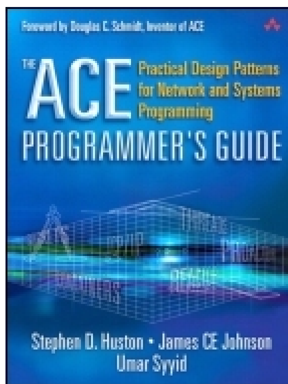


• Table of Contents

ACE Programmer's Guide, The: Practical Design Patterns for Network and Systems Programming

By Stephen D. Huston, James CE Johnson, Umar Syyid

Start Reading

Publisher: Addison Wesley
Pub Date: November 14, 2003
ISBN: 0-201-69971-0
Pages: 544

"If you're designing software and systems that must be portable, flexible, extensible, predictable, reliable, and affordable, this book and the ACE toolkit will enable you to be more effective in all of these areas. Even after spending over a decade developing ACE and using it to build networked software applications, I find that I've learned a great deal from this book, and I'm confident that you will, too."
-Douglas C. Schmidt, Inventor of ACE, from the Foreword

"This book is a must-have for every ACE programmer. For the beginner, it explains step-by-step how to start using ACE. For the more experienced programmer, it explains in detail the features used daily, and is a perfect reference manual. It would have saved me a lot of time if this book had been available some years ago"!
-Johnny Willemsen, Senior Software Engineer, Remedy IT, The Netherlands

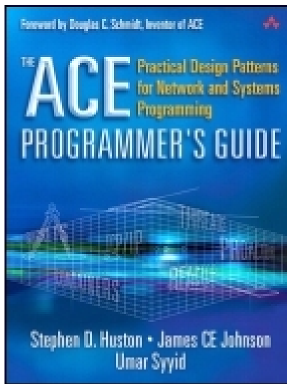
"With a large C++ code base, we rely on ACE to enable a cross-platform client-server framework for data quality and data integration. ACE has improved our design and smoothed over OS idiosyncrasies without sacrificing performance or flexibility. The combination of online reference materials and printed "big picture" guides is indispensable for us, and The ACE Programmer's Guide earns top-shelf status in my office."
-John Lilley, Chief Scientist, DataLever Corporation

"In SITA air-ground division, we are one of the major suppliers of communication services to the airline industry. We started using ACE about a year ago and are now moving most of our new communication-related development to it. I can say that using this toolkit can reduce the development and testing time by at least 50% in our type of application".
-Jean Millo, Senior Architect, SITA

The ADAPTIVE Communication Environment (ACE) is an open-source software toolkit created to solve network programming challenges. Written in C++, with the help of 30 core developers and 1,700 contributors, this portable middleware has evolved to encapsulate and augment a wide range of native OS capabilities essential to support performance-driven software systems.

The ACE Programmer's Guide is a practical, hands-on guide to ACE for C++ programmers building networked

applications and next-generation middleware. The book first introduces ACE to beginners. It then explains how you can tap design patterns, frameworks, and ACE to produce effective, easily maintained software systems with less time and effort. The book features discussions of programming aids, interprocess communication (IPC) issues, process and thread management, shared memory, the ACE Service Configurator framework, timer management classes, the ACE Naming Service, and more.



• Table of Contents

ACE Programmer's Guide, The: Practical Design Patterns for Network and Systems Programming

By Stephen D. Huston, James CE Johnson, Umar Syyid

Start Reading

Publisher: Addison Wesley
Pub Date: November 14, 2003
ISBN: 0-201-69971-0
Pages: 544

Copyright
Illustrations
Tables
Foreword
Preface
Who Should Read This Book
Organization
Conventions Used in This Book
Acknowledgments
Concluding Remarks
Part I. ACE Basics
Chapter 1. Introduction to ACE
Section 1.1. A History of ACE
Section 1.2. ACE's Benefits
Section 1.3. ACE's Organization
Section 1.4. Patterns, Class Libraries, and Frameworks
Section 1.5. Porting Your Code to Multiple Operating Systems
Section 1.6. Smoothing the Differences among C++ Compilers
Section 1.7. Using Both Narrow and Wide Characters
Section 1.8. Where to Find More Information and Support
Section 1.9. Summary
Chapter 2. How to Build and Use ACE in Your Programs
Section 2.1. A Note about ACE Versions
Section 2.2. Guide to the ACE Distribution
Section 2.3. How to Build ACE
Section 2.4. How to Include ACE in Your Applications
Section 2.5. How to Build Your Applications
Section 2.6. Summary
Chapter 3. Using the ACE Logging Facility

Section 3.1. Basic Logging and Tracing
Section 3.2. Enabling and Disabling Logging Severities
Section 3.3. Customizing the ACE Logging Macros
Section 3.4. Redirecting Logging Output
Section 3.5. Using Callbacks
Section 3.6. The Logging Client and Server Daemons
Section 3.7. The LogManager Class
Section 3.8. Runtime Configuration with the ACE Logging Strategy
Section 3.9. Summary
Chapter 4. Collecting Runtime Information
Section 4.1. Command Line Arguments and ACE_Get_Opt
Section 4.2. Accessing Configuration Information
Section 4.3. Building Argument Vectors
Section 4.4. Summary
Chapter 5. ACE Containers
Section 5.1. Container Concepts
Section 5.2. Sequence Containers
Section 5.3. Associative Containers
Section 5.4. Allocators
Section 5.5. Summary
Part II. Interprocess Communication
Chapter 6. Basic TCP/IP Socket Use
Section 6.1. A Simple Client
Section 6.2. Adding Robustness to a Client
Section 6.3. Building a Server
Section 6.4. Summary
Chapter 7. Handling Events and Multiple I/O Streams
Section 7.1. Overview of the Reactor Framework
Section 7.2. Handling Multiple I/O Sources
Section 7.3. Signals
Section 7.4. Notifications
Section 7.5. Timers
Section 7.6. Using the Acceptor-Connector Framework
Section 7.7. Reactor Implementations
Section 7.8. Summary
Chapter 8. Asynchronous I/O and the ACE Proactor Framework
Section 8.1. Why Use Asynchronous I/O?
Section 8.2. How to Send and Receive Data
Section 8.3. Establishing Connections
Section 8.4. The ACE_Proactor Completion Demultiplexer
Section 8.5. Using Timers
Section 8.6. Other I/O Factory Classes
Section 8.7. Combining the Reactor and Proactor Frameworks
Section 8.8. Summary
Chapter 9. Other IPC Types
Section 9.1. Interhost IPC with UDP/IP
Section 9.2. Intrahost Communication
Section 9.3. Summary
Part III. Process and Thread Management
Chapter 10. Process Management
Section 10.1. Spawning a New Process
Section 10.2. Using the ACE_Process_Manager
Section 10.3. Synchronization Using ACE_Process_Mutex
Section 10.4. Summary

- Chapter 11. Signals
  - Section 11.1. Using Wrappers
  - Section 11.2. Event Handlers
  - Section 11.3. Guarding Critical Sections
  - Section 11.4. Signal Management with the Reactor
  - Section 11.5. Summary
- Chapter 12. Basic Multithreaded Programming
  - Section 12.1. Getting Started
  - Section 12.2. Basic Thread Safety
  - Section 12.3. Intertask Communication
  - Section 12.4. Summary
- Chapter 13. Thread Management
  - Section 13.1. Types of Threads
  - Section 13.2. Priorities and Scheduling Classes
  - Section 13.3. Thread Pools
  - Section 13.4. Thread Management Using ACE_Thread_Manager
  - Section 13.5. Signals
  - Section 13.6. Thread Start-Up Hooks
  - Section 13.7. Cancellation
  - Section 13.8. Summary
- Chapter 14. Thread Safety and Synchronization
  - Section 14.1. Protection Primitives
  - Section 14.2. Thread Synchronization
  - Section 14.3. Thread-Specific Storage
  - Section 14.4. Summary
- Chapter 15. Active Objects
  - Section 15.1. The Pattern
  - Section 15.2. Using the Pattern
  - Section 15.3. Summary
- Chapter 16. Thread Pools
  - Section 16.1. Understanding Thread Pools
  - Section 16.2. Half-Sync/Half-Async Model
  - Section 16.3. Leader/Followers Model
  - Section 16.4. Thread Pools and the Reactor
  - Section 16.5. Summary

Part IV. Advanced ACE
- Chapter 17. Shared Memory
  - Section 17.1. ACE_Malloc and ACE_Allocator
  - Section 17.2. Persistence with ACE_Malloc
  - Section 17.3. Position-Independent Allocation
  - Section 17.4. ACE_Malloc for Containers
  - Section 17.5. Wrappers
  - Section 17.6. Summary
- Chapter 18. ACE Streams Framework
  - Section 18.1. Overview
  - Section 18.2. Using a One-Way Stream
  - Section 18.3. A Bidirectional Stream
  - Section 18.4. Summary
- Chapter 19. ACE Service Configurator Framework
  - Section 19.1. Overview
  - Section 19.2. Configuring Static Services
  - Section 19.3. Setting Up Dynamic Services
  - Section 19.4. Setting Up Streams
  - Section 19.5. Reconfiguring Services During Execution

Section 19.6. Using XML to Configure Services and Streams
Section 19.7. Configuring Services without svc.conf
Section 19.8. Singletons and Services
Section 19.9. Summary
Chapter 20. Timers
Section 20.1. Timer Concepts
Section 20.2. Timer Queues
Section 20.3. Prebuilt Dispatchers
Section 20.4. Managing Event Handlers
Section 20.5. Summary
Chapter 21. ACE Naming Service
Section 21.1. The ACE_Naming_Context
Section 21.2. A Single-Process Naming Context : PROC_LOCAL
Section 21.3. Sharing a Naming Context on One Node: NODE_LOCAL
Section 21.4. Sharing a Naming Context across the Network: NET_LOCAL
Section 21.5. Summary
Bibliography
CD-ROM Warranty

Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Figures 7.1, 7.2, 7.3, and 8.1 originally published in Schmidt/Huston, C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks, Copyright © 2003 by Pearson Education, Inc. Reprinted with permission of Pearson Education, Inc.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Huston, Stephen D.
The ACE programmer's guide : practical design patterns for network and systems programming / Stephen D. Huston, James CE Johnson and Umar Syyid.
p. cm.
ISBN 0-201-69971-0 (pbk. : alk. paper)
1. Computer software—Development. 2. Object-oriented programming (Computer science) 3. Software patterns. I. Johnson, James C. E. II. Syyid, Umar. III. Title.

QA76.76.D47H89 2003
005.1'17—dc21 2003014046

Copyright © 2004 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc,
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116

Illustrations

7.1 Acceptor-Connector framework classes

7.2 Steps in ACE_Acceptor connection acceptance

7.3 Reactive shutdown of an ACE_Svc_Handler

8.1 Classes in the Proactor framework

8.2 Sequence diagram for asynchronous data echo example

12.1 Using a queue for communication

13.1 Kernel- and User-Level Threads

15.1 Asynchronous log() method

15.2 Active Object pattern participants

15.3 Active Object collaborations

15.4 Active Object pattern applied to example

18.1 Diagram of an ACE_Stream

18.2 Logical structure of the command stream

20.1 Timer queue class hierarchy

20.2 Timer dispatcher example class diagram

20.3 Timer dispatcher example sequence diagram

20.4 Timer queue template classes

20.5 Upcall handler sequence diagram

Tables

1.1 Data Types Defined by ACE

1.2 ACE Memory Allocation Macros

1.3 Macros Related to Character Width

2.1 Configuration Files for Popular ACE Platforms

2.2 GNU make Options

2.3 ACE Library File Names for Visual C++

3.1 ACE_Log_Msg Logging Severity Levels

3.2 ACE Logging Format Directives

3.3 ACE Logging Macros

3.4 Commonly Used ACE_Log_Msg Methods

3.5 Valid ACE_Log_Msg Flags Values

3.6 Mapping ACE Logging Severity to Windows Event Log Severity

3.7 ACE_Log_Record Attributes

3.8 ACE Logging Strategy Configuration Options

5.1 Allocators Available in ACE

7.1 Meaning of Event Handler Callback Return Values

12.1 ACE Guard Classes

12.2 Various Queue Types

Foreword

I started programming computers in the mid-1980s. Then, like today, computing and communication systems were heterogeneous; that is, there were many different programming languages, operating systems, and networking protocols. Moreover, due to the accidental complexities of programming with low-level and nonportable application programming interfaces (APIs), the types of networked applications available to software developers and end users were relatively primitive, consisting largely of e-mail, remote login, and file transfer. As a result, many applications were either centralized on mainframes and minicomputers or localized on a single stand-alone PC or workstation.

There have been significant advances in information technology (IT) for computing and communication during the intervening two decades. Languages, programming environments, operating systems, networking protocols, and middleware are more mature and standardized. For example, C++ has become an ISO/ANSI standard and is being used in a broad range of application domains. Likewise, the UNIX, Windows, and TCP/IP standards have also become ubiquitous. Even distributed computing middleware standards, such as CORBA, .NET, and J2EE, are becoming widely embraced by the IT industry and end users.

Even with all these advances, however, there is still a layer of the networked software design space—host infrastructure middleware for performance-driven, multiplatform, networked and/or concurrent software systems—that is not well served by standard solutions at other levels of abstraction. Host infrastructure middleware encapsulates native operating system (OS) concurrency and interprocess communication (IPC) mechanisms to alleviate many tedious, error-prone, and nonportable activities associated with developing networked applications via native OS APIs, such as Sockets or POSIX threads (Pthreads). Performance-driven systems also have stringent quality of service (QoS) requirements.

Providing host infrastructure middleware for today's popular computing environments is particularly important for (1) high-performance computing systems, such as those that support scientific visualization, distributed database servers, and online financial trading systems; (2) distributed real-time and embedded systems that monitor and control real-world artifacts, such as avionics mission- and flight-control software, supervisory control and data acquisition systems, and automotive braking systems; and (3) multiplatform applications that must run portably across local- and wide-area networks. These types of systems are increasingly subject to the following trends:

- The need to access native OS mechanisms to meet QoS requirements. For example, multimedia applications that require long-duration, bidirectional bytestream communication services are poorly suited to the synchronous request/response model provided by conventional distribution middleware. Likewise, many distribution middleware implementations incur significant overhead and lack sufficient hooks to manipulate other QoS-related properties, such as latency, throughput, and jitter.
- Severe cost and time-to-market pressures. Global competition and market deregulation are shrinking budgets for the in-house development and quality assurance (QA) of software, particularly for the OS and middleware infrastructure. Moreover, performance-driven users are often unable or less willing to pay for specialized proprietary infrastructure software.
- The demand for user-specific customization. Because performance-intensive software often pushes the limits of technology, it must be optimized for particular runtime contexts and application requirements. General-purpose, one-size-fits-all software solutions often have unacceptable performance.

As these trends continue to accelerate, they present many challenges to developers of tomorrow's networked software systems. In particular, to succeed in today's competitive, fast-paced computing industry, successful middleware and application software must exhibit the following eight characteristics:

1. Affordability, to ensure that the total ownership costs of software acquisition and evolution aren't prohibitively

Preface

ACE (the ADAPTIVE Communication Environment) is a powerful C++ toolkit that helps you develop portable, high-performance applications, especially networked and/or multithreaded applications, more easily and more quickly with more flexibility and fewer errors. And, because of ACE's design and frameworks, you can do all this with much less code than with other development approaches. We've been using ACE for years and have found it exceedingly helpful and well worth any price. What's better is that it's available for free! The historical price many developers have paid to use ACE is a steep learning curve. It's a big toolkit with a large set of capabilities. Until recently, the best documentation has been the source code, which is, of course, freely available, and a set of academic papers born of the research that produced ACE, approachable only by advanced professionals and upper-level students. This barrier to learning has kept ACE's power and easy-to-use elegance one of the best-kept secrets in software development. That's why we're very excited to write this book! It flattens out ACE's learning curve, bringing ACE's power, elegance, and capabilities to all.

This book teaches you about ACE: a bit of its history and approach to development, how it's organized, how to begin using it, and also how to use some of its more advanced capabilities. We teach you how to do things the ACE way in this book, but we could not possibly fit in a complete reference. Use this book to get started with ACE and to begin using it in your work. If you've been using ACE for a while, there are probably descriptions of some capabilities you haven't seen before, so the book is useful for experienced ACE users as well.

