wxWidgets

Mastering wxWidgets

A Complete Practical Guide from Fundamentals to Professional Cross-Platform GUI Development

CROSS-PLATFORM
Windows • macOS • Linux and more

POWERFUL & MATURE
Proven, Reliable, and Production-Ready

DEVELOPER FRIENDLY
Intuitive, Flexible, and Extensible

https://ForgeVM.org

Prepared by
Ayman Alheraki

Created with AI assistance and fully reviewed, validated, and tested.

Mastering wxWidgets

A Complete Practical Guide from Fundamentals to Professional Cross-Platform GUI Development

Prepared by Ayman Alheraki

Produced under the ForgeVM Project

https://ForgeVM.org

English edition. Written as a practical learning and engineering reference for building native desktop tools with modern C++ and wxWidgets.

Inside This Guide

Native GUI	Modern C++23	CMake	NASM Editor
Windows / macOS / Linux	architecture + RAII	reproducible builds	complete capstone

Learning path: application core → controls → layout → events → advanced views → graphics → large-application architecture → editor integration.

Target baseline: wxWidgets 3.3.3 - Modern C++23 - CMake/Ninja examples

Publishing Notes

Prepared by Ayman Alheraki

This book was prepared first and foremost to support the author's own in-depth study of **wxWidgets** and to guide the design and implementation of a practical NASM-oriented assembler editor for the ForgeVM project. The intent is not to create a collection of disconnected recipes, but a coherent engineering reference that explains what each major class does, when it belongs in an architecture, how it interacts with the rest of the framework, and which mistakes commonly make desktop applications fragile.

ForgeVM Project Context

ForgeVM is a low-level software infrastructure project. The editor developed from the techniques in this book is intended to begin as a NASM-focused source editor and later evolve into a reusable development surface for ForgeVM tools. The design emphasis is therefore on native performance, predictable behaviour, maintainability, cross-platform structure, and the ability to integrate compilers, assemblers, diagnostics, files, processes, and future project-specific tooling.

AI Assistance, Review, and Validation Policy

Artificial intelligence was used to accelerate drafting, organization, layout exploration, and the production of explanatory examples. The manuscript has been structured for technical review and reproducible testing. However, no publication should claim that every code path has been compiled on every supported platform unless that validation has actually been performed. The final ForgeVM release process should build the example projects on the intended Windows, Linux, and macOS toolchains and record the results.

Publication and Validation Policy

Independent publication

This is an independent learning guide. It is not an official wxWidgets manual and is not endorsed by the wxWidgets project. Names and marks are used only to identify the technology being discussed. The authoritative reference for exact API signatures and platform support remains the official wxWidgets documentation.

Before a public release

- Compile every example from a clean build directory using each claimed toolchain.
- Launch all GUI examples and exercise their core interactions.
- Verify Windows resources and the application manifest are embedded.
- Test paths containing spaces and Unicode characters.
- Record the exact wxWidgets, compiler, CMake, NASM, and operating-system versions used.
- Re-check API signatures and incompatible changes against the official wxWidgets 3.3 documentation.

Version baseline

The manuscript is organized around wxWidgets 3.3.3, modern C++23, and target-based CMake builds. The 3.3 series is a development branch, so readers upgrading to later 3.3 releases or the future stable series should review the official change log and incompatible-change notes rather than assuming binary or source compatibility in every detail.

Living Reference and Release Record

The LaTeX source is intentionally modular. Chapters, examples, class-reference tables, and appendices can be revised independently as ForgeVM tooling grows or wxWidgets evolves. Treat each public edition as a reproducible engineering snapshot rather than an unversioned collection of examples.

What should be recorded for each edition?

Record the exact wxWidgets tag or commit, compiler family and version, CMake and generator versions, NASM version used by the capstone, operating-system release, and the date on which the example suite was rebuilt. If a platform was not tested, say so explicitly instead of turning an assumption into a compatibility claim.

Validation area	Minimum check	Evidence to retain
Toolchain	Clean configure; Debug/Release build	Compiler, generator, options, and warning-reviewed build log.
Launch	Exercise every GUI sample	Short record of the interactions tested and observed result.
Windows resources	Inspect manifest/resources	Resource script and manifest inspection result.
Paths and files	Spaces plus Unicode	Tested paths and any platform-specific findings.
Process integration	Run NASM asynchronously	Command, exit status, and stdout/stderr capture result.

Platform UI	Claimed OSes and DPI scales	OS/toolchain matrix, layout observations, and known limitations.

Release evidence worth keeping with ForgeVM

- Versioned PDF, exact LaTeX sources, and example tree from the same revision.
- Build/test notes for every claimed platform, including intentionally untested areas.
- Official wxWidgets documentation links used to confirm version-sensitive APIs.

Author's Preface

This book began with a practical need. I wanted to build a serious native editor for assembly language as part of the ForgeVM project, starting with NASM and leaving room for deeper integration with future ForgeVM tools. I did not want to approach wxWidgets as a collection of controls to be memorized. I wanted to understand the framework as an application architecture: how the event loop owns the rhythm of the program, how windows form an ownership hierarchy, how sizers replace fragile coordinates, how resources and manifests affect a Windows executable before my own code even runs, and how larger programs remain manageable as the number of views, commands, documents, workers, and platform-specific details grows.

wxWidgets has supported native desktop software for decades, yet a new C++ developer can still find a gap between the reference manual and the kind of progressive, engineering-focused explanation needed to build a real application. A reference tells you that a class exists. A useful learning guide must also explain why it exists, what problem it solves, what alternative you might choose, what lifetime rules apply, and what symptoms appear when you misuse it.

The editor is therefore not merely the last project in the book. It is the organizing problem behind the book. File handling, menus, accelerators, status reporting, styled text, subprocesses, diagnostics, background tasks, configuration, persistent state, DPI awareness, and cross-platform deployment are all studied because a real development tool needs them.

I also want this reference to remain useful beyond the original editor. The same architecture can support debugger front ends, binary viewers, instruction explorers, build tools, log viewers, configuration utilities, and other applications around ForgeVM. Readers outside the project can use the same material to build their own native C++ desktop software.

The most important promise of the book is progression. Start with a tiny window. Understand

it. Add layout. Understand ownership. Add events. Understand routing. Add data. Understand separation. Add background work. Understand thread boundaries. Only then assemble these ideas into a larger tool. A mature framework becomes much easier when each layer is learned at the moment it becomes necessary.

Why This Matters

The goal is not to memorize wxWidgets. The goal is to develop a mental model strong enough that the API becomes discoverable. Once you understand ownership, event routing, sizing, native controls, and application boundaries, unfamiliar classes become variations on patterns you already know.

Ayman Alheraki
ForgeVM Project
https://ForgeVM.org

How to Use This Book

1. **Read the first six chapters in order.** They build a single mental model: application startup, windows, layout, controls, and events.
2. **Compile the complete examples.** The book intentionally favours complete programs over isolated fragments. Copy them into small projects, build them, and make one deliberate modification after each successful run.
3. **Treat callout boxes as engineering rules.** Blue boxes explain concepts, green boxes capture preferred practices, and red boxes mark mistakes that are expensive in real applications.
4. **Use the class cards as a fast API layer.** Every important class is presented with its purpose, typical use, lifetime model, key methods, and at least one realistic context.
5. **Build the editor incrementally.** Chapter 10 is designed as an integration project. It reuses decisions made earlier instead of introducing a completely new architecture at the end.

Recommended validation loop

1. Configure from a clean build directory.
2. Build with warnings enabled.
3. Launch and exercise the feature manually.
4. Verify file and process paths containing spaces.
5. Test both Debug and Release configurations.

6. On Windows, verify that the resource/manifest is embedded.
7. On cross-platform examples, keep platform-specific code isolated behind narrow interfaces.

Notation and Baseline

The examples target **wxWidgets** 3.3.3, released in July 2026, and modern C++23. Most examples use CMake with target-based linking and show a source-tree workflow through `add_subdirectory()` because it makes the selected wxWidgets configuration explicit and reproducible while learning. Installed-package workflows are covered separately.

On Windows, examples that produce GUI executables include an `app.rc` resource file. This is not decorative: it is part of the executable's platform integration. In particular, the Windows common-control subclassing APIs used by modern GUI code require Comctl32 version 6 to be selected in the application manifest. The resource/manifest chain is therefore treated as part of the program, not as an afterthought.

Practical Tip

When a book example uses `new` for a `wxWindow`-derived child control, do not automatically translate it into `std::unique_ptr`. wxWidgets manages the lifetime of windows through the native window/parent hierarchy. Use ordinary modern C++ ownership for your non-window domain objects, and respect wxWidgets ownership rules for GUI objects.

Contents

CHAPTER 1

Getting Started with wxWidgets

Chapter Outcomes

By the end of this chapter you should be able to explain what wxWidgets is, distinguish its native-widget model from custom-drawn GUI frameworks, describe the startup path from the operating system to the event loop, build the smallest useful application, and recognize the project files that are part of a correct Windows executable.

1.1 What wxWidgets Is

wxWidgets is a mature open-source C++ framework for building native graphical desktop applications. Its central promise is deceptively simple: write one C++ application architecture and map it to the conventions and native facilities of the host platform. A button is not merely a rectangle painted to look approximately like a Windows, macOS, or GTK button. Where practical, wxWidgets wraps the platform's own control or closely integrates with the native toolkit used by that port.

This approach matters because desktop software is more than pixels. Native controls participate in accessibility, keyboard navigation, system themes, high-DPI behaviour, input methods, locale conventions, focus rules, screen readers, and operating-system updates. A framework that delegates these responsibilities to the host platform can provide an application that feels less like a foreign layer placed on top of the desktop and more like software that belongs there.

The library is broad. In addition to windows and controls, it includes file and directory abstractions, streams, dates and times, configuration support, clipboard integration, printing, image handling, processes, sockets, threading primitives, internationalization, rich data views, styled text editing, property grids, docking interfaces, XML resources, and many smaller platform-neutral services. You do not need to use all of them. The important architectural advantage is that a single event-driven object model connects these facilities.

Why This Matters

A code editor is a demanding test of a GUI framework. It needs fast text editing, menus, keyboard shortcuts, file dialogs, status information, subprocess execution, diagnostics, persistent settings, high-DPI rendering, and clean behaviour on multiple operating systems. Learning wxWidgets through the lens of an editor forces us to understand the framework as a system rather than as a catalogue of controls.

1.2 A Mature Native-GUI Philosophy

The project has existed for more than three decades and has accumulated a large body of platform knowledge. That history is important not because age alone proves quality, but because desktop frameworks encounter thousands of edge cases that are difficult to anticipate in a young abstraction: focus changes during modal dialogs, native menu ownership, screen scaling, IME composition, drag-and-drop semantics, printing differences, window destruction order, and message-loop interactions.

wxWidgets generally prefers to expose a portable API while leaving room for controlled platform-specific code when the abstraction cannot express an important native capability. This is a healthy design for serious applications. A cross-platform program should share as much policy and domain logic as possible, but it should not pretend that all operating systems are identical. Later chapters therefore distinguish between *portable application architecture* and *platform adaptation layers*.

1.2.1 Native look and feel versus identical look

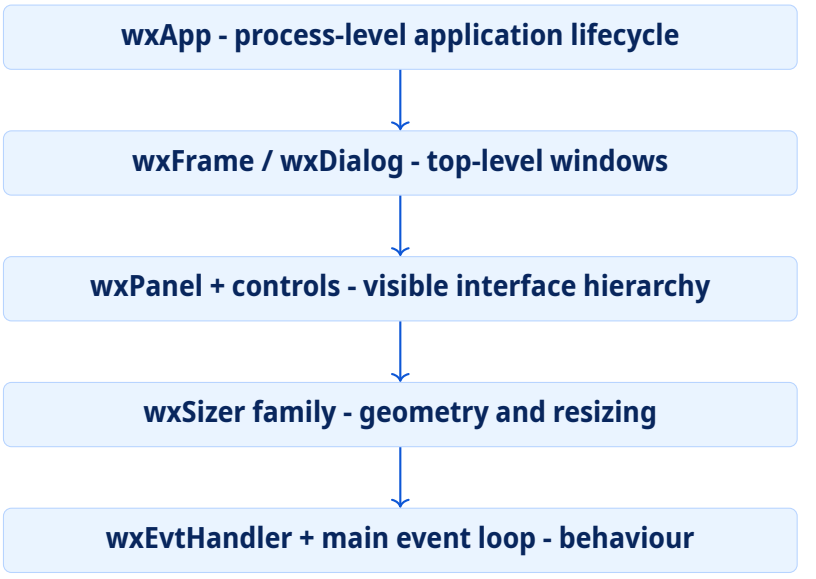
A common misunderstanding is that cross-platform means pixel-for-pixel identical. wxWidgets instead aims for an application to behave appropriately on each target. A menu may occupy a different location on macOS than on Windows. Font metrics differ. Default dialog button ordering differs. Native file dialogs differ. That variation is not automatically a defect; it is often the correct manifestation of the host platform's conventions.

For the ForgeVM editor, the design objective should therefore be **functional and structural consistency**, not forced visual uniformity. Commands, documents, diagnostics, and editor semantics should be stable everywhere, while native controls are allowed to express the desktop environment in which the program runs.

1.3 The Mental Model: Five Layers

Before learning individual classes, build a simple mental model.

1. **Application object.** wxApp represents the running GUI application and performs framework-level initialization.
2. **Top-level windows.** Classes such as wxFrame and wxDialog are independent desktop windows.
3. **Child windows and controls.** Panels, text controls, buttons, lists, trees, and editor widgets form a parent-child hierarchy.
4. **Layout.** Sizers calculate geometry from constraints instead of hard-coded coordinates.
5. **Events.** User actions and system notifications are converted into event objects and dispatched to handlers by the event loop.



1.4 Application Startup from First Principles

A console program often appears to run from the first line of `main()` to the last. A desktop GUI program quickly becomes event-driven instead. Initialization constructs the interface, after which control enters a loop that waits for messages from the operating system and the user.

The macro `wxIMPLEMENT_APP(MyApp)` supplies the framework-specific application entry machinery and associates it with your wxApp-derived class. During startup, wxWidgets creates the application object and calls `OnInit()`. A successful `OnInit()` normally creates one or more top-level windows, shows the first one, and returns `true`. The framework then enters its main event loop.

wxApp

Purpose. Represents the application and owns framework startup/shutdown responsibilities.

Use it when. You need a normal GUI application entry point, command-line processing, global initialization, or app-wide policy.

Remember. Keep heavyweight work out of `OnInit()`. Build the minimum UI needed to become responsive, then schedule longer work after the event loop starts.

wxFrame

Purpose. A normal top-level application window with support for menu bars, toolbars, status bars, and child content.

Use it when. The window is a primary document, workspace, tool, editor, or application shell.

Remember. Put actual content inside a child panel or specialized child window and manage it with sizers.

wxPanel

Purpose. A generic child container designed to host controls and participate in tab traversal and layout.

Use it when. You need a logical region inside a frame or dialog.

Remember. Panels make layout, focus handling, and modular UI construction clearer than placing every child directly on a frame.

1.5 The Minimal Complete Program

The following example deliberately contains a panel and a sizer even though a single static text could be positioned manually. The goal is to establish the correct pattern from the beginning.

Listing 1.1: Complete minimal application

```
#include <wx/wx.h>

class MyApp final : public wxApp
{
public:
    bool OnInit() override
    {
        auto* frame = new wxFrame(
            nullptr,
            wxID_ANY,
            "wxWidgets Minimal Example",
            wxDefaultPosition,
            wxSize(800, 500));

        auto* panel = new wxPanel(frame);
        auto* text = new wxStaticText(
            panel,
            wxID_ANY,
```

```
            "Hello from wxWidgets 3.3.3");

        auto* sizer = new wxBoxSizer(wxVERTICAL);
        sizer->AddStretchSpacer();
        sizer->Add(text, 0, wxALIGN_CENTER | wxALL, 12);
        sizer->AddStretchSpacer();
        panel->SetSizer(sizer);

        frame->Centre();
        frame->Show();
        return true;
    }
};

wxIMPLEMENT_APP(MyApp);
```

The key operations are small but important:

- `new wxFrame(...)` creates the top-level window.
- `new wxPanel(frame)` makes the panel a child of the frame.
- `new wxStaticText(panel,...)` creates a child control owned by the GUI hierarchy.
- `wxBoxSizer` defines vertical layout and adds flexible space above and below the label.
- `panel->SetSizer(sizer)` transfers the sizer into the panel's layout management.
- `frame->Show()` makes the top-level window visible.

Window ownership is framework-aware

The frequent use of new in wxWidgets examples can look suspicious to a modern C++ programmer. For objects derived from `wxWindow`, the framework participates in lifetime management: windows are destroyed with the corresponding GUI objects and child

windows are tied to their parent hierarchy. Do not mechanically wrap every child control in `std::unique_ptr`. Use modern RAII aggressively for your own data structures and services, but follow wxWidgets lifetime rules for window objects.

1.6 Project Anatomy

A small project can remain simple while still being correct:

```
wxMinimal/
|-- CMakeLists.txt
|-- main.cpp
|-- app.rc              # Windows resources + manifest path
`-- build/              # generated; never edit by hand
```

As the program grows, split application code by responsibility rather than by arbitrary file size. A future editor project might use:

```
ForgeEditor/
|-- CMakeLists.txt
|-- app.rc
|-- src/
|   |-- App.cpp
|   |-- MainFrame.cpp
|   |-- EditorView.cpp
|   |-- BuildRunner.cpp
|   `-- Settings.cpp
|-- include/forge_editor/
|   |-- App.hpp
|   |-- MainFrame.hpp
|   |-- EditorView.hpp
|   |-- BuildRunner.hpp
|   `-- Settings.hpp
|-- assets/
```

```
|-- tests/
`-- build/
```

1.7 CMake: Build wxWidgets with the Application

For learning and controlled development, building wxWidgets from a source tree with your application is explicit and convenient. The example below intentionally exposes the root directory as a CMake cache path so you can override it without editing the file.

Listing 1.2: Minimal source-tree build

```
cmake_minimum_required(VERSION 4.1)

project(wxMinimal LANGUAGES C CXX RC)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source tree"
    )

set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
set(wxBUILD_DEMOS OFF CACHE BOOL "" FORCE)

add_subdirectory(
    "${WXWIDGETS_ROOT}"
    "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL
)
```

```
add_executable(wxMinimal WIN32
    main.cpp
    app.rc
)

target_link_libraries(wxMinimal
    PRIVATE
    wx::core
    wx::base
)

if(MINGW OR CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
    target_compile_options(wxMinimal PRIVATE -Wall -Wextra -Wpedantic)
endif()
```

The important target-based idea is that your application does not manually assemble include paths and library filenames. Once wxWidgets has been added as a CMake subproject, you link to exported targets such as `wx::core` and `wx::base`. This allows wxWidgets' own build logic to communicate the include directories, compile definitions, and dependent libraries required by those components.

Practical Tip

During troubleshooting, prefer one coherent toolchain over a mixture of apparently compatible components. "GCC 16.2" from two distributions is not necessarily the same environment if the CRT, threading model, binutils, or library build differs. Keep compiler, linker, runtime libraries, wxWidgets build, and application build from the same selected toolchain.

1.8 The Windows Resource File Is Part of the Program

A lesson worth learning early is that a successful C++ link does not guarantee that a Windows GUI executable is correctly integrated with Windows. Resource scripts can embed icons, version information, and the application manifest. For wxWidgets, a minimal `app.rc` can include the framework's standard Windows resource script:

Listing 1.3: Minimal Windows resource file

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>
```

The CMake target must compile it:

```
add_executable(wxMinimal WIN32
    main.cpp
    app.rc
)
```

This detail is easy to dismiss until a loader error proves otherwise. The Windows common-control subclassing API `GetWindowSubclass` requires Comctl32 version 6 to be selected in the application manifest. If the resource/manifest path is removed, an application can build successfully and still fail at startup with an entry-point error or exhibit older common-control behaviour.

Real-world failure: Entry Point Not Found

If Windows reports that `GetWindowSubclass` cannot be located, do not begin by randomly changing compiler versions. First inspect the missing symbol and DLL, then verify the application manifest and resource file. Microsoft documents `GetWindowSubclass` as requiring Comctl32.dll version 6 in the manifest. For a wxWidgets application, retaining `app.rc` with `wx/msw/wx.rc` is therefore a core build requirement when you rely

on the standard Windows resource setup.

1.9 Debug and Release Are Different Products

A robust build workflow treats Debug and Release configurations as distinct outputs. Debug mode normally provides assertions, symbols, and more diagnosable behaviour. Release mode changes optimization and may expose timing or lifetime assumptions that remain hidden in a debug build.

Use separate build directories or a multi-configuration generator. For Ninja with a single configuration:

```
cmake -S . -B build-debug -G Ninja -DCMAKE_BUILD_TYPE=Debug
cmake --build build-debug -j 20

cmake -S . -B build-release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build-release -j 20
```

Never switch fundamental settings such as shared/static mode, compiler family, architecture, or CRT inside a stale build tree and assume the result is clean. CMake caches configuration state by design.

1.10 Core Classes to Recognize Immediately

At this stage you do not need to master these classes, but you should recognize their roles.

Class	Primary role
wxWindow	Base class for most visible windows and controls; provides parenting, sizing, events, focus, colours, fonts, coordinates, and native handle integration.
wxFrame	Main top-level application window.
wxDialog	Top-level interaction window, commonly modal for focused tasks.
wxPanel	Child container for grouping controls and managing tab traversal.
wxSizer	Abstract layout manager; derived sizers calculate child geometry.
wxEvtHandler	Base event-processing mechanism used throughout the framework.
wxCommandEvent	Event type used by buttons, menus, text changes, and other command-like controls.
wxString	wxWidgets string type used throughout the API and integrated with Unicode/platform conversion facilities.
wxFileName	Cross-platform path and filename abstraction.

wxConfig	Persistent application configuration abstraction.

1.11 A Productive Learning Method

The fastest route to proficiency is not to read every class in alphabetical order. Build one vertical slice at a time:

1. Create a frame and content panel.
2. Lay out two or three controls with a sizer.
3. Bind one event.
4. Save one piece of state.
5. Add a menu command for the same action.
6. Move the action into a non-UI service class.
7. Add an error path and display the result without blocking the interface.

Each slice teaches framework behaviour while preserving a running program. This is exactly how the NASM editor will evolve later.

1.12 Chapter Review

Checklist

You are ready for Chapter 2 when you can answer these questions without guessing:

- What object receives `OnInit()` and why?
- Why does a GUI application spend most of its lifetime in an event loop?

- What is the difference between a top-level frame and a child panel?
- Why are sizers preferred over hard-coded coordinates?
- Why is `app.rc` not merely cosmetic on Windows?
- What does linking to `wx::core` and `wx::base` achieve compared with manually listing library files?

Next step

Now that the conceptual skeleton is clear, Chapter 2 turns the build environment into a reproducible engineering system: compiler choice, source-tree versus installed builds, static versus shared libraries, Windows resources, diagnostic commands, and clean CMake patterns.

CHAPTER 2

Toolchain, CMake, Resources, and Reproducible Builds

Chapter Outcomes

Build wxWidgets and your application with a coherent compiler toolchain; understand source-tree, installed-package, static, and shared workflows; diagnose CMake cache and runtime-loading problems; and treat platform resources as first-class build inputs rather than optional decoration.

2.1 The Toolchain Is a System, Not Just a Compiler

A desktop C++ build crosses several layers before an executable can run. The source compiler is only one component. The linker, C and C++ runtime libraries, threading implementation, resource compiler, platform SDK headers, binary utilities, CMake generator, and wxWidgets library configuration must agree sufficiently for the final program to be coherent.

This explains why a build can fail even when every visible version number looks modern. For example, “GCC 16.2” describes the compiler version but not necessarily the runtime selection, distribution packaging, target CRT, or exact library binaries. When you build wxWidgets with one distribution and your program with another, a static archive may contain assumptions that the second toolchain cannot satisfy.

Build-chain mental model

Source code → **compiler** → object files → **linker** + wxWidgets + runtime libraries → executable → **OS loader** + manifests/resources + dynamic dependencies → running GUI.

A failure belongs to one of these stages. Good diagnosis first identifies the stage, then changes the smallest relevant variable.

2.2 Choose One Toolchain Per Build Tree

For Windows, common choices include MSVC, MSYS2 MinGW-w64, or a self-contained MinGW-w64 distribution such as WinLibs. Any of these can be valid. What matters is that wxWidgets and the application are built with the same intended toolchain when you consume source-built static libraries.

For Linux, use the distribution's GCC or Clang together with the platform development packages required by wxGTK, or use the distribution-provided wxWidgets package when appropriate. On macOS, Apple Clang and the installed SDK are the normal baseline.

Practical Tip

Record the exact compiler path in build logs. A message such as `C:/Program Files/JetBrains/.../mingw/bin/c++.exe` is more informative than a belief that CLion is "using GCC 16". CMake's configured compiler path is authoritative for that build tree.

Useful diagnostics include:

```
# MSYS2 / Unix-like shell
which gcc
which g++
g++ --version
```

```
g++ -v
cmake --version
ninja --version

# Windows cmd.exe
where gcc
where g++
where libstdc++-6.dll
```

Inside CMake, temporary messages can reveal the compiler actually selected:

```
message(STATUS "C compiler   : ${CMAKE_C_COMPILER}")
message(STATUS "C++ compiler : ${CMAKE_CXX_COMPILER}")
message(STATUS "C++ version  : ${CMAKE_CXX_COMPILER_VERSION}")
```

2.3 Two Main Ways to Consume wxWidgets

There are two broad CMake patterns worth understanding.

2.3.1 Build from a source tree with add_subdirectory

This is excellent for controlled learning, development against a known wxWidgets checkout, and projects that want to configure the library as part of the build.

```
set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")

set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)

add_subdirectory(
    "${WXWIDGETS_ROOT}"
    "${CMAKE_BINARY_DIR}/wxwidgets"
```

```
    EXCLUDE_FROM_ALL
)

target_link_libraries(MyApp PRIVATE wx::core wx::base)
```

Advantages include exact control over build options, no separate install step, and a single CMake dependency graph. The cost is longer initial configuration/build time and tighter coupling to the selected source-tree location unless you vendor or fetch the dependency in a controlled way.

2.3.2 Use an installed package

A prebuilt or installed wxWidgets package can reduce build time and improve team consistency. The exact CMake package interface depends on how wxWidgets was installed and on the package's exported configuration. Prefer modern imported targets when the package supplies them, because targets carry their own usage requirements.

Core Concept

Do not combine a source-tree example and an installed-package example in the same CMake target unless you have a deliberate reason. Choose one dependency source for a given build. Otherwise you can accidentally compile against headers from one wxWidgets configuration and link libraries from another.

2.4 Static and Shared wxWidgets

The choice between static and shared wxWidgets is a deployment decision, not a measure of program quality.

2.4.1 Static

With `wxBUILD_SHARED OFF`, wxWidgets components are linked into your executable (subject to how other dependencies are configured). Advantages include fewer wxWidgets DLL/shared-library files to deploy and easier "single application directory" distribution. Costs include larger executables, longer relinks, and the requirement to rebuild the application when the static library changes.

2.4.2 Shared

With `wxBUILD_SHARED ON`, wxWidgets builds shared libraries. The executable is smaller, independent library replacement can be convenient during development, and a suite of applications can potentially share common binaries. Deployment must ensure that the correct shared libraries are discoverable at runtime.

Decision	Static wxWidgets	Shared wxWidgets
Deployment	Fewer wxWidgets runtime files	Must deploy/find matching wxWidgets shared libraries
Executable	Larger	Smaller
Rebuild after wx change	Required	Application may not need relink for ABI-compatible replacement
Runtime DLL mismatch risk	Lower for wx libraries	Higher if search path finds wrong wx DLL

Learning simplicity	Often attractive after basic setup works	Often easiest initially if using official/prebuilt packages

Important Pitfall

The linker option `-static` is broader than `wxBUILD_SHARED OFF`. The first asks the MinGW linker to prefer static libraries generally; the second configures wxWidgets itself to build static libraries. Do not use them as synonyms. Start by choosing the wxWidgets linkage model. Only make the compiler runtime fully static if you have a concrete deployment requirement and have validated the entire dependency chain.

2.5 CMake Cache Discipline

CMake caches configuration choices so that subsequent builds are fast. This is a feature until you fundamentally change the build model. Switching from shared to static wxWidgets, changing compilers, changing architecture, or replacing the source tree can leave a cache that describes yesterday's world.

A reliable response to a structural toolchain change is simple:

```
rm -rf build
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build -j 20
```

On Windows PowerShell:

```
Remove-Item -Recurse -Force build
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build -j 20
```

Practical Tip

"Rebuild" in an IDE usually rebuilds targets inside the existing configured build tree. It does not necessarily discard the CMake cache. When the compiler or library mode changes, deleting the build directory is often the cleanest diagnostic move.

2.6 Windows Resources, Manifest, and Common Controls

A Windows executable can contain a resource section with icons, version metadata, dialogs, strings, and manifests. wxWidgets ships a resource script that helps establish the expected application manifest.

A minimal application resource file can be:

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>
```

and the CMake target should name the resource file:

```
add_executable(MyApp WIN32
    main.cpp
    app.rc
)
```

Why does this belong in a build chapter? Because the Windows loader processes executable metadata before your C++ application can present a useful error dialog. Microsoft documents that `GetWindowSubclass` requires Comctl32.dll version 6 to be specified in the manifest. Removing the manifest can therefore produce a successful compile and link followed by a startup failure.

