Cross-Platform GUI Programming with wxWidgets

By Julian Smart, Kevin Hock, Stefan Csomor

.....

Publisher: **Prentice Hall PTR**
Pub Date: **July 25, 2005**
ISBN: **0-13-147381-6**
Pages: **744**

Table of Contents | Index

Overview

"This book is the best way for beginning developers to learn wxWidgets programming in C++. It is a must-have for programmers thinking of using wxWidgets and those already using it."

Mitch Kapor, founder of Lotus Software and the Open Source Applications Foundation

- Build advanced cross-platform applications that support native look-and-feel on Windows, Linux, Unix, Mac OS X, and even Pocket PC
- Master wxWidgets from start to finisheven if you've never built GUI applications before
- Leverage advanced wxWidgets capabilities: networking, multithreading, streaming, and more
- CD-ROM: library of development tools, source code, and sample applications
- Foreword by Mitch Kapor, founder, Lotus Development and Open Source Application Foundation

wxWidgets is an easy-to-use, open source C++ API for writing GUI applications that run on Windows, Linux, Unix, Mac OS X, and even Pocket PCsupporting each platform's native look and feel with virtually no additional coding. Now, its creator and two leading developers teach you all you need to know to write robust cross-platform software with wxWidgets. This book covers everything from dialog boxes to drag-and-drop, from networking to multithreading. It includes all the tools and code you need to get great results, fast. From AMD to AOL, Lockheed Martin to Xerox, world-class developers are using wxWidgets to save money, increase efficiency, and reach new markets. With this book, you can, too.

- wxWidgets quickstart: event/input handling, window layouts, drawing, printing, dialogs, and more
- Working with window classes, from simple to advanced
- Memory management, debugging, error checking, internationalization, and other advanced topics
- Includes extensive code samples for Windows, Linux (GTK+), and Mac OS X

About the CD-ROM

The CD-ROM contains all of the source code from the book; wxWidgets distributions for Windows, Linux, Unix, Mac OS X, and other platforms; the wxWidgets reference guide; and development tools including the OpenWatcom C++ compiler, the poEdit translation helper, and the DialogBlocks user interface builder.

Cross-Platform GUI Programming with wxWidgets

By Julian Smart, Kevin Hock, Stefan Csomor

Publisher: **Prentice Hall PTR**
Pub Date: **July 25, 2005**
ISBN: **0-13-147381-6**
Pages: **744**

Table of Contents | Index

- Copyright
- Bruce Perens ' Open Source Series
- About Prentice Hall Professional Technical Reference
- Foreword
- Preface
  - Who This Book Is For
  - The CD-ROM
  - How to Use This Book
  - Conventions
  - Chapter Summary
- Acknowledgments
- About the Authors
- Chapter 1. Introduction
  - What Is wxWidgets?
  - Why Use wxWidgets?
  - A Brief History of wxWidgets
  - The wxWidgets Community
  - wxWidgets and Object-Oriented Programming
  - License Considerations
  - The wxWidgets Architecture
  - Summary
- Chapter 2. Getting Started
  - A Small wxWidgets Sample
  - The Application Class
  - The Frame Class
  - The Event Handlers
  - The Frame Constructor
  - The Whole Program
  - Compiling and Running the Program
  - Program Flow
  - Summary
- Chapter 3. Event Handling
  - Event-Driven Programming
  - Event Tables and Handlers
  - Skipping Events
  - Pluggable Event Handlers
  - Dynamic Event Handlers
  - Window Identifiers
  - Defining Custom Events
  - Summary

Chapter 4. Window Basics
Anatomy of a Window
A Quick Guide to the Window Classes
Base Window Classes
Top-Level Windows
Container Windows
Non-Static Controls
Static Controls
Menus
Control Bars
Summary
Chapter 5. Drawing and Printing
Understanding Device Contexts
Drawing Tools
Device Context Drawing Functions
Using the Printing Framework
3D Graphics with wxGLCanvas
Summary
Chapter 6. Handling Input
Mouse Input
Handling Keyboard Events
Handling Joystick Events
Summary
Chapter 7. Window Layout Using Sizers
Layout Basics
Sizers
Programming with Sizers
Further Layout Issues
Summary
Chapter 8. Using Standard Dialogs
Informative Dialogs
File and Directory Dialogs
Choice and Selection Dialogs
Entry Dialogs
Printing Dialogs
Summary
Chapter 9. Writing Custom Dialogs
Steps in Creating a Custom Dialog
An Example: PersonalRecordDialog
Adapting Dialogs for Small Devices
Further Considerations in Dialog Design
Using wxWidgets Resource Files
Summary
Chapter 10. Programming with Images
Image Classes in wxWidgets
Programming with wxBitmap
Programming with wxIcon
Programming with wxCursor
Programming with wxImage
Image Lists and Icon Bundles
Customizing Art in wxWidgets
Summary
Chapter 11. Clipboard and Drag and Drop
Data Objects
Using the Clipboard

Implementing Drag and Drop
Summary
Chapter 12. Advanced Window Classes
wxTreeCtrl
wxListCtrl
wxWizard
wxHtmlWindow
wxGrid
wxTaskBarIcon
Writing Your Own Controls
Summary
Chapter 13. Data Structure Classes
Why Not STL?
Strings
wxArray
wxList and wxNode
wxHashMap
Storing and Processing Dates and Times
Helper Data Structures
Summary
Chapter 14. Files and Streams
File Classes and Functions
Stream Classes
Summary
Chapter 15. Memory Management, Debugging, and Error Checking
Memory Management Basics
Detecting Memory Leaks and Other Errors
Facilities for Defensive Programming
Error Reporting
Providing Run-Time Type Information
Using wxModule
Loading Dynamic Libraries
Exception Handling
Debugging Tips
Summary
Chapter 16. Writing International Applications
Introduction to Internationalization
Providing Translations
Character Encodings and Unicode
Numbers and Dates
Other Media
A Simple Sample
Summary
Chapter 17. Writing Multithreaded Applications
When to Use Threads, and When Not To
Using wxThread
Synchronization Objects
The wxWidgets Threads Sample
Alternatives to Multithreading
Summary
Chapter 18. Programming with wxSocket
Socket Classes and Functionality Overview
Introduction to Sockets and Basic Socket Processing
Socket Flags
Using Socket Streams

Alternatives to wxSocket
Summary
Chapter 19. Working with Documents and Views
Document/View Basics
Other Document/View Capabilities
Strategies for Implementing Undo/Redo
Summary
Chapter 20. Perfecting Your Application
Single Instance or Multiple Instances?
Modifying Event Handling
Reducing Flicker
Implementing Online Help
Parsing the Command Line
Storing Application Resources
Invoking Other Applications
Managing Application Settings
Application Installation
Following UI Design Guidelines
Summary
Appendix A. Installing wxWidgets
Choosing Your Development Tools
Downloading and Unpacking wxWidgets
Configuration/Build Options
WindowsMicrosoft Visual Studio
WindowsMicrosoft Visual C++ Command Line
WindowsBorland C++
WindowsMinGW with MSYS
WindowsMinGW without MSYS
Linux, Unix, and Mac OS XGCC
Modifying Setup.h for Further Customizations
Rebuilding After Updating wxWidgets Files
Using Contrib Libraries
Appendix B. Building Your Own wxWidgets Applications
WindowsMicrosoft Visual Studio
LinuxKDevelop
Mac OS XXcode
Any PlatformMakefiles
Cross-Platform Builds Using Bakefile
Using wx-config
wxWidgets Symbols and Headers
Appendix C. Creating Applications with DialogBlocks
What is DialogBlocks?
Installing DialogBlocks
The DialogBlocks Interface
The DialogBlocks Sample Project
Compiling the Sample
Creating a New Project
Creating a Dialog
Creating a Frame
Creating an Application Object
Debugging Your Application
Further Information
Appendix D. Other Features in wxWidgets
Further Window Classes
ODBC Classes

MIME Types Manager
Network Functionality
Multimedia Classes
Embedded Web Browsers
Accessibility
OLE Automation
Renderer Classes
Event Loops
Appendix E. Third-Party Tools for wxWidgets
Language Bindings
Tools
Add-on Libraries
Appendix F. wxWidgets Application Showcase
Appendix G. Using the CD-ROM
Browsing the CD-ROM
The CD-ROM Contents
Appendix H. How wxWidgets Processes Events
Appendix I. Event Classes and Macros
Appendix J. Code Listings
Custom Dialog Class Implementation
wxWizard Sample Code
Appendix K. Porting from MFC
General Observations
Feature Comparison
Equivalent Functionality
Further Information
GLOSSARY
Index

Copyright

Library of Congress Catalog Number: 2005924108

Printed in the United States of America.

Text printed in the United States on recycled paper at R.R. Donnelley & Sons in Crawfordsville, Indiana.

First printing, July 2005

Bruce Perens ' Open Source Series

http://www.phptr.com/perens

- Java™ Application Development on Linux®

  Carl Albing and Michael Schwarz
- C++ GUI Programming with Qt 3

  Jasmin Blanchette and Mark Summerfield
- Managing Linux Systems with Webmin: System Administration and Module Development

  Jamie Cameron
- The Linux Book

  David Elboth
- Understanding the Linux Virtual Memory Manager

  Mel Gorman
- PHP 5 Power Programming

  Andi Gutmans, Stig Bakken, and Derick Rethans
- Linux® Quick Fix Notebook

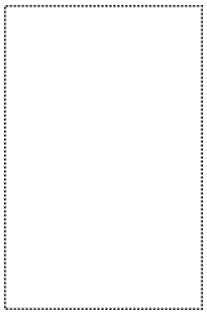
  Peter Harrison
- Linux Desk Reference, Second Edition

  Scott Hawkins
- Implementing CIFS: The Common Internet File System

About Prentice Hall Professional Technical Reference

With origins reaching back to the industry's first computer science publishing program in the 19605, and formally launched as its own imprint in 1986, Prentice Hall Professional Technical Reference (PH PTR) has developed into the leading provider of technical books in the world today. Our editors now publish over 200 books annually, authored by leaders in the fields of computing, engineering, and business.

Our roots are firmly planted in the soil that gave rise to the technical revolution. Our bookshelf contains many of the industry's computing and engineering classics: Kernighan and Ritchie's C Programming Language, Nemeth's UNIX System Administration Handbook, Horstmann's Core Java, and Johnson's High-Speed Digital Design.



PH PTR acknowledges its auspicious beginnings while it looks to the future for inspiration. We continue to evolve and break new ground in publishing by providing today's professionals with tomorrow's solutions.

Foreword

It's a pleasure to introduce you to Cross-Platform GUI Programming with wxWidgets, the first book on wxWidgets since it was originally released more than a decade ago.

wxWidgets is a first-class, open source response to the need for portability in an increasingly heterogeneous computing world. Being tied to specific hardware or a single operating system is often undesirable and sometimes impermissible, hence the well-understood need for cross-platform GUI frameworks. Given the long life of open source products and the often-transient nature of proprietary solutions, developers are wise to base their applications on an infrastructure that is going to survive long-term, as wxWidgets has and will continue to do.

wxWidgets combines countless years' worth of hard-earned wisdom contributed by developers worldwide, abstracting functionality and finding solutions for platform-specific issues. You, the developer, are protected both from shifts in computing trends and from the intricacies and frustrations of each platform's native API.

Becoming a wxWidgets user is an invitation into a community that spans individuals, startups, government organizations, large companies, and open source projects. When you contribute, you are forging a connection between yourself and a community that is broadly representative of the reach of information technology in the 21st century. wxWidgets-based applications may be found not just in the software industry but also in medicine, archaeology, physics, astronomy, processor manufacturing, education, geological exploration, the transport industry, space exploration, and many other fields as well.

"Chandler," the Personal Information Manager now under development at the Open Source Applications Foundation, uses wxWidgets to run under Windows, Mac OS X, and Linux. Some of our developers have become active contributors to the wxWidgets project, following the virtuous circle of open source development.

We look forward to having you join us in the ever-growing community of developers using wxWidgets, and I personally wish you all the best with your wxWidgets projects.

Mitch Kapor, Chair
OSAF
June 2005

Preface

Who This Book Is For

The CD-ROM

How to Use This Book

Conventions

Chapter Summary

Who This Book Is For

This book is a guide to using wxWidgets, an open-source construction kit for writing sophisticated C++ applications targeting a variety of platforms, including Windows, Linux, Mac OS X, and Pocket PC. With help from this book, a competent programmer can create multi-platform applications with confidence. Developers already familiar with wxWidgets should also find it useful for brushing up their knowledge.

This book is accessible to developers with a variety of experience and backgrounds. You may come from a Windows or Unix perspective; you may previously have experience in MFC, OWL, Win32, Mac OS, Motif, or console-mode Unix programming. Or perhaps you have come from a different career entirely and are looking for a way to get up to speed on multiple platforms. The book can't specifically cover the details of the C++ language, but it's common for people to successfully learn C++ and wxWidgets at the same time, and the straightforward nature of the wxWidgets API makes this process easier. The reader does not need to know more advanced C++ techniques like templates, streams, and exceptions. However, wxWidgets does not prevent you from using these techniques.

Managers will find the book useful in discovering what wxWidgets can do for them, particularly in Chapter 1, "Introduction." The combination of the book and the resources on the accompanying CD-ROM will give your staff all they need for getting started on cross-platform programming projects. You'll see how wxWidgets puts tools of tremendous power into your hands, with benefits that include:

- Cost savings from writing code once that will compile on Windows, Unix, Mac OS X, and other platforms.
- Customer satisfaction from delivering stable, fast, attractive applications with a native look and feel.
- Increased productivity from the wide variety of classes that wxWidgets provides, both for creating great GUIs and for general application development.
- Increased market share due to support for platforms you may not have previously considered, and the ability to internationalize your applications.
- Support from a large, active wxWidgets community that answers questions helpfully and provides prompt bug fixing. The sample of third-party add-ons listed in Appendix E, "Third-Party Tools for wxWidgets," is evidence of a thriving ecosystem.
- Access to the source for enhancement and trouble-shooting.

This is a guide to writing wxWidgets application with C++, but you can use a variety of other languages such as Python, Perl, a BASIC variant, Lua, Eiffel, JavaScript, Java, Ruby, Haskell, and C#. Some of these bindings are more advanced than others. For more information, please see Appendix E and the wxWidgets web site at http://www.wxwidgets.org.

We focus on three popular desktop platforms: Microsoft Windows, Linux using GTK+, and Mac OS X. However, most of the book also applies to other platforms supported by wxWidgets. In particular, wxWidgets can be used with most Unix variants.

The CD-ROM

The CD-ROM contains example code from the book, the wxWidgets 2.6 distribution for Windows, Linux, Mac OS X, and other platforms, and several tools to help you use wxWidgets, including the translation tool poEdit. For Windows users, we supply three free compilers you can use with wxWidgets: MinGW, Digital Mars C++, and OpenWatcom C++.

In addition, we provide you with DialogBlocks Personal Edition, a sophisticated rapid application development (RAD) tool for you to create complex windows with very little manual coding. You can use it to compile and run samples that accompany the book as well as to create your own applications for personal use, and it also provides convenient access to the wxWidgets reference manual.

Updates to the book and CD-ROM can be obtained from this site:

http://www.wxwidgets.org/book

How to Use This Book

It's advisable to read at least Chapters 1 through 10 in order, but you can skip to other chapters if you need to complete a particular task. If you haven't installed wxWidgets before, you may want to look at Appendix A, "Installing wxWidgets," early on. MFC programmers will find it useful to read Appendix K, "Porting from MFC," as a point of reference.

Because this book is not a complete API reference, you'll find it useful to keep the wxWidgets reference manual open. The reference manual is available in a number of formats, including Windows HTML Help and PDF, and it should be in your wxWidgets distribution; if not, it can be downloaded from the wxWidgets web site. You can also refer to the many samples in the wxWidgets distribution to supplement the examples given in this book.

Note that the book is intended to be used in conjunction with wxWidgets 2.6 or later. The majority of the book will apply to earlier versions, but be aware that some functionality will be missing, and in a small number of cases, the behavior may be different. In particular, sizer behavior changed somewhat between 2.4 and 2.5. For details, please see the topic "Changes Since 2.4.x" in the wxWidgets reference manual.

Conventions

For code examples, we mostly follow the wxWidgets style guidelines, for example:

- Words within class names and functions have an initial capital, for example MyFunkyClass.
- The m_ prefix denotes a member variable, s_ denotes a static variable, g_ denotes a global variable; local variables generally start with a lowercase letter, for example textCtrl.

You can find more about the wxWidgets style guidelines at http://www.wxwidgets.org/standard.htm.

Sometimes we'll also use comments that can be parsed by the documentation tool Doxygen, such as:

```
/*! A class description
 */

/// A function description
```

Classes, functions, identifiers, variables, and standard wxWidgets objects are marked with a teletype font in the text. User interface commands, such as menu and button labels, are marked in italics.

Chapter Summary

Chapter 1: Introduction

What is wxWidgets, and why use it? A brief history; the wxWidgets community; the license; wxWidgets ports and architecture explained.

Chapter 2: Getting Started

A small wxWidgets sample: the application class; the main window; the event table; an outline of program flow.

Chapter 3: Event Handling

Event tables and handlers; how a button click is processed; skipping events; pluggable and dynamic event handlers; defining custom events; window identifiers.

Chapter 4: Window Basics

The main features of a window explained; a quick guide to the commonest window classes; base window classes such as wxWindow; top-level windows; container windows; non-static controls; static controls; menus; control bars.

Chapter 5: Drawing and Printing

Device context principles; the main device context classes described; buffered drawing; drawing tools; device context drawing functions; using the printing framework; 3D graphics with wxGLCanvas.

Chapter 6: Handling Input

Handling mouse and mouse wheel events; handling keyboard events; keycodes; modifier key variations; accelerators; handling joystick events.

Chapter 7: Window Layout Using Sizers

Layout basics; sizers introduced; common features of sizers; programming with sizers. Further layout issues: dialog units; platform-adaptive layouts; dynamic layouts.

Chapter 8: Using Standard Dialogs

Informative dialogs such as wxMessageBox and wxProgressDialog; file and directory dialogs such as wxFileDialog; choice and selection dialogs such as wxColourDialog and wxFontDialog; entry dialogs such as wxTextEnTRyDialog and wxFindReplaceDialog; printing dialogs: wxPageSetupDialog and wxPrintDialog.

Chapter 9: Writing Custom Dialogs

Steps in creating a custom dialog; an example: PersonalRecordDialog; deriving a new class; designing data storage; coding the controls and layout; data transfer and validation; handling events; handling UI updates; adding help; adapting dialogs for small devices; further considerations in dialog design; using wxWidgets resource files; loading resources; using binary and embedded resource files; translating resources; the XRC format; writing resource handlers; foreign controls.

Chapter 10: Programming with Images

Image classes in wxWidgets; programming with wxBitmap; programming with wxIcon; programming with wxCursor; programming with wxImage; image lists and icon bundles; customizing wxWidgets graphics with wxArtProvider.

Acknowledgments

wxWidgets owes its success to the hard work of many talented people. We would like to thank them all, with special consideration for that essential support network: our long-suffering families and partners. wxWidgets supporters and contributors include the following (apologies for any unintentional omissions):

Yiorgos Adamopoulos, Jamshid Afshar, Alejandro Aguilar-Sierra, Patrick Albert, Bruneau Babet, Mitchell Baker, Mattia Barbon, Nerijus Baliunas, Karsten Ballueder, Jonathan Bayer, Michael Bedward, Kai Bendorf, Yura Bidus, Jorgen Bodde, Borland, Keith Gary Boyce, Chris Breeze, Sylvain Bougnoux, Wade Brainerd, Pete Britton, Ian Brown, C. Buckley, Doug Card, Marco Cavallini, Dmitri Chubraev, Robin Corbet, Cecil Coupe, Stefan Csomor, Andrew Davison, Gilles Depeyrot, Duane Doran, Neil Dudman, Robin Dunn, Hermann Dunkel, Jos van Eijndhoven, Chris Elliott, David Elliott, David Falkinder, Rob Farnum, Joel Farley, Tom Felici, Thomas Fettig, Matthew Flatt, Pasquale Foggia, Josep Fortiana, Todd Fries, Dominic Gallagher, Roger Gammans, Guillermo Rodriguez Garcia, Brian Gavin, Wolfram Gloger, Aleksandras Gluchovas, Markus Greither, Norbert Grotz, Stephane Gully, Stefan Gunter, Bill Hale, Patrick Halke, Stefan Hammes, Guillaume Helle, Harco de Hilster, Kevin Hock, Cord Hockemeyer, Klaas Holwerda, Markus Holzem, Ove Kaaven, Mitch Kapor, Matt Kimball, Hajo Kirchoff, Olaf Klein, Jacob Jansen, Leif Jensen, Mark Johnson, Bart Jourquin, John Labenski, Guilhem Lavaux, Ron Lee, Hans Van Leemputten, Peter Lenhard, Jan Lessner, Nicholas Liebmann, Torsten Liermann, Per Lindqvist, Jesse Lovelace, Tatu Männistö, Lindsay Mathieson, Scott Maxwell, Bob Mitchell, Thomas Myers, Oliver Niedung, Stefan Neis, Ryan Norton, Robert O'Connor, Jeffrey Ollie, Kevin Ollivier, William Osborne, Hernan Otero, Ian Perrigo, Timothy Peters, Giordano Pezzoli, Harri Pasanen, Thomaso Paoletti, Garrett Potts, Robert Rae, Marcel Rasche, Mart Raudsepp, Andy Robinson, Robert Roebling, Alec Ross, Gunnar Roth, Thomas Runge, Tom Ryan, Dino Scaringella, Jobst Schmalenbach, Dimitri Schoolwerth, Arthur Seaton, Paul Shirley, Wlodzimierz Skiba, John Skiff, Vaclav Slavik, Brian Smith, Neil Smith, Stein Somers, Petr Smilauer, Kari Systä, George Tasker, Austin Tate, Arthur Tetzlaff-Deas, Paul Thiessen, Jonathan Tonberg, Jyrki Tuomi, Janos Vegh, Andrea Venturoli, David Webster, Michael Wetherell, Otto Wyss, Vadim Zeitlin, Xiaokun Zhu, Zbigniew Zagórski, Edward Zimmermann. Thanks also to Dotsrc.org and SourceForge for hosting project services.

Thanks are due in particular to Vadim Zeitlin, Vaclav Slavik, Robert Roebling, Stefan Csomor, and Robin Dunn for permission to adapt some of their contributions to the wxWidgets reference manual.

Special thanks go to Stefan Csomor who contributed Chapter 16 and Chapter 17, and to Kevin Ollivier who wrote the Bakefile tutorial in Appendix B. We would also like to thank Mitch Kapor for writing the foreword.

We are very grateful to Mark Taub for his patience and advice throughout. A big thank you goes to Marita Allwood, Harriet Smart, Antonia Smart, Clayton Hock, and Ethel Hock for all their love, support, and encouragement. A debt is also owed to all those who have reviewed and suggested improvements to the book, including: Stefan Csomor, Dimitri Schoolwerth, Robin Dunn, Carl Godkin, Bob Paddock, Chris Elliott, Michalis Kabrianis, Marc-Andre Lureau, Jonas Karlsson, Arnout Engelen, Erik van der Wal, Greg Smith, and Alexander Stigsen.

Finally, we hope that you enjoy reading this book and, most importantly, have fun using wxWidgets to build great-looking, multi-platform applications!

Julian Smart and Kevin Hock
June 2005

About the Authors

Julian Smart has degrees from the University of St. Andrews and the University of Dundee. After working on model-based reasoning at the Scottish Crop Research Institute, he moved to the Artificial Intelligence Applications Institute at the University of Edinburgh, where he founded the wxWidgets project in 1992. Since starting Anthemion Software in 1996, Julian has been helping other companies deploy wxWidgets, and he sells tools for programmers, including DialogBlocks and HelpBlocks. He has worked as a consultant for various companies including Borland and was a member of Red Hat's eCos team, writing GUI tools to support the embedded operating system. In 2004, Julian and his wife Harriet launched a consumer product for fiction writers called Writer's Café, written with wxWidgets. Julian and Harriet live in Edinburgh with their daughter Toni.

Kevin Hock has degrees from Miami University (Oxford, Ohio) in Computer Science and Accounting and has taught courses at Miami in both Java and client-server systems. In 2002, he started work on an instant messaging system and founded BitWise Communications, LLC, in 2003, offering both professional and personal instant messaging. During the course of developing BitWise using wxWidgets, Kevin became a wxWidgets developer and has provided enhancements to all platforms. Kevin lives in Oxford, Ohio.

Stefan Csomor is director and owner of Advanced Concepts AG, a company that specializes in cross-platform development and consulting. In addition to being a qualified medical doctor, he has more than 15 years of experience in object-oriented programming and has been writing software for 25 years. Stefan is the main author of the Mac OS port of wxWidgets.

Chapter 1. Introduction

In this chapter, we answer a few basic questions about what wxWidgets is and what sets it apart from other solutions. We outline the project's history, how the wxWidgets community works, how wxWidgets is licensed, and an overview of the architecture and available parts.

What Is wxWidgets?

wxWidgets is a programmer's toolkit for writing desktop or mobile applications with graphical user interfaces (GUIs). It's a framework, in the sense that it does a lot of the housekeeping work and provides default application behavior. The wxWidgets library contains a large number of classes and methods for the programmer to use and customize. Applications typically show windows containing standard controls, possibly drawing specialized images and graphics and responding to input from the mouse, keyboard, or other sources. They may also communicate with other processes or drive other programs. In other words, wxWidgets makes it relatively easy for the programmer to write an application that does all the usual things modern applications do.

While wxWidgets is often labeled a GUI development toolkit, it is in fact much more than that and has features that are useful for many aspects of application development. This has to be the case because all of a wxWidgets application needs to be portable to different platforms, not just the GUI part. wxWidgets provides classes for files and streams, multiple threads, application settings, interprocess communication, online help, database access, and much more.

Why Use wxWidgets?

One area where wxWidgets differs from many other frameworks, such as MFC or OWL, is its multi-platform nature. wxWidgets has an Application Programming Interface (API) that is the same, or very nearly the same, on all supported platforms. This means that you can write an application on Windows, for example, and with very few changes (if any) recompile it on Linux or Mac OS X. This has a huge cost benefit compared with completely rewriting an application for each platform, and it also means that you do not need to learn a different API for each platform. Furthermore, it helps to future-proof your applications. As the computing landscape changes, wxWidgets changes with it, allowing your application to be ported to the latest and greatest systems supporting the newest features.

Another distinguishing feature is that wxWidgets provides a native look and feel. Some frameworks use the same widget code running on all platforms, perhaps with a theme makeover to simulate each platform's native appearance. By contrast, wxWidgets uses the native widgets wherever possible (and its own widget set in other cases) so that not only does the application look native on the major platforms, but it actually is native. This is incredibly important for user acceptance because even small, almost imperceptible differences in the way an application behaves, compared with the platform standard, can create an alienating experience for the user. To illustrate, Figure 1-1 shows a wxWidgets application called StoryLines, a tool to help fiction writers plot their stories, running on Windows XP.

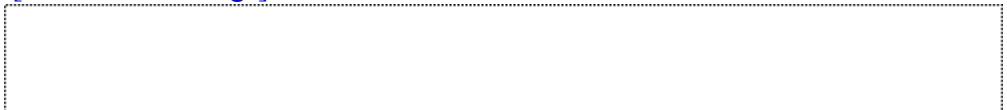
Figure 1-1. StoryLines on Windows

[View full size image]

It's recognizably a Windows application, with GUI elements such as tabs, scrollbars, and drop-down lists conforming to the current Windows theme. Similarly, Figure 1-2 shows StoryLines as a Mac OS X application, with the expected Aqua look and feel. There is no menu bar attached to the StoryLines window because it follows the Mac OS convention of showing the current window's menu bar at the top of the screen.

Figure 1-2. StoryLines on Mac OS X

[View full size image]



A Brief History of wxWidgets

The wxWidgets project started life in 1992 when Julian Smart was working at the University of Edinburgh on a diagramming tool called Hardy. He didn't want to choose between deploying it either on Sun workstations or PCs, so he decided to use a cross-platform framework. Because the range of existing cross-platform frameworks was limited, and the department didn't have a budget for it anyway, there was little choice but to write his own. The university gave him permission to upload wxWidgets 1.0 to the department's FTP site in September 1992, and other developers began to use the code. Initially, wxWidgets targeted XView and MFC 1.0; Borland C++ users complained about the requirement for MFC, so it was rewritten to use pure Win32. Because XView was giving way to Motif, a Motif port quickly followed.

Over time, a small but enthusiastic community of wxWidgets users was established and a mailing list created. Contributions and fixes were sent in, including an Xt port by Markus Holzem. wxWidgets gradually picked up more and more users from all over the world: individuals, academics, government departments, andmost gratifying of allcorporate users who found that wxWidgets offered a better product and better support than the commercial products they had looked at or used.

In 1997, a new wxWidgets 2 API was designed with help from Markus Holzem. Wolfram Gloger suggested that wxWidgets should be ported to GTK+, the up-and-coming widget set being adopted for the GNOME desktop environment. Robert Roebling became the lead developer for wxGTK, which is now the main Unix/Linux port of wxWidgets. In 1998, the Windows and GTK+ ports were merged and put under CVS control. Vadim Zeitlin joined the project to contribute huge amounts of design and code, and Stefan Csomor started a Mac OS port, also in 1998.

1999 saw the addition of Vaclav Slavik's impressive wxHTML classes and the HTML-based help viewer. In 2000, SciTech, Inc. sponsored initial development of wxUniversal, wxWidgets's own set of widgets for use on platforms that have no widget set of their own. wxUniversal was first used in SciTech's port to MGL, their low-level graphics layer.

In 2002, Julian Smart and Robert Roebling added the wxX11 port using the wxUniversal widgets. Requiring only Unix and X11, wxX11 is suitable for any Unix environment and can be used in fairly low-spec systems.

In July 2003, wxWidgets started running on Windows CE, and Robert Roebling demonstrated wxGTK applications running on the GPE embedded Linux platform.

In 2004, wxWidgets was renamed from the original moniker "wxWindows," after objections from Microsoft based on its Windows trademark.

Also during 2004, Stefan Csomor and a host of other contributors completely revamped wxMac for OS X, significantly improving the appearance and functionality of OS X applications. A port using Cocoa was also steadily improved, led by David Elliot, and William Osborne won our challenge to deliver an embryonic Palm OS 6 port that supports the wxWidgets "minimal" sample. Version 2.6 was released in April 2005, incorporating major improvements to all ports.

Future plans for wxWidgets include

- A package management tool, to make it easier to integrate third-party components
- Improved support for embedded applications
- Alternative event handling mechanisms
- Enhanced controls, such as a combined tree and list control

The wxWidgets Community

The wxWidgets community is a vibrant one, with two mailing lists: wx-users (for users) and wx-dev (for contributors). The web site has news, articles, and links to releases, and there is also the wxWidgets "Wiki," a set of web pages where everyone can add information. A forum is also available for developers and users alike. Here's a list of the web addresses for these resources:

- http://www.wxwidgets.org: the wxWidgets home page
- http://lists.wxwidgets.org: the mailing list archives
- http://wiki.wxwidgets.org: the wxWidgets Wiki
- http://www.wxforum.org: the wxWidgets forum

As with most open source projects, wxWidgets is developed using a CVS repository, a source management system that keeps track of code history. In order to prevent a chaotic free-for-all, a small number of developers have write access to CVS, and others can contribute by posting bug reports and patches (currently handled by SourceForge's trackers). Development occurs on two main branches: the "stable" branch, where only binary-compatible bug fixes are allowed, and the "development" branch (CVS head). So-called stable releases are even-numbered (for example, 2.4.x) and development releases are odd-numbered (for example, 2.5.x). Users can wait for new releases or download the source from the appropriate branch by anonymous CVS.

Decisions about API changes and other technical issues are made by consensus, usually by discussion on the wx-dev list. As well as the main wxWidgets community, many projects have spun off and enjoy their own communitiesfor example wxPython and wxPerl (see Appendix E, "Third-Party Tools for wxWidgets").

wxWidgets and Object-Oriented Programming

Like all modern GUI frameworks, wxWidgets benefits from heavy use of object-oriented programming concepts. Each window is represented as a C++ object; these objects have well-defined behavior, and can receive and react to events. What the user sees is the visual manifestation of this interacting system of objects. Your job as a developer is to orchestrate these objects' collective behavior, a task made easier by the default behaviors that wxWidgets implements for you.

Of course, it's no coincidence that object-oriented programming and GUIs mesh wellthey grew up together. The object-oriented language Smalltalk designed by Alan Kay and others in the 1970s was an important milestone in GUI history, making innovations in user interface technology as well as language design, and although wxWidgets uses a different language and API, the principles employed are broadly the same.

License Considerations

The wxWidgets license (officially, the "wxWindows License" for legal and historical reasons) is L-GPL with an exception clause. You can read the license files in detail on the web site or in the docs directory of the distribution, but in summary, you can use wxWidgets for commercial or free software with no royalty charge. You can link statically or dynamically to the wxWidgets library. If you make changes to the wxWidgets source code, you are obliged to make these freely available. You do not have to make your own source code or object files available. Please also consult the licenses for the optional subordinate libraries that are distributed with wxWidgets, such as the PNG and JPEG libraries.

The source code that accompanies this book is provided under the wxWindows License.

The wxWidgets Architecture

Table 1-1 shows the four conceptual layers: the wxWidgets public API, each major port, the platform API used by that port, and finally the underlying operating system.

Table 1-1. wxWidgets Ports

wxWidgets API								
wxWidgets Port								
wxMSW	wxGTK	wxX11	wxMotif	wxMac	wxCocoa	wxOS2	wxPalmOS	wxMGL
Platform API								
Win32	GTK+	Xlib	Motif/Lesstif	Carbon	Cocoa	PM	Palm OS Protein APIs	MGL
Operating System								
Windows/Windows CE	Unix/Linux			Mac OS 9/Mac OS X	Mac OS X	OS/2	Palm OS	Unix/DOS

The following are the main wxWidgets ports that exist at the time of writing.

wxMSW

This port compiles and runs on all 32-bit and 64-bit variants of the Microsoft Windows operating system, including Windows 95, Windows 98, Windows ME, Windows NT, Windows 2000, Windows XP and Windows 2003. It can also be compiled to use Winelib under Linux, and has a configuration that works on Windows CE (see "wxWinCE"). wxMSW can be configured to use the wxUniversal widgets instead of the regular Win32 ones.

wxGTK

wxWidgets for GTK+ can use versions 1.x or 2.x of the GTK+ widget set, on any Unix variant that supports X11 and GTK+ (for example, Linux, Solaris, HP-UX, IRIX, FreeBSD, OpenBSD, AIX, and others). It can also run on embedded platforms with sufficient resourcesfor example, under the GPE Palmtop Environment (see Figure 1-4). wxGTK is the recommended port for Unix-based systems.

Figure 1-4. The wxWidgets "Life!" demo under GPE on an iPAQ PDA

Summary

In this chapter, we've established what wxWidgets is, described a little of its history, summarized the available ports, and taken a brief look at how the library is organized internally.

In the next chapter, "Getting Started," we will look at some sample code and get a feeling for what it's like to write a wxWidgets application.

Chapter 2. Getting Started

In this chapter, we'll get a feel for the structure of a simple wxWidgets program, using a tiny sample. We'll look at where and how a wxWidgets application starts and ends, how to show the main window, and how to react to commands from the user. Following the wxWidgets philosophy of keeping things nice and simple, that's all we're going to cover in this chapter. You may also want to refer to Appendix A, "Installing wxWidgets."

A Small wxWidgets Sample

Figure 2-1 shows what our sample looks like under Windows.

Figure 2-1. Minimal sample under Windows

The minimal wxWidgets application shows a main window (a wxFrame) with a menu bar and status bar. The menus allow you to show an "about box" or quit the program. Not exactly a killer app, but it's enough to show some of the basic principles of wxWidgetsand to reassure you that you can start simple and work your way up to a complete application as your confidence and expertise grow.

The Application Class

Every wxWidgets application defines an application class deriving from wxApp. There is only one instance of it, and this instance represents the running application. At the very least, your class should define an OnInit function that will be called when wxWidgets is ready to start running your code (equivalent to main or WinMain when writing a C or Win32 application).

Here is the smallest application class declaration you can sensibly write:

```
// Declare the application class
class MyApp : public wxApp
{
public:
    // Called on application startup
    virtual bool OnInit();
};
```

The implementation of OnInit usually creates at least one window, interprets any command-line arguments, sets up data for the application, and performs any other initialization tasks required for the application. If the function returns TRue, wxWidgets starts the event loop that processes user input and runs event handlers as necessary. If the function returns false, wxWidgets will clean up its internal structures, and the application will terminate.

A simple implementation of OnInit might look like this:

```
bool MyApp::OnInit()
{
    // Create the main application window
    MyFrame *frame = new MyFrame(wxT("Minimal wxWidgets App"));

    // Show it
    frame->Show(true);

    // Start the event loop
    return true;
}
```

This creates an instance of our new class MyFrame (we'll define this class shortly), shows it, and returns TRue to start the event loop. Unlike child windows, top-level windows such as frames and dialogs need to be shown explicitly after creation.

The frame title is passed to the constructor wrapped in the wxT() macro. You'll see this used a lot in wxWidgets samples and in the library code itselfit converts string and character literals to the appropriate type to allow the application to be compiled in Unicode mode. This macro is also known by the alias _T(). There is no run-time performance penalty for using it. (You'll also see the underscore macro _() used to enclose strings, which tells wxWidgets to translate the string. See Chapter 16, "Writing International Applications," for more details.)

Where is the code that creates the instance of MyApp? wxWidgets does this internally, but you still need to tell wxWidgets what kind of object to create. So you need to add a macro in your implementation file:

```
// Give wxWidgets the means to create a MyApp object
IMPLEMENT_APP(MyApp)
```

Without specifying the class, wxWidgets would not know how to create a new application object. This macro also inserts code that checks that the application and library were compiled using the same build configuration, allowing wxWidgets to report accidental mismatches that might later cause a hard-to-debug run-time failure.

When wxWidgets creates a MyApp object, it assigns the result to the global variable wxTheApp. You can use this in

The Frame Class

Let's look at the frame class MyFrame. A frame is a top-level window that contains other windows, and usually has a title bar and menu bar. Here's our simple frame class declaration that we will put after the declaration of MyApp:

```
// Declare our main frame class
class MyFrame : public wxFrame
{
public:
    // Constructor
    MyFrame(const wxString& title);

    // Event handlers
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

private:
    // This class handles events
    DECLARE_EVENT_TABLE()
};
```

Our frame class has a constructor, two event handlers to link menu commands to C++ code, and a macro to tell wxWidgets that this class handles events.

The Event Handlers

As you may have noticed, the event handler functions in MyFrame are not virtual and should not be virtual. How, then, are they called? The answer lies in the event table, as follows.

```
// Event table for MyFrame
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(wxID_ABOUT, MyFrame::OnAbout)
    EVT_MENU(wxID_EXIT,  MyFrame::OnQuit)
END_EVENT_TABLE()
```

An event table, placed in a class's implementation file, tells wxWidgets how events coming from the user or from other sources are routed to member functions.

With the event table shown previously, mouse clicks on menu items with the identifiers wxID_EXIT and wxID_ABOUT are routed to the functions MyFrame::OnQuit and MyFrame::OnAbout, respectively. EVT_MENU is just one of many event table macros you can use to tell wxWidgets what kind of event should be routed to what function. The identifiers used here are predefined by wxWidgets, but you will often define your own identifiers, using enums, consts, or preprocessor defines.

This kind of event table is a static way of routing events, and cannot be changed at runtime. In the next chapter, we'll describe how to set up dynamic event handlers.

While we're dealing with event tables, let's see the two functions we're using as event handlers.

```
void MyFrame::OnAbout(wxCommandEvent& event)
{
    wxString msg;
    msg.Printf(wxT("Hello and welcome to %s"),
               wxVERSION_STRING);

    wxMessageBox(msg, wxT("About Minimal"),
                 wxOK | wxICON_INFORMATION, this);
}

void MyFrame::OnQuit(wxCommandEvent& event)
{
    // Destroy the frame
    Close();
}
```

MyFrame::OnAbout shows a message box when the user clicks on the About menu item. wxMessageBox takes a message, a caption, a combination of styles, and a parent window.

MyFrame::OnQuit is called when the user clicks on the Quit menu item, thanks to the event table. It calls Close to destroy the frame, triggering the shutdown of the application, because there are no other frames. In fact, Close doesn't directly destroy the frameit generates a wxEVT_CLOSE_WINDOW event, and the default handler for this event destroys the frame using wxWindow::Destroy.

There's another way the frame can be closed and the application shut downthe user can click on the close button on the frame, or select Close from the system (or window manager) menu. How does OnQuit get called in this case? Well, it doesn'tinstead, wxWidgets sends a wxEVT_CLOSE_WINDOW event to the frame via Close (as used in OnQuit). wxWidgets handles this event by default and destroys the window. Your application can override this behavior and provide its own event handlerfor example, if you want to ask the user for confirmation before closing. For more details, please see Chapter 4, "Window Basics."

This sample doesn't need it, but most applications should provide an OnExit function in its application class to clean up data structures before quitting. Note that this function is only called if OnInit returns true.

The Frame Constructor

Finally, we have the frame constructor, which implements the frame icon, a menu bar, and a status bar.

```
#include "mondrian.xpm"

MyFrame::MyFrame(const wxString& title)
       : wxFrame(NULL, wxID_ANY, title)
{
    // Set the frame icon
    SetIcon(wxIcon(mondrian_xpm));

    // Create a menu bar
    wxMenu *fileMenu = new wxMenu;

    // The "About" item should be in the help menu
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(wxID_ABOUT, wxT("&About...\tF1"),
                     wxT("Show about dialog"));

    fileMenu->Append(wxID_EXIT, wxT("E&xit\tAlt-X"),
                     wxT("Quit this program"));

    // Now append the freshly created menu to the menu bar...
    wxMenuBar *menuBar = new wxMenuBar();
    menuBar->Append(fileMenu, wxT("&File"));
    menuBar->Append(helpMenu, wxT("&Help"));

    // ... and attach this menu bar to the frame
    SetMenuBar(menuBar);

    // Create a status bar just for fun
    CreateStatusBar(2);
    SetStatusText(wxT("Welcome to wxWidgets!"));
}
```

This constructor calls the base constructor with the parent window (none, hence NULL), window identifier, and title. The identifier argument is wxID_ANY, which tells wxWidgets to generate an identifier itself. The base constructor creates the actual window associated with the C++ instanceanother way to achieve this is to call the default constructor of the base class, and then explicitly call wxFrame::Create from within the MyFrame constructor.

Small bitmaps and icons can be implemented using the XPM format on all platforms. XPM files have valid C++ syntax and so can be included as shown previously; the SetIcon line creates an icon on the stack using the C++ variable mondrian_xpm defined in mondrian.xpm, and associates it with the frame.

The menu bar is created next. Menu items are added using the identifier (such as the standard identifier wxID_ABOUT), the label to be displayed, and a help string to be shown on the status bar. Within each label, a mnemonic letter is marked by a preceding ampersand, and an accelerator is preceded by the tab character (\t). A mnemonic is the letter a user presses to highlight a particular item when the menu is displayed. An accelerator is a key combination (such as Alt+X) that can be used to perform that action without showing the menu at all.

The last thing that the constructor does is to create a status bar with two fields at the bottom of the frame and set the first field to the string "Welcome to wxWidgets!"

The Whole Program

It's worth putting together the bits so you can see what the whole program looks like. Normally, you'd have a separate header file and implementation file, but for such a simple program, we can put it all in the same file.

Listing 2-1. The Complete Example

```
// Name:        minimal.cpp
// Purpose:     Minimal wxWidgets sample
// Author:      Julian Smart

#include "wx/wx.h"

// Declare the application class
class MyApp : public wxApp
{
public:
    // Called on application startup
    virtual bool OnInit();
};

// Declare our main frame class
class MyFrame : public wxFrame
{
public:
    // Constructor
    MyFrame(const wxString& title);

    // Event handlers
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

private:
    // This class handles events
    DECLARE_EVENT_TABLE()
};

// Implements MyApp& GetApp()
DECLARE_APP(MyApp)

// Give wxWidgets the means to create a MyApp object
IMPLEMENT_APP(MyApp)

// Initialize the application
bool MyApp::OnInit()
{
    // Create the main application window
    MyFrame *frame = new MyFrame(wxT("Minimal wxWidgets App"));

    // Show it
    frame->Show(true);

    // Start the event loop
    return true;
}

// Event table for MyFrame
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(wxID_ABOUT, MyFrame::OnAbout)
    EVT_MENU(wxID_EXIT,  MyFrame::OnQuit)
END_EVENT_TABLE()

void MyFrame::OnAbout(wxCommandEvent& event)
{
    wxString msg;
```

Compiling and Running the Program

The sample can be found on the accompanying CD-ROM in examples/chap02, which you should copy to a folder on your hard drive for compiling. Because it's not possible to provide makefiles that work "out of the box" with every reader's software environment, we provide a DialogBlocks project file with configurations for most platforms and compilers. See Appendix C, "Creating Applications with DialogBlocks," for help with configuring DialogBlocks for your compiler. We also cover compiling wxWidgets applications in detail in Appendix B, "Building Your Own wxWidgets Applications."

Install wxWidgets and DialogBlocks from the accompanying CD-ROM. On Windows, you should install one or more of the compilers provided on the CD-ROM if you do not already own a suitable compiler. After setting your wxWidgets and compiler paths in the DialogBlocks Paths settings page, open the file examples/chap02/minimal.pjd. Select a suitable configuration for your compiler and platform such as MinGW Debug or VC++ Debug (Windows), GCC Debug GTK+ (Linux), or GCC Debug Mac (Mac OS X), and press the green Build and Run Project button. You may be prompted to build wxWidgets if you have not already built it for the selected configuration.

You can also find a similar sample in samples/minimal in your wxWidgets distribution. If you do not wish to use DialogBlocks, you can simply compile this sample instead. See Appendix A, "Installing wxWidgets," for instructions on how to build wxWidgets samples.

Program Flow

This is how the application starts running:

1. Depending on platform, the main, WinMain, or equivalent function runs (supplied by wxWidgets, not the application). wxWidgets initializes its internal data structures and creates an instance of MyApp.
2. wxWidgets calls MyApp::OnInit, which creates an instance of MyFrame.
3. The MyFrame constructor creates the window via the wxFrame constructor and adds an icon, menu bar, and status bar.
4. MyApp::OnInit shows the frame and returns true.
5. wxWidgets starts the event loop, waiting for events and dispatching them to the appropriate handlers.

As noted here, the application terminates when the frame is closed, when the user either selects the Quit menu item or closes the frame via standard buttons or menus (these will differ from one platform to the next).

Summary

This chapter gave you an idea of how a really simple wxWidgets application works. We've touched on the wxFrame class, event handling, application initialization, and creating a menu bar and status bar. However complicated your own code gets, the basic principles of starting the application will remain the same, as we've shown in this small example. In the next chapter, we'll take a closer look at events and how your application handles them.

Chapter 3. Event Handling

This chapter explains the principles behind event-driven programming in wxWidgets, including how events are generated, how an application handles them using event tables, and where window identifiers fit in. We'll also discuss plug-in and dynamic event handlers, and we'll describe how you can create your own event class, types, and macros.

Event-Driven Programming

When programmers encountered an Apple Macintosh for the first time, they were astonished at how different it was from the conventional computer experience of the period. Moving the pointer from one window to another, playing with scrollbars, menus, text controls and so on, it was hard to imagine how the underlying code sorted out this fabulous complexity. It seemed that many different things were going on in parallelin reality, a clever illusion. For many people, the Macintosh was their first introduction to the world of event-driven programming.

All GUI applications are event-driven. That is to say, the application sits in a loop waiting for events initiated by the user or from some other source (such as a window needing to be refreshed, or a socket connection), and then dispatches the event to an appropriate function that handles it. Although it may seem that windows are being updated simultaneously, most GUI applications are not multithreaded, so each task is being done in turnas becomes painfully obvious when something slows your computer to a crawl and you can see each window being repainted, one after the other.

Different frameworks have different ways of exposing event handling to the developerthe primary method in wxWidgets is the use of event tables, as explained in the next section.

Event Tables and Handlers

The wxWidgets event processing system is a more flexible mechanism than virtual functions, allowing us to avoid declaring all possible event handlers in a base class, which would be totally impractical as well as inefficient.

Every class that derives from wxEvtHandler, including frames, buttons, menus, and even documents, can contain an event table to tell wxWidgets how events are routed to handler functions. All window classes (derived from wxWindow), and the application class, are derived from wxEvtHandler.