Who Should Read This Book

This book is meant to serve as both an introductory guide for ACE beginners and a quickly accessible review for experienced ACE users. If you are an ACE beginner, we recommend starting at the beginning and proceeding through the chapters in order. If you are experienced and know what you want to read about, you can quickly find that part of the book and do not need to read the previous sections.

This book is written for C++ programmers who have been exposed to some of the more advanced C++ features, such as virtual inheritance and class templates. You should also have been exposed to basic operating system facilities you plan to use in your work. For example, if you plan to write programs that use TCP/IP sockets, you should at least be familiar with the general way sockets are created, connections are established, and data is transferred.

This book is also an excellent source of material for those who teach others: in either a commercial or an academic setting. ACE is an excellent example of how to design object-oriented software and use C++ to design and write high-performance, easily maintained software systems.

Organization

This book is a hands-on, how-to guide to using ACE effectively. The many source code examples illustrate proper use of the pieces of ACE being described. The source code examples are kept fairly short and to the point. Sometimes, the example source is abridged in order to focus attention on a topic. The complete source code to all examples is on the included CD-ROM and is also available on Riverace Corporation's web site. The included CD-ROM also includes a copy of ACE's source kit, installable versions of ACE prebuilt for a number of popular platforms, and complete reference documentation for all the classes in ACE.

The book begins with basic areas of functionality that many ACE users need and then proceeds to build on the foundations, describing the higher-level features that abstract behavior out into powerful patterns.

- Part I introduces ACE and provides some generally useful information about the facilities ACE provides. Part I also explains how to configure and build ACE, as well as how to build your applications that use ACE. Widely used programming aids, such as logging and tracing, command line processing and configuration access, and ACE's container classes, are also described.

- Part II discusses ACE's facilities for interprocess communication (IPC), beginning with basic, low-level TCP/IP Sockets wrapper classes and proceeding to show how to handle multiple sockets, as well as other events, such as timers and signals, simultaneously using ACE's Reactor and Proactor frameworks. Part II also describes ACE's Acceptor-Connector framework and then ends with a discussion of some of the other IPC wrapper classes ACE offers, many of which are substitutable for TCP/IP wrapper classes in the covered frameworks.

- Part III covers a wide range of topics related to process and thread management using ACE. This part explains how to use ACE's process management classes and then covers signals, followed by three chapters about multithreaded programming, thread management, and the critical areas of thread safety and synchronization. Part III ends with discussions of Active Objects and various ways to use thread pools in ACE—critical topics for effective use of multithreading in high-performance applications.

- Part IV covers advanced ACE topics: shared memory, the ACE Streams framework for assembling modular data-processing streams, and how to make your applications more flexible and configurable by using the ACE Service Configurator framework. Part IV concludes with an in-depth discussion of ACE's timer management classes and the ACE Naming Service, one of ACE's network services components to assist with often needed networked application programming tasks.

The book concludes with a bibliography and an extensive subject index.

Conventions Used in This Book

All ACE classes begin with ACE_. When we refer to patterns instead of the classes they implement, we omit the prefix. For example, the Reactor pattern is implemented by the ACE_Reactor class.

All class member variables are suffixed with '_'. This convention is used in the ACE sources, and we carry it through to the examples in this book as well.

C++ code and file names are set in this font. Command lines are set in this font.

Acknowledgments

We are indebted to the review team that read and commented on the entire manuscript. Craig L. Ching, Dave Mercer, Johnny Willemsen, and Steven P. Witten provided insightful and helpful feedback and ideas that improved the book greatly.

We are also grateful to the members of the ACE user community worldwide who volunteered their free time to review a number of manuscript drafts and provide helpful corrections and advice. Like the many contributors to ACE itself, these individuals show the cooperative nature of many Open-Source developer/user communities, and the ACE users in particular: Bill Cassanova, Ufuk _oban, Todd Cooper, Ning Cui, Alain Decamps, John Fowler, Chris D. Gill, Kelly F. Hickel, Don Hinton, Robert Kindred, Michael Kleck, Franz Klein, Sven Kצster, Dieter Knppel, Theo Landman, Mark Levan, Alexander Libman, John Lilley, Stephen McDonald, Mike Mullen, Mats Nilsson, Jaroslaw Nozderko, Rick Ohnemus, Wojtek Pilorz, Sunanda C. Prasad, Dietrich Quehl, Irma Rastegayeva, Michael Searles, Rich Siebel, Chris Smith, Scott Smith, Edward Thompson, Alain Totouom, Bill Trudell, and Lothar Werzinger.

Our editorial team at Addison-Wesley was very helpful and encouraging during this long process as well. Thanks to our editors: Marina Lang, Debbie Lafferty, Peter Gordon, and Bernie Gaffney. Thank you to our copy editor, Evelyn Pyle, who did a marvelous job molding our differing approaches and styles into a unified whole. Many, many thanks to our production coordinator, Elizabeth Ryan, who has now ushered all three ACE-related books through production with great skill, grace, and patience.

Steve's Acknowledgments

As in all my work, I am deeply indebted to my wonderful wife, Jane, for her abiding love, constant support, and much-needed help during the long process of writing this book. You sacrificed much more than anyone should ever have to and are truly God's gift to me—thank you! As wise Solomon once said, "Of making many books there is no end, and much study wearies the body" (Ecclesiastes 12:12, NIV). I'm a bit weary and thankful to God for the energy to complete this work. I'm also grateful to my late mother, Karen L. Anderson, who would be pleased with the lessons I've finally learned.

James's Acknowledgments

I would like to thank my wife, Karla, and my son, Riley, who was born half-way through this four-year process, for their patience and understanding every time I disappeared into the basement office for hours on end. Without their constant support and encouragement, I don't think I would have ever made it to this point. I would also like to thank my parents for always encouraging me to put forth my best effort and to never settle for second best. And, finally, thanks to Doug Schmidt for creating ACE in the first place and giving us all the opportunity to create this text.

Umar's Acknowledgments

First of all I would like to thank my wife, Ambreen, and my son, Hassan. If it weren't for them, I might not leave the computer at all for a few days; thanks for calling me home. Ambreen deserves my special thanks for enduring my constant babble over the years. Without her support, I would not have written a single page. I would also like to thank my mother for her courage and support during the more difficult times in my life. In addition, I would like to thank my father for making me want to show him that I could. Finally, thanks to Doug Schmidt for driving down to the University of Illinois to provide a graduate seminar that first taught me about a framework called ACE.

Concluding Remarks

This book is a true coauthorship, with each of us writing equal amounts of the text. Therefore, we all share equally in the blame for any problems you may find in the text. Please report anything you do find to either the ACE users mailing list (ace-users@cs.wustl.edu) or to us at ace-tutorial@tragus.org. Writing a book on a topic as broad as ACE is a very difficult task. The fact that we had a team of authors made the task considerably easier. In addition, the help available from the DOC group and the ACE user community has proved invaluable.

Steve Huston
James CE Johnson
Umar Syyid

Part I: ACE Basics

Chapter 1. Introduction to ACE

Chapter 2. How to Build and Use ACE in Your Programs

Chapter 3. Using the ACE Logging Facility

Chapter 4. Collecting Runtime Information

Chapter 5. ACE Containers

Chapter 1. Introduction to ACE

ACE is a rich and powerful toolkit with a distinctively nontraditional history. To help you get a feel for where ACE comes from, this chapter gives a brief history of ACE's development. Before going into programming details, we cover some foundational concepts: class libraries, patterns, and frameworks. Then we cover one of ACE's less glamorous but equally useful aspects: its facilities for smoothing out the differences among operating systems, hardware architectures, C++ compilers, and C++ runtime support environments. It is important to understand these aspects of ACE before we dive into the rest of ACE's capabilities.

1.1 A History of ACE

ACE (the ADAPTIVE—A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment—Communication Environment) grew out of the research and development activities of Dr. Douglas C. Schmidt at the University of California at Irvine. Doug's work was focused on design patterns, implementation, and experimental analysis of object-oriented techniques that facilitate the development of high-performance, real-time distributed object computing frameworks. As is so often the case, that work began on a small handful of hardware and OS (operating system) platforms and expanded over time. Doug ran into the following problems that still vex developers today.

- Trying to rearrange code to try out new combinations of processes, threads, and communications mechanisms is very tedious. The process of ripping code apart, rearranging it, and getting it working again is very boring and error prone.
- Working at the OS API (application programming interface) level adds accidental complexity. Identifying the subtle programming quirks and resolving these issues takes an inordinate amount of time in development schedules.
- Successful projects need to be ported to new platforms and platform versions. Standards notwithstanding, no two platforms or versions are the same, and the complexities begin again.

Doug, being the smart guy he is, came up with a plan. He invented a set of C++ classes that worked together, allowing him to create templates and strategize his way around the bumps inherent in rearranging the pieces of his programs. Moreover, these classes implemented some very useful design patterns for isolating the changes in the various computing platforms he kept getting, making it easy for his programs to run on all the new systems his advisors and colleagues could throw at him. Life was good. And thus was born ACE.

At this point, Doug could have grabbed his Ph.D. and some venture funding and taken the world of class libraries by storm. We'd all be paying serious money for the use of ACE today. Fortunately for the world's C++ developers, Doug decided to pursue further research in this area. He created the large and successful Distributed Object Computing group at Washington University in St. Louis. ACE was released as Open Source,[1] and its development continued with the aid of more than US$7 million funding from visionary commercial sponsors.

[1] If you'd like more information on Open Source Software in general, please visit the Open Source Initiative page on the web at http://www.opensource.org/.

Today, most of the research-related work on ACE is carried out at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, where Doug is now a professor and still heavily involved in ACE's day-to-day development. ACE is increasingly popular in commercial development as well. It enjoys a reputation for solving the cross-platform issues that face all high-performance networked application and systems developers. Additionally, commercial-level technical support for ACE is now available worldwide.

ACE is freely available for any use, including commercial, without onerous licensing requirements. For complete details, see the COPYING file in the top-level directory of the ACE source kit. Another interesting file in the top-level directory is THANKS. It lists all the people who have contributed to ACE over the years—more than 1,700 at the time of this writing! If you contribute improvements or additions to ACE, you too can get your name listed.

1.2 ACE's Benefits

The accidental complexity and development issues that led to ACE reflect lessons learned by all of us who've written networking programs, especially portable, high-performance networking programs.

- Multiplatform source code is difficult to write. The many and varied standards notwithstanding, each new port introduces a new set of problems. To remain flexible and adaptive to new technology and business conditions, software must be able to quickly change to take advantage of new operating systems and hardware. One of the best ways to remain that flexible is to take advantage of libraries and middleware that absorb the differences for you, allowing your code to remain above the fray. ACE has been ported to a wide range of systems—from handheld to supercomputer—running a variety of operating systems and using many of the available C++ compilers.
- Networked applications are difficult to write. Introducing a network into your design introduces a large set of challenges and issues throughout the development process. Network latency, byte ordering, data structure layout, network instability and performance, and handling multiple connections smoothly are a few typical problems that you must be concerned with when developing networked applications. ACE makes dozens of frameworks and design pattern implementations available to you, so you can take advantage of solutions that many other smart people have already come up with.
- Most system-provided APIs are written in terms of C API bindings because C is so common and the API is usable from a variety of languages. If you're considering working with ACE, you've already committed to using C++, though, and the little tricks that seemed so slick with C, such as overlaying the sockaddr structure with sockaddr_in, sockaddr_un, and friends, are just big opportunities for errors. Low-level APIs have many subtle requirements, which sometimes change from standard to standard—as in the series of drafts leading to POSIX 1004.1c Pthreads—and are sometimes poorly documented and difficult to use. (Remember when you first figured out that the socket(), bind(), listen(), and accept() functions were all related?) ACE provides a well-organized, coherent set of abstractions to make IPC, shared memory, threads, synchronization mechanisms, and more easy to use.

1.3 ACE's Organization

ACE is more than a class library. It is a powerful, object-oriented application toolkit. Although it is not apparent from a quick look at the source or reference pages, the ACE toolkit is designed using a layered architecture composed of the following layers:

- OS adaptation layer. The OS adaptation layer provides wrapper functions for most common system-level operations. This layer provides a common base of system functions across all the platforms ACE has been ported to. Where the native platform does not provide a desired function, it is emulated, if possible. If the function is available natively, the calls are usually inlined to maximize performance.
- Wrapper facade layer. A wrapper facade consists of one or more classes that encapsulate functions and data within a type-safe, object-oriented interface [5]. ACE's wrapper facade layer makes up nearly half of its source base. This layer provides much of the same capability available natively and via the OS adaptation layer but in an easier-to-use, type-safe form. Applications make use of the wrapper facade classes by selectively inheriting, aggregating, and/or instantiating them.
- Framework layer. A framework is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications [4] [7]. Frameworks emphasize the integration and collaboration of application-specific and application-independent components. By doing this, frameworks enable larger-scale reuse of software rather than simply reusing individual classes or stand-alone functions. ACE's frameworks integrate wrapper facade classes and implement canonical control flows and class collaborations to provide semicomplete applications. It is very easy to build applications by supplying application-specific behavior to a framework.
- Networked services layer. The networked services layer provides some complete, reusable services, including the distributed logging service we'll see in Chapter 3.

Each layer reuses classes in lower layers, abstracting out more common functionality. This means that any given task in ACE can usually be done in more than one way, depending on your needs and design constraints. Although we sometimes approach a problem "bottom up" because existing knowledge helps transition smoothly to ACE programming techniques, we generally cover the higher-level ways rather than the lower-level ways to approach problems in more depth. It is relatively easy, however, to find the lower-level classes and interfaces for your use when needed. The ACE reference documentation on the included CD-ROM can be used to find all class relationships and complete programming references.

1.4 Patterns, Class Libraries, and Frameworks

Computing power and network bandwidth have increased dramatically over the past decade. However, the design and implementation of complex software remains expensive and error prone. Much of the cost and effort stem from the continuous rediscovery and reinvention of core concepts and components across the software industry. In particular, the growing heterogeneity of hardware architectures and diversity of operating system and communication platforms make it difficult to build correct, portable, efficient, and inexpensive applications from scratch. Patterns, class libraries, and frameworks are three tools the software industry is using to reduce the complexity and cost of developing software.

Patterns represent recurring solutions to software development problems within a particular context. Patterns and frameworks both facilitate reuse by capturing successful software development strategies. The primary difference is that frameworks focus on reuse of concrete designs, algorithms, and implementations in a particular programming language. In contrast, patterns focus on reuse of abstract designs and software microarchitectures.

Frameworks can be viewed as a concrete implementation of families of design patterns that are targeted for a particular application domain. Likewise, design patterns can be viewed as more abstract microarchitectural framework elements that document and motivate the semantics of frameworks in an effective way. When patterns are used to structure and document frameworks, nearly every class in the framework plays a well-defined role and collaborates effectively with other classes in the framework.

Like frameworks, class libraries are implementations of useful, reusable software artifacts. Frameworks extend the benefits of OO (object-oriented) class libraries in the following ways.

- Frameworks define "semicomplete" applications that embody domain-specific object structures and functionality. Components in a framework work together to provide a generic architectural skeleton for a family of related applications. Complete applications are composed by inheriting from and/or instantiating framework components. In contrast, class libraries are less domain specific and provide a smaller scope of reuse. For instance, class library components, such as classes for strings, complex numbers, arrays, and bitsets, are relatively low level and ubiquitous across many application domains.
- Frameworks are active and exhibit "inversion of control" at runtime. Class libraries are typically passive; that is, they are directed to perform work by other application objects, in the same thread of control as those application objects. In contrast, frameworks are active; that is, they direct the flow of control within an application via event dispatching patterns, such as Reactor and Observer. The "inversion of control" in the runtime architecture of a framework is often referred to as the Hollywood Principle: "Don't call us; we'll call you."

In practice, frameworks and class libraries are complementary technologies. For instance, frameworks typically use class libraries internally to simplify the development of the framework. (Certainly, ACE's frameworks reuse other parts of the ACE class library.) Likewise, application-specific code invoked by framework event handlers can use class libraries to perform such basic tasks as string processing, file management, and numerical analysis.

So, to summarize, ACE is a toolkit packaged as a class library. The toolkit contains many useful classes. Many of those classes are related and combined into frameworks, such as the Reactor and Event Handler, that embody semicomplete applications. In its classes and frameworks, ACE implements many useful design patterns.

