchain. You can tell the version of an installed package by using **`pacman -Qe <package-name>`**, e.g., **`pacman -Qe mingw-w64-i686-gcc`**, see Figure 3-1.

Note. If you want to use a compiler producing 64-bit binaries, install the package named *mingw-w64-x86_64-toolchain* instead. In this case the tools are installed in *<MSYS2ROOT>\mingw64\bin* and you must adjust the paths used in this guide accordingly.

Figure 3-1 Displaying version information for MSYS2 packages



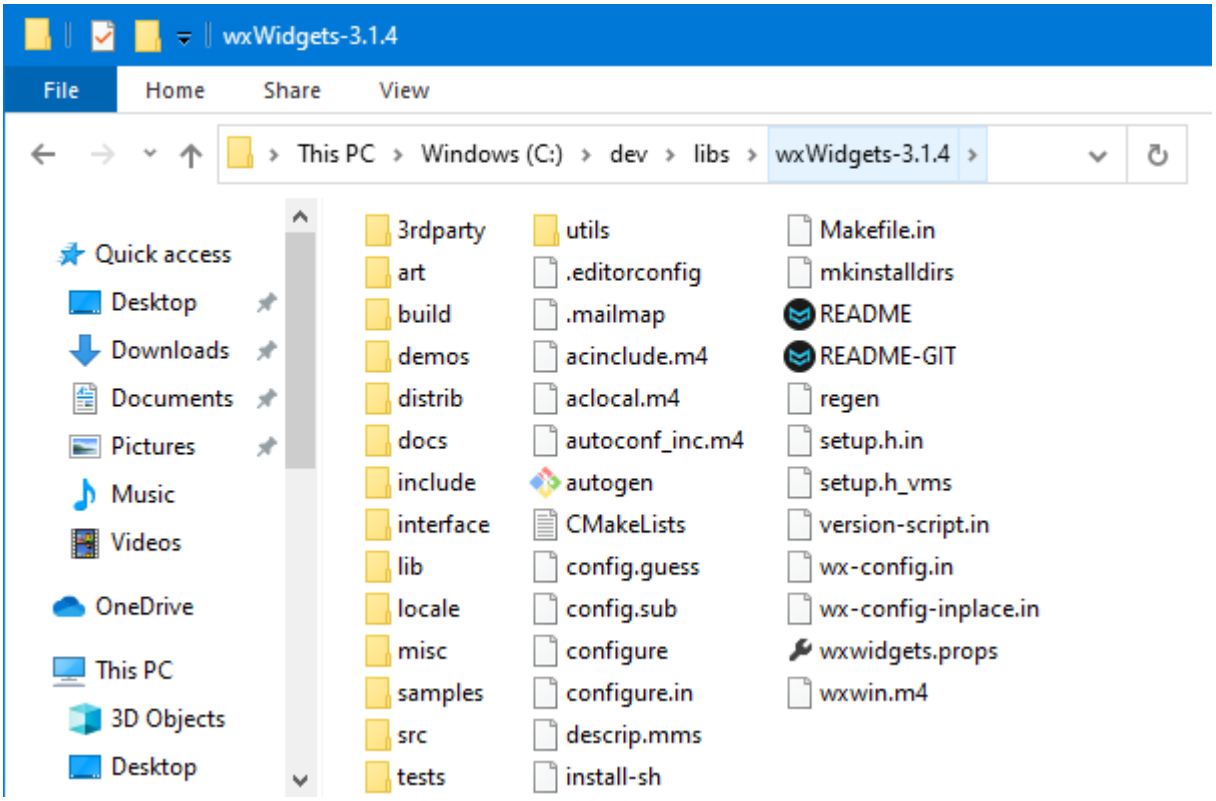
3.4 Downloading wxWidgets Sources

This guide shows how we can build wxWidgets by ourselves, so we will download sources and build the binaries. This guide will use latest development release, which at the time of writing was 3.1.4. Go to https://wxwidgets.org/downloads/ and download the **Windows ZIP** archive file, see Figure 3-2.

Figure 3-2 Downloading wxWidgets source code

Extract the archive *wxWidgets-3.1.4.zip* to a folder with the folder hierarchy preserved (and do not use a path with spaces in it) such as *c:\dev\libs\wxWidgets-3.1.4*, see Figure 3-3. This wxWidgets root folder will be referred to as *WXDIR* further on in this guide. In other words, if this guide tells you to go for example to *WXDIR\samples* folder, you are expected to go to folder such as *c:\dev\libs\wxWidgets-3.1.4\samples*.

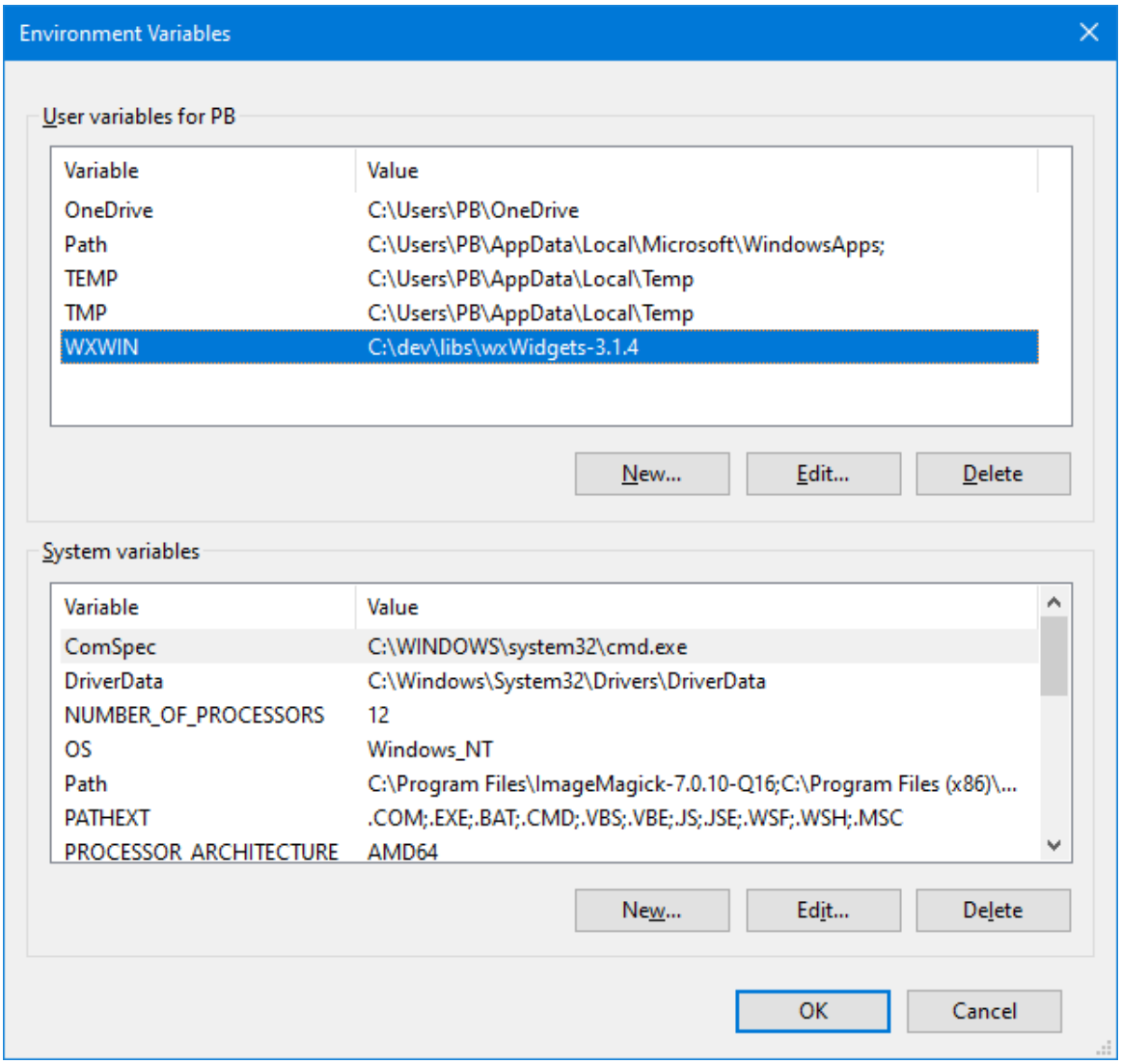
Figure 3-3 wxWidgets root folder shown in File Explorer



3.5 Adding System Environment Variable *WXWIN*

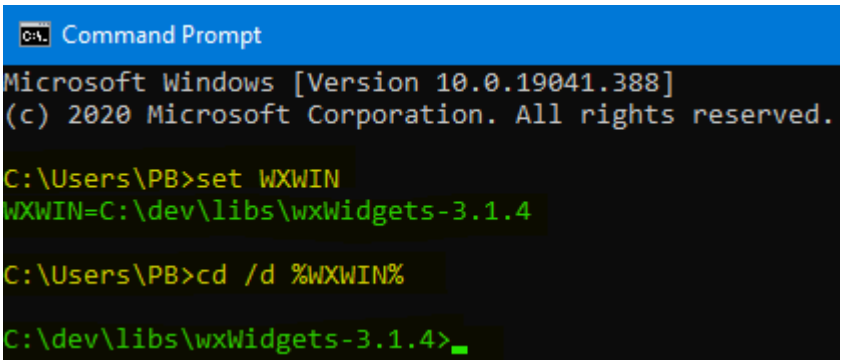
Use internet to find out how to set a system environment variable on your system (this is quite different depending on the Windows version you are using) and create system environment variable *WXWIN* pointing to your *WXDIR* (such as *c:\dev\libs\wxWidgets-3.1.4*), see Figure 3-4. It may be best to restart the computer after that, as the environment variables of running processes are not updated, which can sometimes cause issues.

Figure 3-4 Setting *WXWIN* environment variable



Once you are done, you may launch the Command Prompt and use the command **`set WXWIN`** to verify that the *WXWIN* system environment variable is properly set (Figure 3-5), see also bundled *test-wxwin.bat*.

Figure 3-5 Verifying *WXWIN* environment variable



Note. If you for some odd reason decide not to set *WXWIN* as described above and recommended by wxWidgets developers, you need to replace **`%WXWIN%`** with the hardcoded *WXDIR* path (e.g., *c:\dev\libs\wxWidgets-3.1.4*) everywhere **`%WXWIN%`** is used in this guide.

3.6 The Build

We now have wxWidgets sources as well as the software to build it so let's build wxWidgets. You can use several approaches to build wxWidgets with MinGW (bundled GCC makefile, CMake, configure...), in this guide I will use the bundled GCC makefile located in *WXDIR\build\msw* and Windows command prompt (not MSYS environment).

Note. The batch files referenced in this chapter are available in the bats/build folder of the GIT repository.

3.6.1 Starting Simple

We are going to build wxWidgets in Debug and Release configurations in the shared (DLL) Multilib build using multiple commands. We need to (1) add the compiler directory to *PATH* to make it available for the following commands, (2) change the current directory to the folder with the makefile, and (3) start the shared build for Debug and Release configurations. The commands needed for these three steps are probably best run as a batch file (a text file with .bat extension containing commands for the command interpreter) and they could look like those in Listing 3-1.

Note. I recommend just reading and understanding the build commands in chapters 3.6.1 and 3.6.2 but actually building wxWidgets as explained in chapter 3.6.3, which should be much faster.

Listing 3-1 Basic build commands (available as *build-library-shared-simplest.bat*)

```
set PATH=c:\msys64\mingw32\bin;%PATH%

cd /d %WXWIN%\build\MSW

mingw32-make -f makefile.gcc SHARED=1 BUILD=debug
mingw32-make -f makefile.gcc SHARED=1 BUILD=release
```

The first command adds MinGW's bin folder to *PATH* (if your MinGW location is different, you need to use that one), but only for the commands in this batch file. Please notice that we added the folder at the beginning of *PATH*, which means that our MinGW programs (make, compiler, linker…) will be hopefully picked if there is another GCC-based compiler toolchain in *PATH* (it should not be, see chapter 3.2, but better be safe than sorry).

The second command changes the current directory to the wxWidgets MSW build folder, because that is where *makefile.gcc* is located. The other two commands actually build wxWidgets with the make, using parameters affecting the build.

Firstly, **`-f makefile.gcc`** tells the make we want to use file *makefile.gcc* as the makefile. Secondly, using **`1`** as the value for the parameter **`SHARED`** tells it the build will be dynamic (for the static build we would simply omit the **`SHARED`** parameter as the static build is the default, or use **`SHARED=0`** to make our intent clearer). Thirdly, the value of parameter **`BUILD`** decides whether to build the Debug or the Release build, as we want both, we need to run the make twice with a different value of **`BUILD`**. If parameter **`BUILD`** is not used, the build defaults to Debug. You may notice there is no parameter asking to build the Mulitlib build: Multilib is built by default; we would have to ask for a monolithic build with **`MONOLITHIC=1`**.

3.6.2 Improving Build Commands

There is nothing wrong with the commands used in the previous section but there is still room for improvement. Firstly, you probably want to use all nice modern C++ features, so we will tell the compiler we want to use C++17 standard (GCC 10 defaults to C++14 standard). Secondly, it is possible that the build fails, so it would be great if the command window stayed open in this case so we can see what went wrong. Listing 3-2 contains the improved version of the batch file, with the differences in blue and green. Lines starting with **`REM`** are comments (REMarks) and are not executed.

Listing 3-2 Improved build commands (available as *build-library-shared.bat*)

```
set PATH=c:\msys64\mingw32\bin;%PATH%

cd /d %WXWIN%\build\MSW


REM Build wxWidgets Shared Debug configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17"
IF %ERRORLEVEL% NEQ 0 goto FAIL

REM Build wxWidgets Shared Release configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=release CXXFLAGS="-std=c++17"
IF %ERRORLEVEL% NEQ 0 goto FAIL


goto SUCCESS

:FAIL
echo Build failed!
pause
goto FINISHED

:SUCCESS

:FINISHED
```

If you are not familiar with batch files, the above may appear intimidatingly complex, but it is actually very simple. You probably deduced that **`CXXFLAGS="-std=c++17"`** was used to tell the compiler we want it to use C++17 standard. However, the rest of new commands can look a bit cryptic. Line **`IF %ERRORLEVEL% NEQ 0 goto FAIL`** tells the command processor that if the build failed (make's exit code does not equal 0), the command processing should resume at label **`FAIL`**, where **`echo`** command is used to display "Build failed!" and **`pause`** command to keep the command line window open. You can see nothing is done at label **`SUCCESS`**, which means that if both Debug and Release build succeed, the command line window will be closed.