To create a static event table (one that's created at compile time), you need to

1. Declare a new class that is derived directly or indirectly from wxEvtHandler.
2. Add a member function for each event that must be handled.
3. Declare the event table in the class with DECLARE_EVENT_TABLE.
4. Implement the event table in the source file with BEGIN_EVENT_TABLE... END_EVENT_TABLE.
5. Add event table entries to the table (such as EVT_BUTTON), mapping each event to the appropriate member function.

All event handler functions have the same formtheir return type is void, they are not virtual, and they take a single event object argument. (If you're familiar with MFC, this will come as a relief, because there is no standard signature for message handlers in MFC.) The type of this argument changes according to the type of event being handled; for example, simple control commands and menu commands share the wxCommandEvent class. A size event (caused by a window being resized either by the program or by the user) is represented by the wxSizeEvent class. Each event type has different accessors that you can use to learn about the cause of the event or the resulting UI change, such as a change in the value of a text control. In simple cases (such as a button press), you can often ignore the event object.

Expanding on the example from the previous chapter, let's add a handler for frame sizing, plus an OK button. A simple class declaration for an event-handling class looks like this:

```
// Declare our main frame class
class MyFrame : public wxFrame
{
public:
    // Constructor
    MyFrame(const wxString& title);
    // Event handlers
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
    void OnSize(wxSizeEvent& event);
    void OnButtonOK(wxCommandEvent& event);
private:
    // This class handles events
    DECLARE_EVENT_TABLE()
};
```

The code to add menu items will be similar to the code in the previous chapter, while code to add an OK button might look like this, in the frame constructor:

```
wxButton* button = new wxButton(this, wxID_OK, wxT("OK"),
```

Skipping Events

The wxWidgets event processing system implements something very close to virtual methods in normal C++, which means that it is possible to alter the behavior of a class by overriding its event handling functions. In many cases, this works even for changing the behavior of native controls. For example, it is possible to filter out selected keystroke events sent by the system to a native text control by overriding wxTextCtrl and defining a handler for key events using EVT_KEY_DOWN. This would indeed prevent any key events from being sent to the native control, which might not be what is desired. In this case, the event handler function has to call wxEvent::Skip to indicate that the search for the event handler should continue.

To summarize, instead of explicitly calling the base class version, as you would have done with C++ virtual functions, you should instead call Skip on the event object.

For example, to make the derived text control only accept "a" to "z" and "A" to "Z," we would use this code:

```
void MyTextCtrl::OnChar(wxKeyEvent& event)
{
    if ( wxIsalpha( event.KeyCode() ) )
    {
       // Keycode is within range, so do normal processing.
       event.Skip();
    }
    else
    {
       // Illegal keystroke. We don't call event.Skip() so the
       // event is not processed anywhere else.
       wxBell();
    }
}
```

Pluggable Event Handlers

You don't have to derive a new class from a window class in order to process events. Instead, you can derive a new class from wxEvtHandler, define the appropriate event table, and then call wxWindow::PushEventHandler to add this object to the window's stack of event handlers. Your new event handler will catch events first; if they are not processed, the next event handler in the stack will be searched, and so on. Use wxWindow::PopEventHandler to pop the topmost event handler off the stack, passing true if you want it to be deleted.

With this method, you can avoid a lot of class derivation and potentially use the same event handler object to handle events from instances of different classes.

Normally, the value returned from wxWindow::GetEventHandler is the window itself, but if you have used PushEventHandler, this will not be the case. If you ever have to call a window's event handler manually, use the GetEvent Handler function to retrieve the window's topmost event handler and use that to call the member function in order to ensure correct processing of the event handler stack.

One use of PushEventHandler is to temporarily or permanently change the behavior of the GUI. For example, you might want to invoke a dialog editor in your application. You can grab all the mouse input for an existing dialog box and its controls, processing events for dragging sizing handles and moving controls, before restoring the dialog's normal mouse behavior. This could be a useful technique for online tutorials, where you take a user through a series of steps and don't want them to diverge from the lesson. Here, you can examine the events coming from buttons and windows and, if acceptable, pass them through to the original event handler using wxEvent::Skip. Events not handled by your event handler will pass through to the window's event table.

Dynamic Event Handlers

We have discussed event handling mostly in terms of static event tables because this is the most common way to handle events. However, you can also specify the mapping between events and their handlers at runtime. You might use this method because you want to use different event handlers at different times in your program, or because you are using a different language (such as Python) that can't use static event tables, or simply because you prefer it. Event handlers allow greater granularityyou can turn individual event handlers on and off at runtime, whereas PushEventHandler and PopEventHandler deal with a whole event table. Plus, they allow sharing of event handler functions between different objects.

There are two functions associated with dynamic event tables: wxEvtHandler::Connect and wxEvtHandler::Disconnect. Often, you won't need to call wxEvtHandler::Disconnect because the disconnection will happen when the window object is destroyed.

Let's use our simple frame class with two event handlers as an example.

```
// Declare our main frame class
class MyFrame : public wxFrame
{
public:
    // Constructor
    MyFrame(const wxString& title);

    // Event handlers
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

private:
};
```

Notice that this time, we do not use the DECLARE_EVENT_TABLE macro. To specify event handling dynamically, we add a couple of lines to our application class's OnInit function:

```
frame->Connect( wxID_EXIT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnQuit) );

frame->Connect( wxID_ABOUT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnAbout) );
```

We pass to Connect the window identifier, the event identifier, and finally a pointer to the event handler function. Note that the event identifier (wxEVT_COMMAND_MENU_SELECTED) is different from the event table macro (EVT_MENU); the event table macro makes use of the event identifier internally. The wxCommandEventHandler macro around the function name is necessary to appease the compilercasts are done by static event table macros automatically. In general, if you have a handler that takes an event called wxXYZEvent, then in your Connect call, you will wrap the function name in the macro wxXYZEventHandler.

If we want to remove the mapping between event and handler function, we can use the wxEvtHandler::Disconnect function, as follows:

```
frame->Disconnect( wxID_EXIT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnQuit) );

frame->Disconnect( wxID_ABOUT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventFunction(MyFrame::OnAbout) );
```

Window Identifiers

Window identifiers are integers, and are used to uniquely determine window identity in the event system. In fact, identifiers do not need to be unique across your entire application, just unique within a particular context, such as a frame and its children. You may use the wxID_OK identifier, for example, on any number of dialogs as long as you don't have several within the same dialog.

If you pass wxID_ANY to a window constructor, an identifier will be generated for you automatically by wxWidgets. This is useful when you don't care about the exact identifier, either because you're not going to process the events from the control being created at all, or because you process the events from all controls in one place. In this case, you should specify wxID_ANY in the event table or wxEvtHandler::Connect call as well. The generated identifiers are always negative, so they will never conflict with the user-specified identifiers, which must always be positive.

wxWidgets supplies the standard identifiers listed in Table 3-1. Use the standard identifiers wherever possible: some systems can use the information to provide standard graphics (such as the OK and Cancel buttons on GTK+) or default behavior (such as responding to the Escape key by emulating a wxID_CANCEL event). On Mac OS X, wxID_ABOUT, wxID_PREFERENCES and wxID_EXIT menu items are interpreted specially and transferred to the application menu. Some wxWidgets components, such as wxTextCtrl, know how to handle menu items or buttons with identifiers such as wxID_COPY, wxID_PASTE, wxID_UNDO.

Table 3-1. Standard Window Identifiers

Identifier Name	Description
wxID_ANY	This may be passed to a window constructor as an identifier, and wxWidgets will generate an appropriate identifier
wxID_LOWEST	The lowest standard identifier value (4999)
wxID_HIGHEST	The highest standard identifier value (5999)
wxID_OPEN	File open
wxID_CLOSE	Window close
wxID_NEW	New window, file or document
wxID_SAVE	File save
wxID_SAVEAS	File save as (prompts for a save dialog)
wxID_REVERT	Revert (revert to file on disk)
wxID_EXIT	Exit application
wxID_UNDO	Undo the last operation

Defining Custom Events

If you want to define your own event class and macros, you need to follow these steps:

1. Derive your class from a suitable class, declaring dynamic type information and including a Clone function. You may or may not want to add data members and accessors. Derive from wxCommandEvent if you want the event to propagate up the window hierarchy, and from wxNotifyEvent if you also want handlers to be able to call Veto.

2. Define a typedef for the event handler function.

3. Define a table of event types for the individual events your event class supports. The event table is defined in your header with BEGIN_DECLARE_EVENT_TYPES()... END_DECLARE_EVENT_TYPES() and each type is declared with DECLARE_EVENT_TABLE(name, integer). Then in your implementation file, write DEFINE_EVENT_TYPE(name).

4. Define an event table macro for each event type.

Let's make this clearer with an example. Say we want to implement a new control class, wxFontSelectorCtrl, which displays a font preview; the user can click on the preview to pop up a font selector dialog to change the font. The application may want to intercept the font selection event, so we'll send a custom command event from within our low-level mouse handling code.

We will need to define a new event class wxFontSelectorCtrlEvent. An application will be able to route font change events to an event handler with the macro EVT_FONT_SELECTION_CHANGED(id, func), which uses the single event type wxEVT_COMMAND_FONT_SELECTION_CHANGED. Here's what we need in our new control header file, as well as the control declaration itself (not shown):
[View full width]

```
/*!
 * Font selector event class
 */
class wxFontSelectorCtrlEvent : public wxNotifyEvent
{
public:
    wxFontSelectorCtrlEvent(wxEventType commandType = wxEVT_NULL,
      int id = 0): wxNotifyEvent(commandType, id)
    {}

    wxFontSelectorCtrlEvent(const wxFontSelectorCtrlEvent& event): wxNotifyEvent(event)
    {}

    virtual wxEvent *Clone() const
                { return new wxFontSelectorCtrlEvent(*this); }

DECLARE_DYNAMIC_CLASS(wxFontSelectorCtrlEvent);
};

typedef void (wxEvtHandler::*wxFontSelectorCtrlEventFunction)
                                        (wxFontSelectorCtrlEvent&);

/*!
 * Font selector control events and macros for handling them
 */
BEGIN_DECLARE_EVENT_TYPES()
    DECLARE_EVENT_TYPE(wxEVT_COMMAND_FONT_SELECTION_CHANGED, 801)
END_DECLARE_EVENT_TYPES()

#define EVT_FONT_SELECTION_CHANGED(id, fn) DECLARE_EVENT_TABLE_ENTRY(
➥ wxEVT_COMMAND_FONT_SELECTION_CHANGED, id, -1, (wxObjectEventFunction)
```

Summary

In this chapter, we've discussed how events are propagated through inheritance and window hierarchies, introduced pluggable and dynamic event handlers, talked about window identifiers, and described how you can write your own custom event classes and macros. For more on the mechanics of event handling, please refer to Appendix H, "How wxWidgets Processes Events." Appendix I, "Event Classes and Macros," lists commonly used event classes and macros. You can also look at a number of the wxWidgets samples for examples of event usage, notably samples/event. Next, we'll discuss a range of important GUI components that you can start putting to use in your applications.

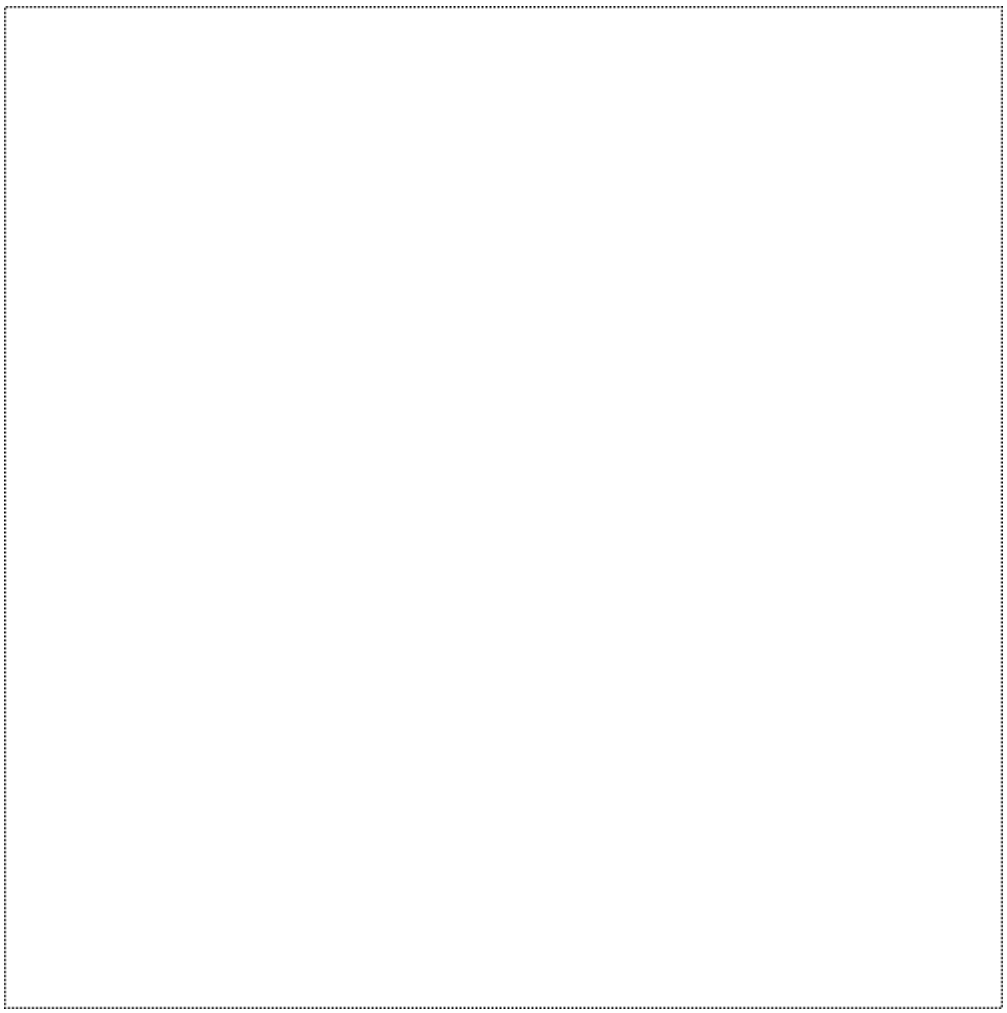
Chapter 4. Window Basics

In this chapter, we'll first look at the main elements of a window before describing the window classes that are most commonly used in applications. These elements will almost certainly be familiar from other programming you've done or from applications you've used. Despite the differences between systems, the available controls are surprisingly similar in functionality, with wxWidgets handling nearly all of the differences. Remaining differences are usually handled using optional platform-specific window styles.

Anatomy of a Window

Naturally, you know what a window is, but for a full understanding of how to use the wxWidgets API, it's good to have a grasp of the window model that wxWidgets uses. This differs in small respects from the window model used on each individual platform. Figure 4-1 shows the basic elements of a window.

Figure 4-1. The elements of a window



The Concept of a Window

A window is any rectangular area with a common set of properties: it can be resized, it can paint itself, it can be shown and hidden, and so on. It may contain other windows (such as a frame with menu bar, toolbar, and status bar), or no child windows (such as a static text control). Normally, a window that you see on the screen in a wxWidgets application has a corresponding object of the wxWindow class or derived class, but this is not always the case: for example, the drop-down list in a native wxComboBox is not usually modeled with a separate wxWindow.

Client and Non-Client Areas

When we refer to the size of a window, we normally include the outer dimensions, including decorations such as the border and title bar. When we refer to the size of the client area of a window, we mean the area inside the window that can be drawn upon or into which child windows may be placed. A frame's client area excludes any space taken by the menu bar, status bar, and toolbar.

Scrollbars

Most windows are capable of showing scrollbars, managed by the window rather than added explicitly by the

A Quick Guide to the Window Classes

The rest of this chapter provides you with enough detailed information about the most commonly used window classes for you to apply them to your own applications. However, if you are reading this book for the first time, you may want to skip ahead to Chapter 5 and browse the window descriptions later.

Here's a summary of the classes we cover, to help you navigate this chapter. For other window classes, see Chapter 12, "Advanced Window Classes," and Appendix E, "Third-Party Tools for wxWidgets."

Base Window Classes

These base classes implement functionality for derived concrete classes.

- wxWindow. The base class for all windows.
- wxControl. The base class for controls, such as wxButton.
- wxControlWithItems. The base class for multi-item controls.

Top-Level Windows

Top-level windows usually exist independently on the desktop.

- wxFrame. A resizable window containing other windows.
- wxMDIParentFrame. A frame that manages other frames.
- wxMDIChildFrame. A frame managed by a parent frame.
- wxDialog. A resizable window for presenting choices.
- wxPopupWindow. A transient window with minimal decoration.

Container Windows

Container windows manage child windows.

- wxPanel. A window for laying out controls.
- wxNotebook. A window for switching pages using tabs.
- wxScrolledWindow. A window that scrolls children and graphics.
-

Base Window Classes

It's worth mentioning base classes that you may or may not be able use directly but that implement a lot of functionality for derived classes. Use the API reference for these (and other) base classes as well as the reference for the derived classes to get a full understanding of what's available.

wxWindow

wxWindow is both an important base class and a concrete window class that you can instantiate. However, it's more likely that you will derive classes from it (or use pre-existing derived classes) than use it on its own.

As we've seen, a wxWindow may be created either in one step, using a non-default constructor, or two steps, using the default constructor followed by Create. For one-step construction, you use this constructor:

```
wxWindow(wxWindow* parent,
    wxWindowID id,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = 0,
    const wxString& name = wxT("panel"));
```

For example:

```
wxWindow* win  = new wxWindow(parent, wxID_ANY,
    wxPoint(100, 100), wxSize(200, 200));
```

wxWindow **Styles**

Each window class may add to the basic styles defined for wxWindow, listed in Table 4-1. Not all native controls support all border styles, and if no border is specified, a default style appropriate to that class will be used. For example, on Windows, most wxControl-derived classes use wxSUNKEN_BORDER by default, which will be interpreted as the border style for the current theme. An application may suppress the default border by using a style such as wxNO_BORDER.

Table 4-1. Basic Window Styles

wxSIMPLE_BORDER	Displays a thin border around the window.
wxDOUBLE_BORDER	Displays a double border.
wxSUNKEN_BORDER	Displays a sunken border, or control border consistent with the current theme.
wxRAISED_BORDER	Displays a raised border.
wxSTATIC_BORDER	Displays a border suitable for a static control. Windows only.
wxNO_BORDER	Displays no border. This overrides any attempt wxWidgets makes to add a suitable border.
wxtrANSPARENT_WINDOW	Specifies a transparent window (one that doesn't receive

Top-Level Windows

Top-level windows are placed directly on the desktop and are not contained within other windows. They can be moved around the screen, and resized if the application permits it. There are three basic kinds of top-level window: wxFrame and wxDialog, both derived from an abstract base class called wxTopLevelWindow, and wxPopupWindow, which has less functionality and is derived directly from wxWindow. A dialog can be either modal or modeless, whereas a frame is almost always modeless. Modal means that flow through the application effectively halts until the user dismisses the dialog. This is very handy for getting a response before continuing, but it's always good to see whether an alternative user interface can be used (for example, a font control on the toolbar rather than in a properties window) to keep the interaction more fluid.

Top-level windows normally have a title bar and can have decorations for closing, minimizing, or restoring the window. A frame often has a menu bar, toolbar, and status bar, but a dialog generally does not. On Mac OS X, a frame's menu bar is not shown at the top of the frame, but at the top of the screen.

Don't confuse this usage of "top-level window" with the window returned by wxApp::GetTopWindow, by which wxWidgets or your application can get hold of the "main window," most often the first frame or dialog you create.

If needed, you can access all current top-level windows using the global variable wxTopLevelWindows, which is a wxWindowList.

wxFrame

wxFrame is a popular choice for the main application window. Figure 4-2 shows the elements that compose a frame. A frame may optionally have a title bar (with decorations such as a button for closing the window), a wxMenuBar, a wxToolBar, and a wxStatusBar. The area that is left for child windows is called the client area, and it is up to the application to arrange the children's size and position if there is more than one child. You should use sizers for this purpose, described in Chapter 7, or perhaps a splitter window (covered later in this chapter) if you need a couple of child windows.

Figure 4-2. The elements of a frame

Because some platforms do not allow direct painting on a frame, and to support keyboard navigation between child

Container Windows

Container windows are designed to hold further visual elements, either child windows or graphics drawn on the window.

wxPanel

wxPanel is essentially a wxWindow with some dialog-like properties. This is usually the class to use when you want to arrange controls on a window that's not a wxDialog, such as a wxFrame. It's often used for pages of a wxNotebook. wxPanel normally takes the system dialog color.

As with wxDialog, you can use InitDialog to send a wxInitDialogEvent to the panel, transferring data to the panel via validators or other means. wxPanel also handles navigation keys such as the Tab key to provide automatic traversal between controls, if the wxTAB_TRAVERSAL style is provided.

wxPanel has the following constructor in addition to the default constructor:

```
wxPanel(wxWindow* parent, wxWindowID id,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = wxTAB_TRAVERSAL|wxNO_BORDER,
    const wxString& name = wxT("panel"));
```

For example:

```
wxPanel* panel = new wxPanel(frame, wxID_ANY,
    wxDefaultPosition, (500, 300));
```

wxPanel **Styles**

There are no specific styles for wxPanel, but see the styles for wxWindow.

wxPanel **Member Functions**

There are no distinct wxPanel functions; please refer to the wxWindow member functions, inherited by wxPanel.

wxNotebook

This class represents a control with several pages, switched by clicking on tabs along an edge of the control. A page is normally a wxPanel or a class derived from it, although you may use other window classes.

Notebook tabs may have images as well as, or instead of, text labels. The images are supplied via a wxImageList (see Chapter 10), and specified by position in the list.

To use a notebook, create a wxNotebook object and call AddPage or InsertPage, passing a window to be used as the page. Do not explicitly destroy the window for a page that is currently managed by wxNotebook; use DeletePage instead, or let the notebook destroy the pages when it is itself destroyed.

Here's an example of creating a notebook with three panels, and a text label and icon for each tab:

```
#include "wx/notebook.h"

#include "copy.xpm"
#include "cut.xpm"
#include "paste.xpm"
```

Non-Static Controls

Non-static controls, such as wxButton and wxListBox, respond to mouse and keyboard input. We'll describe the basic ones here; more advanced controls are described in Chapter 12. You can also download others (see Appendix E) or create your own.

wxButton

A wxButton is a control that looks like a physical push button with a text label, and it is one of the most common elements of a user interface. It may be placed on a dialog box or panel, or almost any other window. A command event is generated when the user clicks on the button.

Here's a simple example of creating a button:

```
#include "wx/button.h"
wxButton* button = new wxButton(panel, wxID_OK, wxT("OK"),
    wxPoint(10, 10), wxDefaultSize);
```

Figure 4-12 shows how a button with the default size looks on Windows XP.

Figure 4-12. A wxButton

wxWidgets obtains the default button size by calling the static function wxButton::GetDefaultSize, calculated appropriately for each platform, but you can let wxWidgets size the button to just fit the label by passing the style wxBU_EXACTFIT.

wxButton **Styles**

Table 4-17 lists the specific window styles for wxButton.

Table 4-17. wxButton Styles

wxBU_LEFT	Left-justifies the label. Windows and GTK+ only.
wxBU_TOP	Aligns the label to the top of the button. Windows and GTK+ only.
wxBU_RIGHT	Right-justifies the bitmap label. Windows and GTK+ only.
wxBU_BOTTOM	Aligns the label to the bottom of the button. Windows and GTK+ only.
wxBU_EXACTFIT	Creates the button as small as possible instead of making it the standard size.
wxNO_BORDER	Creates a flat button. Windows and GTK+ only.

Static Controls

Static controls do not take any input and are used to display information or to enhance the application's aesthetics.

wxGauge

This is a horizontal or vertical bar that shows a quantity (often time) from zero to the specified range. No command events are generated for the gauge. Here's a simple example of creating a gauge:

```
#include "wx/gauge.h"

wxGauge* gauge = new wxGauge(panel, ID_GAUGE,
  200, wxDefaultPosition, wxDefaultSize, wxGA_HORIZONTAL);
gauge->SetValue(50);
```

Under Windows, this is displayed as shown in Figure 4-28.

Figure 4-28. A wxGauge

wxGauge **Styles**

Table 4-42 lists the specific window styles for wxGauge.

Table 4-42. wxGauge Styles

wxGA_HORIZONTAL	Creates a horizontal gauge.
wxGA_VERTICAL	Creates a vertical gauge.
wxGA_SMOOTH	Creates a smooth progress bar with no spaces between steps. This is only supported on Windows.

wxGauge **Events**

Because it only displays information, wxGauge does not generate events.

wxGauge **Member Functions**

These are the major wxGauge functions.

GeTRange and SetRange are accessors for the gauge range (the maximum integer value).

GetValue and SetValue get and set the integer value of the gauge.

IsVertical returns TRue if the gauge is vertical, and false if horizontal.

wxStaticText

Menus

In this section, we'll describe programming with wxMenu, a simple way to present commands without taking up a lot of display space. In the next section, we'll look at how menus are used in menu bars.

wxMenu

A menu is a list of commands that pops up either from a menu bar or on an arbitrary window, often as a "context menu" invoked by clicking the right mouse button (or equivalent). A menu item can be a normal command, or it can have a check or radio button next to the label. A menu item in a disabled state won't respond to commands. A special kind of menu item can display a visual indication of a further pull-right menu, and this can be nested to an arbitrary level. Another kind of menu item is the separator, which simply displays a line or space to indicate separation between two groups of items.

Figure 4-33 shows a typical menu with normal, check, and radio items and a submenu.

Figure 4-33. A typical menu

The example shows the use of both mnemonics and shortcuts. A mnemonic is a highlighted key in a label (such as the "N" in "New") that can be pressed when the menu is shown to execute that command. Specify a mnemonic by preceding the letter with an ampersand ("&"). A shortcut (or accelerator) is a key combination that can be used when the menu is not shown, and it is indicated in a menu item by a key combination following a tab character. For example, the New menu item in the example was created with this code:

```
menu->Append(wxID_NEW, wxT("&New...\tCtrl+N"));
```

For more on creating accelerators via menu items or by programming with wxAcceleratorTable, please see Chapter 6 .

Check and radio items automatically update their state; that is, when the user toggles a check item, it will be shown in the reverse state when the menu is next popped up. Similarly, consecutive radio items form a group and when one item is checked, the other items in the group are unchecked. You can also set these states yourself, for example from a user interface update event handler (see Chapter 9).

You can create a menu and show it at a particular point in a window using wxWindow::PopupMenu, for example:

```
void wxWindow::OnRightClick(wxMouseEvent& event)
{
    if (!m_menu)
    {
        m_menu = new wxMenu;
        m_menu->Append(wxID_OPEN, wxT("&Open"));
        m_menu->AppendSeparator();
```

Control Bars

A control bar provides a convenient way to contain and arrange multiple controls. There are currently three kinds of control bars: wxMenuBar, wxToolBar, and wxStatusBar. wxMenuBar can only belong to a wxFrame. wxToolBar and wxStatusBar are most commonly used with wxFrame, but they can also be children of other windows.

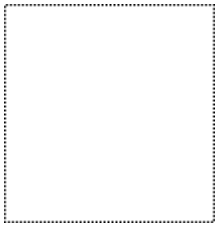
wxMenuBar

A menu bar contains a series of menus accessible from the top of a frame under the title bar. You can replace a frame's current menu bar by calling SetMenuBar. To create a menu bar, use the default constructor and append wxMenu objects. For example:

```
wxMenuBar* menuBar = new wxMenuBar;
wxMenu* fileMenu = new wxMenu;
fileMenu->Append(wxID_OPEN, wxT("&Open..."), wxT("Opens a file"));
fileMenu->AppendSeparator();
fileMenu->Append(wxID_EXIT, wxT("E&xit"), wxT("Quits the program"));
menuBar->Append(fileMenu);
frame->SetMenuBar(menuBar, wxT("&File"));
```

This code creates a one-menu menu bar, as shown in Figure 4-34.

Figure 4-34. A wxMenuBar



You can append submenus to a wxMenu, and you can create check and radio menu items (refer to the "Menus" section earlier in this chapter). As in the previous example, an ampersand in a label indicates that the following character should be used as a mnemonic (pressing that key when the menu is shown executes the associated command).

If you provide a help string, it will be shown in the frame's status bar (if any) by virtue of the default EVT_MENU_HIGHLIGHT handler.

wxMenuBar **Styles**

wxMenuBar takes the wxMB_DOCKABLE style, used under GTK+ to allow the menu bar to be detached from the frame.

wxMenuBar **Events**

Menu bars use the events already covered in the description of wxMenu.

wxMenuBar **Member Functions**

These are the major wxMenuBar functions.

Append adds a menu to the end of the menu bar, which will then own the menu and will destroy it when the menu bar is destroyed (usually by the owning frame). Pass the menu and a label. Insert inserts a menu at the given position.

Summary

This chapter has given you enough information about the capabilities of essential window and control classes to know how to start building useful applications. For more details on these and other window classes, please refer to the reference manual. For further window classes, and how to create your own controls, see Chapter 12. You'll also find it useful to look at the samples in your wxWidgets distribution, such as samples/widgets, samples/toolbar, samples/text, and samples/listbox.

Next, we'll look at how how your application can draw on a variety of surfaces, including windows, bitmaps, and the printed page.

Chapter 5. Drawing and Printing

This chapter introduces the idea of the device context, generalizing the concept of a drawing surface such as a window or a printed page. We will discuss the available device context classes and the set of "drawing tools" that wxWidgets provides for handling fonts, color, line drawing, and filling. Next we describe a device context's drawing functions and how to use the wxWidgets printing framework. We end the chapter by briefly discussing wxGLCanvas, which provides a way for you to draw 3D graphics on your windows using OpenGL.

Understanding Device Contexts

All drawing in wxWidgets is done on a device context, using an instance of a class derived from wxDC. There is no such thing as drawing directly to a window; instead, you create a device context for the window and then draw on the device context. There are also device context classes that work with bitmaps and printers, or you can design your own. A happy consequence of this abstraction is that you can define drawing code that will work on a number of different device contexts: just parameterize it with wxDC, and if necessary, take into account the device's resolution by scaling appropriately. Let's describe the major properties of a device context.

A device context has a coordinate system with its origin at the top-left of the surface. This position can be changed with SetDeviceOrigin so that graphics subsequently drawn on the device context are shiftedthis is used when painting with wxScrolledWindow. You can also use SetAxisOrientation if you prefer, say, the y-axis to go from bottom to top.

There is a distinction between logical units and device units. Device units are the units native to the particular devicefor a screen, a device unit is a pixel. For a printer, the device unit is defined by the resolution of the printer, which can be queried using GetSize (for a page size in device units) or GetSizeMM (for a page size in millimeters).

The mapping mode of the device context defines the unit of measurement used to convert logical units to device units. Note that some device contexts, in particular wxPostScriptDC, do not support mapping modes other than wxMM_TEXT. Table 5-1 lists the available mapping modes.

Table 5-1. Mapping Modes

wxMM_TWIPS	Each logical unit is 1/20 of a point, or 1/1440 of an inch.
wxMM_POINTS	Each logical unit is a point, or 1/72 of an inch.
wxMM_MEtrIC	Each logical unit is 1 millimeter.
wxMM_LOMETRIC	Each logical unit is 1/10 of a millimeter.
wxMM_TEXT	Each logical unit is 1 pixel. This is the default mode.

You can impose a further scale on your logical units by calling SetUser Scale, which multiplies with the scale implied by the mapping mode. For example, in wxMM_TEXT mode, a user scale value of (1.0, 1.0) makes logical and device units identical. By default, the mapping mode is wxMM_TEXT, and the scale is (1.0, 1.0).

A device context has a clipping region, which can be set with SetClipping Region and cleared with DestroyClippingRegion. Graphics will not be shown outside the clipping region. One use of this is to draw a string so that it appears only inside a particular rectangle, even though the string might extend beyond the rectangle boundary. You can set the clipping region to be the same size and location as the rectangle, draw the text, and then destroy the clipping region, and the text will be truncated to fit inside the rectangle.

Just as with real artistry, in order to draw, you must first select some tools. Any operation that involves drawing an outline uses the currently selected pen, and filled areas use the current brush. The current font, together with the foreground and background text color, determines how text will appear. We will discuss these tools in detail later, but first we'll look at the types of device context that are available to us.

Available Device Contexts

These are the device context classes you can use:

-

Drawing Tools

Drawing code in wxWidgets operates like a very fast artist, rapidly selecting colors and drawing tools, drawing a little part of the scene, then selecting different tools, drawing another part of the scene, and so on. Here we describe the wxColour, wxPen, wxBrush, wxFont and wxPalette classes. You will also find it useful to refer to the descriptions of other classes relevant to drawingwxRect, wxRegion, wxPoint, and wxSize, which are described in Chapter 13, "Data Structure Classes."

Note that these classes use "reference-counting," efficiently copying only internal pointers rather than chunks of memory. In most circumstances, you can create color, pen, brush, and font objects on the stack as they are needed without worrying about speed implications. If your application does have performance problems, you can take steps to improve efficiency, such as storing some objects as data members.

wxColour

You use wxColour to define various aspects of color when drawing. (Because wxWidgets started life in Edinburgh, the API uses British spelling. However, to cater for the spelling sensibilities of the New World, wxWidgets defines wxColor as an alias for wxColour.)

You can specify the text foreground and background color for a device context using a device context's SetTextForeground and SetTextBackground functions, and you also use wxColour to create pens and brushes.

A wxColour object can be constructed in a number of different ways. You can pass red, green, and blue values (each 0 to 255), or a standard color string such as WHITE or CYAN, or you can create it from another wxColour object. Alternatively, you can use the stock color objects, which are pointers: wxBLACK, wxWHITE, wxRED, wxBLUE, wxGREEN, wxCYAN, and wxLIGHT_GREY. The stock object wxNullColour is an uninitialized color for which the Ok function always returns false.

Using the wxSystemSettings class, you can retrieve some standard, system-wide colors, such as the standard 3D surface color, the default window background color, menu text color, and so on. Please refer to the documentation for wxSystemSettings::GetColour for the identifiers you can pass.

Here are some different ways to create a wxColour object:

```
wxColour color(0, 255, 0); // green
wxColour color(wxT("RED")); // red

// The color used for 3D faces and panels
wxColour color(wxSystemSettings::GetColour(wxSYS_COLOUR_3DFACE));
```

You can also use the wxTheColourDatabase pointer to add a new color, find a wxColour object for a given name, or find the name corresponding to the given color:

```
wxTheColourDatabase->Add(wxT("PINKISH"), wxColour(234, 184, 184));
wxString name = wxTheColourDatabase->FindName(
                                        wxColour(234, 184, 184));
wxString color = wxTheColourDatabase->Find(name);
```

These are the available standard colors: aquamarine, black, blue, blue violet, brown, cadet blue, coral, cornflower blue, cyan, dark gray, dark green, dark olive green, dark orchid, dark slate blue, dark slate gray dark turquoise, dim gray, firebrick, forest green, gold, goldenrod, gray, green, green yellow, indian red, khaki, light blue, light gray, light steel blue, lime green, magenta, maroon, medium aquamarine, medium blue, medium forest green, medium goldenrod, medium orchid, medium sea green, medium slate blue, medium spring green, medium turquoise, medium violet red, midnight blue, navy, orange, orange red, orchid, pale green, pink, plum, purple, red, salmon, sea green, sienna, sky blue, slate blue, spring green, steel blue, tan, thistle, turquoise, violet, violet red, wheat, white, yellow, and yellow green.

Device Context Drawing Functions

In this section, we'll take a closer look at how we draw on device contexts. The major functions are summarized in Table 5-5. We cover most of them in the following sections, and you can also find more details in the reference manual.

Table 5-5. Device Context Functions

Blit	Copies from one device context to another. You can specify how much of the original to draw, where drawing should start, the logical function to use, and whether to use a mask if the source is a memory device context.
Clear	Fills the device context with the current background brush.

`SetClippingRegion`
`DestroyClippingRegion`
`GetClippingBox`
	Sets and destroys the clipping region, which restricts drawing to a specified area. The clipping region can be specified as a rectangle or a wxRegion. Use GetClippingBox to get the rectangle surrounding the current clipping region.

`DrawArc` `DrawEllipticArc`	Draws an arc or elliptic arc using the current pen and brush.
DrawBitmap	Draws a wxBitmap or wxIcon at the specified location.
DrawIcon	The bitmap may have a mask to specify transparency.
DrawCircle	Draws a circle using the current pen and brush.
DrawEllipse	Draws an ellipse using the current pen and brush.
`DrawLine` `DrawLines`	Draws a line or number of lines using the current pen. The last point of the line is not drawn.
DrawPoint	Draws a point using the current pen.
`DrawPolygon` `DrawPolyPolygon`	DrawPolygon draws a filled polygon using an array of points or list of pointers to points, adding an optional offset coordinate. wxWidgets automatically closes the first and last points. DrawPolyPolygon draws one or more polygons at once, which can be a more efficient operation on some platforms.
`DrawRectangle` `DrawRoundedRectangle`	Draws a rectangle or rounded rectangle using the current pen and brush.

Using the Printing Framework

As we've seen, wxPrinterDC can be created and used directly. However, a more flexible method is to use the wxWidgets printing framework to "drive" printing. The main task for the developer is to derive a new class from wxPrintout, overriding functions that specify how to print a page (OnPrintPage), how many pages there are (GetPageInfo), document setup (OnPreparePrinting), and so on. The wxWidgets printing framework will show the print dialog, create the printer device context, and call appropriate wxPrintout functions when appropriate. The same printout class can be used for both printing and preview.

To start printing, a wxPrintout object is passed to a wxPrinter object, and Print is called to kick off the printing process, showing a print dialog before printing the pages specified by the layout object and the user. For example:
[View full width]

```
// A global object storing print settings
wxPrintDialogData g_printDialogData;

// Handler for Print menu item
void MyFrame::OnPrint(wxCommandEvent& event)
{

    wxPrinter printer(& g_printDialogData);
    MyPrintout printout(wxT("My printout"));

    if (!printer.Print(this, &printout, true))
    {
        if (wxPrinter::GetLastError() == wxPRINTER_ERROR)
            wxMessageBox(wxT("There was a problem printing.\nPerhaps your current
printer
➥ is not set correctly?"), wxT("Printing"), wxOK);
        else
            wxMessageBox(wxT("You cancelled printing"),
                         wxT("Printing"), wxOK);
    }
    else
    {
        (*g_printDialogData) = printer.GetPrintDialogData();
    }
}
```

Because the Print function returns only after all pages have been rendered and sent to the printer, the printout object can be created on the stack.

The wxPrintDialogData class stores data related to the print dialog, such as the pages the user selected for printing and the number of copies to be printed. It's a good idea to keep a global wxPrintDialogData object in your application to store the last settings selected by the user. You can pass a pointer to this data to wxPrinter to be used in the print dialog, and then if printing is successful, copy the settings back from wxPrinter to your global object, as in the previous example. (In a real application, g_printDialogData would probably be a data member of your application class.) See Chapter 8, "Using Standard Dialogs," for more about print and page dialogs and how to use them.

To preview the document, create a wxPrintPreview object, passing two printout objects to it: one for the preview and one to use for printing if the user requests it. You can also pass a wxPrintDialogData object so that the preview picks up settings that the user chose earlier. Then pass the preview object to wxPreviewFrame, call the frame's Initialize function, and show the frame. For example:
[View full width]

```
// Handler for Preview menu item
void MyFrame::OnPreview(wxCommandEvent& event)
{
    wxPrintPreview *preview = new wxPrintPreview(
                              new MyPrintout, new MyPrintout,
```

3D Graphics with wxGLCanvas

It's worth mentioning that wxWidgets comes with the capability of drawing 3D graphics, thanks to OpenGL and wxGLCanvas. You can use it with the OpenGL clone Mesa if your platform doesn't support OpenGL.

To enable wxGLCanvas support under Windows, edit include/wx/msw/ setup.h, set wxUSE_GLCANVAS to 1, and compile with USE_OPENGL=1 on the command line. You may also need to add opengl32.lib to the list of libraries your program is linked with. On Unix and Mac OS X, pass --with-opengl to the configure script to compile using OpenGL or Mesa.

If you're already an OpenGL programmer, using wxGLCanvas is very simple. You create a wxGLCanvas object within a frame or other container window, call wxGLCanvas::SetCurrent to direct regular OpenGL commands to the window, issue normal OpenGL commands, and then call wxGLCanvas::SwapBuffers to show the OpenGL buffer on the window.

The following paint handler shows the principles of rendering 3D graphics and draws a cube. The full sample can be compiled and run from samples/opengl/cube in your wxWidgets distribution.

```
void TestGLCanvas::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);

    SetCurrent();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-0.5f, 0.5f, -0.5f, 0.5f, 1.0f, 3.0f);
    glMatrixMode(GL_MODELVIEW);

    /* clear color and depth buffers */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* draw six faces of a cube */
    glBegin(GL_QUADS);
    glNormal3f( 0.0f, 0.0f, 1.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f,-0.5f, 0.5f); glVertex3f( 0.5f,-0.5f, 0.5f);

    glNormal3f( 0.0f, 0.0f,-1.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f(-0.5f, 0.5f,-0.5f);
    glVertex3f( 0.5f, 0.5f,-0.5f); glVertex3f( 0.5f,-0.5f,-0.5f);

    glNormal3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f( 0.5f, 0.5f,-0.5f);
    glVertex3f(-0.5f, 0.5f,-0.5f); glVertex3f(-0.5f, 0.5f, 0.5f);

    glNormal3f( 0.0f,-1.0f, 0.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f( 0.5f,-0.5f,-0.5f);
    glVertex3f( 0.5f,-0.5f, 0.5f); glVertex3f(-0.5f,-0.5f, 0.5f);

    glNormal3f( 1.0f, 0.0f, 0.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f( 0.5f,-0.5f, 0.5f);
    glVertex3f( 0.5f,-0.5f,-0.5f); glVertex3f( 0.5f, 0.5f,-0.5f);

    glNormal3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f(-0.5f,-0.5f, 0.5f);
    glVertex3f(-0.5f, 0.5f, 0.5f); glVertex3f(-0.5f, 0.5f,-0.5f);
    glEnd();

    glFlush();
    SwapBuffers();
}
```

Summary

In this chapter, you have learned how to draw on device contexts and use the wxWidgets printing framework, and you received a quick introduction to wxGLCanvas. You can look at the following source code in your wxWidgets distribution for examples of drawing and printing code:

- samples/drawing
- samples/font
- samples/erase
- samples/image
- samples/scroll
- samples/printing
- src/html/htmprint.cpp
- demos/bombs
- demos/fractal
- demos/life

For advanced 2D drawing applications, you might want to consider the wxArt2D library, which offers loading and saving of graphical objects using SVG files (Scalable Vector Graphics), flicker-free updating, gradients, vector paths, and more. See Appendix E, "Third-Party Tools for wxWidgets," for where to get wxArt2D.

Next, we'll look at how your application can respond to mouse, keyboard, and joystick input.

Chapter 6. Handling Input

All GUI applications must respond to input in some way. This chapter shows how you can respond to user input from the mouse, keyboard, and joystick.

Mouse Input

Broadly speaking, there are two categories of mouse input. Basic mouse events are sent using wxMouseEvent and are passed uninterpreted to your handler function. Commands associated with controls, on the other hand, are often the result of interpreting a mouse (or other) event as a particular command.

For example, when you add EVT_BUTTON to an event table, you are intercepting a wxCommandEvent that was generated by the wxButton. Internally, the button is intercepting EVT_LEFT_DOWN and generating the command event as a result. (Of course, on most platforms, wxButton is implemented natively and doesn't use low-level wxWidgets event handling, but it's true of custom classes.)

Because we've already seen examples of handling command events, we will concentrate on basic mouse events.

You can intercept button up, button down, and double-click events for left, middle, and right mouse buttons. You can intercept motion events, whether the mouse is moving with or without buttons pressed. You can intercept events telling you that the mouse is entering or leaving the window. Finally, you can intercept scroll wheel events if the hardware provides a scroll wheel.

When you receive a mouse event, you can also check the state of the mouse buttons, and the pressed state of the modifier keys (Shift, Alt, Control, and Meta). You can also retrieve the current mouse position relative to the top-left corner of the window's client area.

Table 6-1 lists the event table macros you can use. wxMouseEvent does not propagate to parents of the originating window, so to handle these events, you must derive from a window class or derive from wxEvtHandler and plug the object into the window with SetEventHandler or PushEventHandler. Alternatively, you can use dynamic event handling with Connect.

Table 6-1. Mouse Event Table Macros

EVT_LEFT_DOWN(func	Handles a wxEVT_LEFT_DOWN event, generated when the left mouse button changes to the "down" state.
EVT_LEFT_UP(func)	Handles a wxEVT_LEFT_UP event, generated when the left mouse button changes to the "up" state.
EVT_LEFT_DCLICK(func)	Handles a wxEVT_LEFT_DCLICK event, generated when the left mouse button is double-clicked.
EVT_MIDDLE_DOWN(func)	Handles a wxEVT_MIDDLE_DOWN event, generated when the middle mouse button changes to the "down" state.
EVT_MIDDLE_UP(func)	Handles a wxEVT_MIDDLE_UP event, generated when the middle mouse button changes to the "up" state.
EVT_MIDDLE_DCLICK(func)	Handles a wxEVT_MIDDLE_DCLICK event, generated when the middle mouse button is double-clicked.
EVT_RIGHT_DOWN(func)	Handles a wxEVT_RIGHT_DOWN event, generated when the right mouse button changes to the "down" state.

Handling Keyboard Events

Keyboard events are represented by the class wxKeyEvent. There are three different kinds of keyboard events in wxWidgets: key down, key up, and character. Key down and up events are untranslated events, whereas character events are translated, which we'll explain shortly. If the key is held down, you will typically get many down events but only one up event, so don't assume that one up event corresponds to each down event.

To receive key events, your window needs to have the keyboard focus, which you can achieve by calling wxWindow::SetFocusfor example, when a mouse button is pressed.

Table 6-2 lists the three keyboard event table macros.

Table 6-2. Keyboard Event Table Macros

EVT_KEY_DOWN(func)	Handles a wxEVT_KEY_DOWN event (untranslated key press).
EVT_KEY_UP(func)	Handles a wxEVT_KEY_UP event (untranslated key release).
EVT_CHAR(func)	Handles a wxEVT_CHAR event (translated key press).

These are the main wxKeyEvent functions that you can use within your key event handler when handling keyboard events.

To get the keycode, call GetKeyCode (in Unicode builds, you can also call GetUnicodeKeyCode). All valid key codes are listed in Table 6-3.

Table 6-3. Key Code Identifiers

WXK_BACK	WXK_RIGHT
WXK_TAB	WXK_DOWN
WXK_RETURN	WXK_SELECT
WXK_ESCAPE	WXK_PRINT
WXK_SPACE	WXK_EXECUTE
WXK_DELETE	WXK_SNAPSHOT
	WXK_INSERT
WXK_START	WXK_HELP
WXK_LBUTTON	

Handling Joystick Events

The wxJoystick class gives your application control over one or two joysticks on Windows or Linux. Typically, you'll create a wxJoystick object passing wxJOYSTICK1 or wxJOYSTICK2 and keep the object on the heap while it's needed. When you need input, call SetCapture passing a window pointer for receiving the joystick events, and then call ReleaseCapture when you no longer need the events. You might set the capture for the lifetime of the application instance (that is, calling SetCapture on initialization and ReleaseCapture on exit).

Before describing the events and functions in more detail, let's take a look at samples/joystick from the wxWidgets distribution. The user can control the joystick to draw a sequence of lines on a canvas by clicking on one of the joystick's buttons. Pressing the button also plays a sound.

The following is a snippet of the initialization code. First, the application checks whether a joystick is installed by creating a temporary joystick object, terminating if a joystick isn't found. The buttonpress.wav sound file is loaded into the wxSound object stored in the application object, and the minimum and maximum joystick positions are stored to permit scaling input to the size of the drawing window.

```
#include "wx/wx.h"
#include "wx/sound.h"
#include "wx/joystick.h"

bool MyApp::OnInit()
{
    wxJoystick stick(wxJOYSTICK1);
    if (!stick.IsOk())
    {
        wxMessageBox(wxT("No joystick detected!"));
        return false;
    }

    m_fire.Create(wxT("buttonpress.wav"));

    m_minX = stick.GetXMin();
    m_minY = stick.GetYMin();
    m_maxX = stick.GetXMax();
    m_maxY = stick.GetYMax();

    // Create the main frame window
    ...

    return true;
}
```

MyCanvas is a window that stores the joystick object and also receives the joystick events. Here's the implementation of MyCanvas.

```
BEGIN_EVENT_TABLE(MyCanvas, wxScrolledWindow)
    EVT_JOYSTICK_EVENTS(MyCanvas::OnJoystickEvent)
END_EVENT_TABLE()

MyCanvas::MyCanvas(wxWindow *parent, const wxPoint& pos,
    const wxSize& size):
    wxScrolledWindow(parent, wxID_ANY, pos, size, wxSUNKEN_BORDER)
{
    m_stick = new wxJoystick(wxJOYSTICK1);
    m_stick->SetCapture(this, 10);
}

MyCanvas::~MyCanvas()
{
    m_stick->ReleaseCapture();
    delete m_stick;
```

Summary

In this chapter, you have learned about mouse, keyboard, and joystick input, and you can now add sophisticated interaction to your applications. For more insight, see wxWidgets samples such as samples/keyboard, samples/joytest, and samples/dragimag, and also the wxThumbnailCtrl class in examples/chap12 on the CD-ROM.