1.5 Porting Your Code to Multiple Operating Systems

Many software systems must be designed to work correctly on a range of computing platforms. Competitive forces and changing technology combine to make it a nearly universal requirement that today's software systems be able to run on a range of platforms, sometimes changing targets during development. Networked systems have a stronger need to be portable, owing to the inherent mix of computing environments needed to build and configure competitive systems in today's marketplace. Standards provide some framework for portability; however, marketing messages notwithstanding, standards do not guarantee portability across systems. As Andrew Tanenbaum said, "The nice thing about standards is that there are so many to choose from" [3]. And rest assured, vendors often choose to implement different standards at different times. Standards also change and evolve, so it's very unlikely that you'll work on more than one platform that implements all the same standards in the same way.

In addition to operating system APIs and their associated standards, compiling and linking your programs and libraries is another area that differs among operating systems and compilers. The ACE developers over the years have developed an effective system of building ACE based on the GNU Make tool. Even on systems for which a make utility is supplied by the vendor, not all makes are created equal. GNU Make provides a common, powerful tool around which ACE has its build facility. This allows ACE to be built on systems that don't supply native make utilities but to which GNU Make has been ported. Don't worry if you're a Windows programmer using Microsoft or Borland compilers and don't have GNU Make. The native Microsoft and Borland build utilities are supported as well.

Data type differences are a common area of porting difficulty that experienced multiplatform developers have found numerous ways to engineer around. ACE provides a set of data types it uses internally, and you are encouraged to use them as well. These are described later, in the discussion of compilers.

ACE's OS adaptation layer provides the lowest level of functionality and forms the basis of ACE's wide range of portability. This layer uses the Wrapper Facade [4] and Fa¬ade [3] patterns to shield you from platform differences. The Wrapper pattern forms a relatively simple wrapper around a function, and ACE uses this pattern to unify the programming interfaces for common system functions where small differences in APIs and semantics are smoothed over. The Fa¬ade pattern presents a single interface to what may, on some platforms, be a complicated set of systems calls. For example, the ACE_OS::thr_create() method creates a new thread with a caller-specified set of attributes: scheduling policy, priority, state, and so on. The native system calls to do all that's required during thread creation vary widely among platforms in form, semantics, and the combination and order of calls needed. The Fa¬ade pattern allows the presentation of one consistent interface across all platforms to which ACE has been ported.

For relatively complex and obviously nonportable actions, such as creating a thread, you would of course think to use the ACE_OS methods—well, at least until you read about the higher-level ACE classes and frameworks later in the book. But what about other functions that are often taken for granted, such as printf() and fseek()? Even when you need to perform a basic function, it is safest to use the ACE_OS methods rather than native APIs. This usage guarantees that you won't be surprised by a small change in arguments to the native calls when you compile your code on a different platform.

The ACE OS adaptation layer is implemented in the ACE_OS class. The methods in this class are all static. You may wonder why a separate namespace wasn't used instead of a class, as that's basically what the class achieves. As we'll soon see, one of ACE's strengths is that it works with a wide variety of old and new C++ compilers, some of which do not support namespaces at all. We are not going to list all the supplied functions here. You won't often use these functions directly. They are still at a very low level, not all that much different from writing in C. Rather, you'll more often use high-level classes that themselves call ACE_OS methods to perform the requested actions. Therefore, we're going to leave a detailed list of these methods to the ACE_OS reference pages.

As you might imagine, ACE contains quite a lot of conditionally compiled code, especially in the OS adaptation layer. ACE does not make heavy use of vendor-supplied compiler macros for this, for a couple of reasons. First, a number of the settings deal with features that are missing or broken in the native platform or compiler. The missing or broken features may change over time; for instance, the OS is fixed or the compiler is updated. Rather than try to find a vendor-supplied macro for each possible item and use those macros in many places in ACE, any setting and vendor-supplied macro checking is done in one place and the result remembered for simple use within ACE. The second reason that vendor-supplied macros are not used extensively is that they may either not exist or may conflict

1.6 Smoothing the Differences among C++ Compilers

You're probably wondering why it's so important to smooth over the differences among compilers, as the C++ standard has finally settled down. There are a number of reasons.

- Compiler vendors are at various levels of conformance to the standard.
- ACE works with a range of C++ compilers, many of which are still at relatively early drafts of the C++ standard.
- Some compilers are simply broken, and ACE works around the problems.

Many items of compiler capability and disability adjustment are used to build ACE properly, but from a usage point of view, ACE helps you work with or around four primary areas of compiler differences:

1. Templates, both use and instantiation
2. Data types, both hardware and compiler
3. Runtime initialization and shutdown
4. Allocating heap memory

1.6.1 Templates

C++ templates are a powerful form of generic programming, and ACE makes heavy use of them, allowing you to combine and customize functionality in powerful ways. This brief discussion introduces you to class templates. If templates are a new feature to you, we suggest that you also read an in-depth discussion of their use in a good C++ book, such as Bjarne Stroustrup's The C++ Programming Language, 3rd Edition [11] or C++ Templates: The Complete Guide by David Vandevoorde and Nicolai M. Josuttis [14]. As you're reading, keep in mind that the dialect of C++ described in your book and that implemented by your compiler may be different. Check your compiler documentation for details, and stick to the guidelines documented in this book to be sure that your code continues to build and run properly when you change compilers.

C++'s template facility allows you to generically define a class or function, and have the compiler apply your template to a given set of data types at compile time. This increases code reuse in your project and enables you to reuse code across projects. For example, let's say that you need to design a class that remembers the largest value of a set of values you specify over time. Your system may have multiple uses for such a class—for example, to track integers, floating-point numbers, text strings, and even other classes that are part of your system. Rather than write a separate class for each possible type you want to track, you could write a class template:

```
template <class T> class max_tracker {
public:
  void track_this (const T val);
private:
  T max_val_;
};
```

1.7 Using Both Narrow and Wide Characters

Developers outside the United States are acutely aware that many character sets in use today require more than one byte, or octet, to represent each character. Characters that require more than one octet are referred to as "wide characters." The most popular wide-character standard is ISO/IEC 10646, the Universal Multiple-Octet Coded Character Set (UCS). Unicode is a separate standard but can be thought of as a restricted UCS subset that uses two octets for each character (UCS-2). Many Windows programmers are familiar with Unicode.

C++ represents wide characters with the wchar_t type, which enables methods to offer multiple signatures that are differentiated by their character type. Wide characters have a separate set of C string manipulation functions, however, and existing C++ code, such as string literals, requires change for wide-character usage. As a result, programming applications to use wide-character strings can become expensive, especially when applications written initially for U.S. markets must be internationalized for other countries. To improve portability and ease of use, ACE uses C++ method overloading and the macros described in Table 1.3 to use different character types without changing APIs.

For applications to use wide characters, ACE must be built with the ACE_HAS_WCHAR configuration setting, which most modern platforms are capable of. Moreover, ACE must be built with the ACE_USES_WCHAR setting if ACE should also use wide characters internally. The ACE_TCHAR and ACE_TEXT macros are illustrated in examples throughout this book.

Table 1.2. ACE Memory Allocation Macros

Macro	Action
ACE_NEW(p, c)	Allocate memory by using constructor c and assign pointer to p. On failure, p is 0 and return;
ACE_NEW_RETURN(p, c, r)	Allocate memory by using constructor c and assign pointer to p. On failure, p is 0 and return r;
ACE_NEW_NORETURN(p, c)	Allocate memory by using constructor c and assign pointer to p. On failure, p is 0 and control continues at next statement.

Table 1.3. Macros Related to Character Width

Macro	Purpose
ACE_HAS_WCHAR	Configuration setting that enables ACE's wide-character methods
ACE_USES_WCHAR	Configuration setting that directs ACE to use wide characters internally
ACE_TCHAR	Matches ACE's internal character width; defined as either char or wchar_t, depending on the lack or presence of ACE_USES_WCHAR
ACE_TMAIN	Properly defines program's main entry point for command line argument type, based on

1.8 Where to Find More Information and Support

As you're aware by the number and scope of items ACE helps with—from compiling to runtime—and from a perusal of the Contents—not to mention the size of this book!—you may be starting to wonder where you can find more information about ACE, how to use all its features, and where to get more help. A number of resources are available: ACE reference documentation, ACE kits, user forums, and technical support.

The ACE reference documentation is generated from specially tagged comments in the source code itself, using a tool called Doxygen.[5] That reference documentation is available in the following places:

[5] For more information on Doxygen, please see http://www.doxygen.org/.

- On the CD-ROM included with this book
- At Riverace's web site: http://www.riverace.com/docs/
- At the DOC group's web site, which includes the reference documentation for ACE and TAO (The ACE ORB) for the most recent stable version, the most recent beta version, and the nightly development snapshot: http://www.dre.vanderbilt.edu/Doxygen/

ACE is, of course, freely available in a number of forms from the following locations.

- The complete ACE 5.3b source and prebuilt kits for selected platforms are on the included CD-ROM.
- The source and kits that are on the CD-ROM are also available, along with older versions, at Riverace's web site: http://www.riverace.com.
- Complete sources for the current release, current BFO (bug fix only) beta, and the most recent beta versions of ACE are available from the DOC group's web site: http://deuce.doc.wustl.edu/Download.html. The various types of kits are explained in Section 2.1.

A number of ACE developer forums are available via e-mail and Usenet news. The authors and the ACE developers monitor traffic to these forums. These people, along with the ACE user community at large, are very helpful with questions and problems. If you post a question or a problem to any of the following forums, please include the information in the PROBLEM-REPORT-FORM file located in the top-level directory of the ACE source kit:

- comp.soft-sys.ace newsgroup.
- ace-users@cs.wustl.edu mailing list. To join this list, send a request to ace-users-request@cs.wustl.edu. Include the following command in the body of the e-mail:

```
subscribe ace-users [emailaddress@domain]
```

You must supply emailaddress@domain only if your message's From address is not the address you wish to subscribe. If you use this alternative address method, the list server will require an extra authorization step

1.9 Summary

This chapter introduced the ACE toolkit's organization and described some foundational concepts and techniques that you'll need to work with ACE. The chapter covered some helpful macros and facilities provided to smooth out the differences among C++ compilers and runtime environments, showed how to take advantage of both narrow and wide characters, and, finally, listed some sources of further information about ACE and the available services to help you make the most of ACE.

Chapter 2. How to Build and Use ACE in Your Programs

Traditional, closed-source libraries and toolkits are often packaged as shared libraries (DLLs), often without source code. Although ACE is available in installable packages for some popular platforms, its open-source nature means that you can not only see the source but also change it to suit your needs and rebuild it. Thus, it is very useful to know how to build it from sources.

You will also, of course, need to build your own applications. This chapter shows how to include ACE in your application's source code and build it.

2.1 A Note about ACE Versions

ACE kits are released periodically as either of two types:

1. Release— two numbers, such as 5.3. These versions are stable and well tested. You should use release versions for product development unless you need to add a new feature to ACE in support of your project.
2. Beta— three numbers, such as 5.3.4. These versions are development snapshots: the "bleeding edge." They may contain bug fixes above the previous release, but they may also contain more bugs or new features whose effects and interactions have not yet been well tested.

The first beta kit following each release—for example, 5.3.1—is traditionally reserved for bug fixes to the release and usually doesn't contain any new features or API changes. This is sometimes referred to as a BFO (bug fix only) version. However, it doesn't go through all the same aggressive testing that a release does.

Riverace Corporation also releases "Fix Kits": fixes made to a release version and labeled with the release version number followed by a letter that increases with each Fix Kit, such as 5.3c. Riverace maintains a separate stream of fixes for each ACE release and generates Fix Kits separately from the main ACE development stream. For this book's purposes, Fix Kits are treated the same as release versions. This book's descriptions and examples are based on ACE 5.3.

2.2 Guide to the ACE Distribution

The ACE distribution is arranged in a directory tree. This tree structure is the same whether you obtain ACE as part of a prebuilt kit or as sources only. The top-level directory in the ACE distribution is named ACE_wrappers, but it may be located in different places, depending on what type of kit you have.

- If you installed a prebuilt kit, ask your system administrator where the kit was installed to.
- If you have a source kit, it is probably a Gzip-compressed tar file. Copy the distribution file, which we'll assume is named ACE-5.3.tar.gz, to an empty directory. If on a UNIX system, use the following commands to unpack the kit:

```
$ gunzip ACE-5.3.tar.gz
$ tar xf ACE-5.3.tar
```

  If on Windows, the WinZip utility is a good tool for unpacking the distribution. It works for either a Gzip'd tar file or a zip file.

In both cases, you'll have a directory named ACE_wrappers that contains the ACE sources and a lot of other useful files and information. Some useful and interesting files in ACE_wrappers are:

- VERSION— Identifies what version of ACE you have.
- PROBLEM-REPORT-FORM— Provides the information you'll need to report when asking for help or advice from Riverace or on the ACE user groups.
- ChangeLog— Provides a detailed list in reverse chronological order of all changes made to ACE. The ChangeLog file gets very long over time, so it's periodically moved to the ChangeLogs directory and a new ChangeLog started. For each new ACE version, Riverace writes release notes that summarize the more noteworthy changes and additions to ACE, but if you're curious about details, they're all in the ChangeLog.
- THANKS— Lists all the people who have contributed to ACE over the years. It is an impressive list, for sure. If you are so inspired to contribute new code or fixes to ACE, your name will be added too.

  The following directories are located below ACE_wrappers:

- ace contains the source code for the ACE toolkit.
- bin contains a number of useful utilities, which we'll describe as their use comes up in this book.
- apps contains a number of ACE-based application programs, such as the JAWS web server and the Gateway message router. These programs are often useful in and of themselves; however, they also contain great examples of how to use ACE in various scenarios.
-

2.3 How to Build ACE

Why would you want to (re)build ACE?

- ACE has been ported to many platforms, but a prebuilt kit may not be available for yours.
- You've fixed a bug or obtained a source patch.
- You've added a new feature.
- You've changed a configuration setting.

Many open-source projects use the GNU Autotools to assist in configuring a build for a particular platform. ACE doesn't use GNU Autotools, because when ACE's build scheme was developed, GNU Autotools wasn't mature enough to handle ACE's requirements on the required platforms. Future versions of ACE will use the GNU Autotools, however.

ACE has its own build scheme that uses GNU Make (version 3.79 or newer is required) and two configuration files.

1. config.h contains platform-specific settings used to select OS features and adapt ACE to the platform's compilation environment.
2. platform_macros.GNU contains commands and command options used to build ACE. This file allows selection of debugging capabilities, to build with threads or not, and so on. This file is not used with Microsoft Visual C++, as all command, feature, and option settings are contained in the Visual C++ project files.

Platform-specific versions of each of these files are included in the ACE source distribution.

The procedure for building ACE follows.

1. If you don't already have it, obtain the ACE source kit. See Section 1.8 for information on where to find ACE kits.
2. The kit comes in a few formats, most commonly a Gzip-compressed tar file. Use gunzip and tar—or WinZip on Windows—to unpack the kit into a new, empty directory.
3. (You can skip this step if using Microsoft Visual C++.) If you haven't already set the ACE_ROOT environment variable, set it to the full pathname of the top-level ACE_wrappers directory: for example, ACE_ROOT=/home/mydir/ace/ACE_wrappers; export ACE_ROOT
4. Create the config.h file in $ACE_ROOT/ace/config.h. The file must include one of the platform-specific configuration files supplied by ACE. For example, the following is for building ACE on Windows:

```
#include "ace/config-win32.h"
```

2.4 How to Include ACE in Your Applications

This book describes many things you can do with ACE and how to do them. Including ACE in your programs has two practical requirements.

1. Include the necessary ACE header files in your sources. For example, to include the basic OS adaptation layer methods, add this to your source:

   ```
   #include "ace/OS.h"
   ```

   You should always specify the ace directory with the file to avoid confusion with choosing a file from somewhere in the include path other than the ACE sources. To be sure that the ACE header files are located correctly, you must include $ACE_ROOT in the compiler's include search path, usually done by using the -I or /I option.

   You should include the necessary ACE header files before including your own headers or system-provided headers. ACE's header files can set preprocessor macros that affect system headers and feature tests, so it is important to include ACE files first. This is especially important for Windows but is good practice to follow on all platforms.

2. Link the ACE library with your application or library. For POSIX platforms, this involves adding -lACE to the compiler link command. If your ACE version was installed from a prebuilt kit, the ACE library was probably installed to a location that the compiler/linker searches by default. If you built ACE yourself, the ACE library is in $ACE_ROOT/ace. You must include this location in the compiler/linker's library search path, usually by using the -L option.