3.6.3 Making wxWidgets Build Faster

MinGW make supports parallel building, where multiple files can be processed simultaneously. If your computer has several physical CPU cores, parallel building can significantly speed up building large code bases such as wxWidgets. There is a `-jN` parameter where `N` is the number of CPU cores MinGW make can use, for example `-j4` will use 4 CPU cores. However, there is an issue with wxWidgets GCC makefile and parallel building due to some dependencies. We can work around this issue with building *setup_h* target first without using parallel compilation and then running make again with the same parameters but without *setup_h* and with `-jN`. Listing 3-3 contains commands for creating a shared build using parallel building with 4 jobs.

Listing 3-3 Build commands for dynamic wxWidgets build using parallel build with 4 jobs (available as *build-library-shared-parallel.bat*)

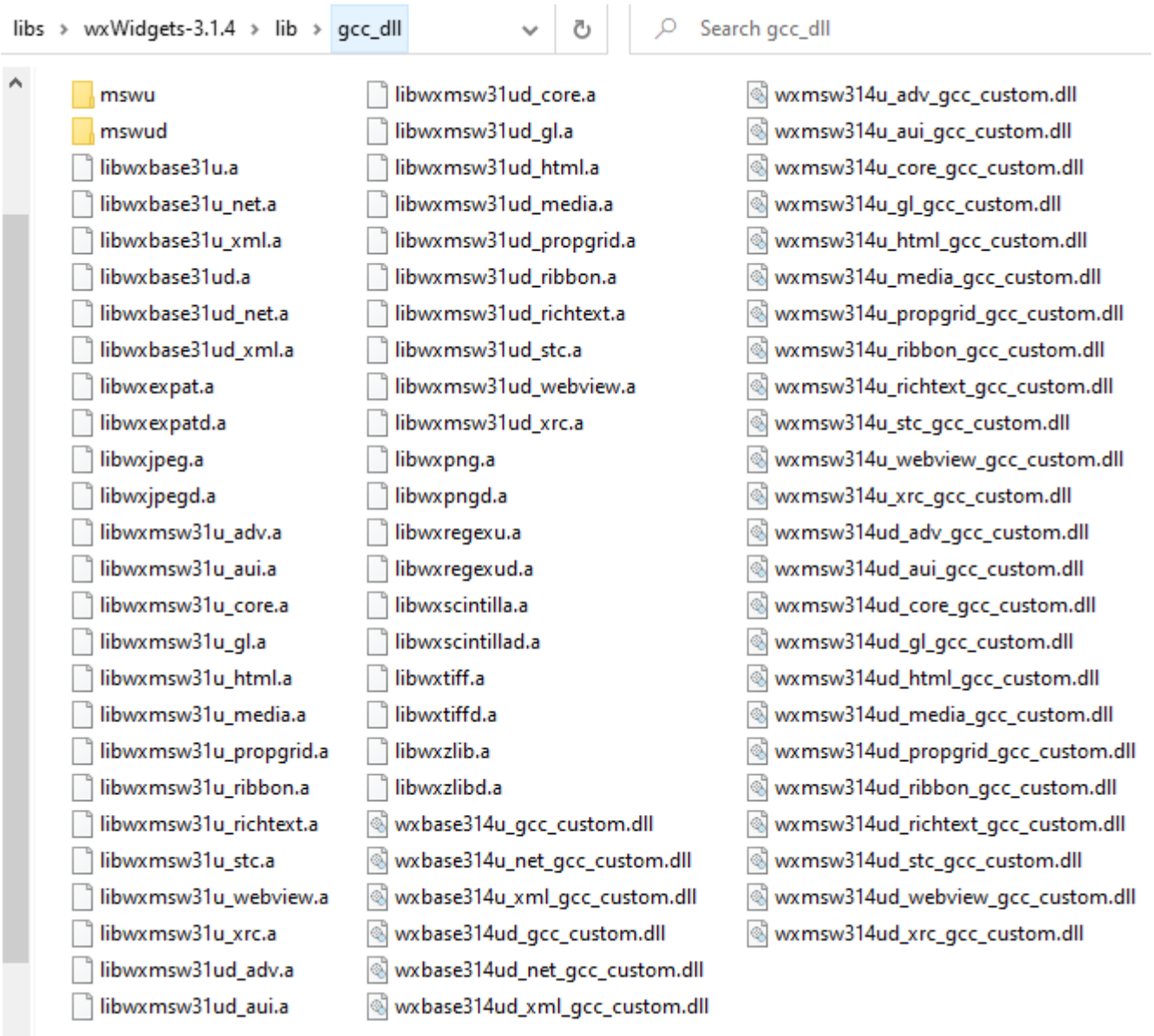
```
set PATH=c:\msys64\mingw32\bin;%PATH%

cd /d %WXWIN%\build\MSW


REM Build wxWidgets Shared Debug configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" setup_h
IF %ERRORLEVEL% NEQ 0 goto FAIL
mingw32-make -j4 -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17"
IF %ERRORLEVEL% NEQ 0 goto FAIL

REM Build wxWidgets Shared Release configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=release CXXFLAGS="-std=c++17" setup_h
IF %ERRORLEVEL% NEQ 0 goto FAIL
mingw32-make -j4 -f makefile.gcc SHARED=1 BUILD=release CXXFLAGS="-std=c++17"
IF %ERRORLEVEL% NEQ 0 goto FAIL


goto SUCCESS

:FAIL
echo Build failed!
pause
goto FINISHED

:SUCCESS

:FINISHED
```

3.6.4 Viewing Build Results

Now run the build commands from Listing 3-3 and wait till the build finishes. If it finished successfully, the command line window should be closed (if not, it will stay open with "Build failed! Press any key to continue..." message at its bottom and the actual error message above it). The libraries for the shared build will be by default generated in *WXDIR\lib\gcc_dll* where 2 folders and 74 files should be present, see Figure 3-6.

Note. Files for the static build are by default created in *WXDIR\lib\gcc_lib*.

Figure 3-6 Content of *WXDIR*\lib\gcc_dll folder



Files starting with *lib* and having an extension *.a* are import libraries used by the linker when building the application while files with an extension *.dll* are dynamic libraries needed by Windows to run the application (e.g., *libwxbase31u.a* vs *wxbase314u_gcc_custom.dll*). You can differentiate files for the Debug and Release configurations as the Debug ones have an extra *d* in their name compared to the Release (e.g., *libwxbase31ud.a* vs *libwxbase31u.a* and *wxbase314ud_gcc_custom.dll* vs *wxbase314u_gcc_custom.dll*, or *libwxzlibd.a* vs *libwxzlib.a*). Remember that you need to ship all required DLLs (compiler's , wxWidgets', and possibly others) with your application and let the installer put them in the same folder where the application executable is (but you can avoid this, see chapter 3.6.7).

The two folders are build-specific (*mswu* for the Release and *mswud* for the Debug build) and contain build-specific *setup.h*. They need to be added to the compiler include directories, in addition to the build-agnostic main include folder (*WXDIR\include)*, see chapter 4.8.1 for more information.

3.6.5 Using the Minimal Sample to Test the Build

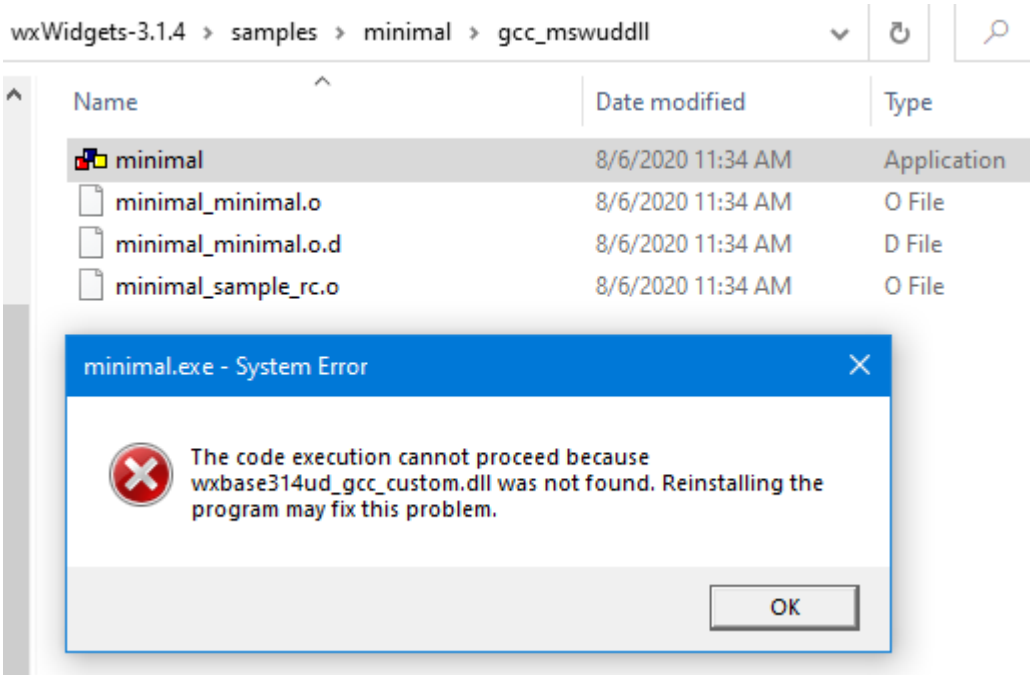
Now let us test our wxWidgets build with the bundled Minimal sample. Go to *WXDIR\samples\minimal* folder and run the commands shown in Listing 3-4.

Listing 3-4 Build commands for the minimal sample (available as *build-sample-minimal-shared-debug.bat*)

```
set PATH=c:\msys64\mingw32\bin;%PATH%
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17"
```

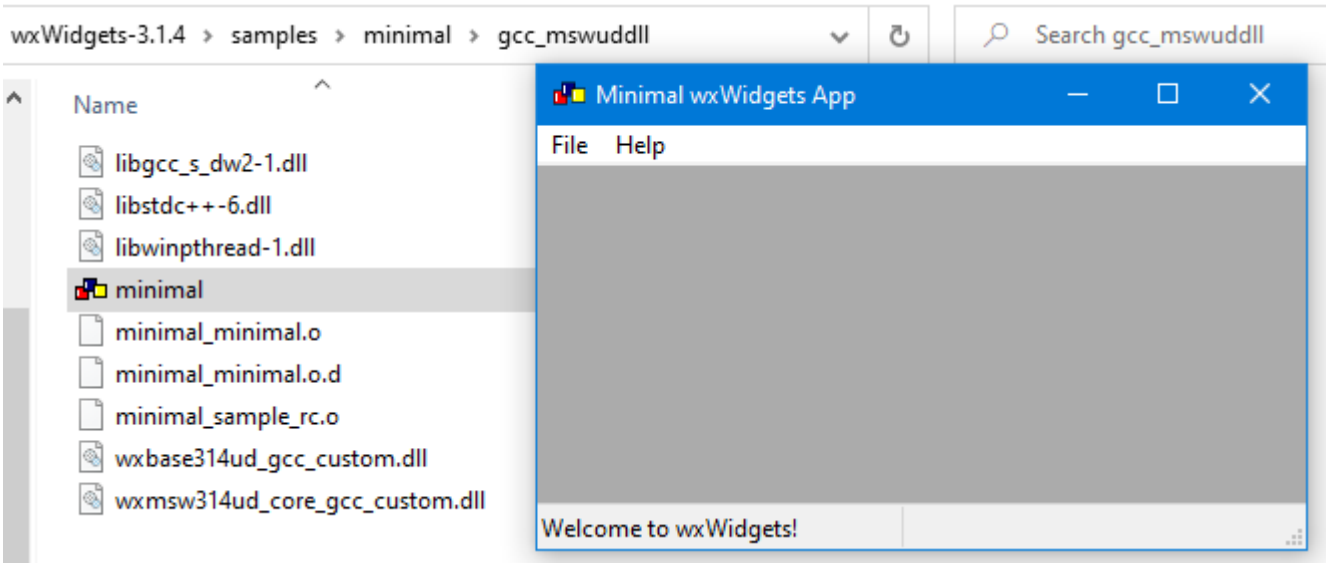
This should build the Debug configuration of the Minimal sample. Please notice that the build parameters must be the same as when building wxWidgets itself. Assuming the build succeeded, you should have file *minimal.exe* in the *gcc_mswuddll* folder of the Minimal sample folder. However, if you just try to launch the sample now, instead of the expected Minimal sample window you will probably end up seeing something like in Figure 3-7, with several error messages about missing DLLs popping one after another.

Figure 3-7 Launching minimal sample with missing DLLs



What happened? The application executable needs both GCC and wxWidgets DLLs. The recommended way is to put those in the same folder as the executable. So, you need to copy all the required DLLs to the folder where *minimal.exe* is. Firstly, from the MinGW bin folder such as *c:\msys64\mingw32\bin* copy *libgcc_s_dw2-1.dll*, *libstdc++-6.dll*, and *libwinpthread-1.dll*. Secondly, copy from folder *WXDIR\lib\gcc_dll* files *wxbase314ud_gcc_custom.dll* and *wxmsw314ud_core_gcc_custom.dll*. Please notice that you may need other wxWidgets DLLs for applications using more wxWidgets features (and release versions of the DLLs for the release build of an application). Now run *minimal.exe* again and you should see what is in Figure 3-8.

Figure 3-8 Launching minimal sample successfully



3.6.6 Cleaning the Build

When changing build parameters for the same configuration it is best to make clean build, to make sure there are no leftovers from the previous build. All you need is to run the same build commands as when building but with **`clean`** target added, see Listing 3-5.