The next chapter describes how you can achieve window layouts that are resizable, portable, translation-friendly, and above all attractive by using our flexible friend, the sizer.

Chapter 7. Window Layout Using Sizers

As graphic designers will testify, people are very sensitive to the way that visual objects are arranged. A GUI framework must allow the creation of a visually appealing layout, but unlike with print layout, an application's windows must often dynamically adapt to changes in size, font preferences, and even language. For platform-independent programming, the layout must also take into account the different sizes of individual controls from one platform to the next. All this means that a näive approach using absolute positions and sizes for controls simply won't work. This chapter describes wxWidgets' system of sizers, which gives you all the flexibility you need for even the most complex layouts. If it seems a bit daunting at first, remember that there are tools that will help you create sizer-based layoutssuch as DialogBlocks, included on the accompanying CD-ROMand you will rarely need to create entire layouts by hand.

Layout Basics

Before taking the plunge into the world of sizers, let's review where you might need to program layout behavior and what options you have.

A simple case is where you have a frame with a single window inside the client area. This is so simple that wxWidgets does the layout itself, fitting the child window to the size of the frame client area. For each frame, wxWidgets also manages the menu bar, one toolbar, and one status bar if they have been associated with the frame. If you need two toolbars, then you have to manage at least one of them yourself. If you have more than one child window in the client area, then wxWidgets expects you to manage them explicitly. You can do this with an OnSize event handler and calculate the position and size for each window and then set them. Or, you can use sizers. Similarly, if you create a custom control that consists of several child windows, you need to arrange for the child windows to resize appropriately when the overall control is resized.

Most applications have custom dialogs, sometimes dozens of them. The dialogs may be resizable, in which case the layout should look sensible even when the dialog is much larger than the initial size. The language may be changed, making some elements much larger or smaller than in the default language. If you had to program a hundred resize-friendly dialogs by hand, even with sizers, it would be almost impossibly daunting, so it's fortunate that editors are available to make this task simpleeven a pleasure.

If (and when!) you choose to use sizers, you need to decide how you will create and deploy them. You or your dialog editor can create code in C++ or another language, or you can use XRC files, which are a declarative XML specification of the sizer layout. XRC files can be loaded dynamically or embedded in the executable by compiling them into C++ files with the utility wxrc. Most dialog editors can generate both code and XRC files. Your choice of code or XRC may be a matter of taste; perhaps you prefer to separate the layout from the class, or maybe you prefer the additional flexibility of tweaking the C++ code and the immediacy of having it in the same file as the class.

The next section describes the principles behind sizers, and the following sections describe how to program with individual sizer classes.

Sizers

The layout algorithm used by sizers in wxWidgets is closely related to layout systems in other GUI toolkits, such as Java's AWT, the GTK+ toolkit, or the Qt toolkit. It is based upon the idea of individual windows reporting their minimal required size and their ability to be stretched if the size of the parent window has changed. This will most often mean that the programmer does not set the initial size of a dialog; instead, the dialog will be assigned a sizer, which will be queried about the recommended size. This sizer in turn will query its children (which can be windows, empty space, or other sizers) and further descendants. Note that wxSizer does not derive from wxWindow and thus does not interfere with tab ordering and requires very few resources compared to a real window. Sizers form a containment hierarchy parallel to the actual window hierarchy: the sizer hierarchy for a complex dialog may be many deep, but the controls themselves will probably all be siblings with the dialog as their parent.

This sizer hierarchy is depicted graphically in wxWidgets dialog editors. Figure 7-1 shows Anthemion Software's DialogBlocks editing the Personal Record dialog that we will use as an example in Chapter 9, "Creating Custom Dialogs." A red border in the editor pane surrounds the currently selected element, and its immediate parent is shown with a blue border. The tree you see on the left represents the sizer view of the hierarchy, but all controls are still parented directly on the dialog as far as the window hierarchy is concerned.

Figure 7-1. Viewing a sizer hierarchy in a dialog editor

[View full size image]

To get a mental picture of how nested sizers work, consider Figure 7-2, a schematic view of the dialog being edited in Figure 7-1. The shaded areas represent actual windows, while the white boxes represent sizers. There are two vertical sizers inside the dialog (to give extra space around the edge of the dialog), and two horizontal sizers within the inner vertical sizer. A spacer is employed inside a horizontal sizer to keep one control away from another group of controls. As you can see, creating a sizer-based layout is really like sorting through a collection of different-sized cardboard boxes, placing smaller boxes in bigger boxes, and adding objects and packing material inside some of them. Of course, the analogy is imperfect because cardboard doesn't stretch!

Figure 7-2. A schematic view of the sizers for PersonalRecordDialog

Programming with Sizers

To create a sizer-based layout, create a top-level sizer (any kind of sizer may be used) and associate it with the parent window with wxWindow::SetSizer. Now you can hang your hierarchy of windows and further sizers from the top-level sizer. If you want the top-level window to fit itself around the contents, you call wxSizer::Fit passing the top-level window. If the window should never be resized smaller than the initial size, call wxSizer::SetSizeHints passing the top-level window. This will call wxWindow::SetSizeHints with the appropriate values.

Instead of the three functions described in the previous paragraph, you can simply call the function wxWindow::SetSizerAndFit, which sets the sizer, calls Fit, and also calls SetSizeHints.

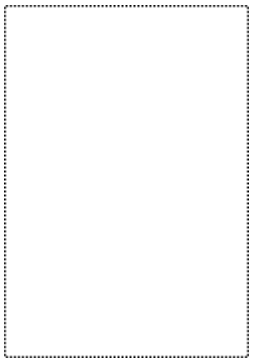
If you have a panel inside a frame, you may be wondering which window gets the top-level sizer. Assuming you only have one panel in the frame, the frame already knows how to size the panel to fill the frame's client area when the frame size is changed. Therefore, you should set the sizer for the panel to manage the panel's children. If you had more than one panel in the frame, you might set a top-level sizer for the frame, which would manage the frame's children. However, you would still need a top-level sizer for each child panel that had its own children to lay out.

The following sections describe each kind of sizer and how to use it.

Programming with wxBoxSizer

wxBoxSizer can lay out its children either vertically or horizontally, depending on the style passed to its constructor. When using a vertical sizer, each child can be centered, aligned to the right, or aligned to the left. Correspondingly, when using a horizontal sizer, each child can be centered, aligned at the bottom, or aligned at the top. The stretch factor described previously is used for the main orientation, so when using a horizontal box sizer, the stretch factor determines how much the child can be stretched horizontally. Figure 7-7 shows the same dialog as in Figure 7-6, except that the sizer is a vertical box sizer.

Figure 7-7. A verical wxBoxSizer



You add child elements to a box sizer with Add:

```
// Add a window
void Add(wxWindow* window, int stretch = 0, int flags = 0,
         int border = 0);

// Add a sizer
void Add(wxSizer* window, int stretch = 0, int flags = 0,
         int border = 0);
```

The first parameter is a window or sizer.

The second is the proportion or stretch factor.

Further Layout Issues

In this section, we'll discuss some further topics to bear in mind when you're working with sizers.

Dialog Units

Although sizers protect you from changes in basic control size on different platforms and in different languages, you may still have some hard-coded sizes in your dialogs (for example, for list boxes). If you would like these sizes to adjust to the current system font (or font supplied by the application), you can use dialog units instead of pixels. Dialog units are based on average character width and height for a window's font, and so the actual pixel dimension for a given dialog unit value will vary according to the current font. wxWindow has functions ConvertDialogToPixels and ConvertPixelsToDialog, and a convenience macro wxDLG_UNIT(window, ptOrSz) that can be used with both wxPoint and wxSize objects. So instead of passing a pixel size to your control, use the wxDLG_UNIT macro, for example:

```
wxListBox* listBox = new wxListBox(parent, wxID_ANY,
    wxDefaultPosition, wxDLG_UNIT(parent, wxSize(60, 20)));
```

Dialog units can be specified in an XRC file by appending "d" to dimension values.

Platform-Adaptive Layouts

Although dialogs on different platforms are largely similar, sometimes the style guides are incompatible. For example, on Windows and Linux, it's acceptable to have right-justified or centered OK, Cancel, and Help buttons, in that order. On Mac OS X, the Help should be on the left, and Cancel and OK buttons are right aligned, in that order.

To help with this issue, wxStdDialogButtonSizer is provided. It's derived from wxBoxSizer, so it can be used in a similar way, but its orientation will depend on platform.

This sizer's constructor has no arguments. There are two ways of adding buttons: pass the button pointer to AddButton, or (if you're not using standard identifiers) call SetAffirmativeButton, SetNegativeButton, and SetCancelButton. If using AddButton, you should use identifiers from this list: wxID_OK, wxID_YES, wxID_CANCEL, wxID_NO, wxID_SAVE, wxID_APPLY, wxID_HELP, and wxID_CONTEXT_HELP.

Then, after the buttons have been added, call Realize so that the sizer can add the buttons in the appropriate order with the appropriate spacing (which it can only do when it knows about all the buttons in the sizer). The following code creates a standard button sizer with OK, Cancel, and Help buttons:

```
wxBoxSizer* topSizer = new wxBoxSizer(wxVERTICAL);
dialog->SetSizer(topSizer);

wxButton* ok = new wxButton(dialog, wxID_OK);
wxButton* cancel = new wxButton(dialog, wxID_CANCEL);
wxButton* help = new wxButton(dialog, wxID_HELP);

wxStdDialogButtonSizer* buttonSizer = new wxStdDialogButtonSizer;
topSizer->Add(buttonSizer, 0, wxEXPAND|wxALL, 10);

buttonSizer->AddButton(ok);
buttonSizer->AddButton(cancel);
buttonSizer->AddButton(help);

buttonSizer->Realize();
```

As a convenience, wxDialog::CreateButtonSizer can be used, indirectly creating a wxStdDialogButtonSizer with buttons based on a flag list. If you look at the dialog implementations in src/generic, you will see that CreateButtonSizer is used for many of them. The flags in Table 7-2 can be passed to this function.

Summary

Sizers takes some getting used to, so don't worry if you found this chapter a bit heavy going. The best way of getting to grips with them is to play with a dialog editor such as DialogBlocks (included on the CD-ROM), experimenting with different layouts and examining the generated code. You can also look at samples/layout in your wxWidgets distribution. After you've tamed them, you'll find sizers a very powerful tool, and their ability to adapt to different platforms and languages will prove to be a huge productivity benefit.

Next, we'll look at the standard dialogs provided by wxWidgets.

Chapter 8. Using Standard Dialogs

This chapter describes the set of standard dialogs that wxWidgets provides for displaying information or getting data from users with just a few lines of code. Becoming familiar with the available standard dialogs is going to save you a lot of coding time, and it will help give your applications a professional feel. Where possible, wxWidgets uses the native dialogs implemented by each windowing system, but some, such as wxTextEntryDialog, are implemented in wxWidgets itself, and these are referred to as "generic" dialogs. In this chapter, we will show pictures of dialogs on more than one platform where there are significant visual differences.

We will divide the dialogs into the categories Informative Dialogs, File and Directory Dialogs, Choice and Selection Dialogs, and Entry Dialogs.

Informative Dialogs

In this section, we'll look at dialogs that present information: wxMessageDialog, wxProgressDialog, wxBusyInfo, and wxShowTip.

wxMessageDialog

This dialog shows a message plus buttons that can be chosen from OK, Cancel, Yes, and No. An optional icon can be shown, such as an exclamation mark or question mark. The message text can contain newlines ("\n").

The return value of wxMessageDialog::ShowModal indicates which button the user pressed.

Figure 8-1 shows how the dialog looks under Windows, Figure 8-2 shows it under GTK+, and Figure 8-3 is the same dialog on Mac OS X.

Figure 8-1. wxMessageDialog under Windows

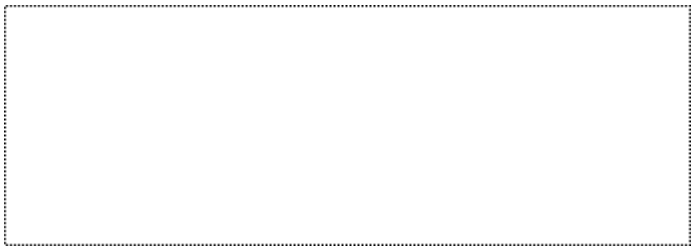


Figure 8-2. wxMessageDialog under GTK+

Figure 8-3. wxMessageDialog under Mac OS X

To create this dialog, pass a parent window, message, and optional caption, style, and position. Then call ShowModal to display the window, and test the returned value.

File and Directory Dialogs

There are two dialogs you can use to get file and directory information from the user: wxFileDialog and wxDirDialog.

wxFileDialog

wxFileDialog can handle the selection of one file or several files, and it has variants for opening and saving files.

Figure 8-7 shows the file dialog under Windows.

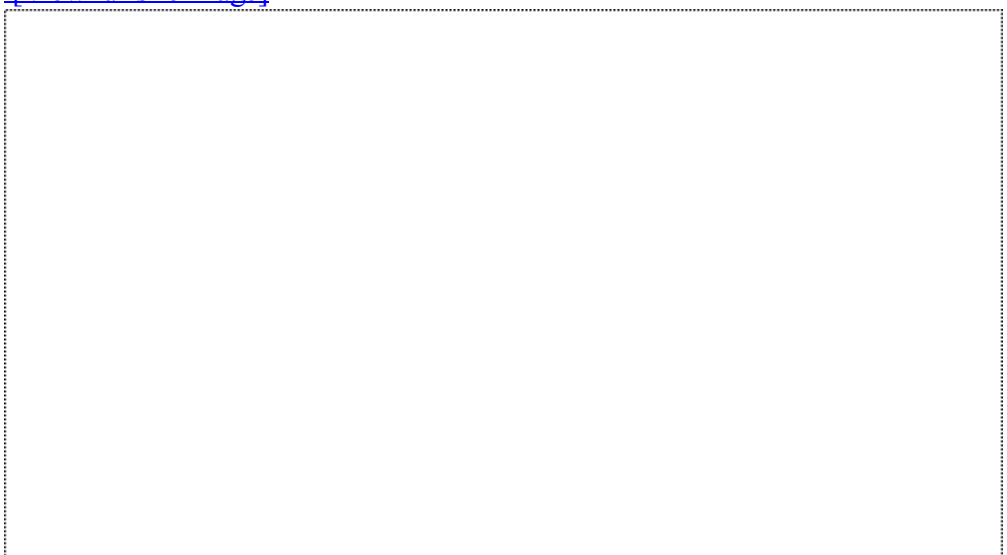
Figure 8-7. wxFileDialog under Windows

[View full size image]

Figure 8-8 and Figure 8-9 show the file dialog under Linux using GTK+ versions 1 and 2, respectively.

Figure 8-8. Generic wxFileDialog under GTK+

[View full size image]



Choice and Selection Dialogs

In this section, we look at dialogs for getting choices and selections from the user: wxColourDialog, wxFontDialog, wxSingleChoiceDialog, and wxMultiChoiceDialog.

wxColourDialog

This dialog allows the user to pick from a standard set or a full range of colors.

Under Windows, the native color selector dialog is used. This dialog contains three main regions: at the top left, a palette of 48 commonly used colors is shown. Below this, there is a palette of 16 custom colors, which can be set by the application. Additionally, the user may add to the custom color palette by expanding the dialog box and choosing a precise color from the color selector panel on the right. Figure 8-14 shows the color selector under Windows in full selection mode.

Figure 8-14. wxColourDialog under Windows

The generic color dialog, shown in Figure 8-15 under GTK+ 1 and X11, shows palettes of 48 standard and 16 custom colors, with the area on the right containing three sliders for the user to select a color from red, green, and blue components. This color may be added to the custom color palette, and it will replace either the currently selected custom color or the first one in the palette if none is selected. The RGB color sliders are not optional in the generic color selector. The generic color selector is also available under Windows and other platforms; use the name wxGenericColourDialog.

Figure 8-15. Generic wxColourDialog under X11

Entry Dialogs

These dialogs ask you to type in information. They include wxNumberEntryDialog, wxTextEntryDialog, wxPasswordEntryDialog, and wxFindReplaceDialog.

wxNumberEntryDialog

wxNumberEntryDialog prompts the user for an integer within a given range. The dialog shows a spin control so that the number can be entered directly or by clicking on the up and down arrows. This dialog is implemented by wxWidgets, so it has the same functionality on all platforms.

Create a wxNumberEntryDialog passing a parent window, message text, prompt text (that will precede the spin control), caption, default value, minimum value, maximum value, and position. Then call ShowDialog and, if wxID_OK is returned, retrieve the number using GetValue.

Figure 8-23 shows what the dialog looks like under Windows.

Figure 8-23. wxNumberEntryDialog under Windows

wxNumberEntryDialog **Example**

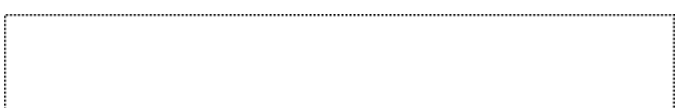
Figure 8-23 was created using the following code:

```
#include "wx/numdlg.h"

wxNumberEntryDialog dialog(parent,
  wxT("This is some text, actually a lot of text\nEven two rows of text"),
  wxT("Enter a number:"), wxT("Numeric input test"), 50, 0, 100);
if (dialog.ShowModal() == wxID_OK)
{
  long value = dialog.GetValue();

}
```

wxTextEntryDialog **and** wxPasswordEntryDialog

wxTextEnTRyDialog and wxPasswordEntryDialog present the user with a single-line text control and a message. They function identically except that the letters typed into a wxPasswordEntryDialog are masked so that they cannot be read. Figure 8-24 shows a wxTextEntryDialog under Windows.

Figure 8-24. wxTextEnTRyDialog under Windows



Printing Dialogs

You use wxPageSetupDialog and wxPrintDialog in applications that print documents. If you use the printing framework (including wxPrintout, wxPrinter, and other classes), you won't need to invoke these dialogs explicitly in your code. For more on printing, refer to Chapter 5, "Drawing and Printing."

wxPageSetupDialog

wxPageSetupDialog contains controls for paper size such as A4 and letter, orientation (landscape or portrait), and controls for setting left, top, right, and bottom margin sizes in millimeters. The user can also set printer-specific options by invoking a further dialog from this one.

Figure 8-27 shows the wxPageSetupDialog dialog under Windows.

Figure 8-27. wxPageSetupDialog under Windows

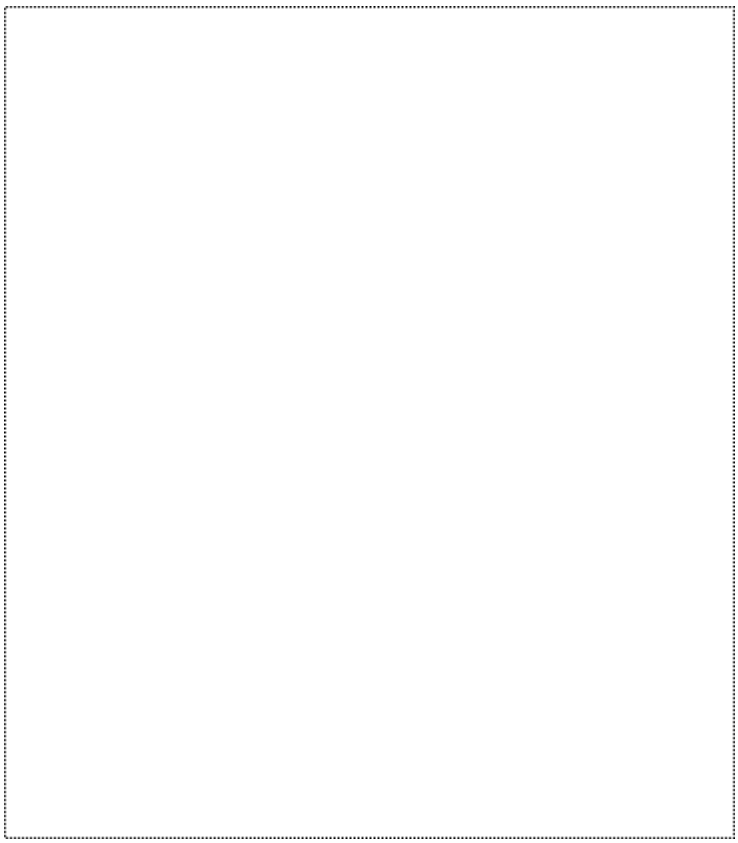


Figure 8-28 shows wxPageSetupDialog using the generic implementation under GTK+. If the GNOME printing libraries are installed, wxWidgets will instead use a native GNOME page setup dialog, as shown in Figure 8-29.

Figure 8-28. wxPageSetupDialog under GTK+ without GNOME printing

Summary

In this chapter, you have learned about the standard dialogs that you can use to present information and retrieve user choices with very little code. For further examples of using standard dialogs, see samples/dialogs in your wxWidgets distribution. Next, we'll show you how to write your own dialogs.

Chapter 9. Writing Custom Dialogs

Sooner or later, you will have to create your own dialogs, whether simple ones with only a few buttons and some text or highly complex dialogs with notebook controls, multiple panels, custom controls, context-sensitive help, and so on. In this chapter, we cover the principles of creating custom dialogs and transferring data between C++ variables and the controls. We also describe the wxWidgets resource system, which enables you to load dialogs and other user interface elements from XML files.

Steps in Creating a Custom Dialog

When you start writing your own specialized dialogs, the fun really starts. Here are the steps you'll typically need to take:

1. Derive a new class from wxDialog.
2. Decide where the data is stored and how the application accesses user choices.
3. Write code to create and lay out the controls.
4. Add code that transfers data between C++ variables and the controls.
5. Add functions and their event table entries to handle events from controls.
6. Add user interface (UI) update handlers to set controls to the correct state.
7. Add help, in particular tooltips, context-sensitive help (not implemented on Mac OS X), and a way of showing an explanation of the dialog in your application's user manual.
8. Invoke the dialog from a suitable place in your application code.

Let's illustrate these steps with a concrete example.

An Example: PersonalRecordDialog

As we saw in the previous chapter, dialogs come in two flavors: modal and modeless. We'll illustrate custom dialog creation with a modal dialog because it's the more common kind and has fewer complications. The application will invoke the dialog with ShowModal and then query the dialog for user selections. Until ShowModal returns, all user interactions with the application will be contained within the little world of your custom dialog (and any further modal dialogs that your dialog may invoke).

Many of the steps involved in creating a custom dialog can be accomplished very easily by using a dialog editor, such as wxDesigner or DialogBlocks. The amount of coding left to do depends on the complexity of your dialog. Here, we will assume handcrafting of all the code in order to demonstrate the principles, but it's highly recommended that you use a tool to help you because it will save you many hours of repetitive work.

We'll illustrate the steps involved in creating a custom dialog with a simple example where the user is required to enter his or her name, age, sex, and whether the user wants to vote. This dialog is called PersonalRecordDialog, as shown in Figure 9-1.

Figure 9-1. Personal record dialog under Windows

The Reset button restores all controls to their default values. The OK button dismisses the dialog and returns wxID_OK from ShowModal. The Cancel button returns wxID_CANCEL and does not update the dialog's variables from the values shown in the controls. The Help button invokes a few lines of text describing the dialog (although in a real application, this button should invoke a nicely formatted help file).

A good user interface should not allow the user to enter data that has no meaning in the current context. In this example, the user should not be able to use the Vote control if Age is less than the voting age (18 in the U.S. or U.K.). So, we will ensure that when the age entered is less than 18, the Vote check box is disabled.

Deriving a New Class

Here's the declaration for our PersonalRecordDialog. We provide run-time type information by using DECLARE_CLASS, and we add an event table with DECLARE_EVENT_TABLE.

```
/*!
 * PersonalRecordDialog class declaration
 */

class PersonalRecordDialog: public wxDialog
{
    DECLARE_CLASS( PersonalRecordDialog )
    DECLARE_EVENT_TABLE()
```

Adapting Dialogs for Small Devices

wxWidgets can be used on mobile and other embedded devices, using GTK+, X11, and Windows CE ports (and others in the future). The most obvious limitation associated with many of these devices is the size of the display, which for a smartphone may be as little as 176x220 pixels.

Many dialogs will need an alternative dialog layout for small displays; some controls may be omitted altogether, especially as the functionality of the application may be reduced compared with a desktop application. You can detect the size of the device with wxSystemSettings::GetScreenType, for example:

```
#include "wx/settings.h"
bool isPda = (wxSystemSettings::GetScreenType() <= wxSYS_SCREEN_PDA);
```

GetScreenType returns one of the values listed in Table 9-1. Because the types increase in value as the screen size increases, you can use integer comparison operators to deal with classes of devices with screens below a certain size, as in the example we've just seen.

Table 9-1. Screen Types

wxSYS_SCREEN_NONE	Undefined screen type
wxSYS_SCREEN_TINY	Tiny screen, less than 320x240
wxSYS_SCREEN_PDA	PDA screen, 320x240 or more but less than 640x480
wxSYS_SCREEN_SMALL	Small screen, 640x480 or more but less than 800x600
wxSYS_SCREEN_DESKTOP	Desktop screen, 800x600 or more

If you need more detail about the display size, there are three ways to get it:

1. Use wxSystemSettings::GetMetric, passing wxSYS_SCREEN_X or wxSYS_SCREEN_Y.
2. Call wxGeTDisplaySize, which returns a wxSize object.
3. Create a wxDisplay object and call GetGeometry, which returns a wxRect containing the bounding rectangle of the display.

When you know you may have a stunted display to run on, what can you do with this information? Here are some strategies you can use:

1. Replace the whole layout by loading a different XRC file or executing different control creation code. If the controls don't change type, you may not need to change the event handling code at all.
2. Reduce the number of controls and space.
3.

Further Considerations in Dialog Design

Here are a few tips to help you create professional-looking dialogs.

Keyboard Navigation

Provide mnemonics in static text labels and other labeled controls by inserting ampersands in front of characters. On some platforms (notably Windows and GTK+), this will help the user navigate between controls.

Always provide a means for the user to cancel the dialog, preferably with the option of using the Escape key. If a dialog has a button with the identifier wxID_CANCEL, its handler will automatically be called when the user presses the Escape key. So, if you have a Close button, consider giving it the wxID_CANCEL identifier.

Provide a default button (often OK)for example, by calling wxButton:: SetDefault. The command for this button will be invoked when the user presses the Enter key.

Data and UI Separation

To simplify the example, the data variables that PersonalRecordDialog uses are stored in the class itself. However, a better design would be to provide a data class separate from the dialog class, with a copy constructor and assignment operator, so that you can pass a copy of the data to the dialog and retrieve the modified data from the dialog only if the user confirms any changes. This is the approach adopted for some of the standard dialogs. As an exercise, you can rewrite the PersonalRecordDialog using a PersonalRecordData class. The dialog constructor will take a PersonalRecordData reference, and there will be a Getdata function so that the calling application can retrieve the data.

In general, always consider how you can separate out the UI functionality from non-UI functionality. The result will usually be code that is more compact and easier to understand and debug. Don't be afraid to introduce new classes to make the design more elegant, and make use of copy constructors and assignment operators so that objects can easily be copied and assigned without the application having to repeat lots of low-level code.

Unless you provide an Apply button that commits your changes to the underlying data, canceling the dialog should leave the application data in the same state as it was before the dialog was opened. The use of a separate data class makes this easier to achieve because the dialog isn't editing "live" data but rather a copy.

Layout

If your dialog looks claustrophobic or somehow odd, it may be due to a lack of space. Try adding a bigger border around the edge of the dialog by using an additional sizer (as in our PersonalRecordDialog example) and adding space between groups of controls. Use wxStaticBoxSizer and wxStaticLine to logically group or separate controls. Use wxGridSizer and wxFlexGridSizer to align controls and their labels so that they don't appear as a random jumble. In sizer-based layouts, use expanding spacers to align a group of controls. For example, often OK, Cancel, and Help buttons are in a right-aligned group, which can be achieved by placing a spacer and the buttons in a horizontal wxBoxSizer and setting the spacer to expand horizontally (give it a positive stretch factor).

If possible and appropriate, make your dialog resizable. Traditionally, Windows dialog boxes haven't often been resizable, but there is no reason why this should be the case, and fiddling with tiny controls on a large display can be a frustrating experience for the user. wxWidgets makes it easy to create resizable dialogs with sizers, and you should be using sizers anyway to allow for font and control size differences and changes in language. Choose carefully which elements should grow; for example, there may be a multi-line text control that is a good candidate for growing and giving the user more elbow room. Again, you can put expanding spacers to good use to preserve alignment in a resized dialog. Note that we're not resizing controls in the sense of zooming in and out, making text bigger or smallerwe're simply giving more or less space for items in the control. See Chapter 7 for more about sizers.

If you find that your dialog is becoming too large, split it up into a number of panels and use a wxNotebook,

Using wxWidgets Resource Files

You can load specifications of dialogs, frames, menu bars, toolbars, and so on from XML files with extension xrc instead of creating these elements explicitly in C++ code. This enables better separation of code and user interface, such as enabling an application's dialog design to be changed at runtime. XRC files can be exported by a range of UI design tools, including wxDesigner, DialogBlocks, XRCed, and wxGlade.

Loading Resources

To use XRC files in your application, you need to include wx/xrc/xmlres.h in your application code.

If you will be converting your XRC files to binary XRS files, as we will describe shortly, install the zip file system handler by placing an AddHandler call in your OnInit function:

```
#include "wx/filesys.h"
#include "wx/fs_zip.h"

wxFileSystem::AddHandler(new wxZipFSHandler);
```

Initialize the XRC system by adding this to your OnInit:

```
wxXmlResource::Get()->InitAllHandlers();
```

Load the XRC file with code like this:

```
wxXmlResource::Get()->Load(wxT("resources.xrc"));
```

This makes wxWidgets aware of the resources in the file; to create a real UI element, we need another call. For example, the following fragment creates a dialog whose resource name is dialog1:

```
MyDialog dlg;
wxXmlResource::Get()->LoadDialog(& dlg, parent, wxT("dialog1"));
dlg.ShowModal();
```

The following code shows how to load menu bars, menus, toolbars, bitmaps, icons, and panels.

```
MyFrame::MyFrame(const wxString& title): wxFrame(NULL, -1, title)
{
    SetMenuBar(wxXmlResource::Get()->LoadMenuBar(wxT("mainmenu")));
    SetToolBar(wxXmlResource::Get()->LoadToolBar(this,
                                                 wxT("toolbar")));

    wxMenu* menu = wxXmlResource::Get()->LoadMenu(wxT("popupmenu"));

    wxIcon icon = wxXmlResource::Get()->LoadIcon(wxT("appicon"));
    SetIcon(icon);

    wxBitmap bitmap = wxXmlResource::Get()->LoadBitmap(wxT("bmp1"));

    // Finish creating panelA after making an instance of it
    MyPanel* panelA = new MyPanel;
    panelA = wxXmlResource::Get()->LoadPanel(panelA, this,
                                                    wxT("panelA"));

    // A second method: get XRC to both create and load panelB
    wxPanel* panelB = wxXmlResource::Get()->LoadPanel(this,
                                                    wxT("panelB"));
}
```

Summary

In this chapter, you have learned the fundamentals of custom dialog design and implementation, including a quick look at sizers, the use of validators, and the advantages of using UI update events. For examples of creating custom dialogs, see samples/dialogs in your wxWidgets distribution. Also see samples/validate for use of the generic and text validator classes. Next, we'll look at how to handle images.

Chapter 10. Programming with Images

This chapter shows what you can do with bitmapped images. Images are great for introducing "design values" into your application, and they can be used with controls such as toolbars, tree controls, notebooks, buttons, HTML windows, or in custom drawing code. Sometimes they can be used invisibly in an application, for example to achieve flicker-free drawing. In this chapter, we cover the different image classes and how to override standard icons and bitmaps used with wxWidgets.

Image Classes in wxWidgets

wxWidgets supports four kinds of bitmap images: wxBitmap, wxIcon, wxCursor, and wxImage.

wxBitmap represents a platform-dependent bitmap, with an optional wxMask to support drawing with transparency. On Windows, wxBitmap is implemented using device-independent bitmaps (DIBs). On GTK+ and X11, each wxBitmap contains the pixmap object of GDK and X11, respectively. On Mac, a PICT is used. A wxBitmap can be converted to and from a wxImage.

wxIcon represents the platform's concept of an icon, a small image with transparency that can be used for giving frames and dialogs a recognizable visual cue, among other things. On GTK+, X11, and Mac, an icon is simply a bitmap that always has a wxMask. On Windows, an icon is represented by an HICON object.

wxCursor represents the mouse pointer image; this is a GdkCursor on GTK+, a Cursor on X11, an HCURSOR in Windows, and a Cursor on Mac. It has the notion of a hotspot (the pixel in the cursor image that is considered to be the exact mouse pointer location) and a mask.

wxImage is the only class of the four with a platform-independent implementation, supporting 24-bit images with an optional alpha channel. A wxImage can be created from data or by using wxBitmap::ConvertToImage. A wxImage can be loaded from a file in a variety of formats, and it is extensible to new formats via image format handlers. Functions are available to set and get image bits, so it can be used for basic image manipulation. Unlike a wxBitmap, a wxImage cannot be drawn directly to a wxDC. Instead, a wxBitmap object must be created from the wxImage. This bitmap can then be drawn in a device context by using wxDC:: DrawBitmap. wxImage supports a mask color indicating transparent areas, and it also supports alpha channel data to allow for more sophisticated transparency effects.

You can convert between these bitmap objects, though there are platform dependencies on some conversion operations.

Note that all image classes are reference-counted, so assignment and copying are very cheap operations because the image data itself is not copied. However, you need to be aware that if you change an image, other image objects that refer to the same image data will also be changed.

All image classes use standard wxBitmapType identifiers for loading and saving bitmap data, as described in Table 10-1.

Table 10-1. Bitmap Types

wxBITMAP_TYPE_BMP	A Windows bitmap file (BMP).
wxBITMAP_TYPE_BMP_RESOURCE	A Windows bitmap to be loaded from the resource part of the executable.
wxBITMAP_TYPE_ICO	A Windows icon file (ICO).
wxBITMAP_TYPE_ICO_RESOURCE	A Windows icon to be loaded from the resource part of the executable.
wxBITMAP_TYPE_CUR	A Windows cursor (CUR).
wxBITMAP_TYPE_CUR_RESOURCE	A Windows cursor to be loaded from the resource part of the executable.

Programming with wxBitmap

These are some of the things you can do with a wxBitmap:

- Draw it on a window via a device context.
- Use it as a bitmap label for classes such as wxBitmapButton, wxStaticBitmap, and wxToolBar.
- Use it to implement double buffering (drawing into an off-screen wxMemoryDC before drawing to a window).

On some platforms (in particular, Windows), the bitmap is a limited resource, so if you have many images to store in memory, you may prefer to work mainly with wxImage objects and convert to a temporary wxBitmap when drawing on a device context.

Before discussing how to create wxBitmap and draw with it, let's summarize the main functions (Table 10-2).

Table 10-2. wxBitmap Functions

wxBitmap	A bitmap can be created given a width and height, another bitmap, a wxImage, XPM data (char**), raw data (char[]), or a file name and type.
ConvertToImage	Converts to a wxImage, preserving transparency.
CopyFromIcon	Creates the bitmap from a wxIcon.
Create	Creates the bitmap from data or a given size.
GetWidth, GetHeight	Returns the bitmap's size.
Getdepth	Returns the bitmap's color depth.
GetMask, SetMask	Returns the wxMask object or NULL.
GetSubBitmap	Returns an area of the bitmap as a new bitmap.
LoadFile, SaveFile	Files can be loaded and (for some formats) saved.
Ok	Returns TRue if the bitmap's data is present.

Creating a wxBitmap

There are several ways to create a wxBitmap object.

You can create the object in an uninitialized state (no bitmap data) by using the default constructor. You will need to

Programming with wxIcon

A wxIcon is a small bitmap that always has a mask. Its uses include

- Setting the icon for a frame or dialog
- Adding icons to a wxTReeCtrl, wxListCtrl, or wxNotebook via the wxImageList class (see more information later in this chapter)
- Drawing an icon on a device context with wxDC::DrawIcon

Table 10-3 summarizes the major icon functions.

Table 10-3. wxIcon Functions

wxIcon	An icon can be created given another icon, XPM data (char**), raw data (char[]), or a file name and type.
CopyFromBitmap	Creates the icon from a wxBitmap.
GetWidth, GetHeight	Returns the icon's size.
Getdepth	Returns the icon's depth.
LoadFile	Files can be loaded.
Ok	Returns TRue if the icon's data is present.

Creating a wxIcon

A wxIcon object can be created from XPM data included in the application, from a wxBitmap object, from raw data, or by loading the icon from a file, such as a transparent XPM file. wxWidgets provides the wxICON macro, which is similar to the wxBITMAP macro described earlier; the icon is loaded either from a platform-specific resource or from XPM data.

On Windows, LoadFile and the equivalent constructor will work for Windows bitmap (BMP) and icon (ICO) resources and files. If you want to load other formats, load the file into a wxBitmap and convert it to an icon.

On Mac OS X and Unix/Linux with GTK+, wxIcon has the same file loading capabilities as wxBitmap.

The following code fragment shows four different ways to create a wxIcon object.

```
// Method 1: load from XPM data
#include "icon1.xpm"
wxIcon icon1(icon1_xpm);

// Method 2: load from an ICO resource (Window and OS/2 only)
wxIcon icon2(wxT("icon2"));

// Method 3: load from an ICO file (Windows and OS/2 only)
```

Programming with wxCursor

A cursor is used to give feedback on the mouse pointer position. You can change the cursor for a given windowusing different cursors gives a cue to the user to expect specific mouse behavior. Like icons, cursors are small, transparent images that can be created using platform-specific as well as generic constructors. Some of these constructors take a hotspot position relative to the top-left corner of the cursor image, with which you specify the location of the actual pointer "tip."

Table 10-4 shows the cursor functions.

Table 10-4. wxCursor Functions

wxCursor	A cursor can be created from a wxImage, raw data (char[]), a stock cursor identifier, or a file name and type.
Ok	Returns TRue if the cursor's data is present.

Creating a wxCursor

The easiest way to create a cursor is to pass a stock cursor identifier to the cursor constructor, as the following example shows.

```
// Create a cursor from a stock identifier
wxCursor cursor(wxCURSOR_WAIT);
```

Table 10-5 lists the available identifiers and their appearances (subject to some variation between platforms).

Table 10-5. Stock Cursor Identifiers

wxCURSOR_ARROW		Standard arrow cursor.
wxCURSOR_RIGHT_ARROW		Standard arrow cursor pointing to the right.
wxCURSOR_BLANK		Transparent cursor.
wxCURSOR_BULLSEYE		Bullseye cursor.
wxCURSOR_CROSS		Cross cursor.
wxCURSOR_HAND		Hand cursor.
wxCURSOR_IBEAM		I-beam cursor (vertical line).
wxCURSOR_LEFT_BUTTON		Represents a mouse with the left button depressed (GTK+ only).

Programming with wxImage

Use wxImage when you need to manipulate images in a platform-independent manner, or as an intermediate step for loading or saving image files. Images are stored using a byte per pixel for each of the red, green, and blue channels, plus a further byte per pixel if an alpha channel is present.

The major wxImage functions are listed in Table 10-6.

Table 10-6. wxImage Functions

wxImage	An image can be created given a width and height, another image, XPM data, raw data (char[]) and optional alpha data, a file name and type, or an input stream.
ConvertAlphaToMask	Converts the alpha channel (if any) to a mask.
ConvertToMono	Converts to a new monochrome image.
Copy	Returns an identical copy without using reference counting.
Create	Creates an image of a given size, optionally initializing it from data.
Destroy	Destroys the internal data if no other object is using it.
GeTData, SetData	Gets and sets its pointer to internal data (unsigned char*).
GetImageCount	Returns the number of images in a file or stream.
GetOption, GetOptionInt, SetOption, HasOption	Gets, sets, and tests for the presence of options.
GetSubImage	Returns an area of the image as a new image.
GetWidth, GetHeight	Returns the image size.
Getred, GetGreen, GetBlue, SetRGB, GetAlpha, SetAlpha	Gets and sets the red, blue, green, and alpha value for a pixel.
HasMask, GetMaskRed, GetMaskGreen, GetMaskBlue, SetMaskColour	Functions for testing for the presence of a mask and setting and getting the mask color.
LoadFile, SaveFile	Files can be loaded and saved using various formats.
Mirror	Mirrors the image in either orientation, returning a new image.

Image Lists and Icon Bundles

Sometimes it's useful to aggregate a number of images. You can use wxImageList directly in your application or in conjunction with some of the wxWidgets controls that require image lists when setting icons. wxNotebook, wxtreeCtrl, and wxListCtrl all support wxImageList to identify the icons used in the controls. You can also draw an individual image in a wxImageList on a device context.

Create a wxImageList with the width and height of each image, a boolean to specify whether a mask will be used, and the initial size of the list (purely for internal optimization purposes). Then add one or more wxBitmap or wxIcon images. You can't add a wxImage directly, but you can pass one to wxBitmap's constructor. wxImageList::Add returns an integer index you can use to identify that image; after you have added an image, the original image can be destroyed because wxImageList makes a copy of it.

Here are some examples of creating a wxImageList and adding images to it.

```
// Create a wxImageList
wxImageList *imageList = new wxImageList(16, 16, true, 1);

// Add a bitmap with transparency from a PNG
wxBitmap bitmap1(wxT("image.png"), wxBITMAP_TYPE_PNG);
imageList->Add(bitmap1);

// Add a bitmap with transparency from another bitmap
wxBitmap bitmap2(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
wxBitmap maskBitmap(wxT("mask.bmp"), wxBITMAP_TYPE_BMP);
imageList->Add(bitmap2, maskBitmap);

// Add a bitmap with transparency specified with a color
wxBitmap bitmap3(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
imageList->Add(bitmap3, *wxRED);

// Add an icon
#include "folder.xpm"
wxIcon icon(folder_xpm);
imageList->Add(icon);
```

You can draw an image to a device context, passing flags that determine how the image will be drawn. Pass wxIMAGELIST_DRAW_TRANSPARENT to draw with transparency, and also one of these values to indicate the state that should be drawn: wxIMAGELIST_DRAW_NORMAL, wxIMAGELIST_DRAW_SELECTED, or wxIMAGELIST_DRAW_FOCUSED.

```
// Draw all the images in the list
wxClientDC dc(window);
size_t i;
for (i = 0; i < imageList->GetImageCount(); i++)
{
    imageList->Draw(i, dc, i*16, 0, wxIMAGELIST_DRAW_NORMAL|
                                    wxIMAGELIST_DRAW_TRANSPARENT);
}
```

To associate icons with notebook tabs, create an image list containing 16x16 icons, and call wxNotebook::SetImageList or wxNotebook::AssignImageList. If you use the first form, the notebook doesn't delete the image list when it is destroyed; with the second form, the notebook takes over management of the list, and you don't have to worry about destroying it yourself. Now when you add pages, you can specify the index of an icon to use as the image next to the text label (or instead of it, if the text label is empty). The following code creates a notebook and adds two pages with icons on the tabs.

```
// Create a wxImageList
wxImageList *imageList = new wxImageList(16, 16, true, 1);
```

Customizing Art in wxWidgets

wxArtProvider is a class that allows you to customize the built-in graphics ("art") in a wxWidgets application. For example, you might want to replace the standard icons used by the wxWidgets HTML Help viewer or the icons used by the generic dialogs such as the log dialog.

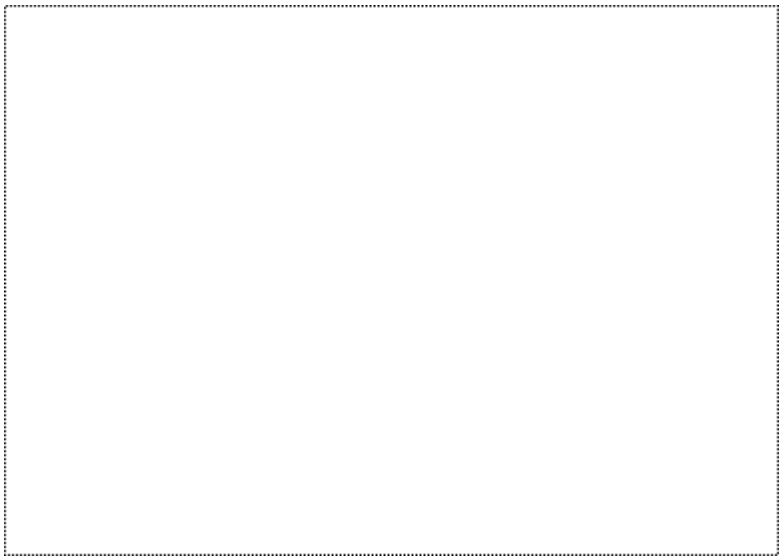
wxWidgets provides a standard wxArtProvider object, and parts of the framework that need icons and bitmaps call wxArtProvider::GetBitmap and wxArtProvider::GetIcon to retrieve a graphic.

Art is specified by two identifiers: the art identifier (wxArtID) and client identifier (wxArtClient). The client identifier is only a hint in case different windows need different graphics for the same art identifier. As an example, the wxHTML help window uses this code to get a bitmap for the Back toolbar button:

```
wxBitmap bmp = wxArtProvider::GetBitmap(wxART_GO_BACK,wxART_TOOLBAR);
```

You can browse the identifiers and graphics that are built into wxWidgets by compiling and running samples/artprov in your wxWidgets distribution. Figure 10-1 shows the browser in action.

Figure 10-1. Art resources browser



To provide your own replacements for wxWidgets art, simply derive a new class from wxArtProvider, override CreateBitmap, and call wxArtProvider::PushProvider from your application's OnInit function to make it known to wxWidgets. Here's an example that replaces most of the wxHTML help window artwork.

```
// XPMs with the art
#include "bitmaps/helpbook.xpm"
#include "bitmaps/helppage.xpm"
#include "bitmaps/helpback.xpm"
#include "bitmaps/helpdown.xpm"
#include "bitmaps/helpforward.xpm"
#include "bitmaps/helpoptions.xpm"
#include "bitmaps/helpsidepanel.xpm"
#include "bitmaps/helpup.xpm"
#include "bitmaps/helpuplevel.xpm"
#include "bitmaps/helpicon.xpm"

#include "wx/artprov.h"

// The art provider class
class MyArtProvider : public wxArtProvider
{
```

Summary

In this chapter, we've seen how to use the four major image classeswxBitmap, wxIcon, wxCursor, and wxImageand two classes for aggregating images wxImageList and wxIconBundle. We've also looked at how you can replace the standard wxWidgets icons and bitmaps with your own images. For examples of using image classes, see samples/image, samples/listctrl, and samples/dragimag in your wxWidgets distribution.

Next, we'll tackle the classes that you use to implement the transfer of data objects via the clipboard or drag and drop.

Chapter 11. Clipboard and Drag and Drop

Most applications offer transfer of data to and from the clipboard via copy, cut, and paste. It's a basic way of implementing interoperation between your application and others. More sophisticated applications also allow the user to drag objects between windows, either within a single application or between two applications. For example, dragging a file from a file browser to an application window causes the data to fill the window, be added to a list, or some other behavior. This can be a much faster way to associate data with the application than using menus and dialogs to achieve the same thing, and your users will appreciate having it as an option.

Clipboard and drag and drop operations share some classes in wxWidgets, reflecting the fact that they both deal with data transfer, and so this chapter deals with both topics together. We'll see how to use the standard data objects that wxWidgets provides, as well as how to implement our own.

Data Objects

The wxDataObject class is at the heart of both clipboard and drag and drop. Instances of classes derived from wxDataObject represent the data that is being dragged by the mouse during a drag and drop operation or copied to or pasted from the clipboard. wxDataObject is a "smart" piece of data because it knows which formats it supports (via GetFormatCount and GetAllFormats) and knows how to render itself in any of them (via GetdataHere). It can also receive its value from outside the application in a format it supports if it implements the SetData method. We'll see how to do that later in the chapter.