2.6.1 A disciplined Entry Point Not Found investigation

When Windows displays an entry-point error, write down two strings before changing anything:

1. the exact missing procedure name;
2. the exact DLL named in the dialog.

Then inspect dependencies. With MinGW binutils:

```
objdump -p MyApp.exe | findstr /I "DLL Name"
```

In MSYS2, `ntldd` can show where each DLL resolves:

```
ntldd ./MyApp.exe
```

If the missing procedure is `GetWindowSubclass` and Common Controls v5.82 is being selected, investigate the manifest before replacing compilers. If the missing procedure is a C++ runtime symbol and `ntldd` resolves `libstdc++-6.dll` from an unexpected directory, investigate DLL search order and toolchain consistency instead.

Do not diagnose by changing everything at once

Switching IDE, compiler distribution, linkage mode, CMake version, and wxWidgets source simultaneously destroys evidence. Change one layer at a time. A good diagnostic experiment should answer a question: “Does the failure still happen when the program has the correct manifest?” is a useful experiment; “What happens if I install three more compilers?” usually is not.

2.7 CMake Targets and Usage Requirements

Modern CMake becomes easier when you think in targets. A target is not just a filename to link; it can carry include directories, compile definitions, transitive dependencies, and platform-specific options.

For an application:

```
add_executable(ForgeEditor WIN32
    src/App.cpp
    src/MainFrame.cpp
    app.rc
)

target_compile_features(ForgeEditor PRIVATE cxx_std_23)

target_link_libraries(ForgeEditor
    PRIVATE
    wx::core
    wx::base
)
```

Avoid global commands such as `include_directories()` and `add_definitions()` unless you truly intend their effect to leak across unrelated targets. Prefer `target_*` commands so every dependency remains attached to the target that needs it.

2.8 Warnings, Sanitizers, and Native GUI Builds

Warnings are part of the engineering feedback loop:

```
if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
    target_compile_options(ForgeEditor PRIVATE
        -Wall
        -Wextra
        -Wpedantic
    )
endif()
```

On platforms/toolchains where sanitizers are supported for your build, a dedicated configuration can catch lifetime and undefined-behaviour defects in your application logic. Be aware that sanitizer compatibility with GUI runtimes and third-party libraries varies; validate the exact environment rather than assuming every combination is supported.

A useful strategy is to keep an application library target containing most non-GUI logic. That target can be unit-tested and sanitized more easily than a monolithic GUI executable.

2.9 Out-of-Source Builds

Never generate CMake build artifacts into the wxWidgets source directory or your own source directory when a separate build tree is practical. Out-of-source builds allow multiple configurations to coexist and make cleanup deterministic.

```
project/
|-- CMakeLists.txt
|-- src/
|-- assets/
|-- build-debug/
|-- build-release/
`-- build-gcc-sanitized/
```

You can delete any build directory without risking source files.

2.10 Windows Console versus GUI Subsystem

The CMake keyword `WIN32` in `add_executable()` selects the Windows GUI subsystem for generators/toolchains that honour it. This affects the application entry model and whether a console window appears automatically.

During early debugging it can be useful to temporarily build without `WIN32` so diagnostic output written to standard streams has an attached console. For the finished desktop editor, use the GUI subsystem and route diagnostics into a log, debug output, or an in-application console pane.

2.11 Portable Paths and Configuration

Do not hard-code your personal wxWidgets path into a project you expect other people to build. Expose it as a cache variable, environment-driven preset, or dependency strategy.

```
set(WXWIDGETS_ROOT "" CACHE PATH "Path to wxWidgets source tree")

if(NOT EXISTS "${WXWIDGETS_ROOT}/CMakeLists.txt")
    message(FATAL_ERROR
        "Set WXWIDGETS_ROOT to a valid wxWidgets source directory")
endif()
```

For your own workstation, a CMake preset can supply the path without baking it into the project repository.

```
{
  "version": 6,
  "configurePresets": [
    {
```

```
      "name": "gcc-release",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/gcc-release",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release",
        "WXWIDGETS_ROOT": "E:/CPPLIB/wxWidgets3.3.3"
      }
    }
  ]
}
```

2.12 Deployment Thinking Begins During the Build

A GUI application is not finished when the linker produces an executable. Deployment includes runtime libraries, assets, configuration defaults, icons, translations, plug-ins, certificates/signatures where applicable, and platform packaging.

For a future ForgeVM editor, keep deployable resources in explicit directories and teach the build system where they belong. Avoid code that assumes the current working directory equals the executable directory; users can launch a program from shortcuts, shells, file associations, or other tools.

Use wxStandardPaths to discover appropriate application and user data directories. Use wxFileName to construct paths instead of manually concatenating separators.

2.13 A Reproducible Minimal Build Recipe

Build Recipe

Windows / MSYS2 UCRT64 example

```
# Verify environment
which gcc
which g++
g++ --version

# Clean configuration
rm -rf build

# Configure and build
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build -j 20

# Inspect dynamic dependencies if needed
ntldd ./build/wxMinimal.exe
```

2.14 Build Failure Taxonomy

Classify the error before fixing it.

Stage	Typical symptom	First evidence to inspect
Configure	CMake cannot find compiler/-package	CMake output, cache variables, compiler path
Compile	Header error, syntax error, missing definition	First compiler diagnostic, include path, feature macro
Link	Undefined reference, duplicate symbol	Full verbose link command, library order, toolchain origin
Load	Entry point not found, missing DLL	Exact symbol + DLL, manifest, `ntldd`/dependency inspection
Runtime	Crash, assertion, bad UI state	Debugger stack, wx assertions, logs, event/lifetime sequence
Packaging	Works on dev machine only	Dependency list, resource paths, clean target machine

2.15 Chapter Review

Checklist

Before continuing, verify that you can:

- identify the exact compiler and linker used by CMake;

- explain the difference between `wxBUILD_SHARED OFF` and linker-wide `-static`;
- delete and regenerate a stale build tree confidently;
- include `app.rc` on Windows and explain why the manifest matters;
- inspect executable DLL dependencies;
- distinguish configure, compile, link, load, and runtime failures;
- express project requirements using target-based CMake commands.

CHAPTER 3

Application Core: wxApp, wxFrame, Lifetime, and Main Loop

Chapter Outcomes

Turn the minimal example into a maintainable application skeleton. You will learn the responsibilities of wxApp, `wxFrame`, `wxWindow`, and `wxEvtHandler`; understand top-level versus child-window lifetime; use application-wide initialization carefully; and create a clean separation between GUI shell and domain services.

3.1 The Application Object Is Policy, Not the Whole Program

The wxApp object is the first wxWidgets object whose lifecycle you normally customize. It gives you well-defined hooks for startup and shutdown and access to application-global behaviour. The temptation is to place every service, setting, document, and command on the application object because it is globally available. Resist that temptation.

A good wxApp implementation answers process-level questions:

- Can the framework initialize successfully?

- What is the first top-level window?
- Which global services must exist before the first frame?
- How should uncaught application-level startup failures be reported?
- Is command-line parsing needed?
- What application name/vendor identity should configuration paths use?

It should not become a universal service locator. Domain logic belongs in ordinary C++ classes with explicit ownership and testable interfaces.

wxApp

Purpose. The process-wide application object and gateway into the wxWidgets event loop.
Use it when. You need GUI startup, shutdown, command-line initialization, app-wide identity, or top-window registration.
Remember. Keep `OnInit()` fast. Construct services deliberately and make ownership explicit.

3.1.1 OnInit()

The typical startup sequence is:

1. wxWidgets creates the wxApp-derived object.
2. Framework initialization occurs.
3. `OnInit()` is called.
4. The application creates its first frame and makes it visible.
5. `OnInit()` returns `true`.
6. The main event loop begins.

If initialization cannot continue safely, return `false`. Avoid leaving partially initialized global state that assumes an event loop will run.

3.1.2 OnExit()

`OnExit()` is useful for process-level cleanup that is not already handled by normal object lifetimes. Prefer RAII objects whose destructors naturally release resources. Use `OnExit()` for framework-level finalization, logging, or explicit teardown ordering only when needed.

Practical Tip

If a service can be an ordinary member of an application model object, make it one. Do not rely on `wxGetApp()` simply because it exists. Global reachability is convenient but couples unrelated parts of the program and makes testing harder.

3.2 wxWindow: The Root of Visible UI Behaviour

Most visible controls inherit from wxWindow. Understanding this base class makes the rest of the toolkit easier because many seemingly unrelated operations come from the same interface:

- parent and child relationships;
- size, position, minimum and best size;
- show/hide and enable/disable state;
- fonts, colours, cursor, and tooltip;
- focus and tab order;
- event handling;
- coordinate conversion;
- high-DPI helpers such as `FromDIP()`;

- invalidation and repainting;
- native window handles where platform-specific integration is necessary.

Parenting is architecture

The parent passed to a control constructor is not merely a drawing coordinate reference. It establishes the native containment hierarchy, affects lifetime, event propagation, focus, layout, clipping, and platform behaviour. Choose parents based on logical UI structure.

3.3 Top-Level Windows versus Child Windows

wxFrame and wxDialog are top-level windows. They can appear independently in the desktop window manager. Controls such as `wxButton`, `wxTextCtrl`, and `wxPanel` are normally child windows belonging to a parent.

This distinction affects lifetime. Child windows are destroyed as part of their parent hierarchy. Top-level windows have their own native lifecycle and should normally be closed/destroyed through wxWidgets rather than deleted arbitrarily while events may still target them.

Important Pitfall

Do not call C++ `delete` on a visible wxWidgets window merely because you allocated it with new. Window destruction can require deferred native cleanup. Use the framework's close/destroy lifecycle and let parent ownership clean up children.

3.4 wxFrame as an Application Shell

A frame has natural locations for desktop application infrastructure:

- menu bar;
- toolbar;
- status bar;
- central content region;
- optional docking panes or splitters;
- window-level commands and update-UI logic.

The frame should coordinate these pieces, not implement every operation itself. A useful editor architecture treats the frame as a shell that owns or references specialized views and services.

wxFrame

Purpose. Primary top-level desktop window with menu/status/toolbar support.
Use it when. Building an editor, IDE-like workspace, document window, utility, monitor, or main application shell.
Remember. Keep command wiring and view composition here; move filesystem, process, parsing, and domain logic into services.

3.5 A Structured Complete Example

The following example separates startup into App and workspace construction into `MainFrame`. Even at this small size, this structure scales much better than placing the entire program in `OnInit()`.

Listing 3.1: App.hpp

```
#pragma once
#include <wx/app.h>

class App final : public wxApp
{
public:
    bool OnInit() override;
    int OnExit() override;
};
```

Listing 3.2: App.cpp

```
#include "App.hpp"
#include "MainFrame.hpp"
#include <wx/log.h>

wxIMPLEMENT_APP(App);

bool App::OnInit()
{
    if (!wxApp::OnInit())
        return false;

    wxLogMessage("Application initialization started");

    auto* frame = new MainFrame();
    SetTopWindow(frame);
    frame->Show();
    return true;
}

int App::OnExit()
{
    wxLogMessage("Application shutdown");
```

```
    return wxApp::OnExit();
}
```

Listing 3.3: MainFrame.hpp

```
#pragma once
#include <wx/frame.h>

class MainFrame final : public wxFrame
{
public:
    MainFrame();

private:
    void BuildMenu();
    void BuildContent();
    void OnAbout(wxCommandEvent& event);
    void OnExit(wxCommandEvent& event);
};
```

Listing 3.4: MainFrame.cpp

```
#include "MainFrame.hpp"
#include <wx/menu.h>
#include <wx/msgdlg.h>
#include <wx/panel.h>
#include <wx/sizer.h>
#include <wx/stattext.h>
#include <wx/statusbr.h>

MainFrame::MainFrame()
    : wxFrame(nullptr, wxID_ANY, "Application Core",
              wxDefaultPosition, wxSize(900, 600))
{
    BuildMenu();
```

```
    BuildContent();
    CreateStatusBar();
    SetStatusText("Ready");
    Centre();
}

void MainFrame::BuildMenu()
{
    auto* fileMenu = new wxMenu();
    fileMenu->Append(wxID_EXIT, "E&xit\tAlt-F4");

    auto* helpMenu = new wxMenu();
    helpMenu->Append(wxID_ABOUT, "&About");

    auto* menuBar = new wxMenuBar();
    menuBar->Append(fileMenu, "&File");
    menuBar->Append(helpMenu, "&Help");
    SetMenuBar(menuBar);

    Bind(wxEVT_MENU, &MainFrame::OnExit, this, wxID_EXIT);
    Bind(wxEVT_MENU, &MainFrame::OnAbout, this, wxID_ABOUT);
}

void MainFrame::BuildContent()
{
    auto* panel = new wxPanel(this);
    auto* title = new wxStaticText(
        panel, wxID_ANY,
        "wxApp owns application policy; wxFrame owns the workspace.");

    auto font = title->GetFont();
    font.MakeBold();
    font.SetPointSize(font.GetPointSize() + 2);
    title->SetFont(font);
```

```
    auto* sizer = new wxBoxSizer(wxVERTICAL);
    sizer->Add(title, 0, wxALL, FromDIP(18));
    sizer->AddStretchSpacer();
    panel->SetSizer(sizer);
}

void MainFrame::OnAbout(wxCommandEvent&)
{
    wxMessageBox(
        "A structured wxWidgets application core.",
        "About",
        wxOK | wxICON_INFORMATION,
        this);
}

void MainFrame::OnExit(wxCommandEvent&)
{
    Close(true);
}
```

The associated build remains ordinary target-based CMake:

Listing 3.5: Application-core CMake project

```
cmake_minimum_required(VERSION 4.1)
project(wxAppCore LANGUAGES C CXX RC)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
```

```
add_subdirectory("${WXWIDGETS_ROOT}" "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL)

add_executable(wxAppCore WIN32
    App.cpp
    MainFrame.cpp
    App.hpp
    MainFrame.hpp
    app.rc
)

target_link_libraries(wxAppCore PRIVATE wx::core wx::base)

target_compile_options(wxAppCore PRIVATE
    $<$<CXX_COMPILER_ID:GNU,Clang>:-Wall;-Wextra;-Wpedantic>
)
```

3.6 SetTopWindow() and Application Intent

Calling `SetTopWindow(frame)` identifies the principal top-level window to the application object. A small program can often work without calling it explicitly because wxWidgets can infer useful state, but setting it documents intent and helps code that queries the application’s top window.

In a multi-window tool, decide whether there is one long-lived workspace frame or multiple independent document frames. This decision influences shutdown logic, command routing, and persistent geometry.

3.7 Window IDs

Every window has an identifier. Built-in IDs such as `wxID_OPEN`, `wxID_SAVE`, `wxID_EXIT`, and `wxID_ABOUT` communicate semantic intent and can integrate with stock labels/platform conventions. Use them when the action genuinely matches the stock meaning.

For application-specific commands, define IDs in a scoped enumeration or use automatically assigned IDs where the API permits it.

```
enum : int
{
    ID_Assemble = wxID_HIGHEST + 1,
    ID_Run,
    ID_ShowSymbols,
    ID_ToggleOutput
};
```

Avoid magic numbers spread across files. IDs are part of event routing and should be treated like named API constants.

3.8 Menus, Status Bars, and Command Ownership

A menu item produces a command event. Binding that event in the frame is natural when the command affects frame-level state. However, the handler should frequently delegate:

```
void MainFrame::OnAssemble(wxCommandEvent&)
{
    const auto file = editor_->CurrentPath();
    buildRunner_.Assemble(file);
}
```

The frame knows which document is active. The build runner knows how to invoke an assembler. This separation prevents command handlers from becoming subprocess implementations.

Why This Matters

Event handlers are adapters between UI events and application operations. Keep them short. The easiest GUI code to maintain is code in which clicking a button or selecting a menu item calls the same underlying operation.

3.9 Modal and Modeless Dialogs

wxDialog can be used modally, blocking interaction with its parent until the dialog is dismissed, or modelessly, remaining available alongside other windows.

Use modal dialogs for short, focused decisions: open/save, confirmation, a compact settings page, or a required input that must be resolved before continuing. Avoid modal dialogs for long-running tasks or passive status information.

A custom modal dialog pattern is:

```
class GotoLineDialog final : public wxDialog
{
public:
    explicit GotoLineDialog(wxWindow* parent)
        : wxDialog(parent, wxID_ANY, "Go to Line")
    {
        auto* line = new wxSpinCtrl(this, wxID_ANY);
        line->SetRange(1, 1'000'000);

        auto* buttons = CreateButtonSizer(wxOK | wxCANCEL);

        auto* sizer = new wxBoxSizer(wxVERTICAL);
        sizer->Add(line, 0, wxEXPAND | wxALL, FromDIP(12));
        sizer->Add(buttons, 0, wxEXPAND | wxLEFT | wxRIGHT | wxBOTTOM,
                   FromDIP(12));
        SetSizerAndFit(sizer);
    }
```

```
};
```

3.10 Application State versus View State

A professional editor has different kinds of state:

State kind	Examples	Best home
Application	recent files, theme preference, NASM path	settings/configuration service
Document	path, modified flag, encoding, source text model	document/editor model
View	caret location, selected pane, splitter position	view or persisted UI state
Command	can Save? can Assemble?	derived command/update state
Process	current build PID, output stream, exit code	process/build service

Keeping these categories distinct is one of the strongest defences against a “god frame” class.

3.11 Logging and Assertions

wxWidgets provides logging facilities that can route messages differently depending on build and platform. During development, use logs to capture startup configuration, file paths, tool invocation, and unexpected states.

Assertions are useful for programmer errors: an internal condition that should never be false if the program is correct. User errors belong in normal validation and error reporting instead.

```
wxASSERT(editor_ != nullptr);

if (sourcePath.empty())
{
    wxMessageBox("Save the source file before assembling.",
                 "Assemble",
                 wxOK | wxICON_INFORMATION,
                 this);
    return;
}
```

3.12 Main-Thread Rule

The GUI event loop runs on the main GUI thread, and wxWidgets does not support arbitrary GUI operations from worker threads. This rule should influence architecture from the beginning: background work produces data; the main thread applies that data to controls.

Later we will use posted events and `CallAfter()`-style techniques to return results safely to the GUI. For now, adopt one rule: **do not create or manipulate windows from worker threads**.

3.13 Close, Destroy, and Shutdown

When a user requests to close a frame, wxWidgets sends a close event. This gives the application a chance to ask about unsaved work. If closure proceeds, the native window is destroyed through the framework lifecycle.

A useful pattern:

```
void MainFrame::OnClose(wxCloseEvent& event)
{
    if (editor_->IsModified())
    {
        const int answer = wxMessageBox(
            "The file has unsaved changes. Close anyway?",
            "Unsaved Changes",
            wxYES_NO | wxNO_DEFAULT | wxICON_WARNING,
            this);

        if (answer != wxYES)
        {
            event.Veto();
            return;
        }
    }

    event.Skip();
}
```

The difference between `Close()` and `Destroy()` matters conceptually. `Close()` initiates the normal close process and can be vetoed; `Destroy()` requests destruction. Prefer the user-visible close path when the action semantically means “close this window.”

3.14 Designing the ForgeVM Editor Core

Before adding controls, define a small set of responsibilities:

```
App
`-- owns process-level services and creates MainFrame

MainFrame
|-- owns/composes visible views
|-- binds commands
|-- coordinates active document
`-- delegates work

EditorView
|-- displays/edits source
`-- exposes document-oriented operations

BuildRunner
|-- launches NASM
|-- captures output
`-- reports completion

Settings
`-- stores paths, UI state, and preferences
```

This is not a mandatory wxWidgets architecture. It is a practical architecture for a tool that will grow.

3.15 Chapter Review

Checklist

You should now be able to:

- keep `wxApp::OnInit()` focused on startup;
- explain why child controls allocated with `new` do not imply ordinary manual-delete ownership;
- use a frame as a shell rather than a domain-logic container;
- distinguish application, document, view, and process state;
- bind stock menu IDs and custom IDs intentionally;
- handle close requests without bypassing unsaved-work logic;
- respect the main-thread-only rule for GUI operations.

CHAPTER 4

Building Interfaces with Core Controls

Chapter Outcomes

Learn the most useful native controls as families rather than isolated classes. You will understand when to use text, choice, selection, numeric, button, list, notebook, splitter, menu, toolbar, and dialog controls; how to read and write their state; how styles affect behaviour; and how to avoid turning forms into tightly coupled event-handler code.

4.1 Controls Are Views onto State

A control is not the application state itself. A `wxTextCtrl` may display a path, but the canonical path belongs in your model or settings object. A check box may display whether line numbers are enabled, but the preference should not disappear when that check box is recreated.

This distinction becomes important as soon as you add multiple ways to change the same setting. If a menu item and a preferences dialog can both toggle line numbers, they should update one application setting and then refresh the relevant views. Treating each control as the only storage location creates synchronization bugs.

Core Concept

Think of controls as adapters between human interaction and typed program state. Read from them at clear boundaries, validate, update your model, and refresh the interface from that model when necessary.

4.2 Static Text and Labels

wxStaticText

Purpose. Displays non-editable text, labels, descriptions, values, and short status content.

Use it when. A user needs to read a label or compact result that is not directly editable.

Remember. Do not hard-code geometry around a label's current pixel width. Translation and DPI can change it. Let a sizer measure the best size.

Use `SetLabel()` to change ordinary labels. If the text participates in dynamic layout, call `Layout()` on the appropriate container when needed so the new best size is respected.

4.3 Buttons and Command Intent

wxButton

Purpose. Invokes a command through a native push button.

Use it when. The user explicitly commits an action such as Browse, Assemble, Apply, or Clear.

Remember. Use stock IDs such as `wxID_OK`, `wxID_CANCEL`, and `wxID_APPLY` when their semantics match.

A button should describe an action, not a vague noun. *Assemble* is clearer than *NASM*. *Choose*

Folder... is clearer than *Path*. Keyboard accelerators and default-button behaviour should reinforce frequent actions without surprising the user.

4.4 Single-Line and Multi-Line Text

wxTextCtrl

Purpose. General-purpose native text entry/display control supporting single-line and multi-line modes.

Use it when. Forms, paths, names, search boxes, compact logs, or ordinary text editing that does not require source-editor features.

Remember. For a programming editor, use `wxStyledTextCtrl`; for simple text input, `wxTextCtrl` is lighter and more native.

Common styles include `wxTE_MULTILINE`, `wxTE_READONLY`, `wxTE_PASSWORD`, and alignment options. Styles are constructor-time behavioural choices; do not assume every style can be changed meaningfully after native creation.

Useful methods include:

Method	Meaning
`GetValue()`	Read the current text.
`SetValue()`	Replace text and normally produce the associated text-change semantics.
`ChangeValue()`	Replace text without generating the normal text-changed event; useful when synchronizing a view from model state.
`AppendText()`	Add text efficiently to the end of the control.
`SetInsertionPointEnd()`	Move caret to the end.

`SetEditable(false)`	Make content non-editable without necessarily choosing a read-only constructor style.

Practical Tip

When programmatically refreshing a control from model state, prefer an API such as `ChangeValue()` when you specifically want to avoid recursively triggering the same user-edit handler that caused the refresh.

4.5 Check Boxes, Radio Buttons, and Toggle State

wxCheckBox

Purpose. Represents an independent Boolean option.
Use it when. A feature can be on/off without excluding another option.
Remember. Use a check box for persistent state, not for a one-shot action.

wxRadioButton

Purpose. Represents one mutually exclusive choice within a group.
Use it when. The user must choose exactly one of a small number of visible alternatives.
Remember. For many alternatives, a `wxChoice` usually scales better.

wxToggleButton

Purpose. A button with persistent pressed/unpressed state.
Use it when. A toolbar-like mode such as “Show whitespace” or “Pin output” benefits from compact toggle interaction.
Remember. Make the pressed meaning visually and semantically obvious.

4.6 Choice and Combo Controls

wxChoice

Purpose. A compact drop-down selection from a fixed list.

Use it when. The user chooses exactly one predefined value and free-form typing is not allowed.

Remember. Use `GetSelection()` for index and `GetStringSelection()` for display text; store a stable domain value when display text may be translated.

wxComboBox

Purpose. Combines editable text with a drop-down list.

Use it when. The user may select a common value or type a new one.

Remember. If arbitrary input is invalid, validate it explicitly or use `wxChoice`.

For the assembler editor, output format is a good `wxChoice`: `win64`, `elf64`, and `macho64` are known modes. A recently used symbol/search box might be a `wxComboBox` because free text is useful.

4.7 Numeric Input

wxSpinCtrl

Purpose. Integer entry with spin buttons and range enforcement.
Use it when. Small bounded integer settings such as tab width, indentation depth, history size, or polling interval.
Remember. Set a realistic range. A spin control without a meaningful range merely hides validation problems.

For floating-point values, investigate `wxSpinCtrlDouble`. For values that require domain-specific formatting, a text control plus a validator may be more appropriate.

4.8 Lists and Simple Collections

wxListBox

Purpose. Displays a simple list of strings with optional single or multiple selection.
Use it when. Recent projects, search matches, small symbol groups, or fixed string collections.
Remember. When rows need columns, icons, virtual data, sorting, or model separation, move to a richer control such as `wxDataViewCtrl`.

wxCheckListBox

Purpose. A list where each string has an independent check state.
Use it when. Feature lists, selectable build steps, or small sets of enable/disable items.
Remember. Do not use it as a substitute for a real table when items have multiple properties.

4.9 Notebooks, Splitters, and Panels

wxNotebook

Purpose. Tabbed pages using the native notebook/tab control.
Use it when. A small number of peer views such as Source, Symbols, and Disassembly.
Remember. Tabs hide content; do not bury commands or critical status in rarely visible pages.

wxSplitterWindow

Purpose. Two resizable child panes separated by a draggable sash.
Use it when. Editor/output, tree/editor, or preview/source layouts where users need to allocate space interactively.
Remember. Persist sash position if it materially affects workflow, but clamp restored values to the current window size.

wxPanel

Purpose. A child container for a logical interface region.
Use it when. Grouping form controls, composing custom pages, or isolating layout/event responsibilities.
Remember. Give complex panels their own classes instead of constructing hundreds of controls in the main frame constructor.

4.10 Menus as a Command Surface

Menus should expose the same application operations used by buttons, toolbars, and shortcuts. Use stock IDs where appropriate and assign accelerators in labels using the platform-supported syntax, for example:

```
auto* file = new wxMenu();
file->Append(wxID_NEW,  "&New\tCtrl-N");
file->Append(wxID_OPEN, "&Open...\tCtrl-O");
file->Append(wxID_SAVE, "&Save\tCtrl-S");
file->AppendSeparator();
file->Append(wxID_EXIT, "E&xit\tAlt-F4");
```

Do not duplicate the actual save implementation inside the menu handler if a toolbar button also saves. Both should call the same `SaveDocument()` operation.

4.11 Toolbars

wxToolBar provides a native/portable command strip. Use it for frequent actions whose icons and tooltips are recognizable. A toolbar supplements a menu; it should not be the only way to discover essential operations.