Listing 3-5 Example of cleaning the build

```
set PATH=c:\msys64\mingw32\bin;%PATH%
cd /d %WXWIN%\build\MSW

REM First clean and then build wxWidgets Shared Debug configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" clean
IF %ERRORLEVEL% NEQ 0 goto FAIL
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17"
IF %ERRORLEVEL% NEQ 0 goto FAIL

goto SUCCESS

:FAIL
echo Build failed!
pause
goto FINISHED

:SUCCESS

:FINISHED
```

To be honest, what I usually do is just delete the *gcc_** folders in *WXDIR\build\msw* and *WXDIR\lib*.

3.6.7 Building wxWidgets Statically

It is possible to build wxWidgets so that at run-time your application does not need any wxWidgets or compiler DLLs. To create such static build, we need to use **SHARED=0** instead of **SHARED=1** we used for dynamic build. To link the compiler libraries statically we will use **-static** flag for the linker. Listing 3-6 contains all the commands, with the differences to the default build described in Listing 3-2 in red.

Listing 3-6 Build commands for static wxWidgets build (available as *build-library-static.bat*)

```
set PATH=c:\msys64\mingw32\bin;%PATH%

cd /d %WXWIN%\build\MSW


REM Build wxWidgets Static Debug configuration with C++17 support and GCC libraries linked statically
mingw32-make -f makefile.gcc SHARED=0 BUILD=debug CXXFLAGS="-std=c++17" LDFLAGS="-static"
IF %ERRORLEVEL% NEQ 0 goto FAIL

REM Build wxWidgets Static Release configuration with C++17 support and GCC libraries linked statically
mingw32-make -f makefile.gcc SHARED=0 BUILD=release CXXFLAGS="-std=c++17" LDFLAGS="-static"
IF %ERRORLEVEL% NEQ 0 goto FAIL


goto SUCCESS

:FAIL
echo Build failed!
pause
goto FINISHED

:SUCCESS

:FINISHED
```

The static wxWidgets libraries will be by default generated in the *gcc_lib* folder of *WXDIR/lib* folder. If the build succeeded, the content of the folder should look similar to that in Figure 3-6, except there will be no DLL files and the libraries (*.a files) will be much larger.

Note. Batch file for static parallel build is available as *build-library-static-parallel.bat*.

3.6.8 More Build Options

There are many more wxWidgets options that affect the build. Some of them can be set in *WXDIR/include/msw/setup.h*, see the comments in that file and/or in the official documentation. Others are set through arguments to make, see their description in the official documentation. Discussing them is beyond the scope of this guide.

Note. Code::Blocks wxWidgets project wizard supports custom **CFG** (see chapter 4.5.9) but does not support custom **COMPILER_PREFIX**.

4 Setting Up wxWidgets Project with Code::Blocks and MinGW

4.1 Installing Code::Blocks

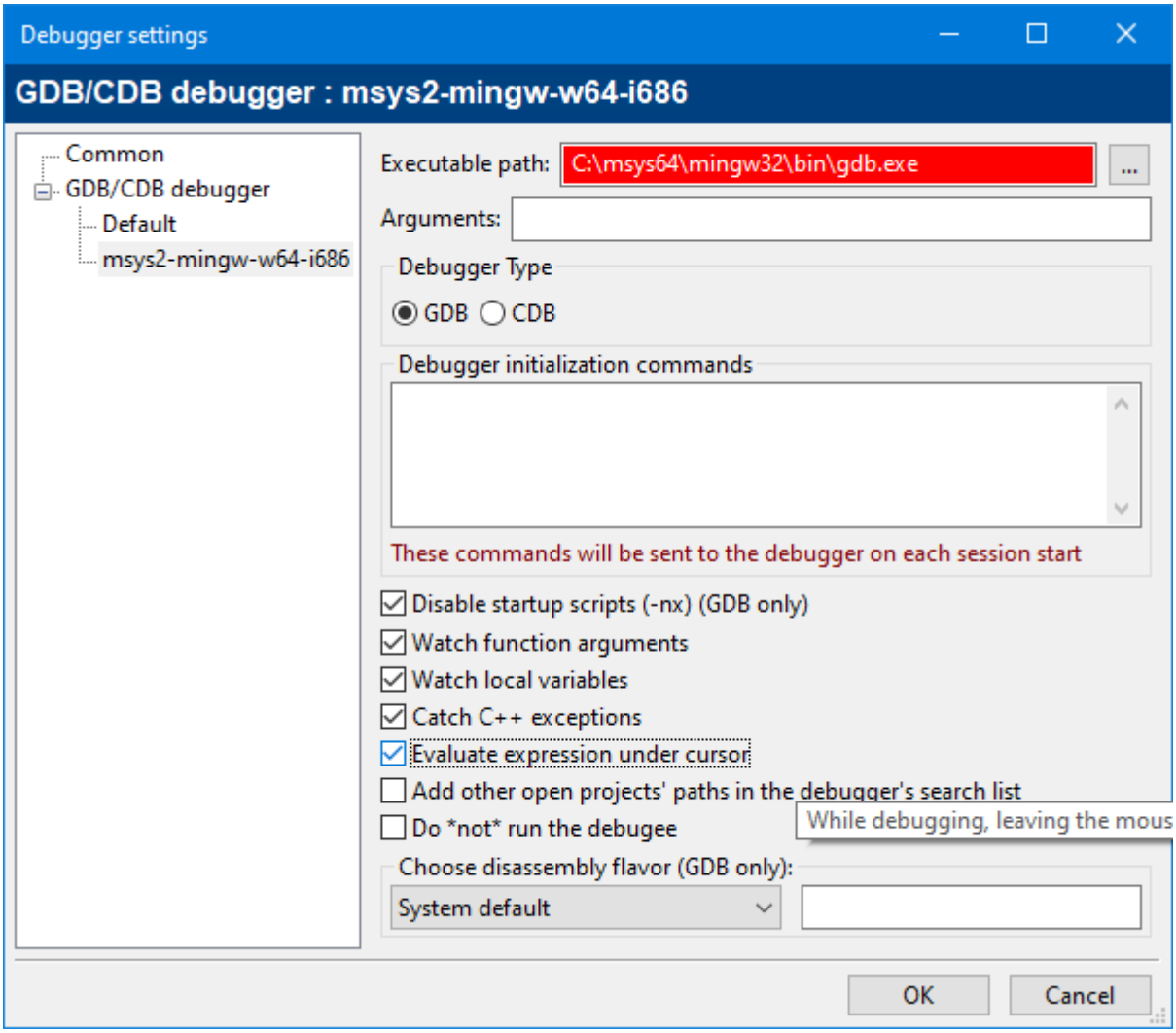
This guide uses Code::Blocks v20.03, which is its latest version available at the time of writing. Go to https://www.codeblocks.org/downloads, click **Download the binary release** and download **codeblocks-20.03-setup.exe**. This is the installer without a compiler bundled, as we are going to use MinGW we installed in chapter 3.2. Launch the installer and install Code::Blocks with the default settings. Now launch Code::Blocks for the first time. It should display a dialog stating it could not auto-detect any (GCC-based) compilers. This is actually a good thing as it means there is no GCC-based compiler toolchain in the path or a known location where it could interfere with our own MinGW installation. Close the dialog with **OK** and continue. Select your choice in the next dialog asking whether to use Code::Blocks as the default application for C++ files and you will end up in the Code::Blocks IDE.

Now we have to tell Code::Blocks that it should use the MinGW compiler toolchain we installed in chapter 3.2 of this guide. Somewhat counter-intuitively, we need to set up the debugger before compiler, because we will use the new debugger configuration in the compiler settings.

4.2 Configuring the Debugger

From the main menu choose **Settings** / **Debugger...**, in the left part of the **Debugger settings** dialog select **GDB/CDB debugger** and click the **Create Config** button. Type **`msys2-mingw-w64-i686`** as the name for the new debugger configuration. Select the newly created debugger configuration and in the **Executable path** set the path to GDB we are using (e.g., *c:\msys64\mingw32\bin\gdb.exe*), see Figure 4-1. Leave all other options at their defaults, perhaps except **Evaluate expressions under cursor**, and close the dialog with the **OK** button.

Figure 4-1 Setting the debugger in Code::Blocks

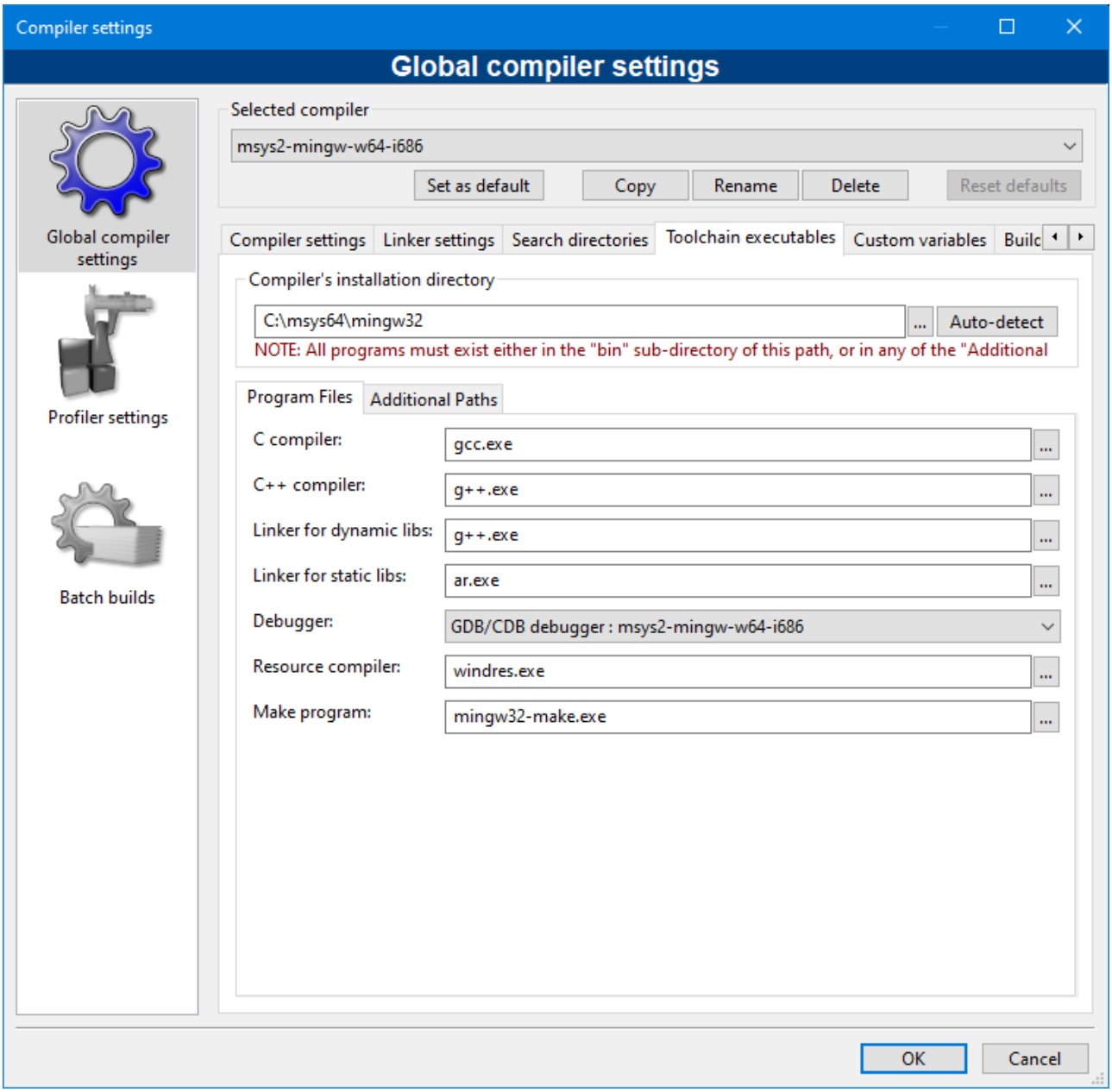


4.3 Configuring the Compiler

From the main menu choose **Settings** / **Compiler…** and in the **Compiler settings** dialog go to page **Global compiler settings**. We will not modify the existing **GNU GCC Compiler** but we will use the **Copy** button to copy **GNU GCC Compiler** to a new compiler we will name after the toolchain we are using **`msys2-mingw-w64-i686`**. Creating a new compiler here allows us to switch between different compilers when needed.

Select the newly added compiler **msys2-mingw-w64-i686**, go to the **Toolchain executables** tab, press the button with three dots next to the field for **Compiler's installation directory** and select the folder where Code::Blocks will look for the build tools. Well, it will actually not look for the tools' executables in this folder – it will use its *bin* subfolder – so for our MinGW installation we set the directory to *c:\msys64\mingw32* (or wherever did you install MSYS2 on your PC). Make sure the file names for the individual tools match those in the ones shown in Figure 4-2 and set **Debugger:** to the configuration we created in chapter 4.2. Now press the **Set as Default** button, confirm the message about the new compiler being default for new projects and close the dialog with the **OK** button.

Figure 4-2 Setting the compiler in Code::Blocks



We will not set any other global compiler settings; if needed, we will specify them in the project settings instead. This may be a bit more work per project, but it allows flexibility not possible with setting them globally. You certainly do not want to add any library specific stuff here (such as search directories for a compiler or linker), these undoubtedly belong to the project settings. If you added something there earlier, please remove it, as it could interfere with our build later.