Standard data formats such as wxDF_TEXT are identified by integers, and custom data formats are identified by a text string. The wxDataFormat class represents both of these kinds of identifiers by virtue of a constructor for each. Table 11-1 lists the standard data formats.

Table 11-1. Standard Data Formats

wxDF_INVALID	An invalid format, used as default argument for functions taking a wxDataFormat argument.
wxDF_TEXT	Text format. Standard data object: wxTexTDataObject.
wxDF_BITMAP	Bitmap format. Standard data object: wxBitmapDataObject.
wxDF_METAFILE	Metafile (Windows only). Standard data object: wxMetafileData Object.
wxDF_FILENAME	A list of file names. Standard data object: wxFileDataObject.

You can also create a custom data format by passing an arbitrary string to the wxDataFormat constructor. The format will be registered the first time it is referenced.

Both clipboard and drag and drop deal with a source (data provider) and a target (data receiver). These may be in the same application and even the same window when, for example, you drag some text from one position to another in a word processor. Let's describe what each should do.

Data Source Duties

The data source is responsible for creating a wxDataObject containing the data to be transferred. Then the data source should either pass the wxDataObject to the clipboard using the SetData function or pass it to a wxDropSource object when dragging starts and call the DoDragDrop function.

The main difference from a clipboard operation is that the object for clipboard transfer must always be created on the heap using new and will be freed by the clipboard when it is no longer needed. Indeed, it is not known in advance when, if ever, the data will be pasted from the clipboard. On the other hand, the object for the drag and drop operation must exist only while DoDragDrop executes and may be safely deleted afterwards, so it can be created either on the heap or on the stack (that is, as a local variable).

Another small difference is that in the case of a clipboard operation, the application usually knows in advance whether it copies or cuts data. In a clipboard cut, the data is copied and then removed from the object being edited. This usually depends on which menu item the user chose. But for drag and drop, the application can only know this information after DoDragDrop returns.

Using the Clipboard

To use the clipboard, you call member functions of the global pointer wxTheClipboard.

Before copying or pasting, you must take temporary ownership of the clipboard by calling wxClipboard::Open. If this operation returns TRue, you now own the clipboard. Call wxClipboard::SetData to put data on the clipboard or wxClipboard::GetData to retrieve data from the clipboard. Call wxClipboard::Close to close the clipboard and relinquish ownership. You should keep the clipboard open only as long as you are using it.

wxClipboardLocker is a helpful class that will open the clipboard (if possible) in its constructor and close it in its destructor, so you can write

```
wxClipboardLocker locker;
if (!locker)
{
   ... report an error and return ...
}
... use the clipboard ...
```

The following code shows how to write text to and read text from the clipboard:

```
// Write some text to the clipboard
if (wxTheClipboard->Open())
{
    // Data objects are held by the clipboard,
    // so do not delete them in the app.
    wxTheClipboard->SetData(new wxTextDataObject(wxT("Some text")));
    wxTheClipboard->Close();
}

// Read some text
if (wxTheClipboard->Open())
{
    if (wxTheClipboard->IsSupported(wxDF_TEXT))
    {
        wxTextDataObject data;
        wxTheClipboard->GetData(data);
        wxMessageBox(data.GetText());
    }
    wxTheClipboard->Close();
}
```

Here's the same thing, but with bitmaps:

```
// Write a bitmap to the clipboard
wxImage image(wxT("splash.png"), wxBITMAP_TYPE_PNG);
wxBitmap bitmap(image.ConvertToBitmap());
if (wxTheClipboard->Open())
{
    // Data objects are held by the clipboard,
    // so do not delete them in the app.
    wxTheClipboard->SetData(new wxBitmapDataObject(bitmap));
    wxTheClipboard->Close();
}

// Read a bitmap
if (wxTheClipboard->Open())
{
    if (wxTheClipboard->IsSupported(wxDF_BITMAP))
    {
        wxBitmapDataObject data;
```

Implementing Drag and Drop

You may implement drag sources, drag targets, or both in your application.

Implementing a Drag Source

To implement a drag sourcethat is, to provide the data that may be dragged by the user to a targetyou use an instance of the wxDropSource class. Note that the following describes what happens after your application has decided that a drag is startingthe logic to detect the mouse motion that indicates the start of a drag is left entirely up to the application. Some controls help you by generating an event when dragging is starting, so you don't have to code the logic yourself (which could potentially interfere with the native mouse behavior for the control). This chapter provides a summary of when wxWidgets notifies you of the start of a drag.

The following steps are involved, as seen from the perspective of the drop source.

1 Preparation

First of all, a data object must be created and initialized with the data you want to drag. For example:

```
wxTextDataObject myData(wxT("This text will be dragged."));
```

2 Drag Start

To start the dragging process, typically in response to a mouse click, you must create a wxDropSource object and call wxDropSource::DoDragDrop, like this:

```
wxDropSource dragSource(this);
dragSource.SetData(myData);
wxDragResult result = dragSource.DoDragDrop(wxDrag_AllowMove);
```

The flags you can pass to DoDragDrop are listed in Table 11-2.

Table 11-2. Flags for DoDragDrop

wxDrag_CopyOnly	Only allow copying.
wxDrag_AllowMove	Allow moving.
wxDrag_DefaultMove	The default operation is to move the data.

When creating the wxDropSource object, you have the option of also specifying the window that initiates the drag, and three cursors for Copy, Move, and Can't Drop feedback. These are actually icons in GTK+ and cursors on other platforms, so the macro wxDROP_ICON can be used to hide this difference, as we'll see in our text drop example shortly.

3 Dragging

The call to DoDragDrop blocks the program until the user releases the mouse button (unless you override the GiveFeedback function to do something special). When the mouse moves in a window of a program that understands the same drag and drop protocol, the corresponding wxDropTarget methods are calledsee the following section, "Implementing a Drop Target."

Summary

In this chapter, we've seen how to transfer data to and from the clipboard. We've also seen how to implement drag and drop from the point of view of both the data being dropped and the window that's receiving the data, and we've covered other areas of wxWidgets related to drag and drop. Check out samples/dnd, samples/dragimag, and samples/treectrl in your wxWidgets distribution for further insight.

In the next chapter, we will be returning to the topic of windows and describing some advanced classes that will help you take your application to new levels of sophistication.

Chapter 12. Advanced Window Classes

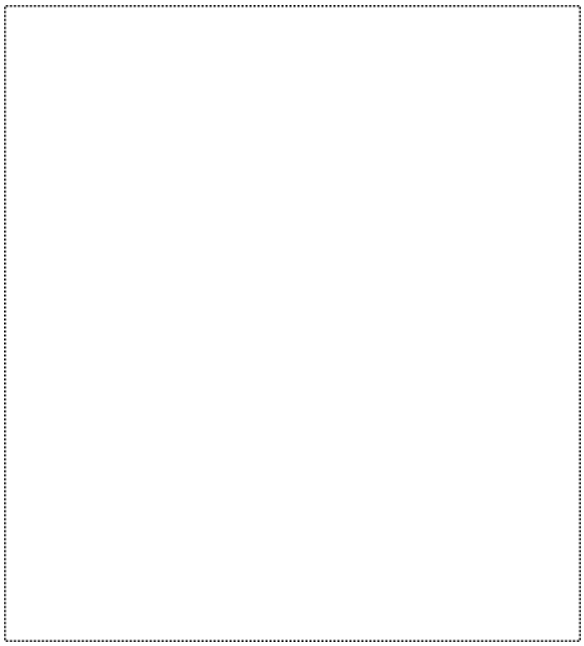
Although this book can't cover all the classes in wxWidgets in detail, it's worth looking at a few of the more advanced GUI classes that can contribute to the creation of a more interesting application. This chapter covers the following topics:

- wxTReeCtrl; a control that helps you model hierarchical data.
- wxListCtrl; a flexible control for showing lists of text labels and icons in several styles.
- wxWizard; a special dialog to guide your user through a specific task using a sequence of pages.
- wxHtmlWindow; a highly versatile, lightweight HTML control for use in anything from "About" boxes to report windows.
- wxGrid; a feature-rich control for displaying tabular data.
- wxTaskBarIcon; a quick way for your users to access features in your application from the system tray or equivalent.
- Writing your own controls. The necessary steps to build a well-behaved "custom control."

wxTreeCtrl

A tree control presents information as a hierarchy, with items that may be expanded or collapsed. Figure 12-1 shows the wxWidgets tree control sample, displaying different font styles and colors. Each item is referenced by the wxtreeItemId type and has text and an optional icon that can be changed dynamically. A tree control can be created in single-selection or multiple selection mode. To associate data with tree items, derive a class from wxTRee ItemData and use wxTReeCtrl::SetItemData and wxTReeCtrl::GetItemData. The tree data will be destroyed when the item or tree is destroyed, so you may want to store only a pointer to the actual data within your tree item data objects.

Figure 12-1. wxTReeCtrl



Because clicks on tree item images can be detected by the application, you can simulate controls within the tree by swapping the images according to the state you want to show for that item. For example, you can easily add simulated check boxes to your tree.

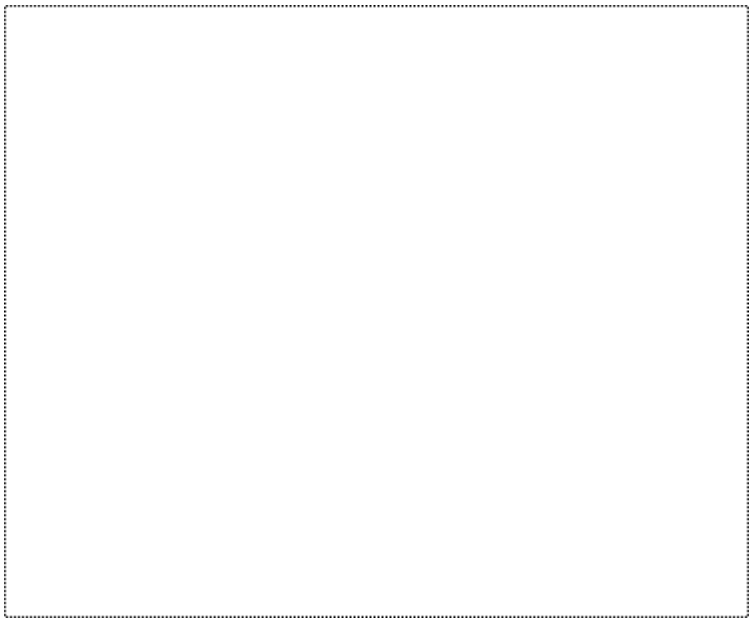
The following fragment shows how to create a tree window with custom tree item data and an image list.

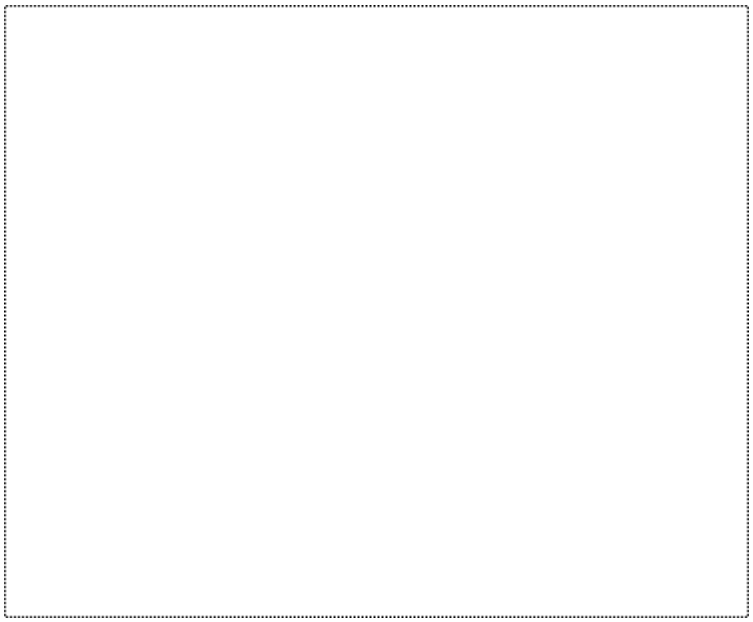
```
#include "wx/treectrl.h"

// Declare a class to hold tree item data
class MyTreeItemData : public wxTreeItemData
{
public:
    MyTreeItemData(const wxString& desc) : m_desc(desc) { }

    const wxString& GetDesc() const { return m_desc; }

private:
    wxString m_desc;
};

// Images for tree items
#include "file.xpm"
#include "folder.xpm"

// Create the tree
wxTreeCtrl* treeCtrl = new wxTreeCtrl(
    this, wxID_ANY, wxPoint(0, 0), wxSize(400, 400),
    wxTR_HAS_BUTTONS|wxTR_SINGLE);
```

wxListCtrl

The list control displays items in one of four views: a multi-column list view, a multi-column report view with optional icons, a large icon view, and a small icon view. See Figure 12-2 for examples of these views from the wxListCtrl sample. Each item in a list control is identified by a long integer index representing its zero-based position in the list, which will change as items are added and deleted or as the content is sorted. Unlike the tree control, the default is to allow multiple selections, and single-selection can be specified with a window style. You can supply a sort function if you need the items to be sorted. The report view can display an optional header, whose columns respond to mouse clicksuseful for sorting the contents of the column, for example. The widths of a report view's columns can be changed either by the application or by dragging on the column dividers.

Figure 12-2. wxListCtrl in report, list, icon, and small icon modes





wxWizard

The wizard is a great way to break a complex set of choices and settings down into a sequence of simple dialogs. It can be presented to novice users to help them get started with a particular feature in an application, such as gathering information for a new project, exporting data, and so on. Often the settings presented in a wizard can be altered elsewhere in the application's user interface, but presenting them in a wizard focuses the user on the essentials for getting a specific task completed.

A wizard comprises a series of dialog-like pages set inside a window that normally has an image on the left (the same for all pages, or different for each page), and a row of buttons along the bottom for navigating between pages and getting help. As the user progresses through the wizard, the old page is hidden and a new one is shown. The path through a wizard can be determined by choices the user makes, so not all available pages are necessarily shown each time a wizard is presented.

When the standard wizard buttons are pressed, events are sent to the pages (and to the wxWizard object). You can catch events either in the page class or in a class derived from wxWizard.

To show a wizard, create an instance of wxWizard (or a derived class) and create the pages as children of the wizard. You can use wxWizardPageSimple (or a derived class) and chain the pages together with wxWizardPageSimple::Chain. Or, if you need to determine the path through the wizard dynamically according to user selections, you can derive from wxWizardPage and override GetPrev and GetNext. Add each page to the sizer returned by GetPageAreaSizer so that the wizard can adapt its size to the largest page.

wxWizard's only special window style is wxWIZARD_EX_HELPBUTTON, which adds a Help button to the wizard's row of standard buttons. This is an "extra" style, which must be set with SetExtraStyle before Create is called.

wxWizard **Events**

wxWizard generates wxWizardEvent events, which are described in Table 12-7. These events are sent first to the page, and if not processed, to the wizard itself. Except for EVT_WIZARD_FINISHED, event handlers can call wxWizard Event::GetPage to determine the currently active page.

Table 12-7. wxWizard Events

EVT_WIZARD_PAGE_CHANGED(id, func)	Use this event to detect when a page has been changed. The event handler function can call wxWizardEvent:: GetDirection (true if going forward).
EVT_WIZARD_PAGE_CHANGING(id, func)	Use to detect when a page is about to be changed (including when the Finish button was clicked); the event can be vetoed. The event handler function can call wxWizardEvent::GetDirection (true if going forward).
EVT_WIZARD_CANCEL(id, func)	Used to detect when the user has clicked the Cancel button; this can be vetoed.
EVT_WIZARD_HELP(id, func)	Use to show help when the user clicks on the Help button.
EVT_WIZARD_FINISHED(id, func)	Use to react to the user clicking on the Finish button. This event is generated just after the dialog has been closed.

wxHtmlWindow

wxHtmlWindow is used by wxWidgets' built-in help system, and it is also a great control to use in your applications whenever you need to display formatted text and graphics, such as reports. It can display a useful subset of HTML, including tables, but not frames. Features include animated GIFs, highlighted links, fonts, background color, nested lists, centering, right-alignment, horizontal rules, character encoding support, and more. It doesn't support style sheets, but you can normally achieve the effects you want by writing or generating the appropriate tags. HTML text is selectable and can be copied to the clipboard or returned to the application as plain text.

Figure 12-4 shows the wxHtmlWindow demo program that you can compile and run in samples/html/test.

Figure 12-4. The wxHtmlWindow demo program

[View full size image]

Because wxHtmlWindow is small and fast (unlike a full web browser), you can use it liberally in your application. Figure 12-5 shows an example of wxHtmlWindow in an "About" box.

Figure 12-5. wxHtmlWindow in an About box

wxGrid

wxGrid is a versatile and somewhat complex class for presenting information in a tabular form. You could make a property sheet out of a grid with name and value columns, create a general-purpose spreadsheet by adding your own formula evaluation code, show a table from a database, or display statistical data generated by your application. In some situations, you might consider wxGrid as an alternative to wxListCtrl in report mode, particularly if you need to display images or arbitrary graphics in cells.

A grid can display optional row and column headers, drawn in a similar way to the headers in a spreadsheet application. The user can drag column and row dividers, select one or more cells, and click a cell to edit it. Each cell in a grid has its own attributes for font, color, and alignment and also may have its own specialized renderer (for drawing the data) and editor (for editing the data). You can write your own renderers and editors: see include/wx/generic/ grid.h and src/generic/grid.cpp in your wxWidgets distribution for guidance. By default, a grid cell will use a simple string renderer and editor. If you have complex requirements for cell formatting, then rather than set attributes for each cell, you can create an "attribute provider" class that dynamically returns attributes for each cell when needed.

You can also create a potentially enormous "virtual" grid where storage is provided by the application, not by wxGrid. To do this, you derive a class from wxGridTableBase and override functions including GetValue, GetNumberRows, and GetNumberCols. These functions will reflect data stored in your application or perhaps in a database. Then plug the table into a grid using SetTable, and the grid will use your data. These more advanced techniques are demonstrated in samples/grid in your wxWidgets distribution, as shown in Figure 12-7.

Figure 12-7. wxGrid

[View full size image]

Listing 12-3 shows an example of creating a simple grid with eight rows and ten columns.

Listing 12-3. Simple Use of wxGrid

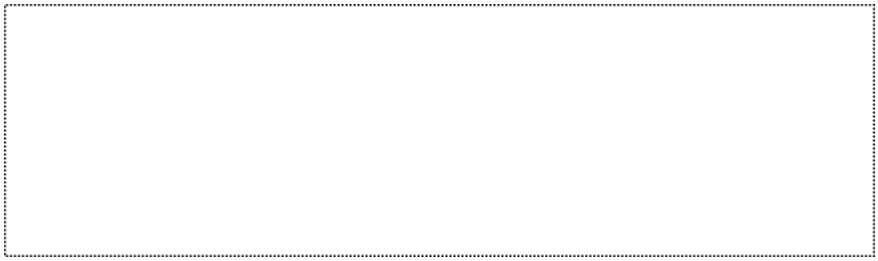
```
#include "wx/grid.h"

// Create a wxGrid object
wxGrid* grid = new wxGrid(frame, wxID ANY,
```

wxTaskBarIcon

This class installs an icon on the system tray (Windows, Gnome, or KDE) or dock (Mac OS X). Clicking on the icon will pop up a menu that the application supplies, and an optional tooltip can be shown when the mouse hovers over the icon. This technique gives quick access to important application functionality without having to use the regular application user interface. The application can display status information by switching icons, as per the battery and connection indicators in Windows.

Figure 12-8 shows the result of running the wxTaskBarIcon sample on Windows (see samples/taskbar). The wxWidgets icon is installed, and hovering the mouse pointer over the icon shows the tooltip text "wxTaskBarIconSample." Right-clicking on the icon shows the menu with three options. Selecting Set New Icon sets the icon to a smiley face and also resets the tooltip text to a new string.

Figure 12-8. wxTaskBarIcon on Windows



The implementation of a wxTaskBarIcon-derived class can be very simple, as Listing 12-4 shows (taken from the wxTaskBarIcon sample). The derived class MyTaskBarIcon overrides CreatePopupMenu and implements event handlers to intercept a left double-click and three menu commands.

Listing 12-4. Deriving from wxTaskBarIcon

```
class MyTaskBarIcon: public wxTaskBarIcon
{
public:
    MyTaskBarIcon() {};

    void OnLeftButtonDClick(wxTaskBarIconEvent&);
    void OnMenuRestore(wxCommandEvent&);
    void OnMenuExit(wxCommandEvent&);
    void OnMenuSetNewIcon(wxCommandEvent&);

    virtual wxMenu *CreatePopupMenu();

DECLARE_EVENT_TABLE()
};

enum {
    PU_RESTORE = 10001,
    PU_NEW_ICON,
    PU_EXIT,
};

BEGIN_EVENT_TABLE(MyTaskBarIcon, wxTaskBarIcon)
    EVT_MENU(PU_RESTORE, MyTaskBarIcon::OnMenuRestore)
    EVT_MENU(PU_EXIT,    MyTaskBarIcon::OnMenuExit)
    EVT_MENU(PU_NEW_ICON,MyTaskBarIcon::OnMenuSetNewIcon)
    EVT_TASKBAR_LEFT_DCLICK  (MyTaskBarIcon::OnLeftButtonDClick)
END_EVENT_TABLE()

void MyTaskBarIcon::OnMenuRestore(wxCommandEvent& )
{
```

Writing Your Own Controls

This section discusses how you can create your own controls in wxWidgets. wxWidgets does not have the concept of a "custom control" in the sense of a binary, drop-in component to which Windows programmers might be accustomed. Third-party controls are usually supplied as source code and follow the same pattern as generic controls within wxWidgets, such as wxCalendarCtrl and wxGrid. We're using the term "control" loosely here because controls do not have to be derived from wxControl; you might want to use wxScrolledWindow as a base class, for example.

Ten major tasks are involved in writing a new control:

1. Write a class declaration that has a default constructor, a constructor that creates the window, a Create function, and preferably an Init function to do shared initialization.
2. Add a function DoGetBestSize that returns the best minimal size appropriate to this control (based on a label size, for example).
3. Add a new event class for the control to generate, if existing event classes in wxWidgets are inadequate. A new type of push button might just use wxCommandEvent, but more complex controls will need a new event class. Also add event handler macros to use with the event class.
4. Write the code to display the information in the control.
5. Write the code to handle low-level mouse and keyboard events in the control, and generate appropriate high-level command events that the application can handle.
6. Write any default event handlers that the control might havefor example, handling wxID_COPY or wxID_UNDO commands or UI update commands.
7. Optionally, write a validator class that an application can use with the control (to make it easy to transfer data to and from the control) and validate its contents.
8. Optionally, write a resource handler class so that your control can be used with the XRC resource system.
9. Test the control on the platforms you want to support.
10. Write the documentation!

Let's take the simple example we used in Chapter 3, "Event Handling," when discussing custom events: wxFontSelectorCtrl, which you can find in examples/chap03 on the CD-ROM. This class shows a font preview on which the user can click to change the font using the standard font selector dialog. Changing the font causes a wxFontSelectorCtrlEvent to be sent, which can be caught by providing an event handler for EVT_FONT_SELECTION_CHANGED(id, func).

The control is illustrated in Figure 12-9 and is shown with a static text control above it.

Summary

This chapter has covered visual classes that you probably won't use in your first explorations of wxWidgets, but you'll almost certainly want to consider them as your applications become more sophisticated. Their source also gives you plenty of tips for learning how to write your own controls, as described in the final part of this chapter.

Refer also to Appendix D, "Other Features in wxWidgets," for other advanced controls distributed with wxWidgets, and Appendix E, "Third-Party Tools for wxWidgets," for third-party controls.

Next, we'll have a look at the data structure classes available in wxWidgets.

Chapter 13. Data Structure Classes

Storing and processing data is an essential part of any application. From simple classes that store information about size and position to complex types such as arrays and hash maps, wxWidgets provides a comprehensive selection of data structures. This chapter presents many of wxWidgets' data structures, highlighting the frequently used methods of each structure. Less frequently used structures and features can be found by reading the complete APIs in the wxWidgets documentation.

Note that data structure theories and implementations are not covered in this book. However, anyone should be able to use the data structure classes, even without understanding their internals.

Why Not STL?

First, let's deal with a question commonly asked about wxWidgets data structure classes: "Why doesn't wxWidgets just use the Standard Template Library (STL)?" The main reason is historical: wxWidgets has existed since 1992, long before STL could reliably be used across different platforms and compilers. As wxWidgets has evolved, many of the data structure classes have gravitated towards an STL-like API, and it is expected that eventually STL equivalents will replace some wxWidgets classes.

Meanwhile, you can still use STL functionality in your wxWidgets applications by setting wxUSE_STL to 1 in setup.h (or by passing enable-stl when configuring) to base wxString and other containers on the STL equivalents. Be warned that using STL with wxWidgets can increase both the library size and compilation time, especially when using GCC.

Strings

The benefits of working with a string class instead of standard character pointers are well established. wxWidgets includes its own string class, wxString, used both internally and for passing and returning information. wxString has all the standard operations you expect to find in a string class: dynamic memory management, construction from other strings, C strings, and characters, assignment operators, access to individual characters, string concatenation and comparison, substring extraction, case conversion, trimming and padding (with spaces), searching and replacing, C-like printf, stream-like insertion functions, and more.

Beyond being just another string class, wxString has other useful features. wxString fully supports Unicode, including methods for converting to and from ANSI and Unicode regardless of your build configuration. Using wxString gives you the ability to pass strings to the library or receive them back without any conversion process. Lastly, wxString implements 90% of the STL std::string methods, meaning that anyone familiar with std::string can use wxString without any learning curve.

Using wxString

Using wxString in your application is very straightforward. Wherever you would normally use std::string or your favorite string implementation, use wxString instead. All functions taking string arguments should take const wxString& (which makes assignment to the strings inside the function faster because of reference counting), and all functions returning strings should return wxString, which makes it safe to return local variables.

Because C and C++ programmers are familiar with most string methods, a long and detailed API reference for wxString has been omitted. Please consult the wxWidgets documentation for wxString, which provides a comprehensive list of all its methods.

You may notice that wxString sometimes has two or more functions that do the same thing. For example, Length, Len, and length all return the length of the string. In all cases of such duplication, the usage of std::string-compatible methods is strongly advised. It will make your code more familiar to other C++ programmers and will let you reuse the same code in both wxWidgets and other programs, where you can typedef wxString as std::string. Also, wxWidgets might start using std::string at some point in the future, so using these methods will make your programs more forward-compatible (although the wxString methods would be supported for some time for backwards compatibility).

wxString, Characters, and String Literals

wxWidgets has a wxChar type which maps either to char or wchar_t depending on the application build configuration (Unicode or ANSI). As already mentioned, there is no need for a separate type for char or wchar_t strings because wxString stores strings using the appropriate underlying C type. Whenever you work directly with strings that you intend to use with a wxWidgets class, use wxChar instead of char or wchar_t directly. Doing so ensures compatibility with both ANSI and Unicode build configuration without complicated preprocessor conditions.

When using wxWidgets with Unicode enabled, standard string literals are not the correct type: an unmodified string literal is always of type char*. In order for a string literal to be used in Unicode mode, it must be a wide character constant, usually marked with an L. wxWidgets provides the wxT macro (identical to _T) to wrap string literals for use with or without Unicode. When Unicode is not enabled, _T is an empty macro, but with Unicode enabled, it adds the necessary L for the string literal to become a wide character string constant. For example:

```
wxChar ch = wxT('*');
wxString s = wxT("Hello, world!");
wxChar* pChar = wxT("My string");
wxString s2 = pChar;
```

For more details about using Unicode in your applications, please see Chapter 16, "Writing International Applications."

wxArray

wxWidgets provides a dynamic array structure using wxArray, similar to C arrays in that the member access time is constant. However, these arrays are dynamic in the sense that they will automatically allocate more memory if there is not enough of it for adding a new element. Adding items to the arrays is also implemented in more or less constant timebut the price is pre-allocating the memory in advance. wxArray also provides range checking, asserting in debug builds or silently returning in release builds (though your program might get an unexpected value from array operations).

Array Types

wxWidgets has three different kinds of arrays. All derive from wxBaseArray, which works with untyped data and cannot be used directly. The macros WX_DEFINE_ARRAY, WX_DEFINE_SORTED_ARRAY, and WX_DEFINE_OBJARRAY are used to define a new class deriving from it. The classes are referred to as wxArray, wxSortedArray, and wxObjArray, but you should keep in mind that no classes with such names actually exist.

wxArray is suitable for storing integer types and pointers, which it does not treat as objects in any waythat is, the element referred to by the pointer is not deleted when the element is removed from the array. It should be noted that all of wxArray's functions are inline, so it costs nothing to define as many array types as you want (either in terms of the executable size or speed). This class has one serious limitation: it can only be used for storing integral types (bool, char, short, int, long, and their unsigned variants) or pointers (of any kind). Data of type float or double should not be stored in a wxArray.

wxSortedArray is a wxArray variant that should be used when you will be searching the array frequently. wxSortedArray requires you to define an additional function for comparing two elements of the array element type and always stores its items in the sorted order (according to the sort function). Assuming that you search the array far less frequently than you add to it, wxSortedArray may lead to huge performance improvements compared to wxArray. It should be noted that wxSortedArray shares wxArray's type restriction and should only be used for storing integral types or pointers.

wxObjArray class treats its elements like objects. It can delete them when they are removed from the array (invoking the correct destructor), and it copies them using the object's copy constructor. The definition of the wxObjArray arrays is split into two parts. First, you should declare the new wxObjArray class using the WX_DECLARE_OBJARRAY macro. Second, you must include the file defining the implementation of template type <wx/arrimpl.cpp> and define the array class with the WX_DEFINE_OBJARRAY macro from a point where the full declaration of the array elements class is in scope. This technique will be demonstrated in the array sample code presented later in this chapter.

wxArrayString

wxArrayString is an efficient container for storing wxString objects and has the same features as the other wxArray classes. It is also very compact and doesn't take more space than a C array wxString[] type (wxArrayString uses its knowledge of internals of wxString class to achieve this). All of the methods available in the other wxArray types are also available in wxArrayString.

This class is used in the same way as other dynamic arrays, except that no WX_DEFINE_ARRAY declaration is needed for ityou can use wxArrayString directly. When a string is added or inserted in the array, a copy of the string is created, so the original string may be safely deleted. In general, there is no need to worry about string memory management when using this classit will always free the memory it uses.

The references returned by Item, Last, or operator[] are not constant, so the array elements may be modified in place:

```
array.Last().MakeUpper();
```

wxList **and** wxNode

The wxList class is a doubly linked list that can store data of an arbitrary type. wxWidgets requires that you explicitly define a new list type for each type of list data, providing strong type checking for the list's data type. The wxList class also allows you to optionally specify a key type for primitive lookups (see the wxHashMap section if you need a structure with fast random access).

The wxList class makes use of an abstract wxNode class. When you define a new list, a new node type deriving from wxNodeBase is also created, providing type-safe node operations. The most important methods of the node class are the self-explanatory GetNext, GetPrevious, and GeTData, which provide access to the next node, the previous node, and the current node's data.

The only remarkable operation for a wxList is data deletion. By default, removing a node does not delete the data being stored by that node. The DeleteContents method allows you to change this behavior and set the data itself to be deleted along with the nodes. If you want to empty a list of all data and delete the data, be sure to call DeleteContents with TRue before calling Clear.

Rather than rehash the contents of the manual, a small but comprehensive code example shows the wxList methods as well as how to create and use your custom list type. Note that the WX_DECLARE_LIST macro would typically appear in a header file, while the WX_DEFINE_LIST macro would almost always appear in a source file.

```
// Our data class to store in the list
class Customer
{
public:
    int CustID;
    wxString CustName;
};

// this part might be in a header or source file
// declare our list class:
// this macro declares and partly implements CustomerList class
// (which derives from wxListBase)
WX_DECLARE_LIST(Customer, CustomerList);

// the only requirement for the rest is to be AFTER the full
// declaration of Customer (for WX_DECLARE_LIST forward declaration
// is enough), but usually it will be found in the source file and
// not in the header
#include <wx/listimpl.cpp>
WX_DEFINE_LIST(CustomerList);

// Used for sorting to compare objects
int listcompare(const Customer** arg1, const Customer** arg2)
{
    return ((*arg1)->CustID < (*arg2)->CustID);
}

// Show List operations
void ListTest()
{
    // Declare an instance of our list
    CustomerList* MyList = new CustomerList();

    bool IsEmpty = MyList->IsEmpty(); // will be true

    // Create some customers
    Customer* CustA = new Customer;
    CustA->CustID = 10;
    CustA->CustName = wxT("Bob");

    Customer* CustB = new Customer;
```

wxHashMap

The wxHashMap class is a simple, type-safe, and reasonably efficient hash map class, with an interface that is a subset of the interface of STL containers. In particular, the interface is modeled after std::map and the non-standard std::hash_map. By using macros to create hash maps, you can choose from several combinations of keys and values, including int, wxString, or void* (arbitrary class).

There are three macros for declaring a hash map. To declare a hash map class named CLASSNAME with wxString keys and VALUE_T values:

```
WX_DECLARE_STRING_HASH_MAP(VALUE_T, CLASSNAME);
```

To declare a hash map class named CLASSNAME with void* keys and VALUE_T values:

```
WX_DECLARE_VOIDPTR_HASH_MAP(VALUE_T, CLASSNAME);
```

To declare a hash map class named CLASSNAME with arbitrary keys or values:

```
WX_DECLARE_HASH_MAP(KEY_T, VALUE_T, HASH_T, KEY_EQ_T, CLASSNAME);
```

HASH_T and KEY_EQ_T are the types used for the hashing function and key comparison. wxWidgets provides three predefined hashing functions: wxInteger Hash for integer types (int, long, short, and their unsigned counterparts), wxStringHash for strings (wxString, wxChar*, char*), and wxPointerHash for any kind of pointer. Similarly, three equality predicates are provided: wxInteger Equal, wxStringEqual, and wxPointerEqual.

The following code example shows the wxHashMap methods as well as how to create and use your custom hash type.

```
// Our data class to store in the hash
class Customer
{
    public:
        int CustID;
        wxString CustName;
};

// Declare our hash map class
// This macro declares and implements CustomerList as a hash map
WX_DECLARE_HASH_MAP(int, Customer*, wxIntegerHash,
                    wxIntegerEqual, CustomerHash);

void HashTest()
{
    // Declare an instance of our list
    CustomerHash MyHash;

    bool IsEmpty = MyHash.empty(); // will be true

    // Create some customers
    Customer* CustA = new Customer;
    CustA->CustID = 10;
    CustA->CustName = wxT("Bob");

    Customer* CustB = new Customer;
    CustB->CustID = 20;
    CustB->CustName = wxT("Sally");

    Customer* CustC = new Customer;
    CustC->CustID = 5;
```

Storing and Processing Dates and Times

wxWidgets provides a comprehensive wxDateTime class for representing date and time information with many operations such as formatting, time zones, date and time arithmetic, and more. Static functions provide information such as the current date and time, as well as queries such as whether a given year is a leap year. Note that the wxDateTime class is the appropriate class to use even when you need to store only date or time information. Helper classes wxTimeSpan and wxDateSpan provide convenient ways for modifying an existing wxDateTime object.

wxDateTime

The wxDateTime class has too many methods to include in a concise discussion; the complete API reference is available in the wxWidgets documentation. What is presented here is an overview of the most frequently used wxDateTime methods and operations.

Note that although time is always stored internally in Greenwich Mean Time (GMT), you will usually work in the local time zone. Because of this, all wxDateTime constructors and modifiers that compose a date or time from components (for example hours, minutes, and seconds) assume that these values are for the local time zone. All methods returning date or time components (month, day, hour, minute, second, and so on) will also return the correct values for the local time zone by default; no effort is required to get correct results for your time zone. If you want to manipulate time zones, please refer to the documentation.

wxDateTime **Constructors and Modifiers**

wxDateTime objects can be constructed from Unix timestamps, time-only information, date-only information, or complete date and time information. For each constructor, there is a corresponding Set method that modifies an existing object to have the specified date or time. There are also individual modifiers such as SetMonth or SetHour that change just one component of the date or time.

wxDateTime(time_t) constructs an object with the date and time set according to the specified Unix timestamp.

wxDateTime(const struct tm&) constructs an object using the data from the C standard tm structure.

wxDateTime(wxDateTime_t hour, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0) constructs an object based on the specified time information.

wxDateTime(wxDateTime_t day, Month month = Inv_Month, int year = Inv_Year, wxDateTime_t hour = 0, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0) constructs an object with the specified date and time information.

wxDateTime **Accessors**

The accessors for wxDateTime are mostly self-explanatory: GetYear, GetMonth, Getday, GetWeekDay, GetHour, GetMinute, GetSecond, GetMillisecond, GeTDayOfYear, GetWeekOfYear, GetWeekOfMonth, and GetYearDay. wxDateTime also provides the following:

- GetTicks returns the date and time in Unix timestamp format (seconds since January 1, 1970 at midnight).
- IsValid returns whether or not the object is in a valid state (the object could have been constructed but never given a date or time).

Getting the Current Time

wxDateTime provides two static methods for retrieving the current time:

Helper Data Structures

wxWidgets makes use of several data structures internally and as parameters and return types in public library methods. Application programmers are encouraged to use the wxWidgets helper data structures in their projects.

wxObject

The wxObject class is the base class of all wxWidgets classes, providing run-time type information, reference counting, virtual destructor declaration, and optional debugging versions of new and delete. The wxClassInfo class is used to store meta-data about classes and is used by some of the wxObject methods.

```
MyWindow* window = wxDynamicCast(FindWindow(ID_MYWINDOW), MyWindow);
```

IsKindOf takes a wxClassInfo pointer and returns TRue if the object is of the specified type. For example:

```
bool tmp = obj->IsKindOf(CLASSINFO(wxFrame));
```

Ref takes a const wxObject& and replaces the current object's data with a reference to the passed object's data. The reference count of the current object is decremented, possibly freeing its data, and the reference count of the passed object is incremented.

UnRef decrements the reference count of the associated data and deletes the data if the reference count has fallen to 0.

wxLongLong

The wxLongLong class represents a 64-bit long number. A native 64-bit type is always used when available, and emulation code is used when the native type is unavailable. You would usually use this type in exactly the same manner as any other (built-in) arithmetic type. Note that wxLongLong is a signed type; if you want unsigned values, use wxULongLong, which has exactly the same API as wxLongLong except for some logical exceptions (such as the absolute value method). All of the usual mathematical operations are defined, as well as several convenient accessors:

- Abs returns the absolute value of the value as a wxLongLong, either as a copy if used on a constant reference or modifying it in place if mutable.
- ToLong returns a long representation of the stored value, triggering a debug assertion if any precision was lost.
- ToString returns the string representation of the stored value in a wxString.

wxPoint **and** wxRealPoint

wxPoint is used throughout wxWidgets for specifying integer screen or window locations. As the names imply, the point classes store coordinate pairs as x and y values. The data members are declared as public and can be accessed directly as x and y. wxPoint provides + and operators that allow you to add or subtract by wxPoint or wxSize. wxRealPoint stores coordinates as double rather than int and provides + and operators accepting only other wxRealPoint objects.

Constructing a wxPoint is very straightforward:

```
wxPoint myPoint(50, 60);
```

Summary

The data structures provided by wxWidgets allow you to easily pass and receive structured data to and from wxWidgets and within your own applications. By providing powerful data processing methods and classes such as wxRegEx, wxStringTokenizer, wxDateTime, and wxVariant, almost any data storage and processing needs can be met by wxWidgets without having to use third-party libraries.

Next, we'll look at what wxWidgets offers for reading and writing data using files and streams.

Chapter 14. Files and Streams

In this chapter, we'll look at the classes that wxWidgets provides for low-level file access and streaming. wxWidgets' stream classes not only protect your application from the idiosyncrasies of different standard C++ libraries but also provide a complete set of classes including compression, writing to zip archives, and even socket streaming. We'll also describe wxWidgets' virtual file system facility, which lets your application easily take data from sources other than normal disk files.

File Classes and Functions

wxWidgets provides a range of functionality for platform-independent file handling. We'll introduce the major classes before summarizing the file functions.

wxFile and wxFFile

wxFile may be used for low-level input/output. It contains all the usual functions to work with integer file descriptors (opening/closing, reading/writing, seeking, and so on), but unlike the standard C functions, it reports errors via wxLog and closes the file automatically in the destructor. wxFFile brings similar benefits but uses buffered input/output and contains a pointer to a FILE handle.

You can create a wxFile object by using the default constructor followed by Create or Open, or you can use the constructor that takes a file name and open mode (wxFile::read, wxFile::write, or wxFile::read_write). You can also create a wxFile from an existing file descriptor, passed to the constructor or Attach function. A wxFile can be closed with Close, which is called automatically (if necessary) when the object is destroyed.

You can get data from the object with Read, passing a void* buffer and the number of bytes to read. Read will return the actual number of bytes read, or wxInvalidOffset if there was an error. Use Write to write a void* buffer or wxString to the file, calling Flush if you need the data to be written to the file immediately.

To test for the end of the file, use Eof, which will return TRue if the file pointer is at the end of the file. (In contrast, wxFFile's Eof will return TRue only if an attempt has been made to read past the end of the file.) You can determine the length of the file with Length.

Seek and SeekEnd seek to a position in the file, taking an offset specified as starting from the beginning or end of the file, respectively. Tell returns the current offset from the start of the file as a wxFileOffset (a 64-bit integer if the platform supports it, or else a 32-bit integer).

Call the static Access function to determine whether a file can be opened in a given mode. Exists is another static function that tests the existence of the given file.

The following code fragment uses wxFile to open a data file and read all the data into an array.

```
#include "wx/file.h"

if (!wxFile::Exists(wxT("data.dat")))
    return false;

wxFile file(wxT("data.dat"));

if ( !file.IsOpened() )
    return false;

// get the file size
wxFileOffset nSize = file.Length();
if ( nSize == wxInvalidOffset )
    return false;

// read the whole file into memory
wxUint* data = new wxUint8[nSize];

if ( fileMsg.Read(data, (size_t) nSize) != nSize )
{
    delete[] data;
    return false;
}

file.Close();
```

Stream Classes

Streams offer a higher-level model of reading and writing data than files; you can write code that doesn't care whether the stream uses a file, memory, or even sockets (see Chapter 18, "Programming with wxSocket," for an example of using sockets with streams). Some wxWidgets classes that support file input/output also support stream input/output, such as wxImage.

wxStreamBase is the base class for streams, declaring functions such as OnSysRead and OnSysWrite to be implemented by derived classes. The derived classes wxInputStream and wxOutputStream provide the foundation for further classes for reading and writing, respectively, such as wxFileInputStream and wxFileOutputStream. Let's look at the stream classes provided by wxWidgets.

File Streams

wxFileInputStream and wxFileOutputStream are based on the wxFile class and can be initialized from a file name, a wxFile object, or an integer file descriptor. Here's an example of using wxFileInputStream to read in some data, seek to the beginning of the stream, and retrieve the current position.

```
#include "wx/wfstream.h"

// The constructor initializes the stream buffer and opens the
// file descriptor associated with the name of the file.
// wxFileInputStream will close the file descriptor on destruction.
wxFileInputStream inStream(filename);

// Read some bytes
int byteCount = 100;
char data[100];

if (inStream.Read((void*) data, byteCount).
             LastError() != wxSTREAM_NOERROR)
{
    // Something bad happened.
    // For a complete list, see the wxStreamBase documentation.
}

// You can also get the last number of bytes really read.
size_t reallyRead = inStream.LastRead();

// Moves to the beginning of the stream. SeekI returns the last position
// in the stream counted from the beginning.
off_t oldPosition = inStream.SeekI(0, wxFromBeginning);

// What is my current position?
off_t position = inStream.TellI();
```

Using wxFileOutputStream is similarly straightforward. The following code uses a wxFileInputStream and wxFileOutputStream to make a copy of a file, writing in 1024-byte chunks for efficiency. Error checking has been omitted for the sake of brevity.

```
// Copy a fixed size of data from an input stream to an
// output stream, using a buffer to speed it up.
void BufferedCopy(wxInputStream& inStream, wxOutputStream& outStream,
                  size_t size)
{
    static unsigned char buf[1024];
    size_t bytesLeft = size;

    while (bytesLeft > 0)
    {
        size_t bytesToRead = wxMin((size_t) sizeof(buf), bytesLeft);
```

Summary

This chapter has given an overview of the classes that wxWidgets offers to let your application work with files and streams in a portable way. We've also touched on the virtual file system facility that makes it easy to get data from compressed archives, data in memory, and Internet files.

Next, we'll tackle issues that are not directly related to what your users see on their screens but are nevertheless crucial: memory management, debugging, and error checking.

Chapter 15. Memory Management, Debugging, and Error Checking

Tracking down errors is an essential, if unglamorous, part of developing an application. This chapter describes the facilities that wxWidgets provides to detect memory problems and also to encourage "defensive programming"checking for problems as early as possiblewhich makes for much more reliable and easily debugged software. We also explain when you should create objects on the heap and when to create them on the stack, and we discuss how to use the run-time type information facilities, the module mechanism, and wxWidgets C++ exception support. We finish with some general debugging tips.

Memory Management Basics

As in all C++ programming, you will create objects either on the stack, or on the heap using new. An object created on the stack is only available until it goes out of scope, at which time its destructor is called and it no longer exists. An object created on the heap, on the other hand, will stay around until either it is explicitly deleted using the delete operator or the program exits.

Creating and Deleting Window Objects

As a general rule, you will create window objects such as wxFrame and wxButton on the heap using new. Window objects normally have to exist for an indeterminate amount of timethat is, until the user decides the window will be closed. Note that wxWidgets will destroy child objects automatically when the parent is destroyed. Thus, you don't have to destroy a dialog's controls explicitly: just delete the dialog with Destroy. Similarly, upon deletion, a frame automatically deletes any children contained within it. However, if you create a top-level window (such as a frame) as a child of another top-level window (such as another frame), the parent frame does not destroy the child frame. An exception to this is MDI (Multiple Document Interface), where the child frames are not independent windows and are therefore destroyed by the parent.

You can create dialogs on the stack, but they must be modal dialogs: call ShowModal to enter an event loop so that all required interaction can happen before the dialog object goes out of scope and is deleted.

The mechanics of closing and deleting frames and dialogs can be a source of confusion. To destroy a frame or modeless dialog, the application should use Destroy, which delays deletion until the event queue is empty to avoid events being sent to non-existent windows. However, for a modal dialog, EndModal should first be called to exit the event loop. Event handlers (for example, for an OK button) should not normally destroy the dialog because if the modal dialog is created on the stack, it will be destroyed twice: once by the event handler, and again when the dialog object goes out of scope. When the user closes a modal dialog, the wxEVT_CLOSE_WINDOW event is triggered, and the corresponding event handler should call EndModal (but should not destroy the dialog). The default "close" behavior when clicking on the close button in the title bar is to emulate a wxID_CANCEL command event, whose handler will normally close the dialog. The dialog is then deleted when it goes out of scope. This is how standard dialogs such as wxFileDialog and wxColourDialog work, allowing you to retrieve values from the dialog object when the event loop returns. You can design modal dialogs that destroy the dialog from event handlers, but then you will not be able to create such a dialog on the stack or retrieve values from the dialog object when the user has dismissed it.

Here are two ways of using a wxMessageDialog.

```
// 1) Creating the dialog on the stack: no explicit destruction
wxMessageDialog dialog(NULL, _("Press OK"), _("App"), wxOK|wxCANCEL);
if (dialog.ShowModal() == wxID_OK)
{
    // 2) Creating the dialog on the heap: must delete with Destroy()
    wxMessageDialog* dialog = new wxMessageDialog(NULL,
        _("Thank you! "), _("App"), wxOK);
    dialog->ShowModal();
    dialog->Destroy();
}
```

Modeless dialogs and frames will usually need to destroy themselves when closed, either from a control or from the standard window close button or menu. They cannot be created on the stack because they would immediately go out of scope and be destroyed.

If you maintain pointers to windows, be sure to reset them to NULL when the corresponding windows have been destroyed. Code to reset the pointer can be written in the window destructor or close handler. For example:

```
void MyFindReplaceDialog::OnCloseWindow(wxCloseEvent& event)
{
    wxGetApp().SetFindReplaceDialog(NULL);
```

Detecting Memory Leaks and Other Errors

Ideally, when your application exits, all objects will be cleaned up either by your application or by wxWidgets itself, and no allocated memory will be left for the operating system to clean up automatically. Although it may be tempting simply not to bother with some cleanup, you really should take care to clean everything up yourself. Often such memory leaks are symptomatic of a problem with your code that could lead to large amounts of memory being wasted during a session. It's much harder to go back and figure out where leaks are coming from after you have moved on to another aspect of your application, so try to have zero tolerance for leaks.