For editor design:

- New, Open, Save are natural toolbar commands.
- Assemble and Run are natural project commands.
- Toggle Output can be a check/toggle tool.
- Rare configuration commands belong in menus or preferences rather than consuming permanent toolbar space.

Use scalable bitmap/icon strategies appropriate to modern DPI environments. Avoid shipping a toolbar composed only of tiny 16x16 assets designed for a single scale factor.

4.12 Status Bars

wxStatusBar is ideal for concise, non-modal state: current line/column, encoding, insert/overwrite mode, build state, and short command feedback.

Do not turn the status bar into a logging system. Long diagnostics belong in an output pane or log view. A status bar should answer "what is the application doing now?" at a glance.

4.13 Dialogs

Use standard dialogs where the operating system already has a strong convention:

Class	Typical use
wxFileDialog	Open/save files with platform-native selection UI.
wxDirDialog	Choose a directory.
wxMessageDialog / wxMessageBox	Short notifications and explicit yes/no decisions.
wxColourDialog	Choose colours when a platform-native colour chooser is appropriate.
wxFontDialog	Choose fonts.
wxTextEntryDialog	Request one compact text value.

	Request a bounded integer.
wxNumberEntryDialog	

Practical Tip

Native file dialogs provide more than appearance. They integrate with platform navigation, recent locations, security conventions, and user expectations. Prefer them unless your application has a domain-specific file browser that truly needs a custom workflow.

4.14 Complete Controls Example

The following program creates a small settings form resembling options an assembler editor might expose. Notice that controls are stored as members only when later code needs to read or update them.

Listing 4.1: Core controls in a working form

```
#include <wx/wx.h>
#include <wx/choice.h>
#include <wx/listbox.h>
#include <wx/spinctrl.h>

class App final : public wxApp
{
public:
    bool OnInit() override;
};

class MainFrame final : public wxFrame
{
public:
    MainFrame();
```

```
private:
    wxTextCtrl* name_{};
    wxChoice* mode_{};
    wxSpinCtrl* tabWidth_{};
    wxCheckBox* lineNumbers_{};
    wxStaticText* summary_{};

    void UpdateSummary();
};

wxIMPLEMENT_APP(App);

bool App::OnInit()
{
    auto* frame = new MainFrame();
    frame->Show();
    return true;
}

MainFrame::MainFrame()
    : wxFrame(nullptr, wxID_ANY, "Controls Example",
              wxDefaultPosition, wxSize(720, 520))
{
    auto* panel = new wxPanel(this);

    auto* form = new wxFlexGridSizer(2, FromDIP(8), FromDIP(12));
    form->AddGrowableCol(1, 1);

    form->Add(new wxStaticText(panel, wxID_ANY, "Profile name:"),
              0, wxALIGN_CENTER_VERTICAL);
    name_ = new wxTextCtrl(panel, wxID_ANY, "NASM");
    form->Add(name_, 1, wxEXPAND);

    form->Add(new wxStaticText(panel, wxID_ANY, "Output format:"),
              0, wxALIGN_CENTER_VERTICAL);
```

```
    mode_ = new wxChoice(panel, wxID_ANY);
    mode_->Append("win64");
    mode_->Append("elf64");
    mode_->Append("macho64");
    mode_->SetSelection(0);
    form->Add(mode_, 1, wxEXPAND);

    form->Add(new wxStaticText(panel, wxID_ANY, "Tab width:"),
              0, wxALIGN_CENTER_VERTICAL);
    tabWidth_ = new wxSpinCtrl(panel, wxID_ANY);
    tabWidth_->SetRange(2, 12);
    tabWidth_->SetValue(4);
    form->Add(tabWidth_, 0, wxEXPAND);

    form->AddSpacer(1);
    lineNumbers_ = new wxCheckBox(panel, wxID_ANY, "Show line numbers");
    lineNumbers_->SetValue(true);
    form->Add(lineNumbers_, 0, wxEXPAND);

    auto* apply = new wxButton(panel, wxID_APPLY, "Apply");
    summary_ = new wxStaticText(panel, wxID_ANY, "");

    auto* root = new wxBoxSizer(wxVERTICAL);
    root->Add(form, 0, wxEXPAND | wxALL, FromDIP(16));
    root->Add(apply, 0, wxLEFT | wxRIGHT | wxBOTTOM, FromDIP(16));
    root->Add(new wxStaticLine(panel), 0, wxEXPAND | wxLEFT | wxRIGHT,
              FromDIP(16));
    root->Add(summary_, 0, wxEXPAND | wxALL, FromDIP(16));
    panel->SetSizer(root);

    apply->Bind(wxEVT_BUTTON, [this](wxCommandEvent&) { UpdateSummary(); });
    name_->Bind(wxEVT_TEXT, [this](wxCommandEvent&) { UpdateSummary(); });
    mode_->Bind(wxEVT_CHOICE, [this](wxCommandEvent&) { UpdateSummary(); });
    tabWidth_->Bind(wxEVT_SPINCTRL, [this](wxSpinEvent&) { UpdateSummary(); });
    lineNumbers_->Bind(wxEVT_CHECKBOX, [this](wxCommandEvent&) { UpdateSummary()
```

```
    ; });

    UpdateSummary();
    Centre();
}

void MainFrame::UpdateSummary()
{
    summary_->SetLabel(wxString::Format(
        "Profile: %s | Format: %s | Tab width: %d | Line numbers: %s",
        name_->GetValue(),
        mode_->GetStringSelection(),
        tabWidth_->GetValue(),
        lineNumbers_->GetValue() ? "on" : "off"));
}
```

Listing 4.2: Controls example build

```
cmake_minimum_required(VERSION 4.1)
project(wxControls LANGUAGES C CXX RC)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
add_subdirectory("${WXWIDGETS_ROOT}" "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL)
add_executable(wxControls WIN32 main.cpp app.rc)
target_link_libraries(wxControls PRIVATE wx::core wx::base)
```

The example demonstrates a useful pattern: multiple control events all call one `UpdateSummary()` function. That prevents five handlers from duplicating formatting logic.

4.15 Styles: Behaviour Chosen at Construction

wxWidgets constructors commonly accept a `long style` bit mask. Styles alter native control creation. Examples include read-only text, multi-selection list modes, border types, scroll behaviour, and alignment.

Use styles intentionally. A huge OR-expression of flags copied from a sample can create platform-specific surprises because flags may interact. Read the class documentation for the control you are constructing and choose the smallest set that expresses the intended behaviour.

4.16 Enable, Disable, Show, Hide

Use `Enable(false)` when a control or command is visible but temporarily unavailable. Use `Show(false)` when the entire element should disappear.

For dynamic forms, hiding a control does not always automatically produce the layout you expect unless the sizer is recalculated. After changing visibility of sizer-managed windows, call layout on the containing window when appropriate.

Core Concept

Disabling communicates "this exists, but cannot be used now." Hiding communicates "this is not part of the current interface." These are different messages to the user and should not be chosen only for convenience.

4.17 Focus and Tab Order

Keyboard navigation is part of interface quality. The default tab order generally follows creation order. For carefully structured forms, construct controls in a logical sequence and use the wxWindow tab-order APIs when necessary.

Avoid forcing focus unexpectedly after every command. Focus changes should correspond to user intent: opening Find can focus the search field; completing an assemble command should not steal focus from the source editor unless the user requested navigation to an error.

4.18 Validation Boundaries

A control can constrain input, but domain validation still belongs in application logic. A path text box can contain a syntactically valid string that points to a missing executable. A spin control can enforce a number from 2 to 12, but the project may support only 2, 4, or 8.

A useful flow is:

1. read UI values;
2. convert to domain types;
3. validate cross-field rules;
4. update model/settings;
5. persist if appropriate;
6. refresh dependent views.

4.19 Designing Controls for the NASM Editor

A first editor workspace can be mapped to controls without overdesign:

Need	Control	Reason
Source editing	wxStyledTextCtrl	Source-aware editing, margins, styling, markers
Output/diagnostics	wxTextCtrl or wxStyledTextCtrl	Multiline read-only output; STC if rich styling/markers are useful
Open/save	wxFileDialog	Native file selection
Format choice	wxChoice	Fixed set of NASM output formats
Assembler path	wxTextCtrl + Browse button	User-visible path plus native selection
Editor/output sizing	wxSplitterWindow	User-resizable work areas
Commands	wxMenuBar + wxToolBar	Discoverability plus fast access
Line/column state	wxStatusBar	Compact continuous feedback

4.20 Chapter Review

Checklist

You should now be able to select a control by behaviour rather than appearance, distinguish model state from control state, construct forms with appropriate native widgets, expose commands consistently through menus/buttons/toolbars, and reserve `wxStyledTextCtrl` for the source-editor role that justifies its richer feature set.

CHAPTER 5

Layout Management: Sizers, DPI, and Responsive Desktop Design

Chapter Outcomes

Master the layout system that makes wxWidgets interfaces portable. You will learn proportions, expansion, alignment, borders, best sizes, nested sizers, flex grids, static-box grouping, splitters, DPI-aware spacing, and the practical rules that keep dialogs and editor workspaces usable under resizing, translation, and different platform metrics.

5.1 Why Fixed Coordinates Fail

A desktop UI that looks correct at one resolution, language, font, and theme can break when any of those variables change. Hard-coded coordinates assume that text widths, control heights, border metrics, and available pixel density remain stable. They do not.

wxWidgets sizers solve this by describing *relationships*: this editor consumes the remaining vertical space; this label keeps its best size; these two form columns align; this button row sits beneath the content; this border uses a DPI-scaled logical distance.

The result is not "responsive web design" transplanted onto desktop software. It is constraint-based native window geometry appropriate to desktop controls.

Why This Matters

Sizers are one of the most important skills in wxWidgets. Many UI problems blamed on "wxWidgets layout" are actually caused by mixing manual sizes, inconsistent parents, and misunderstood sizer flags. Once the sizing model is clear, layouts become predictable.

5.2 The wxSizer Contract

wxSizer is an abstract layout manager. It stores items, calculates minimum/best geometry, and assigns rectangles to children. You normally instantiate derived classes such as `wxBoxSizer`, `wxFlexGridSizer`, or `wxGridBagSizer`.

Adding a child commonly involves four concepts:

1. **Item** - a window, child sizer, or spacer.
2. **Proportion** - whether the item receives a share of extra space in the sizer's main direction.
3. **Flags** - expansion, alignment, and border sides.
4. **Border** - the amount of border space when border flags are present.

5.3 wxBoxSizer: The Workhorse

wxBoxSizer

Purpose. Arranges children in one horizontal or vertical line.

Use it when. Most rows, columns, dialog bodies, button bars, and nested layout compositions.

Remember. Master box sizers before reaching for more complex layouts. Most professional interfaces are largely nested rows and columns.

5.3.1 Proportion

In a vertical box sizer, proportion controls how extra *vertical* space is shared. In a horizontal box sizer, it controls extra *horizontal* space.

```
auto* root = new wxBoxSizer(wxVERTICAL);
root->Add(toolbar, 0, wxEXPAND);      // keep best height
root->Add(editor,  1, wxEXPAND);     // consume remaining height
root->Add(status,  0, wxEXPAND);     // keep best height
```

If two items both have proportion 1, they share extra space equally. Proportions 1 and 2 divide it approximately one-third/two-thirds.

5.3.2 wxEXPAND

Proportion and wxEXPAND answer different questions. Proportion allocates space along the main axis. wxEXPAND allows the item to fill its assigned rectangle along the cross axis (and, depending on context, use allocated area instead of retaining best width/height).

A common error is to assign proportion 1 to a multi-line text control but omit wxEXPAND, leaving it narrow even though vertical space grows.

5.4 Borders and Spacing

Borders belong to items, not to the visual background. A readable pattern is:

```
root->Add(control, 0, wxEXPAND | wxLEFT | wxRIGHT | wxTOP,
          FromDIP(12));
```

Use `FromDIP()` for pixel-like dimensions that should scale with display density. Treat literal pixel constants as a warning sign in a high-DPI desktop UI.

Practical Tip

Choose a small spacing scale - for example 4, 8, 12, 16 logical pixels - and reuse it. Consistency looks more professional than individually tuned gaps around every control.

5.5 Alignment Flags

Alignment is most useful when an item is not expanding. Examples include centering a compact button, vertically centering a label beside a text field, or right-aligning a button group.

```
row->Add(label, 0, wxALIGN_CENTER_VERTICAL | wxRIGHT, FromDIP(8));
row->Add(text, 1, wxEXPAND);
```

Avoid contradictory combinations such as attempting to horizontally center an item while also asking it to expand horizontally. Debug builds can detect some inconsistent sizer flag combinations.

5.6 Nested Sizers Build Real Interfaces

A layout tree is usually easier to reason about than a complex grid that attempts to describe everything at once.

```
vertical root
|-- horizontal command row
|   |-- Open
|   |-- Save
|   |-- Assemble
|   `-- flexible spacer + configuration label
|-- splitter (proportion 1)
|   |-- editor
|   `-- output
`-- optional status region
```

Each level owns one simple relationship. This makes future changes local: adding another toolbar button does not alter the editor/output geometry.

5.7 wxFlexGridSizer for Forms

wxFlexGridSizer

Purpose. Grid layout with rows/columns where selected rows or columns can grow.
Use it when. Settings forms, property forms, label/control alignment, two-dimensional layouts with regular structure.
Remember. Mark the value column growable so text fields absorb horizontal resizing while labels retain best size.

Example:

```
auto* grid = new wxFlexGridSizer(2, FromDIP(8), FromDIP(12));
grid->AddGrowableCol(1, 1);
```

```
grid->Add(new wxStaticText(panel, wxID_ANY, "Assembler:"),
          0, wxALIGN_CENTER_VERTICAL);
grid->Add(pathCtrl, 1, wxEXPAND);
```

A fixed `wxGridSizer` makes all cells uniform and is better for equal-sized button matrices or similar regular content.

5.8 wxGridBagSizer for Irregular Grids

wxGridBagSizer

Purpose. Places items in a grid with row/column spanning and explicit cell positions.
Use it when. A form truly needs irregular spans that cannot be expressed clearly through nested box/flex-grid sizers.
Remember. Use sparingly. If the coordinates become an alternate manual-layout language, restructure the UI instead.

Grid bag layouts can be powerful for compact professional dialogs, but the clarity cost is higher because each item carries grid coordinates and spans.

5.9 wxStaticBoxSizer for Semantic Groups

wxStaticBoxSizer

Purpose. Sizer associated with a labelled static box.
Use it when. A settings page contains a small number of visually meaningful option groups.
Remember. Do not surround every two controls with a box; excessive framing creates

visual noise.

For an assembler settings dialog, groups such as “Toolchain”, “Output”, and “Editor” can be meaningful. Use whitespace as the primary grouping tool and boxes where a label adds real information.

5.10 Spacers and Stretch Spacers

Spacers are deliberate layout items, not hacks.

```
row->Add(saveButton, 0);
row->AddStretchSpacer();
row->Add(statusLabel, 0, wxALIGN_CENTER_VERTICAL);
```

The stretch spacer consumes extra space and pushes later controls to the far side. This is clearer than guessing a fixed width.

5.11 SetSizer, SetSizerAndFit, and Layout

`SetSizer()` associates a sizer with a window. The window then uses the sizer to lay out its children. `SetSizerAndFit()` also adjusts the window’s size to fit the sizer’s minimum requirements; this is particularly convenient for dialogs.

Use `Layout()` when dynamic changes (show/hide, label size, page composition) require recalculating child geometry. Do not call it continuously without reason; layout is normally triggered through size changes and framework behaviour.

5.12 Best Size, Minimum Size, and User Resizing

Controls can report a best size based on native metrics and content. Sizers combine those sizes to determine the minimum useful geometry. Avoid imposing arbitrary fixed sizes unless the control represents a genuinely fixed-size visual element.

For a main editor window, set a reasonable initial size but let the user resize freely. For small dialogs, use `SetSizerAndFit()` and optionally set a minimum size so translation or longer paths do not make the dialog unusable.

Important Pitfall

Calling `SetSize()` on every child control after putting it in a sizer fights the layout system. Decide whether geometry is sizer-managed or manual for a given child; do not use both simultaneously without a specific custom-layout reason.

5.13 Splitter Windows for Tool Workspaces

wxSplitterWindow is not a sizer, but it solves an important workspace problem: user-controlled allocation between two major panes.

A typical editor uses a vertical or horizontal split between source and output. Set a minimum pane size and use live updates where appropriate. Persist the sash position as view state, but validate restored values against the current client size.

5.14 Complete Editor-Like Layout Example

Listing 5.1: Toolbar row + resizable editor/output workspace

```
#include <wx/wx.h>
#include <wx/splitter.h>

class App final : public wxApp
{
public:
    bool OnInit() override;
};

class MainFrame final : public wxFrame
{
public:
    MainFrame();
};

wxIMPLEMENT_APP(App);

bool App::OnInit()
{
    auto* frame = new MainFrame();
    frame->Show();
    return true;
}

MainFrame::MainFrame()
    : wxFrame(nullptr, wxID_ANY, "Sizer Layout Lab",
              wxDefaultPosition, wxSize(980, 650))
{
    auto* rootPanel = new wxPanel(this);

    auto* toolbarRow = new wxBoxSizer(wxHORIZONTAL);
```

```
    toolbarRow->Add(new wxButton(rootPanel, wxID_OPEN, "Open"), 0, wxRIGHT,
    FromDIP(6));
    toolbarRow->Add(new wxButton(rootPanel, wxID_SAVE, "Save"), 0, wxRIGHT,
    FromDIP(6));
    toolbarRow->Add(new wxButton(rootPanel, wxID_ANY, "Assemble"), 0, wxRIGHT,
    FromDIP(6));
    toolbarRow->AddStretchSpacer();
    toolbarRow->Add(new wxStaticText(rootPanel, wxID_ANY, "NASM / win64"),
                    0, wxALIGN_CENTER_VERTICAL);

    auto* splitter = new wxSplitterWindow(
        rootPanel, wxID_ANY, wxDefaultPosition, wxDefaultSize,
        wxSP_LIVE_UPDATE | wxSP_3D);

    auto* editor = new wxTextCtrl(
        splitter, wxID_ANY,
        "; source editor area\nsection .text\nglobal main\nmain:\n    xor eax,
    eax\n    ret\n",
        wxDefaultPosition, wxDefaultSize,
        wxTE_MULTILINE | wxTE_DONTWRAP);

    auto* output = new wxTextCtrl(
        splitter, wxID_ANY,
        "Build output appears here.",
        wxDefaultPosition, wxDefaultSize,
        wxTE_MULTILINE | wxTE_READONLY);

    splitter->SplitHorizontally(editor, output, -FromDIP(170));
    splitter->SetMinimumPaneSize(FromDIP(90));

    auto* root = new wxBoxSizer(wxVERTICAL);
    root->Add(toolbarRow, 0, wxEXPAND | wxALL, FromDIP(10));
    root->Add(splitter, 1, wxEXPAND | wxLEFT | wxRIGHT | wxBOTTOM,
              FromDIP(10));
    rootPanel->SetSizer(root);
```

```
    Centre();
}
```

Listing 5.2: Layout example build

```
cmake_minimum_required(VERSION 4.1)
project(wxLayout LANGUAGES C CXX RC)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
add_subdirectory("${WXWIDGETS_ROOT}" "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL)
add_executable(wxLayout WIN32 main.cpp app.rc)
target_link_libraries(wxLayout PRIVATE wx::core wx::base)
```

Notice how almost no absolute geometry appears. The frame has an initial size, spacing uses `FromDIP()`, and the splitter receives the expandable share of the window.

5.15 DPI Awareness

High-DPI support is both an OS integration issue and a layout issue. On Windows the manifest participates in declaring awareness. In application code, avoid assuming that a constant number of physical pixels corresponds to the same perceived size on every monitor.

Use helpers such as `FromDIP()` for custom spacing, icon dimensions, margins, and hand-drawn geometry. Native controls and sizers will handle many metrics automatically when you do not override them unnecessarily.

5.16 Multi-Monitor Reality

A window can move between monitors with different scale factors. Avoid caching large amounts of pixel geometry as if it were permanent. If you draw custom content or maintain scale-dependent bitmaps, be prepared to recreate them when DPI changes.

Persist window position and size cautiously. A coordinate valid yesterday may be off-screen after a monitor is disconnected. Restore geometry with bounds checks and sensible fallbacks.

5.17 Localization and Layout

Translation can make labels dramatically longer. A row that assumes “Run” is always three letters is fragile. Sizers allow controls to request their natural size under the current translation.

This is another reason to avoid fixed dialog widths. Let the contents establish a minimum, provide growth where useful, and test at least one verbose translation if internationalization matters.

5.18 Layout Debugging Strategy

When a layout looks wrong, inspect it systematically:

1. Is the control parent the container you think it is?
2. Is the control actually added to the sizer?
3. Is the top-level sizer attached with `SetSizer()`?
4. Is proportion correct along the sizer’s main axis?
5. Is `wxEXPAND` needed on the cross axis?
6. Are hidden windows still reserving space intentionally?

7. Did a fixed minimum size constrain the result?
8. Are you fighting the sizer with manual `SetSize()` calls?

5.19 Layout Patterns for the ForgeVM Editor

5.19.1 Main workspace

A vertical root sizer can hold a command/filter row plus a central splitter. The splitter can place an editor above an output/diagnostic pane.

5.19.2 Settings dialog

Use a vertical root, a `wxNotebook` for categories if needed, and `wxFlexGridSizer` inside each page. Put OK/Cancel buttons at the bottom using the platform's standard dialog button sizer helpers.

5.19.3 Find bar

Use a horizontal box sizer: search text with proportion 1, then Previous, Next, Match Case, Regex, and Close. Hide/show the entire bar through the parent sizer and relayout the panel.

5.20 Chapter Review

Checklist

You are ready to continue when you can predict the effect of `proportion`, `wxEXPAND`, alignment, and border flags; build a form with `wxFlexGridSizer`; use nested box sizers without hard-coded coordinates; scale custom distances with `FromDIP()`; and choose a splitter when the user should control pane allocation.

CHAPTER 6

Event Handling Deep Dive

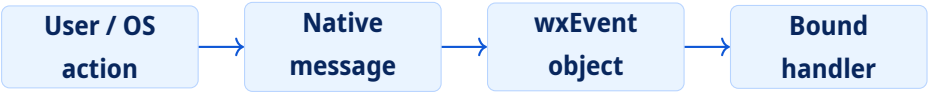
Chapter Outcomes

Understand how wxWidgets turns native messages and user actions into typed C++ events. Learn dynamic binding, handler signatures, event IDs, propagation, command events, update-UI events, timers, custom events, asynchronous notifications, and event-driven design patterns that keep a desktop program responsive and testable.

6.1 The Event Loop Is the Program's Rhythm

After initialization, a GUI application spends most of its lifetime waiting. The operating system reports mouse movement, key presses, window resizing, timers, paint requests, menu selections, process notifications, and many other messages. wxWidgets translates these into event objects and routes them to event handlers.

The conceptual flow is:



Event-driven programming therefore reverses the question a console programmer often asks. Instead of "what line executes next?", ask "what can happen next, and which object owns the response?"

6.2 wxEvtHandler

wxEvtHandler is the event-processing base used widely throughout the framework. Windows derive from it, but non-window event handlers can also participate in event-based designs.

wxEvtHandler

Purpose. Provides the machinery for binding and processing wxWidgets events.
Use it when. An object must receive framework events or custom application events.
Remember. Keep handlers short and delegate expensive work or domain operations to ordinary C++ services.

6.3 Dynamic Bind() versus Event Tables

wxWidgets supports traditional event tables and dynamic `Bind()` calls. Modern application code frequently prefers `Bind()` because bindings remain near construction/composition code and can bind member functions, lambdas, ranges of IDs, and specific event sources.

```
Bind(wxEVT_MENU, &MainFrame::OnOpen, this, wxID_OPEN);
button->Bind(wxEVT_BUTTON, &MainFrame::OnAssemble, this);
```

Event tables remain useful in existing codebases and in some coding styles. Understanding them is worthwhile, but do not mix styles randomly inside one class without a reason.

6.4 Handler Signatures

The event type determines the handler parameter. Examples:

```
void OnButton(wxCommandEvent& event);
void OnClose(wxCloseEvent& event);
void OnSize(wxSizeEvent& event);
void OnKeyDown(wxKeyEvent& event);
void OnTimer(wxTimerEvent& event);
void OnPaint(wxPaintEvent& event);
void OnUpdateUI(wxUpdateUIEvent& event);
```

The event object contains information specific to the event and also control-flow methods such as `Skip()` for many event-routing situations.

6.5 Command Events and Propagation

Buttons, menu items, check boxes, text controls, and many other controls generate wxCommandEvent-derived events. Command events have propagation behaviour that can allow a child-generated event to be handled by a parent when the child does not consume it.

This is powerful for component design. A custom panel can contain a button yet allow the frame to handle the command semantically. However, implicit propagation can also hide ownership if overused. Decide whether a component should translate low-level child events into a higher-level custom event or expose bindings intentionally.

Core Concept

A good component boundary does not force the main frame to know every internal button ID inside every panel. As UI sections grow, translate internal widget interactions into higher-level operations or custom events.

6.6 event.Skip()

Calling `event.Skip()` tells wxWidgets that your handler observed the event but did not want to prevent normal processing from continuing. Whether you should call it depends on the event and intended behaviour.

For example, a text-change handler may update application state and then skip so other interested handlers can continue. A close handler that vetoes closure should not skip into a path that defeats the veto logic.

Do not mechanically add `Skip()` to every handler. Understand the event's routing and default processing.

6.7 IDs as Routing Keys

When binding a menu or command event to a frame, the ID can identify the semantic command. Stock IDs should be used for stock semantics. Application IDs can begin beyond `wxID_HIGHEST`.

```
enum : int
{
    ID_Assemble = wxID_HIGHEST + 1,
    ID_Run,
    ID_ToggleOutput,
    ID_ShowSymbols
};
```

As the application grows, consider a command abstraction so the UI is not littered with ID-specific business logic.

6.8 Update UI Events: Derive Availability

wxUpdateUIEvent is a valuable wxWidgets mechanism for enabling, disabling, checking, or updating command UI based on current state.

Instead of manually remembering to disable Assemble every time the active document becomes invalid, derive availability:

```
void MainFrame::OnUpdateAssemble(wxUpdateUIEvent& event)
{
    event.Enable(document_ && document_->HasSavedPath()
                 && !buildRunner_.IsRunning());
}
```

The same semantic ID can keep menu and toolbar state consistent.

Why This Matters

Command availability is derived state. The more places that manually call `Enable(false)` and `Enable(true)`, the easier it becomes for UI surfaces to disagree. Update-UI handlers centralize the rule.

6.9 Timers

wxTimer delivers events at intervals without creating another thread. It is appropriate for lightweight periodic UI work: refresh a status indicator, coalesce updates, blink a caret-like custom indicator, or poll a cheap state when no better notification exists.

Timers are not background computation. The timer handler runs in the event loop. If it performs a two-second task, your interface freezes for two seconds.

6.10 Idle Events

Idle processing can perform small pieces of deferred work while the event queue is quiet. It is useful for incremental updates, but abusing idle events for continuous computation can consume CPU and make application behaviour difficult to reason about.

Prefer explicit events, timers, or worker tasks with completion notification. Use idle processing when the work naturally means "do a little when the GUI has nothing more urgent."

6.11 Keyboard Events and Accelerators

Text editors need careful keyboard design. Menu accelerators handle application commands such as Save or Assemble. Key events inside the editor handle editing-specific behaviour. Do not globally intercept every key and reimplement the native text control's editing semantics.

When a keyboard shortcut represents a command, prefer an accelerator/menu command path so the same operation is available through discoverable UI.

6.12 Mouse Events

Mouse events are essential for custom drawing, drag interactions, and specialized views. Ordinary native controls already implement their own pointer behaviour; bind low-level mouse events only when your component owns custom interaction.

Remember capture semantics if a drag must continue receiving mouse events after the pointer leaves the window. Release capture reliably, including cancellation paths.

6.13 Paint Events

Custom windows draw in response to `wxEVT_PAINT`. Do not paint arbitrary persistent graphics from a button handler using a temporary device context and expect them to survive. Instead, update component state, invalidate the relevant region, and render that state in the paint handler. Chapter 8 develops this model fully.