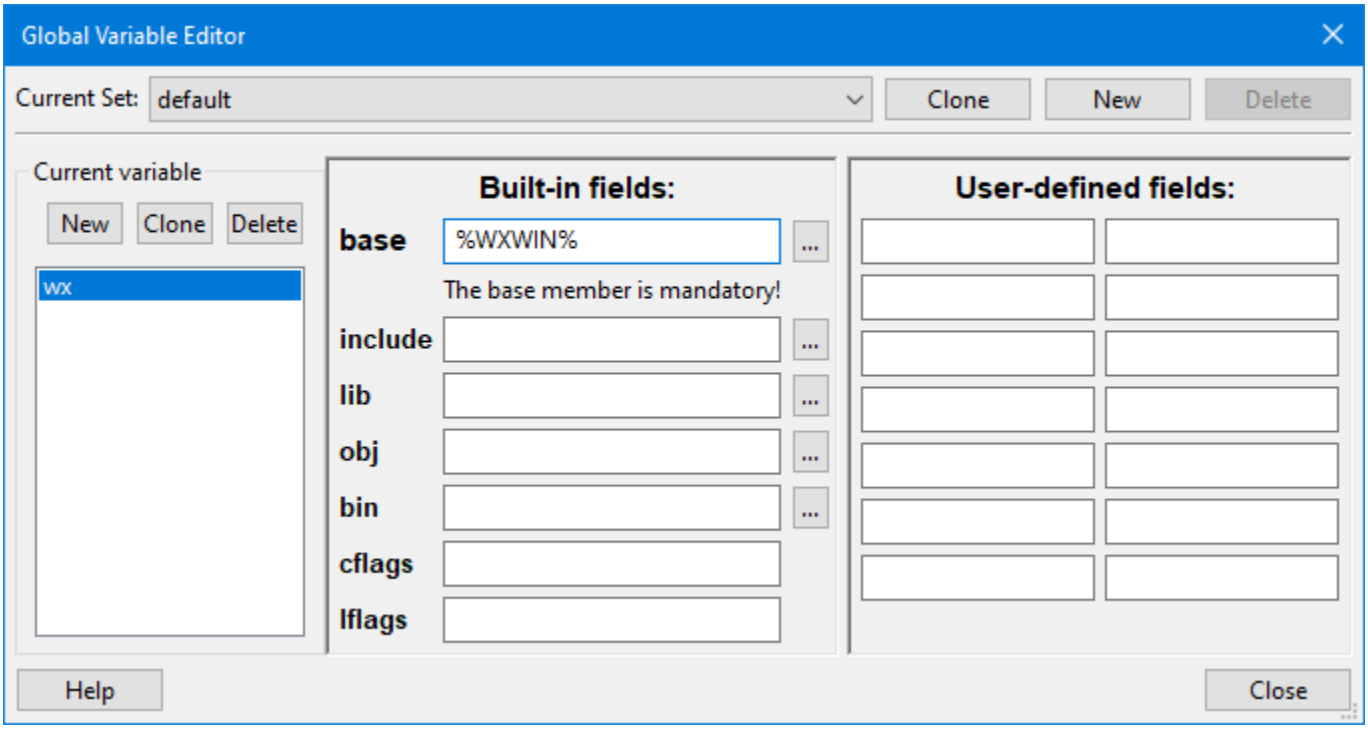
Note. If Code::Blocks cannot find a tool it needs, it complains about it, but just shows a pop-up in the bottom right corner of the screen, which could be easy to miss on a bigger display. Make sure there is no such pop-up, if there is, make sure the **Compiler's installation directory** is correct and the file names for the individual tools are the same as in Figure 4-2. However, it seems that sometimes Code::Blocks gets confused when creating a new compiler and complains even when everything is set correctly, either after closing the Compiler Settings dialog or after restarting it. It seems to fix itself after closing and restarting Code::Blocks again.

4.4 Creating Global Variable *wx*

As we do not want to use hard-coded paths in our projects, which would make difficult to transfer the project between computers with different folder hierarchies, we will not use a hard-coded path to wxWidgets. We will use a Code::Blocks global variable named *wx*. This variable will use system environment variable *WXWIN* we created in chapter 3.5.

From the main menu choose **Settings** / **Global variables...** and in the **Global Variable Editor** dialog use the **New** button (below **Current variable**) to create variable named **`wx`**. In the **Built-in-fields** set **base** to **`%WXWIN%`** (Figure 4-3). Please notice that there are percent signs around *WXWIN*, these means that this is not a filesystem path but a system environment variable that will be expanded to an actual path. Leave all other fields empty and use the **Close** button to close the dialog.

Figure 4-3 Creating global variable *wx*



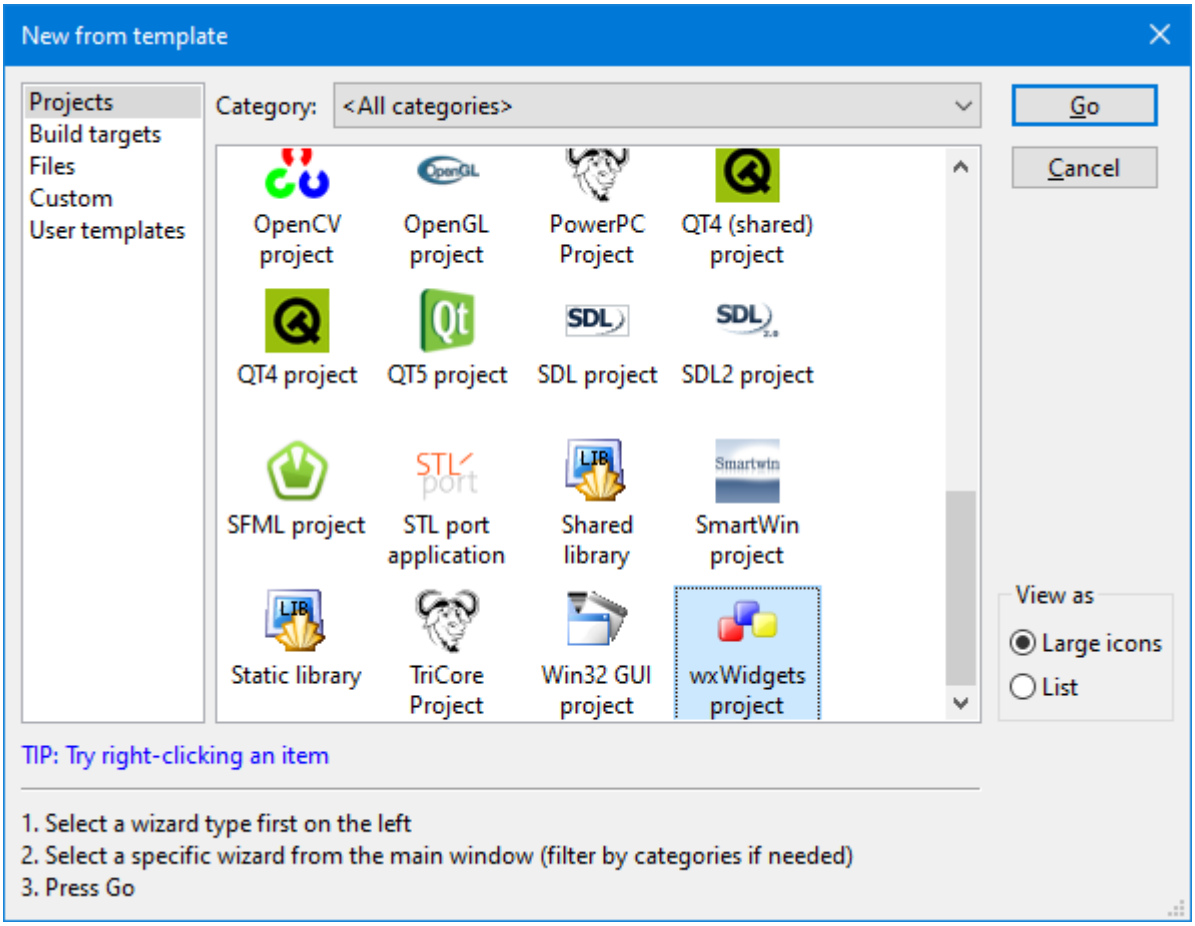
4.5 Creating a wxWidgets Project with Project Wizard

4.5.1 Selecting wxWidgets Project Template

Now let us create our first wxWidgets project, using a Code::Blocks template wizard. The procedure shown in this chapter assumes you built both Debug and Release Shared builds of wxWidgets (see chapter 3.6) and installed and configured Code::Blocks as shown chapters 4.2 and 4.4.

This wizard has quite a few steps, but each individual step is easy and simple. From the main menu go to **File** / **New** / **Project...** and in the **New from template** dialog scroll down to the bottom of the project list, select **wxWidgets project** (see Figure 4-4) and press the **Go** button.

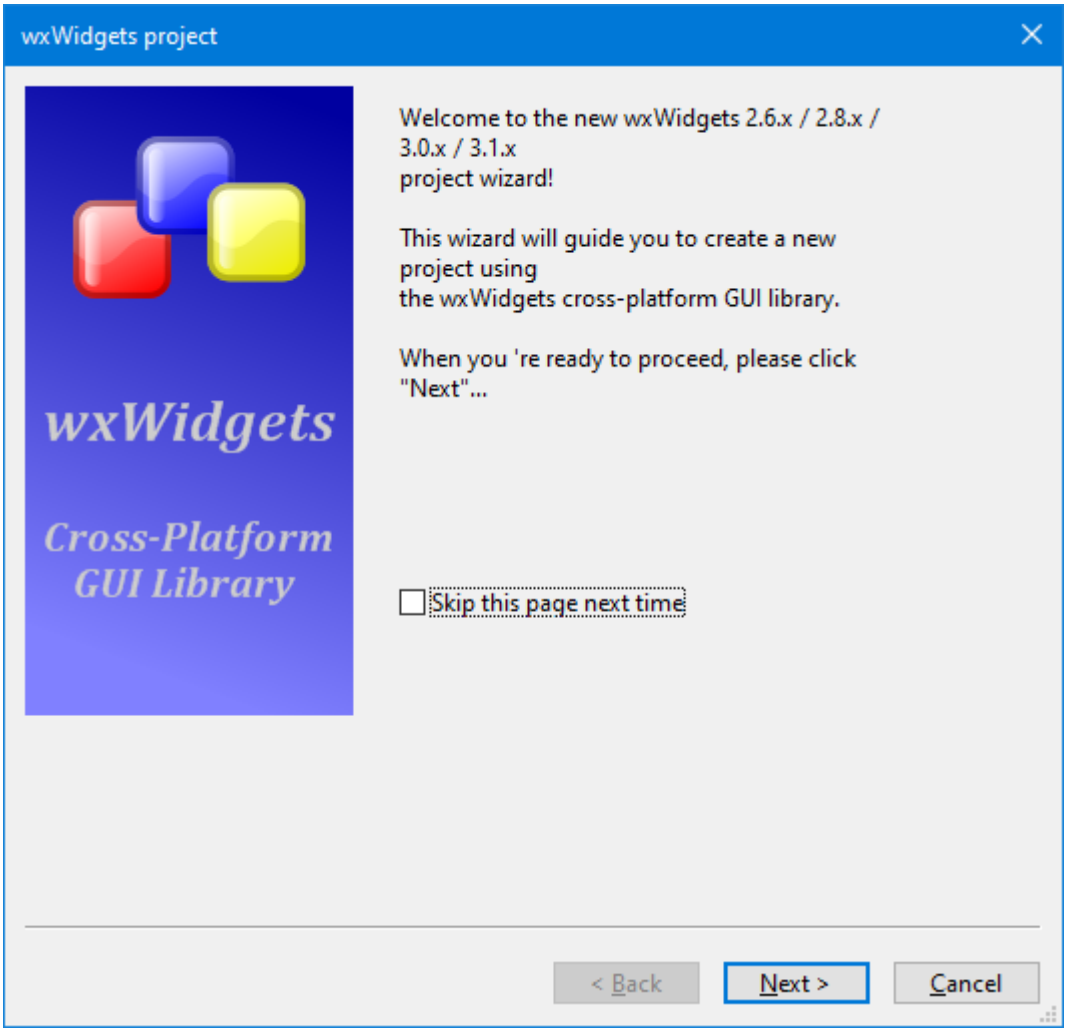
Figure 4-4 Selecting wxWidgets project template



4.5.2 Wizard Page 1: Welcome

The first wizard page (Figure 4-5) only welcomes you and has no settings so just press **Next**.

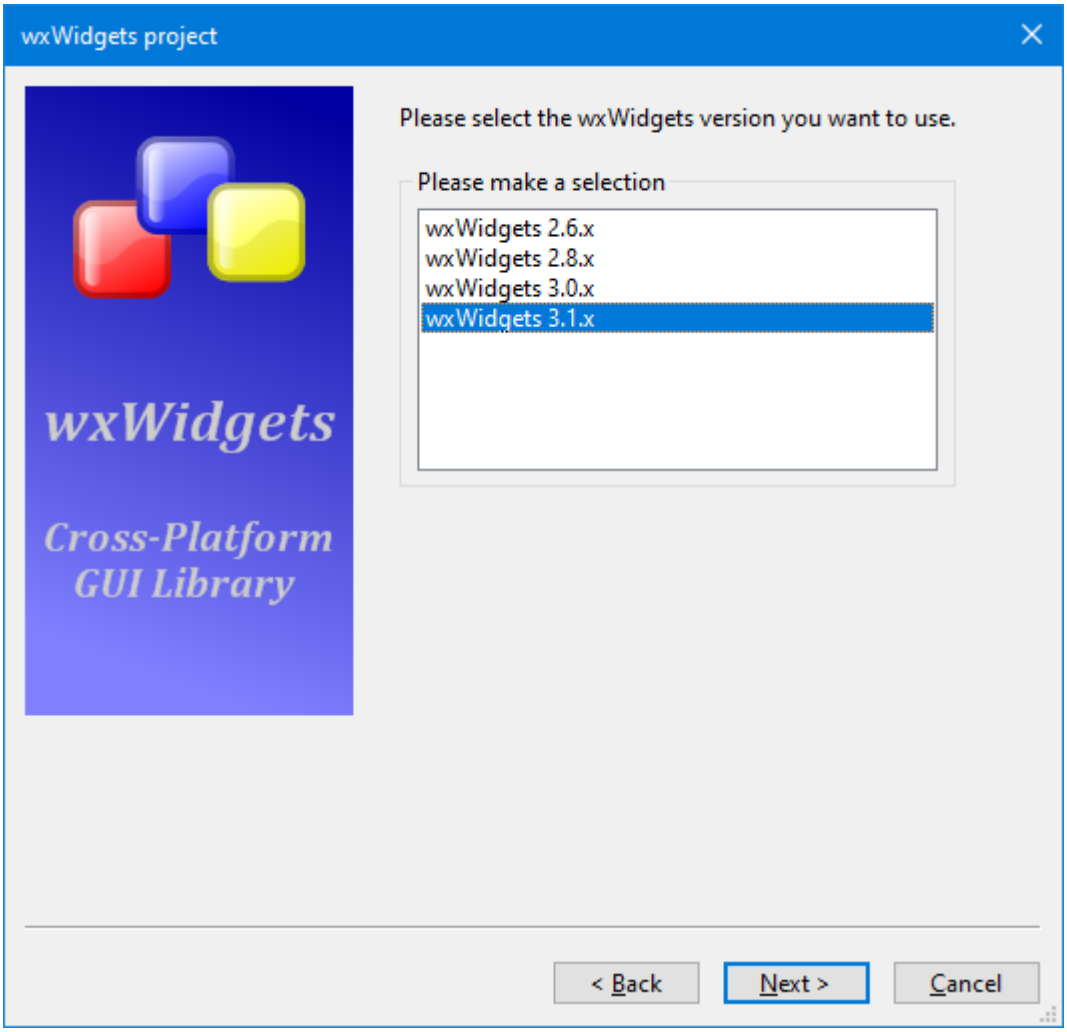
Figure 4-5 Welcome page



4.5.3 Wizard Page 2: wxWidgets Version

In this step we need to select wxWidgets version we are using: Select **wxWidgets 3.1.x** (Figure 4-6) and press **Next**.