So how do you know whether your application is leaking memory? Various third-party memory-checking tools are available to do this and more, and wxWidgets has a simple built-in memory checker. To use this checker for your debug configuration, you need to set some switches in setup.h (Windows) or configure (other platforms or GCC on Windows).

On Windows, these are:

```
#define wxUSE_DEBUG_CONTEXT 1
#define wxUSE_MEMORY_TRACING 1
#define wxUSE_GLOBAL_MEMORY_OPERATORS 1
#define wxUSE_DEBUG_NEW_ALWAYS 1
```

For configure, pass these switches:

```
--enable-debug --enable-mem_tracing --enable-debug_cntxt
```

There are some restrictions to this system: it doesn't work for MinGW or Cygwin (at the time of writing), and you cannot use wxUSE_DEBUG_NEW_ALWAYS if you are using STL in your application or the CodeWarrior compiler.

If wxUSE_DEBUG_NEW_ALWAYS is on, then all instances of the new operator in wxWidgets and your code will be defined to be new(__TFILE__,__LINE__), which has been reimplemented to use custom memory allocation and deletion routines. To use this version of new explicitly, without defining new, use WXDEBUG_NEW where you would normally write new.

The easiest way to use the memory checking system is to do nothing special at all: just run your application in the debugger, quit the application, and see if any memory leaks are reported. Here's an example of a report:

```
There were memory leaks.

- Memory dump -
.\memcheck.cpp(89): wxBrush at 0xBE44B8, size 12
..\..\src\msw\brush.cpp(233): non-object data at 0xBE55A8, size 44
.\memcheck.cpp(90): wxBitmap at 0xBE5088, size 12
..\..\src\msw\bitmap.cpp(524): non-object data at 0xBE6FB8, size 52
.\memcheck.cpp(93): non-object data at 0xBB8410, size 1000
.\memcheck.cpp(95): non-object data at 0xBE6F58, size 4
.\memcheck.cpp(98): non-object data at 0xBE6EF8, size 8

- Memory statistics -
1 objects of class wxBitmap, total size 12
5 objects of class nonobject, total size 1108
1 objects of class wxBrush, total size 12

Number of object items: 2
Number of non-object items: 5
Total allocated size: 1132
```

This example tells us that a wxBrush and a wxBitmap were allocated but not freed, along with some other objects.

Facilities for Defensive Programming

Often a bug will only surface some time after the logic error actually occurs. If an inconsistent or invalid value isn't detected early, the program can execute thousands of lines of code before it crashes or you get mysterious results. You can spend a lot of time trying to figure out the real cause of the error. However, if you add regular checks to your codefor example, to detect bad values passed to functionsyour code will end up being much more robust, and you will save yourself (and your users) a potentially huge amount of trouble. This technique is called defensive programming, and your classes and functions can defend themselves both against improper use by other code and against internal logic errors. Because most of these checks are compiled out in release builds, there is no overhead.

As you would expect, wxWidgets does a lot of error checking internally, and you can use the same macros and functions in your own code. There are three main families of macros: variations of wxASSERT, wxFAIL, and wxCHECK. wxASSERT macros show an error message if the argument doesn't evaluate to TRue; the checks only occur in debug builds. wxFAIL will always generate an error message and is equivalent to wxASSERT(false). These checks are also removed in the release build. wxCHECK checks that the condition is TRue and returns a given value if not. Unlike the others, occurrences of wxCHECK remain (but do not display a message) in release builds. These macros have variations that let you display a custom error message in the assertion message box.

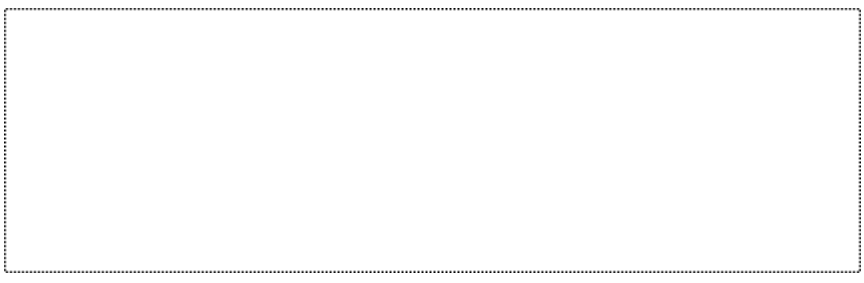
Here are some examples of using these macros.

```
// Add two positive numbers under 100
int AddPositive(int a, int b)
{
    // Check if a is positive
    wxASSERT(a > 0);

    // Check if b is positive, with custom error message
    wxASSERT_MSG(b > 0, wxT("The second number must be positive!"));

    int c = a + b;
    // Return -1 if the result was not positive
    wxCHECK_MSG(c > 0, -1, wxT("Result must be positive!"));

    return c;
}
```

You can also use wxCHECK2 and wxCHECK2_MSG to execute an arbitrary operation if the check failed instead of just returning a value. wxCHECK_RET can be used in void functions and returns no value. For less frequently used macros, such as wxCOMPILE_TIME_ASSERT and wxASSERT_MIN_BITSIZE, please refer to the reference manual.

Figure 15-1 shows the assertion dialog that appears when an assertion condition has evaluated to false. The user can stop the program (Yes), ignore the warning (No), or ignore all further warnings (Cancel). If the program is running under a debugger, stopping the program will cause a break in the debugger, and you can navigate the call stack to see exactly where the problem was and the values of variables at that point.

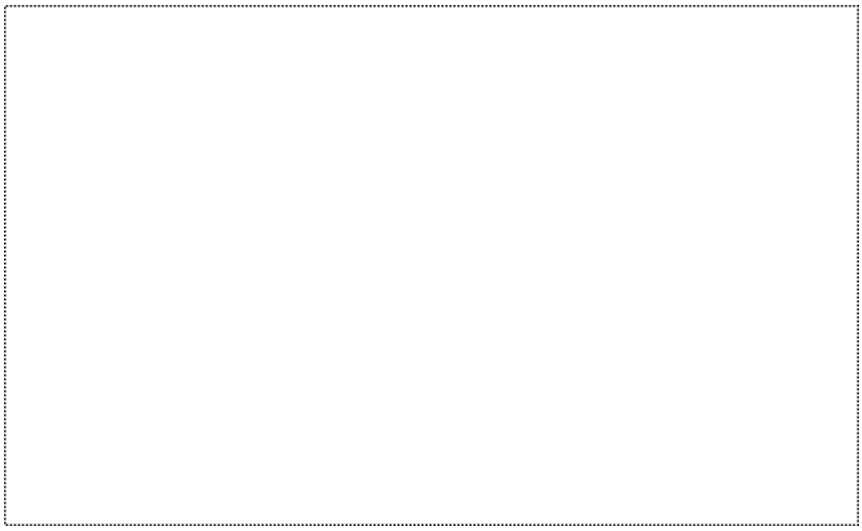
Figure 15-1. Assertion alert



Error Reporting

Sometimes you need to display messages on the console or in a dialog to help with debugging, or when an error occurs that would not normally be detected by application code. wxWidgets provides a range of logging functions for the different ways in which you might want to use logging. For example, if allocation fails when trying to create a very large wxBitmap, an error will be reported in a dialog box using wxLogError (see Figure 15-2). Or if you simply want to print the values of a function's arguments on the debugger window, you can use wxLogDebug. Where the error actually appears (in a dialog, on the debugger window, or on the standard error stream) depends on the name of the logging function used and also on the wxLog "target" that is currently active, as described later.

Figure 15-2. Log dialog



All logging functions have the same syntax as printf or vprintfthat is, they take the format string as the first argument and a variable number of arguments or a variable argument list pointer. For example:

```
wxString name(wxT("Calculation"));
int nGoes = 3;

wxLogError(wxT("%s does not compute! You have %d more goes."),
           name.c_str(), nGoes);
```

The logging functions are as follows.

wxLogError is the function to use for error messages that must be shown to the user. The default behavior is to pop up a message box to inform the user about the error. Why not just use wxMessageBox? Well, messages that come from wxLogError can be suppressed by creating an object of wxLogNull and log errors are also queued up and shown in idle time in a single dialog. So if a series of errors happens, the user won't have to click on OK for each and every error.

wxLogFatalError is like wxLogError, but it also terminates the program with exit code 3 (using the standard abort function). Unlike all the other logging functions, this function can't be overridden by changing the log target.

wxLogWarning is like wxLogError, but the information is presented as a warning, not an error.

wxLogMessage is for all normal, informational messages. They also appear in a message box by default.

wxLogVerbose is for verbose output. Normally, the messages are suppressed, but they can be activated by calling wxLog::SetVerbose if the user wants to know more details about the program's progress.

Providing Run-Time Type Information

In common with most frameworks, wxWidgets provides methods to define more RTTI than C++'s own RTTI. This is useful for making run-time decisions based on an object's type and for error reporting as we have seen in the last section, and it also enables you to create objects dynamically simply by providing a string containing the name of the class. Only classes that have wxObject as an ancestor class can have wxWidgets RTTI.

If you don't need the dynamic creation ability, use the macro DECLARE_CLASS(class) in the class declaration and IMPLEMENT_CLASS(class, base Class) in the class implementation file. If you need dynamic creation, use DECLARE_DYNAMIC_CLASS(class) in the class declaration and IMPLEMENT_DYNAMIC_ CLASS(class, baseClass) in the class implementation file. In the dynamic case, you will also need to make sure there is a default constructor for the class; otherwise the compiler will complain when it comes across the function that wxWidgets generates to create an object of this class.

Here's an example of using RTTI to allow dynamic creation of objects.

```
class MyRecord: public wxObject
{
DECLARE_DYNAMIC_CLASS(MyRecord)
public:
    MyRecord() {}
    MyRecord(const wxString& name) { m_name = name; }

    void SetName(const wxString& name) { m_name = name; }
    const wxString& GetName() const { return m_name; }
private:
    wxString m_name;
};

IMPLEMENT_DYNAMIC_CLASS(MyRecord, wxObject)

MyRecord* CreateMyRecord(const wxString& name)
{
    MyRecord* rec = wxDynamicCast(wxCreateDynamicObject(wxT("MyRecord")), MyRecord);
    if (rec)
        rec->SetName(name);
    return rec;
}
```

When code calls CreateMyRecord with the name to be set, wxCreateDynamicObject creates the object, and wxDynamicCast confirms that it really is an object of type MyRecordit will return NULL if not. Although it might not appear useful at first sight, dynamic object creation is very handy when loading a complex file containing objects of different types. An object's data can be stored along with the name of its class, and when the file is read back, a new instance of the class can be created and then the object can read in its data.

There are other RTTI macros you can use, as follows.

CLASSINFO(class) returns a pointer to the wxClassInfo object associated with a class. You can use it with wxObject::IsKindOf to test the type of a class:

```
if (obj->IsKindOf(CLASSINFO(MyRecord)))
{
    ...
}
```

Use DECLARE_ABSTRACT_CLASS(class) and IMPLEMENT_ABSTRACT_CLASS(class, baseClass) with abstract classes.

Use DECLARE_CLASS2(class) and IMPLEMENT_CLASS2(class, baseClass1, baseClass2) where there are

Using wxModule

The module system is a very simple mechanism to allow applications (and parts of wxWidgets itself) to define initialization and cleanup functions that are automatically called on wxWidgets startup and exit. It can save an application from having to call a lot of initialization and cleanup code in its OnInit and OnExit functions, depending on the features that it uses.

To define a new kind of module, derive a class from wxModule, override the OnInit and OnExit functions, and add the DECLARE_DYNAMIC_CLASS and IMPLEMENT_DYNAMIC_CLASS to the class and implementation (which can be in the same file). On initialization, wxWidgets will find all classes derived from wxModule, create an instance of each, and call each OnInit function. On exit, wxWidgets will call the OnExit function for each module instance.

For example:

```
// A module to allow DDE initialization/cleanup
class wxDDEModule: public wxModule
{
DECLARE_DYNAMIC_CLASS(wxDDEModule)
public:
    wxDDEModule() {}
    bool OnInit() { wxDDEInitialize(); return true; };
    void OnExit() { wxDDECleanUp(); };
};

IMPLEMENT_DYNAMIC_CLASS(wxDDEModule, wxModule)
```

Loading Dynamic Libraries

If you need to run functions in dynamic libraries, you can use the wxDynamicLibrary class. Pass the name of the dynamic library to the constructor or Load, and pass wxDL_VERBATIM if you don't want wxWidgets to append an appropriate extension, such as .dll on Windows or .so on Linux. If the library was loaded successfully, you can load functions by name using GetSymbol. Here's an example that loads and initializes the common controls library on Windows:

```
#include "wx/dynlib.h"

INITCOMMONCONTROLSEX icex;
icex.dwSize = sizeof(icex);
icex.dwICC = ICC_DATE_CLASSES;

// Load comctl32.dll
wxDynamicLibrary dllComCtl32(wxT("comctl32.dll"), wxDL_VERBATIM);

// Define the ICCEx_t type
typedef BOOL (WINAPI *ICCEx_t)(INITCOMMONCONTROLSEX *);

// Get the InitCommonControlsEx symbol
ICCEx_t pfnInitCommonControlsEx =
    (ICCEx_t) dllComCtrl32.GetSymbol(wxT("InitCommonControlsEx"));

// Call the function to initialize the common controls library
if ( pfnInitCommonControlsEx )
{
    (*pfnInitCommonControlsEx)(&icex);
}
```

You could also write the GetSymbol line more succinctly using the wxDYNLIB_FUNCTION macro:

```
wxDYNLIB_FUNCTION( ICCEx_t, InitCommonControlsEx, dllComCtl32 );
```

wxDYNLIB_FUNCTION allows you to specify the type only once, as the first parameter, and creates a variable of this type named after the function but with a pfn prefix.

If the library was loaded successfully in the constructor or Load, the function Unload will be called automatically in the destructor to unload the library from memory. Call Detach if you want to keep a handle to the library after the wxDynamicLibrary object has been destroyed.

Exception Handling

wxWidgets was created long before exceptions were introduced in C++ and has had to work with compilers with varying levels of exception support, so exceptions are not used throughout the framework. However, it safe to use exceptions in application code, and the library tries to help you.

There are several choices for using exceptions in wxWidgets programs. First, you can avoid using them at all. The library doesn't throw any exceptions by itself, so you don't have to worry about exceptions at all unless your own code throws them. This is the simplest solution, but it may be not the best one to deal with all possible errors.

Another strategy is to use exceptions only to signal truly fatal errors. In this case, you probably don't expect to recover from them, and the default behaviorto simply terminate the programmay be appropriate. If it is not, you can override OnUnhandledException in your wxApp-derived class to perform any cleanup tasks. Note that any information about the exact exception type is lost when this function is called, so if you need this information, you should override OnRun and add a TRy/catch clause around the call of the base class version. This would enable you to catch any exceptions generated during the execution of the main event loop. To deal with exceptions that may arise during the program startup and shutdown, you should insert TRy/catch clauses in OnInit and OnExit.

Finally, you might also want the application to continue running even when certain exceptions occur. If all your exceptions can happen only in the event handlers of a single class (or only in the classes derived from it), you can centralize your exception handling code in the ProcessEvent method of this class. If this is impractical, you might also consider overriding the wxApp::HandleEvent, which allows you to handle all the exceptions thrown by any event handler.

To enable exception support in wxWidgets, you need to build it with wxUSE_EXCEPTIONS set to 1. This should be the case by default, but if it isn't, you should edit include/wx/msw/setup.h under Windows or run configure with --enable-exceptions under Unix. If you do not plan to use exceptions, setting this flag to 0 or using --disable-exceptions results in a leaner and slightly faster library. Also, if you have Visual C++ and want a user-defined wxApp::OnFatalException function to be called instead of a GPF occurring, set wxUSE_ON_FATAL_EXCEPTION to 1 in your setup.h. Conversely, if you would rather be dropped into the debugger when an error in your program occurs, set this to 0.

Please look at samples/except for examples of using exceptions with wxWidgets.

Debugging Tips

Defensive programming, error reporting, and other coding techniques can only go so faryou also need a debugger that lets you step through your code examining variables and tells you exactly where your program is misbehaving or has crashed. So, you will need to maintain at least two configurations of your applicationa debug version and a release version. The debug version will contain more error checking, will have compiler optimizations switched off, and will contain the source file, line, and other debug information that the debugger needs. The preprocessor symbol __WXDEBUG__ will always be defined in debug mode, and you can test for this when you need to write debug-only code. Some functions, such as wxLogDebug, will be removed in release mode anyway, reducing the need to test for this symbol.

A surprising number of users will try to get away without using a debugger, but the effort expended in getting to know your tools will pay off. On Windows, Visual C++ comes with a very good debugger; if using GCC on Windows or Unix, you can use the basic GDB package (from a command line or editor), and there is a selection of IDEs that use GDB but present a more friendly GUI to the debugger. For information on these, see Appendix E, "Third-Party Tools for wxWidgets."

wxWidgets allows multiple configurations to be used simultaneously. On Windows, you can pass BUILD=debug or BUILD=release to the wxWidgets library makefile, or when using configure, you configure and build in two or more separate directories, passing --enable-debug or --disable-debug. Some IDEs don't allow multiple configurations of your application to be maintained simultaneously without changing settings and recompiling in debug mode and then changing the settings back and recompiling in release modefor obvious reasons, avoid such tools!

Debugging X11 Errors

Rarely, your wxGTK application might crash with X11 errors, and the program will immediately exit without giving you a stack trace. This makes it very hard to find where the error occurred. In this case, you need to set an error handler, as the following code shows.
[View full width]

```
#if defined(__WXGTK__)
#include <X11/Xlib.h>

typedef int (*XErrorHandlerFunc)(Display *, XErrorEvent *);

XErrorHandlerFunc gs_pfnXErrorHandler = 0;

int wxXErrorHandler(Display *display, XErrorEvent *error)
{
    if (error->error_code)
    {
          char buf[64];

          XGetErrorText (display, error->error_code, buf, 63);

          printf ("** X11 error in wxWidgets for GTK+: %s\n  serial %ld error_code %d
➥ request_code %d minor_code %d\n",
           buf,
           error->serial,
           error->error_code,
           error->request_code,
                  error->minor_code);
    }

    // Uncomment to forward to the default error handler
#if 0
    if (gs_pfnXErrorHandler)
        return gs_pfnXErrorHandler(display, error);
#endif
    return 0;
}
```

Summary

In this chapter, we've covered various aspects of memory management and error checking. You now know when to use new and when to create objects on the stack, how your application should clean itself up, how to identify memory leaks, and how to use macros for "defensive programming." You've seen how to write code that creates objects dynamically, and you should be able to decide when to use wxLogDebug and when to use wxLogError. You have also seen how to use C++ exceptions with wxWidgets, and we have presented some tips to help you debug your application. Next, we'll show how you can make your application work in many languages.

Chapter 16. Writing International Applications

If you target multiple languages as well as multiple platforms, you have the potential to reach a huge audience, which greatly increases your application's chances of success. This chapter covers what you need to do to make your application amenable to internationalization, which is sometimes abbreviated to "i18n" (an "i" followed by "18" characters followed by "n").

Introduction to Internationalization

When taking your application to an international market, the first thing that comes to mind is translation. You will need to provide a set of translations for all the strings your application presents in each foreign language that it supports. In wxWidgets, you do this by providing message catalogs and loading them through the wxLocale class. This technique may differ from how you are used to implementing translations if you currently use string tables. Instead of referring to each string by an identifier and switching tables, message catalogs work by translating a string using an appropriate catalog. (Alternatively, you can use your own system for handling translations if you prefer, but be aware that messages in the wxWidgets library itself rely on catalogs.)

Without message catalogs, the untranslated strings in the source code will be used to display text in menus, buttons, and so on. If your own language contains non-ASCII characters, such as accents, you will need a separate "translation" (message catalog) for it because source code should only contain ASCII.

Representing text in a different language can also involve different character encodings, which means that the same byte value might represent different characters when displayed on-screen. You need to make sure that your application can correctly set up the character encodings used by your GUI elements and in your data files. You need to be aware of the specific encoding used in each case and how to translate between encodings.

Another aspect of internationalization is formatting for numbers, date, and time. Note that formatting can be different even for the same language. For example, the number represented in English by 1,234.56 is represented as 1.234,56 in Germany and as 1'234.56 in the German-speaking part of Switzerland. In the USA, the 10th of November is represented as 11/10, whereas the same date for a reader in the UK means the 11th of October. We'll see shortly how wxWidgets can help here.

Translated strings are longer than their English counterpart, which means that the window layout must adapt to different sizes. Sizers are best suited to solve this part and are explained in Chapter 7, "Window Layout Using Sizers." Another layout problem is that for Western languages, the flow of reading goes from left to right, but other languages such as Arabic and Hebrew are read from right to left (called RTL), which means that the entire layout must change orientation. There is currently no specific mechanism for implementing RTL layout in wxWidgets.

The last group of elements to be adapted to a different language or culture consists of images and sounds. For example, if you are writing a phone directory application, you might have a feature that speaks the numbers, which will be language-dependent, and you might want to display different images depending on the country.

Providing Translations

wxWidgets provides facilities for message translation using the wxLocale class and is itself fully translated into several languages. Please consult the wxWidgets home page for the most up-to-date translations.

The wxWidgets approach to internationalization closely follows the GNU gettext package. wxWidgets uses message catalogs, which are binary compatible with gettext catalogs allowing you to use all the gettext tools. No additional libraries are needed during runtime because wxWidgets is able to read message catalogs.

During program development, you will need the gettext package for working with message catalogs (or poEdit; see the next section). There are two kinds of message catalog: source catalogs, which are text files with extension .po, and binary catalog which are created from the source files with the msgfmt program (part of the gettext package) and have the extension .mo. Only the binary files are needed during program execution. For each language you support, you need one message catalog.

poEdit

You don't have to use command-line tools for maintaining your message catalogs. Vaclav Slavik has written poEdit, a graphical front-end to the gettext package available from http://www.poedit.org. poEdit, shown in Figure 16-1, helps you to maintain message catalogs, generate .mo files, and merge in changes to strings in your application code as your application changes and grows.

Figure 16-1. poEdit

[View full size image]

Step-by-Step Guide to Using Message Catalogs

These are the steps you need to follow to create and use message catalogs:

1. Wrap literal strings in the program text that should be translated with wxGettranslation or equivalently with the _() macro. Strings that will not be translated should be wrapped in wxT() or the alias _T() to make them Unicode-compatible.

2. Extract the strings to be translated from the program into a .po file. Fortunately, you don't have to do this by hand; you can use the xgettext program or, more easily, poEdit. If you use xgettext, you can use the -k option to recognize wxGettranslation as well as _(). poEdit can also be configured to recognize wxGettranslation via the

Character Encodings and Unicode

There are more characters around on Earth than can fit into the 256 possible byte values that the classical 8-bit character represents. In order to be able to display more than 256 different glyphs, another layer of indirection has been added: the character encoding or character set. (The "new and improved" solution, Unicode, will be presented later in this section.)

Thus, what is represented by the byte value 161 is determined by the character set. In the ISO 8859-1 (Latin-1) character set, this is ¡an inverted exclamation mark. In ISO 8859-2 (Latin-2), it represents a Ą(Aogonek).

When you are drawing text on a window, the system must know about the encoding used. This is called the "font encoding," although it is just an indication of a character set. Creating a font without indicating the character set means "use the default encoding." This is fine in most situations because the user is normally using the system in his or her language.

But if you know that something is in a different encoding, such as ISO 8859-2, then you need to create the appropriate font. For example:

```
wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
               false, wxT("Arial"), wxFONTENCODING_ISO8859_2);
```

Otherwise, it will not be displayed properly on a western system, such as ISO 8859-1.

Note that there may be situations where an optimal encoding is not available. In these cases, you can try to use an alternative encoding, and if one is available, you must convert the text into this encoding. The following snippet shows this sequence: a string text in the encoding enc should be shown in the font facename. The use of wxCSConv will be explained shortly.

```
// We have a string in an encoding 'enc' which we want to
// display in a font called 'facename'.
//
// First we must find out whether there is a font available for
// rendering this encoding

wxString text; // Contains the text in encoding 'enc'
if (!wxFontMapper::Get()->IsEncodingAvailable(enc, facename))
{
   // We don't have an encoding 'enc' available in this font.
   // What alternative encodings are available?

   wxFontEncoding alternative;
   if (wxFontMapper::Get()->GetAltForEncoding(enc, &alternative,
                                              facename, false))
   {
       // We do have a font in an 'alternative' encoding,
       // so we must convert our string into that alternative.

       wxCSConv convFrom(wxFontMapper::GetEncodingName(enc));
       wxCSConv convTo(wxFontMapper::GetEncodingName(alternative));
       text = wxString(text.wc_str(convFrom), convTo) ;

       // Create font with the encoding alternative

       wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
                false, facename , alternative);
       dc.SetFont(myFont);
   }
   else
   {
      // Unable to convert; attempt a lossy conversion to
      // ISO 8859-1 (7-bit ASCII)
```

Numbers and Dates

The current locale is also used for formatting numbers and dates. The printf-based formatting in wxString takes care of this automatically. Consider this snippet:

```
wxString::Format(wxT("%.1f") , myDouble);
```

Here, Format uses the correct decimal separator. And for date formatting:

```
wxDateTime t = wxDateTime::Now();
wxString str = t.Format();
```

Format presents the current date and time in the correct language and format. Have a look at the API documentation to see all the possibilities for passing a format description to the Format calljust don't forget that you will probably have to translate this format string as well for using it in a different language because the sequence for different parts of the date can differ among languages.

If you want to add the correct thousands separator or just want to know the correct character for the decimal point, you can ask the locale object for the corresponding strings using the GetInfo method:

```
wxString info = m_locale.GetInfo(wxLOCALE_THOUSANDS_SEP,
                                 wxLOCALE_CAT_NUMBER) ;
```

Other Media

You can also load other data such as images and sounds depending on the locale. You can use the same mechanism as for text, for example:

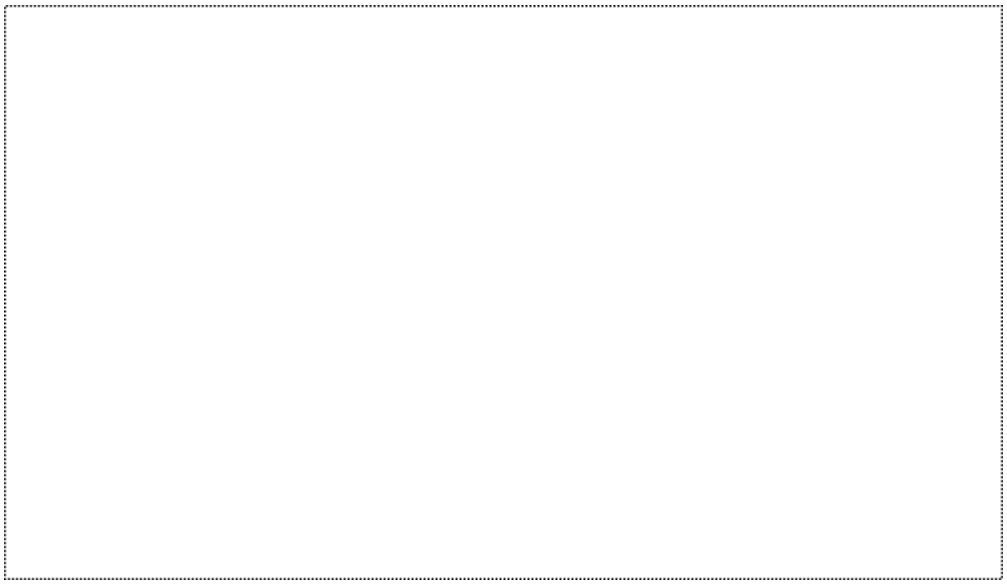
```
wxBitmap bitmap(_("flag.png"));
```

This code will cause flag.png to appear on your list of strings to translate, so you just translate the string flag.png into the appropriate file name for your platform, for example de/flag.png. Make sure that the translated versions are also available as true files in your application, or you can load them from a compressed archive (refer to Chapter 14, "Files and Streams").

A Simple Sample

To illustrate some of the concepts that we've covered, you can find a little sample in examples/chap16 on the CD-ROM. It shows some strings and a flag graphic for three languages: English, French, and German. You can change the language from the File menu, which will change the menu strings, the wxStaticText strings, and the flag graphic to suit the newly selected language. To demonstrate the different behavior of _() and wxT(), the menu help in the status line remains in English.

Figure 16-2. The internationalization samples



The sample's application class contains a wxLocale pointer and a function SelectLanguage that will re-create the locale object with the appropriate language. This is the application class declaration and implementation:

```
class MyApp : public wxApp
{
public:
    ~MyApp() ;

    // Initialize the application
    virtual bool OnInit();

    // Recreates m_locale according to lang
    void SelectLanguage(int lang);

private:
    wxLocale* m_locale; // 'our' locale
};

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
{
    wxImage::AddHandler( new wxPNGHandler );
    m_locale = NULL;
    SelectLanguage( wxLANGUAGE_DEFAULT );

    MyFrame *frame = new MyFrame(_("i18n wxWidgets App"));

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

void MyApp::SelectLanguage(int lang)
```

Summary

We've discussed the variety of ways in which wxWidgets helps you handle translations as well as formatting issues related to time and date, currency, and so on. You should work with someone familiar with the target languages or locales who will be able to find differences that you might have missed.

For another example of a translated application, see samples/internat in your wxWidgets distribution. It demonstrates translation of strings in menu items and dialogs for ten languages.

Next, we'll take a look at how you can make your applications perform several tasks at once with multithreading.

Chapter 17. Writing Multithreaded Applications

Most of the time, the event-driven nature of GUI programming maintains a good illusion of handling multiple tasks simultaneously. Redrawing a window usually takes a tiny fraction of a second, and user input can be handled rapidly. However, there are times when a task cannot easily be broken down into small chunks handled by a single thread, and this is where multithreaded programming becomes useful. This chapter shows you how threads can be controlled in wxWidgets, and it ends with some alternatives to using threads.

When to Use Threads, and When Not To

A thread is basically a path of execution through a program. Threads are sometimes called lightweight processes, but the fundamental difference between threads and processes is that the memory spaces of different processes are separate, whereas all threads in the same process share the same address space. Although this makes it much easier to share common data between several threads, multithreading also makes it easier to shoot oneself in the proverbial foot by accessing the same data simultaneously, so careful use of synchronization objects such as mutexes and critical sections is recommended.

When used properly, multithreading enables the programmer to simplify the application architecture by decoupling the user interface from the "real work." Note that this won't result in faster applications unless the computer has multiple processors, but the user interface will be more responsive.

wxWidgets provides both a thread class and the necessary synchronization objects (mutexes and critical sections with conditions). The threading API in wxWidgets resembles the POSIX threading API (pthreads), although several functions have different names, and some features inspired by the Win32 thread API are also available.

These classes make writing multithreaded applications easier, and they also provide some extra error checking compared with the native thread API. However, using threads is still a non-trivial undertaking, especially for large projects. Before starting a multithreaded application or adding multithreaded features to an existing one, it is worth considering alternatives to threads to implement the same functionality. In some situations, threads are the only reasonable choice, such as an FTP server application that launches a new thread for each new client. However, using an extra thread to show a progress dialog during a long computation would be overkill. In this case, you could do the calculations in an idle handler and call wxWindow::Update periodically to update the screen. For more details, see "Alternatives to Multithreading" toward the end of this chapter.

If you decide to use threads in your application, it is strongly recommended that only the main thread call GUI functions. The wxWidgets thread sample shows that it is possible for many different threads to call GUI functions at once, but in general, it is a very poor design choice. A design that uses one GUI thread and several worker threads that communicate with the main one using events is much more robust and will undoubtedly save you countless problems. For example, under Win32, a thread can only access GDI objects such as pens, brushes, and so on, created by itself, not those created by other threads.

For communication between threads, you can use wxEvtHandler::Add PendingEvent or its short version, wxPostEvent. These functions have thread-safe implementations so that they can be used for sending events between threads.

Using wxThread

If you want to implement functionality using threads, you write a class that derives from wxThread and implements at least the virtual EnTRy method, which is where the work of the thread takes place. Let's say you wanted to use a separate thread to count the number of colors in an image. Here's the declaration of the thread class:

```
class MyThread : public wxThread
{
public:
    MyThread(wxImage* image, int* count):
        m_image(image), m_count(count) {}
    virtual void *Entry();
private:
    wxImage* m_image;
    int*     m_count;
};

// An identifier to notify the application when the
// work is done
#define ID_COUNTED_COLORS    100
```

The Entry function does the calculation and returns an exit code that will be returned by Wait (for joinable threads only). Here's the implementation for EnTRy:

```
void *MyThread::Entry()
{
    (* m_count) = m_image->CountColours();

    // Use an existing event to notify the application
    // when the count is done
    wxCommandEvent event(wxEVT_COMMAND_MENU_SELECTED,
                         ID_COUNTED_COLORS);
    wxGetApp().AddPendingEvent(event);

    return NULL;
}
```

For simplicity, we're using an existing event class to send a notification to the application when the color count is done.

Creation

Threads are created in two steps. First the object is instantiated, and then the Create method is called:

```
MyThread *thread = new MyThread();
if ( thread->Create() != wxTHREAD_NO_ERROR )
{
    wxLogError(wxT("Can't create thread!"));
}
```

There are two different types of threads: the ones that you start and then forget about and the ones from which you are awaiting a result. The former are called detached threads, and the latter are joinable threads. Thread type is indicated by passing wxTHREAD_DETACHED (the default) or wxThrEAD_JOINABLE to the constructor of a wxTHRead. The result of a joinable thread is returned by the Wait function. You cannot wait for a detached thread.

You shouldn't necessarily create all threads as joinable, however, because joinable threads have a disadvantage; you must wait for the thread using wxThread::Wait or else the system resources that it uses will never be freed, and you must delete the corresponding wxTHRead object yourself (once used, it cannot be reused). In contrast, detached threads are of the "fire-and-forget" kind. You only have to start a detached thread, and it will terminate and destroy itself.

Synchronization Objects

In almost any use of threads, data is shared between different threads. When two threads attempt to access the same data, whether it is an object or a resource, then the access has to be synchronized to avoid data being accessed or modified by more than one thread at the same time. There are almost always so-called invariants in a programassumptions about related elements of data, such as having a correct first element pointer in a list and having each element point to the next element and a NULL pointer in the last element. During insertion of a new element, there is a moment when this invariant is broken. If this list is used from two threads, then you must guard against this moment so that no other client of the list is using it in this intermediate state.

It is the programmer's responsibility to make sure that shared data is not just grabbed by any thread but rather is accessed in a controlled manner. This section describes the classes you can use to achieve this protection.

wxMutex

The name comes from mutual exclusion and is the easiest synchronization element to use. It makes sure that only one thread is accessing a particular piece of data. To gain access to the data, the application calls wxMutex::Lock, which blocks (halts execution) until the resource is free. wxMutex::Unlock frees the resource again. Although you can use wxMutex's Lock and Unlock functions directly, by using the wxMutexLocker class, you can be sure that the mutex is always released correctly when the instance is destroyed, even if an exception occurred in your code.

In the following example, we assume that the MyApp class contains an m_mutex member of type wxMutex.

```
void MyApp::DoSomething()
{
    wxMutexLocker lock(m_mutex);
    if (lock.IsOk())
    {
        ... do something
    }
    else
    {
        ... we have not been able to
        ... acquire the mutex, fatal error
    }
}
```

There are three important rules for using mutexes:

1. A thread cannot generally lock a mutex that is already locked (no mutex recursion). Although there are systems that allow this, it is not portable across all operating systems.
2. A thread cannot unlock a mutex that a different thread has locked. If you need such a construct, you must use semaphores (discussed later).
3. If you are in a thread that is able to do other work if it cannot lock the mutex, you should call wxMutex::TRyLock. It returns immediately and indicates whether it was able to lock the mutex (wxMUTEX_NO_ERROR) or not (wxMUTEX_DEAD_LOCK or wxMUTEX_BUSY). This is especially important for the main (GUI) thread, which should never be blocked because your application would become unresponsive to user input.

Deadlocks

A deadlock occurs if two threads are waiting for resources that the other thread has already acquired. So supposing

The wxWidgets Threads Sample

You can find a working example of many of the features we have described in samples/thread in your wxWidgets distribution (see Figure 17-1). In this example, you can start, stop, pause, and resume threads. It demonstrates a "worker thread" that periodically posts events to the main thread with wxPostEvent, indicated by a progress dialog that cancels the thread when it reaches the end of its range.

Figure 17-1. wxWidgets Threads Sample

[View full size image]

Alternatives to Multithreading

If you find threads daunting, you may well be able to get away with a simpler approach, using timers, idle time processing, or both.

Using wxTimer

The wxTimer class lets your application receive periodic notification, either as a "single shot" or repeatedly. You can use wxTimer as an alternative to threads if you can break your task up into small chunks that are performed every few milliseconds, giving enough time for the application to respond to user interface events.

You can choose how your code will be notified. If you prefer to use a virtual function, derive a class from wxTimer and override the Notify function. If you prefer to receive a wxTimerEvent event, pass a wxEvtHandler pointer to the timer object (in the constructor or using SetOwner), and use EVT_TIMER(id, func) to connect the timer to an event handler function.

Optionally, you can pass an identifier that you passed to the constructor or SetOwner to uniquely identify the timer object and then pass that identifier to EVT_TIMER. This technique is useful if you have several timer objects to handle.

Start the timer by calling Start, passing a time interval in milliseconds and wxTIMER_ONE_SHOT if only a single notification is required. Calling Stop stops the timer, and IsRunning can be used to determine whether the timer is running.

The following example shows the event handler approach.

```
#define TIMER_ID 1000

class MyFrame : public wxFrame
{
public:
    ...
    void OnTimer(wxTimerEvent& event);

private:
    wxTimer m_timer;
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_TIMER(TIMER_ID, MyFrame::OnTimer)
END_EVENT_TABLE()

MyFrame::MyFrame()
       : m_timer(this, TIMER_ID)
{
    // 1 second interval
    m_timer.Start(1000);
}

void MyFrame::OnTimer(wxTimerEvent& event)
{
    // Do whatever you want to do every second here
}
```

Note that your event handler is not guaranteed to be called exactly every n milliseconds; the actual interval depends on what other processing was happening before the timer event was processed.

While we're on the subject of marking time, wxStopWatch is a useful class for measuring time intervals. The constructor starts the timer; you can pause and resume it and get the elapsed time in milliseconds. For example:

Summary

Just as giving a cashier two lines to process instead of one won't make her process more customers per hour, multithreading won't make your application go faster (at least without special hardware). However, it can seem faster to the user because the user interface is more responsive, and like a cashier waiting for credit-card clearance on one of the lines while processing the other, multithreading can use available resources more efficiently. It can also be a good way to solve certain problems more elegantly than would be possible if using only a single thread. In this chapter, we've also touched briefly on avoiding multithreading by using timers, idle event processing, and yielding.

There is more to multithreaded programming than this chapter can cover. For further reading, we recommend Programming with POSIX Threads by David R. Butenhof.

Our next chapter looks at using programming with sockets to pass data between processes.

Chapter 18. Programming with wxSocket

A socket is a conduit for data. A socket doesn't care what kind of data passes through it, where the data is going, or where the data is coming from; its goal is to transport data from point A to point B. Sockets are used every time you surf the web, check your email, or sign on to an instant messenger. One of the neatest aspects of sockets is that they can be used to connect any two devices that support sockets, even if one of them is a computer and the other is a refrigerator!

The socket API was originally a part of the BSD Unix operating system, and because that socket API originated from only one source, it has become the standard. All modern operating systems offer a socket layer, providing the ability to send data over a network (such as the Internet) using common protocols such as TCP or UDP. Using wxWidgets' wxSocket classes, you can reliably communicate any amount of data from one computer to another. This chapter assumes some basic socket terminology knowledge, but socket operations are generally straightforward.

Even though the basic socket features and functions are very similar on Windows, Linux, and Mac OS X, each socket API implementation has its own nuances, usually necessitating platform-specific tweaks. More importantly, event-based sockets have very different APIs from one platform to the next, often making it a significant challenge to use them. wxWidgets provides socket classes that make it easy to use sockets in advanced applications without having to worry about platform-specific implementations or quirks.

Please note that wxWidgets does not, at the time of this writing, support sending and receiving datagrams using the UDP protocol. Future releases of wxWidgets might add UDP capabilities.

Socket Classes and Functionality Overview

At the core of socket operations is wxSocketBase, which provides the basic socket functionality for sending and receiving data, closing, error reporting, and so on. Establishing a listening socket or connecting to a server requires wxSocketServer or wxSocketClient, respectively. wxSocketEvent is used to notify the application of an event that has occurred on a socket. The abstract class wxSocketBase and its children such as wxIPV4address enable you to specify remote hosts and ports. Lastly, stream classes such as wxSocketInputStream and wxSocketOutputStream can be coupled with other streams to move and transform data over a socket. Streams were discussed in Chapter 14, "Files and Streams."

Sockets in wxWidgets can operate in different ways, as discussed later in the "Socket Flags" section. The traditional threaded socket approach is handled by disabling the socket events and using blocking socket calls. On the other hand, you can enable socket events and eliminate the need for a separate thread; wxWidgets will send an event to your application when processing is required on a socket. By letting the data arrive in the background and processing data only when it is present, you avoid blocking the GUI, and you avoid the complexity of putting each socket in its own thread.

This chapter provides examples of both methods as well as a thorough explanation of the API for wxSocket and related classes. The examples and reference can be read and used independently, although the examples are intended to preface the explanation of the APIs.

Introduction to Sockets and Basic Socket Processing

As an introduction to socket programming with wxWidgets, let's jump right in to an event-based client/server example. The code is fairly readable with just a basic background in socket programming. For brevity, the GUI elements of the program are omitted, and we focus only on the socket functions; the complete application is available on the CD-ROM in examples/chap18. The detailed socket API reference follows the order of the code in the example.

The program performs a very simple task. The server listens for connections, and when a connection is made, the server reads ten characters from the client and then sends those same ten characters back to the client. Likewise, the client creates a connection, sends ten characters, and then receives ten characters in return. The string sent by the client is hard-coded in the example to 0123456789. The server and client programs are illustrated in Figure 18-1.

Figure 18-1. Socket server and client programs

The Client

This is the code for the client program.

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(CLIENT_CONNECT, MyFrame::OnConnectToServer)
    EVT_SOCKET(SOCKET_ID,    MyFrame::OnSocketEvent)
END_EVENT_TABLE()

void MyFrame::OnConnectToServer(wxCommandEvent& WXUNUSED(event))
{
    wxIPV4address addr;
    addr.Hostname(wxT("localhost"));
    addr.Service(3000);

    // Create the socket
    wxSocketClient* Socket = new wxSocketClient();
```

Socket Flags

The behaviors of a socket when using the socket classes can be quite different depending on which socket flags are set. The socket flags and their meanings are described in Table 18-3 and in more detail below.

Table 18-3. Socket Flags

wxSOCKET_NONE	Normal functionality (the behavior of the underlying send and recv functions).
wxSOCKET_NOWAIT	Read or write as much data as possible and return immediately.
wxSOCKET_WAITALL	Wait for all required data to be read or written unless an error occurs.
wxSOCKET_BLOCK	Block the GUI while reading or writing data.

If no flag is specified (the same as wxSOCKET_NONE), I/O calls will return after some data has been read or written, even when the transfer might not be complete. This is the same as issuing exactly one blocking low-level call to recv or send. Note that blocking here refers to when the function returns, not whether the GUI blocks during this time.

If wxSOCKET_NOWAIT is specified, I/O calls will return immediately. Read operations will retrieve only the available data, and write operations will write as much data as possible, depending on how much space is available in the output buffer. This is the same as issuing exactly one non-blocking low-level call to recv or send. Note that non-blocking here refers to when the function returns, not whether the GUI blocks during this time.

If wxSOCKET_WAITALL is specified, I/O calls won't return until all the data has been read or written (or until an error occurs), blocking if necessary and issuing several low-level calls if needed. This is the same as having a loop that makes as many blocking low-level calls to recv or send as needed to transfer all the data. Again, blocking here refers to when the function returns, not whether the GUI blocks during this time. Note that ReadMsg and WriteMsg will implicitly use wxSOCKET_WAITALL and ignore wxSOCKET_NONE and wxSOCKET_NOWAIT.

The wxSOCKET_BLOCK flag controls whether the GUI blocks during I/O operations. If this flag is specified, the socket will not yield during I/O calls, so the GUI will remain blocked until the operation completes. If it is not used, then the application must take extra care to avoid unwanted re-entrance.

To summarize:

- wxSOCKET_NONE will try to read at least some data, no matter how much.
- wxSOCKET_NOWAIT will always return immediately, even if it cannot read or write any data.
- wxSOCKET_WAITALL will only return when it has read or written all the data.
- wxSOCKET_BLOCK has nothing to do with the previous flags; it controls whether the GUI blocks during socket operations.

Using Socket Streams

Using wxWidgets' streams, it is easy to move and transform large amounts of data with only a few lines of code. Consider the task of sending a file over a socket. One approach would be to open the file, read the entire file into memory, and then send the memory block to the socket. This approach is fine for small files, but reading a large multi-megabyte file into memory might not be very speedy on a slower, low-memory computer. Furthermore, what if you were required to also compress the file as you send it to reduce network traffic? Reading a large file into memory, compressing it all at once, and then writing it to a socket simply would not be efficient or practical.

A second approach might be to read the file in small pieces, such as several kilobytes, compress the pieces, and then output these pieces over the socket. Unfortunately, compressing the individual pieces is not going to be as efficient as compressing the whole file at once. A refinement might be to maintain stateful compression streams from one piece to the next (where the compression of one frame can use compression information from the previous, most notably avoiding the need for each frame to have its own header), but you're already looking at dozens of lines of code and the delicate synchronization of reading from the file, compressing, and sending. With wxWidgets, there is a better way.

Because wxWidgets provides both wxSocketInputStream and wxSocket OutputStream classes, it is very easy to stream data in and out of sockets through other streams. Consider that wxWidgets provides streams for files, strings, text, memory, and Zlib compression, and some very interesting possibilities become apparent for using sockets in unique and powerful ways. If we revisit the file sending with compression problem with streams on our tool belt, a new solution is available. To send a file, we can stream data from the file to the Zlib compression to the socket, and suddenly we have stateful file compression, resulting in a completely compressed file being sent without reading more than a few kilobytes at a time. On the receiving end, we can stream the data from the socket through the Zlib decompression, and finally into the output file. All this can be done in only a few lines of code.

We will do our file streaming in a thread so that we can block on the socket operations and not worry about blocking the GUI or running into the 100% CPU usage issue detailed earlier, which is quite possible if we were to send large multi-megabyte files.

The complete socket stream sources can be found on the accompanying CD-ROM in examples/chap18.

File Sending Thread

The sending thread demonstrates using streams allocated dynamically on the heap. FileSendThread derives from wxTHRead.