2.5 How to Build Your Applications

The scheme used to build ACE can also be used to build your applications. The advantage to using the supplied scheme is that you take advantage of the built-in knowledge about how to compile and link both libraries and executable programs properly in your environment. One important aspect of that knowledge is having the correct compile and link options to properly include ACE and the necessary vendor-supplied libraries in each step of your build. Even if you don't use the ACE build scheme, you should read about how the compile and link options are set for your platform to be sure that you do compatible things in your application's build scheme.

This is a small example of how easy it is to use the GNU Make-based system. Microsoft Visual C++ users will not need this information and can safely skip to Section 2.5.1.

If you have a program called hello_ace that has one source file named hello_ace.cpp, the Makefile to build it would be:

```
BIN   = hello_ace
BUILD = $(VBIN)
SRC = $(addsuffix .cpp,$(BIN))
LIBS = -lMyOtherLib
LDFLAGS = -L$(PROJ_ROOT)/lib
#---------------------------------------------------
#Include macros and targets
#---------------------------------------------------
include $(ACE_ROOT)/include/makeinclude/wrapper_macros.GNU
include $(ACE_ROOT)/include/makeinclude/macros.GNU
include $(ACE_ROOT)/include/makeinclude/rules.common.GNU
include $(ACE_ROOT)/include/makeinclude/rules.nonested.GNU
include $(ACE_ROOT)/include/makeinclude/rules.bin.GNU
include $(ACE_ROOT)/include/makeinclude/rules.local.GNU
```

That Makefile would take care of compiling the source code and linking it with ACE and would work on each ACE platform that uses the GNU Make-based scheme. The LIBS = -lMyOtherLib line specifies that, when linking the program, -lMyOtherLib will be added to the link command; the specified LDFLAGS value will as well. This allows you to include libraries from another part of your project or from a third-party product. The ACE make scheme will automatically add the options to include the ACE library when the program is linked. Building an executable program from multiple source files would be similar:

```
BIN = hello_ace
FILES = Piece2 Piece3
SRC= $(addsuffix .cpp,$(FILES))
OBJ= $(addsuffix .o,$(FILES))
BUILD   = $(VBIN)
#---------------------------------------------------------
# Include macros and targets
#---------------------------------------------------------
include$(ACE_ROOT)/include/makeinclude/wrapper_macros.GNU
include$(ACE_ROOT)/include/makeinclude/macros.GNU
include$(ACE_ROOT)/include/makeinclude/rules.common.GNU
include$(ACE_ROOT)/include/makeinclude/rules.nonested.GNU
include$(ACE_ROOT)/include/makeinclude/rules.bin.GNU
include$(ACE_ROOT)/include/makeinclude/rules.local.GNU
```

This Makefile would add Piece2.cpp and Piece3.cpp to the hello_ace program, first compiling the new files and then linking all the object files together to form the hello_ace program.

The following example shows how to build a shared library from a set of source files:

2.6 Summary

In this chapter, we learned about ACE kits and version types, how to build ACE, and why you might need to. We also learned how to build both an executable program and a shared library (DLL) that use ACE. Finally, we learned some details about building and using shared libraries on Windows platforms.

Next, we look at one of the most basic developer needs—a flexible way to log runtime information and trace program execution—and how to use the facilities ACE provides to meet those needs.

Chapter 3. Using the ACE Logging Facility

Every program needs to display diagnostics: error messages, debugging output, and so on. Traditionally, we might use a number of printf() calls or cerr statements in our application in order to help trace execution paths or display helpful runtime information. ACE's logging facility provides us with ways to do these things while at the same time giving us great control over how much of the information is printed and where it is directed.

It is important to have a convenient way to create debug statements. In this modern age of graphical source-level debuggers, it might seem strange to pepper your application with the equivalent of a bunch of print statements. However, diagnostic statements are useful both during development and long after an application is considered to be bug free.

- They can record information while the program is running and a debugger isn't available or practical, such as with a server.
- They can record output during testing for regression analysis, as the ACE test suite does.

The ACE mechanisms allow us to enable and disable these statements at compile time. When compiled in, they can also be enabled and disabled at will at runtime. Thus, you don't have to pay for the overhead—in either CPU cycles or disk space—under normal conditions. But if a problem arises, you can easily cause copious amounts of debugging information to be recorded to assist you in finding and fixing it. It is an unfortunate fact that many bugs will never appear until the program is in the hands of the end user.

In this chapter, we cover how to

- Use basic logging and tracing techniques
- Enable and disable display of various logging message severities
- Customize the logging mechanics
- Direct the output messages to various logging sinks
- Capture log messages before they're output
- Use the distributed ACE logging service
- Combine various logging facility features
- Dynamically configure logging sinks and severity levels

3.1 Basic Logging and Tracing

Three macros are commonly used to display diagnostic output from your code: ACE_DEBUG, ACE_ERROR, and ACE_TRACE. The arguments to the first two are the same; their operation is nearly identical, so for our purposes now, we'll treat them the same. They both take a severity indicator as one of the arguments, so you can display any message using either; however, the convention is to use ACE_DEBUG for your own debugging statements and ACE_ERROR for warnings and errors. The use of these macros is the same:

```
ACE_DEBUG ((severity, formatting-args));
ACE_ERROR ((severity, formatting-args));
```

The severity parameter specifies the severity level of your message. The most common levels are LM_DEBUG and LM_ERROR. All the valid severity values are listed in Table 3.1.

The formatting-args parameter is a printf()-like set of format conversion operators and formatting arguments for insertion into the output. The complete set of formatting directives is described in Table 3.2. One might wonder why printf()-like formatting was chosen instead of the more natural—to C++ coders—C++ iostream-style formatting. In some cases, it would have been easier to correctly log certain types of information with type-safe insertion operators. However, an important factor in the logging facility's design is the ability to effectively "no-op" the logging statements at compile time. Note that the ACE_DEBUG and ACE_ERROR invocations require two sets of parentheses. The outer set delimits the single macro argument. This single argument comprises all the arguments, and their enclosing parentheses, needed for a method call. If the preprocessor macro ACE_NDEBUG is defined, the ACE_DEBUG macro will expand to a blank line, ignoring the content of the inner set of parentheses. Achieving this same optimization with insertion operators would have resulted in a rather odd usage:

```
ACE_DEBUG ((debug_info << "Hi Mom" << endl));
```

Table 3.1. ACE_Log_Msg Logging Severity Levels

Severity Level	Meaning
LM_TRACE	Messages indicating function-calling sequence
LM_DEBUG	Debugging information
LM_INFO	Messages that contain information normally of use only when debugging a program
LM_NOTICE	Conditions that are not error conditions but that may require special handling
LM_WARNING	Warning messages
LM_ERROR	Error messages
LM_CRITICAL	Critical conditions, such as hard device errors

3.2 Enabling and Disabling Logging Severities

Consider this slightly modified code:

```
#include "ace/Log_Msg.h"

void foo(void);

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_TRACE (ACE_TEXT ("main"));

  ACE_LOG_MSG->priority_mask (LM_DEBUG | LM_NOTICE,
                              ACE_Log_Msg::PROCESS);
  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IHi Mom\n")));
  foo ();
  ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("%IGoodnight\n")));

  return 0;
}

void foo(void)
{
  ACE_TRACE (ACE_TEXT ("foo"));

  ACE_DEBUG ((LM_NOTICE, ACE_TEXT ("%IHowdy Pardner\n")));
}
```

The following output is produced:

```
(1024) calling main in file `Simple2.cpp' on line 7
      Howdy Pardner
   Goodnight
```

In this example, we changed the logging level at runtime so that only messages logged with LM_DEBUG and LM_NOTICE priority are displayed; all others are ignored. The LM_INFO "Hi Mom" message is not displayed, and there is no ACE_TRACE output.

We've also revealed a little more about how ACE's logging facility works. The ACE_Log_Msg class implements the log message formatting capabilities in ACE. ACE automatically maintains a thread-specific singleton instance of the ACE_Log_Msg class for each spawned thread, as well as the main thread. ACE_LOG_MSG is a shortcut for obtaining the pointer to the thread's singleton instance. All the ACE logging macros use ACE_LOG_MSG to make method calls on the correct ACE_Log_Msg instance. There is seldom a reason to instantiate an ACE_Log_Msg object directly. ACE automatically creates a new instance for each thread spawned, keeping each thread's logging output separate.

Table 3.3. ACE Logging Macros

Macro	Function	Disabled by
ACE_ASSERT(test)	Much like the assert() library call. If the test fails, an assertion message including the file name and line number, along with the test itself, will	ACE_NDEBUG

3.3 Customizing the ACE Logging Macros

In most cases, people will use the standard ACE tracing and logging macros shown in Table 3.3. Sometimes, however, their behavior may need to be customized. Or you might want to create wrapper macros in anticipation of future customization.

Table 3.4. Commonly Used ACE_Log_Msg Methods

Method	Purpose
op_status	The return value of the current function. By convention, –1 indicates an error condition.
errnum	The current errno value.
linenum	The line number on which the message was generated.
file	File name in which the message was generated.
msg	A message to be sent to the log output target.
inc	Increments nesting depth. Returns the previous value.
dec	Decrements the nesting depth. Returns the new value.
trace_depth	The current nesting depth.

`start_tracing`
`stop_tracing`
`tracing_enabled`
	Enable/disable/query the tracing status for the current ACE_Log_Msg instance. The tracing status of a thread's ACE_LOG_MSG singleton determines whether an ACE_Trace object generates log messages.

priority_mask	Get/set the set of severity levels—at instance or process level—for which messages will be logged.
log_priority_enabled	Return non-zero if the requested priority is enabled.
set	Sets the line number, file name, op_status, and several other characteristics all at once.
conditional_set	Sets the line number, file name, op_status, and errnum values for the next log message; however, they take effect only if the next logging message's severity level is enabled.

3.3.1 Wrapping ACE_DEBUG

Perhaps you want to ensure that all your LM_DEBUG messages contain a particular text string so that you can easily

3.4 Redirecting Logging Output

As our previous examples have shown, the default logging sink for ACE's logging facility is the standard error stream. In this section, we discuss output to the standard error stream, as well as two other common and useful targets:

- The system logger (UNIX syslog or NT Event Log)
- A programmer-specified output stream, such as a file

3.4.1 Standard Error Stream

Output to the standard error stream (STDERR) is so common that it is, in fact, the default sink for all ACE logging messages. Our examples so far have taken advantage of this. Sometimes, you may want to direct your output not only to STDERR but also to one of the other targets available to you. In these cases, you will have to explicitly include STDERR in your choices:

```
int ACE_TMAIN (int, ACE_TCHAR *argv[])
{
  // open() requires the name of the application
  // (e.g. -- argv[0]) because the underlying
  // implementation may use it in the log output.
  ACE_LOG_MSG->open (argv[0], ACE_Log_Msg::STDERR);
```

or

```
ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("%IHi Mom\n")));
ACE_LOG_MSG->set_flags (ACE_Log_Msg::STDERR);
foo ();
```

If you choose the second approach, it may be necessary to invoke clr_flags() to disable any other output destinations. Everything after the set_flags() will be directed to STDERR until you invoke clr_flags() to prevent it. The complete signatures of these methods are:

```
// Enable the bits in the logger's options flags.
void set_flags (unsigned long f);

// Disable the bits in the logger's options flags.
void clr_flags (unsigned long f);
```

The set of defined flag values are listed in Table 3.5.

Table 3.5. Valid ACE_Log_Msg Flags Values

Flag	Meaning
STDERR	Write messages to STDERR

3.5 Using Callbacks

To this point, we've been content to give our logging output to ACE_Log_Msg, which formatted the messages and directed them to the configured logging sinks. For most cases, that will be fine. What if, though, we want to do something with that output ourselves? Can we inspect or even modify the logging output before it reaches its final destination? Of course. That's where ACE_Log_Msg_Callback comes in.

Using a callback object is quite easy. Follow these steps:

1. Derive a callback class from ACE_Log_Msg_Callback, and reimplement the following method:

   ```
   virtual void log (ACE_Log_Record &log_record);
   ```

2. Create an object of your new callback type.
3. To register the callback object with an ACE_Log_Msg instance, pass a pointer to your callback object to the ACE_Log_Msg::msg_callback() method.
4. Call ACE_Log_Msg::set_flags() to enable output to your callback object.

Once registered and enabled, your callback object's log() method will be invoked with an ACE_Log_Record object any time ACE_Log_Msg::log() is invoked. As it turns out, that is exactly what happens when an output-producing ACE logging macro is used.

Some important caveats to remember when using the callback approach are documented on the ACE_Log_Msg_Callback reference page. They bear repeating here.

- Callback registration and enabling are specific to each ACE_Log_Msg instance. Therefore, a callback set up in one thread won't be used by any other thread in your application.
- Callback objects are not inherited by the ACE_Log_Msg instances created for any threads you create. So if you're going to be using callback objects with multithreaded applications, you need to take special care that each thread is given an appropriate callback instance. It is possible to use a single object safely: see the description of ACE_Singleton in Section 1.6.3.
- As with the OSTREAM caveat, be sure that you don't delete a callback instance that might still be used by the ACE_Log_Msg instance it's registered with.

A simple callback implementation follows:

```
#include "ace/streams.h"
#include "ace/Log_Msg.h"
#include "ace/Log_Msg_Callback.h"
#include "ace/Log_Record.h"

class Callback : public ACE_Log_Msg_Callback
```

3.6 The Logging Client and Server Daemons

Put simply, the ACE Logging Service is a configurable two-tier replacement for UNIX syslog. Both syslog and the Windows Event Logger are pretty good at what they do and can even be used to capture messages from remote hosts. But if you have a mixed environment, they simply aren't sufficient.

Table 3.7. ACE_Log_Record Attributes

Attribute	Description
type	The log record type from Table 3.1
priority	Synonym for type
priority_name	The log record's priority name
length	The length of the log record, set by the creator of the log record
time_stamp	The timestamp—generally, creation time—of the log record; set by the creator of the log record
pid	ID of the process that created the log record instance
msg_data	The textual message of the log record
msg_data_len	Length of the msg_data attribute

The ACE netsvcs logging framework has a client/server design. On one host in the network, you run the logging server that will accept logging requests from any other host. On that and every host in the network where you want to use the distributed logger, you invoke the logging client. The client acts somewhat like a proxy by accepting logging requests from clients on the local system and forwarding them to the server. This may seem to be a bit of an odd design, but it helps prevent pounding the server with a huge number of client connections, many of which may be transient. By using the proxy approach, the proxy on each host absorbs a little bit of the pounding, and everyone is better off.

To configure our server and client proxy, we will use the ACE Service Configurator framework. The Service Configurator is an advanced topic that is covered in Chapter 19. We will show you just enough here to get things off the ground. Feel free to jump ahead and read a bit more about the Service Configurator now, or wait and read it later.

To start the server, you need to first create a file server.conf with the following content:

[View full width]

```
dynamic Logger Service_Object * ACE:_make_ACE_Logging_Strategy() "-s foobar -f
➥ STDERR|OSTREAM|VERBOSE"

dynamic Server_Logging_Service Service_Object *
netsvcs:_make_ACE_Server_Logging_Acceptor
➥() active "-p 20009"
```

3.7 The LogManager Class

The preceding sections explained how to direct the logging output to several places. We noted that you can change your mind at runtime and direct the logging output somewhere else. Unfortunately, what you need to do when you change your mind isn't always consistent. Let's take a look at a simple class that attempts to hide some of those details:

```
class LogManager
{
public:
  LogManager ();
  ~LogManager ();

  void redirectToDaemon
    (const ACE_TCHAR *prog_name = ACE_TEXT (""));
  void redirectToSyslog
    (const ACE_TCHAR *prog_name = ACE_TEXT (""));
  void redirectToOStream (ACE_OSTREAM_TYPE *output);
  void redirectToFile (const char *filename);
  void redirectToStderr (void);
  ACE_Log_Msg_Callback * redirectToCallback
    (ACE_Log_Msg_Callback *callback);

  // ...
};
```

The idea is pretty simple: An application will use the redirect* methods at any time to select the output destination:

```
void foo (void);

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  LOG_MANAGER->redirectToStderr ();
  ACE_TRACE (ACE_TEXT ("main"));
  LOG_MANAGER->redirectToSyslog ();
  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IHi Mom\n")));
  foo ();
  LOG_MANAGER->redirectToDaemon ();
  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IGoodnight\n")));

  return 0;
}

void foo (void)
{
  ACE_TRACE (ACE_TEXT ("foo"));
  LOG_MANAGER->redirectToFile ("output.test");
  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IHowdy Pardner\n")));
}
```

"But wait," you say. "Where did LOG_MANAGER come from?" This is an example of the ACE_Singleton template, mentioned in Section 1.6.3. That's what we're using behind LOG_MANAGER. ACE_Singleton simply ensures that we create one single instance of the LogManager class at runtime, even if multiple threads all try to create one at the same time. Using a singleton gives you quick access to a single instance of an object anywhere in your application. To declare our singleton, we add the following to our header file:

3.8 Runtime Configuration with the ACE Logging Strategy

Thus far, all our decisions about what to log and where to send the output have been determined at compile time. In many cases, it is unreasonable to require a recompile to change the logging options. We could, of course, provide parameters or a configuration file to our application, but we would have to spend valuable time writing and debugging that code. Fortunately, ACE has already provided us with a convenient solution in the form of the ACE_Logging_Strategy object.