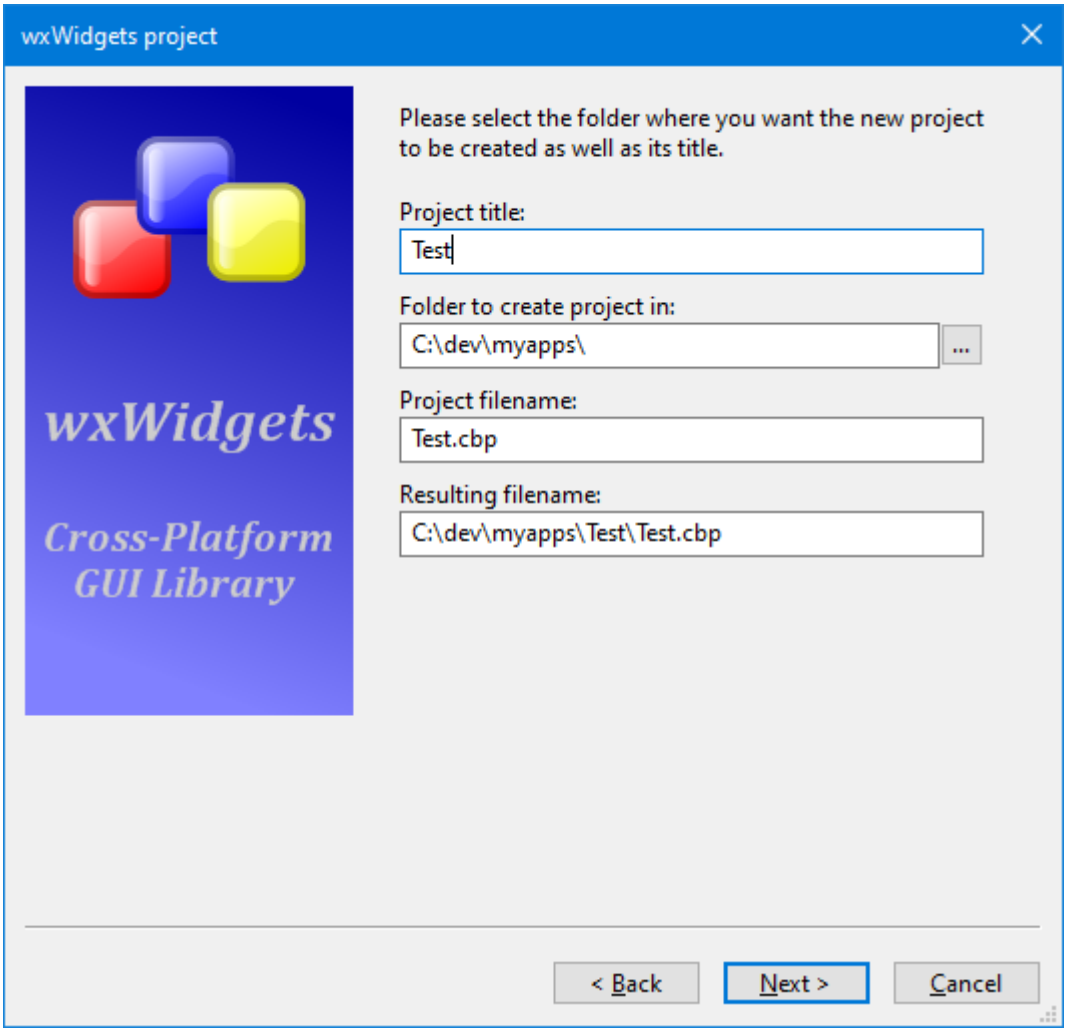
Figure 4-6 Selecting wxWidgets version



4.5.4 Wizard Page 3: Project Title and Paths

Fill the project title and the folder where to create the project (Figure 4-7), I strongly recommend against using a path with spaces in it, and press **Next**.

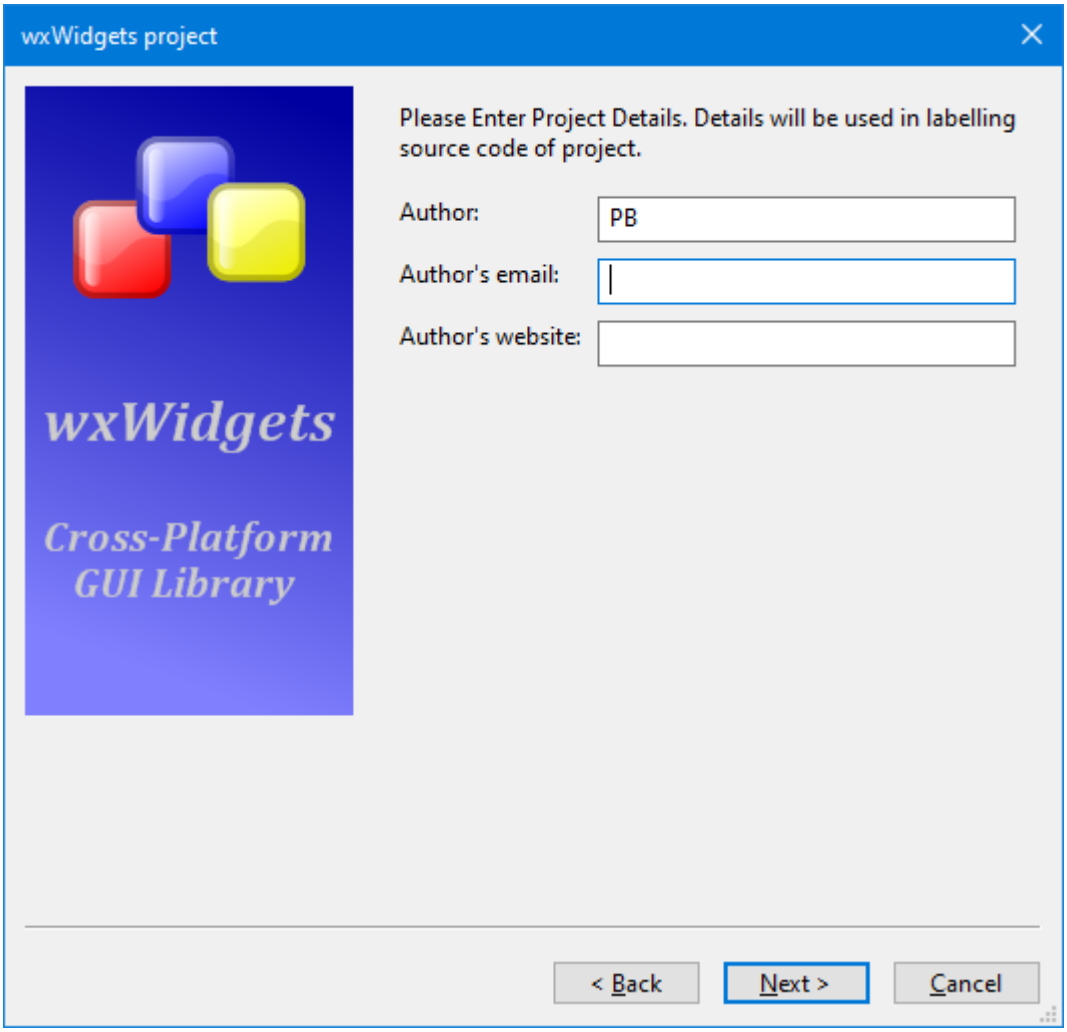
Figure 4-7 Setting project title and paths



4.5.5 Wizard Page 4: Project Author

Fill the information about the project's author (Figure 4-8) or leave it empty and press **Next** again.

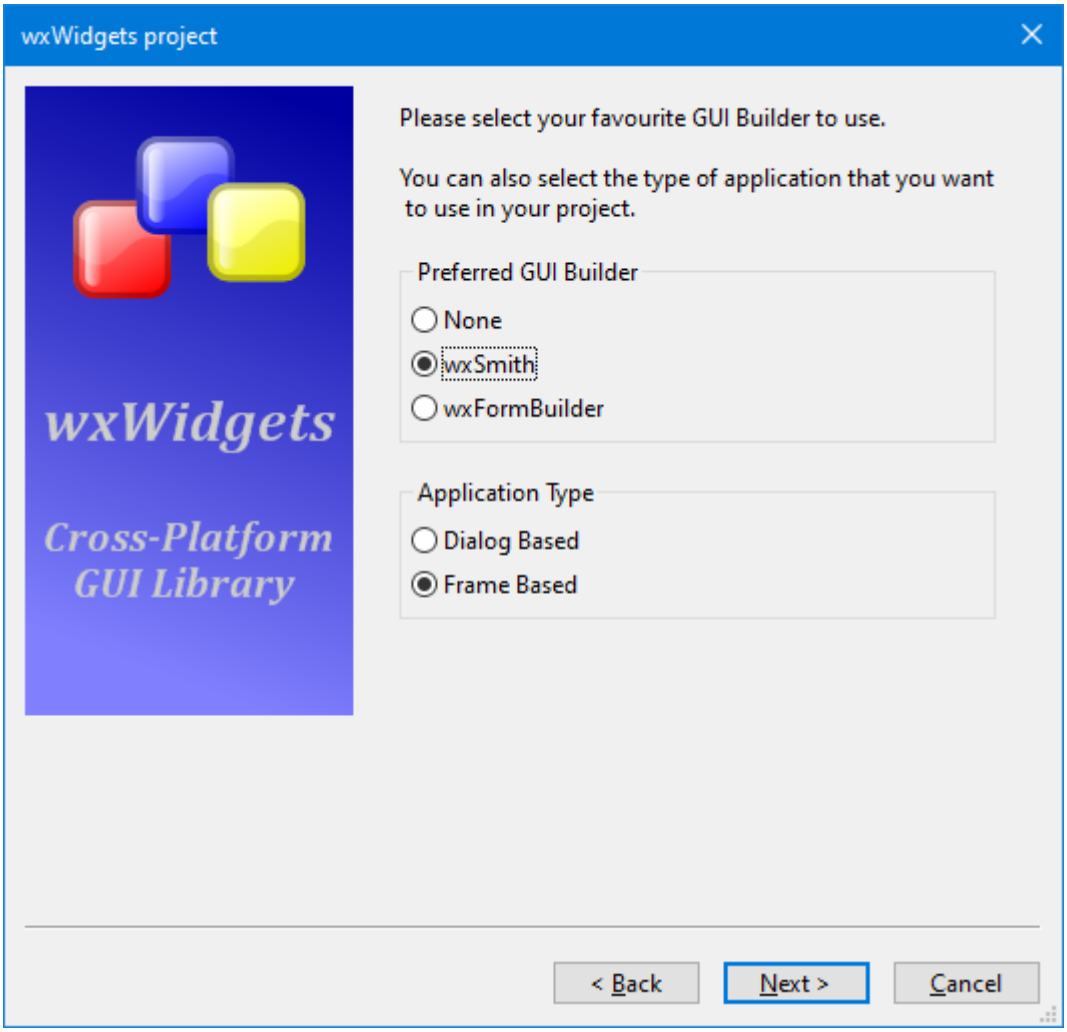
Figure 4-8 Setting Project Author



4.5.6 Wizard Page 5: GUI Builder and Application Type

Select your preferred GUI builder and application type (Figure 4-9), I chose **wxSmith** and **Frame Based**, and press **Next.**

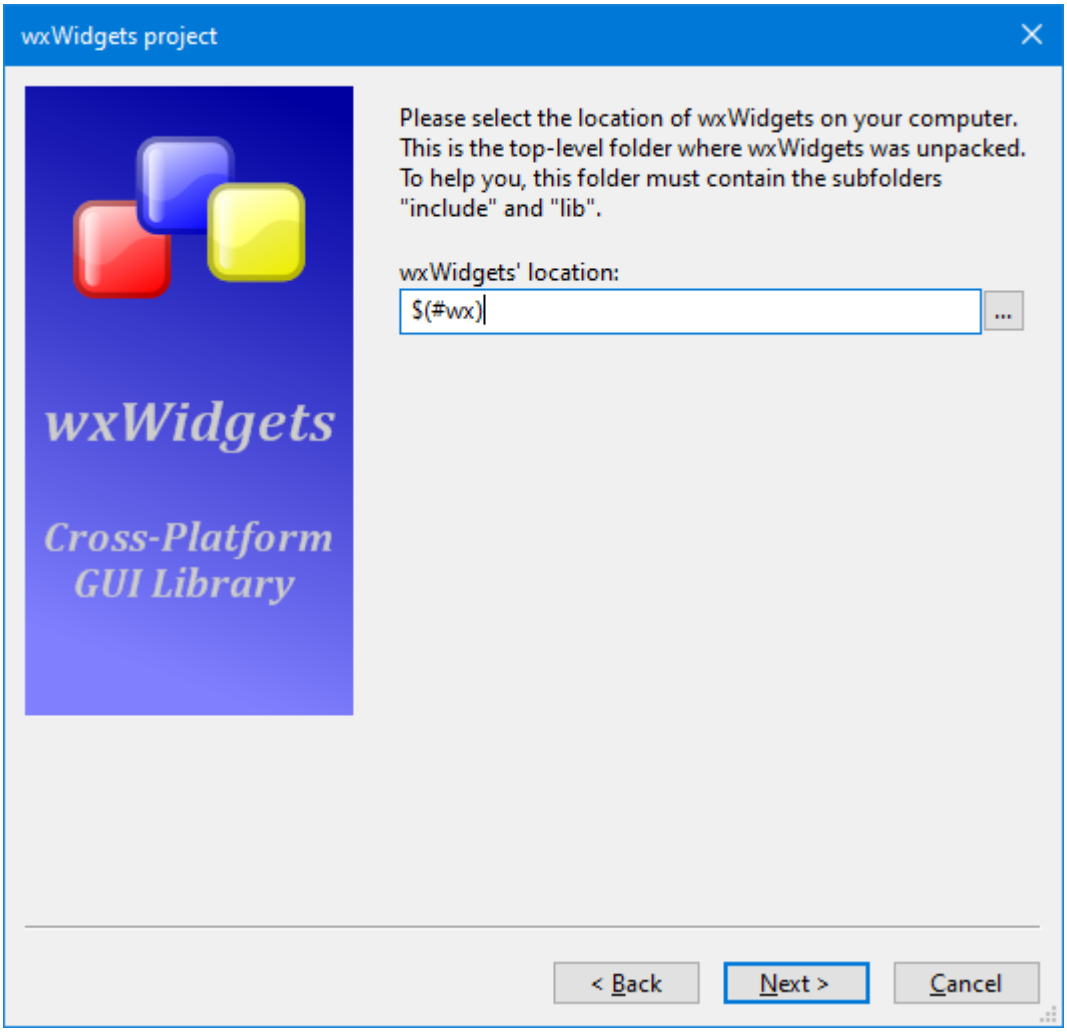
Figure 4-9 Selecting GUI Buillder and Application Type



4.5.7 Wizard Page 6: wxWidgets Location

Now we are asked to enter wxWidgets location (Figure 4-10). Instead of using a hard-coded path such as *c:\dev\libs\wxWidgets-3.1.4*, we will set it to global variable *wx* we created in chapter 4.4. Set **wxWidgets' location** to `$(#wx)` and press **Next**.