6.14 Custom Events

As components become independent, custom events provide a clean way to announce domain-relevant occurrences without exposing internal controls.

A custom event can communicate "build finished", "active symbol changed", or "document path changed" rather than "button 417 was clicked." This improves architecture because event consumers depend on meaning instead of implementation detail.

At a high level:

```
wxDECLARE_EVENT(EVT_BUILD_FINISHED, wxCommandEvent);
// In one .cpp file:
wxDEFINE_EVENT(EVT_BUILD_FINISHED, wxCommandEvent);
```

The producer can populate event fields or define a custom event class when richer typed data is required, then queue/post the event to the target handler.

6.15 Posting versus Processing Immediately

An event can be processed synchronously in the current call stack or queued for later dispatch. Queuing is often safer when crossing component or thread boundaries because the recipient handles the event in its normal event-loop context.

For worker threads, post a thread-safe notification to the GUI rather than directly manipulating controls. The official wxWidgets FAQ explicitly warns against using GUI functions from threads other than the main GUI thread.

6.16 CallAfter() and Main-Thread Re-entry

A common pattern is to perform non-GUI work in a worker and schedule a small lambda/member call back onto the event-loop thread. The important design idea is not the exact helper name but the ownership boundary: the worker produces a result, then the main thread applies it.

```
// Conceptual pattern from a worker completion path:
CallAfter([this, result = std::move(result)]() mutable {
    outputView_->ShowResult(result);
});
```

Ensure the receiving object will still exist when the deferred call runs. For long-lived services, use clear cancellation/shutdown rules rather than capturing raw pointers into unconstrained asynchronous work.

6.17 Complete Event Example

Listing 6.1: Button, text, timer, and update-UI events

```
#include <wx/wx.h>
#include <wx/timer.h>

namespace
{
enum : int
{
    ID_Assemble = wxID_HIGHEST + 1,
    ID_Timer
};
}

class MainFrame final : public wxFrame
{
public:
    MainFrame();

private:
    wxTextCtrl* input_{};
    wxTextCtrl* log_{};
    wxTimer timer_;
    int ticks_{};

    void OnAssemble(wxCommandEvent& event);
    void OnTextChanged(wxCommandEvent& event);
    void OnTimer(wxTimerEvent& event);
    void OnUpdateAssemble(wxUpdateUIEvent& event);
};

class App final : public wxApp
{
```

```
public:
    bool OnInit() override
    {
        auto* frame = new MainFrame();
        frame->Show();
        return true;
    }
};

wxIMPLEMENT_APP(App);

MainFrame::MainFrame()
    : wxFrame(nullptr, wxID_ANY, "Events Example",
              wxDefaultPosition, wxSize(760, 520)),
      timer_(this, ID_Timer)
{
    auto* panel = new wxPanel(this);
    input_ = new wxTextCtrl(panel, wxID_ANY, "source.asm");
    auto* assemble = new wxButton(panel, ID_Assemble, "Assemble");
    log_ = new wxTextCtrl(panel, wxID_ANY, "",
                          wxDefaultPosition, wxDefaultSize,
                          wxTE_MULTILINE | wxTE_READONLY);

    auto* row = new wxBoxSizer(wxHORIZONTAL);
    row->Add(input_, 1, wxRIGHT, FromDIP(8));
    row->Add(assemble, 0);

    auto* root = new wxBoxSizer(wxVERTICAL);
    root->Add(row, 0, wxEXPAND | wxALL, FromDIP(12));
    root->Add(log_, 1, wxEXPAND | wxLEFT | wxRIGHT | wxBOTTOM, FromDIP(12));
    panel->SetSizer(root);

    Bind(wxEVT_BUTTON, &MainFrame::OnAssemble, this, ID_Assemble);
    input_->Bind(wxEVT_TEXT, &MainFrame::OnTextChanged, this);
    Bind(wxEVT_TIMER, &MainFrame::OnTimer, this, ID_Timer);
```

```
    Bind(wxEVT_UPDATE_UI, &MainFrame::OnUpdateAssemble, this, ID_Assemble);

    timer_.Start(1000);
    Centre();
}

void MainFrame::OnAssemble(wxCommandEvent&)
{
    log_->AppendText("Assemble requested for: " + input_->GetValue() + "\n");
}

void MainFrame::OnTextChanged(wxCommandEvent& event)
{
    SetStatusText("Edited: " + input_->GetValue());
    event.Skip();
}

void MainFrame::OnTimer(wxTimerEvent&)
{
    ++ticks_;
    if ((ticks_ % 5) == 0)
        log_->AppendText("Timer: UI is still responsive.\n");
}

void MainFrame::OnUpdateAssemble(wxUpdateUIEvent& event)
{
    event.Enable(!input_->GetValue().Trim().empty());
}
```

Listing 6.2: Event example build

```
cmake_minimum_required(VERSION 4.1)
project(wxEvents LANGUAGES C CXX RC)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
set(CMAKE_CXX_EXTENSIONS OFF)
set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
add_subdirectory("${WXWIDGETS_ROOT}" "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL)
add_executable(wxEvents WIN32 main.cpp app.rc)
target_link_libraries(wxEvents PRIVATE wx::core wx::base)
```

This example uses a timer to demonstrate responsiveness, but the timer deliberately performs trivial work. The Assemble button's enabled state is derived from the text input using an update-UI handler.

6.18 Avoid Blocking the Event Loop

The most visible GUI defect is often not a crash; it is a frozen window. If a handler performs a long assembler invocation synchronously, the event loop cannot repaint, move the window, respond to Cancel, or process input.

Classify work:

Work	Main thread?	Strategy
Update label	Yes	Direct handler call
Open native file dialog	Yes	Modal UI operation
Parse tiny settings file	Usually yes	Keep simple if demonstrably fast
Assemble large project	No/blocking process	Asynchronous process execution
Index thousands of symbols	Worker	Post progress/result events
Paint custom view	Yes	Paint event; precompute expensive data elsewhere

6.19 Event Handler Design Rules

A professional handler typically performs four steps:

1. Extract relevant UI/event data.
2. Validate immediate preconditions.
3. Call an application/domain operation.
4. Reflect the result or initiate an asynchronous flow.

If a handler grows to hundreds of lines, it probably contains domain logic, file I/O, process management, or parsing that belongs elsewhere.

6.20 Chapter Review

Checklist

You should now understand dynamic binding, handler types, command propagation, `Skip()`, update-UI events, timers, custom events, main-thread rules, and the architectural reason to translate widget-level events into domain-level operations as the application grows.

CHAPTER 7

Data Views and Advanced Controls

Chapter Outcomes

Move beyond simple forms into data-rich desktop tools. Learn when to use list controls, tree controls, data-view controls and models, notebooks, splitters, property grids, styled text, and docking interfaces. Understand how model/view separation prevents large datasets and complex editor state from becoming inseparable from the widgets that display them.

7.1 From Controls to Views

A button can own almost all the state it needs. A symbol browser cannot. A real symbol browser may contain thousands of entries produced by parsing, assembling, or reading debug metadata. It may support sorting, filtering, grouping, lazy loading, and multiple presentations. The data must therefore exist independently from the widget.

This is the point where GUI architecture changes from "read values from controls" to "controls observe and manipulate models." wxWidgets offers both convenience controls that store their own simple data and richer model-aware classes for larger applications.

7.2 Choosing a Collection Control

Control	Best for	Avoid when
wxListBox	One-column string lists	You need multiple columns, icons, large virtual data, rich sorting
wxListCtrl	Traditional report/icon/list views	You want a stronger model/view abstraction for application data
wxTreeCtrl	Hierarchical items with a direct tree widget API	Data is large/complex enough to deserve a reusable model
wxDataViewCtrl	Structured data with columns and a model	A tiny fixed list would be simpler in a convenience control
wxDataViewListCtrl	Table-like data with convenient row storage	You need custom lazy models, tree hierarchy, or very large data

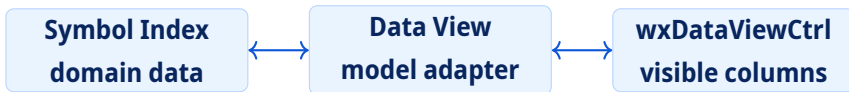
For the first ForgeVM symbol view, `wxDataViewListCtrl` is a good stepping stone: it gives columns and native data-view presentation without requiring a custom model on day one. If the symbol index later becomes large or shared among several views, migrate to a dedicated data model.

7.3 wxDataViewCtrl and Models

wxDataViewCtrl

Purpose. A data-oriented view supporting columns and model-backed rows/tree items.
Use it when. Displaying structured application data such as symbols, diagnostics, registers, sections, or build artifacts.
Remember. Keep the canonical data outside the control when it is shared, large, computed, or needs independent testing.

The conceptual relationship is:



The model adapter answers questions the view asks: how many children does an item have, what value belongs in a particular column, what type is the value, and can it be edited? The domain model remains free of GUI inheritance.

7.4 wxDataViewListCtrl: Convenience First

wxDataViewListCtrl

Purpose. Convenience data-view control with row-oriented storage managed by the widget.
Use it when. A moderate table that needs columns quickly without a custom model.
Remember. Do not make it the permanent database for data used elsewhere in the application.

A symbol table can create columns such as Symbol, Kind, Section, Address, and Size. Selection

events then identify the chosen row and allow another panel to show details.

Listing 7.1: Working symbol table with a details pane

```
#include <wx/wx.h>
#include <wx/dataview.h>
#include <wx/splitter.h>
#include <wx/treectrl.h>

class App final : public wxApp
{
public:
    bool OnInit() override;
};

class MainFrame final : public wxFrame
{
public:
    MainFrame();

private:
    wxDataViewListCtrl* symbols_{};
    wxTextCtrl* details_{};
    void PopulateSymbols();
    void OnSelection(wxDataViewEvent& event);
};

wxIMPLEMENT_APP(App);

bool App::OnInit()
{
    auto* frame = new MainFrame();
    frame->Show();
    return true;
}
```

```
MainFrame::MainFrame()
    : wxFrame(nullptr, wxID_ANY, "Data View Example",
              wxDefaultPosition, wxSize(900, 580))
{
    auto* panel = new wxPanel(this);
    auto* splitter = new wxSplitterWindow(panel, wxID_ANY);

    symbols_ = new wxDataViewListCtrl(
        splitter, wxID_ANY, wxDefaultPosition, wxDefaultSize,
        wxDV_ROW_LINES | wxDV_VERT_RULES);
    symbols_->AppendTextColumn("Symbol", wxDATAVIEW_CELL_INERT, 220);
    symbols_->AppendTextColumn("Kind", wxDATAVIEW_CELL_INERT, 120);
    symbols_->AppendTextColumn("Address", wxDATAVIEW_CELL_INERT, 130,
                               wxALIGN_RIGHT);

    details_ = new wxTextCtrl(
        splitter, wxID_ANY, "Select a symbol.",
        wxDefaultPosition, wxDefaultSize,
        wxTE_MULTILINE | wxTE_READONLY);

    splitter->SplitVertically(symbols_, details_, FromDIP(500));
    splitter->SetMinimumPaneSize(FromDIP(180));

    auto* root = new wxBoxSizer(wxVERTICAL);
    root->Add(splitter, 1, wxEXPAND | wxALL, FromDIP(10));
    panel->SetSizer(root);

    symbols_->Bind(wxEVT_DATAVIEW_SELECTION_CHANGED,
                   &MainFrame::OnSelection, this);

    PopulateSymbols();
    Centre();
}

void MainFrame::PopulateSymbols()
```

```
{
    symbols_->AppendItem({"main", "global", "0x00000000"});
    symbols_->AppendItem({"parse_loop", "local", "0x00000018"});
    symbols_->AppendItem({"buffer", "data", "0x00001000"});
    symbols_->AppendItem({"buffer_size", "equ", "4096"});
}

void MainFrame::OnSelection(wxDataViewEvent&)
{
    const int row = symbols_->ItemToRow(symbols_->GetSelection());
    if (row == wxNOT_FOUND)
        return;

    const auto symbol = symbols_->GetTextValue(row, 0);
    const auto kind = symbols_->GetTextValue(row, 1);
    const auto address = symbols_->GetTextValue(row, 2);

    details_->ChangeValue(wxString::Format(
        "Symbol: %s\nKind: %s\nAddress/Value: %s\n",
        symbol, kind, address));
}
```

Listing 7.2: Data-view example build

```
cmake_minimum_required(VERSION 4.1)
project(wxDataViews LANGUAGES C CXX RC)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
add_subdirectory("${WXWIDGETS_ROOT}" "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL)
```

```
add_executable(wxDataViews WIN32 main.cpp app.rc)
target_link_libraries(wxDataViews PRIVATE wx::core wx::base)
```

7.5 wxTreeCtrl for Project and Structure Browsing

wxTreeCtrl

Purpose. Native hierarchical tree of items.

Use it when. Project files, outline trees, include relationships, section/group navigation, or modest hierarchical data.

Remember. For a huge or multi-view hierarchy, prefer a model-driven data-view design.

A project tree should represent meaningful hierarchy, not merely mirror the filesystem blindly. For an assembler project, top-level groups might include Sources, Includes, Generated, and Build Artifacts even when their physical paths differ.

Use item data objects or a mapping to stable domain identifiers so selection can locate the corresponding project entity. Avoid treating display text as a unique key.

7.6 Notebooks and Page-Based Workspaces

wxNotebook

Purpose. Tabbed container with native tab appearance.

Use it when. A small set of peer pages such as Source, Symbols, Registers, and Documentation.

Remember. Hundreds of open files may need a more specialized notebook/document architecture and careful lifecycle management.

A notebook is easy to create but can become a hidden state machine if every page holds

different global controls. Keep page classes self-contained and let the frame coordinate active-document commands.

For multi-document editors, define a clear interface for editor pages: current path, modified state, save, caret position, and perhaps selected text. The frame should not downcast pages to unrelated implementation classes for every command.

7.7 wxSplitterWindow Revisited

Splitters are especially valuable when data views accompany a primary editor. Common patterns include:

- project tree | editor;
- editor / output;
- symbol list | symbol details;
- disassembly | register/memory inspector.

Nested splitters can create an IDE-like workspace, but each sash adds cognitive and layout complexity. Use docking (AUI) when panes need to float, hide, and rearrange dynamically rather than nesting many fixed splitters.

7.8 wxPropertyGrid

wxPropertyGrid

Purpose. Specialized property editor for named values grouped into categories.
Use it when. Inspecting/editing structured settings, object properties, build options, or metadata.
Remember. Simple forms are clearer when there are only a handful of fields.

Property grids are excellent for extensible tools because properties can be generated from metadata. They are less suitable for workflows requiring extensive explanation or complex multi-field validation; a custom settings page can communicate those relationships better.

Depending on how wxWidgets is built, property grid support may be a distinct component/target. Keep optional advanced controls isolated so the build can enable only what the application uses.

7.9 wxStyledTextCtrl: The Foundation of a Code Editor

wxStyledTextCtrl

Purpose. wxWidgets wrapper around the Scintilla editing component, providing source-editor features far beyond a normal text control.
Use it when. Programming-language editing, syntax styles, line numbers, markers, folding, indicators, autocomplete, and large editable source documents.
Remember. A simple form field or plain log does not need this complexity.

For the NASM editor, this is one of the most important classes in the book. It supports:

- line-number margins;
- syntax lexing/styling;
- markers for errors/breakpoints;
- indicators for ranges;
- brace matching;
- folding;
- autocomplete and call tips;
- configurable indentation and tabs;

- selection/caret APIs;
- file loading/saving support;
- event notifications for modifications and UI updates.

Do not configure every styling constant in the main-frame constructor. Create an `EditorView` class derived from or containing `wxStyledTextCtrl`, then give the class focused operations such as:

- `ConfigureNasmSyntax()` for lexer/style setup;
- `SetErrorMarkers()` for diagnostic presentation; and
- `ApplyEditorSettings()` for user-configurable editor policy.

7.10 Scintilla Lexers and NASM

The exact lexer and style constants available depend on the Scintilla version integrated with the wxWidgets build. For an assembly editor, first inspect the current `wx/stc/stc.h` constants and wxStyledTextCtrl documentation rather than copying decade-old lexer names. If the available assembly lexer does not meet the desired NASM semantics, you can still use container/custom styling or build a lightweight lexer layer that applies styles based on tokens.

Practical Tip

Treat syntax highlighting as presentation, not parsing. The assembler or a dedicated parser is authoritative about whether source is valid. A highlighter should remain fast and tolerant of incomplete code while the user is typing.

7.11 Markers for Diagnostics

Styled text markers are ideal for line-oriented diagnostics. Instead of coloring the entire editor background or inserting error text into the source, define a margin marker and place it on lines reported by NASM.

Indicators can underline exact token ranges, while markers can show an icon in the margin. Keep diagnostic objects in a model with file, line, column, severity, and message, then project them into the active editor view.

7.12 Docking with wxAUI

wxAuiManager

Purpose. Advanced User Interface manager for dockable/floating panes.
Use it when. An IDE-like workspace needs panes the user can dock, hide, resize, or rearrange.
Remember. For a fixed two-pane application, a splitter is simpler and easier to persist.

AUI can provide editor notebooks, project panes, output panes, symbol views, and tool windows. It adds substantial state and persistence complexity, so introduce it after the basic editor behaviour is stable.

A good evolution path is:

1. single frame + splitter;
2. notebook for multiple editor documents;
3. optional project/symbol panes;
4. AUI only when users genuinely benefit from rearrangement.

7.13 Data Refresh without Flicker

Large views should not be rebuilt on every keystroke. Decide what invalidates the data and update at the right granularity.

Examples:

- Current-line/column status: every caret move is fine.
- Symbol index: debounce or update after parse/assemble completion.
- Project tree: update when files change, not every editor event.
- Diagnostics: replace as a batch after a build result is parsed.

Where a model API supports item-level notification, use it instead of clearing and repopulating the entire view.

7.14 Selection Is Not Identity

A selected row index is transient. Sorting or filtering can move the same symbol to another row. Store stable IDs or pointers/handles with a clear lifetime rather than assuming row 17 will always represent `main`.

This principle generalizes to tree items, notebook pages, and diagnostics. The visible position is a view detail; application identity belongs in the model.

7.15 Virtual and Large Data

When data is extremely large, constructing one GUI object per item can be wasteful. wxWidgets offers virtual/model-driven approaches in several controls. Design the domain layer so data can be queried on demand and views do not need to own every string copy.

For a processor instruction atlas or huge symbol table, paging/filtering at the model layer may matter more than micro-optimizing widget calls.

7.16 Architecture for ForgeVM Data Views

A scalable direction is:

```
Assembler/Parser Output
        |
        v
DiagnosticModel ----> OutputView
SymbolIndex     ----> SymbolDataViewModel ----> SymbolView
ProjectModel    ----> ProjectTreeAdapter  ----> ProjectTree
DocumentModel   ----> EditorView
```

Each arrow represents an adapter or observer relationship, not necessarily inheritance.

7.17 Chapter Review

Checklist

You should now be able to choose between simple lists, trees, and model-driven data views; keep stable identity outside row positions; justify `wxStyledTextCtrl` for source editing; use markers/indicators as projections of diagnostics; and delay AUI until docking provides real workflow value rather than visual complexity.

CHAPTER 8

Graphics, Resources, DPI, and User Experience

Chapter Outcomes

Understand when to rely on native controls and when to draw custom content. Learn the wxDC painting model, buffering, colours, fonts, bitmaps, icons, DPI-aware geometry, themes, accessibility-minded interaction, clipboard and drag-and-drop basics, and the UX principles that make a technical desktop tool feel stable rather than merely functional.

8.1 Native First, Custom Where It Adds Value

wxWidgets is strongest when you let native controls perform ordinary control jobs. Repainting a button yourself to achieve a slightly different shade usually sacrifices more than it gains: keyboard semantics, accessibility, focus appearance, high-DPI handling, theme integration, and platform conventions are already solved by the native control.

Custom drawing belongs where the application owns a visualization: waveform, timeline, graph, memory map, instruction encoding diagram, minimap, custom ruler, or specialized editor decoration.

Core Concept

Use native widgets for interaction primitives; use custom drawing for domain visualization. This single rule prevents a large amount of unnecessary GUI engineering.

8.2 The Painting Model

A custom window paints when the framework sends a paint event. The operating system can invalidate your window for many reasons: resize, uncover, theme change, moving another window, or explicit refresh. Therefore the paint handler must be capable of reconstructing the visible content from application state.

Do not think of drawing as modifying a permanent canvas. Think of it as rendering a current model.

8.3 wxDC Families

wxDC is the abstract device-context interface for drawing. Specific contexts represent different destinations.

Class	Typical role
wxPaintDC	Drawing inside a paint event.
wxAutoBufferedPaintDC	Paint-event drawing with buffering to reduce flicker.
wxClientDC	Immediate client-area drawing outside normal paint flow; use carefully because content is not persistent.

wxMemoryDC	Draw into/select a bitmap in memory for off-screen composition.
wxPrinterDC	Printer-oriented drawing within the printing framework.

The book's default for custom controls is `wxAutoBufferedPaintDC` when repaint flicker is possible.

8.4 Paint State, Not History

Suppose a diagnostic chart contains five stages. Do not store "we drew stage 1, then stage 2" as the only representation. Store the stage data, then render all currently visible stages in `OnPaint()`.

When data changes:

```
model_.SetBuildStages(stages);
Refresh();   // invalidate; paint event will redraw from model state
```

This architecture survives minimization, resizing, occlusion, and display changes.

8.5 Complete Buffered Drawing Example

Listing 8.1: DPI-aware custom timeline panel

```
#include <wx/wx.h>
#include <wx/dcbuffer.h>

class TimelinePanel final : public wxPanel
{
public:
```

```
    explicit TimelinePanel(wxWindow* parent)
        : wxPanel(parent)
    {
        SetBackgroundStyle(wxBG_STYLE_PAINT);
        Bind(wxEVT_PAINT, &TimelinePanel::OnPaint, this);
        Bind(wxEVT_SIZE, &TimelinePanel::OnSize, this);
    }

private:
    void OnPaint(wxPaintEvent&)
    {
        wxAutoBufferedPaintDC dc(this);
        dc.SetBackground(wxBrush(GetBackgroundColour()));
        dc.Clear();

        const wxSize size = GetClientSize();
        const int margin = FromDIP(24);
        const int y = size.y / 2;

        dc.SetPen(wxPen(wxColour(35, 95, 190), FromDIP(2)));
        dc.DrawLine(margin, y, size.x - margin, y);

        const wxString labels[] = {"Edit", "Save", "Assemble", "Link", "Run"};
        const int count = static_cast<int>(std::size(labels));
        const int available = size.x - (2 * margin);

        for (int i = 0; i < count; ++i)
        {
            const int x = margin + (available * i) / (count - 1);
            const int radius = FromDIP(7);

            dc.SetBrush(wxBrush(wxColour(46, 134, 255)));
            dc.SetPen(*wxTRANSPARENT_PEN);
            dc.DrawCircle(x, y, radius);
```

```
            const wxSize textSize = dc.GetTextExtent(labels[i]);
            dc.SetTextForeground(GetForegroundColour());
            dc.DrawText(labels[i], x - textSize.x / 2, y + FromDIP(16));
        }
    }

    void OnSize(wxSizeEvent& event)
    {
        Refresh();
        event.Skip();
    }
};

class App final : public wxApp
{
public:
    bool OnInit() override
    {
        auto* frame = new wxFrame(nullptr, wxID_ANY, "Custom Drawing",
                                  wxDefaultPosition, wxSize(860, 420));
        auto* panel = new TimelinePanel(frame);
        frame->Centre();
        frame->Show();
        return true;
    }
};

wxIMPLEMENT_APP(App);
```

Listing 8.2: Custom drawing example build

```
cmake_minimum_required(VERSION 4.1)
project(wxGraphics LANGUAGES C CXX RC)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
set(CMAKE_CXX_EXTENSIONS OFF)
set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source")
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
add_subdirectory("${WXWIDGETS_ROOT}" "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL)
add_executable(wxGraphics WIN32 main.cpp app.rc)
target_link_libraries(wxGraphics PRIVATE wx::core wx::base)
```

The example calculates geometry from the current client size and uses `FromDIP()` for logical dimensions. It does not cache pixels that become invalid after a resize.

8.6 Pens, Brushes, and Text

A wxPen describes line colour, width, and style. A wxBrush describes fill. Text uses the device context’s current font and foreground colour.

Prefer semantic colour roles in your own application model instead of scattering RGB constants:

```
struct Theme
{
    wxColour editorBackground;
    wxColour editorText;
    wxColour diagnosticError;
    wxColour diagnosticWarning;
    wxColour accent;
};
```

A theme object can then be regenerated when the system appearance changes.

8.7 Fonts

For ordinary controls, inherit platform fonts whenever possible. Hard-coding a font family across the entire application can make the UI look foreign and fail on systems where the family is absent.

Source editors are different: a user-configurable monospace font is part of the tool. Store a `wxFont` or font description in editor settings and apply it to `wxStyledTextCtrl` styles.

8.8 Bitmaps, Icons, and Bundles

Desktop applications need scalable visual assets. Modern wxWidgets APIs can work with multi-resolution bitmap/icon representations. The practical principle is to provide sufficient resolution and let the framework/platform choose an appropriate image for the current DPI rather than stretching one tiny raster asset indefinitely.

For a toolbar, prefer simple, high-contrast icons that remain recognizable at multiple scales. Test disabled, dark-theme, and high-DPI appearance.

8.9 DPI: Coordinate Systems Matter

DPI awareness has two layers:

1. **Platform declaration/integration.** On Windows, the manifest participates in how Windows scales and reports the application.
2. **Application geometry.** Custom spacing, line widths, bitmap sizes, and drawing coordinates should be converted from logical/DIP-like units where appropriate.

The most robust code minimizes explicit pixel assumptions. Sizers and native controls already solve much of the problem. Custom graphics should use conversion helpers and current client

dimensions.

8.10 Dark Mode and System Appearance

Dark appearance support differs by platform and wxWidgets version/port. Do not implement dark mode as "set every control background to black." Native controls may have platform-specific theming rules, and overriding colours can produce unreadable text or inconsistent disabled states.

A sensible policy is:

- let native controls use native appearance unless customization is necessary;
- theme custom-drawn surfaces and editor styles explicitly;
- derive colours from semantic roles rather than literal names;
- test focus, disabled state, selection, hover, and error colours;
- avoid relying on colour alone to convey diagnostics.

8.11 Accessibility Is Architecture

A technically advanced editor that cannot be operated by keyboard or understood by assistive technology has a quality defect. Native controls help, but application design still matters.

Key practices:

- maintain logical tab order;
- provide keyboard alternatives to pointer-only operations;
- associate labels with controls where supported;
- make command names descriptive;

- preserve focus visibility;
- do not encode error/warning distinction only through red/yellow;
- keep status changes available as text, not just animation or icon changes.

8.12 Clipboard

The clipboard is essential for an editor and diagnostics tool. wxWidgets provides a cross-platform clipboard abstraction. For ordinary source text, the editor control already exposes copy/cut/paste behaviour. For custom views, provide copy commands that place meaningful textual data on the clipboard rather than only a bitmap when possible.

A symbol row copied as:

```
main    global    .text    0x00000000
```

is more useful for developers than a screenshot of that row.

8.13 Drag and Drop

Drag-and-drop can improve workflows such as opening source files by dropping them onto the editor. Implement it as an additional path to the same open-document operation, not as a separate file-loading system.

Validate dropped paths exactly as paths chosen through the File menu. Keep the domain operation independent of the interaction mechanism.

8.14 Printing and Export

wxWidgets includes printing support, but printing a source editor is not simply dumping the current screen. Printing has pagination, margins, fonts, headers, line wrapping, and printer device metrics. If ForgeVM eventually needs printable listings, design a print document model separate from screen layout.

For generated technical reports, exporting to HTML/PDF through a dedicated formatting path may be more appropriate than screen-control printing.

8.15 UX for Developer Tools

Developer tools prioritize information density but still require hierarchy. Good density is not the absence of whitespace; it is the absence of wasted interaction.

A strong editor workspace normally has:

- one dominant source area;
- commands arranged by frequency;
- output visible when relevant and hideable when not;
- diagnostics that navigate to source;
- status information that does not interrupt typing;
- persistent layout preferences;
- predictable keyboard shortcuts;
- no modal progress dialog for operations that can run asynchronously.