```
FileSendThread::FileSendThread(wxString Filename,
                               wxSocketBase* Socket)
{
    m_Filename = Filename;
    m_Socket = Socket;

    Create();
    Run();
}

void* FileSendThread::Entry()
{
    // If we can't write anything for 10 seconds, assume a timeout
    m_Socket->SetTimeout(10);

    // Wait for all the data to write, blocking on the socket calls
    m_Socket->SetFlags(wxSOCKET_WAITALL | wxSOCKET_BLOCK);

    // Read from the specified file
    wxFileInputStream* FileInputStream =
        new wxFileInputStream(m_Filename);
```

Alternatives to wxSocket

Although wxSocket provides a lot of flexibility and is nicely integrated into wxWidgets, it's not the only method you can use to communicate with other processes. If you just want to perform FTP or HTTP operations, you can use wxFTP or wxHTTP, which use wxSocket. However, these classes are incomplete, and you may be better off using CURL, a popular library that gives you a very straightforward API for transferring files using a variety of common Internet protocols. There is even a wxWidgets wrapper for CURL available, called wxCURL.

wxWidgets also provides a high-level interprocess communication facility that uses the classes wxServer, wxClient, and wxConnection and an API based on Microsoft's DDE (Dynamic Data Exchange) protocol. In fact, on Windows, these classes use DDE, and on other platforms, sockets. The main advantage of using this higher-level API is its ease of use compared with using wxSocket. Another advantage is that on Windows, your applications can be DDE-aware, and other applications (not necessarily written with wxWidgets) can access it. A disadvantage is that on platforms other than Windows, it is not a recognized protocol to which non-wxWidgets applications can easily conform. However, if you just need to communicate between two wxWidgets applications, it can fit the bill. We show a very simple example in the section "Single Instance or Multiple Instances?" in Chapter 20, "Perfecting Your Application."

For more information, please refer to the topic "Interprocess Commun-ication Overview" in the reference manual and the source in samples/ipc in your wxWidgets distribution. You can also see these classes in action in the standalone help viewer in utils/helpview/src, again in the wxWidgets distribution.

Summary

We've seen how the wxSocket class provides wxWidgets integration and numerous enhancements to the underlying C sockets layer. To make socket programming even easier, wxWidgets also gives you socket stream classes that can stream data through a variety of classes. As long as you carefully consider the socket flags discussion, you will get consistent and reliable socket operations across Windows, Linux, and Mac OS X when using wxSocket and its related classes.

Next, we'll look at how your application design and implementation can be simplified by using the wxWidgets document/view framework.

Chapter 19. Working with Documents and Views

This chapter discusses how the document/view framework provided by wxWidgets can dramatically reduce the amount of code you need to write for a document-based application. It also discusses the related topic of providing undo and redo in your application, a seemingly miraculous facility that users now take for granted.

Document/View Basics

The document/view system is found in most application frameworks because it can greatly simplify the code required to build many kinds of applications.

The idea is that you can model your application primarily in terms of documents, which store data and provide GUI-independent operations upon it, and views, which display and manipulate the data. It's similar to the Model-View-Controller model (MVC), but here the concept of controller is not separate from the view.

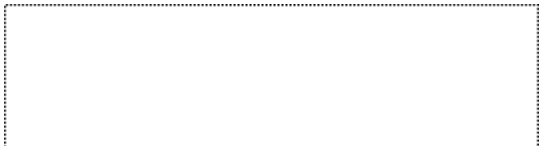
wxWidgets can provide many user interface elements and behaviors based on this architecture. After you have defined your own classes and the relationships between them, the framework takes care of showing file selectors, opening and closing files, asking the user to save modifications, routing menu commands to appropriate code, and even some default print and print preview functionality and support for command undo and redo. The framework is highly modular, enabling the application to override and replace functionality and objects to achieve more than the default behavior.

These are the main steps involved in creating an application based on the document/view framework, once you've decided that this model is appropriate for your application. The ordering is to some extent arbitrary: for example, you may choose to write your document class before thinking about how a document will be presented in the application.

1. Decide what style of interface you will use: Microsoft's MDI (multiple document child frames surrounded by an overall frame), SDI (a separate, unconstrained frame for each document), or single-window (one document open at a time, as in Windows Write).
2. Use the appropriate parent and child frame classes, based on the previous decision, such as wxDocParentFrame and wxDocChildFrame. Construct an instance of the parent frame class in your wxApp::OnInit, and a child frame class (if not single-window) when you initialize a view. Create menus using standard menu identifiers such as wxID_OPEN and wxID_PRINT.
3. Define your own document and view classes, overriding a minimal set of member functions for input/output, drawing, and initialization. If you need undo/redo, implement it as early as possible, instead of trying to retrofit your application with it later.
4. Define any subwindows (such as a scrolled window) that are needed for the view(s). You may need to route some events to views or documents; for example, your paint handler needs to be routed to wxView::OnDraw.
5. Construct a single wxDocManager instance at the beginning of your wxApp::OnInit and then as many wxDocTemplate instances as necessary to define relationships between documents and views. For a simple application, there will be just one wxDocTemplate.

We will illustrate these steps using a little application called Doodle (see Figure 19-1). As its name suggests, it supports freehand drawing in a window, with the ability to save and load your doodles. It also supports simple undo and redo.

Figure 19-1. Doodle in action



Other Document/View Capabilities

We've shown a simple example that demonstrates the essential steps you need to follow for a complete document-centric application. This section presents further tips on using the document/view framework.

Standard Identifiers

The document/view system has default behavior for the following standard identifiers: wxID_OPEN, wxID_CLOSE, wxID_CLOSE_ALL, wxID_REVERT, wxID_NEW, wxID_SAVE, wxID_SAVEAS, wxID_UNDO, wxID_REDO, wxID_PRINT, and wxID_PREVIEW. Use these identifiers in your menu bars and toolbars to take full advantage of the framework. The functions that handle these identifiers are mostly implemented in wxDocManager: for example, OnFileOpen, OnFileClose, and OnUndo. These handlers then call appropriate functions in the currently active document. If you want, you can override these commands in your frame class or in a class derived from wxDocManager, but this is not usually necessary.

Printing and Previewing

By default, wxID_PRINT and wxID_PREVIEW use the standard wxDocPrintout class to implement printing and print preview, taking advantage of wxView:: OnDraw if implemented. However, this has very limited functionalityit only copes with a single-page documentso you're likely to want to create your own wxPrintout class and override the standard handlers for wxID_PRINT and wxID_PREVIEW. The quickest method is to use the wxHtmlEasyPrinting class, as described in "HTML Printing" in Chapter 12, "Advanced Window Classes."

File History

When your application initializes, you can load the file history shown at the bottom of the File menu from a wxConfig object with wxDocManager::File HistoryLoad and save it in your application cleanup with wxDocManager:: FileHistorySave. For example, to load the file history, type the following:

```
// Load the file history
wxConfig config(wxT("MyApp"), wxT("MyCompany"));
config.SetPath(wxT("FileHistory"));
m_docManager->FileHistoryLoad(config);
config.SetPath(wxT("/"));
```

If you load the file history before you create your main frame and its menu bar, you will need to explicitly call wxDocManager::FileHistoryAddFilesToMenu.

If you want, you can use the wxFileHistory class independently of the other document/view classes, or to implement file history in a different way (for example, if your application requires a separate file history for each document frame).

Explicit Document Creation

Sometimes you may want to create a document explicitly in your documentfor example, if you are opening the last document that was viewed. You can open an existing document like this:

```
wxDocument* doc = m_docManager->CreateDocument(filename,
                                               wxDOC_SILENT);
```

Or create a new document like this:

```
wxDocument* doc = m_docManager->CreateDocument(wxEmptyString,
                                               wxDOC_NEW);
```

Strategies for Implementing Undo/Redo

The way that undo/redo is implemented in your application will be determined by the nature of the document data and how the user manipulates that data. In our simple Doodle example, we only operate on one piece of data at a time, and our operations are very simple. In many applications, however, the user can apply operations to multiple objects. In this case, it's useful to introduce a further level of command granularity, perhaps called CommandState, holding the information for a particular object in the document. Your command class should maintain a list of these states and accept a list of them in the command class constructor. Do and Undo will iterate through the state list and apply the current command to each state.

One key to implementing undo/redo is noting that the user can only traverse the command history one step at a time. Therefore, your undo/redo implementation is free to take snapshots of the document state for later restoration, with the knowledge that the stored state will always be correct no matter how many times the user has gone backward or forward in the command history. All your code has to know is how to switch between "done" and "undone" states.

A common strategy is to store a copy of each document object within the command state object and also a pointer to the actual object. Do and Undo simply swap the current and stored states. Let's say the object is a shape, and the user changes the color from red to blue. The application creates a new state identical to the existing red object, but it sets the internal color attribute to blue. When the command is first executed, Do takes a copy of the current object state, applies the new command state (including the blue color) to the visible state, and repaints the object. Undo simply does the same thing. It takes a copy of the current object state, applies the current state (including red), and repaints. So code for Do and Undo is actually identical in this case. Not only that, it can be reused for other operations as well as color changes because the state of the entire object is copied. You can make this process straightforward by implementing an assignment operator and copy constructor for each class that represents an object that the user can edit.

Let's make this idea more concrete. Say we have a document of shapes, and the user can change the color of all selected shapes. We might have a command handler called ShapeView::OnChangeColor, as in the following, where a new state is created for each selected object, before being applied to the document.

```
// Changes the color of the selected shape(s)
void ShapeView::OnChangeColor(wxCommandEvent& event)
{
    wxColour col = GetSelectedColor();

    ShapeCommand* cmd = new ShapeCommand(wxT("Change color"));

    ShapeArray arrShape;
    for (size_t i = 0; i < GetSelectedShapes().GetCount(); i++)
    {
        Shape* oldShape = GetSelectedShapes()[i];
        Shape* newShape = new Shape(*oldShape);
        newShape->SetColor(col);
        ShapeState* state = new ShapeState(SHAPE_COLOR, newShape, oldShape);
        cmd->AddState(state);
    }
    GetDocument()->GetCommandProcessor()->SubmitCommand(cmd);
}
```

Because the implementation for this kind of state change is the same for both Do and Undo, we have a single DoAndUndo function in the ShapeState class, where we do state swapping:

```
// Incomplete implementation of the state's DoAndRedo:
// for some commands, do and undo share the same code
void ShapeState::DoAndUndo(bool undo)
{
    switch (m_cmd)
    {
    case SHAPE_COLOR:
```

Summary

We've seen how taking advantage of the document/view model can simplify an application's implementation, letting wxWidgets handle much of the housekeeping such as showing file dialogs and creating document and view objects. You should now also have an understanding of how to implement undo/redo in wxWidgetsa facility that should be built in at an early stage because trying to bolt it on afterwards will involve a considerable amount of rewriting.

In our final chapter, we discuss a number of ways in which you can refine your application.

Chapter 20. Perfecting Your Application

There's a world of difference between an application that does its job adequately and an application that's intuitive and enjoyable to use. For internal use, a basic, no-frills application may be adequate, but if you're planning on distributing your brainchild to a wide audience, you'll want it to be as compelling and easy to use as applications from the biggest software publishers. You'll need to conform to some fundamental expectations and standards, such as the provision of configuration dialogs and online help. In this final chapter, we'll cover the following topics to help you polish your application:

- Single instance or multiple instances? How you can prevent "clones" of your application from proliferating.
- Modifying event handling. How to change the order in which events are processed.
- Reducing flicker. How to improve the visual quality of your application by cutting down on annoying flicker.
- Implementing online help. Suggestions for providing a variety of help for your users.
- Parsing the command line. Give your users more control over your application with command-line options and switches.
- Storing application resources. Ways to package files needed by your application.
- Invoking other applications. From simple program execution to capturing another process's input and output.
- Managing application settings. The use of wxConfig to save and load settings, and tips for presenting settings.
- Application installation. Suggestions for how users can easily install your application on the three major platforms.
- Following UI design guidelines. Some observations about conforming to the design recommendations on each platform.

Single Instance or Multiple Instances?

Depending on the nature of your application, you might either want to allow users to keep invoking multiple instances of the application, or reuse the existing instance when the user tries to relaunch the application or open more than one document from the system's file manager. It's quite common to give users the option of running the application in single-instance or multiple-instance mode, depending on their working habits. One problem with allowing multiple instances is that the order in which the instances write their configuration data (usually when the application exits) is undefined, so user settings might be lost. It might also confuse novice users who don't realize they are launching new instances of the application. The behavior of allowing multiple applications to be launched is the default on all platforms (except when launching applications and opening documents via the Finder on Mac OS), so you will need to write additional code if you want to disable multiple instances.

On Mac OS (only), opening multiple documents using the same instance is easy. Override the MacOpenFile function, taking a wxString file name argument, which will be called when a document associated with this application is opened from the Finder. If the application is not already running, Mac OS will run it first before calling MacOpenFile (unlike on other platforms, it does not pass the file name in the command-line arguments). If you are using document/view, you might not need to provide this function because the default implementation on Mac OS X is as follows:

```
void wxApp::MacOpenFile(const wxString& fileName)
{
    wxDocManager* dm = wxDocManager::GetDocumentManager() ;
    if ( dm )
        dm->CreateDocument(fileName, wxDOC_SILENT) ;
}
```

However, even on Mac OS, this will not prevent the user from running the application multiple times if launching the executable directly, as opposed to opening an associated document. One way in which you can detect and forbid more than one instance from being run is by using the wxSingleInstanceChecker class. Create an object of this class that persists for the lifetime of the instance, and in your OnInit, call IsAnotherRunning. If this returns TRue, you can quit immediately, perhaps after alerting the user. For example:

```
bool MyApp::OnInit()
{
    const wxString name = wxString::Format(wxT("MyApp-%s"),
                               wxGetUserId().c_str());
    m_checker = new wxSingleInstanceChecker(name);
    if ( m_checker->IsAnotherRunning() )
    {
        wxLogError(_("Program already running, aborting."));
        return false;
    }

    ... more initializations ...

    return true;
}

int MyApp::OnExit()
{
    delete m_checker;

    return 0;
}
```

But what if you want to bring the existing instance to the front or open a file that was passed to the second instance in the first instance? In general, this requires the instances to be able to communicate with each other. We can use wxWidgets' high-level interprocess communication classes to do this.

In the following example, we'll set up communication between instances of the sample application and allow

Modifying Event Handling

Normally, wxWidgets sends an event to the window (or other event handler) that generated it. If it's a command event, it might work its way up the window hierarchy before being processed (see Appendix H, "How wxWidgets Processes Events," for details). For example, clicking on a copy toolbar button will cause the toolbar event table to be searched, then the frame that contains it, and then the application object. Although this may work fine if you support only one kind of copy command, there is a problem if you also want the copy command to apply both to your main editing window (say a drawing application) and to any focused text controls in the main window, for example. The text controls will never get copy commands from the toolbar button (or menu item) because it is wired to a custom event handler. In this case, it would be more appropriate to send the command to the focused control first. Then, if the focused control implements this command (such as wxID_COPY), it will be processed. If it doesn't, then the command will rise up the window hierarchy until it gets to the custom wxID_COPY event handler. The end result will be a more natural way of working, with commands applying to the data that the user is currently editing.

We can override the main frame's ProcessEvent function to catch command events and redirect them to the focused control (if any), as follows:

```
bool MainFrame::ProcessEvent(wxEvent& event)
{
    static wxEvent* s_lastEvent = NULL;

    // Check for infinite recursion
    if (& event == s_lastEvent)
        return false;

    if (event.IsCommandEvent() &&
       !event.IsKindOf(CLASSINFO(wxChildFocusEvent)) &&
       !event.IsKindOf(CLASSINFO(wxContextMenuEvent)))
    {
        s_lastEvent = & event;

        wxControl *focusWin = wxDynamicCast(FindFocus(), wxControl);
        bool success = false;

        if (focusWin)
            success = focusWin->GetEventHandler()
                              ->ProcessEvent(event);
        if (!success)
            success = wxFrame::ProcessEvent(event);

        s_lastEvent = NULL;
        return success;
    }
    else
    {
        return wxFrame::ProcessEvent(event);
    }
}
```

Currently, this is most useful when the focused control is a wxTextCtrl because for this control (on most platforms), wxWidgets supplies standard UI update and command handlers for common commands, including wxID_COPY, wxID_CUT, wxID_PASTE, wxID_UNDO, and wxID_REDO. However, you can always implement these handlers for arbitrary controls in your application or subclasses of existing wxWidgets controls such as wxStyledTextCtrl (see examples/chap20/pipedprocess for a wxStyledTextCtrl implementation enhanced in this way).

Reducing Flicker

Flicker is a perennial annoyance in GUI programming. Often an application will need some tweaking to reduce it; here are some tips that might help.

On Windows, if your window is using the wxFULL_REPAINT_ON_RESIZE style, try removing it. This will cause the repaint area to be restricted to only the parts of the window "damaged" by resizing, so less erasing and redrawing is done. Otherwise, even if the window was resized a tiny amount, the whole window will be refreshed, causing flicker. However, this will not work if the appearance of your window contents depends on the size because the whole window will require updating.

You may sometimes need to use the wxCLIP_CHILDREN style on Windows to prevent a window refresh from affecting its children. The style has no effect on other platforms.

When you are drawing in a scrolled window, you can do quite a lot to improve refresh speed and reduce flicker. First, optimize the way you find the appropriate data to draw: you need a way to gather just the information that is in the current view (see wxWindow::GetViewStart and wxWindow:: GetClientSize), and in your paint handler, you should also be able to eliminate the graphics that are not in the update region (wxWindow::GetUpdateRegion). You need data structures that can get you to the start of the view quickly before you start drawing. You may be able to calculate that position from your graphics (rather than search), for example if you have columns with constant width. A linked list or array should be fairly fast to search. If it's time-consuming to calculate the current position, then perhaps you can use a caching mechanism to store the current list position for the last set of data displayed, and then you can simply scan back or forward to find the data at the start of the current scroll position. You can keep a tally of the window position for each piece of data so that you don't have to recalculate the whole thing from the start to find where the data should be drawn.

When implementing scrolling graphics, you can use wxWindow::ScrollWindow to physically move the pixels on the window, and this will mean that only the remaining "dirty" area of the window will need refreshing, which will further reduce flickering. (wxScrolledWindow already does this for you, by default.) GetUpdateRegion will reflect the smaller amount of screen that you need to update.

As mentioned in Chapter 5, "Drawing And, Printing," you may want to define your own erase background handler and leave it empty to stop the framework from clearing the background itself. Then you can draw the whole graphic (including as much of the background as necessary) on top of the old one without the flashing caused by clearing the entire background before drawing the content. Use wxWindow::SetBackgroundStyle with wxBG_STYLE_CUSTOM to tell wxWidgets not to automatically erase the background. Chapter 5 also discusses use of wxBufferedDC and wxBufferedPaintDC, which you can use in combination with the other techniques mentioned here.

Another problem is inefficiency and flicker caused by doing many updates to a window in quick succession. wxWidgets provides wxWindow:: Freeze and wxWindow::Thaw to switch off updates while code is executed between these function calls. For example, you might want to use this when adding a lot of lines to a text control one by one or appending many items to a list box. When Thaw is called, the window will normally be completely refreshed. Freeze and Thaw are implemented on Windows and Mac OS X for wxWindow and on GTK+ for wxTextCtrl. You can also implement it for your own window classes to avoid doing unnecessary processing and updating (our wxThumbnailCtrl example from Chapter 12, "Advanced Window Classes," does this; see examples/chap12/thumbnail).

Implementing Online Help

Although you should make your application as intuitive as possible so that the user rarely has to resort to the manual, it's important to supply online help for all but the simplest application. You could supply it as a PDF file or even an HTML file to be viewed by the user's favorite browser, but it will make a much better impression if you use a special-purpose help system and link to appropriate topics from all your dialogs and from your main window.

wxWidgets has the concept of help controller, a class that your application uses to load help files and show topics. Several help controller classes are supported:

- wxWinHelpController, for controlling RTF-based WinHelp files on Windows (extension hlp). This format is deprecated, and new applications should use wxCHMHelpController.
- wxCHMHelpController, for controlling MS HTML Help files on Windows (extension chm).
- wxWinceHelpController, for controlling Windows CE Help files (extension htm).
- wxHtmlHelpController, for controlling wxWidgets HTML Help files on all platforms (extension htb).

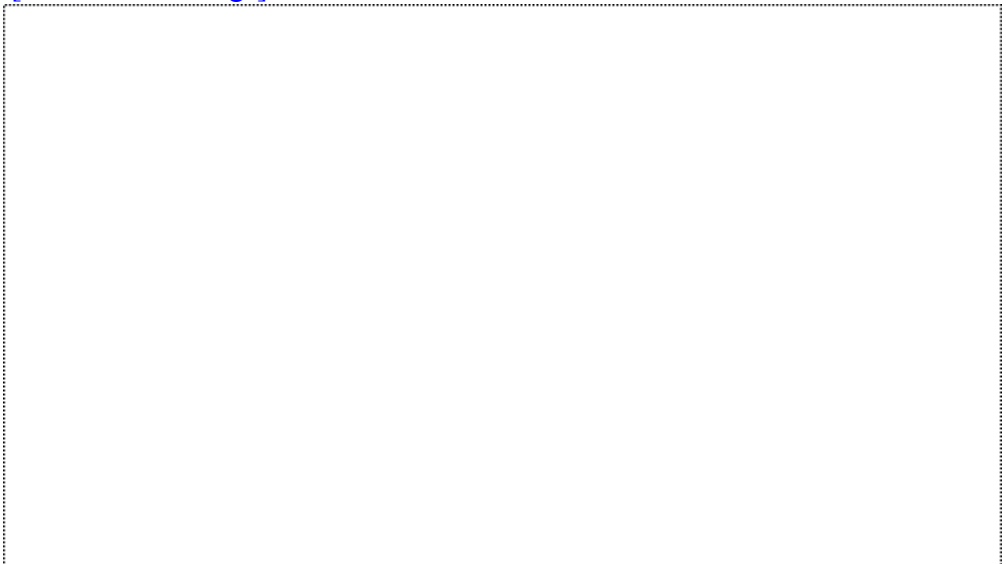
wxHtmlHelpController is different from the others in that instead of merely being a wrapper for an external help viewer, it is part of an entire help system implementation, and the help window resides in the same process as the application. If you want to use the wxWidgets HTML Help viewer as a separate process, compile HelpView in utils/src/helpview in your wxWidgets distribution. The files remhelp.h and remhelp.cpp implement a remote help controller (wxRemoteHtmlHelpController) that you can link with your application to control the HelpView instance.

Note that at the time of writing, there is no special help controller class for Mac OS X help files. You can use wxWidgets HTML Help files on this platform.

Figure 20-1 and Figure 20-2 show the same topic displayed in MS HTML Help and wxWidgets HTML Help viewers under Windows. The two provide similar functionality: HTML content on the right, and on the left, a hierarchy of topics, an index of topic names, and a keyword search facility. In addition, wxWidgets HTML Help can load multiple help files.

Figure 20-1. MS HTML Help

[View full size image]



Parsing the Command Line

Passing commands to the application to be read on initialization is often useful, and if the application is document-oriented, you should allow files to be loaded in this way. You also might want to let the application be run from the operating system command line, for example to automate tasks from a makefile, in which case command-line options can be used to tell the application that it should be run without a user interface. Although most configuration of an application will be done via the user interface, command-line configuration options can be appropriate in some cases, such as turning on a debug mode.

wxWidgets provides the wxCmdLineParser class to make this programming task quite easy, avoiding the need to process wxApp::argc and wxApp::argv directly. The class looks for switches (such as -verbose), options (such as-debug:1), and parameters (such as "myfile.txt"). It can recognize both short and long forms of switches and options, and each item can have a help string, which will be used by Usage when writing help text to the current log target.

Here's an example, showing how to parse switches, options, and parameters:
[View full width]

```
#include "wx/cmdline.h"

static const wxCmdLineEntryDesc g_cmdLineDesc[] =
{
    { wxCMD_LINE_SWITCH, wxT("h"), wxT("help"),    wxT("displays help on the command
line
➥ parameters") },
    { wxCMD_LINE_SWITCH, wxT("v"), wxT("version"), wxT("print version") },
    { wxCMD_LINE_OPTION, wxT("d"), wxT("debug"), wxT("specify a debug level") },

    { wxCMD_LINE_PARAM,  NULL, NULL, wxT("input file"), wxCMD_LINE_VAL_STRING,
➥ wxCMD_LINE_PARAM_OPTIONAL },

    { wxCMD_LINE_NONE }
};

bool MyApp::OnInit()
{
    // Parse command line
    wxString cmdFilename;
    wxCmdLineParser cmdParser(g_cmdLineDesc, argc, argv);
    int res;
    {
        wxLogNull log;
        // Pass false to suppress auto Usage() message
        res = cmdParser.Parse(false);
    }

    // Check if the user asked for command-line help
    if (res == -1 || res > 0 || cmdParser.Found(wxT("h")))
    {
        cmdParser.Usage();
        return false;
    }

    // Check if the user asked for the version
    if (cmdParser.Found(wxT("v")))
    {
#ifndef __WXMSW__
        wxLog::SetActiveTarget(new wxLogStderr);
#endif
        wxString msg;
        wxString date(wxString::FromAscii(__DATE__));
        msg.Printf(wxT("Anthemion DialogBlocks, (c) Julian Smart, 2005 Version %.2f,
%s"),
```

Storing Application Resources

A simple application might consist of no more than an executable, but more realistically, you'll have a help file, maybe HTML files and images, and application-specific data files. Where do these things go?

Reducing the Number of Data Files

You can do several things to create a more tidy installation. Firstly, you can include XPM images in your C++ files (with #include) instead of loading them from disk. Secondly, if you are using XRC files, you can compile them into C++ using the wxrc utility in utils/wxrc in your wxWidgets distribution:

```
wxrc resources.xrc --verbose --cpp-code --output resources.cpp
```

You can then call the InitXmlResource function defined in the generated C++ file to load the resources.

Thirdly, you can archive your data files into a standard zip file and extract them using streams and virtual file systems, as explained in for data files. Include "wx/fileloc.h" and then call the class's static functions, including GetConfigDir, GetInstallDir, GeTDataDir, GetLocalDataDir, and GetUserConfig Data. Refer to the wxWidgets manual for details of these functions' return values on each platform.

On Mac OS X, you'll need to create an application bundle: a file hierarchy with standard locations for the executable, data files, and so on. See later in the chapter for information on creating a bundle.

Finding the Application Path

A common request from wxWidgets developers is a function to find the path of the currently executing application to enable the application to load data that is in the same directory. There is no such function in wxWidgets, partly because it's difficult to achieve this reliably on all platforms, and partly to encourage the installation of data into standard locations (particularly on Linux). However, it can be convenient to put all the application's files in the same folder, so in examples/chap20/findapppath, you can find code for a function wxFindAppPath:

```
// Find the absolute path the application has been run from.

wxString wxFindAppPath(const wxString& argv0, const wxString& cwd,
                                    const wxString& appVariableName = wxEmptyString,
                                    const wxString& appName = wxEmptyString);
```

argv0 is the value of wxApp::argv[0], which on some platforms gives the full pathname for the application.

cwd is the current working directory (obtained with wxGetCwd), which is a hint that can be used on some platforms.

appVariableName is the name of an environment variable, such as MYAPPDIR, that may have been set externally to indicate the application's location.

appName is the prefix used in a bundle to allow the function to check the bundle contents for application location. For example, in the case of DialogBlocks, the argument is DialogBlocks, and on Mac OS X, the function will try to return <currentdir>/DialogBlocks.app/Content/MacOS for the executable location.

Here's an example of using wxFindAppPath:

```
bool MyApp::OnInit()
{
    wxString currentDir = wxGetCwd();
    m_appDir = wxFindAppPath(argv[0], currentDir, wxT("MYAPPDIR"),
                             wxT("MyApp"));
    ...
```

Invoking Other Applications

Sometimes you may want to run other applications from your own application, whether it's an external browser or another of your own applications. wxExecute is a versatile function that can be used to run a program with or without command-line arguments, synchronously or asychronously, optionally collecting output from the launched process, or even redirecting input to and output from the process to enable the current application to interact with it.

Running an Application

Here are some simple examples of using wxExecute:

```
// Executes asychronously by default (returns immediately)
wxExecute(wxT("c:\\windows\\notepad.exe"));

// Does not return until the user has quit Notepad
wxExecute(wxT("c:\\windows\\notepad.exe c:\\temp\\temp.txt"),
          wxEXEC_SYNC);
```

Note that you can optionally enclose parameters and the executable in quotation marks, which is useful if there are spaces in the names.

Launching Documents

If you want to run an application to open an associated document, you can use the wxMimeTypesManager class on Windows and Linux. You can find out the file type associated with the extension and then use it to get the required command to pass to wxExecute. For example, to view an HTML file:

```
wxString url = wxT("c:\\home\\index.html");
bool ok = false;
wxFileType *ft = wxTheMimeTypesManager->
                 GetFileTypeFromExtension(wxT("html"));
if ( ft )
{
     wxString cmd;
      ok = ft->GetOpenCommand(&cmd,
                          wxFileType::MessageParameters(url, wxEmptyString));
      delete ft;

      if (ok)
      {
           ok = (wxExecute(cmd, wxEXEC_ASYNC) != 0);
       }
}
```

Unfortunately, this doesn't work under Mac OS X because OS X uses a completely different way of associating document types with applications. For arbitrary documents, it's better to ask the Finder to open a document, and for HTML files, it's better to use the special Mac OS X function, ICLaunchURL. wxExecute is not always the best solution under Windows either, where ShellExecute may be a more effective function to use for HTML files. Even on Unix, there may be specific fallback scripts you want to use if an associated application is not found, such as htmlview.

To work around these problems, we include the files launch.h and launch.cpp in examples/chap20/launch. This implements the functions wxLaunchFile, wxViewHTMLFile, wxViewPDFFile, and wxPlaySoundFile that work on Windows, Linux, and Mac OS X.

wxLaunchFile is a general-purpose document launcher. Pass a document file name or an executable file name with or without arguments and an optional error message string that can be presented to the user if the operation fails. If an HTML file is passed, wxLaunchFile will call wxViewHTMLFile. On Mac OS X, it will use the finder to launch a

Managing Application Settings

Most applications have options to let the user to change the program's behavior, such as toggling tooltips on and off, changing text fonts, or suppressing a splash screen. The developer must make choices about how these settings will be stored and displayed. For storage, the usual solution is to use the wxConfig family of classes, which is especially intended to store typed application settings. The choice of user interface is much more varied, and we'll explore some possibilities shortly.

Storing Settings

The configuration classes that wxWidgets provides are derived from wxConfigBase, which is where you'll find the documentation in the reference manual. On each platform, wxConfig is an alias for an appropriate class: on Windows it's wxRegConfig (using the Windows Registry), and on all other platforms it's wxFileConfig (using text files). wxIniConfig is available, using Windows 3.1-style .ini files, but this is rarely needed. wxFileConfig is available on all platforms.

wxConfig provides the overloaded functions Read and Write to read and store items of type wxString, long, double, and bool. Each item is referenced by a name comprising a list of slash-separated pathsfor example, "/General/UseTooltips". By using wxConfig::SetPath, you can set the current position in the hierarchy, and subsequent references will be relative to that position if they do not have a leading slash. You can use paths to group your settings.

wxConfig constructors take an application name and vendor name, which are used to determine the location of the settings. For example:

```
#include "wx/config.h"

wxConfig config(wxT("MyApp"), wxT("Acme"));
```

wxRegConfig constructs a location from the vendor name and application name, and in the previous example, the settings will be stored under HKEY_CURRENT_USER/Software/Acme/MyApp in the Registry. Using wxFileConfig on Unix, the settings are stored by default in a file called ~/.MyApp. On Mac OS X, they are stored in õ /Library/Preferences/MyApp Preferences. This location can be changed by passing a third parameter to the wxConfig constructor.

Here are some examples of wxConfig usage:

```
// Read some values
wxString str;
if (config.Read(wxT("General/DataPath"), & str))
{
    ...
}
bool useToolTips = false;
config.Read(wxT("General/ToolTips"), & useToolTips));
long usageCount = 0;
config.Read(wxT("General/Usage"), & usageCount));

// Write some values
config.Write(wxT("General/DataPath"), str))
config.Write(wxT("General/ToolTips"), useToolTips));
config.Write(wxT("General/Usage"), usageCount));
```

You can also iterate through groups and entries, query whether a group or entry exists, and delete an entry or group, among other operations.

wxConfig can be used as a temporary way of reading and writing data that you store elsewhere in your application, or you can create an instance of wxConfig that is not deleted until the application exits. wxWidgets has the concept of

Application Installation

A trouble-free installation goes a long way to reassure the user about your product before the application is even running. We'll deal with installation on Windows, Linux, and Mac OS X in turn. Information about where to get the third-party tools mentioned in this section is provided in Appendix E.

Installation on Windows

On Windows, an installer is pretty much necessary, both because users expect it and because of the way file associations and shortcuts need to be set up.

Because it's such a specialized area, wxWidgets doesn't attempt to provide an installation utility for your applications. Several installer creators are available, such as NSIS and InstallShield; a favorite with many developers is Inno Setup, a very capable, free installer that can be driven via a simple script or tailored using Pascal. Several third-party applications are listed on the Inno Setup web site for graphically creating a script that can be used to create an installer.

If you release your application frequently, you will probably want to automate the installer creation via a release script. In examples/chap20/install, we have provided example scripts for you to adapt for your own needs. Because they are Unix shell scripts, you will need MinGW's MSYS package, which is also supplied on the CD-ROM. Given a directory of files, makeinno.sh creates a portion of the Inno Setup script listing the subdirectories and files. It requires you to supply the hand-tailored top and bottom parts of the script specifying other details. You can invoke makeinno.sh with a command like this:

```
sh makeinno.sh c:/temp/imagedir innotop.txt innobott.txt myapps.iss
```

This will create the Inno Setup script myapp.iss from the files in c:/temp/imagedir.

You can adapt the release script makesetup.sh for building your own application's Windows installer. makesetup.sh copies required files into an "image" directory, which is used to build the installer script before invoking Inno Setup, creating the setup.exe file. This script uses variables specified in the file setup.var. You can add your own release functions, from building the application to copying files to your FTP site using the Curl utility.

When you distribute an application, don't forget to include a Windows XP "manifest" file with your application. The manifest is an XML file that tells Windows XP to apply theming to your application. You can include the manifest simply by including the wxWidgets standard resource file in your own .rc file:

```
aardvarkpro ICON aardvarkpro.ico
#include "wx/msw/wx.rc"
```

This includes a standard manifest file. If you want to include your own instead, define the wxUSE_NO_MANIFEST symbol before including wx.rc, and specify your own, as follows:

```
aardvarkpro ICON aardvarkpro.ico

#define wxUSE_NO_MANIFEST 1
#include "wx/msw/wx.rc"

1 24 "aardvark.manifest"
```

You may also simply include the manifest file alongside the application. For more information on the manifest syntax, see the file docs/msw/winxp.txt in your wxWidgets distribution.

Installation on Linux

Following UI Design Guidelines

It's worth researching the style guidelines for each platform. Much of the time, wxWidgets handles the differences, but there are some aspects that cannot be automated. For example, button layout differs between platforms. Apple is particularly strict about standard button ordering and spacing on Mac OS X. Here are just a few of the points to be aware of, both platform-specific rules and general observations. Also play with existing programs on different platforms for inspiration and to help you come up with an application design that is adaptable to your target platforms.

Standard Buttons

On Windows and Linux, groups of standard buttons can be centered or right justified, often in the order OK, Cancel, Help. On Mac OS X, however, the Help button (shown automatically as a question mark when using wxID_HELP) should be left justified, and the other standard buttons should be right justified with the default command as the right-most button: for example, ?, space, Cancel, OK.

Use the standard wxWidgets control identifiers whenever possible (such as wxID_OK, wxID_CLOSE, wxID_APPLY) because some ports (notably wxGTK) can map them to special buttons with appropriate graphics.

Refer to the section on "Platform-Adaptive Layouts" in Chapter 7, "Window Layout Using Sizers," for information about the wxStdDialogButton Sizer class, which can be used to position standard buttons appropriately on each platform.

Menus

A menu bar should have no "empty" menus. Careful attention should be paid to label capitalizationeach significant word normally has a capital first letterand providing mnemonics (such as &File) and shortcuts (such as Ctrl+O). Common commands should be supported where appropriate (such as Copy, Paste, Undo, and so on), and there should be neither a small number of very long menus, nor a very large number of smaller menus (nine or ten menus is probably the maximum reasonable number). Instead of supplying lots of separate menu items for different configuration operations, consider merging them into one command that invokes a tabbed dialog.

As with standard buttons, use standard identifiers where possible: in particular, use wxID_HELP, wxID_PREFERENCES, and wxID_HELP so that Mac OS X can move them to the application menu. You will need to design your menus so that removal of these items doesn't cause problems such as an empty Help menu or two consecutive separators.

Icons

Icons in your toolbar, frames, and other elements give a very strong impression of the quality of your application. Neglecting these can really let your application down. This is particularly true on Mac OS X, where there is a high standard of aesthetics. Consider having icons specially created, or a much cheaper alternative is to purchase packs of stock icons, perhaps getting the remaining non-standard icons designed for you in the same style. The investment will normally be well worth it, and your application will gain a stronger sense of identity. You can also find some icons on the webfor example, the L-GPL'ed Ximian collection at http://www.novell.com/coolsolutions/feature/1637.html.

Fonts and Colors

Don't be tempted to use a lot of different fonts and colors for your dialogs. As well as looking garish, using non-default fonts and colors makes it hard for wxWidgets to adapt to the current settings and themes to give a "native" look and feel. However, you can still allow the user to customize fonts for specific windows that present a lot of textual information, such as a report window. Refer to the platform guidelines on use of color. wxWidgets will do some adaptation here, but you may need to make further adaptations on custom windows that you implement.

Application Termination Behavior

Summary

This chapter covered a variety of topics for applying finishing touches to your application, and presented code to plug a few gaps left by wxWidgets. We've finished off with some tips and further reading to improve your awareness of UI design issues.

We hope that having read this book, you will agree with us that by adopting wxWidgets, you get a very powerful set of tools bringing you highly compelling benefits, including these:

- Your applications will have a native look and feel.
- The rich variety of classes, including simple and advanced controls, lightweight HTML functionality, wizards, online help, multithreading, interprocess communication, streams, virtual file systems, and so on, makes programming in wxWidgets highly productive and enjoyable.
- You save money by using the same code on all platforms.
- You can easily port your applications to platforms you might not have considered before, such as Pocket PC and Mac OS X, gaining market share in the process.
- By using rapid application development tools such as DialogBlocks, and the powerful sizer mechanism, you can quickly create complex, attractive, resizable and (above all) portable dialogs and other windows.
- wxWidgets' open source nature lets you modify the library and understand its workings.
- You get the advantages of the large wxWidgets community for quickly resolving problems and answering questions, plus the many third-party contributions (see Appendix E for a list of controls and tools).

We do hope that you've enjoyed reading the book and browsing the samples and tools on the CD-ROM and are now eager to apply what you've learned to your own cross-platform applications! Good luck, and we look forward to seeing you soon on the wxWidgets mailing lists and forum.

Appendix A. Installing wxWidgets

The process of installing wxWidgets has been made as simple as possible by using tools and techniques native to each platform. This appendix introduces the different methods for installing wxWidgets on various platforms with different compilers and discusses some of the main configuration options available. Only the most popular compilers and environments are covered in this appendix; Digital Mars C++, OpenWatcom C++, and Cygwin are all supported by wxWidgets but are not described here. However, you can find instructions for these compilers in the wxWidgets documentation under docs/msw.

If you want, you can download and unpack wxWidgets and then go straight to Appendix C, "Creating Applications with DialogBlocks," and use DialogBlocks to compile both wxWidgets and your application.

Choosing Your Development Tools

Before we discuss wxWidgets installation, let's briefly look at choices in compiler tools to use with wxWidgets. Note that although cross-compilation is possible (compiling on one platform to run on another), you will generally need to compile, debug, and test on each platform that you support. However, you probably have a favorite platform you prefer to develop on, just compiling and testing on the others. If you only have one machine for Windows and Linux development, consider using a tool such as the excellent VMware virtual machine software to run several operating systems simultaneously.

Tools on Windows

Microsoft's Visual Studio (see the section "WindowsMicrosoft Visual Studio") has a very good IDE, which makes debugging productive, and the compiler is good at optimizing executables for space and speed. It's also reasonably fast to run (though slower than Borland C++). This compiler is highly recommended for wxWidgets work, and it is the tool of choice for most wxWidgets developers on Windows.

Borland C++ (BC++) is a fast compiler, but the linker has trouble with wxWidgets executable sizes, and the compiler is not actively supported by Borland. In addition, there is no good free debugger for BC++. You can download Borland C++ from http://www.borland.com/products/downloads/download_cbuilder.html.

GCC is available in two forms on Windows: MinGW and Cygwin. MinGW makes use of the Windows run-time libraries and so is a better choice for "native" Windows applications. MinGW has an accompanying Unix-like shell, MSYS, which you will need if you want to use the configure method of compiling. MinGW can also compile wxWidgets and applications from the Windows command prompt using its own make tool and makefile.gcc makefiles. MinGW can be used on its own or with an IDE such as Dev-C++ or DialogBlocks. Download it from http://www.mingw.org or install it from the CD-ROM. Unfortunately, GCC is a slow compiler that creates huge libraries and executables, and its GDB debugger (or IDE equivalents) cannot compare with the convenience of VC++'s debugger.

Digital Mars C++ also works with wxWidgets and is quite fast; the IDE and debugger need to be purchased. Download Digital Mars C++ from http://www.digitalmars.com or copy it from the CD-ROM.

OpenWatcom C++ does not implement all C++ standards, but it works with wxWidgets and can be downloaded from http://www.openwatcom.org or copied from the CD-ROM.

CodeWarrior will also work with wxWidgets and provide a consistent environment for those using CodeWarrior on MacOS.

See also Appendix E, "Third-Party Tools for wxWidgets," for other IDEs and tools.

All the compilers mentioned here can be "driven" by DialogBlocks (on the accompanying CD-ROM), so you can design your dialogs and other user interface elements in addition to building and running your application. See Appendix C for more on DialogBlocks.

Tools on Linux and Mac OS X

On Linux, GCC is usually installed by default. However, you can use it in a number of different ways. You can use command-line tools (configure, make, and GDB for debugging), or you can use an IDE such as KDevelop, as described in this chapter (though you'll still compile the wxWidgets libraries from the command line). You also can use the command line to compile and then use GDB within Emacs to step through your application, or you can use the graphical debugger DDD.

On Mac OS X, you can download the GCC-based development tools from Apple's web site after registering as an Apple Developer Connection (ADC) member. You can use configure and make in exactly the same way as on Linux, but you can also use Apple's Xcode IDE to compile applications, again described later in this chapter. If you prefer,

Downloading and Unpacking wxWidgets

The latest version of wxWidgets can be downloaded from the wxWidgets web site, or you can use the version included on the CD-ROM. After you have downloaded the library source or copied it from the CD-ROM, you will need to extract the files to a location of your choice. Windows users also have the option of a self-extracting installer.

After you have unpacked wxWidgets, you may want to familiarize yourself with the organization of the files. The structure is similar to other open source projects, with separate directories for include files (include), source files (src), compiled library objects (lib), documentation (docs), sample programs (samples), and build information (build). Within some directories, there are further classifications of files based on which operating system(s) they are for:

- common: Used by all platforms, such as strings, streams, and database connectivity.
- expat: An XML parser library.
- generic: Widgets and dialogs implemented by wxWidgets for use where native versions are not available.
- gtk: For GTK+.
- html: For the HTML controls designed for wxWidgets.
- iodbc: The iODBClibrary for platforms without native ODBC support.
- jpeg: The JPEG library for platforms without native JPEG support.
- mac: For the Mac, further subdivided into classic (OS 9), carbon (OS 9 and OS X), and corefoundation (OS X).
- msw: For Microsoft Windows.
- png: The PNG image library for platforms without native PNG support.
- regex: A regular expression processor for platforms without native regular expression processing (or when Unicode is enabled).
- tiff: The TIFF image library for platforms without native TIFF support.
- unix: Used by all Unix-based platforms, including Linux and Mac OS X.
- xrc: wxWidgets' XML resource library.
-

Configuration/Build Options

wxWidgets can be built using almost countless permutations of configuration options. However, four main options are available on all platforms:

- Release versus Debug: Debug builds include debug information and extra checking code, whereas release builds omit debug information and enable some level of optimization. It is strongly recommended that you develop your application using a debug version of the library to benefit from the debug assertions that can help locate bugs and other problems.
- Unicode versus non-Unicode: As discussed in Chapter 16, "Writing International Applications," Unicode facilitates the display of non-ASCII characters such as Cyrillic, Greek, Korean, Japanese, and so on. wxWidgets natively supports Unicode; there is no penalty for using Unicode in terms of available features (with the exception of wxODBC on Unix), nor is any additional coding required. Windows 95, 98, and ME do not have built-in Unicode support, but wxWidgets can be configured to use the add-on MSLU library for these operating systems.
- Static versus Shared: A static version of the library is linked with the application at compile time, whereas a shared version is linked at runtime. Using a static library results in a larger application but also relieves the end user of the need for installing shared libraries on his or her computer. On Linux and Mac OS X, static libraries end in .a, and on Windows, they end in .lib. On Linux and Mac OS X, shared libraries end in .so, and on Windows, they end in .dll. Note that there are certain memory and performance implications of static and shared libraries that are not specific to wxWidgets; they are beyond the scope of this book.
- Multi-lib versus Monolithic: The entire wxWidgets library is made up of modules that separate the GUI functionality from the non-GUI functionality as well as separate advanced features that are not frequently needed. This separation enables you to use wxWidgets modules selectively depending on your specific needs. Alternatively, you can build wxWidgets into one "monolithic" library that contains all of the wxWidgets modules in one library file.

You can build and simultaneously use more than one version and/or configuration of wxWidgets. The way you access the different builds depends on your platform, but all library files are created using a standard naming convention: wx<library><version><u><d> where u and d indicate Unicode and debug, respectively. For example, the file wxbase26ud.lib is the Unicode debug base library for wxWidgets 2.6, and wxmsw26u_core contains the core GUI elements for a Unicode release build on Windows.

After building wxWidgets, you can confirm that the library is built correctly by compiling and running one of the samples. The samples are located in the samples folder of the main wxWidgets directory, with each sample in its own subfolder. You might try samples/minimal first because it's the simplest. For each compiler, directions for building one of the samples are included in docs/<platform>/install.txt. There are also more complete applications in the demos directory, and though they are not discussed here, they are built almost identically to the samples in most cases.

It is important to note that you can build wxWidgets with makefiles on any platform but still use an IDE to develop your projects. For example, you can use the Microsoft Visual C++ command-line compiler to build wxWidgets but still use Microsoft Visual Studio to create your applications.

WindowsMicrosoft Visual Studio

Many IDEs and compilers are available for Windows. The dominant IDE on Windows is Microsoft Visual Studio; wxWidgets requires at least Visual Studio 5. Microsoft has released Visual Studio 2005 Express for home enthusiasts and students, and it features all of the common Visual Studio tools, omitting only the high-powered tools typically used in large projects or a corporate environment. Although it is a free beta download at the time of writing, it will cost around $50 once it is officially released. See http://lab.msdn.microsoft.com/vs2005/.

You must have the Windows Platform SDK installed in order to build wxWidgets, which is a free download from http://www.microsoft.com/msdowload/platformsdk/sdkupdate.

When using the Platform SDK and the free compiler, install both the Core and Internet modules, and build wxWidgets with the run-time library linked statically (pass RUNTIME_LIBS=static to make or set Runtime Linking to Static in your DialogBlocks configurations). If using a project file, you will need to modify this setting manually, both in the wxWidgets and application project files. You may also need to remove the library odbc32.lib from the linker settings because the Platform SDK does not contain this library.

The SDK is already included with any of the professional versions of Visual Studio.

Regardless of which version of Visual Studio you are using, the process for building wxWidgets is the same. From your wxWidgets installation, open the wx.dsw workspace from the build\msw directory. If you are using a version later than 6.0, you may be prompted that the sub-projects must be converted to the current Visual C++ project format. Allow Visual Studio to proceed with the conversion; if you need the files in their original format, you can always extract the originals again. After all the projects in the workspace have been opened, you can browse the sources and classes.

If you are using Visual C++ version 7 or 8 (.NET or 2005), build wxWidgets by selecting Build Solution from the Build menu. Different solutions are used to build different library configurations, selectable from the Configuration Manager from the Build menu. Simply select which configuration you want to use, such as Debug, Unicode Release, or DLL Unicode Debug.

If you are using Visual Studio 5 or 6, build wxWidgets by selecting Batch Build from the Build menu and checking the desired library configurations. The wxWidgets project is set up to allow building release and debug versions of both Unicode and non-Unicode libraries, both static and shared (DLL). Ensure that you are building all of the sub-projects for the configurations that you want to use.

Most developers only need to build the release and debug libraries, or if you have a need for Unicode, the Unicode release and Unicode debug libraries. As a rule, using the static (non-DLL) libraries makes it easy to distribute a single-file application rather than needing to also distribute (or otherwise require) the correct wxWidgets DLL. Using the DLL builds can also result in quirks relating to application startup, so they should be used only when needed and only if you have a thorough understanding of how DLLs are utilized by both wxWidgets and Windows.

The compiled library files are placed into the lib directory under your wxWidgets directory. Two directories are created: vc_dll and vc_lib, for the shared and static builds, respectively.

If the process of building via project files seems fiddly (the number of configurations in the project file can be confusing!), consider using the command-line alternative as described in the section "WindowsMicrosoft Visual C++ Command Line," or build wxWidgets via DialogBlocks.

Compiling a wxWidgets Sample Program

Workspace and project files are included for every wxWidgets sample. Use Visual Studio to open a workspace for one of the samples (in the samples directory within your wxWidgets installation), and then select a configuration matching any of the wxWidgets library builds that you compiled. For example, if you have built the Unicode debug library, compile the sample using the Unicode debug build. The samples are created to look within the wxWidgets tree for the include and library files in their default locations. As long as the library built successfully and the structure

WindowsMicrosoft Visual C++ Command Line

Microsoft's C++ compiler is part of any Visual C++ installation (or can be downloaded free from Microsoft) and can be used from the command line. The Microsoft compiler makefile is in the build\msw directory. From there, invoke the Microsoft compiler using a command such as the following ones, which demonstrate how to toggle all of the major configuration options:

```
nmake -f makefile.vc UNICODE=0 SHARED=0 BUILD=release MONOLITHIC=0
nmake -f makefile.vc UNICODE=0 SHARED=0 BUILD=debug MONOLITHIC=0
```

The compiled library files are placed into the lib directory under your wxWidgets directory. Two directories are created: vc_dll and vc_lib, for the shared and static builds, respectively.

If you need ODBC or OpenGL functionality, set wxUSE_ODBC or wxUSE_GLCANVAS to 1 in include\wx\msw\setup.h before compiling, and also pass USE_ODBC=1 or USE_OPENGL=1 on the nmake command line.

To remove the object files and libraries, append the target clean to the same command line. However, be aware that this won't clean up the copy of setup.h that is placed under the vc_lib and vc_dll directories, so if you make any edits to include\wx\msw\setup.h, delete the vc_lib or vc_dll directories yourself before recompiling.

Compiling a wxWidgets Sample Program

If you change into one of the sample program directories, you will see a makefile for Microsoft's compiler called makefile.vc. The sample programs are built using the same command and switches as the library itself, and they are designed to look within the wxWidgets tree for the include and library files in their default locations. Be sure that you specify flags for a configuration of the library that has been built, or the sample will fail to link. That is, if you have only built a Unicode static release version of wxWidgets, use the same build options for the sample. For example:

```
cd samples\widgets
nmake -f makefile.vc UNICODE=1 SHARED=0 BUILD=release MONOLITHIC=0
vc_mswd\widgets.exe
```

As with the library makefile, you can clean a sample by appending the clean target, for example:

```
nmake -f makefile.vc UNICODE=0 SHARED=0 BUILD=debug clean
```

WindowsBorland C++

Borland's command-line C++ compiler (BC++) is a free download, but it is also part of Borland's professional C++Builder IDE. Building wxWidgets with BC++ is done from the command line, using flags to the compiler to specify the library build configuration. The BC++ makefile is in the build\msw directory. From there, invoke the BC++ compiler using a command such as the following, which demonstrates how to toggle all of the major build options:

```
make -f makefile.bcc UNICODE=0 SHARED=0 BUILD=release MONOLITHIC=0
```

The compiled library files are placed into the lib directory under your wxWidgets directory. Two directories are created: bcc_dll and bcc_lib, for the shared and static builds, respectively.

To remove the object files and libraries, append the target clean to the same command line. However, be aware that this won't clean up the copy of setup.h that is placed under the bcc_lib and bcc_dll directories, so if you make any edits to include\wx\msw\setup.h, delete the bcc_lib or bcc_dll directories yourself before recompiling.

Compiling a wxWidgets Sample Program

If you change into one of the sample program directories, you will see a makefile for BC++ called makefile.bcc. The sample programs are built using the same command and switches as the library itself and are designed to look within the wxWidgets tree for the include and library files in their default locations. Be sure that you specify flags for a configuration of the library that has been built, or the sample will fail to link. That is, if you have only built a Unicode static release version of wxWidgets, use the same build options for the sample. For example:

```
cd samples\widgets
make -f makefile.bcc UNICODE=1 SHARED=0 BUILD=release MONOLITHIC=0
bcc_mswd\widgets.exe
```

As with the library makefile, you can clean a sample by appending the clean target, for example:

```
make -f makefile.bcc UNICODE=0 SHARED=0 BUILD=debug clean
```

WindowsMinGW with MSYS

MinGW is a GNU toolset that includes Windows headers and import libraries for building Windows applications. MSYS provides an environment closely resembling a Linux or Unix shell in which to use MinGW. The process for using MinGW is very similar to the process described later for GCC.

It is recommended that you create a subdirectory within the wxWidgets directory for each configuration that you want to build. Most developers create directories using the same naming conventions as wxWidgets; if you want to compile the library in Unicode debug mode, you might create a directory named buildud or build26ud depending on how many different configurations you want to have.

After you have created your build directory, change into the build directory. The configure script, now located one directory up, takes many different parameters, the most important ones being those that control the main build options. For example, you could run

```
../configure --enable-unicode --disable-debug --disable-shared --disable-monolithic
```

The configure script will analyze the build environment and then generate the makefiles for use with MinGW. When configure has finished, it will display a summary of the library build configuration.