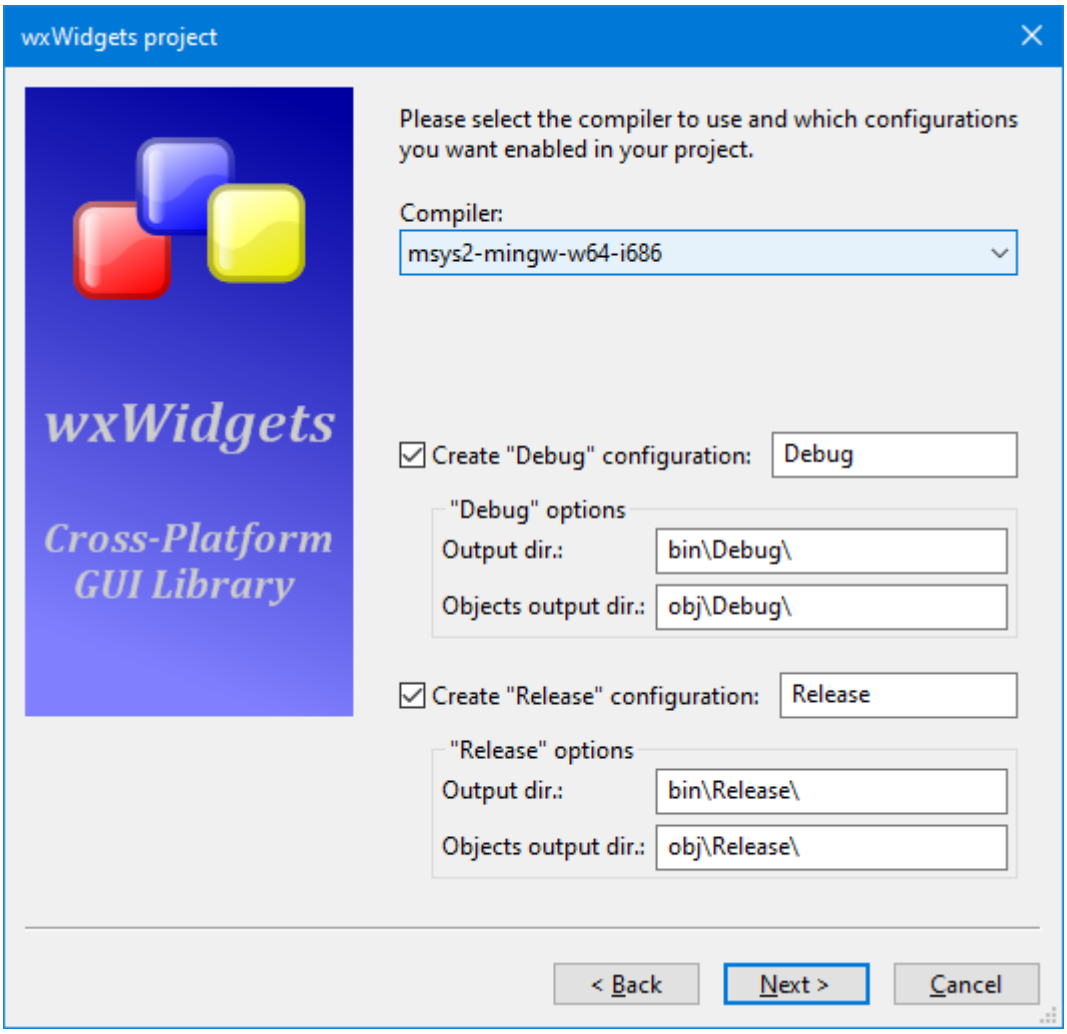
Figure 4-10 Setting wxWidgets location



4.5.8 Wizard Page 7: Compiler and Debug/Release Configurations

On this page, make sure that **Compiler** contains **msys2-mingw-w64-i686** (see chapter 4.2) and that you have checked both **Create "Debug" configuration** and **Create "Release" configuration** (Figure 4-11) Leave the rest of fields at their defaults and press **Next**.

Figure 4-11 Selecting Compiler and setting Debug/Release configurations



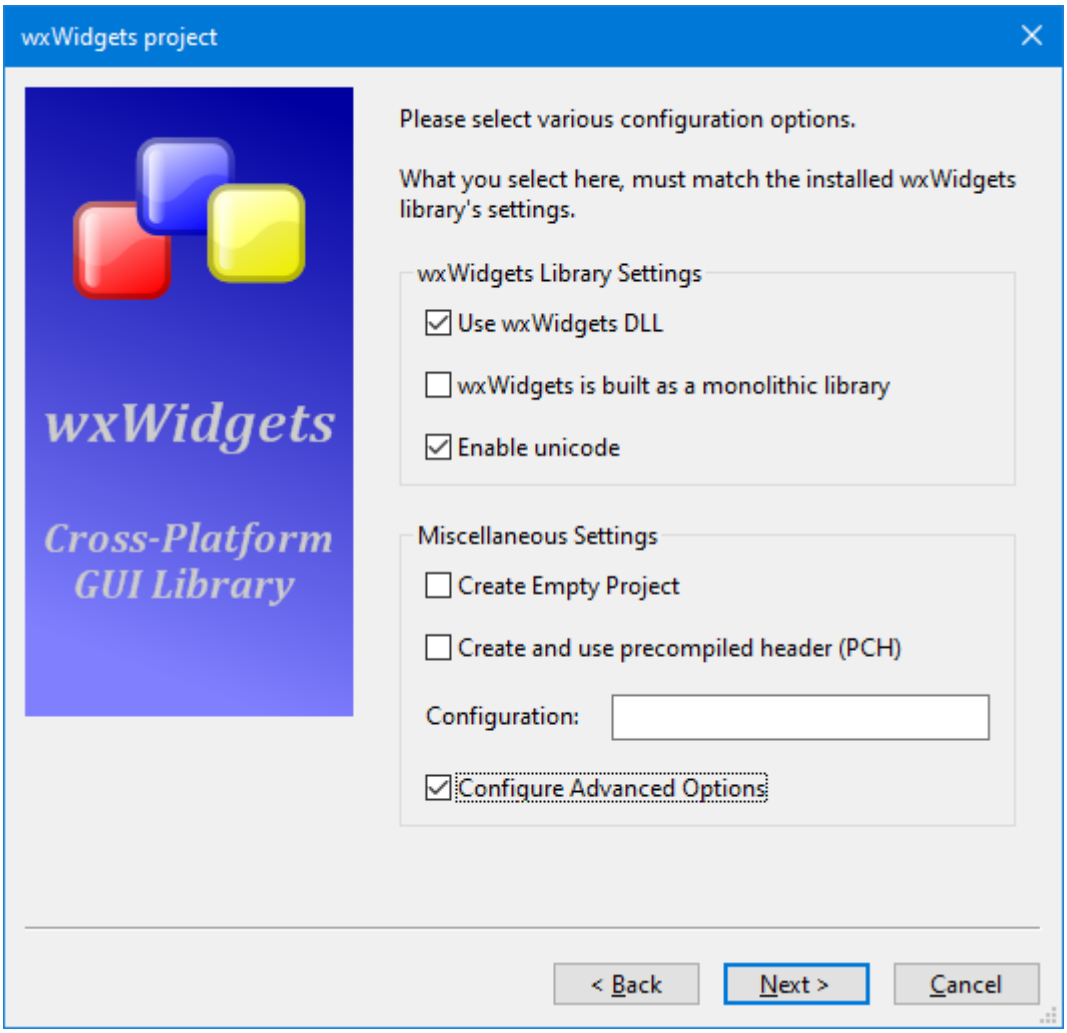
4.5.9 Wizard Page 8: Various Configuration Options

This page lets you set other configuration related options (Figure 4-12), check **Use wxWidgets DLL**, **Enable unicode**, and **Configure Advanced Options.** Make sure that **wxWidgets DLL is built as a monolithic library** is **not** checked (unless you have a monolithic wxWidgets build) and press **Next**.

If you see error message such as "A matching Debug configuration cannot be found ....", it means you did something wrong. You either did not build the configurations you asked for (Debug and/or Release), did not build it in requested configuration (e.g., you checked on **Use wxWidgets DLL** here but you built only the static wxWidgets build, see chapters 2.2 and 3.6.7) or you did not set the *wx* variable so that it points to your wxWidgets folder (see chapter 4.4). If you built wxWidgets using a custom configuration (with `CFG=SomeString`), you need to enter that `SomeString` into **Configuration**, so that Code::Blocks can find your custom build. Please check which it is and correct the issue.

Note. If you want to use the static build of wxWidgets, do not check **Use wxWidgets DLL** (see also chapter 4.6.3).

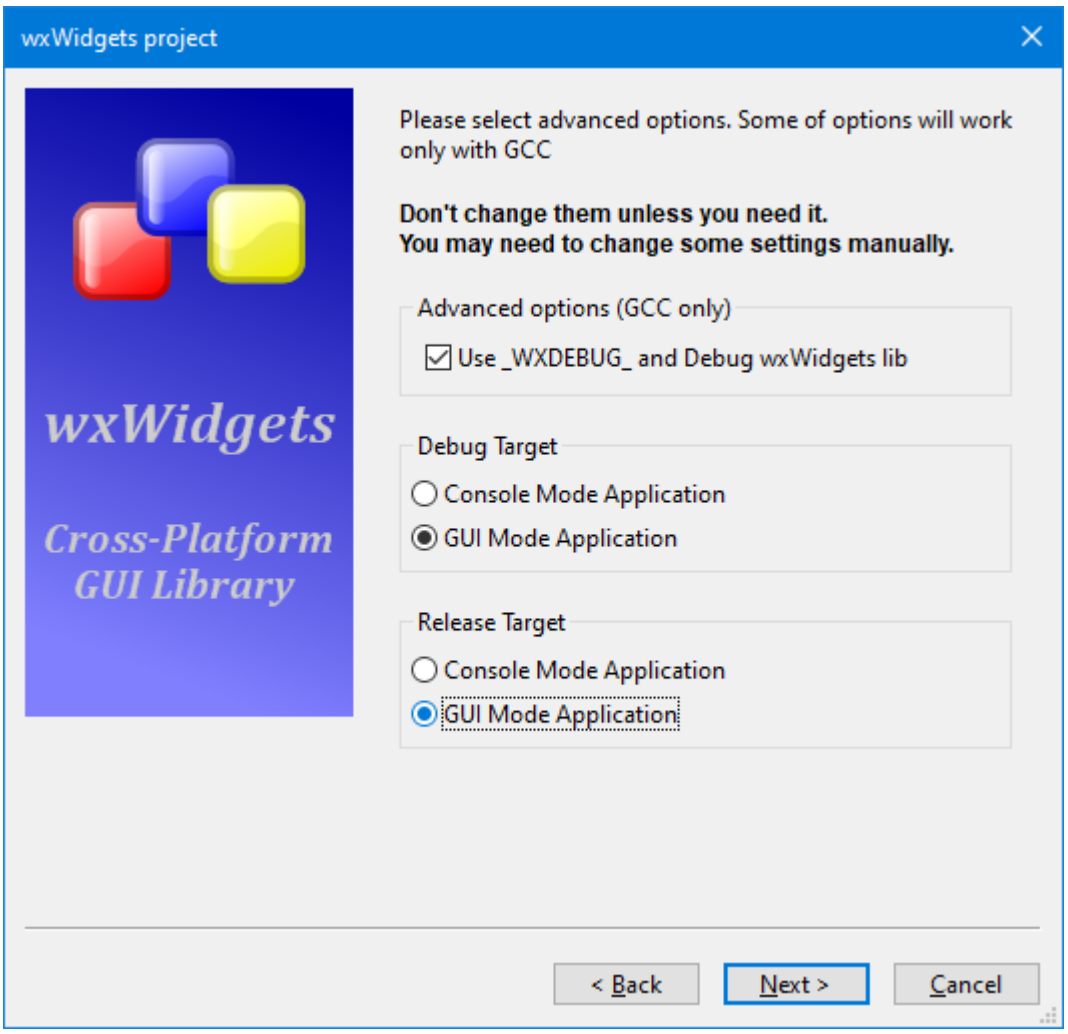
Figure 4-12 Setting various configuration options



4.5.10 Wizard Page 9: Advanced Options

On this page check **Use _WXDEBUG_ and Debug wxWidgets lib**, select **GUI Mode Application** for both **Debug Target** and **Release Target** (Figure 4-13) and press **Next**.