8.16 Error Messages

An error message should answer three questions:

1. What failed?
2. What can the user do next?
3. Where can detailed diagnostics be found?

Bad: *Error 2*.
Better: *NASM could not be started. Verify the assembler path in Settings > Toolchain. The attempted path was ...*

For compiler/assembler failures, show the actual tool output in the output pane and use a short status/banner to summarize the result.

8.17 Non-Blocking Feedback

Use modal message boxes for decisions that truly require immediate acknowledgement. Use status bars, output panes, banners, or inline diagnostics for normal build failures and background completion.

A developer expects assembly errors to be part of the editing loop, not exceptional application failures. Treat them as data.

8.18 Visual Consistency without Fighting the Platform

Define consistency in layers:

- consistent spacing scale;
- consistent command names and icon meaning;

- consistent panel hierarchy;
- consistent diagnostic severity model;
- consistent editor theme;
- native platform differences in standard controls and dialogs.

This balance preserves both product identity and platform familiarity.

8.19 Chapter Review

Checklist

You should now understand paint-event rendering, buffered drawing, DPI-aware geometry, semantic colours, native-first control strategy, theme limitations, accessibility fundamentals, and the difference between an error as application failure versus a diagnostic as normal developer-tool data.

CHAPTER 9

Structuring Large Applications

Chapter Outcomes

Scale beyond a demonstration program. Learn how to separate domain logic from GUI classes, organize commands and documents, persist settings, run subprocesses and background work without freezing the event loop, manage cancellation and shutdown, create reusable panels, test non-GUI logic, log effectively, and isolate platform-specific code behind narrow boundaries.

9.1 The God-Frame Failure Mode

A beginner's wxWidgets program often begins with one `MainFrame` class. That is appropriate. The failure occurs when the frame remains the destination for every new responsibility: file parsing, configuration, process launching, theme data, recent-file management, document content, symbol indexing, build output, project scanning, and networking.

A 5,000-line frame compiles, but its internal coupling makes every change dangerous. The solution is not to invent a complicated framework of your own. Use ordinary C++ composition and a few stable boundaries.

Core Concept

GUI classes should own GUI composition and interaction. Domain/service classes should own application behaviour that can exist without a visible window. The bridge between them is a small set of commands, models, callbacks, or events.

9.2 A Practical Layered Architecture

For ForgeVM's editor, a useful shape is:

```
Presentation
  App, MainFrame, EditorView, OutputView, SymbolView, SettingsDialog

Application services
  DocumentManager, CommandRouter, BuildRunner, RecentFiles, Settings

Domain/data
  Document, Diagnostic, Symbol, BuildResult, Project

Infrastructure
  FileSystem helpers, Process adapter, Config backend, platform adapters
```

These are responsibilities, not mandatory directories. Keep the structure as small as the current application can justify.

9.3 Document Model

A source document should have an identity independent from its tab page. At minimum:

```
struct DocumentState
{
    wxString path;
    bool modified{};
```

```
    wxString encoding{"UTF-8"};
};
```

The actual text may live in the styled text control for performance/convenience in an early editor, but the application should still expose document operations rather than allowing every menu handler to manipulate the control directly.

A richer document model can support file watching, reload detection, encoding, line-ending policy, external modifications, and independent parse results.

9.4 Document Manager

A `DocumentManager` answers application-level questions:

- Which documents are open?
- Which document is active?
- Is a path already open?
- Can all documents close safely?
- Which documents are modified?

It should not know the pixel position of a notebook tab. The presentation layer maps documents to editor views.

9.5 Command Model

Menus, toolbar buttons, accelerators, command palette entries, and context menus are multiple presentations of commands. Define commands semantically.

```
enum class Command
{
```

```
        NewFile,
        OpenFile,
        Save,
        SaveAs,
        Assemble,
        Run,
        ToggleOutput,
        GotoLine
    };
```

A simple command router can map these to operations. wx IDs remain necessary for wxWidgets event wiring, but they do not need to become your domain model.

9.6 Persistent Settings with wxConfig

wxConfig and related config classes abstract persistent key/value settings using a platform-appropriate backend.

Good candidates include:

- NASM executable path;
- preferred output format;
- editor font and tab width;
- window size/position;
- splitter sash position;
- recently opened files;
- last project directory.

Avoid storing transient state that should be recomputed. Version settings deliberately so future releases can migrate old values.

```
auto* config = wxConfigBase::Get();
config->Write("Toolchain/NasmPath", nasmPath);
config->Write("Editor/TabWidth", tabWidth);
config->Flush();
```

At startup:

```
wxString nasmPath;
long tabWidth = 4;
config->Read("Toolchain/NasmPath", &nasmPath, "nasm");
config->Read("Editor/TabWidth", &tabWidth, 4L);
```

Wrap these details in a `Settings` class rather than scattering config keys throughout the UI.

9.7 Paths with wxFileName and wxStandardPaths

wxFileName handles cross-platform path components, normalization, extension changes, and path composition. wxStandardPaths locates user configuration/data directories and executable-related locations according to platform conventions.

Never assume "current working directory" means "application directory." The working directory can depend on how the program was launched.

9.8 Asynchronous Processes

An assembler editor must run external tools. Calling a synchronous process API from a button handler freezes the event loop until the tool exits. Instead, launch asynchronously and receive completion notification.

wxProcess integrates child processes with the event system and can redirect standard streams. `wxExecute()` can launch commands synchronously or asynchronously depending on flags.

A robust process service should own:

- command/executable path and arguments;
- current PID/process object;
- redirected output streams;
- cancellation/termination policy;
- exit status;
- output decoding;
- conversion of tool output into `Diagnostic` objects.

Important Pitfall

Command construction is a portability and security boundary. Do not concatenate arbitrary user-controlled text into a shell command without understanding quoting and whether a shell is involved. Prefer APIs/argument construction that preserves argument boundaries when available, and quote executable/file paths carefully on every supported platform.

9.9 Background Threads

Some work is better performed in-process on a worker: symbol indexing, expensive parsing, hashing, project scans, compression, or analysis. The wxWidgets GUI must remain on its GUI thread.

A safe architecture is:

1. Copy or move the input required by the worker into worker-owned data.
2. Run computation without touching wxWindow objects.

3. Produce a result object.
4. Queue/post a completion event to a live GUI coordinator.
5. Apply the result to controls on the main thread.

Shutdown must cancel or join workers before their event target disappears.

9.10 std::jthread and Cooperative Cancellation

Modern C++ offers `std::jthread` and `std::stop_token`, which are useful for cooperative worker cancellation. The worker function should check the token at sensible boundaries rather than relying on unsafe forced termination.

```
std::jthread worker_;

void SymbolIndexer::Start(std::filesystem::path root)
{
    worker_ = std::jthread(
        [this, root = std::move(root)](std::stop_token stop)
        {
            auto result = BuildIndex(root, stop);
            if (stop.stop_requested())
                return;

            // Queue result to the GUI coordinator here.
        });
}
```

Do not capture UI pointers casually. Define who owns the worker, who receives completion, and what happens when the window closes mid-task.

9.11 Thread-Safe Result Delivery

wxWidgets supports posting events to event handlers. For richer results, a custom event can own/carry a shared or moved data object depending on your design. Keep the data type independent from the control that will display it.

For frequent progress updates, throttle notifications. Posting thousands of GUI events per second simply moves the performance problem into the event queue.

9.12 Cancellation and Shutdown

Closing a large application is an asynchronous edge case. Consider:

- unsaved documents;
- running assembler processes;
- background parsing/indexing;
- pending queued GUI callbacks;
- configuration writes;
- temporary files.

Define a shutdown order. A common sequence is: ask documents to close; stop accepting new commands; request worker cancellation; terminate/wait for child processes according to policy; detach/stop event producers; persist state; allow frame destruction.

9.13 Reusable Panels

Turn complex regions into classes:

```
class OutputPanel final : public wxPanel
{
public:
    explicit OutputPanel(wxWindow* parent);

    void Clear();
    void AppendProcessOutput(const wxString& text);
    void ShowDiagnostics(const std::vector<Diagnostic>& diagnostics);
};
```

This is preferable to exposing the underlying `wxTextCtrl*` globally. The panel interface speaks application language.

9.14 Testing Strategy

A GUI test has value, but most logic should not require a GUI to test.

Extract functions/classes for:

- NASM command-line construction;
- diagnostic parsing;
- settings validation;
- recent-file list policy;
- path normalization;
- project scanning;
- symbol model transformation.

These can be exercised by ordinary C++ unit tests. GUI tests then focus on wiring, lifetime, command state, and visible behaviour.

9.15 CMake Architecture for Larger Projects

Instead of compiling every source directly into the executable, create a library for core logic:

```
add_library(ForgeEditorCore
    src/DiagnosticParser.cpp
    src/CommandBuilder.cpp
    src/Settings.cpp
)

target_compile_features(ForgeEditorCore PUBLIC cxx_std_23)

target_link_libraries(ForgeEditorCore
    PUBLIC
    wx::base
)

add_executable(ForgeEditor WIN32
    src/App.cpp
    src/MainFrame.cpp
    src/EditorView.cpp
    app.rc
)

target_link_libraries(ForgeEditor
    PRIVATE
    ForgeEditorCore
    wx::core
    wx::stc
)
```

Keeping core logic out of `wx::core` where it does not need GUI facilities improves build structure and testability.

9.16 Logging as an Engineering Surface

A developer tool should record enough context to diagnose failures:

- application version/build;
- selected toolchain executable paths;
- exact assembler arguments (with sensitive data excluded if applicable);
- working directory;
- process exit code;
- major settings migration events;
- unexpected exceptions/assertions;
- plugin/module loading if added later.

Logs should complement user-facing diagnostics rather than replace them.

9.17 Error Types

Not every failure deserves the same UI.

Failure	Example	Presentation
Expected domain error	NASM reports syntax error	Diagnostic list + source marker
Configuration error	NASM path missing	Inline/settings message + actionable fix
Transient I/O	File temporarily locked	Non-destructive error with retry option
Programmer invariant	null required service	Assertion/log; fix code
Fatal startup	required resource cannot initialize	Clear error and abort startup safely

9.18 Internationalization

wxWidgets includes internationalization support. Even if ForgeVM begins English-only, avoid architecture that makes future translation unnecessarily difficult. User-visible strings should be centralized/translatable rather than constructed from fragments with language-dependent grammar.

Paths, assembler output, and source code are not the same as UI strings. Keep tool output verbatim when accuracy matters, and translate the surrounding application explanation if localization is later added.

9.19 Platform-Specific Code

Cross-platform does not mean “never use #ifdef”. It means platform differences are narrow and deliberate.

Bad:

```
// dozens of #ifdef blocks spread through MainFrame.cpp
```

Better:

```
class PlatformIntegration
{
public:
    static void RevealInFileManager(const wxString& path);
    static void ConfigureNativeTheme(wxWindow& window);
};
```

Its implementation can be separated by platform when necessary.

9.20 Performance: Measure the Right Layer

wxWidgets overhead is rarely the main performance issue in an editor. More likely bottlenecks include full-document rescans, excessive view repopulation, synchronous file/process work, thousands of queued events, or repeatedly rebuilding large layout trees.

Instrument operations that users feel: file-open latency, keystroke-to-highlight latency, build start time, diagnostic parse time, large symbol-view refresh, and memory consumption with many documents.

9.21 Chapter Review

Checklist

You should now be able to split a large desktop program into presentation, application services, domain data, and infrastructure; keep long-running work off the event loop; persist settings through a dedicated layer; test non-GUI logic; and define explicit shutdown/cancellation rules before concurrency becomes complex.

CHAPTER 10

Building a Practical NASM Editor

Chapter Outcomes

This chapter turns the preceding concepts into a complete development tool. You will design a small but extensible NASM editor around wxStyledTextCtrl, implement file commands, unsaved-document protection, syntax colouring, line navigation, diagnostic markers, asynchronous NASM execution, output capture, and command-state updates. The result is intentionally modest enough to understand in one sitting but structured so it can grow into a ForgeVM development surface.

10.1 Define the Product Before the Widgets

The easiest way to produce a fragile editor is to begin by placing controls on a frame and deciding what the program means later. Start instead with a small product contract. Version 1 of the editor will provide the following vertical slice:

- create, open, edit, and save one assembly source document;
- present NASM-oriented syntax highlighting and line numbers;
- navigate to a one-based source line;

- choose a NASM object format such as `win64`, `elf64`, or `macho64`;
- execute NASM without freezing the user interface;
- capture standard output and standard error;
- mark lines reported by diagnostics;
- prevent accidental loss of modified text;
- keep menu and button availability synchronized with application state.

This definition is small enough to finish, yet it touches nearly every architectural topic required by a serious native editor. More advanced features—multiple documents, project trees, code intelligence, debugger integration, binary views, and ForgeVM-specific tools—can be layered on without changing the fundamentals.

Why This Matters

The purpose of the first editor is not to compete with a mature IDE. It is to create a trustworthy architectural nucleus. A small editor with clean document state, process boundaries, diagnostics, and command routing is a much better foundation than a visually impressive prototype whose frame directly owns every responsibility.

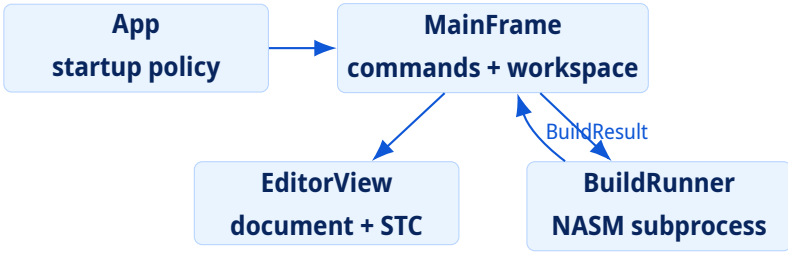
10.2 Architecture at a Glance

The sample uses four primary pieces:

Component	Responsibility
App	Framework startup, application identity, top-window creation.

`MainFrame`	Command surface, workspace composition, user-flow coordination, status feedback.
`EditorView`	Text editing, NASM lexer configuration, document path, save point, line navigation, diagnostic markers.
`BuildRunner`	External NASM process lifetime, redirected output, polling, completion callback.

The split is deliberate. The frame coordinates actions but does not implement syntax colouring or process plumbing itself. This keeps the visible shell from becoming a "god frame" as features accumulate.



10.3 Project Layout

The complete sample is stored with the book as a directly buildable project:

```
examples/nasm_editor/
|-- CMakeLists.txt
|-- app.rc
|-- App.cpp
|-- MainFrame.hpp
|-- MainFrame.cpp
|-- EditorView.hpp
|-- EditorView.cpp
```

```
|-- BuildRunner.hpp
`-- BuildRunner.cpp
```

The book keeps the sample flat to make copying easy. A growing production repository can move headers under `include/forgevm/editor/`, source files under `src/`, and tests under `tests/` without changing the design.

10.4 CMake: Add the Components You Actually Use

A text editor based on Scintilla needs the wxWidgets STC component in addition to core and base. The source-tree build used throughout this guide therefore links `wx::stc` explicitly.

Listing 10.1: Complete NASM editor build

```
cmake_minimum_required(VERSION 4.1)

project(ForgeVMNasmEditor LANGUAGES C CXX RC)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3" CACHE PATH "wxWidgets source tree"
    )

set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
set(wxBUILD_DEMOS OFF CACHE BOOL "" FORCE)

add_subdirectory(
    "${WXWIDGETS_ROOT}"
    "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL
```

```
)

add_executable(ForgeVMNasmEditor WIN32
    App.cpp
    MainFrame.cpp
    MainFrame.hpp
    EditorView.cpp
    EditorView.hpp
    BuildRunner.cpp
    BuildRunner.hpp
    app.rc
)

target_link_libraries(ForgeVMNasmEditor
    PRIVATE
    wx::core
    wx::base
    wx::stc
)

if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
    target_compile_options(ForgeVMNasmEditor PRIVATE
        -Wall
        -Wextra
        -Wpedantic
    )
endif()
```

Component-based linking

Link the highest-level wxWidgets components that your target actually requires. A target using `wxStyledTextCtrl` should link `wx::stc`; a target using AUI docking should add `wx::aui`; a target using property grids should add the corresponding property-grid

component. This is clearer than manually naming archive files and allows the wxWidgets CMake targets to propagate required includes, definitions, and dependent libraries.

On Windows, keep `app.rc` in the executable target:

Listing 10.2: Windows resource and manifest

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>
```

10.5 Application Startup

The application object remains intentionally small:

Listing 10.3: Application entry point

```
#include "MainFrame.hpp"

#include <wx/app.h>

class App final : public wxApp
{
public:
    bool OnInit() override
    {
        if (!wxApp::OnInit())
            return false;

        SetAppName("ForgeVMNasmEditor");
        SetVendorName("ForgeVM");

        auto* frame = new MainFrame();
        SetTopWindow(frame);
        frame->Show();
```

```
        return true;
    }
};

wxIMPLEMENT_APP(App);
```

Two details are worth carrying into larger programs. First, the application sets a stable app and vendor name. Configuration backends can use these values to derive native storage locations. Second, `SetTopWindow()` makes application intent explicit and gives framework services a reliable primary window.

10.6 EditorView: Put Editing Knowledge in the Editor

wxStyledTextCtrl wraps the Scintilla editing component and is the natural starting point for a programming editor. It provides text storage, undo/redo, markers, margins, lexers, styles, folding, search primitives, indicators, multiple selections, and many lower-level editor features without forcing the application to reinvent a text engine.

Listing 10.4: EditorView declaration

```
#pragma once

#include <wx/stc/stc.h>
#include <wx/string.h>

class EditorView final : public wxStyledTextCtrl
{
public:
    explicit EditorView(wxWindow* parent);

    bool OpenFile(const wxString& path);
    bool Save();
    bool SaveAs(const wxString& path);
```

```
    [[nodiscard]] bool HasPath() const noexcept { return !path_.empty(); }
    [[nodiscard]] bool IsModified() const { return GetModify(); }
    [[nodiscard]] const wxString& Path() const noexcept { return path_; }

    void NewDocument();
    void GotoLineOneBased(int line);
    void ClearDiagnostics();
    void MarkErrorLine(int line);

private:
    wxString path_;

    void ConfigureEditor();
    void ConfigureNasmLexer();
};
```

The view stores the current path because that path is part of the document-facing editing surface. A later multi-document architecture may move path and dirty state into a separate document model, but the first version benefits from keeping the vertical slice compact.

10.6.1 Editor fundamentals

The constructor configures behaviour before any document is loaded:

- spaces are used instead of literal tabs;
- line wrapping is disabled for source code;
- a monospaced font is selected;
- margin 0 displays line numbers;
- margin 1 is reserved for diagnostic symbols;
- the caret line receives a subtle highlight.

Listing 10.5: Editor implementation

```
#include "EditorView.hpp"

#include <algorithm>
#include <wx/colour.h>
#include <wx/font.h>

namespace
{
constexpr int LineNumberMargin = 0;
constexpr int DiagnosticMargin = 1;
constexpr int ErrorMarker = 1;
}

EditorView::EditorView(wxWindow* parent)
    : wxStyledTextCtrl(parent, wxID_ANY)
{
    ConfigureEditor();
    ConfigureNasmLexer();
}

void EditorView::ConfigureEditor()
{
    SetUseTabs(false);
    SetTabWidth(4);
    SetIndent(4);
    SetWrapMode(wxSTC_WRAP_NONE);
    SetViewEOL(false);

    StyleSetFont(wxSTC_STYLE_DEFAULT,
                 wxFontInfo(11).Family(wxFONTFAMILY_TELETYPE));
    StyleClearAll();

    SetMarginType(LineNumberMargin, wxSTC_MARGIN_NUMBER);
    SetMarginWidth(LineNumberMargin,
```

```
                    TextWidth(wxSTC_STYLE_LINENUMBER, "_99999"));

    SetMarginType(DiagnosticMargin, wxSTC_MARGIN_SYMBOL);
    SetMarginWidth(DiagnosticMargin, FromDIP(16));
    SetMarginSensitive(DiagnosticMargin, true);

    MarkerDefine(ErrorMarker, wxSTC_MARK_CIRCLE,
                 wxColour(255, 255, 255),
                 wxColour(200, 45, 45));

    SetCaretLineVisible(true);
    SetCaretLineBackground(wxColour(242, 247, 255));
}

void EditorView::ConfigureNasmLexer()
{
    SetLexer(wxSTC_LEX_ASM);

    StyleSetForeground(wxSTC_ASM_COMMENT, wxColour(92, 115, 92));
    StyleSetForeground(wxSTC_ASM_COMMENTBLOCK, wxColour(92, 115, 92));
    StyleSetForeground(wxSTC_ASM_NUMBER, wxColour(145, 70, 20));
    StyleSetForeground(wxSTC_ASM_STRING, wxColour(20, 120, 75));
    StyleSetForeground(wxSTC_ASM_CHARACTER, wxColour(20, 120, 75));
    StyleSetForeground(wxSTC_ASM_OPERATOR, wxColour(70, 70, 90));
    StyleSetForeground(wxSTC_ASM_CPUINSTRUCTION, wxColour(25, 80, 190));
    StyleSetForeground(wxSTC_ASM_MATHINSTRUCTION, wxColour(25, 80, 190));
    StyleSetForeground(wxSTC_ASM_REGISTER, wxColour(120, 45, 150));
    StyleSetForeground(wxSTC_ASM_DIRECTIVE, wxColour(0, 110, 130));
    StyleSetForeground(wxSTC_ASM_DIRECTIVEOPERAND, wxColour(0, 110, 130));

    SetKeyWords(0,
        "mov lea push pop call ret jmp je jne jz jnz ja jae jb jbe "
        "cmp test add sub inc dec neg and or xor not shl shr sal sar "
        "imul mul idiv div nop int syscall cmovz cmovnz setz setnz");
```

```
    SetKeyWords(1,
        "fadd fsub fmul fdiv paddb paddw paddd paddq addps addpd "
        "vaddps vaddpd vxorps vxorpd");

    SetKeyWords(2,
        "al ah ax eax rax bl bh bx ebx rbx cl ch cx ecx rcx "
        "dl dh dx edx rdx sil si esi rsi dil di edi rdi "
        "bpl bp ebp rbp spl sp esp rsp r8 r8d r8w r8b r9 r9d r9w r9b "
        "r10 r10d r11 r11d r12 r12d r13 r13d r14 r14d r15 r15d "
        "xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7 ymm0 ymm1 zmm0 zmm1");

    SetKeyWords(3,
        "section segment global extern bits default rel abs org equ times "
        "db dw dd dq dt do dy dz resb resw resd resq align struc endstruc "
        "%define %xdefine %assign %if %ifdef %ifndef %else %elif %endif "
        "%include %macro %endmacro");
}

bool EditorView::OpenFile(const wxString& path)
{
    if (!LoadFile(path))
        return false;

    path_ = path;
    SetSavePoint();
    EmptyUndoBuffer();
    ClearDiagnostics();
    return true;
}

bool EditorView::Save()
{
    if (path_.empty())
        return false;
```

```
    if (!SaveFile(path_))
        return false;

    SetSavePoint();
    return true;
}

bool EditorView::SaveAs(const wxString& path)
{
    if (!SaveFile(path))
        return false;

    path_ = path;
    SetSavePoint();
    return true;
}

void EditorView::NewDocument()
{
    ClearAll();
    path_.clear();
    SetSavePoint();
    EmptyUndoBuffer();
    ClearDiagnostics();
}

void EditorView::GotoLineOneBased(int line)
{
    const int zeroBased = std::max(0, line - 1);
    GotoLine(zeroBased);
    EnsureCaretVisible();
    SetFocus();
}

void EditorView::ClearDiagnostics()
```

```
{
    MarkerDeleteAll(ErrorMarker);
}

void EditorView::MarkErrorLine(int line)
{
    if (line > 0)
        MarkerAdd(line - 1, ErrorMarker);
}
```

Practical Tip

Use `FromDIP()` for interface dimensions that should scale with display density. Do not treat a literal pixel count as a platform-independent physical size. For source text itself, select a sensible point-size font and let the platform font machinery handle rendering.

10.7 NASM Syntax Highlighting

Scintilla includes an assembly lexer exposed through wxSTC as `wxSTC_LEX_ASM`. The lexer recognizes categories such as comments, numbers, strings, registers, CPU instructions, directives, and directive operands. The application chooses the colours and populates keyword sets.

A practical NASM editor should not confuse syntax colouring with semantic analysis. A lexer can identify token categories quickly, but it does not prove that an instruction is valid for the selected architecture, that an operand size is legal, or that a symbol exists. Keep colouring fast and local. Deeper diagnostics belong to an assembler, parser, language service, or ForgeVM analysis layer.

Lexer versus semantic intelligence

Lexer: categorizes text for fast visual presentation.

Assembler/parser: understands grammar and validates source structure.

Semantic engine: resolves symbols, instruction constraints, architecture modes, and project context.

Editor: presents all three layers without confusing one for another.

10.8 Save Points and Modified State

Scintilla tracks a save point. Calling `SetSavePoint()` records the current text state as clean; `GetModify()` tells you whether edits have moved away from that state. This is more reliable than maintaining a separate boolean in every handler.

The sequence is simple:

1. load or save a file successfully;
2. call `SetSavePoint();`
3. display an asterisk in the frame title when `GetModify()` becomes true;
4. before New, Open, or Close, ask the user what to do if the document is modified.

Do not ask after destroying state

Prompt before replacing or destroying the current document. Once a control has been cleared or a frame is half-way through destruction, recovery logic becomes much more complicated. Data-loss prevention belongs at the command boundary.

10.9 The Main Frame as Coordinator

The frame declaration describes the command surface and the services it coordinates:

Listing 10.6: MainFrame declaration

```
#pragma once

#include "BuildRunner.hpp"

#include <wx/frame.h>

class EditorView;
class wxChoice;
class wxSplitterWindow;
class wxStyledTextEvent;
class wxTextCtrl;

class MainFrame final : public wxFrame
{
public:
    MainFrame();

private:
    enum : int
    {
        ID_Assemble = wxID_HIGHEST + 1,
        ID_Format,
        ID_GotoLine
    };

    EditorView* editor_{};
    wxTextCtrl* output_{};
    wxChoice* format_{};
    wxSplitterWindow* splitter_{};
```

```
    BuildRunner buildRunner_;
    wxString nasmPath_{"nasm"};

    void BuildMenu();
    void BuildWorkspace();
    void BindEvents();
    void UpdateTitle();

    void OnNew(wxCommandEvent& event);
    void OnOpen(wxCommandEvent& event);
    void OnSave(wxCommandEvent& event);
    void OnSaveAs(wxCommandEvent& event);
    void OnAssemble(wxCommandEvent& event);
    void OnGotoLine(wxCommandEvent& event);
    void OnClose(wxCloseEvent& event);
    void OnUpdateSave(wxUpdateUIEvent& event);
    void OnUpdateAssemble(wxUpdateUIEvent& event);
    void OnEditorChange(wxStyledTextEvent& event);

    bool SaveDocument(bool forceSaveAs);
    bool ConfirmDiscardChanges();
    void HandleBuildResult(BuildResult result);
    void ParseDiagnosticLines(const wxString& text);
};
```

The implementation builds three visible areas: a small command row, the source editor, and a read-only output pane separated by wxSplitterWindow. Menus expose the same core commands as keyboard shortcuts and buttons.

Listing 10.7: MainFrame implementation

```
#include "MainFrame.hpp"
#include "EditorView.hpp"

#include <algorithm>
```

```
#include <utility>
#include <wx/choice.h>
#include <wx/filedlg.h>
#include <wx/filename.h>
#include <wx/menu.h>
#include <wx/msgdlg.h>
#include <wx/numdlg.h>
#include <wx/panel.h>
#include <wx/regex.h>
#include <wx/sizer.h>
#include <wx/splitter.h>
#include <wx/statusbr.h>
#include <wx/textctrl.h>
#include <wx/toolbar.h>

MainFrame::MainFrame()
    : wxFrame(nullptr, wxID_ANY, "ForgeVM NASM Editor",
              wxDefaultPosition, wxSize(1100, 760)),
      buildRunner_([this](BuildResult result)
      {
          HandleBuildResult(std::move(result));
      })
{
    BuildMenu();
    BuildWorkspace();
    BindEvents();