```
Configured wxWidgets 2.6.0 for `i686-pc-mingw32'

  Which GUI toolkit should wxWidgets use?                msw
  Should wxWidgets be compiled into single library?      no
  Should wxWidgets be compiled in debug mode?            no
  Should wxWidgets be linked as a shared library?        no
  Should wxWidgets be compiled in Unicode mode?          yes
  What level of wxWidgets compatibility should be enabled?
                                     wxWidgets 2.2       no
                                     wxWidgets 2.4       yes
  Which libraries should wxWidgets use?
                                     jpeg                builtin
                                     png                 builtin
                                     regex               builtin
                                     tiff                builtin
                                     zlib                builtin
                                     odbc                no
                                     expat               builtin
                                     libmspack           no
                                     sdl                 no
```

Many other individual features can be enabled or disabled using the configure script, a list of which can be found in the wxWidgets documentation or by passing --help to the configure script. After configure has generated the makefiles, build the library by running make in the same directory as the one in which you ran configure. The compiled library files are placed in the lib subdirectory of the directory used for the build, not the lib directory at the root of the wxWidgets tree. You may optionally use make install as root after the build has completed, to copy the library and the necessary headers into /usr/local so that all users may have access to compile, build, and run wxWidgets programs.

Compiling a wxWidgets Sample Program

The configure script creates a samples directory in your build directory, with further subdirectories for each sample. If you change into one of the sample program directories, you will see a makefile, which has been generated for your build directory and build configuration. Run make to build the sample. For example:

```
cd /c/wx/build26ud/samples/minimal
make
./minimal
```

WindowsMinGW without MSYS

MinGW can also be used from Microsoft's command line (cmd.exe or command.com). You can control your build style by using flags to the compiler. The MinGW makefile is in the build\msw directory. From there, invoke MinGW using a command such as the following, which demonstrates how to toggle all of the major build options:

```
mingw32-make -f makefile.gcc UNICODE=1 SHARED=0 BUILD=release \
    MONOLITHIC=0
```

(The backslash is only there to denote that the command should be typed all on one line.)

The compiled library files are placed into the lib directory under your wxWidgets directory. Two directories are created: gcc_dll and gcc_lib, for the shared and static builds, respectively.

To remove the object files and libraries, append the target clean to the same command line. However, be aware that this won't clean up the copy of setup.h that is placed under the gcc_lib and gcc_dll directories, so if you make any edits to include\wx\msw\setup.h, delete the gcc_lib or gcc_dll directories yourself before recompiling.

Note

If you are using MinGW without MSYS but have MSYS installed, the MSYS directories should not be in your path; otherwise, the wrong version of make could be used. If the wrong version of make is used, you will get cryptic error messages for processes that are otherwise correct. This conflict occurs because MinGW and MSYS share common file names for non-identical files that are not interchangeable.

Compiling a wxWidgets Sample Program

If you change into one of these sample program directories, you will see a makefile for MinGW called makefile.gcc. The sample programs are built using the same command and switches as the library itself and are designed to look within the wxWidgets tree for the include and library files in their default locations. Be sure that you specify flags for a configuration of the library that has been built, or the sample will fail to link. That is, if you have only built a Unicode static release version of wxWidgets, use the same build options for the sample. For example:

```
cd c:\wx\samples\minimal
mingw32-make -f makefile.gcc UNICODE=1 SHARED=0 BUILD=release \
    MONOLITHIC=0
minimal.exe
```

(The backslash is only there to denote that the command should be typed all on one line.)

To clean the sample, append clean to the make command.

Linux, Unix, and Mac OS XGCC

GCC is the de facto standard compiler on Linux and many Unix platforms, including Darwin, the BSD foundation for Mac OS X. Building wxWidgets with GCC simply requires following the typical configure and make routine common on Unix and Unix-like environments.

It is recommended that you create a subdirectory within the wxWidgets directory for each different configuration that you want to build. Most developers create directories using the same naming conventions as wxWidgets; if you want to compile the library in Unicode debug mode, you might create a directory named buildud, build26ud, or even buildGTK26ud depending on how many different configurations you want to have.

After you have created your build directory, change into the build directory. The configure script, now located one directory up, takes many different parameters, the most important ones being those that control the main build options. For example, you could run

```
../configure --enable-unicode --disable-debug --disable-shared \
    --disable-monolithic
```

(The backslash is only there to denote that the command should be typed all on one line.)

The configure script will analyze the build environment and then generate the makefiles for use with GCC. When configure has finished, it will display a summary of the library build configuration. The previous configure options would produce the following summary on Linux:

```
Configured wxWidgets 2.6.0 for `i686-pc-linux-gnu'

  Which GUI toolkit should wxWidgets use?                GTK+ 2
  Should wxWidgets be compiled into single library?      no
  Should wxWidgets be compiled in debug mode?            no
  Should wxWidgets be linked as a shared library?        no
  Should wxWidgets be compiled in Unicode mode?          yes
  What level of wxWidgets compatibility should be enabled?
                                     wxWidgets 2.2       no
                                     wxWidgets 2.4       yes
  Which libraries should wxWidgets use?
                                     jpeg                sys
                                     png                 sys
                                     regex               builtin
                                     tiff                sys
                                     zlib                sys
                                     odbc                no
                                     expat               sys
                                     libmspack           no
                                     sdl                 no
```

Many other individual features can be enabled or disabled using the configure script, a list of which can be found in the wxWidgets documentation or by passing --help to the configure script. Most significantly, you can also choose to override the default GUI toolkit (GTK+ 2) with GTK+, Motif, or X11.

After configure has generated the makefiles, build the library by running make in the same directory as you ran configure. The compiled library files are placed in the lib subdirectory of the directory used for the build, not the lib directory at the root of the wxWidgets tree. You may optionally use make install as root after the make has completed to copy the library and the necessary headers into /usr/local so that all users may have access to compile, build, and run wxWidgets programs.

Compiling a wxWidgets Sample Program

The configure script creates a samples directory in your build directory, with further subdirectories for each sample. If

Modifying Setup.h for Further Customizations

If you want to further customize your wxWidgets library by enabling or disabling certain features, all of the configuration options are centralized into a file called setup.h. A separate setup.h file is automatically created from the default setup.h file for each library build and is placed in the lib directory of that build. The exact subdirectory depends on the compiler. For compilers using configure and make, setup.h is in lib/wx/include/<configname>/wx from the build directory. For other compilers, setup.h is in lib/<compiler>_lib/ <configname>/wx.

setup.h mostly contains a long list of wxUSE_... defines. For example, #define wxUSE_THREADS 1 indicates to build the library with support for threads using the wxTHRead class. If you change the 1 to a 0, wxTHRead will no longer be compiled into the library, and you will be unable to build programs that use wxTHRead. By enabling only the features that you need, you can build your own smaller, customized library. Most features are enabled by default, but some specialized features, such as ODBC, must be enabled if you want to use them. The setup.h file is heavily commented, pointing out possible side effects of enabling or disabling certain key features. Note that a change to setup.h requires recompiling the entire library because setup.h is at the very top of the wxWidgets include chain. On Windows, you may also need to pass extra options to the command line, such as USE_OPENGL=1 or USE_ODBC=1.

On Windows, there is a common setup.h in include/wx/msw that is copied, the first time the library is built, to the library configuration's build directory. Changing the setup.h in include/wx/msw will result in changes for all configurations, whereas changing the setup.h in the lib/XX_lib directory will only change that one configuration. This does not apply if you are using configure and make under MinGW/MSYS, which creates the setup.h file as part of the configure process.

Rebuilding After Updating wxWidgets Files

Changes to Source or Header Files

There may be times that you change the wxWidgets sources, perhaps to apply some code that fixes a bug or adds a new feature that you need. If you change only source files, you can simply rebuild the library using the same commands used to build it originally. However, if you are rebuilding after changing header files, you may need to do some additional cleaning in order for wxWidgets to build correctly due to precompiled headers that may not be automatically updated to reflect the latest changes to the header files.

- Microsoft Visual Studio: After changing header files, you will want to rebuild the library rather than just build it, forcing any precompiled headers to be discarded and re-created.
- GCC or MinGW using configure: Remove the .deps directory before rebuilding. This one is easy to miss because the ls command does not list the directory by default.

If you continue to receive errors that indicate that an old header file is being used (for example, the compiler can't find a function that was just added and that did not exist the first time you built wxWidgets), you may need to completely remove the build directory and rebuild the library. The build directories should not contain anything besides object files and compiled library files, so you won't lose any data related to your projects.

Changes to setup.h on Windows

Whenever you change the common setup.h (in include/wx/msw), it is very important that it be properly copied to its installed location in the lib subdirectory. If the setup.h being used to build your applications is out of sync with the setup.h used to compile the library, you will almost certainly receive link errors due to missing symbols. Although not hard to do, it is an easy step to overlook. If you are using configure and make under MSYS or MinGW, you can re-run configure. Otherwise, you will need to delete all of the setup.h files for each library build configuration in the lib subdirectory. Because the whole library needs to be rebuilt anyway, it is just as easy to delete the entire XX_lib or XX_dll directory.

Using Contrib Libraries

Included with wxWidgets is a variety of contributed libraries, found in the contrib subdirectory of your wxWidgets distribution. These are libraries that are distributed with wxWidgets but are not part of the core toolkit. Although they are not officially supported and may not be actively maintained by the core wxWidgets developers, they can still be very useful. It's easy to build the contrib libraries and use them in your projects.

You can build one or all of the contrib libraries using the same process as building the samples, discussed briefly with each compiler earlier. For Windows compilers, the libraries are compiled from within the contrib/ build/<name> subdirectories. Compiled libraries are automatically placed alongside the main wxWidgets library files, eliminating the need to add any additional directories to the link path. You do, however, need to add contrib/include to your include path.

For descriptions of the more important contrib libraries, such as wxStyledTextCtrl, see Appendix D, "Other Features in wxWidgets."

Appendix B. Building Your Own wxWidgets Applications

This appendix shows you how to use Visual Studio for Windows, KDevelop for Linux, and Xcode for Mac OS X to create your own projects in an IDE. There is also a basic discussion of makefiles that applies to any compiler on any platform, a description of the Bakefile makefile-generation system, information on using the wx-config command, and a list of wxWidgets preprocessor symbols you will find useful when writing your applications.

If you're new to wxWidgets, we recommend that you initially skip this chapter and use DialogBlocks to get started with compiling your own application, as detailed in Appendix C, "Creating Applications with DialogBlocks."

If your favorite compiler or IDE is not detailed here, or if you just want the quick overview, there are essentially three major sets of files required to build a project using wxWidgets:

1. Include files. The wxWidgets headers must be available to the compiler when compiling your project; otherwise, none of the wxWidgets classes are declared.
2. Library files. The wxWidgets compiled library must be available to link with your compiled program.
3. System files. Whatever system libraries are needed to link for the platform must be linked in with your program.

Important: You must have compiled and built at least one configuration of the wxWidgets library, as covered in Appendix A, "Installing wxWidgets," prior to proceeding with creating your own wxWidgets projects.

WindowsMicrosoft Visual Studio

Before covering the step-by-step directions for creating wxWidgets projects with Visual Studio, it should be noted that there are alternatives for generating Visual Studio project files using third-party tools. Although not covered here, utilities like DialogBlocks, wxHatch, wxWinWizard, and wxVisualSetup can be used to generate project files that can be loaded and used in Visual Studio. If you want detailed control over your project or need to make specific modifications, however, it may be necessary to have an understanding of the individual settings.

Microsoft Visual Studio enables you to quickly and easily build wxWidgets projects by adding just a few wxWidgets directories and files into your project settings. Although the exact location for each setting is slightly different in each version of Visual Studio, the options are named roughly the same; please look at nearby options or consider alternate wordings if the setting isn't listed exactly as specified here.

1. Create a new Win32 Project or Solution. Choose the most minimal options available.

2. Open the Project Settings/Properties dialog.

3. Find the Preprocessor Definitions under the C/C++ Settings. You will want to be sure that all of the following are defined: WIN32, __WXMSW__, _WINDOWS. If you are using a debug version of wxWidgets to build your project, you will also need to define _DEBUG, __WXDEBUG__. Note that some versions of Visual Studio will not show the C/C++ options until a C or C++ source file has been added to the project.

4. Find the Additional Include Directories under the C/C++ settings and enter both the core include directory (such as c:\wxWidgets\include) and the include directory for the library configuration that you are using (such as c:\wxWidgets\lib\vc_lib\mswu). This configuration-specific include directory must be included for the compiler to find the setup.h that specifies what features are enabled for that build.

5. Find the Additional Libraries Directories under the Link Settings and enter the directory where the compiled wxWidgets library files are located (such as c:\wxWidgets\lib\vc_lib).

6. Find the Additional Dependencies or Object Modules setting to specify additional libraries to link with your application. The wxWidgets library names will vary depending on your build configuration but will follow the library naming conventions outlined in the installation appendix. For example, for the debug build, you would need to link at least wxmsw26d_core.lib and wxbase26d.lib. Depending on which wxWidgets features you are using, you may need to link with more of the libraries: wxbase26d_net.lib, wxbase26d_odbc.lib, wxexpatd.lib, wxjpegd.lib, wxmsw26d_adv.lib, wxmsw26d_grid.lib, wxmsw26_gl.lib, wxmsw26d_html.lib, wxmsw26d_xrc.lib, wxmsw26d_xml, wxpngd.lib, wxregexd.lib, wxtiffd.lib, and wxzlibd.lib. Again, remember the build configuration postfixes for the build you are using. You can always look at the compiled library files for a complete list of libraries.

7. The Win32 libraries that you need to link, depending on your build configuration and what features you have enabled, could include some or all of the following: kernel32.lib, user32.lib, gdi32.lib, winspool.lib, comdlg32.lib, advapi32.lib, shell32.lib, ole32.lib, oleaut32.lib, uuid.lib, odbc32.lib, odbccp32.lib, winmm.lib, comctl32.lib, rpcrt4.lib, and wsock32.lib. Visual Studio usually adds some or all of these libraries when the project is created. Not all libraries are always needed; for example, if you do not use sockets, you would not need to link wsock32.lib.

8. Find the Run-time Library selection from the C/C++ Code Generation settings and select either Multithread DLL or Multithread Debug DLL. You will not be able to link your program using the single-thread libraries or the non-DLL libraries. If you see a handful of linking errors about Windows symbols already being defined, you probably did not select a DLL run-time library.

9. You can now add your source files to the project, if you have not already done so, and build your wxWidgets application.

LinuxKDevelop

KDevelop, a subproject of the KDE desktop environment, is arguably the most robust, mature, and stable IDE for developing applications on Linux. Not only does KDevelop provide a powerful editor, but it also generates and updates your makefiles for you, saving you a considerable amount of work. You only need to add a few settings to KDevelop's project settings for it to compile your wxWidgets application.

1. Create a New Project (from the Project menu).

2. Expand C++ and choose Simple Hello World Program or some other minimal project setting. Some newer versions of KDevelop have a wxWidgets project option, but it is not as reliable or as flexible as setting up the project yourself. Finish creating the project in a place of your choosing.

3. Delete any source files created by KDevelop, most likely a simple source file with a main and little else.

4. Open the Project Options from the Project menu and select Configure Options from the pane of options on the left. You should see several tabs, including General, C, and C++.

5. Click on the C++ tab. Paste the results of using wx-config --cxxflags from the command line (see the later section "Using wx-config"), clearing any options that may already be present. These flags are the necessary header includes and defines to build your wxWidgets application.

6. Click on the General tab. Paste the results of using wx-config --libs from the command line. These are the necessary libraries for linking wxWidgets as well as linking the necessary X11, GTK+, and other system libraries.

7. After clicking OK to close the project options dialog, KDevelop will prompt you to re-run configure for this build configuration. Although this is a necessary step later, do not run configure yet.

8. You can now add your wxWidgets source files to the project. You will need to add at least one source file to the project before proceeding to the next step. You can add sources by using the Automake Manager tab. If you haven't written any code yet, begin your application and then come back to these steps, even if you only write a skeletal program that has a wxApp class and little else.

9. Run automake by selecting Run Automake & Friends from the Build menu.

10. Run configure by selecting Run Configure from the Build menu.

11. You can now build your wxWidgets application. As your application grows, simply add the new files using the Automake Manager, and KDevelop will automatically update the makefiles.

KDevelop allows you to create multiple build configurations. By default, configurations named "debug" and "optimized" are created. If you want to be able to create both debug and release versions of your application, you can place the wx-config flags specifying the different library configurations into separate KDevelop configurations, giving you the flexibility of choosing to build a debug or a release simply by switching your configuration from within KDevelop.

Mac OS XXcode

Apple makes it easy for developers to create Mac OS X applications by providing Xcode, a free IDE and front-end for GCC. Xcode is a rich IDE, and this short guide is not intended to replace books or other resources on all of Xcode's features and options. Fortunately, it takes only a few steps to create an Xcode project and add the necessary build flags for wxWidgets.

1. Create a New Project (File menu).
2. From the New Project Assistant, select Empty Project for the project type.
3. Select the location for your project files.
4. From the Project menu, select New Target and choose Carbon/Application. This will tell Xcode that you are creating an application. You will be asked to provide a name for the target. The Target Info window will then appear, allowing you to change the configuration for the new target.
5. From the Settings drop-down menu, choose Language under GNU C/C++ Compiler, and find the Other C++ Flags option. This is where you will paste the results of using wx-config --cxxflags from the command line (see the later section "Using wx-config"). These flags are the necessary header includes and defines to build your wxWidgets application.
6. From the settings drop-down menu, choose Linking under General, and find the Other Linker Flags option. This is where you will paste the results of using wx-config --libs from the command line. The libs are the necessary libraries for linking wxWidgets as well as linking the necessary Mac OS X system files.
7. You can now add your wxWidgets source files to the project by selecting Add to Project from the Project menu. Because you have provided Xcode with the necessary flags from wx-config, your wxWidgets programs will compile and link right from within Xcode.

Your Xcode project will also contain a file matching your project's name, ending in .plist. This is an XML file that contains information about your application and is included in your application bundle. On Mac OS X, every application is actually an application bundle, a complete directory with a hierarchy of files that are part of the application. This allows Mac OS X programs to be easily copied and moved as a single icon from Finder while still giving developers a chance to include any needed auxiliary files. For example, interface translations can be a part of the application bundle, so the application can be shown in the user's native language without downloading any additional files.

Xcode allows you to specify multiple targets and multiple build styles for each target. For example, Xcode automatically creates deployment and development build styles for each target. If you want to be able to create both debug and release versions of your application, you could place the wx-config flags specifying the different library configurations into the build styles rather than the target, giving you the flexibility of choosing to build a debug or a release simply by switching your build style.

Any PlatformMakefiles

Makefiles are available when building with almost any compiler and are largely standardized. Makefile syntax is beyond the scope of this book, but to get started quickly with makefiles, you can create a DialogBlocks project, add suitable configurations and some source files, and select Generate Makefile from the Build menu. Invoke the resulting makefile from the command line with CONFIG=<config> where <config> is one of the configurations listed when you invoke the makefile with the help target.

If you built wxWidgets with configure, your application makefile is likely to contain references to wx-config to supply the correct flags (see the later section "Using wx-config"). For example:

```
CC = gcc

minimal: minimal.o
	$(CC) -o minimal minimal.o `wx-config libs`

minimal.o: minimal.cpp mondrian.xpm
	$(CC) `wx-config cxxflags` -c minimal.cpp -o minimal.o

clean:
	rm -f *.o minimal
```

Modifying makefiles by hand can quickly become tedious as your project grows. The next section describes how to use Bakefile to generate makefiles for all platforms.

Cross-Platform Builds Using Bakefile

Maintaining a large number of different project files and formats can quickly become overwhelming. To simplify the maintenance of these formats, Vaclav Slavik created Bakefile, an XML-based makefile wrapper that generates all the native project files for wxWidgets. So now, even though wxWidgets supports all these formats, wxWidgets developers need only update one filethe bakefileand Bakefile handles the rest. Fortunately, Bakefile isn't specific to wxWidgets in any wayyou can use Bakefile for your own projects.

Note that this tutorial assumes that you are familiar with how to build software using one of the supported makefile systems, that you have some basic familiarity with how makefiles work, and that you are capable of setting environment variables on your platform. Also note that the terms "Unix" and "Unix-based" refer to all operating systems that share a Unix heritage, including FreeBSD, Linux, Mac OS X, and various other operating systems.

Getting Started

First, you'll need to install Bakefile. You can always find the latest version for download at Bakefile's web site: http://bakefile.sf.net.

It is also available on the CD-ROM included with this book. A binary installer is provided for Windows users, whereas users of Unix-based operating systems will need to unpack the tarball and run Packages for some distributions are also available; check the web site for details.

```
configure && make && make install
```

Setting Up Your wxWidgets Build Environment

Before you can build wxWidgets software using Bakefile or any other build system, you need to make sure that wxWidgets is built and that wxWidgets projects can find the wxWidgets includes and library files. wxWidgets build instructions can be found by going to the docs subfolder, then looking for the subfolder that corresponds to your platform (such as msw, gtk, mac) and reading install.txt there. After you've done that, the following sections provide some extra steps you should take to make sure your Bakefile projects work with wxWidgets.

On Windows

After you've built wxWidgets, you should create an environment variable named WXWIN and set it to the home folder of your wxWidgets source tree. (If you use the command line to build, you can also set or override WXWIN at build time by passing it in as an option to your makefile.)

On Unix and Mac OS X

In a standard install, you need not do anything as long as wx-config is on your PATH. wx-config is all you need; see "Using wx-config" later in this appendix.

A Sample wxWidgets Project Bakefile

Now that everything is set up, it's time to take Bakefile for a test run. It is recommended that you use the wxWidgets sample bakefile to get started. It can be found in the build/bakefiles/wxpresets/sample directory in the wxWidgets source tree. Here is the minimal.bkl bakefile used in the sample:

```
<?xml version="1.0" ?>
<! $Id: minimal.bkl,v 1.1 2005/01/27 22:47:37 VS Exp $ >

<makefile>

    <include file="presets/wx.bkl"/>
```

Using wx-config

When you build wxWidgets using configure and make, wxWidgets creates a special script called wx-config, which produces the necessary compiler flags for compiling and linking wxWidgets programs, and manages returning the correct flags when you have multiple versions of the wxWidgets libraries installed in the same location (such as /usr/local). You can use wx-config from makefiles or run it by hand to see what flags you need to insert into your IDE settings. If you're using Visual Studio, BC++, or MinGW without MSYS, you won't use wx-config.

If you look in your wxWidgets build directory, you will see wx-config there, and also in /usr/local/bin if you ran make install.

The flags that produce the necessary compile and link settings are --cxxflags and --libs, respectively. For example, on Mac OS X, the output of each could be

```
$ wx-config --cxxflags
-I/usr/local/lib/wx/include/mac-unicode-release-static-2.6
-I/usr/local/include/wx-2.6 -D__WXMAC__ -D_FILE_OFFSET_BITS=64
-D_LARGE_FILES -DWX_PRECOMP -DNO_GCC_PRAGMA

$ wx-config --libs
-L/usr/local/lib   -framework QuickTime -framework IOKit -framework Carbon
-framework Cocoa -framework System  /usr/local/lib/libwx_macu_xrc-2.6.a
/usr/local/lib/libwx_macu_html-2.6.a /usr/local/lib/libwx_macu_adv-2.6.a
/usr/local/lib/libwx_macu_core-2.6.a
/usr/local/lib/libwx_base_carbonu_xml-2.6.a
/usr/local/lib/libwx_base_carbonu_net-2.6.a
/usr/local/lib/libwx_base_carbonu-2.6.a -framework WebKit -lexpat -lz
-lpthread -liconv -lwxregexu-2.6 -lwxtiff-2.6 -lwxjpeg-2.6-lwxpng-2.6
```

Without passing these flags to the compiler and linker when building your own applications, you are likely to receive hundreds of errors because the compiler won't know anything about the wxWidgets classes that your program is using.

Using wx-config from the Build Directory

If you did not install the wxWidgets files into /usr/local, you will need to use wx-config "in-place," meaning that it will produce absolute paths to that build's files right from the build directory. This should be done by passing the ---inplace flag to wx-config with whatever flags you are requesting. Even if you did run make install, nothing prevents you from using the in-place flag with a particular build of wxWidgets. When running in-place, you do not need to pass any build configuration parameters because an in-place wx-config knows only about one build, the build from which you are running wx-config in-place.

Using wx-config from /usr/local and Choosing Your Configuration

When you have built and installed multiple configurations of wxWidgets, you can specify to wx-config which configuration's build flags you would like returned. For example, if you are creating both debug and release builds of your project, you should be sure to get the correct flags for each configuration from wx-config, for example:

```
wx-config --debug=no
```

or

```
wx-config --debug=yes
```

Many options are available to choose specific build configurations. The most commonly used flags are

[View full width]

wxWidgets Symbols and Headers

Although wxWidgets defines a lot of symbols, there is only a handful that you are likely to need to use in your projects. Sometimes, it may be necessary to execute certain code only on certain platforms or under certain conditions, such as the following:

```
#ifdef __WXMAC__
// Do something Mac-only
#endif
```

For your convenience, Table B-4 lists the common symbols and when they are defined. Additional symbols may be defined by the ports not covered in this book (such OS/2, Palm, and Cocoa).

Table B-4. Platform and Toolkit Symbols

Platform and Toolkit Symbols	
__WXMSW__	Microsoft Windows
__WXWINCE__	Microsoft Windows CE
__WXMAC__	Mac OS X
__WXGTK__	Using the GTK library, version 1 or 2
__WXGTK20__	Using the GTK library, version 2
__UNIX__	Unix-based platform (for example, Linux, Mac OS X, HP-UX)
__DARWIN__	Open-source BSD variant used by Mac OS X
_LINUX__	Any Linux-based platform
Library / Build Options	
wxUSE_UNICODE	Enable string Unicode support.
__WXDEBUG__	Library compiled with debugging support.
WX_PRECOMP	Use pre-compiled headers.

The use of defined symbols ties in very closely to wxWidgets' directory structure, as discussed in Appendix A. Consider for a moment that you can include a single file, regardless of the target platform, and yet wxWidgets always uses the correct platform information. Nearly all of the header files in include/wx have a block like the following (taken from combobox.h):

```
#if defined(  WXUNIVERSAL  )
```

Appendix C. Creating Applications with DialogBlocks

As we noted in Chapter 9, "Creating Custom Dialogs," when creating your own dialogs, you really need a resource editor to take the pain out of tasks such as laying out the controls and maintaining event tables. DialogBlocks from Anthemion Software is one such tool, and you'll find a special version of DialogBlocks on the accompanying CD-ROM to accelerate your wxWidgets learning and development on Windows, Linux, or Mac OS X. This appendix describes the tool and how you can use it to create, compile, and run your own applications. If you use DialogBlocks, you can avoid manually performing most of the procedures detailed in the previous two appendices because the tool can compile both wxWidgets and your own application.

What is DialogBlocks?

DialogBlocks is a Rapid Application Development (RAD) tool that can generate XRC or C++ code for your dialogs and frames. It can also generate a skeleton application and makefile and even compile your code using a range of popular compilers. DialogBlocks can invoke the GDB debugger for simple debugging with the GCC compiler, but it's recommended that you use a more sophisticated debugger for intensive use.

When DialogBlocks generates code, it inserts special comments, and it is free to replace code between these comments. You can edit the file outside these blocks, either using the DialogBlocks source editor or in an external editor or IDE. When you switch back to DialogBlocks from another application, DialogBlocks will prompt you to reload changed files as appropriate.

Unlike other dialog editor tools you may be used to, DialogBlocks doesn't support drag and drop for placing elements. Instead, you place elements inside sizers, which know how to lay out their children. So to lay out three buttons in a row, for example, you create a horizontal box sizer and insert three buttons. You build up a hierarchy of sizers and controls, and this provides the portability, adaptation to translations, and resizing ability described in Chapter 7, "Window Layout Using Sizers." If it feels a little odd at first, persevereafter a short while, it will "click," and then you'll find it a very powerful and easy way to build complex and attractive dialogs.

DialogBlocks Personal Edition has the following restrictions: wizards are limited to four pages, only one "custom control definition" can be created, there is the occasional nag screen, Windows RC file import is disabled, and the tool is for personal rather than commercial use. The full version of DialogBlocks can be purchased from http://www.anthemion.co.uk/dialogblocks.

Installing DialogBlocks

Windows

Just run the setup program in the DialogBlocks directory on the CD-ROM (or via the CD-ROM's HTML interface) and follow the prompts. Run DialogBlocks from its menu group or the desktop icon. To uninstall, use the Uninstall DialogBlocks icon in the DialogBlocks program group, or you can uninstall via the Control Panel.

Linux

Unarchive the tarred, gzipped file to a suitable location in your file system. A directory of the form DialogBlocks-x.xx (where x.xx is the version number) will be created.

Add the location to your PATH variable and run the dialogblocks command. You will need to set the environment variable DIALOGBLOCKSDIR so that DialogBlocks can find its data files.

For example:

```
% cd ~
% tar xvfz DialogBlocks-1.90.tar.gz
% export DIALOGBLOCKSDIR=`pwd`/DialogBlocks-1.90
% export PATH=$PATH:$DIALOGBLOCKSDIR
% dialogblocks
```

If you don't want to change your PATH, you could place a script in a location already on your path, such as /usr/local/bin. For example:

```
#!/bin/sh
# Invokes DialogBlocks
export DIALOGBLOCKSDIR=/home/mydir/DialogBlocks-1.90
$DIALOGBLOCKSDIR/dialogblocks $*
```

To uninstall, delete the DialogBlocks folder.

Mac OS X

Clicking on the dmg file will mount the virtual disk containing the DialogBlocks folder. To install, simply drag this folder to a suitable location on your hard disk. To uninstall, drag the folder to the trash can.

Upgrading DialogBlocks

You can get bug fixes for DialogBlocks Personal Edition simply by uninstalling DialogBlocks and reinstalling the latest download from the DialogBlocks web site. The registration information is not removed when uninstalling, so a newer version will still run in Personal Edition mode. You need to install from the CD-ROM and run DialogBlocks once to make sure the registration information is available for subsequent upgrades.

The DialogBlocks Interface

When you run DialogBlocks, you see a project tree on the left, various content windows on the right including the dialog editor itself, and at the bottom, an optional output window to show the results of compiling an application. Figure C-1 shows a typical session. The dialog editor is available when you have clicked on a dialog element in the project tree, and it shows a visual representation of the dialog on the left with the selected element's properties on the right.

Figure C-1. The DialogBlocks main window

[View full size image]

New controls and sizers can be selected from drop-down menus attached to the dialog editor's toolbar buttons (above the property editor). The toolbar also gives shortcuts to various sizer properties such as alignment, stretch factor, and spacing.

There are also tabs to view the C++ and XRC code generated for the current element, and further tabs let you define variables and event handlers for the selected element.

You can view the wxWidgets reference by clicking on the Reference tab (pressing F1 goes to the appropriate topic for the selected element). The Support page lists useful links for the wxWidgets developer, which on Windows can be browsed from within DialogBlocks.

The DialogBlocks Sample Project

If you open the sample "Acme" project, you can get a feel for how DialogBlocks works by navigating through the project. There are several top-level elements, including dialogs, a frame, and an application object. The latter generates the wxApp-derived class that is necessary for a wxWidgets application to compile and run. It's not essential to have this element in DialogBlocks if you are writing your application class by hand and are just using DialogBlocks to create visual elements.

If you click on a dialog element, the dialog editor will be shown, and you can click on either the dialog editor or the project tree to select controls and sizers within the dialog. A selected element is shown with a red outline, and its parent with a blue outline, to allow you to see the context of the element. Try double-clicking on a control: normally you will be shown an editor for the default property, for example a button label. Other properties can be edited via the property editor: scroll down and single-click on the value of the property you're interested in, or double-click to invoke a specialized editor such as a multiline text editor or color selector.

Compiling the Sample

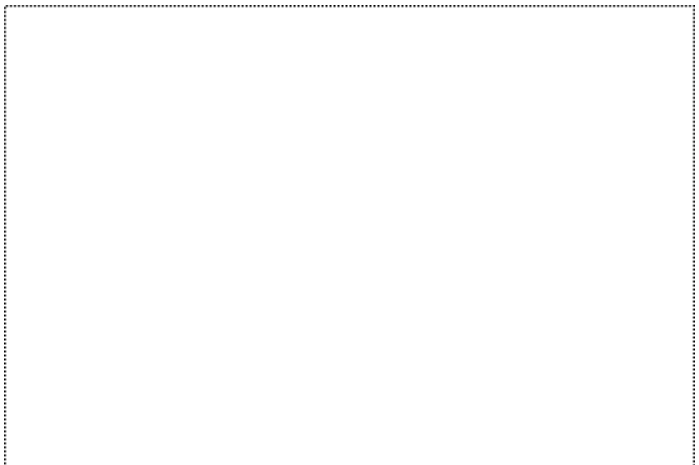
You can compile your project from within DialogBlocks for a range of popular compilers including Visual C++, Borland C++, GCC, MinGW, and Digital Mars C++. DialogBlocks lets you to add as many configurations as you want, each based on a particular compiler. You will typically have debug and release configurations for each compiler you use. You could have separate configurations for different versions of wxWidgets, too.

The Acme sample has several configurations, one of which you can select via the second drop-down list on the toolbar. You can add more by clicking on the Configuration panel on the settings dialog (Figure C-2) and clicking on Add. Or, to quickly add standard Debug and Release configurations, click on Standard, and choose a compiler (Figure C-3). Configuration properties can be edited in the scrolling property panel: changing a high-level property (such as Build Mode) can change the default value for a low-level property (such as Preprocessor Flags).

Figure C-2. DialogBlocks configurations dialog

[View full size image]

Figure C-3. DialogBlocks Standard Configurations Dialog



Creating a New Project

The New Project Wizard (invoked from the New Project menu item or toolbar button) leads you through a series of pages. In the first page, enter the full path of the new DialogBlocks file, and optionally your name and a copyright string to be inserted into generated files. The second page gives you the opportunity to decide to generate C++ code that uses XRC files instead of creating all elements in C++ code (you can change this option later). The third page lets you tailor the name of application class that DialogBlocks will generate (if any). Clear the check box if you don't want DialogBlocks to generate the application class. Finally, you will be asked if you want to create some default debug and release configurations for a selected compiler.

After you've created the project, you can add dialogs and other elements as required.

Creating a Dialog

Create a new dialog either using the drop-down menu on the New Top-Level Element toolbar button or from the Element menu. Fill in the title, class name, and C++ file names for this dialog. A dialog starts out with a default vertical box sizer, but you can delete this and replace it with a different sizer.

Select the top-level sizer and add a wxButton by clicking on the left-most button in the dialog editor toolbar (just above the property editor). Now add a single-line wxTextCtrl. It appears beneath the button because the containing sizer is a vertical box sizer. Now try adding a horizontal box sizer, then several buttons. The horizontal box sizer will stretch to fit the new buttons, expanding the dialog (Figure C-5).

Figure C-5. A simple dialog with two sizers

The first button you created is now centered in the horizontal axis because this is the default. You can experiment with different alignments using the yellow toolbar buttons.

Now select the wxTextCtrl you created earlier and add a wxListBox. It's created just after the text control with a default size; you can make it stretch to fit the available horizontal space by clicking on the Expand Horizontally button. Try previewing the dialog with F5 and resizing it. You'll notice that there's a lot of empty space if you make the dialog big (Figure C-6).

Figure C-6. Wasted space when resizing

Close the preview, click on the list box, and click on the double-headed arrow ("the stretch factor"). Now resizing the dialog makes the list box grow to take up available space (Figure C-7).

Creating a Frame

A frame starts off empty, and you can add a menu bar, a toolbar, a status bar, and either a single control or subwindow to fill the client area or a single top-level sizer under which you can add further controls and/or subwindows. You can't add multiple windows directly under the frame, but you can use a sizer or a parent window to contain further windows.

Creating an Application Object

If you didn't ask the New Project Wizard to create an application object, you can still add one from the toolbar or menu bar. This will create the class and OnInit function, which you can edit as required. Select a window identifier in the Main Window property to have DialogBlocks generate the code to create the main window object.

Debugging Your Application

If you are using GCC on any platform, you can use the Debug Project command on the Build menu to show GDB in a window, with a toolbar to accelerate common debugging commands. A manual for GDB is supplied with DialogBlocks under the Reference tab. Or you can use GDB standalone via Emacs or via a graphical front-end such as Insight, DDD, or KDbg (KDE only).

On Windows, you can use the debugger that comes with your compiler. In particular, to use the Visual Studio debugger, create a configuration for VC++ Project, select this configuration in the DialogBlocks toolbar, generate the project file, and open it in Visual Studio.

Further Information

This introduction has only scratched the surface of what you can do with DialogBlocks. Please refer to the online help for further information, in particular the "How To..." section. See also the DialogBlocks web site and mailing list at http://www.anthemion.co.uk/dialogblocks.

Appendix D. Other Features in wxWidgets

wxWidgets is a large system, and we couldn't cover all its features in depth. Here we present a sampler of other aspects that may be useful to you; details on most of them can be found in the wxWidgets reference manual. See also Appendix E, "Third-Party Tools for wxWidgets," for third-party classes.

Further Window Classes

wxGenericDirCtrl shows a hierarchy of directories, and optionally files; it can be used to build browsers.

wxCalendarCtrl (see Figure D-1) is an attractive way for the user to enter date information. This control can be customized in various ways, including highlighting special dates and starting the week with either Monday or Sunday. wxCalendarCtrl is in the core wxWidgets library, and the sample is in samples/calendar.

Figure D-1. wxCalendarCtrl

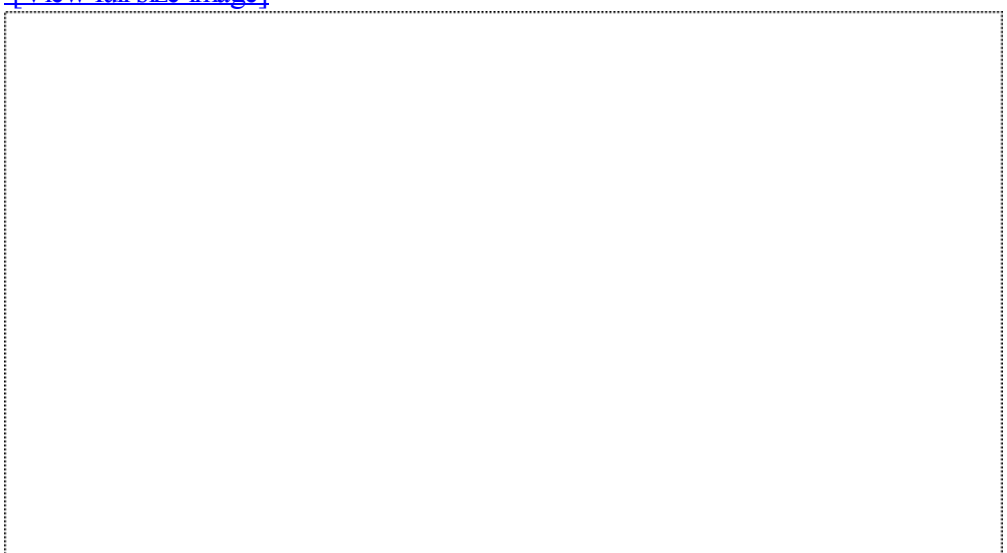
wxDatePickerCtrl is a more compact control than wxCalendarCtrl that allows the user to select a date. On Windows, it uses a native Win32 control, and on other platforms, a generic wxWidgets version is used.

wxTipWindow is a kind of wxPopupWindow that can be used for showing tooltips, and it is used by wxSimpleHelpProvider to show popup help. The tooltip text is provided in the constructor itself.

wxStyledTextCtrl is a wrapper around Scintilla, a highly capable code editor with highlighting, wrapping, and many other features. With minimal code, your application can support editing text files for a large number of different file formats and programming languages. The Scintilla project can be found at www.scintilla.org, and wxStyledTextCtrl can be found in the contrib hierarchy of your wxWidgets distribution. Check out the demo in contrib/samples/stc (see Figure D-2); documentation is available at http://www.yellowbrain.com/stc.

Figure D-2. wxStyledTextCtrl example

[View full size image]



ODBC Classes

ODBC is a cross-platform standard for accessing databases, so it's natural for wxWidgets to provide support for it. Using wxODBC, you can access a wide variety of databases including DB2, DBase, Firebird, INFORMIX, Interbase, MS SQL Server, MS Access, MySQL, Oracle, Pervasive SQL, PostgreSQL, Sybase, XBase, Sequiter, and VIRTUOSO. The main wxODBC classes are wxDb and wxDbTable. For more information, see "Database Classes Overview" in the wxWidgets reference manual. See also samples/db and demos/dbbrowse (see Figure D-7). You may also be interested in reusing the generic tab and page controls that are part of this demo.

Figure D-7. wxWidgets dbbrowse ODBC demo

For other wxWidgets database access wrappers, see wxOTL and wxSQLite in Appendix E.

MIME Types Manager

The wxMimeTypesManager class enables an application to retrieve information about all known MIME types from a system-specific location, including the file extension for each MIME type. An application uses a global instance of it, called wxTheMimeTypesManager.

Network Functionality

wxWidgets provides more than just the wxSocket class for working with networks. These are some of the network-related wxSocket classes that you can find more about from the manual.

wxURI can be used to extract information from a URI (Uniform Resource Identifier). A URL is a subset of a URI, so the wxURL class (for parsing and streaming from URLs) is derived from wxURI.

wxIPV4address is used to represent a standard Internet address.

wxFTP and wxHTTP can be used for performing FTP and HTTP operations. However, for a fuller implementation, consider using the CURL library instead (see Appendix E for wxCURL and wxCurlDAV).

wxDialUpManager encapsulates functions dealing with verifying the connection status of the computer (connected to the Internet via a direct connection, connected through a modem, or not connected at all) and to establish this connection if required. The application can also request notification about changes in the connection status. This class is currently only supported on Windows.

The wxEmail class in contrib/src/net can be used to send mail via SMAPI on Windows or the sendmail program on Linux.

Multimedia Classes

You can use the wxSound class on all platforms to load and play short sound files. On Windows, wxSound uses wave files (.wav). On Linux, the Open Sound System is used where available, and so it supports the formats handled by OSS. On Mac OS X, wxSound uses Apple's QuickTime to play wave and other formats.

From wxWidgets 2.5.4, the wxMediaCtrl class is available on Windows, Mac OS X, and Linux. wxMediaCtrl can play sound and video files, and it uses DirectShow on Windows, QuickTime on Mac OS X, and GStreamer on Linux.

Embedded Web Browsers

Although wxHtmlWindow is a fantastic lightweight HTML viewer, sometimes you need full web facilities in your application. Eventually wxWidgets will provide a single class to embed an appropriate browser on each platform, but for now there are different classes to achieve this.

On Mac OS X, wxWidgets comes with wxWebKitCtrl, which you need to enable by passing --enable-webkit to configure when building wxWidgets.

On Linux, you can download wxMozilla (see Appendix E). Be warned that your application distribution will swell by many megabytes if embedding Mozilla.

On Windows, you can download wxIE (see Appendix E) to embed Internet Explorer in your application. You can also consider using wxMozilla on Windows.

Accessibility

Because wxWidgets uses native widgets wherever possible, applications built with it are already quite accessible and tend to be friendly towards screen-reading applications. However, there is a wxAccessible class that can be used to make an application more accessible on Windows by deriving from the class, providing implementations of virtual classes, and associating an instance of it with the appropriate window instance. The class should be enabled with wxUSE_ACCESSIBILITY to 1 in setup.h. Currently, only Microsoft Active Accessibility is supported.

OLE Automation

The wxAutomationObject class represents an OLE automation object containing a single data member, an IDispatch pointer. It contains a number of functions that make it easy to perform automation operations and set and get properties. The class makes heavy use of the wxVariant class and is only available under Windows.

The usage of these classes is quite close to OLE automation usage in Visual Basic. The API is high-level, and the application can specify multiple properties in a single string. The following example gets the current Excel instance, and if it exists, makes the active cell bold:

```
wxAutomationObject excelObject;
if (excelObject.GetInstance("Excel.Application"))
      excelObject.PutProperty("ActiveCell.Font.Bold", true);
```

Renderer Classes

The wxRendererNative class and derivatives abstract high-level drawing operations for widgets or parts of widgets, such as buttons, splitter bars, and so on. This allows windows that are drawn "generically" using wxWidgets to use native or at least consistent components. For more information, please see the documentation for wxRendererNative.

Event Loops

The event loop is modeled with the wxEventLoop class. Start the loop by calling Run, test whether the loop is running with IsRunning, and exit the loop with Exit. This class is used for the main loop of the application, and you can also use it for subordinate event loops, as used when showing a modal dialog.

Appendix E. Third-Party Tools for wxWidgets

This appendix lists a selection of libraries and tools you can use with wxWidgets. More can be found on the wxWidgets web site, particularly in the Resources and Contributions sections, and also on http://wxcode.sf.net.

Language Bindings

C++ is not the only language you can use for wxWidgets programming: the following projects integrate wxWidgets with other languages. Some are more fully developed than others, and not all work on all major platforms. wxPython is the most mature and popular of all bindings, and it works on Windows, Linux, and Mac.

wxPython combines wxWidgets and the Python language to create a powerful and popular tool for rapid GUI programming. Figure E-1 shows wxPython running its demo programeven if you won't be using the Python language, it's worth installing it and trying out the demo because it covers a lot of wxWidgets functionality. wxPython is on the accompanying CD-ROM and is also available from http://www.wxpython.org.

Figure E-1. The wxPython demo

[View full size image]

wxPerl adds wxWidgets GUI programming to the Perl language. It's available from http://wxperl.sourceforge.net.

wxBasic is a combination of wxWidgets and a variant of the BASIC language. It's available from http://wxbasic.sourceforge.net.

wxLua binds the lightweight Lua language to wxWidgets and is easy to integrate into applications as a GUI-enabled extension language. It's available from http://www.luascript.thersgb.net.

wxJavaScript integrates wxWidgets with JavaScript. It's available from http://wxjs.sourceforge.net.

wx4j is a binding of Java to wxWidgets. It's available from http://www.wx4j.org.

wxRuby combines wxWidgets and the Ruby language. It's available from http://wxruby.rubyforge.org.

wxEiffel combines wxWidgets and the Eiffel language. It's available from http://elj.sourceforge.net.

wx.NET is a C# binding for .NET and Mono. It's available from http://wxnet.sourceforge.net.

wxHaskell is a Haskell binding for wxWidgets. It's available from http://wxhaskell.sourceforge.net.

Tools

These are tools that are either specifically designed to help you with your wxWidgets application development or are generally useful for application development.

wxDesigner is a commercial dialog editor and RAD tool and can write C++, Python, Perl, and C# code directly. Its interface lets anyone somewhat familiar with wxWidgets to create aesthetically pleasant cross-platform dialogs in a matter of minutes. Features such as supporting copy/cut/paste, infinite undo/redo, and previewing make testing easy and safe. It's available for Windows, Linux, and Mac OS X from http://www.roebling.de.

DialogBlocks is a sizer-based resource editor that creates professional-looking dialogs, wizards, and frames for deployment on any supported wxWidgets platform. DialogBlocks also generates makefiles and project files for a range of compilers. It's available for Windows, Linux, and Mac OS X from http://www.anthemion.co.uk/dialogblocks .

poEdit is a gettext catalog (.po file) editor. Unlike other catalogs editors, poEdit shows data in very compact way. Entries are arranged in a list so that you can easily navigate large catalogs and immediately get an idea about how much of the catalog is already translated, what needs translating, and which parts are only translated in a "fuzzy" way. It's available for Windows, Linux, and Mac OS X from http://poedit.sourceforge.net.

Bakefile generates makefiles for multiple platforms. Originally designed for the wxWidgets project, it can also be applied to your own applications and libraries. It's available for Windows, Linux, and Mac OS X from http://bakefile.sourceforge.net.

HelpBlocks is an authoring tool for MS HTML Help and wxWidgets HTML Help files, available on Windows, Linux, and Mac (beta) from http://www.helpblocks.com.

wxVisualSetup integrates wxWidgets help with Microsoft Visual Studio and includes a wxWidgets project wizard, Intellisense support, dynamic help, and tips and tricks. It's available from http://www.litwindow.com/wxVisualSetup.

wxGlade is a GUI designer built with wxPython, generating Python, C++, and XRC code. It's available from http://wxglade.sourceforge.net.

wxDev-CPP is a wxWidgets form designer plugin for the Dev-C++ IDE. It's available from http://wxdsgn.sourceforge.net.

wxHatch is a free RAD tool for wxWidgets. It's available from http://biolpc22.york.ac.uk/wx/wxhatch.

XRCed is a wxWidgets and wxPython development tool written in wxPython. It's available from http://xrced.sourceforge.net.

wxWinWiz is a Visual C++ wizard for creating wxWidgets projects. It's available from http://www.koansoftware.com/en/prd_svil_wxdownload.htm.

wxCRP is a tool to generate snippets of code from templates. It's available from http://www.xs4all.nl/~jorgb/wxcrp.

Chinook Developer Studio is a multi-platform C/C++ integrated development environment. It's available for Windows and Linux from http://www.degarrah.com.

MinGW Developer Studio is an IDE written in, and for, developing programs with wxWidgets and MinGW. It's available for Windows and Linux from http://www.parinyasoft.com.

CodeBlocks is an IDE written with wxWidgets, available for Windows and Linux from http://www.codeblocks.org.

KDevelop is a capable, free IDE for Linux. It's available from http://www.kdevelop.org.

Add-on Libraries

This section presents a selection of add-on libraries for wxWidgets available at the time of writing. These tend to be distributed as source code, which you must compile, and there is currently no standard way of doing this. With some, you can simply add the source files to your own make or project file; others come with their own make or project file or even use a configure script, creating libraries for you to link against. Please see the individual packages for build instructions. A package manager and standards for third-party code are being developed to make this easier in the future.

wxMozilla is a project to develop a wxWindows component for embedding the Mozilla browser into any wxWidgets application. It's available from http://wxmozilla.sourceforge.net.

wxIndustrialControls provides a set of widgets for showing digital and analog values. Includes Angular Meter, Linear Meter, Angular Regulator, Linear Regulator, Bitmap Switcher, Bitmap Check Box, LCD Display, and LCD Clock. It's available from http://www.koansoftware.com/en/prd_svil_wxindctrl.htm.

wxCURL is a simplified and integrated interface between LibCURL and wxWidgets. wxCURL provides several classes for simplified interfaces to HTTP, WebDAV, FTP, and Telnet-based resources. It's available from http://sourceforge.net/projects/wxcurl.

wxCurlDAV is a C++ class designed for people using wxWidgets to simply and easily add WebDAV functionality to their application. It's available from http://homepage.mac.com/codonnell/wxcurldav.

ToasterBox is a cross-platform library to make the creation of MSN style "Toaster" popups easier (message windows that slide up and down). It's available from http://toasterbox.sourceforge.net.

wxVTK enables the 3D graphics library VTK to render to and interact with wxWidgets. It's available from http://wxvtk.sourceforge.net.

wxDockIt is a docking library by Mark McCormack. It's available from http://sourceforge.net/projects/wxextended.

wxIFM (Interface Management System) is a docking library based on a plug-in architecture. It's available from http://www.snakesoft.net/wxifm.

wxMathPlot is a framework for mathematical graph plotting in wxWidgets. It's available from http://wxmathplot.sourceforge.net.

wxTreeMultiCtrl enables you to add any wxWindow-derived class to a tree shaped structure similar to a wxtreeCtrl. It is well suited for a scrollable property sheet. It's available from http://www.solidsteel.nl/jorg/components/treemultictrl/wxTreeMultiCtrl.php.

wxVirtualDirTreeCtrl is a very handy component for quickly creating project browsers, repository views, and smart directory selectors based upon event handlers, which can be overridden to make it look even more flexible. It is based upon a wxtreeCtrl component and designed in such a way that the native functionality of this class can still be used. It's available from http://www.solidsteel.nl/jorg/components/virtualdirtreectrl/wxVirtualDirTreeCtrl.php.

wxPropertyGrid is a specialized two-column grid for editing properties such as strings, numbers, flagsets, string arrays, and colors. It's available from http://www.geocities.com/jmsalli/propertygrid/index.html.

wxSMTP is a an email framework including an SMTP class. See also contrib/src/net in the wxWidgets distribution for simple email-sending functionality. wxSMTP is available from http://www.frank-buss.de/wxwindows/wino.html.

wxResizeableControl provides a user interface for child windows placed within other windows. The user can drag them around and resize them using eight marks drawn on the edges and corners of the control. This class sends notification events to its parent when the window is created, resized, moved, or deleted. It's available from http://de.geocities.com/markusgreither/resizec.htm.

Appendix F. wxWidgets Application Showcase

There are hundreds, if not thousands, of applications using wxWidgets. All of the applications in this selection are mature, usable programs that are actively developed and supported. See also the "Tools" section in Appendix E, which also contains good examples of wxWidgets applications. In addition, there are some small but complete applications in the demos directory of the wxWidgets distribution, including a card game, a poetry viewer, a cellular automaton toy, and a fractal mountain generator.

AOL Communicator is a convenient way for AOL members to manage their email, address book, and much more. Communicator integrates email (including multiple email addresses), spam filtering, AIM instant messaging, and news in one interface. It's available for Windows and Mac OS X from http://communicator.aol.com.

Audacity is a free audio editor that can record and play sounds, import and export WAV, AIFF, Ogg Vorbis, and MP3 files, and more. It also has a built-in amplitude envelope editor, a customizable spectrogram mode, and a frequency analysis window for audio analysis applications. It's available for Windows, Linux, and Mac OS X (see Figure F-1) from http://audacity.sourceforge.net.

Figure F-1. Audacity running on MacOS X

[View full size image]

AVG Antivirus is a comprehensive antivirus program. The unique combination of detection methods (heuristic analysis, generic detection, scanning, and integrity checking) ensures maximum protection on multiple levels. There are versions for workstations, networks, email servers, and file servers. It's available for Windows and Linux from http://www.grisoft.com.

BitWise IM is an instant messenger with text, file transfer/sharing, whiteboard, and voice capability. All data sent between clients is automatically encrypted for privacy. The cross-platform interoperability of all features is unmatched by any other instant messenger. A free Personal and a paid Professional version are available for Windows, Linux, and Mac OS X (see Figure F-2) from http://www.bitwiseim.com.

Figure F-2. BitWise on Mac OS X

[View full size image]

Appendix G. Using the CD-ROM

Browsing the CD-ROM

The CD-ROM Contents

Browsing the CD-ROM

The CD-ROM can be read under Windows, Mac OS X, and Linux (and other UNIX systems). The HTML contents should open automatically when you insert the CD-ROM under Windows and Linux. Otherwise, please open the file BrowseMe.htm with your preferred web browser.

The CD-ROM Contents

Here's what's on the disk:

- Code examples. Paste snippets into your own application, or compile the examples with your preferred compiler. In addition to the examples covered in this book, we include a bonus sample, "Riffle." This little image browser comes complete with source, installation scripts, and binaries for four platforms.
- wxWidgets 2.6. wxWidgets supports Windows (for desktop and Pocket PC), Unix/Linux, Mac OS X, and other platforms not covered by this book. Refer to Appendix A, "Installing wxWidgets," for installation details for the three major platforms.
- DialogBlocks Personal Edition. A version of the commercial DialogBlocks dialog editor/RAD tool for personal use. DialogBlocks runs on Windows, Linux, and Mac OS X with powerful sizer-based layout tools and the ability to compile your source using popular compilers. See Appendix C, "Creating Applications with DialogBlocks," to find out how to install and use DialogBlocks.
- Compilers. For Windows, we supply MinGW, Digital Mars C++, and OpenWatcom C++. (For Linux, GCC can be installed from your distribution, and for Mac OS X, the Apple Developer Tools are available from the Apple web site.)
- poEdit. poEdit is an essential tool to help you create message catalogs for your internationalized application.
- wxPython. wxPython is a powerful blend of wxWidgets and the Python language.

For updates, please see http://www.wxwidgets.org/book.

Appendix H. How wxWidgets Processes Events

This appendix takes a closer look at how wxWidgets processes events, going into details omitted from the simplified view we've seen so far.

When an event is received from the windowing system, wxWidgets calls wxEvtHandler::ProcessEvent on the first event handler object belonging to the window generating the event.

Figure H-1 summarizes the order of event table searching by ProcessEvent. Here's how it works:

1. If the object is disabled (via a call to wxEvtHandler::SetEvtHandlerEnabled), the function skips to Step 6.

2. If the object is a wxWindow, ProcessEvent is recursively called on the window's wxValidator. If this returns true, the function exits.

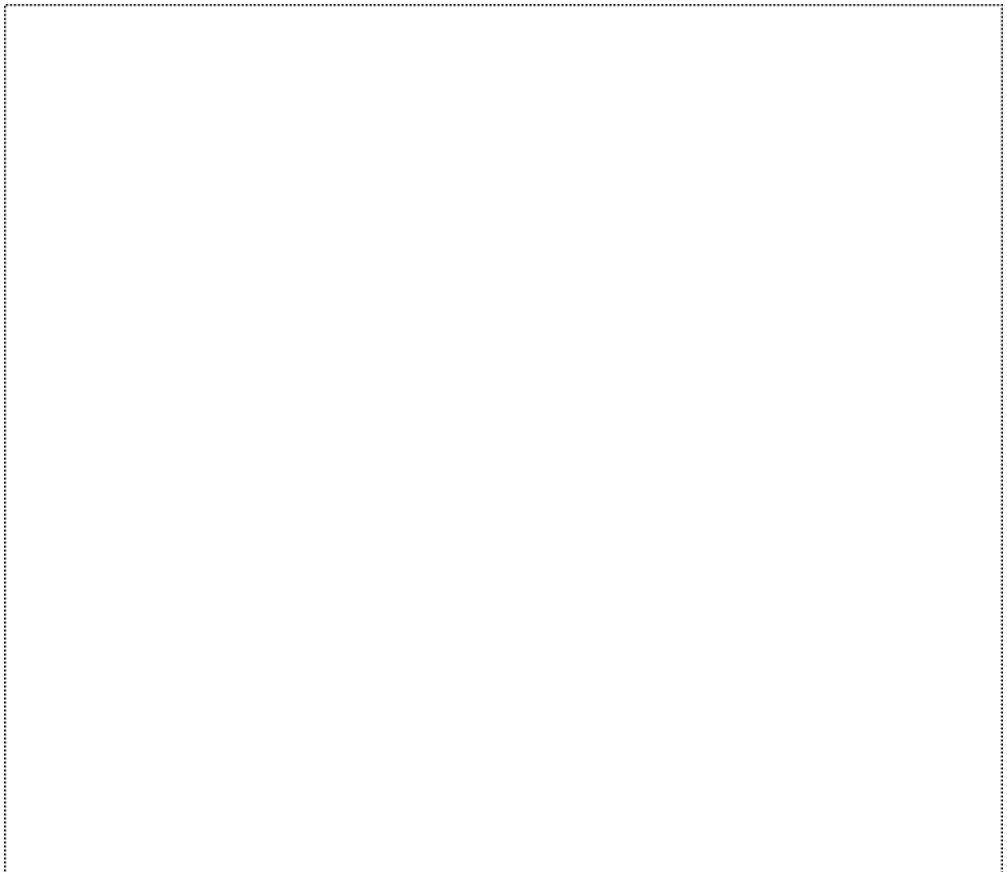
3. SearchEventTable is called for this event handler. If this fails, the base class table is tried, the next base class table is tried, and so on, until no more tables exist or an appropriate function is found, in which case the function exits.

4. The search is applied down the entire chain of event handlers. (Usually the chain has a length of one.) If this succeeds, the function exits.

5. If the object is a wxWindow and the event is set to propagate (only wxCommandEvent objects are normally set to propagate), ProcessEvent is recursively applied to the parent window's event handler. If this returns true, the function exits.

6. Finally, ProcessEvent is called on the wxApp object.

Figure H-1. Event processing flow



Appendix I. Event Classes and Macros

In the wxWidgets reference, the documentation for specific event macros is organized by their event classes (such as wxCommandEvent), which are themselves referenced by the control class documentation (such as wxButton). You can refer to these sections for details, but it's useful to summarize the most commonly used event classes and macros, as shown in Table I-1.

Table I-1. Commonly Used Event Macros

Class: wxActivateEvent	
EVT_ACTIVATE(func)	Sent when a user activates or deactivates a top-level window.
EVT_ACTIVATE_APP(func)	Sent when a user activates or deactivates the window of a different application.
Class: wxCommandEvent	
EVT_COMMAND(id, event, func)	The same as EVT_CUSTOM, but expects a member function with a wxCommandEvent argument.
EVT_COMMAND_RANGE (id1, id2, event, func)	The same as EVT_CUSTOM_RANGE, but expects a member function with a wxCommandEvent argument.
EVT_BUTTON(id, func)	Processes a wxEVT_COMMAND_BUTTON_CLICKED event, generated when the user left-clicks on a wxButton.
EVT_CHECKBOX(id, func)	Processes a wxEVT_COMMAND_CHECBOX_CLICKED event, generated when the user checks or unchecks a wxCheckBox control.
EVT_CHECKLISTBOX(id, func)	Processes a wxEVT_COMMAND_CHECKLISTBOX_TOGGLED event, generated by a wxCheckListBox control when the user checks or unchecks an item.
EVT_CHOICE(id, func)	Processes a wxEVT_COMMAND_CHOICE_SELECTED event, generated by a wxChoice control when the user selects an item in the list.
EVT_COMBOBOX(id, func)	Processes a wxEVT_COMMAND_COMBOBOX_SELECTED event, generated by a wxComboBox control when the user selects an item in the list.

Appendix J. Code Listings

Custom Dialog Class Implementation

wxWizard Sample Code

Custom Dialog Class Implementation

These are the header and implementation files for the PersonalRecordDialog class described in Chapter 9, "Creating Custom Dialogs." It can be found in examples/chap09 on the CD-ROM.

Listing J-1. Header File for PersonalRecordDialog

```
/////////////////////////////////////////////////////////////////////
// Name:        personalrecord.h
// Purpose:     Dialog to get name, age, sex, and voting preference
// Author:      Julian Smart
// Created:     02/28/04 06:52:49
// Copyright:   (c) 2004, Julian Smart
// Licence:     wxWindows license
/////////////////////////////////////////////////////////////////////

#ifndef _PERSONALRECORD_H_
#define _PERSONALRECORD_H_

#ifdef __GNUG__
#pragma interface "personalrecord.cpp"
#endif

/*!
 * Includes
 */

#include "wx/spinctrl.h"
#include "wx/statline.h"

/*!
 * Control identifiers
 */
enum {
    ID_PERSONAL_RECORD = 10000,
    ID_NAME = 10001,
    ID_AGE = 10002,
    ID_SEX = 10003,
    ID_VOTE = 10006,
    ID_RESET = 10004
};

/*!
 * PersonalRecordDialog class declaration
 */

class PersonalRecordDialog: public wxDialog
{
    DECLARE_CLASS( PersonalRecordDialog )
    DECLARE_EVENT_TABLE()

public:

//// Constructors

    PersonalRecordDialog( );

    PersonalRecordDialog( wxWindow* parent,
      wxWindowID id = ID_PERSONAL_RECORD,
      const wxString& caption = wxT("Personal Record"),
      const wxPoint& pos = wxDefaultPosition,
      const wxSize& size = wxDefaultSize,
      long style = wxCAPTION|wxRESIZE_BORDER|wxSYSTEM_MENU );

    /// Member initialization
```

wxWizard Sample Code

This code is the full listing for the wxWizard example as described in Chapter 12, "Advanced Window Classes." It can be found in examples/chap12 on the CD-ROM.

Listing J-3. Wizard Sample Code
[View full width]

```
/////////////////////////////////////////////////////////////////////
// Name:        wizard.cpp
// Purpose:     wxWidgets sample demonstrating wxWizard control
// Author:      Vadim Zeitlin
// Licence:     wxWindows licence
/////////////////////////////////////////////////////////////////////

// headers

#include "wx/wx.h"
#include "wx/wizard.h"

#include "wiztest.xpm"
#include "wiztest2.xpm"

// constants

// ids for menu items
enum
{
    Wizard_Quit = 100,
    Wizard_Run,
    Wizard_About = 1000
};

// private classes

class MyApp : public wxApp
{
public:
    // override base class virtuals
    virtual bool OnInit();
};

class MyFrame : public wxFrame
{
public:
    // ctor(s)
    MyFrame(const wxString& title);

    // event handlers
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
    void OnRunWizard(wxCommandEvent& event);
    void OnWizardCancel(wxWizardEvent& event);
    void OnWizardFinished(wxWizardEvent& event);

private:
    DECLARE_EVENT_TABLE()
};
// some pages for our wizard
// this shows how to simply control the validity of the user input
// by just overriding TransferDataFromWindow() - of course, in a
// real program, the check wouldn't be so trivial and the data
// will be probably saved somewhere too
//
// it also shows how to use a different bitmap for one of the pages
```

Appendix K. Porting from MFC

There are thousands of applications and millions of lines of code written in MFC, and it's increasingly common for organizations and individuals to migrate from MFC to wxWidgets to take advantage of other platforms and markets. Porting is easier than you might think because the wxWidgets API has many concepts and constructs that are similar to those in MFC. This appendix gives you an idea of what's involved, provides comparisons for particular features, and suggests some porting strategies.

General Observations

The processes of programming with MFC and programming with wxWidgets are fairly similar. In each, you write application, window, and other classes; create dialogs using suitable tools; define how user interaction relates to code; and compile and link the code with the GUI library. You can continue to use (say) Visual Studio to edit, compile, and debug your wxWidgets applications, using external tools such as DialogBlocks to create your dialogs. Or, you can use an open source IDE such as Dev-C++ or Eclipse; you can even switch to Linux or Mac as your main platform.

On Windows, you can combine MFC and wxWidgets code into a single executable: see samples/mfc in wxWidgets. So if you really need to, you may be able to delay porting parts of your code as long as you only need it on Windows. However, the complexity and space overhead of combining two frameworks means that a clean break from MFC is a better strategy.

Feature Comparison

We continue with a selection of topics to show how MFC constructs can be ported to wxWidgets.

Application Initialization

Like MFC, a wxWidgets application is driven by an application class: wxApp instead of CWinApp. The CWinApp::InitInstance override is replaced by wxApp::OnInit, which returns a boolean value to indicate that the event loop should be started. Application cleanup is done in wxApp::OnExit instead of CWinApp::ExitInstance.

wxWidgets applications require the IMPLEMENT_APP declaration to be placed in the application class implementation file, for example:

```
IMPLEMENT_APP(MyApp)
```

MFC applications access the command line with the m_lpCmdLine string member of CWinApp (or GetCommandLine), whereas in wxWidgets, you access the argc and argv members of wxApp, which are in the same format as the parameters of the traditional C main function. To help convert your existing command-line parsing code, use the wxCmdLineParser class explained in Chapter 20, "Perfecting Your Application."

Message Maps

Where MFC has message maps, wxWidgets uses event tables. (wxWidgets can also route events dynamically, using a different syntax.) The principle of event tables should be immediately familiar to MFC programmers. Although message handler functions can have an arbitrary number of parameters, wxWidgets event handlers always take a single event argument, through which information can be passed in and out of the event handler. As with MFC, event tables can be placed in a window or in the application class, and in document and view classes when using the document/view framework. Let's compare an MFC message map with a wxWidgets event table. Here's the application class and message map in MFC (note that the class declaration and message map would normally be in separate files):

```
//
// MFC message map
//

class CDemoApp: public CWinApp
{
public:
    CDemoApp();

    // Overrides
    virtual BOOL InitInstance();

// Implementation

    afx_msg void OnAppAbout();
    afx_msg void OnFileSave();
    afx_msg void OnStretchMode();
    afx_msg void OnUpdateStretchMode(CCmdUI* pCmdUI);

// Attributes
private:

    int      m_stretchMode;
    MyFrame *m_mainFrame;

    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CDemoApp, CWinApp)
```

Equivalent Functionality

The following tables compare MFC and wxWidgets constructs, grouped by macros and classes.

Equivalent Macros in MFC and wxWidgets

Table K-1 lists some important MFC macros and their wxWidgets equivalents.

Table K-1. MFC and wxWidgets Macros

MFC Version	wxWidgets Version
BEGIN_MESSAGE_MAP	BEGIN_EVENT_TABLE
END_MESSAGE_MAP	END_EVENT_TABLE
DECLARE_DYNAMIC	DECLARE_CLASS
DECLARE_DYNCREATE	DECLARE_DYMAMIC_CLASS
IMPLEMENT_DYNAMIC	IMPLEMENT_CLASS
IMPLEMENT_DYNCREATE	IMPLEMENT_DYNAMIC_CLASS
IsKindOf(RUNTIME_CLASS(CWindow))	IsKindOf(CLASSINFO(wxWindow))

Equivalent Classes in wxWidgets

Table K-2 lists the main MFC classes and their wxWidgets equivalents. MFC classes not present in the table have no direct equivalent.

Table K-2. MFC and wxWidgets Classes

Miscellaneous Classes	
MFC Version	wxWidgets Version
CWinApp	wxApp
CObject	wxObject
CCmdTarget	wxEvtHandler
CCommandLineInfo	wxCmdLineParser
CMenu	wxMenu, wMenuBar, wxMenuItem

Further Information

A useful article on this subject is Porting MFC Applications to Linux, by Markus Neifer, found at http://www-106.ibm.com/developerworks/library/l-mfc/?n-l-4182.

This includes full source code for a wxWidgets document/view sample application, together with the original MFC application.

GLOSSARY

accelerator

A key combination, such as Control-S, that allows selection of menu items, buttons, and other controls using the keyboard.

accessibility

An accessible application is one that can be used by people with impaired vision or other disabilities.

ANSI

An acronym for American National Standards Institute, an organization that defined a standard character set. wxWidgets distinguishes between compilation in ANSI and Unicode mode. In ANSI mode, character encodings must be used to switch between languages, but this is not necessary with Unicode. In this context, ANSI and ASCII are interchangeable, as the differences are slight.

alpha channel

Extra information in a bitmap that can be used to draw the bitmap with translucency. Each pixel in the bitmap has an 8-bit alpha value indicating the pixel's intensity.

API

Application Programming Interface, the published set of classes and functions that a library offers to an application.

assertion

A test supplied in debug mode that causes an error message to be shown if the test fails.

bakefile

A makefile generation system used within wxWidgets that can also be used for other libraries and applications.

bit-list

An integer containing flags that are combined with the binary OR operator ("|")for example, wxSUNKEN_BORDER|wxTAB_TRAVERSAL. The presence of a flag can be tested by using the binary AND operator ("&").

block

Describes the suspension of a function's execution while waiting for something, such as for a modal dialog to be dismissed or data to become available on a socket.

bundle

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

.NET
3D graphics
  wxGLCanvas 2nd 3rd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

accelerators 2nd 3rd
accessibility
accessing
  screens
    with wxScreenDC 2nd
accessors
  wxDateTime
Acme
adapting
  dialogs
    for small devices 2nd 3rd 4th
ADC (Apple Developer Connection)
add-on libraries 2nd 3rd 4th 5th 6th
AddFile
adding
  help
    creating custom dialogs 2nd 3rd 4th 5th 6th
  icons
    to GNOME desktop
    to KDE desktop
  source and includes
    Bakefile 2nd
addition
AddRoot
aesthetics
  designing dialogs
alignment
  sizers
alternatives
  to dialogs
  to wxNotebook 2nd
  to wxSocket 2nd
  to wxSplitterWindow 2nd
Anthemion Software
  DialogBlocks [See DialogBlocks]
AOL Communicator
Apple Developer Connection (ADC)
application class 2nd 3rd 4th 5th
application objects
  creating
    with DialogBlocks
  initializing
application paths
  finding 2nd 3rd
application resources
  storing
    finding application paths 2nd 3rd
    reducing data files
    standard locationis
application settings

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

backgrounds
  erasing window backgrounds 2nd 3rd
Bakefile
  adding
    source and includes 2nd
  bakefile_gen
  build options 2nd 3rd
  creating wxWidgets applications 2nd 3rd
    Mac OS X
    Windows
  includes 2nd 3rd
  presets 2nd 3rd
  project types 2nd
    autoconf 2nd
  sample wxWidgets project 2nd 3rd
  targets 2nd
  templates 2nd 3rd
bakefile_gen
base window classes 2nd
  wxControl
  wxControlWithItems 2nd
    member functions 2nd
  wxWindow
    wxWindow member functions 2nd 3rd 4th 5th 6th 7th
    wxWindow styles 2nd 3rd
BASIC
BC++ 2nd 3rd
  compiling sample programs
behavior
  of sockets
    with socket flags 2nd
binary resource files 2nd 3rd
bindings 2nd
bitmap buttons
  wxButton 2nd 3rd
bitmap resources
  packaging 2nd
bitmaps
  drawing 2nd 3rd 4th 5th 6th
  drawing on
    with wxMemoryDC 2nd
BitWise
Blit 2nd 3rd 4th
blocking
  sockets
Bodde, Jorgen
borders
  sizers
Borland C++ 2nd 3rd
  compiling sample programs

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

C pointer conversions
  wxString 2nd 3rd 4th
C string functions 2nd
CalcScrolledPosition
CalcUnscrolledPosition
CanHandle
CaptureMouse
carets
CFBundleDevelopmentRegion
CFBundleIconFile
CFBundleLocalizations
CFBundleTypeIconFile
changing [See modifying]
character encodings 2nd 3rd 4th
  converting data
    help files 2nd
    outside of a temporary buffer 2nd
    wxCSConv 2nd 3rd
    wxEncodingConverter
character sets
  font encoding
characters
  wxString 2nd
Chess Commander
Chinook Developer Studio
choice and selection dialogs
  wxColourDialog 2nd 3rd 4th 5th 6th
  wxFontDialog 2nd 3rd 4th 5th
  wxMultiChoiceDialog 2nd 3rd
  wxSingleChoiceDialog 2nd 3rd
choice control 2nd
  events
  member functions
choosing
  configurations 2nd
  development tools
  interface styles
    document/view systems 2nd 3rd
circular arcs
classes
  application class 2nd 3rd 4th 5th
  comparing MFC versus wxWidgets 2nd 3rd
  deriving new classes 2nd
  document classes
    document/view systems 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th
  file classes [See file classes]
    wxFile 2nd 3rd
  frame class 2nd
  frame classes
    creating and using for document/view systems 2nd 3rd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

DAO (Data Access Objects)
data
  designing dialogs 2nd
Data Access Objects (DAO)
data files
  reducing number of
  reducing number of Chapter 14, ÒFiles and Streams.˜ For more on using wxrc, Using wxWidgets Resource Files in Chapter 9.Standard LocationsThe wxStandardPaths class (in wxWidgets 2.5.4 and above) provides a portable way to find appropriate locations
data formats
  standard data formats
data objects 2nd
  data source duties 2nd
  data target duties
data source
  responsibilities of 2nd
data storage
  designing 2nd
data structures
  helper data structures
    wxLongLong 2nd
    wxObject 2nd
    wxPoint
    wxRealPoint
    wxRect 2nd
    wxRegion 2nd 3rd
    wxSize 2nd
    wxVariant 2nd
data target
  responsibilities of
data transfer
  creating custom dialogs 2nd 3rd 4th 5th
data types
  reading and writing 2nd 3rd 4th
database access
  MFC versus wxWidgets
date arithmetic 2nd 3rd 4th
dates
  formatting for internationalization 2nd
  wxDateTime 2nd
    accessors
    constructors and modifiers
    date arithmetic 2nd 3rd 4th
    date comparisons
    formatting dates
    getting current time
deadlocks
  synchronizing objects 2nd
debug builds
  versus release builds

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

EarthVision
editing
  application settings 2nd 3rd
Eiffel
Emacs
embedded resource files 2nd 3rd
embedded web browsers
embedding
  windows
    in HTML pages 2nd 3rd
Enable
encoding
  fonts
encodings 2nd 3rd
  converting data
    help files 2nd
    outside of a temporary buffer 2nd
    wxCSConv 2nd 3rd
    wxEncodingConverter
EndDoc
Endmodal
Entry
  wxThread
entry dialogs
  wxFindReplaceDialog 2nd 3rd 4th 5th 6th 7th 8th
  wxNumberEntryDialog 2nd
  wxPasswordEntryDialog
  wxTextEntryDialog 2nd
EPM 2nd
erasing
  window backgrounds 2nd 3rd
error notifications
  socket status 2nd 3rd
error reporting 2nd 3rd 4th 5th 6th
  wxMessageOutput
    versus wxLog 2nd 3rd
errors
  detecting 2nd 3rd 4th 5th
  X11 errors 2nd
Euphoria
event classes
  defining
    writing your own controls
  defining custom events 2nd 3rd 4th 5th
event handlers 2nd 3rd 4th 5th 6th
  defining default event handlers
    writing your own controls 2nd 3rd
  dynamic event handlers 2nd 3rd 4th
  example character event handler 2nd
  pluggable event handlers 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

face names
  fonts
file and directory dialogs
  wxDirDialog 2nd 3rd 4th
  wxFileDialog 2nd 3rd 4th 5th 6th
file classes
  wxDir 2nd 3rd
  wxFFile 2nd 3rd
  wxFile 2nd 3rd
  wxFileName 2nd
  wxTempFile
  wxTextFile
file functions 2nd 3rd
file history
  document/view systems 2nd
file receiving threads
  socket streams 2nd
file sending threads
  socket streams 2nd 3rd
file streams 2nd 3rd
file systems
  virtual file systems 2nd 3rd 4th 5th 6th
files
  include files
  Info.plist file
  library files
    resource files [See resource files]
  system files
  updating wxWidgets files 2nd 3rd
  writing
filling
  arbitrary areas 2nd
filter streams 2nd
FindFocus
finding
  application paths 2nd 3rd
FindWindow
FinishedIdleTask
Fit
FL (Frame Layout)
flag.png
flags
  DoDragDrop
    mask flags
      wxListItem
    socket flags [See socket flags]
flicker
  reducing 2nd 3rd 4th
FloodFill 2nd 3rd
font encoding

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

GCC 2nd 3rd 4th
  compiling sample programs
GDB
  Emacs
GetAdjustedBestSize
GetAllFiles
GetAlpha
GetBrush
GetClassDefaultAttributes
GetCurrentPage
GetDataHere
GetDC
GetDeviceOrigin
GetFirst
GetInternalRepresentation
GetKeyCode
GetLinesPerAction
GetLogicalFunction
GetMapMode
GetNext
GetNextToken
GetPageAreaSizer
GetPageInfo
GetPageSize
GetPageSizeMM
GetPageSizePixels
GetPartialTextExtents
GetPixel 2nd
GetPPIPrinter
GetPPIScreen
GetPreferredFormat
GetRGB
GetScreenType
GetSize 2nd
GetSizeMM 2nd
GetTextBackground
GetTextExtent 2nd
GetTextForeground
GetUserScale
GetWheelDelta
GetWheelRotation
Gloger, Wolfram
GNOME
  adding icons to desktop
graphics
  scrolling graphics
grids [See wxGrid]
GTK+
  printing under Unix 2nd
  wxButton labels

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

handling
  events
    creating custom dialogs
  UI updates
    creating custom dialogs 2nd 3rd
Hardy
HasAlpha
HASH_T
HasInput
Haskell
HasMoreTokens
HasPage
header files
  modifying
headers 2nd 3rd 4th 5th
help
  adding
    creating custom dialogs 2nd 3rd 4th 5th 6th
  authoring 2nd
Help
  buttons
help
  context-sensitive help 2nd 3rd
  help controllers 2nd 3rd 4th
  menu help
  MS HTML Help
  online help 2nd 3rd 4th
    context-senstive help 2nd
    implementing 2nd 3rd
    menu help
    tooltips 2nd
  tooltips 2nd 3rd
  wxModalHelp
help controllers 2nd 3rd 4th 5th 6th 7th 8th
help files 2nd
Help Viewer
HelpBlocks
helper data structures
  wxLongLong 2nd
  wxObject 2nd
  wxPoint
  wxRealPoint
  wxRect 2nd
  wxRegion 2nd 3rd
  wxSize 2nd
  wxVariant 2nd
hierarchies
  sizers
Holzem, Markus
HTML

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

i18n [See internationalization]
icon bundles 2nd 3rd 4th 5th 6th
icons [See wxTaskBarIcon]
  adding
    to GNOME desktop
    to KDE desktop
  associating with applications 2nd
  Mac OS X
  UI design guidelines
idel time
  windows
identifiers
  document/view systems
  font families
  window identifiers
IDEs
  Microsoft Visual Studio [See Microsoft Visual Studio]
idle time processing 2nd 3rd 4th
IFM (Interface Management System)
image classes
  wxBitmap
  wxCursor
  wxIcon
  wxImage
image handlers
image lists 2nd 3rd 4th 5th 6th
images
  3D graphics
    wxGLCanvas 2nd 3rd
  loading and saving
    wxImage 2nd 3rd 4th
implementing
  DoodleApp
  DoodleCanvas
  DoodleView
  drag sources 2nd 3rd 4th 5th 6th 7th
  drop targets 2nd 3rd
  online help 2nd 3rd
  resource handlers
    writing your own controls
  undo/redo 2nd 3rd 4th
  validators
    writing your own controls 2nd 3rd 4th
include files
includes
  adding
    to Bakefile 2nd
  Bakefile 2nd 3rd
Info.plist file
informative dialogs

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Java 2nd
JavaScript
joysticks 2nd 3rd 4th 5th
  wxJoystick
  wxJoystick events 2nd
  wxJoystickEvent

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Kay, Alan
KDE
  adding
    icons to desktop
KDevelop 2nd
Kdevelop
  creating wxWidgets applications 2nd 3rd
key code translation 2nd
KEY_EQ_T
keybinder
keyboard navigation
  designing dialogs
keyboards 2nd
  accelerators 2nd 3rd
  example character event handler 2nd
  key code translation 2nd
  modifier key variations 2nd 3rd
KICAD
Kirix Strata
Koan Software
  wxDao

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

labels
  button labels
    wxButton 2nd 3rd
language bindings 2nd
launching
  documents 2nd 3rd 4th
layout
  basics of layout 2nd
  dialog units 2nd
  dynamic layouts
  platform-adaptive layouts 2nd 3rd 4th
LayoutFrame
LayoutMDIFrame
layouts
  coding 2nd 3rd 4th 5th
  designing dialogs 2nd
leaks
  detecting memeory leaks
LED
libgnomeprint
libgnomeprintui
libraries
  add-on libraries 2nd 3rd 4th 5th 6th
  contributed libraries
  dynamic libraries
    loading 2nd 3rd
  libgnomeprint
  libgnomeprintui
  LitWindow Library
  multi-libraries
    versus monolithic libraries
  shared libraries
    Linux 2nd
    versus static libraries
library files
licenses
lightweight processes [See threads]
limitations
  of HTML Help
lines
  drawing 2nd 3rd 4th
Linus
  KDevelop [See KDevelop]
Linux
  development tools 2nd
  GCC 2nd 3rd
    compiling sample programs
  installing
    DialogBlocks 2nd
  installing applications 2nd 3rd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Mac OS
  instances
Mac OS X
  Bakefile 2nd
  development tools 2nd
  GCC 2nd 3rd
    compiling sample programs
  icons
    installing
    DialogBlocks
  installing applications 2nd 3rd 4th
  StoryLines
  Xcode
Macintosh
  event-driven programming
MacOpenFile
macros
  comparing MFC versus wxWidgets
  definining custom events 2nd 3rd 4th 5th
  joystick event table macros
  keyboard event table macros
  mouse event table macros
  wxDYNLIB_FUNCTION
macrows
  arrays
Mahogany Mail
Makefiles
  creating wxWidgets applications 2nd
makeinno.sh
maketarball.sh
manipulating
  wxImage data directly 2nd
mapping modes
mask flags
  wxListItem
MDI (MultipleDocument Interface)
media
  internationalization
member functions [See functions, member functions]
  wxButton
  wxRadioBox
  wxWindow 2nd 3rd 4th 5th 6th 7th
memory leaks
  detecting 2nd 3rd 4th 5th
memory management
  arrays
  cleaning up applications
  creating and copying drawing objects 2nd
  creating and deleting window objects 2nd 3rd
    debugging [See debugging]

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Navigate
NDEBUG
network-related classes
new
New Project Wizard
  DialogBlocks 2nd
non-blocking
  sockets
non-rectangular frames
  wxFrame 2nd 3rd 4th 5th
non-stataic controls
non-static controls
  wxBitmapButton 2nd 3rd
    events
    member functions
    styles 2nd
  wxButton 2nd
    events
    labels 2nd 3rd
    member functions
    styles
  wxCheckBox 2nd 3rd
    events
    member functions
    styles
  wxCheckListBox 2nd 3rd
    events
    member functions
    styles
  wxChoice 2nd
    events
    member functions
  wxComboBox 2nd
    events
    member functions 2nd
    styles
  wxGauge 2nd
    member functions
    styles
  wxListBox 2nd 3rd
    events
    member functions
    styles
  wxRadioBox 2nd
    events
    member functions
    styles
  wxRadioButton 2nd
    events
    member functions

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Object Graphics Library (OGL)
object-oriented programming
  and wxWidgets
objects
  application objects
    initializing
  drawing objects [See drawing objects]
  synchronizing
    deadlocks 2nd
    wxCondition 2nd 3rd 4th 5th 6th
    wxCriticalSection
    wxMutex 2nd 3rd
    wxSemaphore 2nd
  windows objects [See objects;windows objects]
ODBC classes
OGL (Object Graphics Library)
Ok
OLE automation
  wxAutomationObject
OnAcceptConnection
OnBeginDocument
OnBeginPrinting
OnChangeLanguage 2nd
OnDraw
OnEndPrinting
OnExecute
OnExit
OnGetItemAttr
OnGetItemImage
OnGetItemLabel
OnInit 2nd 3rd
OnInternalIdle
online help 2nd 3rd 4th
  implementing 2nd 3rd
  menu help
OnPaint
OnPaint handler
OnPreparePrinting
OnPrintPage
OnRun
OnSize event handler
OnSysRead
OnSysWrite
OnTerminate
output
  redirecting 2nd 3rd 4th 5th 6th

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

packaging
  bitmap resources 2nd
painting
  smooth painting and scrolling
  windows
palettes
  wxPalette
ParseFormat
parsing
  command lines 2nd 3rd 4th 5th
  dates
paste
  clipboard
Pause
pausing
  threads 2nd
Perl
PersonalRecoordDialog 2nd
  adding help 2nd 3rd 4th 5th 6th
  coding controls and layouts 2nd 3rd 4th 5th
  data transfer and validation 2nd 3rd 4th 5th
  deriving new classes 2nd
  designing data storage 2nd
  handling events
  handling UI updates 2nd 3rd
  invoking the dialog
PersonalRecordDialog
  header file for 2nd 3rd 4th
  implementation file for 2nd 3rd 4th 5th 6th
pgAdmin III
platform symbols
platform-adaptive layouts 2nd 3rd 4th
playing
  audio files
pluggable event handlers 2nd
poEdit
point size
popup menus
POSIX regular expressions
presets
  Bakefile 2nd 3rd
previewing
  document/view systems
  printing 2nd 3rd 4th 5th 6th
Print
PrintClasses
printf
  wxString
    numbers and dates for internationalization
printing

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

RAD (Rapid Application Development) tool
RAD tool
  wxHatch
radio buttons
Rapid Application Development (RAD) tool
RC files
reading
  data types 2nd 3rd 4th
  socket data
recreating
  dialogs
    MFC 2nd
redirecting
  process input and output 2nd 3rd 4th 5th 6th
reducing
  data files 2nd
  flicker 2nd 3rd 4th
Reference tab
  DialogBlocks
refresh speed
Register buttons
RegisterProcess
release builds
  debugging 2nd
  versus debug builds
ReleaseCapture
ReleaseMouse
renderer classes
resource files
  binary resource files 2nd 3rd
  converting dialogs 2nd
  embedded resource files 2nd 3rd
  foreign controls 2nd 3rd
  loading resources 2nd 3rd
  translating resources
  writing resource handlers 2nd 3rd
  XRC format 2nd
resource handlers
  implementing
    writing your own controls
  writing 2nd 3rd
resources
  loading
    in resource files 2nd 3rd
  translating
Resume
Roebling, Robert
RPM
RTTI (run-time type information) 2nd 3rd 4th 5th
Ruby

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

SashHitTest
saving
  images
    wxImage 2nd 3rd 4th
scaling
  for printing 2nd 3rd 4th 5th 6th
Scintilla
SciTech, Inc
screens
  accessing
    with wxScreenDC 2nd
scrollbars
  windows
scrolling
  without wxScrolledWindow 2nd
scrolling graphics
selectLanguage
SelectObject
sending
  commands
    to debuggers
serialization
Serialize function
servers
  connecting sockets to
    socket addresses 2nd
  socket servers
    creating 2nd 3rd
  sockets 2nd 3rd
SetAxisOrientation 2nd
SetCap
SetCapture
SetCheckpoint
SetData 2nd
SetDefaultStyle
SetDeviceOrigin 2nd
SetItemData
SetLogicalFunction
SetMapMode
SetMask
SetPageSize
SetScrollbar
SetScrollRate
SetSizeHints
SetState
SetTextBackground
SetTextForeground
settings [See application settings]
SetToolTip
setup.h

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

tables
  Flags for CreateButtonSizer
  Joystick Event Table Macros
  Keyboard Event Table Macros
  Modifier Keys under Windows, Unix, and Mac OS X
  Mouse Event Table Macros
  Sizer flags
  wxGrid [See wxGrid]
targets
  Bakefile 2nd
taskbar icons
  showing [See wxTaskBarIcon]
templates
  Bakefile 2nd 3rd
terminating
  applications
    UI design guidelines
  threads
TestDestroy
text
  drawing 2nd 3rd 4th 5th
text controls
  multi-line text controls
Thaw
third-party tools 2nd 3rd 4th
threads [See also multithreaded applications]
  creating 2nd 3rd
    specifying priority
    specifying stack size
  deciding when to use 2nd
  file receiving threads
    socket streams 2nd
  file sending threads
    socket streams 2nd 3rd
  GUI functions
  pausing 2nd
  sample wxWidgets thread
  starting
  terminating
  waiting for external conditions 2nd
time
  wxDateTime 2nd
    accessors
    constructors and modifiers
    date arithmetic 2nd 3rd 4th
    date comparisons
    formatting dates
    getting current time
timers
  wxTimer 2nd 3rd 4th

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

UDP
UI design guidelines
  application termination behavior
  colors
  fonts
  icons
  menus
  standard buttons 2nd
UI separation
  designing dialogs 2nd
UI updates
  handling
    creating custom dialogs 2nd 3rd
  windows
unary minus
underlines
  fonts
undo/redo
  implementing 2nd 3rd 4th
Unicode
  convTo
  Microsoft Visual Studio
  string literals
  wxCSConv
Unicode debug build
Unicode-versus non-Unicode
Unix
  Bakefile
  GCC 2nd 3rd
    compiling sample programs
  printing
    with GTK+ 2nd
Unlock
unpacking
  wxWidgets 2nd
UnregisterProcess
update region
UpdateBackingFromWindow
updates
  UI updates
updating
  wxWidgets files 2nd 3rd
upgrading
  DialogBlocks
useMask

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Validate
validation
  creating custom dialogs 2nd 3rd 4th 5th
  MFC 2nd
validators
  implementing
    writing your own controls 2nd 3rd 4th
  wxFontSelectorValidator
VideoLAN client (VLC)
view classes
  defining
    for document/view systems 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th
views
  MFC versus wxWidgets 2nd
virtual file systems 2nd 3rd 4th 5th 6th
virtual list controls 2nd 3rd
Visual Studio [See Microsoft Visual Studio]
VLC (VideoLAN client)
VMware
VTK

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Wait
WaitOnConnect
  wxSocketClient
web browsers
  embedded web browsers
weight
  fonts
window classes
  base classes [See base window classes]
  container windows
  control bars
  document/view systems 2nd 3rd 4th
  FL (Frame Layout)
  menus
  non-static controls
  OGL
  static controls
  top-level windows
  wxCalendarCtrl
  wxDatePickerCtrl
  wxEditableListBox
  wxFoldPanelBar
  wxGenericDirCtrl
  wxGIFAnimationCtrl
  wxLEDNumberCtrl
  wxSplashScreen
  wxStaticPicture
  wxStyledTextCtrl
  wxTipWindow
window identifiers 2nd
window objects
  creating and deleting 2nd 3rd
window variant
windows
Windows
  Bakefile
  bitmap colors 2nd
  Borland C++ 2nd
    compiling sample programs
windows
  carets
Windows
  changes to setup.h
windows
  client areas
  color
  concept of
    container windows [See container windows]
    control bars [See control bars]
  coordinate systems

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

X11 errors 2nd
xCHM
Xcode
  creating wxWidgets applications 2nd
XPM
  programming with wxBitmap 2nd
XRC files
  implementing resource handlers
    writing your own controls
XRC format
  resource files 2nd
XRCed

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

yielding

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Zeitlin, Vadim
zip streams 2nd 3rd