Figure 4-13 Setting advanced options



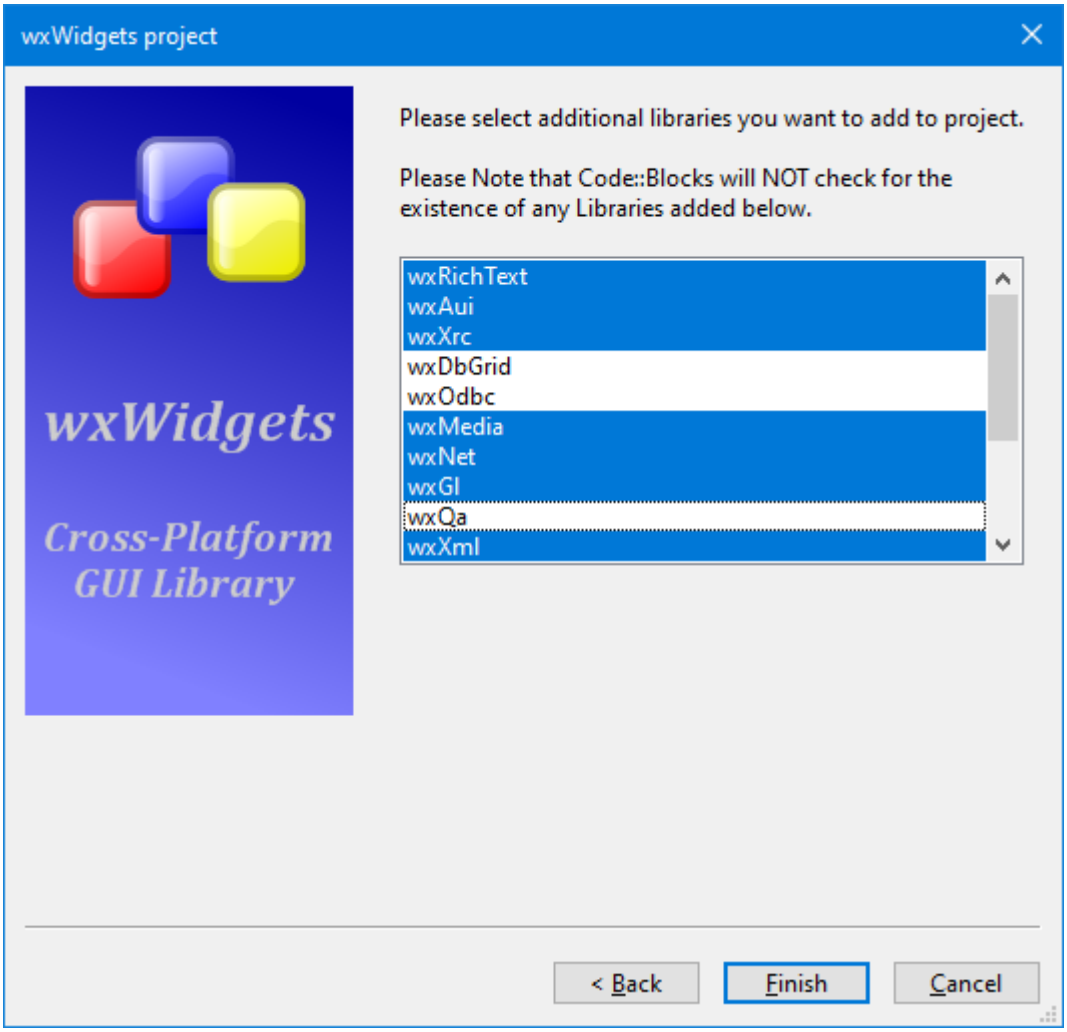
4.5.11 Wizard Page 10: Additional Libraries

We are on the last page of the project wizard and all that is left is to select additional wxWidgets libraries. For simplicity sake select them all now (Figure 4-14, including those not shown in the figure) except for **wxDbGrid** and **wxOdbc** (these two existed only in old versions of wxWidgets) and **wxQa** (which is only for Microsoft Visual C++).

You are probably not going to need some of these additional libraries in your actual project. If you want to know more about using wxWidgets libraries in a Code::Blocks project, see chapter 4.8.3.

Press **Finish** to complete the project wizard.

Figure 4-14 Selecting additional libraries



4.6 Adjusting Project Settings

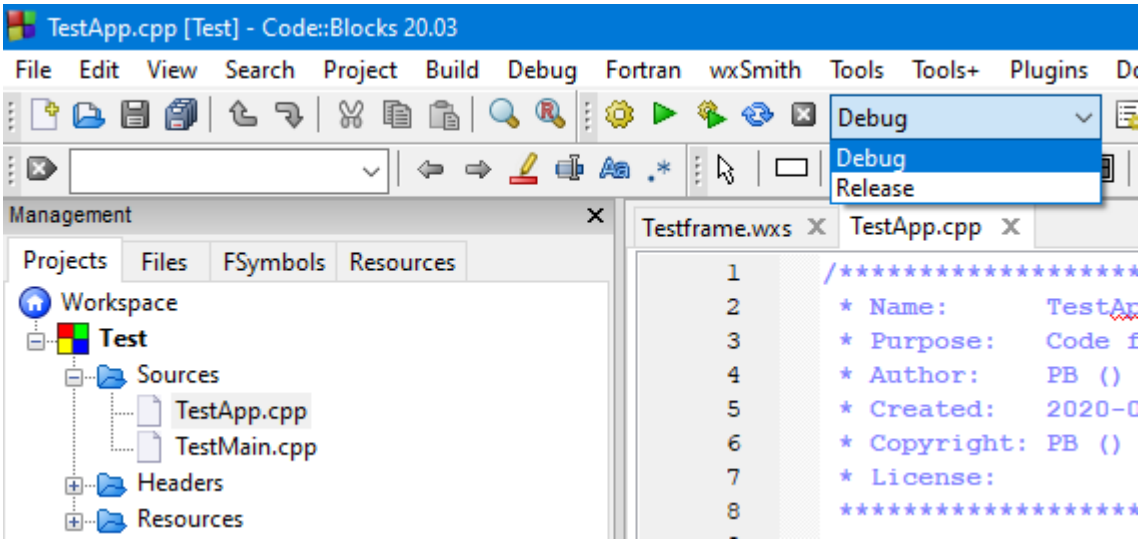
After finishing the project wizard, we now have a basic Code::Blocks project. There is some boilerplate C++ code, basic resource file, the paths for C++ and resource compilers as well as for linker libraries are set, and the required wxWidgets and Windows libraries are added.

Our project was created with two build targets: Debug and Release (see chapter 4.5.8). The Debug target serves for the developer, using debug versions of the libraries. The debug libraries may be slower and take more memory than release ones, but they contain very useful run-time checks that can save enormous amount of time, effort, and frustration when dealing with bugs. The executable produced in the Release target is the one to be distributed to the end users. If you want to know more, see chapter 4.8 for more details about the project settings.

You can switch between the targets using the toolbar (Figure 4-15) or menu **Build** / **Select target**. The build commands (**Build**, **Run**, **Build and run**, …) apply to the currently selected build target. You will probably mostly use Debug target during development, but when you want to build the Release one, you need to switch to it. Unlike for example Microsoft Visual Studio, Code::Blocks does not have “Batch build” command that allows to build multiple targets at once.

Some project settings are common for all build targets, some are specific for a build target. Before we build our first application, we need to adjust some of them.

Figure 4-15 Switching the build target using toolbar

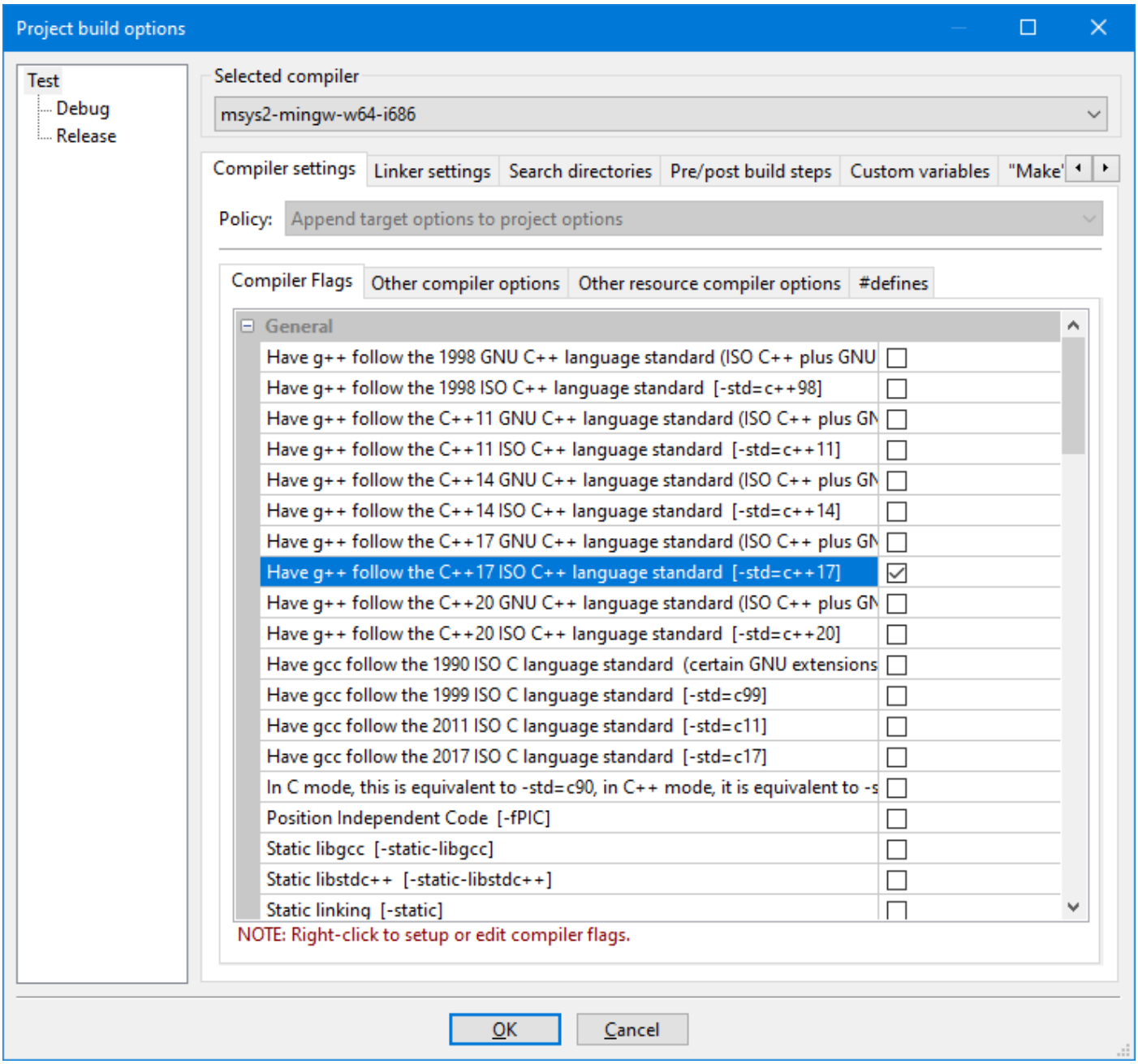


4.6.1 Setting C++ Standard Used in the Project

From the main menu choose **Project** / **Build Options** which will show the Project build options dialog. Please notice that in the left part of the dialog is a tree control, where the root item is the project name (**Test** in our project) and its children are individual build targets (**Debug** and **Release**). Options set when the root item is selected apply to all build targets, but they are not shown in the individual target settings.

We built wxWidgets with C++17 standard, so we must tell the compiler that our project also uses it. Some people set this in the Global Compiler Settings, but I prefer to keep that setting per project which allows to use different C++ standards between projects, when needed. Make sure that **Test** is selected, go to **Compiler settings** tab, there to **Compiler flags** tab and check **Have g++ follow the C++17 ISO C++ language standard (-std=c++17)**, see Figure 4-16. Press the **OK** button to close the dialog.