    CreateStatusBar(2);
    SetStatusWidths({-1, FromDIP(180)});
    SetStatusText("Ready", 0);
    SetStatusText("NASM", 1);

    UpdateTitle();
    Centre();
}
```

```
void MainFrame::BuildMenu()
{
    auto* file = new wxMenu();
    file->Append(wxID_NEW, "&New\tCtrl-N");
    file->Append(wxID_OPEN, "&Open...\tCtrl-O");
    file->AppendSeparator();
    file->Append(wxID_SAVE, "&Save\tCtrl-S");
    file->Append(wxID_SAVEAS, "Save &As...");
    file->AppendSeparator();
    file->Append(wxID_EXIT, "E&xit\tAlt-F4");

    auto* build = new wxMenu();
    build->Append(ID_Assemble, "&Assemble\tF7");

    auto* navigate = new wxMenu();
    navigate->Append(ID_GotoLine, "&Go to Line...\tCtrl-G");

    auto* bar = new wxMenuBar();
    bar->Append(file, "&File");
    bar->Append(build, "&Build");
    bar->Append(navigate, "&Navigate");
    SetMenuBar(bar);
}

void MainFrame::BuildWorkspace()
{
    auto* panel = new wxPanel(this);

    auto* commandRow = new wxBoxSizer(wxHORIZONTAL);
    commandRow->Add(new wxStaticText(panel, wxID_ANY, "Output format:"),
                    0, wxALIGN_CENTER_VERTICAL | wxRIGHT, FromDIP(6));

    format_ = new wxChoice(panel, ID_Format);
    format_->Append("win64");
```

```
    format_->Append("elf64");
    format_->Append("macho64");
    format_->SetStringSelection("win64");
    commandRow->Add(format_, 0, wxRIGHT, FromDIP(8));

    auto* assemble = new wxButton(panel, ID_Assemble, "Assemble (F7)");
    commandRow->Add(assemble, 0);
    commandRow->AddStretchSpacer();

    splitter_ = new wxSplitterWindow(
        panel, wxID_ANY, wxDefaultPosition, wxDefaultSize,
        wxSP_LIVE_UPDATE | wxSP_3D);

    editor_ = new EditorView(splitter_);
    output_ = new wxTextCtrl(
        splitter_, wxID_ANY, "",
        wxDefaultPosition, wxDefaultSize,
        wxTE_MULTILINE | wxTE_READONLY | wxTE_DONTWRAP);

    splitter_->SplitHorizontally(editor_, output_, -FromDIP(190));
    splitter_->SetMinimumPaneSize(FromDIP(90));

    auto* root = new wxBoxSizer(wxVERTICAL);
    root->Add(commandRow, 0, wxEXPAND | wxALL, FromDIP(8));
    root->Add(splitter_, 1, wxEXPAND | wxLEFT | wxRIGHT | wxBOTTOM,
              FromDIP(8));
    panel->SetSizer(root);
}

void MainFrame::BindEvents()
{
    Bind(wxEVT_MENU, &MainFrame::OnNew, this, wxID_NEW);
    Bind(wxEVT_MENU, &MainFrame::OnOpen, this, wxID_OPEN);
    Bind(wxEVT_MENU, &MainFrame::OnSave, this, wxID_SAVE);
    Bind(wxEVT_MENU, &MainFrame::OnSaveAs, this, wxID_SAVEAS);
```

```
    Bind(wxEVT_MENU, &MainFrame::OnAssemble, this, ID_Assemble);
    Bind(wxEVT_BUTTON, &MainFrame::OnAssemble, this, ID_Assemble);
    Bind(wxEVT_MENU, &MainFrame::OnGotoLine, this, ID_GotoLine);
    Bind(wxEVT_MENU, [this](wxCommandEvent&) { Close(true); }, wxID_EXIT);
    Bind(wxEVT_CLOSE_WINDOW, &MainFrame::OnClose, this);

    Bind(wxEVT_UPDATE_UI, &MainFrame::OnUpdateSave, this, wxID_SAVE);
    Bind(wxEVT_UPDATE_UI, &MainFrame::OnUpdateAssemble, this, ID_Assemble);

    editor_->Bind(wxEVT_STC_CHANGE, &MainFrame::OnEditorChange, this);
}

void MainFrame::UpdateTitle()
{
    wxString name = editor_->HasPath()
        ? wxFileName(editor_->Path()).GetFullName()
        : "Untitled.asm";

    if (editor_->IsModified())
        name += " *";

    SetTitle(name + " - ForgeVM NASM Editor");
}

void MainFrame::OnNew(wxCommandEvent&)
{
    if (!ConfirmDiscardChanges())
        return;

    editor_->NewDocument();
    output_->Clear();
    UpdateTitle();
}

void MainFrame::OnOpen(wxCommandEvent&)
```

```
{
    if (!ConfirmDiscardChanges())
        return;

    wxFileDialog dialog(
        this, "Open Assembly Source", {}, {},
        "Assembly source (*.asm;*.s)|*.asm;*.s|All files (*.*)|*.*",
        wxFD_OPEN | wxFD_FILE_MUST_EXIST);

    if (dialog.ShowModal() != wxID_OK)
        return;

    if (!editor_->OpenFile(dialog.GetPath()))
    {
        wxMessageBox("Could not open the selected file.", "Open",
                     wxOK | wxICON_ERROR, this);
        return;
    }

    output_->Clear();
    UpdateTitle();
}

void MainFrame::OnSave(wxCommandEvent&)
{
    SaveDocument(false);
}

void MainFrame::OnSaveAs(wxCommandEvent&)
{
    SaveDocument(true);
}

bool MainFrame::SaveDocument(bool forceSaveAs)
{
```

```
    if (!forceSaveAs && editor_->HasPath())
    {
        if (!editor_->Save())
        {
            wxMessageBox("Could not save the file.", "Save",
                         wxOK | wxICON_ERROR, this);
            return false;
        }
        UpdateTitle();
        return true;
    }

    wxFileDialog dialog(
        this, "Save Assembly Source", {}, "untitled.asm",
        "Assembly source (*.asm)|*.asm|All files (*.*)|*.*",
        wxFD_SAVE | wxFD_OVERWRITE_PROMPT);

    if (dialog.ShowModal() != wxID_OK)
        return false;

    if (!editor_->SaveAs(dialog.GetPath()))
    {
        wxMessageBox("Could not save the file.", "Save As",
                     wxOK | wxICON_ERROR, this);
        return false;
    }

    UpdateTitle();
    return true;
}

bool MainFrame::ConfirmDiscardChanges()
{
    if (!editor_->IsModified())
        return true;
```

```
    const int answer = wxMessageBox(
        "The current source has unsaved changes. Save them first?",
        "Unsaved Changes",
        wxYES_NO | wxCANCEL | wxYES_DEFAULT | wxICON_WARNING,
        this);

    if (answer == wxCANCEL)
        return false;
    if (answer == wxYES)
        return SaveDocument(false);
    return true;
}

void MainFrame::OnAssemble(wxCommandEvent&)
{
    if (buildRunner_.IsRunning())
        return;

    if (!editor_->HasPath() || editor_->IsModified())
    {
        if (!SaveDocument(false))
            return;
    }

    editor_->ClearDiagnostics();
    output_->Clear();
    output_->AppendText("Starting NASM...\n");

    wxFileName object(editor_->Path());
#ifdef __WXMSW__
    object.SetExt("obj");
#else
    object.SetExt("o");
#endif
```

```
    const wxString format = format_->GetStringSelection();
    if (!buildRunner_.Start(nasmPath_, editor_->Path(),
                            object.GetFullPath(), format))
    {
        output_->AppendText("Could not start NASM. Check the executable path.\n"
    );
        return;
    }

    SetStatusText("Assembling...", 0);
}

void MainFrame::HandleBuildResult(BuildResult result)
{
    if (!result.output.empty())
        output_->AppendText(result.output);

    ParseDiagnosticLines(result.output);

    if (result.exitCode == 0)
    {
        output_->AppendText("\nAssembly completed successfully.\n");
        SetStatusText("Assembly succeeded", 0);
    }
    else
    {
        output_->AppendText(wxString::Format(
            "\nNASM exited with code %d.\n", result.exitCode));
        SetStatusText("Assembly failed", 0);
    }
}

void MainFrame::ParseDiagnosticLines(const wxString& text)
{
```

```
    // Common NASM diagnostic shape: path:line: error: message
    wxRegEx pattern("^.*:([0-9]+):[[:space:]]*(error|warning):", wxRE_ADVANCED);

    wxArrayString lines = wxSplit(text, '\n', '\0');
    for (const auto& line : lines)
    {
        if (!pattern.Matches(line))
            continue;

        unsigned long number = 0;
        if (pattern.GetMatch(line, 1).ToULong(&number))
            editor_->MarkErrorLine(static_cast<int>(number));
    }
}

void MainFrame::OnGotoLine(wxCommandEvent&)
{
    const long line = wxGetNumberFromUser(
        "Enter a one-based line number:", "Line:", "Go to Line",
        editor_->GetCurrentLine() + 1,
        1,
        std::max(1, editor_->GetLineCount()),
        this);

    if (line > 0)
        editor_->GotoLineOneBased(static_cast<int>(line));
}

void MainFrame::OnClose(wxCloseEvent& event)
{
    if (buildRunner_.IsRunning())
    {
        const int answer = wxMessageBox(
            "NASM is still running. Close the editor and stop the process?",
            "Build Running",
```

```
            wxYES_NO | wxNO_DEFAULT | wxICON_WARNING,
            this);

        if (answer != wxYES)
        {
            event.Veto();
            return;
        }
    }

    if (!ConfirmDiscardChanges())
    {
        event.Veto();
        return;
    }

    event.Skip();
}

void MainFrame::OnUpdateSave(wxUpdateUIEvent& event)
{
    event.Enable(editor_ && editor_->IsModified());
}

void MainFrame::OnUpdateAssemble(wxUpdateUIEvent& event)
{
    event.Enable(editor_ && !buildRunner_.IsRunning());
}

void MainFrame::OnEditorChange(wxStyledTextEvent& event)
{
    UpdateTitle();
    SetStatusText(wxString::Format(
        "Line %d, Column %d",
        editor_->GetCurrentLine() + 1,
```

```
            editor_->GetColumn(editor_->GetCurrentPos()) + 1), 1);
        event.Skip();
    }
```

10.9.1 Why bind commands in one place?

A command such as Assemble may be triggered from a menu item, toolbar button, keyboard accelerator, command palette, or future project tree. The underlying operation should still have one state machine. The sample binds both the menu item and the button to the same `OnAssemble()` handler.

As the application grows, move command logic out of the frame into command/service objects and let multiple UI surfaces invoke the same operation. Chapter 9 introduced that separation; the sample demonstrates the smallest useful version.

10.10 Update UI: Availability Is Derived State

The Save command is enabled only when the editor is modified. Assemble is disabled while a previous NASM invocation is running. Instead of manually chasing every transition and calling `Enable()` in many handlers, the sample binds `wxEVT_UPDATE_UI` and derives availability from current state.

This pattern scales well:

```
void MainFrame::OnUpdateSave(wxUpdateUIEvent& event)
{
    event.Enable(editor_ && editor_->IsModified());
}

void MainFrame::OnUpdateAssemble(wxUpdateUIEvent& event)
{
    event.Enable(editor_ && !buildRunner_.IsRunning());
```

```
}
```

Why This Matters

Derived command state prevents contradictions. A menu item and toolbar button do not need separate rules; both reflect the same capability. This becomes essential when an editor has dozens or hundreds of commands.

10.11 Run NASM without Freezing the Interface

Executing an assembler synchronously on the UI thread is tempting because the code looks simple. It is also the wrong default. Even a fast command can stall on antivirus scanning, network storage, a child process, a large source tree, or a misconfiguration. The event loop must remain free to paint, move, cancel, and close windows.

The sample uses wxProcess with redirected streams and the asynchronous form of `wxExecute()`, selected with the `wxEXEC_ASYNC` flag. A timer drains output while the process runs, and `wxEVT_END_PROCESS` reports completion.

Listing 10.8: BuildRunner declaration

```
#pragma once

#include <functional>
#include <wx/process.h>
#include <wx/string.h>
#include <wx/timer.h>

struct BuildResult
{
    int exitCode{-1};
    wxString output;
};
```

```
class BuildRunner final : public wxEvtHandler
{
public:
    using Completion = std::function<void(BuildResult)>;

    explicit BuildRunner(Completion completion);
    ~BuildRunner() override;

    BuildRunner(const BuildRunner&) = delete;
    BuildRunner& operator=(const BuildRunner&) = delete;

    bool Start(const wxString& nasmPath,
               const wxString& sourcePath,
               const wxString& outputPath,
               const wxString& format);

    [[nodiscard]] bool IsRunning() const noexcept { return process_ != nullptr; }


private:
    wxProcess* process_{};
    long pid_{};
    wxTimer pollTimer_;
    wxString captured_;
    Completion completion_;

    void OnPoll(wxTimerEvent& event);
    void OnEndProcess(wxProcessEvent& event);
    void DrainStreams();
    void Drain(wxInputStream* stream, const wxString& prefix = {});
};
```

Listing 10.9: BuildRunner implementation

```
#include "BuildRunner.hpp"

#include <array>
#include <utility>
#include <wx/utils.h>

namespace
{
wxString Quote(const wxString& value)
{
    wxString result = value;
    result.Replace("\"", "\\\"");
    return "\"" + result + "\"";
}
}

BuildRunner::BuildRunner(Completion completion)
    : pollTimer_(this), completion_(std::move(completion))
{
    Bind(wxEVT_TIMER, &BuildRunner::OnPoll, this);
    Bind(wxEVT_END_PROCESS, &BuildRunner::OnEndProcess, this);
}

BuildRunner::~BuildRunner()
{
    pollTimer_.Stop();

    if (process_ && pid_ != 0)
        wxProcess::Kill(pid_, wxSIGTERM, wxKILL_CHILDREN);

    delete process_;
}

bool BuildRunner::Start(const wxString& nasmPath,
                        const wxString& sourcePath,
```

```
                          const wxString& outputPath,
                          const wxString& format)
{
    if (IsRunning())
        return false;

    captured_.clear();
    process_ = new wxProcess(this);
    process_->Redirect();

    const wxString command =
        Quote(nasmPath) + " -f " + format + " " +
        Quote(sourcePath) + " -o " + Quote(outputPath);

    pid_ = wxExecute(command, wxEXEC_ASYNC, process_);
    if (pid_ == 0)
    {
        delete process_;
        process_ = nullptr;
        return false;
    }

    pollTimer_.Start(40);
    return true;
}

void BuildRunner::OnPoll(wxTimerEvent&)
{
    DrainStreams();
}

void BuildRunner::OnEndProcess(wxProcessEvent& event)
{
    pollTimer_.Stop();
    DrainStreams();
```

```
    BuildResult result;
    result.exitCode = event.GetExitCode();
    result.output = std::move(captured_);

    delete process_;
    process_ = nullptr;
    pid_ = 0;

    if (completion_)
        completion_(std::move(result));
}

void BuildRunner::DrainStreams()
{
    if (!process_)
        return;

    Drain(process_->GetInputStream());
    Drain(process_->GetErrorStream());
}

void BuildRunner::Drain(wxInputStream* stream, const wxString& prefix)
{
    if (!stream)
        return;

    std::array<char, 4096> buffer{};

    while (stream->CanRead())
    {
        stream->Read(buffer.data(), buffer.size() - 1);
        const auto count = stream->LastRead();
        if (count == 0)
            break;
```

```
            captured_ += prefix;
            captured_ += wxString::FromUTF8(buffer.data(), count);
        }
    }
```

10.11.1 Why redirect output?

Compiler and assembler diagnostics are part of the development experience. Redirecting stdout/stderr lets the editor:

- display the raw tool output for transparency;
- parse line numbers and severity;
- add diagnostic markers beside source lines;
- eventually build a structured Problems view;
- retain logs for reproducible bug reports.

10.11.2 Command construction and quoting

Paths containing spaces are normal on Windows and common everywhere. Never concatenate an unquoted executable path and assume it will remain one argument. The sample contains a small quoting helper because `wxExecute()` receives a command string in this form.

For a production tool, prefer an argument-oriented process abstraction where possible, centralize platform quoting rules, and test deliberately with paths containing spaces, non-ASCII characters, and shell-sensitive characters.

A process launcher is a security boundary

Do not pass untrusted project text directly into a shell command. Treat executable paths and argument lists as structured data. If the application later exposes custom build commands, make the choice to invoke a shell explicit and document the security consequences.

10.12 From Text Diagnostics to Source Markers

NASM commonly emits diagnostics containing a source path, line number, severity, and message. The sample demonstrates a deliberately small regular expression that extracts a line number from a familiar shape and calls `EditorView::MarkErrorLine()`.

The marker API is ideal for editor annotations because the marker follows the Scintilla document line model instead of being painted manually. A richer implementation can reserve marker numbers for errors, warnings, breakpoints, execution position, bookmarks, and code-review notes.

Concern	Suggested primitive	Design note
Error line	STC marker	Persistent visual symbol in a margin.
Warning span	STC indicator	Underline or highlight a character range.
Breakpoint	STC marker	Distinct marker number and click-sensitive margin.
Search result	STC indicator	Multiple ranges can remain visible simultaneously.

Current execution	Marker + line background	Use a separate marker identity from errors.
Inline message	Annotation or companion panel	Keep long text out of the source itself.

10.13 Output Pane Design

A read-only wxTextCtrl is sufficient for the first version. If the output becomes large or needs clickable links, severity styling, folding, or incremental filtering, move to a richer control. The important design rule is that build output is a view of build-session state, not the build engine itself.

Eventually introduce a `BuildSession` model containing:

- start and finish timestamps;
- command/tool identity;
- exit code;
- structured diagnostics;
- raw output;
- cancellation state;
- produced artifacts.

This model can feed a text pane, Problems view, history panel, or automated test without rerunning NASM.

10.14 Go to Line

The sample uses `wxGetNumberFromUser()` for a compact Go to Line command. It passes one-based values to the user and converts to Scintilla's zero-based line indexing in one method. Centralizing the conversion avoids repeated off-by-one errors.

A future command palette can reuse the same navigation operation without knowing how Scintilla indexes lines.

10.15 Close Behaviour and Process Lifetime

Closing an editor is a multi-state operation. Ask these questions in this order:

1. Is an external build still running?
2. Does the current document contain unsaved changes?
3. Can pending background work be cancelled safely?
4. Are there settings or window geometry that should be persisted?
5. Is it safe to allow the close event to continue?

The sample vetoes the close event when the user declines either build termination or unsaved-change handling. Only after these decisions does it call `event.Skip()` and allow normal destruction.

10.16 What the First Version Intentionally Does Not Do

Good architecture includes explicit non-goals. The sample does not yet attempt:

- multiple open documents;
- project/workspace management;

- semantic completion;
- macro/include navigation;
- debugger or emulator integration;
- configurable themes;
- persistent user settings;
- structured diagnostic lists;
- asynchronous file indexing;
- plugin loading.

Each is valuable, but adding them before the core document/build loop is reliable would hide foundational mistakes under more code.

10.17 Evolution Path toward a ForgeVM Editor

A sensible sequence is shown below.

Stage	Capability	Architectural change
1	Single NASM document	Current sample: one frame, one editor, one build runner.
2	Multiple documents	Add document objects and notebook/editor tabs; frame coordinates active document only.
3	Workspace tree	Introduce project model and filesystem watcher; tree displays model state.

4	Structured diagnostics	Parse tool output into typed diagnostics and feed a data-view model.
5	Toolchain profiles	Persistent configuration for NASM path, format, architecture, include paths, output directory.
6	ForgeVM assembler	Replace/augment NASM adapter with ForgeVM tool adapter behind the same build interface.
7	Debug/run	Add process session model, registers/memory views, breakpoints, execution markers.
8	Language intelligence	Incremental parser/index, symbols, completion, navigation, hover information.
9	Extensible workbench	Dockable panes, command registry, reusable service interfaces, optional plugin boundary.

10.18 A Better Build Abstraction for the Next Iteration

Do not let the editor know that every assembler command is NASM forever. Introduce a narrow interface:

```
struct AssembleRequest
{
    std::filesystem::path source;
    std::filesystem::path output;
    std::string objectFormat;
    std::vector<std::filesystem::path> includePaths;
};

struct Diagnostic
{
    enum class Severity { Note, Warning, Error };
```

```
    Severity severity{};
    std::filesystem::path file;
    int line{};
    int column{};
    std::string message;
};

struct AssembleResult
{
    int exitCode{};
    std::string rawOutput;
    std::vector<Diagnostic> diagnostics;
};
```

The GUI can then depend on a service that accepts `AssembleRequest` and eventually returns `AssembleResult`. One implementation invokes NASM. Another can call a ForgeVM assembler library directly. The UI does not need to be redesigned when the backend changes.

GUI framework versus domain architecture

Use wxWidgets for windows, events, native integration, timers, process notifications, dialogs, and desktop behaviour. Use ordinary modern C++ types for assembler requests, diagnostics, documents, project models, and testable domain logic. The strongest wxWidgets applications are not “all wxWidgets”; they are clean C++ applications whose presentation layer uses wxWidgets well.

10.19 Testing the Editor

A first manual test matrix should include more than the happy path.

Editor acceptance checklist

- Start with no document and create a new source file.
- Type text, verify the title marks the document modified, then save it.
- Open a path containing spaces and non-ASCII characters.
- Cancel Open and Save As dialogs; state should remain unchanged.
- Attempt New/Open/Close with unsaved text and exercise Yes, No, and Cancel paths.
- Assemble valid source and confirm exit code 0 plus object creation.
- Assemble invalid source and confirm raw diagnostics plus line markers.
- Configure a missing NASM executable and verify a useful error is shown.
- Start a build and attempt to close the editor while the process is active.
- Resize aggressively and verify editor/output panes remain usable.
- Test 100%, 150%, and 200% display scaling on Windows where available.
- Build Debug and Release from clean directories.

Automated tests should focus first on non-UI services: diagnostic parsing, path derivation, command creation, configuration loading, and future document models. UI automation is valuable later, but the easiest code to test is code that was not unnecessarily coupled to widgets in the first place.

10.20 Performance Expectations

Do not optimize the frame construction of a one-document editor. Measure the operations that can actually become expensive:

- opening and saving multi-megabyte source files;

- syntax styling on large documents;
- incremental parsing/indexing;
- rendering thousands of diagnostics;
- filesystem scans over large workspaces;
- process output bursts;
- project-tree refreshes.

Scintilla is designed for editor workloads, but application code can still create latency by rebuilding whole models, performing synchronous I/O on the main thread, or flooding the event queue with redundant updates.

10.21 Security and Trust Boundaries

Developer tools process untrusted input routinely: source files, project files, compiler messages, paths, and external executables. Treat the editor as an engineering tool, not a toy.

- Do not execute a project-supplied command silently.
- Make toolchain paths explicit and inspectable.
- Avoid shell expansion unless the user intentionally requested shell semantics.
- Normalize and validate generated output paths.
- Do not interpret compiler output as markup without escaping it.
- Keep update/download mechanisms outside the first editor until authenticity and rollback are designed.

10.22 Packaging the First ForgeVM Editor

For an internal tool, reproducibility matters more than installer polish. Record:

- exact wxWidgets version;
- compiler/toolchain identity;
- CMake generator and version;
- static/shared configuration;
- NASM version used for validation;
- operating-system versions exercised;
- example source files that should assemble successfully and fail predictably.

Create a small `VALIDATION.md` alongside releases. A build that "worked on one machine" is not yet a reproducible development tool.

10.23 Chapter Review

Checklist

You should now be able to explain and implement:

- why `wxStyledTextCtrl` is preferable to a basic text control for source editing;
- how save points provide a reliable modified-state signal;
- how menus and buttons can share one command path;
- why `wxEVT_UPDATE_UI` is useful for command availability;
- how to launch NASM asynchronously and capture its streams;
- how diagnostic line numbers become editor markers;

- why the GUI should depend on an assembler abstraction rather than NASM command syntax forever;
- what should be moved out of the frame as the editor grows.

The book's destination is a beginning

The completed sample is deliberately the first credible version of the ForgeVM editor, not its final form. The value of the preceding chapters is that the next features can now be added on top of understood patterns: native windows, constraint-based layout, event routing, structured data views, asynchronous work, persistent state, and narrow platform boundaries.

CHAPTER A

Build Patterns and CMake Recipes

This appendix collects build configurations used frequently while learning or deploying wxWidgets applications. Treat each recipe as a starting point, not as a reason to stop understanding what CMake is doing.

A.1 Minimal Source-Tree Build

Build Recipe

Use this when wxWidgets source is available locally and you want the library configuration to be part of the application build.

```
cmake_minimum_required(VERSION 4.1)
project(MyWxApp LANGUAGES C CXX RC)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(WXWIDGETS_ROOT "E:/CPPLIB/wxWidgets3.3.3"
    CACHE PATH "wxWidgets source tree")
```

```
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
set(wxBUILD_SAMPLES OFF CACHE BOOL "" FORCE)
set(wxBUILD_TESTS OFF CACHE BOOL "" FORCE)
set(wxBUILD_DEMOS OFF CACHE BOOL "" FORCE)

add_subdirectory(
    "${WXWIDGETS_ROOT}"
    "${CMAKE_BINARY_DIR}/wxwidgets"
    EXCLUDE_FROM_ALL
)

add_executable(MyWxApp WIN32 main.cpp app.rc)
target_link_libraries(MyWxApp PRIVATE wx::core wx::base)
```

A.2 Add Styled Text, AUI, and Property Grid

```
target_link_libraries(MyWxApp PRIVATE
    wx::core
    wx::base
    wx::stc
    wx::aui
    wx::propgrid
)
```

Keep component use intentional. A small utility does not need to drag editor, AUI, networking, or XML resource code into its architecture merely because those facilities exist.

A.3 Static wxWidgets, Dynamic Compiler Runtime

```
set(wxBUILD_SHARED OFF CACHE BOOL "" FORCE)
```

This selects static wxWidgets libraries when wxWidgets is built as part of the project. It does *not* mean every runtime and system library becomes statically linked. That distinction is healthy: decide library configuration and runtime deployment separately.

A.4 Shared wxWidgets

```
set(wxBUILD_SHARED ON CACHE BOOL "" FORCE)
```

A shared build produces wxWidgets DLLs/shared objects that must be deployed with the application or found through a correct installation/runtime search path. Shared builds can speed iteration when the same wxWidgets build serves many executables, but deployment becomes a versioned dependency problem.

A.5 Static libgcc and libstdc++ under MinGW

If you deliberately want the GNU compiler support and standard C++ library linked statically while leaving Windows system libraries dynamic:

```
if(MINGW)
    target_link_options(MyWxApp PRIVATE
        -static-libgcc
        -static-libstdc++
    )
endif()
```

Do not add global `-static` merely because “static” sounds self-contained. It changes resolu-

tion for a much wider set of libraries and can expose CRT/thread-runtime constraints. First make the ordinary build correct; then choose deployment policy intentionally.

A.6 Warnings

```
if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
    target_compile_options(MyWxApp PRIVATE
        -Wall -Wextra -Wpedantic
    )
elseif(MSVC)
    target_compile_options(MyWxApp PRIVATE /W4 /permissive-)
endif()
```

Warnings are target properties. Avoid dumping flags globally when a repository contains third-party source whose warning policy you do not control.

A.7 Sanitizer Configuration for Your Own Code

GUI programs can benefit from AddressSanitizer and UndefinedBehaviorSanitizer, but availability and runtime behaviour depend on platform/toolchain. Keep sanitizer options behind an explicit project option and apply them to your own targets.

```
option(FORGE_ENABLE_SANITIZERS "Enable ASan/UBSan" OFF)

if(FORGE_ENABLE_SANITIZERS AND
   CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
    target_compile_options(MyWxApp PRIVATE
        -fsanitize=address,undefined
        -fno-omit-frame-pointer
    )
    target_link_options(MyWxApp PRIVATE
        -fsanitize=address,undefined
```

```
        )
    endif()
```

A.8 Debug and Release with Ninja

```
cmake -S . -B build-debug -G Ninja -DCMAKE_BUILD_TYPE=Debug
cmake --build build-debug -j 20

cmake -S . -B build-release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build-release -j 20
```

Never rely on a build folder whose history you no longer understand when diagnosing ABI, CRT, static/shared, or manifest problems. A new build directory is cheap evidence.

A.9 Ninja Multi-Config

Where available, a multi-config generator keeps several configurations under one generated tree:

```
cmake -S . -B build -G "Ninja Multi-Config"
cmake --build build --config Debug
cmake --build build --config Release
```

Use whichever model your team understands consistently. The important rule is that configuration identity remains visible and reproducible.

A.10 Installed wxWidgets Package

An installed/package workflow depends on how wxWidgets was installed and which CMake package interface that installation provides. Keep the package discovery explicit and verify imported targets available in that environment rather than copying paths from another machine.

A common configuration-package style is conceptually:

```
find_package(wxWidgets CONFIG REQUIRED COMPONENTS core base)
add_executable(MyWxApp WIN32 main.cpp app.rc)
target_link_libraries(MyWxApp PRIVATE wx::core wx::base)
```

If an installation exposes the older module variables instead, follow the package's documented interface. Do not mix the two styles blindly.

A.11 CMake Presets

Presets make compiler/generator/cache choices reviewable and repeatable. A small `CMakePresets.json` might contain a GCC release profile and a sanitized debug profile.

```
{
  "version": 8,
  "configurePresets": [
    {
      "name": "gcc-release",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/gcc-release",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release"
      }
    },
    {
```

```
      "name": "gcc-sanitized",
      "inherits": "gcc-release",
      "binaryDir": "${sourceDir}/build/gcc-sanitized",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "FORGE_ENABLE_SANITIZERS": "ON"
      }
    }
  ]
}
```

A.12 Windows Resource Inclusion

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>
```

```
add_executable(MyWxApp WIN32
    main.cpp
    app.rc
)
```

The resource compiler is part of the build graph. If the target does not list the resource file, the manifest cannot magically appear because the C++ source compiled successfully.

A.13 Console Window during Diagnostics

While diagnosing startup, stdout, or stderr on Windows, temporarily omitting `WIN32` from `add_executable()` can be useful because it selects a console subsystem executable. This is a diagnostic choice, not a cure for a missing manifest or loader problem.

A.14 Compiler Identity Diagnostics

Add these temporarily when you suspect the wrong toolchain is being used:

```
message(STATUS "C compiler   : ${CMAKE_C_COMPILER}")
message(STATUS "C++ compiler : ${CMAKE_CXX_COMPILER}")
message(STATUS "C++ ID       : ${CMAKE_CXX_COMPILER_ID}")
message(STATUS "C++ version  : ${CMAKE_CXX_COMPILER_VERSION}")
message(STATUS "Generator    : ${CMAKE_GENERATOR}")
```

Then inspect the verbose link command:

```
cmake --build build --verbose
# or
ninja -C build -v
```

A.15 Dependency Inspection on Windows

Useful diagnostic commands include:

```
objdump -p MyWxApp.exe | findstr /I "DLL Name"
where.exe libstdc++-6.dll
where.exe libgcc_s_seh-1.dll
where.exe libwinpthread-1.dll
```

With MSYS2's `ntldd` installed:

```
ntldd ./MyWxApp.exe
```

The objective is not to collect long dependency lists. It is to answer concrete questions: Which DLL will Windows load? From where? Does the executable depend on a runtime you intended to link statically? Is a system common-control version being selected through the expected manifest?

A.16 Portable Source Layout

A scalable repository can use:

```
project/
|-- CMakeLists.txt
|-- CMakePresets.json
|-- cmake/
|-- include/project/
|-- src/
|-- resources/
|-- tests/
|-- tools/
|-- third_party/
`-- build/             # ignored/generated
```

Do not place generated artifacts beside source files merely because a small tutorial does. The build tree should be disposable.

A.17 Common Cache Reset Triggers

Delete/recreate the build directory after changes to:

- compiler family or toolchain root;
- 32/64-bit target architecture;
- UCRT/MSVCRT environment;
- static/shared wxWidgets mode;
- major wxWidgets source tree;
- generator family;
- fundamental dependency installation or ABI.

A cache is a feature when it remembers valid expensive decisions. It is a liability when the decisions no longer describe the system you think you are building.

CHAPTER B

Windows Resources, Manifests, and Loader Diagnostics

This appendix exists because Windows GUI correctness begins before `wxApp::OnInit()`. Executable resources, activation context, common-control selection, DPI declarations, runtime DLL resolution, and subsystem metadata can determine whether the process starts at all.

B.1 The Minimal wxWidgets Resource File

For the standard wxWidgets setup:

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>
```

Compile the file into the executable target:

```
add_executable(MyApp WIN32
    main.cpp
    app.rc
)
```

B.2 Why the Manifest Matters

Windows manifests can participate in activation-context selection, DPI awareness, supported-OS declarations, requested privileges, and use of modern common controls. The exact content evolves with framework and platform versions; this is precisely why including the framework-supplied resource is safer than copying an old manifest fragment from a random project.

The wxWidgets Windows FAQ notes that controls can appear in an old classic style when no application manifest is present and that wxWidgets provides a manifest through `wx/msw/wx.rc`. For a modern application, treat that resource chain as executable configuration.

B.3 Case Study: GetWindowSubclass

A particularly instructive failure is:

```
Entry Point Not Found
The procedure entry point GetWindowSubclass could not be located ...
```

`GetWindowSubclass` is a common-control helper API associated with Comctl32 version 6 usage. If an application silently falls back to an older common-control activation context because its manifest was removed, it can build successfully and still fail during loader/control initialization.

The diagnostic lesson is broader than this one symbol:

1. Record the exact missing entry-point name.
2. Record the exact DLL named by Windows.
3. Inspect the executable's imported DLLs.
4. Inspect which physical DLL paths are resolved at runtime.
5. Verify resource/manifest embedding before changing compilers randomly.

Important Pitfall

Do not diagnose “Entry Point Not Found” from the phrase alone. The missing symbol is evidence. `GetWindowSubclass`, a C++ mangled symbol, a UCRT function, and a graphics API entry point imply very different causes.

B.4 Inspect Imported DLL Names

With GNU binutils:

```
objdump -p MyApp.exe | findstr /I "DLL Name"
```

A static wxWidgets build should not import wxWidgets DLLs. If `-static-libgcc` and `-static-libstdc++` were applied successfully under MinGW, the corresponding GNU runtime DLLs should normally disappear from the import list as well.

Windows system DLLs such as `KERNEL32.dll`, `USER32.dll`, `GDI32.dll`, `OLE32.dll`, and `COMCTL32.dll` are normal dependencies of a native desktop application.

B.5 Inspect Resolved Physical DLL Paths

An import table tells you names. Runtime resolution tells you which files are actually used. MSYS2’s `ntldd` is useful for a MinGW environment:

```
ntldd ./MyApp.exe
```

Look for accidental mixing such as:

- application built by UCRT64 but runtime DLL loaded from MINGW64;
- WinLibs executable loading an unrelated MSYS2 runtime from PATH;
- stale DLL copied beside the executable;

- old runtime dropped into a global Windows directory.

B.6 The Application Directory Has Priority

A stale runtime DLL sitting next to the executable can be more dangerous than an incorrect global PATH because local application dependencies are deliberately searched early. During diagnostics, list every DLL beside the executable and remove files you did not intentionally deploy.

```
# PowerShell
Get-ChildItem . -Filter *.dll
```

B.7 Do Not Put MinGW Runtimes into System32

Do not "solve" deployment by copying `libstdc++-6.dll`, `libgcc_s_seh-1.dll`, or `libwinpthread-1.dll` into `C:\Windows \System 32`. Application-specific runtimes belong with the application or in a deliberately managed environment, not in the Windows system directory.

B.8 UCRT and Toolchain Families

Modern MinGW distributions can target different Windows C runtime arrangements. Names such as MSYS2 UCRT64 and `MINGW64` are not decorative labels. Keep compiler, CRT choice, pthread runtime, binutils, third-party libraries, and wxWidgets build coherent.

A version string such as "GCC 16.2" does not by itself prove two distributions are ABI-identical. The build root is part of the identity.

B.9 Verbose Link Commands Are Evidence

When a linker error appears, inspect the actual command, not the IDE label. A verbose line reveals the executable compiler driver, library search roots, static archives, runtime flags, and system libraries.

```
cmake --build build --verbose
```

If the path contains an unexpected IDE-bundled MinGW installation, you have learned something concrete. If it points to the intended toolchain but the loader still fails, stop blaming the IDE and inspect runtime resolution and manifests.

B.10 Resource Rebuild Discipline

Resource changes are compiled and linked like source changes. If the build system is confused after changing a manifest or switching static/shared settings, remove the build directory and configure again. Do not assume an old resource object has been replaced merely because the C++ source rebuilt.

B.11 DPI Awareness

High-DPI support is both a manifest/configuration concern and an application-layout concern. Declaring DPI awareness prevents unwanted bitmap scaling, but your application must also avoid fixed-pixel assumptions. Use sizers, best sizes, `FromDIP()`, font metrics, and scalable bitmap/icon facilities.

B.12 Useful Windows Diagnostics Checklist

Checklist

1. Capture the exact loader dialog text.
2. Confirm the executable being launched is the newly built one.
3. Confirm architecture is correct (x86-64 versus x86).
4. Inspect imported DLL names with `objdump -p` or an equivalent PE viewer.
5. Inspect resolved DLL paths with `ntldd` or Dependencies.
6. List DLLs beside the executable.
7. Verify `app.rc` is part of the target.
8. Verify `wx/msw/wx.rc` is included unless you deliberately supply an equivalent manifest/resources setup.
9. Reconfigure in a clean build directory.
10. Only then investigate compiler/runtime defects.

B.13 Resource File with an Application Icon

A project can extend the standard wxWidgets resource instead of replacing it:

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>

APP_ICON ICON "assets/forgevm-editor.ico"
```

Keep the icon path relative to the resource file/build rules in a way the resource compiler can resolve consistently.

B.14 Version Information

Production Windows applications commonly embed version metadata as a `VERSIONINFO` resource. This improves Explorer properties, crash-report identification, deployment diagnostics, and support conversations. Generate numeric/string versions from the build rather than editing them manually in multiple places.

B.15 Final Rule

Why This Matters

A GUI executable is more than linked C++ code. On Windows, its resources and manifest are part of its runtime contract. Treat `app.rc` with the same seriousness as `main.cpp`: review it, version it, build it, and test the resulting binary.

CHAPTER C

Core Class Quick Reference

This appendix is a learning-oriented map, not a replacement for the official API reference. It answers three practical questions: *What is this class for? When would I reach for it? What should I remember before using it?* Exact overloads, platform availability, and version-specific details should always be checked against the wxWidgets 3.3 manual.

Practical Tip

Learn classes by family. If you understand the role of `wxWindow`, `wxEvtHandler`, sizers, model-driven views, and process/thread boundaries, most unfamiliar wxWidgets classes become variations on a small number of architectural patterns.

C.1 Application, lifetime, and event infrastructure

Class	Purpose	Use / engineering note
`wxApp`	Process-level GUI application object; initialization, command line, global application policy.	Override OnInit() for GUI startup; use SetTopWindow() and app/vendor identity deliberately.
`wxAppConsole`	Base application object without GUI-window requirements.	Useful for console utilities that still want wxWidgets base services.

Class	Purpose	Use / engineering note
`wxEvtHandler`	Core event-processing base class.	Bind events dynamically; keep handler lifetime consistent with bound objects.
`wxEvent`	Base event object.	Usually consume a specific derived event type rather than wxEvent directly.
`wxCommandEvent`	Command-like event used by buttons, menus, text controls, and many controls.	Command events often propagate upward through parent windows.
`wxUpdateUIEvent`	Event used to derive enabled/checked/visible command state.	Prefer derived state over scattering Enable() calls through handlers.
`wxTimer`	Schedules periodic timer events on the event loop.	Do not use a timer as a high-precision real-time clock; keep handlers short.
`wxTimerEvent`	Event delivered by wxTimer.	Bind wxEVT_TIMER to the owning event handler.
`wxIdleEvent`	Notification when the event loop is otherwise idle.	Use sparingly; avoid continuous work that keeps the process permanently busy.
`wxProcessEvent`	Signals asynchronous child-process termination.	Combine with wxProcess for non-blocking external tools.
`wxThreadEvent`	Thread-friendly event carrying data back to GUI code.	Post to GUI objects instead of touching windows from worker threads.

C.2 Windows, containers, and application shells

Class	Purpose	Use / engineering note
`wxWindow`	Base of most visible windows and controls; parenting, sizing, focus, events, coordinates.	Understand parent-child lifetime and never hard-code geometry as your primary layout strategy.
`wxTopLevelWindow`	Common base of top-level desktop windows.	Usually work through wxFrame or wxDialog.
`wxFrame`	Primary top-level application shell with menus, toolbars, and status bar support.	Use a child panel or specialized view as the real content surface.
`wxDialog`	Top-level interaction window, often modal.	Use modal dialogs for focused, bounded decisions; avoid abusing them for long workflows.
`wxPanel`	Generic child container with good focus/tab traversal behavior.	A strong default for grouping controls inside frames and dialogs.
`wxScrolledWindow`	Child window providing scrollable content.	Set scroll rate/virtual size or use sizer-driven scrolling appropriately.
`wxSplitterWindow`	Resizable split between two child windows.	Ideal for editor/output, tree/editor, or inspector/workspace arrangements.
`wxNotebook`	Tabbed pages using native notebook conventions.	Good for settings/pages; for complex document work consider model separation beyond tabs.

Class	Purpose	Use / engineering note
`wxSimplebook`	Page container without visible tab UI.	Useful for wizard-like or state-driven page switching.
`wxCollapsiblePane`	Expandable/collapsible child region.	Good for optional advanced settings without custom animation logic.
`wxMiniFrame`	Small top-level tool window where supported.	Use only when a separate floating tool window genuinely fits platform UX.

Class	Purpose	Use / engineering note

C.3 Fundamental controls

Class	Purpose	Use / engineering note
`wxStaticText`	Read-only text label.	Let sizers handle label growth caused by localization and font changes.
`wxStaticBitmap`	Displays a bitmap/icon.	Prefer scalable assets/bitmap bundles for high-DPI interfaces.
`wxButton`	Standard push button.	Connect command intent through events; use standard IDs where semantics match.
`wxBitmapButton`	Button displaying a bitmap.	Ensure accessible text/tooltip and scalable imagery.
`wxToggleButton`	Button with persistent on/off state.	Use for mode state where a toggled visual is clearer than a check box.
`wxCheckBox`	Independent Boolean or tri-state option.	Use when several options can be selected independently.
`wxRadioButton`	Mutually exclusive option in a logical group.	Keep group semantics visually obvious and keyboard accessible.

Class	Purpose	Use / engineering note
`wxTextCtrl`	Native text entry/edit control.	Excellent for forms and modest multiline text; not a source-code editor replacement.
`wxSearchCtrl`	Search-oriented text control with native affordances where available.	Useful for filter bars and search fields.
`wxChoice`	Drop-down selection from fixed choices.	Prefer when arbitrary text entry is not meaningful.
`wxComboBox`	Editable or read-only combo control.	Use when selection plus optional text entry is a genuine requirement.
`wxListBox`	Simple list of strings.	Use for modest lists without rich columns or model complexity.
`wxCheckListBox`	List of strings with check states.	Useful for feature/module selection lists.
`wxSlider`	Continuous or stepped numeric choice.	Use when relative position is meaningful; pair with a label/value where precision matters.
`wxSpinCtrl`	Integer spin control.	Good for bounded integer values.
`wxSpinCtrlDouble`	Floating-point spin control.	Specify increment and precision according to domain needs.

Class	Purpose	Use / engineering note
`wxGauge`	Progress/activity indicator.	Do not fake precision; use pulse/indeterminate mode where work amount is unknown.
`wxHyperlinkCtrl`	Native-style hyperlink control.	Use for documentation/support links; avoid disguising arbitrary actions as links.

Class	Purpose	Use / engineering note

C.5 Layout and geometry

Class	Purpose	Use / engineering note
`wxSizer`	Abstract layout manager.	Never instantiate directly; understand Add(), proportion, flags, and borders.
`wxBoxSizer`	One-dimensional horizontal or vertical layout.	The workhorse for most application composition.
`wxStaticBoxSizer`	BoxSizer paired with a labeled group box.	Use for semantic grouping when the platform style supports it well.
`wxGridSizer`	Equal-cell regular grid.	Good for uniform button/icon matrices.
`wxFlexGridSizer`	Grid whose selected rows/columns can grow.	Strong for forms and aligned labels/fields.
`wxGridBagSizer`	Grid with explicit row/column spans and irregular placement.	Use when a form truly needs spanning, not as a substitute for nested simple sizers.
`wxWrapSizer`	Wraps children to additional rows/columns as space changes.	Useful for tag/chip/tool palettes.
`wxSizerItem`	Represents one item managed by a sizer.	Mostly accessed when modifying an existing layout dynamically.

C.6 Dialogs and standard interactions

Class	Purpose	Use / engineering note
`wxFileDialog`	Native file open/save dialog abstraction.	Use flags such as wxFD_FILE_MUST_EXIST and wxFD_OVERWRITE_PROMPT intentionally.
`wxDirDialog`	Native directory chooser abstraction.	Use for selecting folders rather than forcing path text entry.
`wxMessageDialog`	Configurable message dialog.	Prefer clear action-oriented text; don't overload users with modal warnings.
`wxTextEntryDialog`	Small modal text prompt.	Good for simple one-value input; use a real dialog for validation-heavy workflows.
`wxPasswordEntryDialog`	Masked text-entry dialog.	Suitable for small password prompts, not complete credential architecture.
`wxNumberEntryDialog`	Simple bounded numeric prompt.	Convenient for Go to Line and small numeric commands.
`wxColourDialog`	Native/platform colour selection interface.	Persist domain colour values separately from dialog objects.

Class	Purpose	Use / engineering note
`wxFontDialog`	Font chooser.	Remember code-editor font settings belong in preferences/state, not the dialog.
`wxProgressDialog`	Modal-ish progress UI with cancellation options.	Use cautiously; background/non-modal status is often better for development tools.

Class	Purpose	Use / engineering note

C.7 Lists, trees, and model-driven views

Class	Purpose	Use / engineering note
`wxListCtrl`	Classic list/report/icon control.	Useful for traditional report lists; for richer models consider wxDataViewCtrl.
`wxTreeCtrl`	Classic hierarchical tree control.	Good for project trees with explicit item data and IDs.
`wxDataViewCtrl`	Model-driven tabular/tree data view.	Strong choice for structured diagnostics, symbols, build results, and virtual data.
`wxDataViewListCtrl`	Convenience list-backed data view.	Use when a custom model would be needless overhead.
`wxDataViewTreeCtrl`	Convenience tree-oriented data view.	Useful for simpler tree data without a custom model.
`wxDataViewModel`	Abstract model contract for wxDataViewCtrl.	Keep stable item identity and notify views when model data changes.

Class	Purpose	Use / engineering note
`wxDataViewVirtualListModel`	Virtual row model addressed by row number.	Good for very large flat result sets.
`wxDataViewColumn`	Column definition for a data view.	Own renderers/editability/sort behavior at the presentation boundary.
`wxPropertyGrid`	Property-oriented name/value editor.	Excellent for inspector/settings metadata; link wx::propgrid.
`wxPropertyGridManager`	Higher-level property grid with pages/tooling.	Useful for large categorized inspectors.

Class	Purpose	Use / engineering note

C.9 Graphics, images, printing, and display

Class	Purpose	Use / engineering note
`wxDC`	Abstract drawing context.	Use a concrete DC appropriate to paint, memory, print, or client context.
`wxPaintDC`	Drawing context used during paint events.	Create in paint handlers when required by the platform/paint model.
`wxAutoBufferedPaintDC`	Buffered paint DC that reduces flicker.	A good default for custom-drawn views that repaint substantial content.
`wxMemoryDC`	Draws into a bitmap in memory.	Useful for off-screen rendering and image composition.
`wxGraphicsContext`	Higher-level vector/antialiased drawing API.	Use for richer paths, transforms, and high-quality custom graphics.
`wxPen`	Stroke style for DC drawing.	Treat as drawing state; avoid needless recreation inside tight loops.
`wxBrush`	Fill style for DC drawing.	Use transparent/solid/patterned brushes according to the selected DC API.
`wxFont`	Font description used by controls and drawing.	Respect user/platform fonts; don't assume exact metrics.

Class	Purpose	Use / engineering note
`wxColour`	Portable colour value.	Use system colours or theme-aware choices for native UI where possible.
`wxBitmap`	Raster bitmap resource.	Prefer wxBitmapBundle or multiple resolutions for high-DPI UI imagery.
`wxBitmapBundle`	Scale-aware collection/source of bitmaps.	Preferred for toolbar/menu/window imagery on mixed-DPI systems.
`wxIcon`	Window/application icon abstraction.	Provide platform-appropriate sizes/resources.
`wxImage`	Platform-neutral image data and transformations.	Use for load/scale/format work, then convert to display bitmap when needed.
`wxDisplay`	Information about monitors/displays.	Use when restoring windows or handling multi-monitor workspaces.
`wxPrinter`	Printing orchestration.	Keep printable document/layout logic separated from live widget geometry.
`wxPrintout`	Abstract printable content description.	Implement page count, printing, and preview drawing consistently.

C.10 Files, paths, configuration, and streams

Class	Purpose	Use / engineering note
`wxFileName`	Cross-platform path/filename abstraction.	Use for extension changes, normalization, path components, and file tests.
`wxStandardPaths`	Platform-conventional locations for config/data/cache/documents.	Never assume the executable directory is writable.
`wxFile`	Low-level file wrapper.	For raw file operations when std::filesystem/streams are not the chosen boundary.
`wxFFile`	FILE*-style file wrapper.	Useful with C-runtime semantics inside wxWidgets code.
`wxTextFile`	Line-oriented text file abstraction.	Convenient for smaller textual files when preserving line structure matters.
`wxDir`	Directory enumeration.	Use with care on huge trees; move scans off the GUI thread when needed.
`wxFileSystemWatcher`	Watches filesystem changes.	Project editors can refresh changed files/workspaces without polling.
`wxConfig`	Abstract persistent configuration interface.	Use stable logical keys; avoid leaking backend/storage assumptions across the app.

Class	Purpose	Use / engineering note
`wxFileConfig`	File-backed configuration implementation.	Useful for explicit config files or portable setups.
`wxInputStream`	Abstract readable byte stream.	Many wxWidgets subsystems expose stream-based I/O.
`wxOutputStream`	Abstract writable byte stream.	Pair with concrete file/memory/socket streams as needed.
`wxMemoryInputStream`	Input stream backed by memory.	Useful for embedded/generated data.
`wxMemoryOutputStream`	Output stream accumulating in memory.	Useful for serialization, generated content, and tests.

C.11 Processes, threading, and synchronization

Class	Purpose	Use / engineering note
`wxProcess`	Represents/redirects an external child process.	Use asynchronous execution for build tools; drain redirected streams.
`wxExecute`	Function family that starts external commands.	Choose synchronous/asynchronous flags deliberately and test quoting.

Class	Purpose	Use / engineering note
`wxThread`	wxWidgets thread abstraction.	For new application-domain code, standard C++ threading can also be appropriate; never touch GUI from workers.
`wxMutex`	Mutual-exclusion primitive.	Prefer narrow protected state; don't hold locks while posting UI work or doing slow I/O.
`wxCriticalSection`	Lightweight mutual-exclusion wrapper.	Useful in wx-style code for small critical regions.
`wxCondition`	Condition variable paired with a mutex.	Use for worker coordination rather than busy-wait loops.
`wxSemaphore`	Counting synchronization primitive.	Appropriate for resource/producer-consumer constraints.

Class	Purpose	Use / engineering note

C.12 Clipboard, drag/drop, system integration, and utilities

Class	Purpose	Use / engineering note
`wxClipboard`	Clipboard access abstraction.	Open/use/close carefully; clipboard availability can be transient.
`wxTextDataObject`	Clipboard/drag payload for text.	Use UTF-aware wxString content.
`wxFileDataObject`	Clipboard/drag payload for file paths.	Useful for file drag-and-drop integration.
`wxDropTarget`	Base class for accepting drag-and-drop data.	Implement only where drag/drop improves a real workflow.
`wxFileDropTarget`	Convenience drop target for file lists.	Good for dropping source/project files onto an editor.
`wxDropSource`	Initiates a drag operation.	Use with data objects and platform feedback.
`wxMimeTypesManager`	Queries MIME/file-type associations.	Useful for open-with and file association related tooling.

Class	Purpose	Use / engineering note
`wxLaunchDefaultApplication`	Opens a document with the OS default application.	Use for external docs/logs when appropriate.
`wxLaunchDefaultBrowser`	Opens a URL in the default browser.	Use for project documentation/support links.
`wxLocale`	Locale/i18n support.	Plan translation boundaries early if the product will localize.
`wxTranslation`	Translation catalog/service support in newer wxWidgets APIs.	Keep translatable UI strings separate from protocol/tool output.
`wxDateTime`	Date/time value and formatting facilities.	Use for logs, timestamps, UI dates; distinguish local time from machine durations.
`wxStopWatch`	Elapsed-time measurement utility.	Useful for lightweight instrumentation, not high-end profiling.
`wxLog`	Logging facade.	Install appropriate log targets and avoid using modal logs for routine background diagnostics.
`wxLogWindow`	Window-based log target.	Can provide an internal diagnostic console during development.

Class	Purpose	Use / engineering note
`wxSystemSettings`	Queries system colours, fonts, and metrics.	Prefer system-derived values over hard-coded native-UI styling.
`wxPlatformInfo`	Runtime/build platform information.	Useful for diagnostics/about reports, not for scattering behavior branches.

C.13 High-Frequency Method Maps

C.13.1 wxWindow

Method	Meaning
`Show(bool)`	Show or hide the window.
`Enable(bool)`	Enable or disable user interaction.
`SetSizer(wxSizer*)`	Attach a layout manager to a window.
`Layout()`	Re-run layout when dynamic content/constraints changed.
`SetFocus()`	Request keyboard focus.

`FromDIP(...)`	Convert device-independent dimensions for the current window/display.
`GetParent()`	Retrieve the parent window in the ownership/event hierarchy.
`Destroy()`	Schedule safe destruction of a window in GUI lifetime rules.
`Bind(...)`	Dynamically bind an event type to a handler.
`CallAfter(...)`	Schedule a callable to execute later on the event loop.

C.13.2 wxStyledTextCtrl

Method	Meaning
`SetLexer(...)`	Select a Scintilla lexer.
`SetKeyWords(set,text)`	Supply keywords for lexer-defined keyword sets.
`StyleSetForeground(...)`	Set syntax-style foreground colour.
`SetMarginType(...)`	Configure line-number/symbol/text margins.
`MarkerDefine(...)`	Define a marker symbol and colours.
`MarkerAdd(line,id)`	Place a marker on a document line.
`IndicatorSetStyle(...)`	Configure a range indicator, e.g. diagnostics/search.
`SetSavePoint()`	Declare the current text state unmodified.
`GetModify()`	Query whether the text differs from the save point.
`GotoLine(line)`	Navigate to a zero-based document line.

	Scroll as necessary to reveal the caret.
`EnsureCaretVisible()`	

C.13.3 wxDataViewCtrl

Operation	Meaning
`AssociateModel(...)`	Connect a custom data model to the view.
`AppendTextColumn(...)`	Add a convenience text column.
`GetSelection()`	Obtain the selected model item.
`Select(...)`	Select a model item programmatically.
`Expand(...)` / `Collapse(...)`	Control tree-item expansion where applicable.

C.13.4 wxFileName

Operation	Meaning
`GetFullPath()`	Reconstruct the complete path.
`GetFullName()`	Filename plus extension without directory.
`SetExt(...)`	Change the extension while preserving other path parts.
`FileExists()`	Test whether the represented file exists.
`DirExists()`	Test whether the represented directory exists.
`Normalize(...)`	Normalize path components according to selected flags/-context.

C.13.5 wxProcess

Operation	Meaning
`Redirect()`	Request redirected child standard streams.
`GetInputStream()`	Read child's stdout after redirection.
`GetErrorStream()`	Read child's stderr after redirection.
`Kill(...)`	Request termination/signal handling according to platform support.
`wxExecute(...) + wxEXEC_ASYNC`	Start the child asynchronously and return a process identifier.