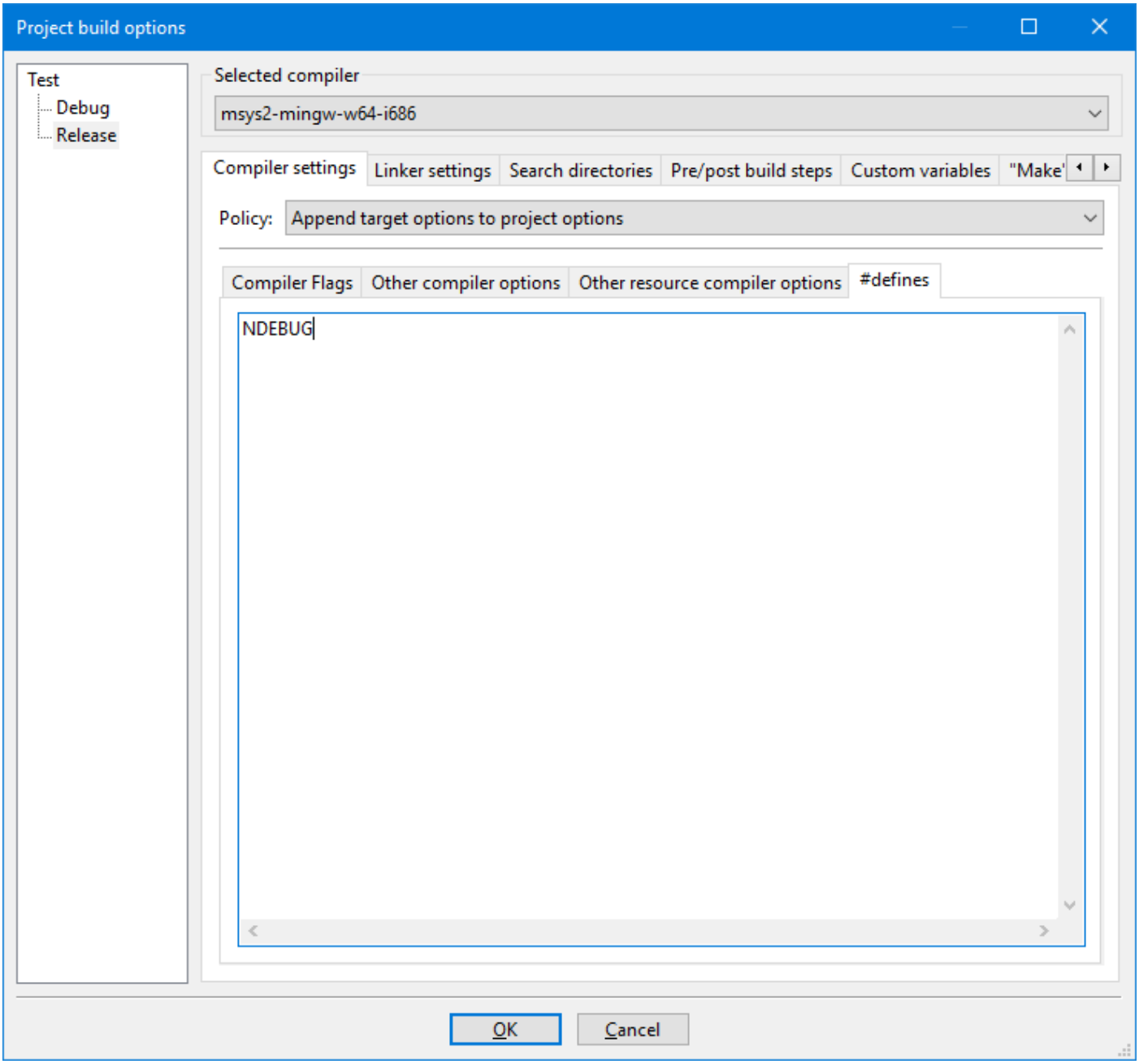
Figure 4-16 Setting C++ standard for all build targets



4.6.2 Setting Preprocessor Define *NDEBUG* for the Release Build Target

Now we want to set *NDEBUG* preprocessor define in the Release target to disable showing assert messages to the end user. From the main menu choose **Project** / **Build Options** which will show the Project Build options dialog. In the top left part of the dialog select the **Release** target. Go to **Compiler settings** tab, there to **#defines** tab and type **`NDEBUG`** there, see Figure 4-17. Press the **OK** button to close the dialog.

Figure 4-17 Setting preprocessor define *NDEBUG* for the Release build target



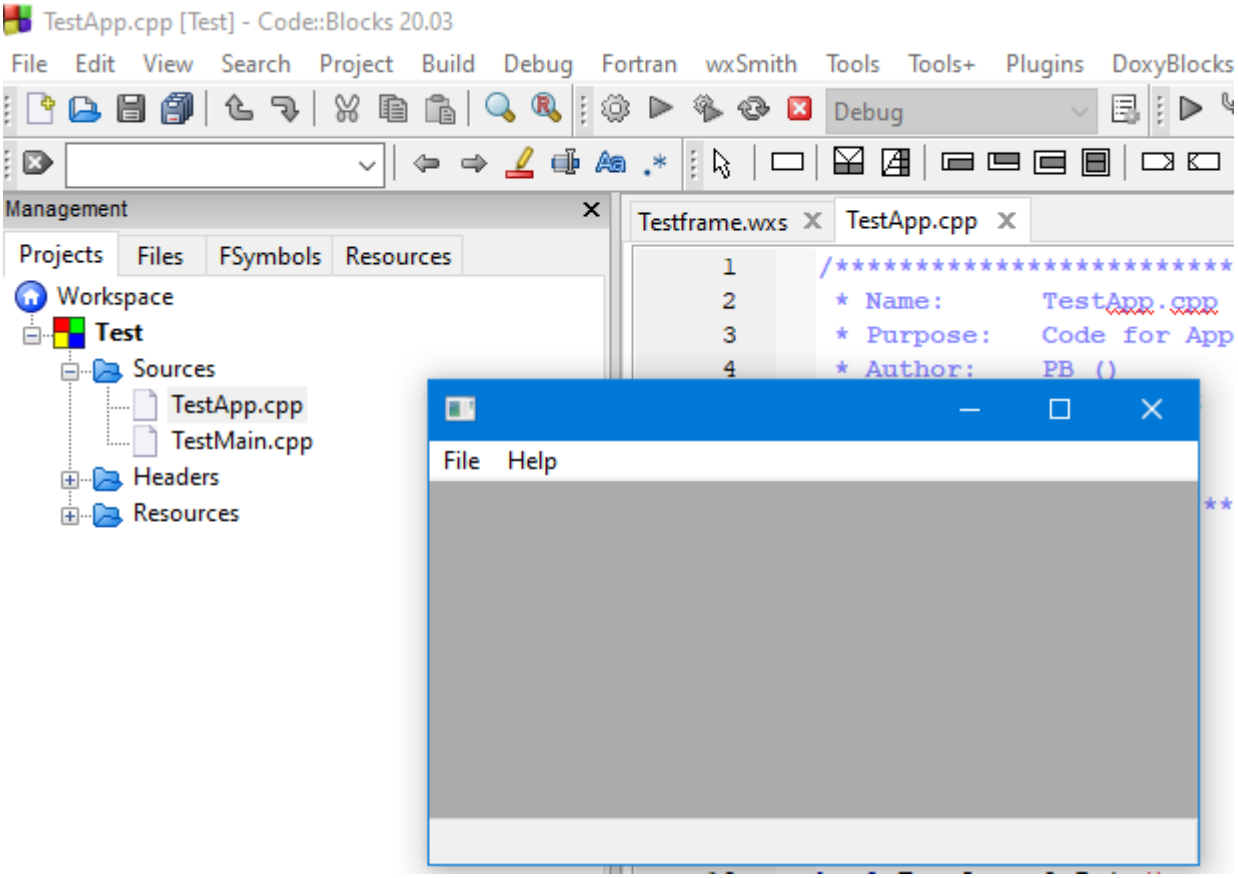
4.6.3 Setting Other Project Options

We will not set any other options for this project. But, for example, if you want your project to link to the compiler libraries statically, you can check **Static linking [-static]** in **Compiler flags** tab in **Compiler settings** tab. Of course, doing that assumes you (1) built wxWidgets statically with static linking of compiler libraries (see chapter 3.6.7) and (2) did not check “Use wxWidgets DLL” in the wizard (see chapter 4.5.9).

4.7 Finally Finished!

We are done, From the main menu choose **Build** / **Build and run** (or just press <F9>) to build and run our test application. If you followed this guide carefully, you should see the build to succeed and the test application launched (Figure 4-18).

Figure 4-18 Main window of the running test application shown on top of Code::Blocks



You may ask how it is possible that the application is running: We used the dynamic linking but did not copy any DLLs anywhere? It is because Code::Blocks is helpful and when it runs the application it adds the compiler path and the linker libraries paths to the application's executable environment, so it can find its DLLs. However, if you want to run the application outside Code::Blocks, you still need to copy the required DLLs to the same folder where the executable is (see chapter 3.6.5, and also chapters 3.6.7 and 4.6.3).

4.8 Extra Reading: wxWidgets Code::Blocks Project Under the Hood

When creating a project with template wizard (see chapter 4.5.1), Code::Blocks will set all necessary project settings according to our choices. However, sometimes it is good or even necessary to know what setting they are. When using a compiled library such as wxWidgets, in a project you usually need to set at least these things:

1. compiler include folder(s),
2. preprocessor defines,
3. linker settings: names of libraries and where to find them.

The following chapters describe how to set wxWidgets related information in a Code::Blocks project and what are the differences in settings needed when using a static build of wxWidgets. When referring to wxWidgets root, we will use Code::Blocks variable *wx* we created in chapter 4.4 and used in the wizard in chapter 4.5. In other words, when `$(#wx)` is used, it translates to system environment variable *WXWIN* we created in chapter 3.5, which in turn points to the actual folder, such as *c:\dev\libs\wxWidgets-3.1.4*.

4.8.1 Compiler Include Folders

wxWidgets uses two include folders. One is the main and is set for the whole project, it should be `$(#wx)/include`. The other is build-specific and must be set differently for each build target, should be set to `$(#wx)/lib/gcc_dll/mswud` for the Debug build target and to `$(#wx)/lib/gcc_dll/mswu` for the Release build target. Both include folders must be set for the C++ compiler, only the main include folder is needed for the Resource compiler.

Note. The build-specific include folder not being set for C++ compiler is the cause of a common build error with error message such as "*wx/platform.h:136:22: fatal error: wx/setup.h: No such file or directory*".

4.8.2 Preprocessor Defines

The only necessary project-wide settings are `__WXMSW__` and `wxUSE_UNICODE`. If you are using, as default in this guide, a shared (DLL) build of wxWidgets, `WXUSINGDLL` must be defined as well.

I also recommend defining `NDEBUG` for the Release build, see chapter 4.6.2.

4.8.3 Linker Settings

The project wizard should have set the linker folder and added all necessary libraries. However, you may sometime need to change this.

The folder with libraries is set in **Project** / **Build Options** in **Search directories** tab in **Linker**, where there should be **`$(#wx)/lib/gcc_dll`** for the shared build; and is same for both Debug and Release build target.

The individual libraries are added in in **Project** / **Build Options** in **Linker Settings** tab in **Link libraries**. The libraries need to be set separately for the Debug and Release build target. Debug versions of wxWidgets libraries have an extra **`d`** in their name, see chapter 3.6.4 where the file naming pattern for wxWidgets libraries is explained.

For the record, here is the list of all wxWidgets libraries for the Debug and Release build targets, in the order they must be listed (see the Note below):

Debug	*Release*
libwxmsw31ud_richtext.a	libwxmsw31u_richtext.a
libwxmsw31ud_xrc.a	libwxmsw31u_xrc.a
libwxmsw31ud_aui.a	libwxmsw31u_aui.a
libwxmsw31ud_media.a	libwxmsw31u_media.a
libwxbase31ud_net.a	libwxbase31u_net.a
libwxmsw31ud_gl.a	libwxmsw31u_gl.a
libwxbase31ud_xml.a	libwxbase31u_xml.a
libwxmsw31ud_adv.a[1]	*libwxmsw31u_adv.a*[1]
libwxmsw31ud_html.a	libwxmsw31u_html.a
libwxmsw31ud_core.a	**libwxmsw31u_core.a**
libwxregexud.a[2]	**libwxregexu.a**[2]
libwxbase31ud.a	**libwxbase31u.a**
libwxpngd.a	**libwxpng.a**
libwxjpegd.a[3]	libwxjpeg.a[3]
libwxtiffd.a[3]	libwxtiff.a[3]
libwxzlibd.a	**libwxzlib.a**
libwxexpatd.a	libwxexpat.a

[1]This library is not actually needed since wxWidgets 3.1.2 and is kept only for backwards compatibility with older projects which may still refer to it.

[2]The dependency of wxbase on wxregex was introduced in wxWidgets 3.1.4 but was removed shortly after 3.1.4 release.

[3]These libraries are needed for any application using static build of wxWidgets.

You can remove libraries you do not use; many applications will not use at least some of them. The libraries in bold are the most basic ones even the simplest GUI application must link with. See wxWidgets library list to see which classes require which libraries and for inter-library dependencies.

Note. Please notice that with GCC (unlike with MSVC), the library order matters. When *libraryA* requires *libraryB*, *libraryA* must be listed **before** *libraryB*. For example, *libwxpng.a* must be listed before *libwxzlib.a* on which it depends.

4.8.4 Using Static Instead of Shared wxWidgets Build

The only relevant differences when using the static wxWidgets compared to the shared one are:

1. In the names of include and linker folders, wherever there is **`gcc_dll`** for the shared build, there must be **`gcc_lib`** for the static build.
2. In **Project** / **Build Options** do **not** define **`WXUSINGDLL`** in **Compiler settings** tab in **#defines**.
3. Tell the compiler you want GCC libraries to be linked statically (assumes you built wxWidgets like that as well, see chapter 3.6.7), as described in chapter 4.6.3.

Document History

Version 1.0, released June 4, 2020

Initial release

Version 1.1, released June 7, 2020

- Add information that wxAdvanced library is no longer needed.
- Improve formatting and wording.

Version 1.2, released June 22, 2020

- Add missing “\lib” in the include and linker paths in chapters 4.7.1 and 4.7.3.
- Remove incorrect statement about the pre-built binaries being available only in the Release configuration.
- Suggest that using the shared wxWidgets build is preferable to using the static one.
- Make minor improvements in writing style and text formatting.

Version 1.3, released June 30, 2020

- Correct Figure 4.16 where a preprocessor define was wrongly entered in the “Other compiler options” field.
- Add a note about not setting WXWIN system environment variable.
- Mention that mingw-w64 is used to build the official MS Windows binaries.
- In chapter 3.5 link to the bat files for building wxWidgets available in the GIT repository.
- Fix typos and improve wording.

Version 2.0, released August 13, 2020

- Update for wxWidgets 3.1.4.
- Switch compiler toolchain from project mingw-w64 to MSYS2.
- Include information about setting up the debugger in Code::Blocks.
- Minor text improvements.

Version 2.1, released August 14, 2020

- Fix invalid hyperlinks to some chapters and TOC formatting.
- In chapter 2.3 fix the mistake about monolithic build being used in this guide.
- Update the information about dependency of the base library on the regex one.

Version 2.2, released August 26, 2020

- Add more information about which wxWidgets libraries are needed for a project in chapter 4.8.3.
- State that this guide is not a reference manual but a tutorial for new users.

Version 2.3, released September 8, 2020

- Make some language corrections.
- Add information about using a custom configuration with Code::Blocks’ wxWidgets project wizard.
- Clarify that the build commands are for Windows command prompt, not MSYS environment.

- Move chapter Making wxWidgets Build Faster before chapter Viewing Build Results.

Version 2.4, released September 22, 2020

- In chapter Viewing Build Results refer to the listing using the parallel build.
- Add a note regarding lack of Code::Blocks wxWidgets project wizard support for a custom compiler prefix for a wxWidgets build.
- Make some language corrections.