C.14 Choosing the Right Family Quickly

Need	Start by looking at
Main application window	wxFrame, wxPanel, wxSizer family
Settings form	wxDialog, wxPanel, wxFlexGridSizer, standard input controls

Source editor	wxStyledTextCtrl, wxSplitterWindow, wxAUI for larger workspaces
Project tree	wxTreeCtrl or wxDataViewCtrl with a model
Problems/diagnostics table	wxDataViewCtrl / wxDataViewListCtrl
Inspector	wxPropertyGrid / wxPropertyGridManager
Run compiler/assembler	wxProcess, wxExecute, wxProcessEvent, timers or stream-drain strategy
Background CPU work	standard C++ threads or wxThread plus posted/call-after GUI results
Persistent preferences	wxConfig / wxFileConfig plus typed application settings layer
Portable paths	wxFileName, wxStandardPaths, optionally std::filesystem in domain code
Custom drawing	paint event + wxPaintDC/wxAutoBufferedPaintDC or wxGraphicsContext
Clipboard / drag drop	wxClipboard + data objects / drop targets

Why This Matters

The most useful wxWidgets reference is the one that helps you choose a family first. Once the family is correct, the official manual can answer the exact constructor, style flag, event, and platform-availability question without forcing you to search the entire framework blindly.

CHAPTER D

Cross-Platform Engineering Notes

wxWidgets lets one application architecture serve several desktop systems, but mature cross-platform software succeeds by respecting differences rather than hiding them carelessly.

D.1 Portable Policy, Native Adaptation

Share document models, commands, project logic, diagnostics, parsers, settings schemas, and most UI composition. Isolate platform-specific work around narrow capabilities such as native handles, registry integration, shell services, packaging, special file locations, or operating-system-specific appearance features.

D.2 Windows

- Keep the resource/manifest path explicit.
- Test high-DPI scaling and multiple monitors.
- Test paths containing spaces and Unicode.
- Understand the selected CRT/toolchain family when using MinGW.
- Use the GUI subsystem for production GUI binaries; use a console subsystem intentionally during diagnostics/tools.

- Validate dark/light mode behaviour on the exact wxWidgets version you ship because platform support continues to evolve.

D.3 Linux / wxGTK

- Prefer distribution packages for stable system deployment when appropriate, or build a controlled wxWidgets version for a bundled application.
- Test under more than one desktop theme; GTK theme metrics can expose layout assumptions.
- Do not hard-code a single font family that may not exist on the target distribution.
- File-dialog behaviour and desktop integration can vary with GTK and portal environment.
- Package runtime dependencies according to the chosen distribution/AppImage/Flatpak strategy rather than assuming a development machine represents users.

D.4 macOS

- Respect native menu placement and application conventions.
- Understand application bundles and resource locations.
- Test Retina/high-DPI presentation with scalable assets.
- Code signing and notarization are deployment architecture, not last-minute cosmetic steps.
- Avoid Windows-specific assumptions about executable-relative writable directories.

D.5 Paths

Use wxFileName, wxStandardPaths, or standard C++ filesystem abstractions at well-defined boundaries. Never construct a portable path by blindly concatenating backslashes.

D.6 Line Endings and Text Encoding

Source editors must preserve or deliberately normalize line endings. Decide whether the editor will:

- preserve the file's existing EOL mode;
- expose CRLF/LF selection in the status bar;
- normalize only when explicitly configured;
- write UTF-8 by default;
- detect/handle BOMs where needed.

Assembler source can travel between platforms and build systems. Silent transformations deserve a documented policy.

D.7 Object Formats Are Not the Operating System

NASM object formats such as `win64`, `elf64`, and `macho64` describe object-file targets. The editor can expose them as build profiles, but a real toolchain profile also includes linker choice, ABI assumptions, include paths, output extension, and run/debug strategy.

D.8 Keyboard Conventions

Accelerators should respect platform expectations. A command model should express “Save” independently from a literal key chord so the presentation layer can map appropriate shortcuts per platform.

D.9 Menus

Menu location and ordering differ across desktop systems. Use wxWidgets menu abstractions and standard IDs where appropriate instead of forcing a Windows-shaped menu convention onto every target.

D.10 Fonts and Text Metrics

The same point-size font has different metrics across fonts/platforms. Sizers should absorb those differences. For code, choose a monospaced family through platform conventions or user settings; do not depend on one proprietary font being installed.

D.11 DPI and Scaling

A correct cross-platform interface is built from constraints and scalable units:

- sizers, not fixed coordinates;
- best/minimum sizes, not screenshots as geometry specifications;
- `FromDIP()` for dimensions representing device-independent interface intent;
- bitmap bundles/vector sources where supported;
- testing at non-100% scale factors.

D.12 Native Dialogs

Native file/colour/font dialogs can differ significantly. Your code should consume their result, not assume internal child-control layout or button order.

D.13 Process Execution

Quoting and shell rules vary. Keep external-tool invocation behind an adapter with tests for each supported platform. Treat argument arrays as structured data and make shell interpretation explicit.

D.14 Configuration Storage

wxConfig can map settings to platform-appropriate backends. Keep the logical keys stable even if physical storage differs. For project-local settings, use an explicit project file rather than hiding everything in user-global configuration.

D.15 Packaging

A release checklist should include platform-native concerns:

Platform	Release concerns
Windows	EXE/DLL set, manifest/resources, runtime choice, signing, installer/portable package, SmartScreen expectations.
Linux	distro/runtime dependencies, desktop file/icon, AppImage/Flatpak/deb/rpm strategy, executable permissions.

macOS	app bundle, resources, Info.plist, signing, notarization, universal/architecture strategy.

D.16 Architecture and CPU

wxWidgets abstracts GUI platforms, not arbitrary machine-code backends. If ForgeVM later supports x86-64, AArch64, and RISC-V tooling, keep processor-specific domain logic away from GUI classes. The same editor view should display diagnostics for any architecture through common models.

D.17 Conditional Compilation

Use platform conditionals at narrow adaptation boundaries:

```
#ifdef __WXMSW__
    // Windows-specific integration.
#elif defined(__WXOSX__)
    // macOS-specific integration.
#elif defined(__WXGTK__)
    // GTK-specific integration.
#endif
```

If large portions of your frame are wrapped in platform macros, the abstraction boundary is too wide.

D.18 Cross-Platform Validation Matrix

For each release, record at least:

- compiler and version;

- wxWidgets version/configuration;
- OS version;
- CPU architecture;
- Debug/Release build result;
- startup test;
- file open/save Unicode path test;
- external process test;
- high-DPI test;
- packaging smoke test.

Why This Matters

Cross-platform is not a property granted once by choosing wxWidgets. It is a continuous engineering practice: constrain shared code to portable concepts, isolate genuine differences, and validate on real targets.

CHAPTER E

Event, Command, and Style Atlas

This appendix is a field map for the event-driven vocabulary encountered repeatedly in desktop applications. It emphasizes intent rather than attempting to reproduce every constant in the official reference.

Event	Typical source	Engineering note

E.1 Common Event Families

Event	Typical source	Engineering note
wxEVT_BUTTON	wxButton	Route to a command handler; keep business work outside the button object.
wxEVT_TOGGLEBUTTON	wxToggleButton	Treat the checked state as view state for a mode or bind it to a model setting.
wxEVT_CHECKBOX	wxCheckBox	Read the new Boolean/tri-state value from the event/control.
wxEVT_RADIOBUTTON	wxRadioButton	Useful when individual radio controls need explicit handling.
wxEVT_CHOICE	wxChoice	Respond to a fixed-list selection change.
wxEVT_COMBOBOX	wxComboBox	Selection changed through the combo list.
wxEVT_TEXT	wxTextCtrl and text-like controls	Fires frequently; avoid expensive validation on every keystroke unless designed for it.

Event	Typical source	Engineering note
wxEVT_TEXT_ENTER	wxTextCtrl	Requires an appropriate style for Enter processing; useful for search/location bars.
wxEVT_LISTBOX	wxListBox	Selection changed.
wxEVT_LISTBOX_DCLICK	wxListBox	Activate/open selected item without inventing mouse-only semantics for the command.
wxEVT_SLIDER	wxSlider	Value changed; consider throttling expensive previews.
wxEVT_SPINCTRL	wxSpinCtrl	Integer value changed.
wxEVT_MENU	wxMenuItem / accelerator	Core command event for menu and keyboard command routing.
wxEVT_UPDATE_UI	Command/UI surface	Derive enabled/checked/visible state from current application capability.
wxEVT_TOOL	wxToolBar	Usually route to the same command as the equivalent menu item.
wxEVT_CLOSE_WINDOW	Top-level window	Veto for unsaved work/cancellation; otherwise allow normal close.
wxEVT_SHOW	Window	Visibility changed; do not use as a substitute for application lifecycle policy.

Event	Typical source	Engineering note
`wxEVT_SIZE`	Window	New client size; sizers normally handle children without manual geometry code.
`wxEVT_MOVE`	Window	Useful for persistence/monitor logic, not continuous heavy work.
`wxEVT_ACTIVATE`	Top-level window	Application/window activation changed.
`wxEVT_SET_FOCUS`	Window	Focus entered; keep focus logic minimal and accessible.
`wxEVT_KILL_FOCUS`	Window	Focus left; be cautious with modal validation that traps users.
`wxEVT_CHAR`	Window	Character-level keyboard handling after translation.
`wxEVT_KEY_DOWN`	Window	Physical/key-code oriented handling; accelerators are better for commands.
`wxEVT_KEY_UP`	Window	Rarely needed for ordinary commands; useful for interaction modes.
`wxEVT_LEFT_DOWN`	Window	Mouse press; capture only when custom interactions require it.
`wxEVT_LEFT_UP`	Window	Mouse release; pair carefully with capture/drag state.

Event	Typical source	Engineering note
wxEVT_LEFT_DCLICK	Window	Double click; never make an essential action inaccessible by keyboard.
wxEVT_MOTION	Window	Mouse motion; can be high frequency, so keep handlers light.
wxEVT_MOUSEWHEEL	Window	Scrolling/zoom gestures; respect platform conventions and modifiers.
wxEVT_ENTER_WINDOW	Window	Pointer entered window region.
wxEVT_LEAVE_WINDOW	Window	Pointer left window region.
wxEVT_PAINT	Window	Repaint request; draw current state, not an accumulated history of drawing commands.
wxEVT_ERASE_BACKGROUND	Window	Background erase notification; custom rendering can manage flicker carefully.
wxEVT_TIMER	wxTimer	Periodic/one-shot event-loop scheduling.
wxEVT_IDLE	Application/window	Opportunistic idle processing; avoid permanent busy idle loops.
wxEVT_END_PROCESS	wxProcess	Asynchronous child process terminated.

Event	Typical source	Engineering note
`wxEVT_THREAD`	Posted worker event	Deliver thread results to GUI code safely.
`wxEVT_FILEPICKER_CHANGED`	File picker control	User selected a different file.
`wxEVT_DIRPICKER_CHANGED`	Directory picker control	User selected a different directory.
`wxEVT_NOTEBOOK_PAGE_CHANGED`	wxNotebook	Page selection completed.
`wxEVT_NOTEBOOK_PAGE_CHANGING`	wxNotebook	Page transition about to occur and may be vetoable.
`wxEVT_SPLITTER_SASH_POS_CHANGED`	wxSplitterWindow	Persist user workspace proportions if appropriate.
`wxEVT_TREE_SEL_CHANGED`	wxTreeCtrl	Tree selection changed; selection is not durable item identity.
`wxEVT_TREE_ITEM_ACTIVATED`	wxTreeCtrl	User activated an item; map to an Open/Navigate command.
`wxEVT_DATAVIEW_SELECTION_CHANGED`	wxDataViewCtrl	Model-view selection changed.

Event	Typical source	Engineering note
wxEVT_DATAVIEW_ITEM_ACTIVATED	wxDataViewCtrl	Activate selected model item.
wxEVT_DATAVIEW_COLUMN_HEADER_CLICK	wxDataViewCtrl	Column interaction; often sorting or view configuration.
wxEVT_STC_CHANGE	wxStyledTextCtrl	Document text changed; update title/dirty-dependent state, but avoid reparsing everything synchronously.
wxEVT_STC_UPDATEUI	wxStyledTextCtrl	Caret/selection/editor UI state changed; useful for line/column status.
wxEVT_STC_MARGINCLICK	wxStyledTextCtrl	Margin interaction for folds, breakpoints, diagnostics, or bookmarks.
wxEVT_STC_CHARADDED	wxStyledTextCtrl	Character inserted; can drive indentation/completion triggers with care.
wxEVT_STC_DWELLSTART	wxStyledTextCtrl	Mouse dwell began; useful for hover information with cancellation.
wxEVT_STC_DWELLEND	wxStyledTextCtrl	Mouse dwell ended; dismiss or cancel hover content.

E.2 Bind Patterns

Bind a member handler:

```
Bind(wxEVT_MENU, &MainFrame::OnOpen, this, wxID_OPEN);
```

Bind a specific control:

```
button->Bind(wxEVT_BUTTON, &Panel::OnRun, this);
```

Bind a narrow lambda when the behaviour is truly local:

```
Bind(wxEVT_MENU,
     [this](wxCommandEvent&) { Close(true); },
     wxID_EXIT);
```

Practical Tip

Lambdas are excellent for short local glue. If a lambda grows into a workflow, give the operation a name and move it into a handler/service. Anonymous complexity is still complexity.

E.3 Propagation and Skip

Command events commonly propagate upward through the window hierarchy, while many low-level events do not. Calling `event.Skip()` does not mean "ignore this event"; it asks wxWidgets to continue normal processing/propagation according to that event's rules. Learn the event family before using `Skip()` mechanically.

E.4 Standard IDs

wxWidgets defines IDs for common commands such as New, Open, Save, Save As, Exit, Copy, Cut, Paste, Undo, Redo, About, and Preferences. Use a standard ID when your action has the corresponding standard semantics. Platforms can then apply conventional labels, menu positions, or behavior more naturally.

For application-specific commands, allocate IDs above `wxID_HIGHEST` or use a robust ID-generation strategy appropriate to your version/design.

E.5 Accelerators Are a Presentation of Commands

A keyboard shortcut should invoke the same command path as the menu or toolbar. Do not create "Ctrl-S code" and "Save menu code" separately. One command, multiple surfaces.

E.6 High-Frequency Event Rules

Treat these as potentially high-frequency:

- mouse motion;
- text changes;
- STC update UI;
- size/move during dragging;
- paint;
- idle;
- filesystem watcher bursts.

Do not perform full-project scans, blocking disk access, or heavyweight parsing directly in such

handlers. Coalesce, debounce, queue, or move expensive work to a background service.

E.7 Common Window Style Categories

Style flags are constructor-time behavior choices. They vary by control and platform, so verify exact applicability in the official manual.

Style / category	Intent
wxTE_MULTILINE	Multiline text control.
wxTE_READONLY	Text view cannot be edited by the user.
wxTE_PROCESS_ENTER	Generate text-enter event where supported/appropriate.
wxTE_PASSWORD	Mask text entry for password-like input.
wxTE_DONTWRAP	Avoid line wrapping in multiline text where supported.
wxLB_SINGLE / wxLB_MULTIPLE / wxLB_EXTENDED	Selection model for list boxes.
wxCB_READONLY	Combo selection constrained to supplied choices.

`wxSL_HORIZONTAL / wxSL_VERTICAL`	Slider orientation.
`wxSP_LIVE_UPDATE`	Splitter updates panes while sash is dragged.
`wxTR_HAS_BUTTONS`	Tree shows expand/collapse affordances where platform supports them.
`wxTR_MULTIPLE`	Tree permits multiple selection.
`wxDV_ROW_LINES`	Data view row-line presentation where supported.
`wxDV_VERT_RULES`	Data view vertical rules where supported.
`wxBORDER_NONE / wxBORDER_SIMPLE / wxBORDER_SUNKEN`	Border presentation families; native behavior differs.
`wxHSCROLL / wxVSCROLL`	Scrolling behavior for applicable windows.
`wxTAB_TRAVERSAL`	Container participates in keyboard tab traversal.

E.8 Sizer Flags You Will Use Constantly

Flag	Meaning
`wxEXPAND`	Expand the item in the sizer's cross-axis/available allocation.
`wxALL`	Apply border spacing on all four sides.
`wxLEFT / wxRIGHT / wxTOP / wxBOTTOM`	Apply border spacing selectively.
`wxALIGN_CENTER`	Center according to applicable alignment semantics.
`wxALIGN_CENTER_VERTICAL`	Vertical alignment within allocated cell/row.
`wxALIGN_CENTER_HORIZONTAL`	Horizontal alignment within allocated cell/column.
`wxSHAPED`	Preserve aspect-ratio-like shape behavior for suitable items.

E.9 Command-State Checklist

For each command ask:

1. Is the command currently available?
2. Is it checked/toggled?
3. Does it have a keyboard accelerator?
4. Is the same operation exposed by menu, toolbar, context menu, or palette?
5. Is the handler independent of the surface that invoked it?
6. Does it block the event loop?
7. Is failure reported in a useful, non-destructive way?

Why This Matters

Events are not merely callback syntax. They are the messaging fabric between native input, widgets, command state, background results, and application policy. Treat event design as architecture and large wxWidgets programs remain comprehensible.

CHAPTER F

Troubleshooting Playbook

Troubleshooting improves dramatically when each observation eliminates categories of causes. This appendix organizes problems by the stage at which they occur: configure, compile, link, load, start, interact, or shut down.

F.1 Stage 1: CMake Configure Fails

Symptoms include missing packages, unknown targets, compiler detection failure, or options apparently ignored.

Questions

1. Is the expected `cmake` executable running?
2. Which compiler paths did CMake cache?
3. Is wxWidgets being added as a source tree or found as an installed package?
4. Does the selected package actually expose the target names used by the project?
5. Is an old cache remembering another toolchain or another wxWidgets tree?

Actions

```
cmake --version
cmake -S . -B build -G Ninja
cmake -LAH -N build
```

Add temporary status messages for compiler paths/versions. If the fundamental environment changed, delete `build/` and configure from zero.

F.2 Stage 2: Compilation Fails

Header-not-found and C++ errors are compilation-stage failures. Do not diagnose them by copying DLLs.

Typical causes

- target not linked to the wxWidgets component that propagates required include paths;
- missing component such as STC/AUI/property grid;
- code written for another wxWidgets version;
- C++ standard/toolchain mismatch;
- platform-specific header included without the correct platform guard.

F.3 Stage 3: Linking Fails

Undefined references mean compilation produced objects but the linker could not resolve required symbols.

Ask what owns the symbol

A missing `wxStyledTextCtrl` symbol suggests the STC library/component. A missing Windows API symbol suggests a system import library or SDK mismatch. A C++ ABI symbol can indicate mixed runtimes/objects.

Always read the verbose link command before adding random libraries.

```
cmake --build build --verbose
```

F.4 Stage 4: Executable Builds but Windows Says Entry Point Not Found

This is a loader-stage problem. Capture the exact function and DLL.

Loader triage

1. What is the exact missing symbol?
2. Which DLL does the dialog name?
3. Does `objdump -p` show unexpected runtime DLLs?
4. Which physical DLL paths does `ntldd` resolve?
5. Are stale DLLs located beside the executable?
6. Is the expected application manifest embedded?
7. Is `app.rc` still part of the Windows target?

F.4.1 GetWindowSubclass specifically

If the missing function is `GetWindowSubclass`, inspect the common-control manifest path immediately. For the standard wxWidgets Windows setup, verify:

```
#define wxUSE_DPI_AWARE_MANIFEST 2
#include <wx/msw/wx.rc>
```

and:

```
    add_executable(MyApp WIN32 main.cpp app.rc)
```

Do this before spending hours cycling through unrelated compiler versions.

F.5 Stage 5: Program Starts but No Window Appears

Check:

- Did `OnInit()` return true?
- Was a top-level window created?
- Was `Show()` called?
- Did initialization throw/abort before the show call?
- Did a hidden/modal dialog block startup?
- Is the window restored to an off-screen monitor position from old settings?

Temporarily center the frame and remove geometry restoration to distinguish persistence errors from creation errors.

F.6 Stage 6: Window Is Blank or Controls Are Missing

Likely layout/parenting problems:

- control created with the wrong parent;
- sizer constructed but never assigned with `SetSizer()`;
- proportion/`wxEXPAND` rules prevent expected growth;
- control hidden or disabled unintentionally;
- minimum size collapses a pane;
- splitter has not been split.

Give containers temporary background colours/borders during debugging to reveal actual bounds.

F.7 Stage 7: Resize Behavior Is Bad

Do not fix every resize issue inside `wxEVT_SIZE`. First inspect the sizer tree. Most layout problems are constraint problems, not missing manual calls to `SetSize()`.

Test:

- very small window;
- very wide and very tall windows;
- 150–200% scaling;
- longer translated labels;
- larger system fonts.

F.8 Stage 8: Button/Menu Does Nothing

Check event routing:

1. Correct event type?
2. Correct ID?
3. Binding executed after object creation?
4. Handler signature matches event type?
5. Object receiving event still alive?
6. Another handler consuming/propagating event differently?

Add a temporary log line at handler entry. Prove whether routing failed before debugging the command body.

F.9 Stage 9: Interface Freezes

A frozen GUI usually means the event loop is blocked.

Search for:

- synchronous process execution;
- large filesystem scans;
- network calls;
- waiting on futures/threads;
- long parsing/indexing;
- sleep loops;
- locks held while waiting for GUI work.

Move work off the GUI thread, then post compact results back. Do not "fix" a freeze by sprinkling manual event pumping into arbitrary code.

F.10 Stage 10: Random Crashes During Close

Shutdown exposes lifetime assumptions.

Check:

- timers stopped before owners are destroyed;
- workers cancelled/joined or detached under a deliberate policy;
- child processes terminated or allowed to finish safely;
- callbacks cannot target destroyed windows;
- bindings/lambdas do not capture dead service objects;

- background events are ignored after shutdown begins.

Introduce an explicit application shutdown state for large tools.

F.11 Stage 11: Works in Debug, Fails in Release

Investigate undefined behavior, uninitialized data, lifetime races, timing assumptions, and assertions that performed unintended work. Enable sanitizers where supported and reduce the failing behavior to a minimal reproducible example.

F.12 Stage 12: Works on One PC Only

Capture a deployment report:

```
Compiler/toolchain:
wxWidgets version/config:
Architecture:
Imported DLLs:
Resolved DLL paths:
OS version:
Display scale:
Working directory:
Executable path:
```

Environmental differences are data, not noise.

F.13 Stage 13: Wrong MinGW Appears in the Link Command

IDE settings and terminal PATH are not proof. The link line is proof. If the command points into an IDE-bundled `mingw` directory while you intended MSYS2/WinLibs, change the selected CMake toolchain/profile and delete the old build tree.

F.14 Stage 14: Multiple MinGW Installations

Multiple toolchains are perfectly valid if their outputs remain isolated. Problems begin when one build resolves headers/libraries/runtime DLLs from another installation.

Use:

```
where.exe gcc
where.exe g++
where.exe libstdc++-6.dll
where.exe libgcc_s_seh-1.dll
where.exe libwinpthread-1.dll
```

Inside MSYS2:

```
which gcc
which g++
echo $MSYSTEM
```

F.15 Stage 15: MSYS2 Shell Has No gcc

If `which gcc` reports nothing and PATH contains only `/usr/bin`, you likely opened the plain MSYS shell rather than a MinGW environment such as UCRT64. Open the intended environment and confirm `/ucrt64/bin` (or chosen environment) appears in PATH.

F.16 Stage 16: Source Editor Performance Drops

Measure before redesigning:

- document size;
- time spent in change/update handlers;
- full-document lexer/parser invocations;
- diagnostic marker count;
- repeated status/title updates;
- synchronous project/model refreshes.

Debounce semantic analysis and process only the changed scope where possible. Let Scintilla handle text editing; avoid maintaining duplicate full text buffers in the UI layer unless required.

F.17 Stage 17: External NASM Does Not Start

Check:

1. executable path exists or is discoverable;
2. command path/arguments are quoted;
3. working directory is intentional;
4. output path parent directory exists;
5. selected object format is supported;
6. security/antivirus policies did not block execution;
7. redirected streams are drained.

Expose the final command/tool profile in a diagnostic view so the user can reproduce it in a terminal.

F.18 Stage 18: Diagnostics Point to Wrong Lines

Clarify indexing:

- tools usually report one-based source lines;
- Scintilla line APIs commonly use zero-based indices;
- columns may be bytes, code units, Unicode characters, or display columns depending on the tool.

Convert at one boundary and name methods explicitly, e.g. `GotoLineOneBased()`.

F.19 Stage 19: High-DPI Problems

Verify both halves:

1. executable declares appropriate DPI awareness via manifest/framework setup;
2. layout/assets/code do not assume physical pixels.

Use `FromDIP()`, sizers, scalable imagery, and real multi-scale testing.

F.20 Stage 20: When to Make a Minimal Reproducer

Create a tiny reproducer when:

- a problem survives two substantially different application contexts;
- you cannot tell whether wxWidgets or your architecture owns the behavior;
- loader/toolchain diagnostics are contaminated by a large build;

- you are about to report an upstream bug.

A good reproducer has one CMake file, one or two source files, the required resource file, and exact configure/build/run commands.

Why This Matters

The shortest troubleshooting path is usually: classify the stage, capture exact evidence, eliminate one category at a time, and preserve a known-good minimal baseline. Random toolchain changes generate more states; they do not generate understanding.

References and Further Study

The wxWidgets API evolves, especially on the 3.3 development branch. Use this book for explanation and architecture, but verify exact signatures, availability, and incompatible changes against the official documentation for the version you build.

Primary wxWidgets Sources

1. **wxWidgets Project.** Official website, downloads, news, supported platforms, and project information. https://wxwidgets.org/
2. **wxWidgets 3.3 Reference Manual.** Development-branch API reference and overviews. Available from https://wxwidgets.org/docs/
3. **wxWidgets Windows FAQ.** Windows resources, manifest/theming guidance, DPI notes, and common platform questions. https://wxwidgets.org/docs/faq/windows/
4. **wxWidgets Source Repository.** Source code, samples, CMake configuration, issue history, and implementation details. https://github.com/wxWidgets/wxWidgets
5. **wxWidgets Coding Guidelines.** Particularly useful when reading or contributing to wxWidgets 3.3+ source. https://wxwidgets.org/develop/coding-guidelines/

Build and Language Tools

1. **CMake Documentation.** https://cmake.org/documentation/
2. **Ninja Build System.** https://ninja-build.org/
3. **GCC Documentation.** https://gcc.gnu.org/onlinedocs/
4. **Clang Documentation.** https://clang.llvm.org/docs/
5. **MSYS2 Documentation.** Environments, package management, UCRT64/MINGW64 distinctions, and CMake usage. https://www.msys2.org/docs/

Editor and Assembler Technologies

1. **Scintilla.** Editing component underlying wxStyledTextCtrl. https://www.scintilla.org/
2. **NASM - Netwide Assembler.** Documentation, command-line options, object formats, and language reference. https://www.nasm.us/

Windows Platform References

1. **Microsoft Windows Desktop Documentation.** Windows API, application manifests, common controls, DPI, accessibility, and deployment material. https://learn.microsoft.com/windows/win32/
2. **GetWindowSubclass function.** Common-control subclass helper and version requirements. Search the current Windows SDK documentation for `GetWindowSubclass` before relying on details in older material.

Books and Historical Background

1. Julian Smart, Kevin Hock with Stefan Csomor, *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005. A major historical book on application development with wxWidgets; API details should be interpreted in the context of its era.

Validation Baseline for This Manuscript

The manuscript is organized around wxWidgets 3.3.3, released July 7, 2026, and modern C++23 examples. wxWidgets identifies 3.3.x as the development branch leading toward the next stable series, while 3.2.x remains the stable branch at the time of writing. Readers who need long-term ABI/API stability should evaluate the stable branch; readers following the newest features should review the 3.3 incompatible-changes notes when upgrading.

Publication validation requirement

Before publishing a final release of this book, run the example matrix on the toolchains the book claims to support and record the exact results. AI assistance and careful review are valuable, but they are not substitutes for compiling, linking, launching, and exercising the examples on real target environments.

Index