Consider the following file:

```
dynamic Logger Service_Object * ACE:_make_ACE_Logging_Strategy()
"-s log.out -f STDERR|OSTREAM -p INFO"
```

We've seen this kind of thing before when we were talking about the distributed logging service. In this case, we're instructing the ACE Service Configurator to create and configure a logging strategy instance just like the distributed logging server. Again, the Service Configurator is an advanced topic with many exciting features[2] and is covered in Chapter 19.

[2] One of the most exciting is the ability to reconfigure the service object while the application is running. In the context of our logging strategy, this means that you can change the -p value to reconfigure the logging level without stopping and restarting your application!

The following sample application uses the preceding file:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  if (ACE_Service_Config::open (argc,
                                argv,
                                ACE_DEFAULT_LOGGER_KEY,
                                1,
                                0,
                                1) < 0)
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("Service Config open")),
                      1);
    ACE_TRACE (ACE_TEXT ("main"));
    ACE_DEBUG ((LM_NOTICE, ACE_TEXT ("%t%IHowdy Pardner\n")));
    ACE_DEBUG ((LM_INFO, ACE_TEXT ("%t%IGoodnight\n")));

    return 0;
}
```

The key is the call to ACE_Service_Config::open(), which is given our command line parameters. By default it will open a file named svc.conf, but we can specify an alternative by specifying -f someFile. In either case, the file's content would be something like the preceding, which tells the logging service to direct the output to both STDERR and the file log.out.

Be careful that you call ACE_Service_Config::open() as shown rather than with the default parameters. If the final parameter is not 1, the open() method will restore the logging flags to their preopen values. Because the logging service loads its configuration and sets the logging flags from within the service configuration's open(), you will be unpleasantly surprised to find that the logging strategy had no effect on the priority mask once open() completes.

Recall that, by default, all logging severity levels are enabled at a processwide level. If you specify -p INFO in your config file, you will probably be surprised when you get other logging levels also; they were already enabled by

3.9 Summary

Every program needs to have a good logging mechanism. ACE provides you with more than one way to handle such things. Consider your application and how you expect it to grow over time. Your choices range from the simple ACE_DEBUG macros to the highly flexible logging service. You can run "out of the box" or customize things to fit your specific environment. Take the time to try out several approaches before settling on one. With ACE, changing your mind is easy.

Chapter 4. Collecting Runtime Information

Most applications offer ways for users to direct or alter runtime behavior. Two of the most common approaches are

- Accepting command line arguments and options. This approach is often used for information that can reasonably change on each application invocation. For example, the host name to connect to during an FTP (file transfer protocol) or TELNET session is usually different each time the command is run.
- Reading configuration files. Configuration files usually hold site- or user-specific information that doesn't often change or that should be remembered between application invocations. For example, an installation script may store file system locations to read from or record log files to. The configuration information may indicate which TCP or UDP (user datagram protocol) ports a server should listen on or whether to enable various logging severity levels.

Any information that can reasonably change at runtime should be made available to an application at runtime, not built into the application itself. This allows the information to change without having to rebuild and redistribute the application. In this chapter, we look at the following ACE classes that help in this effort:

- ACE_Get_Opt: to access command line arguments and options
- ACE_Configuration: to manipulate configuration information on all platforms using the ACE_Configuration_Heap class and, for the Windows registry, the ACE_Configuration_Win32Registry class

4.1 Command Line Arguments and ACE_Get_Opt

ACE_Get_Opt is ACE's primary class for command line argument processing. This class is an iterator for parsing a counted vector of arguments, such as those passed on a program's command line via argc/argv. POSIX developers will recognize ACE_Get_Opt's functionality because it is a C++ wrapper facade for the standard POSIX getopt() function. Unlike getopt(), however, each instance of ACE_Get_Opt maintains its own state, so it can be used reentrantly. In addition, ACE_Get_Opt is easier to use than getopt(), as the option definition string and argument vector are passed only once to the constructor rather than to each iterator call.

ACE_Get_Opt can parse two kinds of options:

1. Short, single-character options, which begin with a single dash ('-')
2. Long options, which begin with a double dash ('--')

For example, the following code implements command line handling for a program that offers command line option -f, which takes an argument—the name of a configuration file—and an equivalent long option --config:

```
static const ACE_TCHAR options[] = ACE_TEXT (":f:");
ACE_Get_Opt cmd_opts (argc, argv, options);
if (cmd_opts.long_option
    (ACE_TEXT ("config"), 'f', ACE_Get_Opt::ARG_REQUIRED) == -1)
  return -1;
int option;
ACE_TCHAR config_file[MAXPATHLEN];
ACE_OS_String::strcpy (config_file, ACE_TEXT ("HAStatus.conf"));
while ((option = cmd_opts ()) != EOF)
  switch (option) {
  case 'f':
    ACE_OS_String::strncpy (config_file,
                            cmd_opts.opt_arg (),
                            MAXPATHLEN);
    break;
  case ':':
    ACE_ERROR_RETURN
      ((LM_ERROR, ACE_TEXT ("-%c requires an argument\n"),
        cmd_opts.opt_opt ()), -1);
  default:
    ACE_ERROR_RETURN
      ((LM_ERROR, ACE_TEXT ("Parse error.\n")), -1);
  }
```

This example uses the cmd_opts object to extract the command line arguments. This example illustrates what you must do to process a command line.

- Define the valid options. To define short options, build a character string containing all valid option letters. A colon following an option letter means that the option requires an argument. In the preceding example, -f requires an argument. Use a double colon if the argument is optional. To add equivalent long options, use the long_option() method to equate a long option string with one of the short options. Our example equates the --config option with -f.
- Use operator() to iterate through the command line options. It returns the short option character when located and the short option equivalent when a long option is processed. The option's argument is accessed via the

4.2 Accessing Configuration Information

Many applications are installed via installation scripts that store collected information in a file that the application reads at runtime. On modern versions of Microsoft Windows, this information is often stored in the Windows registry; in earlier versions, a file was used. Most other platforms use files as well. The ACE_Configuration class defines the configuration interface for the following two classes available for accessing and manipulating configuration information.

1. ACE_Configuration_Heap, available on all platforms, keeps all information in memory. The memory allocation can be customized to use a persistent backing store, but the most common use is with dynamically allocated heap memory; hence its name.
2. ACE_Configuration_Win32Registry, available only on Windows, implements the ACE_Configuration interface to access and manipulate information in the Windows registry.

In both cases, configuration values are stored in hierarchically related sections. Each section has a name and zero or more settings. Each setting has a name and a typed data value. Even though the configuration information can be both read and modified, resist the temptation to use it as a database, with frequent updates. It's not designed for that.

The following example shows how the Home Automation system uses ACE's configuration facility to configure each subsystem's TCP port number. The configuration uses one section per subsystem, with settings in each section used to configure an aspect of that subsystem. Thus, the configuration for the entire system is managed in a central location. The example uses the config_file command line argument read in the example on page 78. After importing the configuration data, the program looks up the ListenPort value in the HAStatus section to find out where it should listen for status requests:

```
ACE_Configuration_Heap config;
if (config.open () == -1)
  ACE_ERROR_RETURN
    ((LM_ERROR, ACE_TEXT ("%p\n"), ACE_TEXT ("config")), -1);
ACE_Registry_ImpExp config_importer (config);
if (config_importer.import_config (config_file) == -1)
  ACE_ERROR_RETURN
    ((LM_ERROR, ACE_TEXT ("%p\n"), config_file), -1);

ACE_Configuration_Section_Key status_section;
if (config.open_section (config.root_section (),
                         ACE_TEXT ("HAStatus"),
                         0,
                         status_section) == -1)
  ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                     ACE_TEXT ("Can't open HAStatus section")),
                    -1);

u_int status_port;
if (config.get_integer_value (status_section,
                              ACE_TEXT ("ListenPort"),
                              status_port) == -1)
  ACE_ERROR_RETURN
    ((LM_ERROR,
      ACE_TEXT ("HAStatus ListenPort does not exist\n")),
     -1);
this->listen_addr_.set (ACE_static_cast (u_short, status_port));
```

To remain portable across all ACE platforms, this example uses the ACE_Configuration_Heap class to access the configuration data. Whereas the ACE_Configuration_Win32Registry class operates directly on the Windows registry, the contents of each ACE_Configuration_Heap object persist only as long as the object itself. Therefore, the data

4.3 Building Argument Vectors

Section 4.1 showed how to process an argument vector, such as the argc/argv passed to a main program. Sometimes, however, it is necessary to parse options from a single long string containing tokens similar to a command line. For example, a set of options may be read as a string from a configuration file. In this case, it is helpful to convert the string to an argument vector in order to use ACE_Get_Opt. ACE_ARGV is a good class for this use.

Let's say that a program that obtains its options from a string wants to parse the string by using ACE_Get_Opt. The following code converts the cmdline string into an argument vector and instantiates the cmd_opts object to parse it:

```
#include "ace/ARGV.h"
#include "ace/Get_Opt.h"

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  static const ACE_TCHAR options[] = ACE_TEXT (":f:h:");
  static const ACE_TCHAR cmdline[] =
    ACE_TEXT ("-f /home/managed.cfg -h $HOSTNAME");
  ACE_ARGV cmdline_args (cmdline);
  ACE_Get_Opt cmd_opts (cmdline_args.argc (),
                        cmdline_args.argv (),
                        options,
                        0);          // Don't skip any args
```

Note that the ace/ARGV.h header needs to be included to use the ACE_ARGV class. Another useful feature of ACE_ARGV is its ability to substitute environment variable names while building the argument vector. In the example, the value of the HOSTNAME environment variable is substituted where $HOSTNAME appears in the input string. This feature can be disabled by supplying a 0 value to the second argument on the ACE_ARGV constructor; by default, it is 1, resulting in environment variable substitution.

Note that the environment variable reference uses the POSIX-like leading $, even on platforms such as Windows, where environment variable references do not normally use a $ delimiter. This keeps the feature usable on all platforms that support the use of environment variables. One shortcoming in this feature, however, is that it substitutes only when an environment variable name is present by itself in a token. For example, if the cmdline literal in the previous example contained "-f $HOME/managed.cfg", the value of the HOME environment variable would not be substituted, because it is not in a token by itself.

The preceding example also uses the skip_args parameter on the ACE_Get_Opt constructor. Whereas the argument vector passed to the main() program entry point includes the command name in argv[0], our built vector starts in the first element. Supplying a 0 forces ACE_Get_Opt to start parsing at the first token in the argument vector.

4.4 Summary

Collecting runtime information is a basic part of many applications. Developing code to parse command lines and collect configuration information can be very time consuming and platform dependent. This chapter showed ACE's facilities for collecting and processing runtime information in a portable, customizable, and easy-to-use way.

Chapter 5. ACE Containers

Robust container classes are one of the most useful tools one can obtain from a toolkit. Although the standard template library (STL), with its powerful containers and generic programming constructs, has been standardized by the C++ committee, some compilers and platforms continue to lack support for it. On some platforms, container classes remain unavailable.

ACE initially bundled an implementation of the STL with the ACE source distribution. Unfortunately, many compilers on which ACE ran did not support the C++ constructs used by the STL, so it was dropped from the ACE distribution. In its place, the ACE developers created a separate set of containers that are used internally by the library and are also exported for client development use. Although not as elegant as the STL containers, ACE's containers provide high performance and in many cases have a footprint much smaller than that of the standard C++ containers.

That being said, the standard C++ containers are recommended for application development when you are using ACE. However, the ACE containers are very useful and are recommended in any of the following situations.

- The standard C++ containers are not available.
- Standard C++ containers cannot be used, owing to footprint issues.
- You need to use ACE's special-purpose memory allocators (described in Chapter 17), owing to performance or predictability issues.

This chapter briefly reviews container concepts and then discusses the various template-based containers available in ACE, including both sequence-type containers and associative containers. We end the chapter by discussing some of the allocators that are available with ACE and that plug right into the containers.

5.1 Container Concepts

Let's first review a few container concepts that are applicable to C++. General-purpose containers can be designed and built using various design styles. What is available usually depends on the programming language and the design paradigms supported. For example, Java programmers will find object-based containers available, whereas C programmers will find libraries that support typeless (void*-based) containers.

C++ is a language that supports multiple design paradigms and can therefore support a variety of design methods when it comes to containers. In particular, template-based containers, object-based containers, and typeless containers can be built. ACE supports two of these categories of containers: template-based type-safe containers and object-based containers.

5.1.1 Template-Based Containers

Template-based containers use the C++ templates facility, which allows you to create a "type-specific" container at compile time. For example, if you wanted to store information about all the people in a household, you could create a People list that would allow insertion of only People objects into the list.

This is in sharp contrast to the C way of creating reusable lists of typeless pointers—that is, lists of void*—or the general object-oriented way of creating lists of object pointers or any common base type. An object container allows operations on a single base type; in Java, for example, the java.lang.Object type is often used. An object container allows you to insert any subtype into the container. Therefore, you could conceptually have a single list that included both People and Car objects, which is probably not desired and can occur accidentally. Such errors are usually determined at runtime when you use object containers. Typeless containers offer even less error protection, as any type can usually be added to the container. On the other hand, when a template container is instantiated, you explicitly specify what types of objects are allowed in the container.

We briefly discussed templates and explicit template instantiation in Section 1.6.1. If you are unfamiliar with these concepts and have not gone over them in this book, it might be a good idea to read that section.

One important facet of templates that we skipped in Chapter 1 is specialization. C++ allows you to specialize template classes. That is, it allows you to create special versions for certain template parameters of a class template. For example, we can write special code for an optimized Dynamic_Array<void*> class independent of the Dynamic_Array<T> class template. A user who requests Dynamic_Array<void*> will pick up the special optimized definition and will not instantiate a new class using the Dynamic_Array<T> class template. ACE uses this C++ feature to specialize several useful functors for the ACE_Hash and ACE_Equal_To class templates, which you will find in $ACE_ROOT/ace/Functor.h. You will get to see several examples of this as we progress through this chapter.

5.1.2 Object-Based Containers

Object-based containers support insertion and deletion of a class of object types. If you have programmed with Java or Smalltalk, you will recognize these containers as supporting insertion of the generic object type. ACE has a few containers of this type, built for specific uses, such as the ACE_Message_ Queue class. We will not be discussing these in this chapter but instead will defer the discussion until their specific use comes up.

5.1.3 Iterators

Another important concept to keep in mind is the iterator. Iterators can be thought of as a generalization of the pointer concept in C. Iterators point to a particular location in a container and can be moved to the next or previous location. They can also be dereferenced to obtain the value they are pointing to and subsquently can be used to modify the underlying value in the container. The method of supported iteration and dereferencing semantics is dependent on the type of container and iterator. Some iterators allow only forward iteration; others allow bidirectional iteration. Similarily, constant iterators allow only read access to values. On many containers, ACE provides two iterator APIs: one that to a certain degree follows the C++ standard and a second that is an older ACE proprietary API.

5.2 Sequence Containers

A sequence is a container whose elements are arranged sequentially in a linear order. The ordering will not change, owing to iteration within the container. Lists, stacks, queues, arrays, and sets are all examples of sequences represented by ACE classes.

5.2.1 Doubly Linked List

Doubly linked lists maintain both forward and reverse links within the sequence, allowing efficient forward and reverse traversal within the sequence. However, you cannot randomly access elements. Therefore, you will find the iterator concept handy. The following example illustrates these features as they are provided by the doubly linked list in ACE, ACE_DLList.

ACE_DLList is a template-based container, so we need to specify in advance what element type is allowed in our list. For this purpose, we have created a simple type that wraps an int called DataElement:

```
// A simple data element class.
class DataElement
{
  friend class DataElementEx;

public:
  DataElement () { count_++; }

  DataElement (int data) : data_(data) { count_++; }

  DataElement (const DataElement& e)
  {
    data_ = e.getData ();
    count_++;
  }

  DataElement & operator= (const DataElement& e)
  {
    data_ = e.getData ();
    return *this;
  }

  bool operator== (const DataElement& e)
  { return this->data_ == e.data_; }

  ~DataElement () { count_--; }

  int getData (void) const { return data_; }

  void setData (int val) { data_ = val; }

  static int numOfActiveObjects (void) { return count_; }

 private:
  int data_;
  static int count_;
};
```

One nice feature of the DataElement class is that it remembers how many instances of it currently exist. We will use this feature to illustrate the lifetime of these elements as they are put inside and then taken out of various container types.

To make things easy, let's start by creating a convenient type definition to represent our doubly linked list of

5.3 Associative Containers

Associative containers support efficient retrieval of elements, based on keys instead of positions within the container. Examples of associative containers are maps and binary trees. Associative containers support insertion and retrieval based on keys and do not provide a mechanism to insert an element at a particular position within the container.

5.3.1 Map Manager

ACE supports a simple map type via the ACE_Map_Manager class template. This class maps a key type to a value type. Therefore, the template takes the key and value types as parameters, and insertions require two parameters of these types: a key and the value that is to be associated with that key. Later retrievals require the key and return the value that was associated with the key.

The ACE_Map_Manager is implemented as a dynamic array of entries. Each entry constitutes a key/value pair. Once the dynamic array is full, new memory is allocated, and the size of the array is increased. When an element is removed from the map, the corresponding entry in the dynamic array is marked empty and added to a free list. All new insertions are done using the free list. If the free list happens to be empty, meaning that there is no space for the new entry, a new allocation takes place, and all elements are copied into the new array. The point at which copying takes place is controlled with the ACE_HAS_LAZY_MAP_MANAGER configuration setting. If this flag is set, the movement of free elements in the dynamic array to the free list is deferred until the free list is empty. This allows deletion of elements through an iterator. That is, elements can be deleted during iteration in lazy map managers.

So what does all this mean to you? Insertions in ACE_Map_Manager have a best-case time complexity of O(1); however, in the worst case, it can be O(n). Retrievals are always a linear O(n) operation; for faster operations with an average-case retrieval complexity of O(1), the ACE_Hash_Map_Manager class template is provided.

ACE_Map_Manager requires that the key, or external, element be comparable. Therefore, the equality operator must be defined on the keys that are being inserted into the map. This requirement can be relaxed by using template specialization.

The following example illustrates the fundamental operations on a map with key type KeyType (external type) and value type DataElement. We define KeyType::operator==() per the requirements of ACE_Map_Manager:

```
// Forward declaration.
class KeyType;
bool operator == (const KeyType&, const KeyType&);

class KeyType
{
public:
  friend bool operator == (const KeyType&, const KeyType&);
  KeyType () {}
  KeyType (int i) : val_(i) {}
  KeyType (const KeyType& kt) { this->val_ = kt.val_; };
  operator int() { return val_; };

private:
  int val_;
};

bool operator == (const KeyType& a, const KeyType& b)
{
  return (a.val_ == b.val_);
}
```

We start in the MapExample::run() method by creating 100 bindings of new records in a map. Each bind() call will cause a new KeyType object and DataElement to be created on the stack and then passed by reference into the

5.4 Allocators

As in most container libraries, ACE allows you to specify an allocator class that encapsulates the memory allocation routines the container will use to manage memory. This allows fine-grained control over how you want to manage this memory.

You can either build your own custom allocator or use one of the allocators provided by ACE. All allocators must support the ACE_Allocator interface and are usually provided to the container during construction or during the open() call on the container. In most cases, it is advisable to use the open() call instead of the constructors, as open() returns error codes that can be used in the absence of exceptions.

If you are familiar with allocators in the standard C++ library, you will notice two significant differences in the way allocators work in ACE.

1. Allocators are passed in during object instantiation, not as a type when you instantiate the template. A reference to the provided allocator object is then kept within in the container. The container uses this reference to get to the allocator object. This causes problems if you want allocation to occur in shared memory.
2. The ACE allocators operate on raw untyped memory in the same way that C's malloc() does. They are not type aware. This is in sharp contrast to the standard C++ library's allocators, which are instantiated with supplied types and are type aware. An exception to this rule is ACE_Cached_Allocator, which is strongly typed.

Table 5.1. Allocators Available in ACE

Allocator	Description
ACE_New_Allocator	Allocates memory by using the new operator directly from the heap.
ACE_Static_Allocator	Preallocates a fixed-size pool and then allocates memory from this pool in an optimized fashion. Memory is never deallocated.
ACE_Cached_Allocator	A fixed-size strongly typed allocator that allocates a well-defined number of strongly typed fixed-sized blocks of memory. These blocks are returned on allocation and returned to a free list on deallocation.
ACE_Dynamic_Cached_Allocator	A cached-allocator version that allows you to specify the size and number of the cached blocks at runtime instead of at compile time. Unlike the ACE_Cached_Allocator, these blocks are not strongly typed.

5.4.1 ACE_Allocator

The ACE_Allocator interface is the one that all ACE containers require an allocator to support. This is a true C++ interface, with all methods being pure virtual functions. ACE provides several allocator implementations, which are described in Table 5.1.

5.5 Summary

ACE provides a rich set of efficient, platform-independent containers that you can use in your own applications. Most of these containers are used within ACE to build out further features, so if you are going to be reading through the source, an understanding of these types is neccesary.

Although the recommended application-level containers are the standard C++ library containers, which can coexist with the ACE framework, the ACE containers come in handy not only when a complete standard C++ library is not available but also when you need to fine-tune such features as memory allocation and synchronization. Many of the ACE containers provide standardlike features, making it easier for you to switch between container types easily.

Part II: Interprocess Communication

Chapter 6. Basic TCP/IP Socket Use

Chapter 7. Handling Events and Multiple I/O Streams

Chapter 8. Asynchronous I/O and the ACE Proactor Framework

Chapter 9. Other IPC Types

Chapter 6. Basic TCP/IP Socket Use

This chapter introduces you to basic TCP/IP programming using the ACE toolkit. We begin by creating simple clients and then move on to explore simple servers. After reading this chapter, you will be able to create simple yet robust client/server applications.

The ACE toolkit has a rich set of wrapper facades encapsulating many forms of interprocess communication (IPC). Where possible, those wrappers present a common API, allowing you to interchange one for another without restructuring your entire application. This is an application of the Strategy pattern [3], which allows you to change your "strategy" without making large changes to your implementation. To facilitate changing one set of IPC wrappers for another, ACE's IPC wrappers are related in sets:

- Connector: Actively establishes a connection
- Acceptor: Passively establishes a connection
- Stream: Transfers data
- Address: Defines the means for addressing endpoints

For TCP/IP programming, we use ACE's Sockets-wrapping family of classes:

- ACE_SOCK_Connector
- ACE_SOCK_Acceptor
- ACE_SOCK_Stream
- ACE_INET_Addr

Each class abstracts a bit of the low-level mess of traditional socket programming. All together, the classes create an easy-to-use type-safe mechanism for creating distributed applications. We won't show you everything they can do, but what we do show covers about 80 percent of the things you'll normally need to do.

The basic handling of TCP/IP sockets is foundational to most networked applications, so it's important that you understand the material in this chapter. Be aware that most of the time, you can—and probably should—use higher-level framework classes to simplify your application. We'll look more at these higher-level classes in Section 7.6.

6.1 A Simple Client

In BSD (Berkeley Software Distribution) Sockets programming, you have probably used a number of low-level operating system calls, such as socket(), connect(), and so forth. Programming directly to the Sockets API is troublesome because of such accidental complexities [6] as

- Error-prone APIs. For example, the Sockets API uses weakly typed integer or pointer types for socket handles, and there's no compile-time validation that a handle is being used correctly. For instance, the compiler can't detect that a passively listening handle is being passed to the send() or recv() function.
- Overly complex APIs. The Sockets API supports many communication families and modes of communication. Again, the compiler can offer no help in diagnosing improper use.
- Nonportable and nonuniform APIs. Despite its near ubiquity, the Sockets API is not completely portable. Furthermore, on many platforms, it is possible to mix Sockets-defined functions with OS system calls, such as read() and write(), but this is not portable to all platforms.

With ACE, you can take an object-oriented approach that is easier to use, more consistent, and portable.

Borrowing a page from Stevens's venerable UNIX Network Programming [10], we start by creating a simple client with a few lines of code. Our first task is to fill out a sockaddr_in structure. For purposes of our example, we'll connect to the Home Automation Status Server on our local computer:

```
struct sockaddr_in srvr;

memset (&srvr, 0, sizeof(srvr));
srvr.sin_family      = AF_INET;
srvr.sin_addr.s_addr = inet_addr ("127.0.0.1");
srvr.sin_port        = htons (50000);
```

Next, we use the socket() function to get a file descriptor on which we will communicate and the connect() function to connect that file descriptor to the server process:

```
fd = socket (AF_INET, SOCK_STREAM, 0);

assert (fd >= 0);

assert(
  connect (fd,
           (struct sockaddr *)&srvr,
           sizeof(srvr)) == 0);
```

Now, we can send a query to the server and read the response:

```
write (fd, "uptime\n", 7);
bc = read (fd, buf, sizeof(buf));

write (1, buf, bc);

close (fd);
```

6.2 Adding Robustness to a Client

Let's consider a new client that will query our Home Automation Server for some basic status information and forward that to a logging service. Our first task is, of course, to figure out how to address these services. This time, we'll introduce the default constructor and one of the set() methods of ACE_INET_Addr:

```
ACE_INET_Addr addr;
...
addr.set ("HAStatus", ACE_LOCALHOST);
...
addr.set ("HALog", ACE_LOCALHOST);
```

The set() method is as flexible as the constructors. In fact, the various constructors simply invoke one of the appropriate set() method signatures. You'll find this frequently in ACE when a constructor appears to do something nontrivial. By creating only one address object and reusing it, we can save a few bytes of space. That probably isn't important to most applications, but if you find yourself working on an embedded project where memory is scarce, you may be grateful for it. The return value from set() is more widely used. If set() returns –1, it failed, and ACE_OS::last_error() should be used to check the error code. ACE_OS::last_error() simply returns errno on UNIX and UNIX-like systems. For Windows, however, it uses the GetLastError() function. To increase portability of your application, you should get in the habit of using ACE_OS::last_error().

Now let's turn our attention to ACE_SOCK_Connector. The first thing we should probably worry about is checking the result of the connect() attempt. As with most ACE method calls, connect() returns 0 for success and –1 to indicate a failure. Even if your application will exit when a connection fails, it should at least provide some sort of warning to the user before doing so. In some cases, you may even choose to pause and attempt the connection later. For instance, if connect() returns –1 and errno has the value ECONNREFUSED, it simply means that the server wasn't available to answer your connect request. We're all familiar with heavily loaded web servers. Sometimes, waiting a few seconds before reattempting the connection will allow the connection to succeed.

If you look at the documentation for ACE_SOCK_Connector, you will find quite a few constructors available for your use. In fact, you can use the constructor and avoid the connect() method call altogether. That can be pretty useful, and you'll probably impress your friends, but be absolutely certain that you check for errors after constructing the connector, or you will have one nasty bug to track down:

```
ACE_SOCK_Stream status;
ACE_OS::last_error(0);
ACE_SOCK_Connector statusConnector (status, addr);
if (ACE_OS::last_error())
  ACE_ERROR_RETURN ((LM_ERROR,
                     ACE_TEXT ("%p\n"),
                     ACE_TEXT ("status")), 100);
```

In this example, we explicitly set the last error value to 0 before invoking the ACE_SOCK_Connector constructor. System functions do not generally reset errno to 0 on successful calls, so a previous error value may be noticed here as a false failure.

Don't fret if you don't want to use the active constructors but do like the functionality they provide. There are just as many connect() methods as there are constructors to let you do whatever you need. For instance, if you think that the server may be slow to respond, you may want to time out your connection attempt and either retry or exit:

```
ACE_SOCK_Connector logConnector;
ACE_Time_Value timeout (10);
```

6.3 Building a Server

Creating a server is generally considered to be more difficult than building a client. When you consider all the many things a server must do, that's probably true. However, when you consider only the networking bits, you'll find that the two efforts are practically equal. Much of the difficulty in creating a server centers on such issues as concurrency and resource handling. Those things are beyond the scope of this chapter, but we'll come back to them in Part III.

To create a basic server, you first have to create an ACE_INET_Addr that defines the port on which you want to listen for connections. You then use an ACE_SOCK_Acceptor object to open a listener on that port:

```
ACE_INET_Addr port_to_listen ("HAStatus");
ACE_SOCK_Acceptor acceptor;

if (acceptor.open (port_to_listen, 1) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     ACE_TEXT ("%p\n"),
                     ACE_TEXT ("acceptor.open")),
                    100);
```

The acceptor takes care of the underlying details, such as bind() and accept(). To make error handling a bit easier, we've chosen to go with the default constructor and open() method in our example. If you want, however, you can use the active constructors that take the same parameters as open(). The basic open() method looks like this:

```
int open (const ACE_Addr &local_sap,
          int reuse_addr = 0,
          int protocol_family = PF_UNSPEC,
          int backlog = ACE_DEFAULT_BACKLOG,
           int protocol = 0);
```

This method creates a basic BSD-style socket. The most common usage will be as shown in the preceding example, where we provide an address at which to listen and the reuse_addr flag. The reuse_addr flag is generally encouraged so that your server can accept connections on the desired port even if that port was used for a recent connection. If your server is not likely to service new connection requests rapidly, you may also want to adjust the backlog parameter.

Once you have an address defined and have opened the acceptor to listen for new connections, you want to wait for those connection requests to arrive. This is done with the accept() method, which closely mirrors the accept() function:

```
if (acceptor.accept (peer) == -1)
  ACE_ERROR_RETURN ((LM_ERROR,
                     ACE_TEXT ("(%P|%t) Failed to accept ")
                     ACE_TEXT ("client connection\n")),
                    100);
```

This use will block until a connection attempt is made. To limit the wait time, supply a timeout:

```
if (acceptor.accept (peer, &peer_addr, &timeout, 0) == -1)
  {
    if (ACE_OS::last_error () == EINTR)
      ACE_DEBUG ((LM_DEBUG,
```

6.4 Summary

The ACE TCP/IP socket wrappers provide you with a powerful yet easy-to-use set of tools for creating client/server applications. Using what you've learned in this chapter, you will be able to convert nearly all your traditionally coded, error prone, nonportable networked applications to true C++ object-oriented, portable implementations.

By using the ACE objects, you can create more maintainable and portable applications. Because you're working at a higher level of abstraction, you no longer have to deal with the mundane details of network programming, such as remembering to zero out those sockaddr_in structures and when and what to cast them to. Your application becomes more type safe, which allows you to avoid more errors. Furthermore, when there are errors, they're much more likely to be caught at compile time than at runtime.

Chapter 7. Handling Events and Multiple I/O Streams

Many applications, such as the server example in Chapter 6, can benefit greatly from a simple way to handle multiple events easily and efficiently. Event handling often takes the form of an event loop that continually waits for events to occur, decides what actions need to be taken, and dispatches control to other functions or methods appropriate to handle the event(s). In many networked application projects, the event-handling code is often the first piece of the system to be developed, and it's often developed over and over for each new project, greatly adding to the time and cost for many projects.

The ACE Reactor framework was designed to implement a flexible event-handling mechanism in such a way that applications need never write the central, platform-dependent code for their event-handling needs. Using the Reactor framework, applications need do only three things to implement their event handling.

1. Derive one or more classes from ACE_Event_Handler and add application-specific event-handling behavior to virtual callback methods.
2. Register the application's event-handling objects with the ACE_Reactor class and associate each with the event(s) of interest.
3. Run the ACE_Reactor event loop.

After we see how easy it is to handle events, we'll look at ways ACE's Acceptor-Connector framework simplifies and enables implementation of services. The examples here are much easier, even, than the ones we saw in Chapter 6
.

7.1 Overview of the Reactor Framework

Many traditional applications handle multiple I/O sources, such as network connections, by creating new processes—a process-per-connection model—or new threads—a thread-per-connection model. This is particularly popular in servers needing to handle multiple simultaneous network connections. Although these models work well in many circumstances, the overhead of process or thread creation and maintenance can be unacceptable in others. Moreover, the added code complexity for thread or process management and control can be much more trouble than it's worth in many applications.

The approach we discuss in this chapter is called the reactive model, which is based on the use of an event demultiplexer, such as the select(), poll(), or WaitForMultipleObjects() system functions. These excellent alternatives allow us to handle many events with only one process or thread. Writing portable applications that use these can be quite challenging, however, and that's where the ACE Reactor framework helps us out. It insulates us from the myriad details we would otherwise have to know to write a portable application capable of responding to I/O events, timers, signals, and Windows waitable handles.

The classes we visit in this chapter are

- ACE_Reactor
- ACE_Event_Handler
- ACE_Time_Value
- ACE_Sig_Set
- ACE_Acceptor
- ACE_Connector
- ACE_Svc_Handler

7.2 Handling Multiple I/O Sources

One of the most common uses of the Reactor framework is to handle I/O from various sources. Any program—client, server, peer to peer, or anything that needs to perform I/O and has other things to do at the same time—is a good candidate for using the Reactor. An excellent example is simple server in Chapter 6, on page 138, which could handle only one request on one connection. We'll restructure that server to illustrate how simple it is to take advantage of the Reactor framework's power and how little code we need to write to do so.

A simple server scenario usually requires two event handler classes: one to process incoming connection requests and one to process a client connection. When designed this way, your application will have N + 1 event handlers registered with the Reactor at any one time, where N is the number of currently connected clients. This approach allows your application to easily and efficiently handle many connected clients while consuming minimal system resources.

7.2.1 Accepting Connections

The first thing a server must be able to do is accept a connection request from a potential client. In Section 6.3, we used an ACE_SOCK_Acceptor instance to accept a connection. We'll be using that again here, but this time it will be wrapped in an event handler. By doing this, we can accept any number of connections and simultaneously process client requests on all open client connections.

First, we'll see the declaration of our connection-accepting event handler:

```
#include "ace/Auto_Ptr.h"
#include "ace/Log_Msg.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Reactor.h"

class ClientAcceptor : public ACE_Event_Handler
{
public:
  virtual ~ClientAcceptor ();

  int open (const ACE_INET_Addr &listen_addr);

  // Get this handler's I/O handle.
  virtual ACE_HANDLE get_handle (void) const
    { return this->acceptor_.get_handle (); }

  // Called when a connection is ready to accept.
  virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);

  // Called when this handler is removed from the ACE_Reactor.
  virtual int handle_close (ACE_HANDLE handle,
                            ACE_Reactor_Mask close_mask);

protected:
  ACE_SOCK_Acceptor acceptor_;
};
```

Each class that will handle Reactor events of any type must be derived from ACE_Event_Handler. Although we could come up with a scheme in which one class controlled the connection acceptance and all the client connections, we've chosen to create separate classes for accepting and servicing connections.

- It's a better encapsulation of data and behavior. This class accepts connections from clients, and that's all it does.

7.3 Signals

Responding to signals on POSIX systems traditionally involves providing the signal() system function with the numeric value of the signal you want to catch and a pointer to a function that will be invoked when the signal is received. The newer POSIX set of signal-handling functions (sigaction() and friends) are somewhat more flexible than the tried-and-true signal() function, but getting everything right can be a bit tricky. If you're trying to write something portable among various versions of UNIX, you then have to account for subtle and sometimes surprising differences.

As always, ACE provides us with a nice, clean API portable across dozens of operating systems. Handling signals is as simple as defining a class derived from ACE_Event_Handler with your code in it the new handler class's handle_signal() method and then registering an instance of your object with one of the two appropriate register_handler() methods.

7.3.1 Catching One Signal

Suppose that we need a way to shut down our reactor-based server from Section 7.2. Recall that the main program simply runs the event loop by calling ACE_Reactor::run_reactor_event_loop() but that there's no way to tell it to stop. Let's implement a way to catch the SIGINT signal and stop the event loop:

```
class LoopStopper : public ACE_Event_Handler
{
public:
  LoopStopper (int signum = SIGINT);

  // Called when object is signaled by OS.
  virtual int handle_signal (int signum,
                             siginfo_t * = 0,
                             ucontext_t * = 0);
};

LoopStopper::LoopStopper (int signum)
{
  ACE_Reactor::instance ()->register_handler (signum, this);
}

int
LoopStopper::handle_signal (int, siginfo_t *, ucontext_t *)
{
  ACE_Reactor::instance ()->end_reactor_event_loop ();
  return 0;
}
```

The LoopStopper class registers the single specified signal with the reactor singleton. When the signal is caught, the reactor calls the handle_signal() callback method, which simply calls the end_reactor_event_loop() method on the reactor singleton. On return to the event loop, the event loop will end. If we instantiate one of these objects in the server's main program, we will quickly and easily add the ability to shut the server down cleanly by sending it a SIGINT signal.

7.3.2 Catching Multiple Signals with One Event Handler

We could register many signals using the same technique as in LoopStopper. However, calling register_handler() once for each signal can start getting ugly pretty quickly. Another register_handler() method takes an entire set of signals instead of only one. We will explore that as we extend our server to be able to turn ACE's logging message output on and off with signals:

7.4 Notifications

Let's go back and examine the rest of our LogSwitcher class, which turns logging on and off using signals. The handle_signal() method follows:

```
int
LogSwitcher::handle_signal (int signum, siginfo_t *, ucontext_t *)
{
  if (signum == this->on_sig_ || signum == this->off_sig_)
    {
      this->on_off_ = signum == this->on_sig_;
      ACE_Reactor::instance ()->notify (this);
    }
  return 0;
}
```

As you can see, handle_signal() did not do anything related to ACE_Log_Msg, so how do we turn logging output on and off? We didn't—yet. You need to keep in mind a subtle issue when handling signals. When in handle_signal(), your code is not in the normal execution flow but rather at signal state, and on most platforms, you're restricted from doing many things in signal state. Check your OS documentation for details to be sure whether you need to do work at this point, but the safest thing to do is usually set some state information and find a way to transfer control back to normal execution context. One particularly useful way to do this is by using the reactor's notification mechanism.

The reactor notification mechanism gives you a way to queue a callback event of a type you choose to a handler you choose. The queueing of this event will wake the reactor up if it's currently waiting on its internal event demultiplexer, such as select() or WaitForMultipleObjects(). In fact, the reactor code uses this mechanism internally to make a waiting reactor thread wake up and recheck its records of what events and handles to wait for events on, which is why you can register and unregister handlers from multiple threads while the reactor event loop is running.

Our handle_signal() method checks whether it should turn logging on or off and then calls notify() to queue a callback to its own handle_exception() method. (EXCEPT_MASK is the default event type; because we wanted that one, we left the argument off the notify() call.) After returning to the reactor, the reactor will complete its handling of signals and return to waiting for events. Shortly thereafter, it will notice the queued notification event and dispatch control to handle_exception():

```
int
LogSwitcher::handle_exception (ACE_HANDLE)
{
  if (this->on_off_)
    ACE_LOG_MSG->clr_flags (ACE_Log_Msg::SILENT);
  else
    ACE_LOG_MSG->set_flags (ACE_Log_Msg::SILENT);
  return 0;
}
```

The handle_exception() method examines the saved state to see what action it should take and sets or clears the SILENT flag.

7.5 Timers

Sometimes, your application needs to perform a periodic task. A traditional approach would likely create a dedicated thread or process with appropriate sleep() calls. A process-based implementation might define a timerTask() function as follows:

```
pid_t timerTask (int initialDelay,
                 int interval,
                 timerTask_t task)
{
  if (initialDelay < 1 && interval < 1)
    return -1;

  pid_t pid = fork ();

  if (pid < 0)
    return -1;

  if (pid > 0)
    return pid;

  if (initialDelay > 0)
    sleep (initialDelay);

  if (interval < 1)
    return 0;

  while (1)
    {
      (*task) ();
      sleep (interval);
    }

  return 0;
}
```

The timerTask() function will create a child process and return its ID to the calling function for later cleanup. Within the child process, we simply invoke the task function pointer at the specified interval. An optional initial delay is available. One-shot operation can be achieved by providing a nonzero initial delay and zero or negative interval:

```
int main (int, char *[])
{
  pid_t timerId = timerTask (3, 5, foo);
  programMainLoop ();
  kill (timerId, SIGINT);
  return 0;
}
```

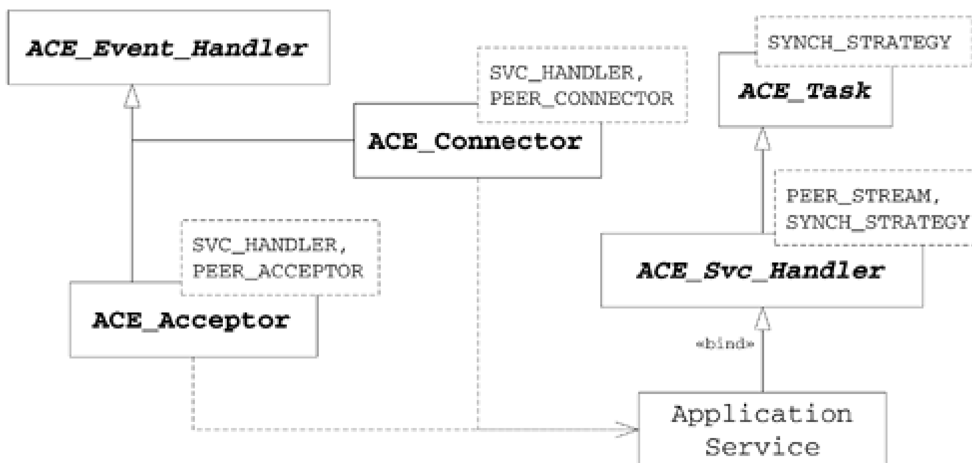
Our main() function creates the timer-handling process with an initial delay of 3 seconds and an interval of 5 seconds. Thus, the foo() function will be invoked 3 seconds after timerTask() is called and every 5 seconds thereafter. Then main() does everything else your application requires. When done, it cleans up the timer by simply killing the process:

```
void foo ()
{
  time_t now = time (0);
  cerr << "The time is " << ctime (&now) << endl;
```

7.6 Using the Acceptor-Connector Framework

In Section 7.2, we hinted that making two classes that know something of each other's internal needs was asking for some trouble down the road. We also said that ACE had already addressed this issue, and we discuss that here. The Acceptor-Connector framework implements the common need of making a connection—not necessarily TCP/IP, but that's obviously a common case—and creating a service handler to run the service on the new connection. The framework's main classes were also designed so that most everything can be changed by specifying different template arguments. The classes involved in the framework can seem a bit tangled and difficult to grasp at first. You can refer to Figure 7.1 to see how they relate. In this section, we rework the example from Section 7.2 to use the Acceptor-Connector framework.

Figure 7.1. Acceptor-Connector framework classes



7.6.1 Using ACE_Acceptor and ACE_Svc_Handler

The actions we had to take in our previous example are very common. In terms of creating new connections, we had to open the ACE_SOCK_Acceptor, register it with the reactor, and handle new connections by creating service-handling objects and initializing them. As we said in Section 1.1, a framework implements a semicomplete application by codifying the canonical way of performing a number of related tasks. ACE_Acceptor plays a role in the Acceptor-Connector framework by codifying the usual way the preceding tasks are done. However, the framework stays flexible by allowing the particular acceptor-type and service handler-type classes to be specified using template arguments. Let's take a look at how our new acceptor code looks:

```
#include "ace/Log_Msg.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Reactor.h"
#include "ace/Acceptor.h"

typedef ACE_Acceptor<ClientService, ACE_SOCK_ACCEPTOR>
  ClientAcceptor;
```

That's it. All the behavior we had to write code for previously is part of the framework, and we need write it no more. That was easy. We simply created a type definition specifying that ACE_SOCK_Acceptor is the class that accepts new service connections and that each new service is handled by a new instance of ClientService. Note that we used the ACE_SOCK_ACCEPTOR macro because ACE_Acceptor also needs to know the addressing trait of the acceptor class; the macro makes it compile correctly with compilers that handle this kind of trait correctly and those that don't.

The main program changed a bit to use the new type:

7.7 Reactor Implementations

Most applications will use the default reactor instance provided by ACE_Reactor::instance(). In some applications, however, you may find it necessary to specify a preferred implementation. In fact, nine reactor implementations are available at the time of this writing. The API declared by ACE_Reactor has proved flexible enough to allow for easy integration with third-party toolkits, such as the Xt framework of the X Window System and the Windows COM/DCOM framework.

You may even create your own extension of one of the existing implementations. ACE's internal design simplifies this extensibility by using the Bridge pattern [3]. The Bridge pattern uses two separate classes: one is the programming interface, and the second is the implementation that the first forwards operations to. When using the defaults in ACE, you need never know how this works. However, if you wish to change the implementation, a new implementation class must be specified for the Bridge. For example, to use the thread-pool reactor implementation, your application would first create the ACE_TP_Reactor implementation instance and then a new ACE_Reactor object that specifies the implementation:

```
ACE_TP_Reactor *tp_reactor = new ACE_TP_Reactor;

ACE_Reactor *my_reactor = new ACE_Reactor (tp_reactor, 1);
```

The second argument to the ACE_Reactor constructor directs ACE_Reactor to also delete the tp_reactor object when my_reactor is destroyed.

To use the specialized reactor object as your program's singleton, use:

```
ACE_Reactor::instance (my_reactor, 1);
```

The second argument directs ACE to delete the my_reactor instance at program termination time. This is a good idea to prevent memory leaks and to allow for a clean shutdown.

7.7.1 ACE_Select_Reactor

ACE_Select_Reactor is the default reactor implementation used on every platform except Windows. The select() system function is ultimately used on these systems to wait for activity. The ACE_Select_Reactor is designed to be used by one thread at a time. That thread is referred to as the owner. The thread that creates the ACE_Select_Reactor—in most cases, the initial program thread—is the initial owner. The owner thread is set by using the owner() method. Only the owner thread can run the reactor's event loop. Most times when the call to run the event loop returns –1 immediately, it's because it was called by a thread that doesn't own the reactor.

7.7.2 ACE_WFMO_Reactor and ACE_Msg_WFMO_Reactor

ACE_WFMO_Reactor is the default reactor implementation on Windows. Instead of using the select() demultiplexer, the implementation uses WaitForMultipleObjects(). There are some tradeoffs to remember when using ACE_WFMO_Reactor. These tradeoffs favor use of ACE_WFMO_Reactor in the vast majority of use cases; however, it's prudent to be aware of them and evaluate the tradeoffs in the context of your projects:

- Handle limit. The ACE_WFMO_Reactor can register only 62 handles. The underlying WaitForMultipleObjects() function imposes a limit of 64, and ACE uses 2 of them internally.
-

7.8 Summary

The Reactor framework is a very powerful and flexible system for handling events from many sources, seemingly simultaneously, without incurring the overhead of multiple threads. At the same time, you can use the reactor in a multithreaded application and have the best of both worlds. A single reactor instance can easily handle activity of timers, signals, and I/O events. In addition to its use with sockets, as shown in this chapter, most reactor implementations can handle I/O from any selectable handle: pipes, UNIX-domain sockets, UDP sockets, serial and parallel I/O devices, and so forth. With a little ingenuity, your reactor-based application can turn on the foyer light when someone pulls into your driveway or mute the television when the phone rings!

Chapter 8. Asynchronous I/O and the ACE Proactor Framework

Applications that must perform I/O on multiple endpoints—whether network sockets, pipes, or files—historically use one of two I/O models:

1. Reactive. An application based on the reactive model registers event handler objects that are notified when it's possible to perform one or more desired I/O operations, such as receiving data on a socket, with a high likelihood of immediate, successful completion. The ACE Reactor framework, described in Chapter 7, supports the reactive model.
2. Multithreaded. An application spawns multiple threads that each perform synchronous, often blocking, I/O operations. This model doesn't scale very well for applications with large numbers of open endpoints.

Reactive I/O is the most common model, especially for networked applications. It was popularized by wide use of the select() function to demultiplex I/O across file descriptors in the BSD Sockets API. Asynchronous I/O, also known as proactive I/O, is often a more scalable way to perform I/O on many endpoints. It is asynchronous because the I/O request and its completion are separate, distinct events that occur at different times. Proactive I/O allows an application to initiate one or more I/O requests on multiple I/O endpoints in parallel without blocking for their completion. As each operation completes, the OS notifies a completion handler that then processes the results.

Asynchronous I/O has been in use for many years on such OS platforms as OpenVMS and on IBM mainframes. It's also been available for a number of years on Windows and more recently on some POSIX platforms. This chapter explains more about asynchronous I/O and the proactive model and then explains how to use the ACE Proactor framework to your best advantage.

8.1 Why Use Asynchronous I/O?

Reactive I/O operations are often performed in a single thread, driven by the reactor's event-dispatching loop. Each thread, however, can execute only one I/O operation at a time. This sequential nature can be a bottleneck, as applications that transfer large amounts of data on multiple endpoints can't use the parallelism available from the OS and/or multiple CPUs or network interfaces.

Multithreaded I/O alleviates the main bottleneck of single-threaded reactive I/O by taking advantage of concurrency strategies, such as the thread-pool model, available using the ACE_TP_Reactor and ACE_WFMO_Reactor reactor implementations, or the thread-per-connection model, which often uses synchronous, blocking I/O. Multithreading can help parallelize an application's I/O operations, which may improve performance. This technique can also be very intuitive, especially when using serial, blocking function calls. However, it is not always the best choice, for the following reasons:

- Threading policy tightly coupled to concurrency policy. A separate thread is required for each desired concurrent operation or request. It would be much better to define threading policy by available resources, possibly factoring in the number of available CPUs, using a thread pool.
- Increased synchronization complexity. If request processing requires shared access to data, all threads must serialize data access. This involves another level of analysis and design, as well as further complexity.
- Synchronization performance penalty. Overhead related to context switching and scheduling, as well as interlocking/competing threads, can degrade performance significantly.

Therefore, using multiple threads is not always a good choice if done solely to increase I/O parallelism.

The proactive I/O model entails two distinct steps.

1. Initiate an I/O operation.
2. Handle the completion of the operation at a later time.

These two steps are essentially the inverse of those in the reactive I/O model.

1. Use an event demultiplexer to determine when an I/O operation is possible and likely to complete immediately.
2. Perform the operation.

Unlike conventional reactive or synchronous I/O models, the proactive model allows a single application thread to initiate multiple operations simultaneously. This design allows a single-threaded application to execute I/O operations concurrently without incurring the overhead or design complexity associated with conventional multithreaded mechanisms.

Choose the proactive I/O model when

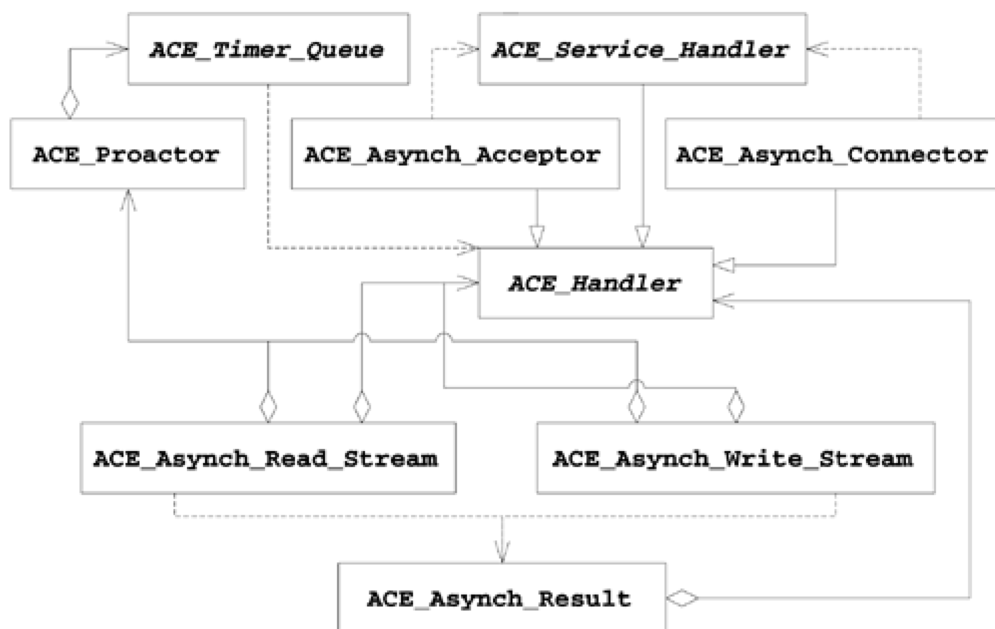
- The IPC mechanisms in use, such as Windows Named Pipes, require it

8.2 How to Send and Receive Data

The procedure for sending and receiving data asynchronously is a bit different from using synchronous transfers. We'll look at an example, explore what the example does, and point out some similarities and differences between using the Proactor framework and the Reactor framework.

The Proactor framework encompasses a relatively large set of highly related classes, so it's impossible to discuss them in order without forward references. We will get through them all by the end of the chapter. Figure 8.1 shows the Proactor framework's classes in relation to each other; you can use the figure to keep some context as we progress through the chapter.

Figure 8.1. Classes in the Proactor framework



The following code declares a class that performs the same basic work as the examples in the previous two chapters, introducing the primary classes involved in initiating and completing I/O requests on a connected TCP/IP socket:

```
#include "ace/Asynch_IO.h"

class HA_Proactive_Service : public ACE_Service_Handler
{
public:
  ~HA_Proactive_Service ()
   {
     if (this->handle () != ACE_INVALID_HANDLE)
       ACE_OS::closesocket (this->handle ());
   }

  virtual void open (ACE_HANDLE h, ACE_Message_Block&);

  // This method will be called when an asynchronous read
  // completes on a stream.
  virtual void handle_read_stream
    (const ACE_Asynch_Read_Stream::Result &result);

  // This method will be called when an asynchronous write
  // completes on a stream.
  virtual void handle_write_stream
    (const ACE_Asynch_Write_Stream::Result &result);

private:
  ACE_Asynch_Read_Stream reader_;
```

8.3 Establishing Connections

ACE provides two factory classes for proactively establishing TCP/IP connections using the Proactor framework:

1. ACE_Asynch_Acceptor, to initiate passive connection establishment
2. ACE_Asynch_Connector to initiate active connection establishment

When a TCP/IP connection is established using either of these classes, the ACE Proactor framework creates a service handler derived from ACE_Service_Handler, such as HA_Proactive_Service, to handle the new connection. The ACE_Service_Handler class, the base class of all asynchronously connected services in the ACE Proactor framework, is derived from ACE_Handler, so the service class can also handle I/O completions initiated in the service.

ACE_Asynch_Acceptor is a fairly easy class to program with. It is very straightforward in its default case and adds two hooks for extending its capabilities. The following example uses one of the hooks:

```
#include "ace/Asynch_Acceptor.h"
#include "ace/INET_Addr.h"

class HA_Proactive_Acceptor :
    public ACE_Asynch_Acceptor<HA_Proactive_Service>
{
public:
  virtual int validate_connection
    (const ACE_Asynch_Accept::Result& result,
     const ACE_INET_Addr &remote,
     const ACE_INET_Addr &local);
};
```

We declare HA_Proactive_Acceptor to be a new class derived from ACE_Asynch_Acceptor. As you can see, ACE_Asynch_Acceptor is a class template, similar to the way ACE_Acceptor is. The template argument is the type of ACE_Service_Handler-derived class to use for each new connection.

The validate_connection() method is a hook method defined on both ACE_Asynch_Acceptor and ACE_Asynch_Connector. The framework calls this method after accepting a new connection, before obtaining a new service handler for it. This method gives the application a chance to verify the connection and/or the address of the peer. Our example checks whether the peer is on the same IP network as we are:

```
int
HA_Proactive_Acceptor::validate_connection (
   const ACE_Asynch_Accept::Result&,
   const ACE_INET_Addr& remote,
   const ACE_INET_Addr& local)
{

  struct in_addr *remote_addr =
    ACE_reinterpret_cast (struct in_addr*,
                          remote.get_addr ());
  struct in_addr *local_addr =
    ACE_reinterpret_cast (struct in_addr*,
                          local.get_addr ());
  if (inet_netof (*local_addr) == inet_netof (*remote_addr))
    return 0;
```

8.4 The ACE_Proactor Completion Demultiplexer

The ACE_Proactor class drives completion handling in the ACE Proactor framework. This class waits for completion events that indicate that one or more operations started by the I/O factory classes have completed, demultiplexes those events to the associated completion handlers, and dispatches the appropriate hook method on each completion handler. Thus, for any asynchronous I/O completion event processing to take place—whether I/O or connection establishment—your application must run the proactor's event loop. This is usually as simple as inserting the following in your application:

```
ACE_Proactor::instance ()->proactor_run_event_loop ();
```

Asynchronous I/O facilities vary wildly between operating systems. To maintain a uniform interface and programming method across all of them, the ACE_Proactor class, like ACE_Reactor, uses the Bridge pattern to maintain flexibility and extensibility while allowing the Proactor framework to function with differing asynchronous I/O implementations. We briefly describe the implementation-specific Proactor classes next.

8.4.1 ACE_WIN32_Proactor

ACE_WIN32_Proactor is the ACE_Proactor implementation on Windows. This class works on Windows NT 4.0 and newer Windows platforms, such as Windows 2000 and Windows XP, but not on Windows 95, 98, ME, or CE, however, as these platforms don't support asynchronous I/O.

ACE_WIN32_Proactor uses an I/O completion port for completion event detection. When initializing an asynchronous operation factory, such as ACE_Asynch_Read_Stream or ACE_Asynch_Write_Stream, the I/O handle is associated with the Proactor's I/O completion port. In this implementation, the Windows GetQueuedCompletionStatus() function paces the event loop. Multiple threads can execute the ACE_WIN32_Proactor event loop simultaneously.

8.4.2 ACE_POSIX_Proactor

The ACE Proactor implementations on POSIX systems present multiple mechanisms for initiating I/O operations and detecting their completions. Moreover, Sun's Solaris Operating Environment offers its own proprietary version of asynchronous I/O. On Solaris 2.6 and higher, the performance of the Sun-specific asynchronous I/O functions is significantly higher than that of Solaris's POSIX.4 AIO implementation. To take advantage of this performance improvement, ACE also encapsulates this mechanism in a separate set of classes.

The encapsulated POSIX asynchronous I/O mechanisms support read() and write() operations but not TCP/IP connection related operations. To support the functions of ACE_Asynch_Acceptor and ACE_Asynch_Connector, a separate thread is used to perform connection-related operations. Therefore, you should be aware that your program will be running multiple threads when using the Proactor framework on POSIX platforms. The internals of ACE keep you from needing to handle events in different threads, so you don't need to add any special locking or synchronization. Just be aware of what's going on if you're in the debugger and see threads that your program didn't spawn.

8.5 Using Timers

In addition to its I/O-related capabilities, the ACE Proactor framework offers settable timers, similar to those offered by the ACE Reactor framework. They're programmed in a manner very similar to programming timers with the Reactor framework, but the APIs are slightly different. Check the reference documentation for complete details.

8.6 Other I/O Factory Classes

As with the Reactor framework, the Proactor framework has facilities to work with many different types of I/O endpoints. Unlike the synchronous IPC wrapper classes in ACE, which have a separate class for each type of IPC, the Proactor framework offers a smaller set of factory classes and relies on you to supply each with a handle. An I/O handle from any ACE IPC wrapper class, such as ACE_SOCK_Stream or ACE_FILE_IO, may be used with these I/O factory classes as listed:

- ACE_Asynch_Read_File and ACE_Asynch_Write_File for files and Windows Named Pipes
- ACE_Asynch_Transmit_File to transmit files over a connected TCP/IP stream
- ACE_Asynch_Read_Dgram and ACE_Asynch_Write_Dgram for UDP/IP datagram sockets

8.7 Combining the Reactor and Proactor Frameworks

Sometimes, you have a Reactor-based system and need to add an IPC type that doesn't work with the Reactor model. Or, you may want to use a Reactor feature, such as signals or signalable handles, with a Proactor-based application. These situations occur most often on Windows or in a multiplatform application in which Windows is one of its platforms. Sometimes, your application's I/O needs work better with the Proactor in some situations and better with the Reactor in others and you want to simplify development and maintenance as much as possible. Three different scenarios can usually be used to accommodate mixing of the two frameworks.

8.7.1 Compile Time

It's possible to derive your application's service handler class(es) from either ACE_Svc_Handler or ACE_Service_Handler, switchable at compile time, based on whether you're building for the Reactor framework or the Proactor framework. Rather than perform any real data processing in the callbacks, arrange your class to follow these guidelines.

- Standardize on handling data in ACE_Message_Block objects. Using the Proactor framework, you already need to do this, so this guideline has the most effect when working in the Reactor world. You simply need to get used to working with ACE_Message_Block instead of native arrays.
- Centralize the data-processing functionality in a private, or protected, method that's not one of the callbacks. For example, move the processing code to a method named do_the_work() or process_input(). The work method should accept an ACE_Message_Block with the data to work on. If the work requires that data also be sent in the other direction, put it in another ACE_Message_Block and return it.
- (Proactor): In the completion handler callback—for example, handle_read_stream(), after checking transfer status, pass the message block with the data to the work method.
- (Reactor): When receiving data in handle_input(), read it into an ACE_Message_Block and then call the work method, just as you do in the Proactor code.

8.7.2 Mix Models

Recall that it's possible to register a signalable handle with the ACE_WFMO_Reactor on Windows. Thus, if you want to use overlapped Windows I/O, you could use an event handle with the overlapped I/O and register the event handle with the reactor. This is a way to add a small amount of nonsockets I/O work—if, for example, you need to work with a named pipe—to the reactor on Windows but don't have the inclination or the interest in mixing Reactor and Proactor event loops.

8.7.3 Integrating Proactor and Reactor Events Loops

Both the Proactor and Reactor models require event-handling loops, and it is often useful to be able to use both models in the same program. One possible method for doing this is to run the event loops in separate threads. However, that introduces a need for multithreaded synchronization techniques. If the program is single threaded, however, it would be much better to integrate the event handling for both models into one mechanism. ACE provides this integration mechanism for Windows programs by providing a linkage from the Windows implementation of the ACE_Proactor class to the ACE_WFMO_Reactor class, which is the default reactor type on Windows.

The ACE mechanism is based on the ACE_WFMO_Reactor class's ability to include a HANDLE in the event sources it waits for (see Section 7.7.2). The ACE_WIN32_Proactor class uses an I/O completion port internally to manage its event dispatching. However, because an I/O completion port handle is not waitable, it can't be registered

8.8 Summary

The ACE Proactor framework provides a portable way to implement asynchronous I/O capabilities into your application. Asynchronous I/O can often be an efficient way to handle more I/O endpoints than you can efficiently use with the Reactor framework. Asynchronous I/O can also be a good choice for situations in which you can benefit from highly parallelized I/O operations but don't want to use multiple threads.

This chapter described the Proactor framework's capabilities and showed how to implement the example server from earlier chapters, using the Proactor framework. Because asynchronous I/O is not universally available and not completely interchangeable with the Reactor framework, we also discussed ways to work with both frameworks in the same application.

Chapter 9. Other IPC Types

So far, we have focused on TCP/IP (ACE_SOCK_Stream and friends). ACE also offers many other IPC wrapper classes that support both interhost and intrahost communication. Keep in mind that intrahost communications is a very simplified host-to-host communication situation, and interhost IPC mechanisms all work perfectly fine for communication between collocated entities. Like the TCP/IP Sockets wrappers, most of the IPC wrappers offer an interface compatible with using them in the ACE Acceptor-Connector framework (ACE_Acceptor, ACE_Connector, and ACE_Svc_Handler classes).

9.1 Interhost IPC with UDP/IP

UDP is a datagram-oriented protocol that operates over IP. Therefore, as with TCP/IP, UDP uses IP addressing. Also as with TCP, datagrams are demultiplexed within each IP address, using a port number. UDP port numbers have the same range as TCP port numbers but are distinct. Because the addressing information is so similar between UDP and TCP, ACE's UDP classes use the same addressing class as those wrapping TCP do: ACE_INET_Addr.

When deciding whether to use UDP communication, consider these three differences between UDP and TCP.

1. UDP is datagram based, whereas TCP is stream based. If a TCP peer sends, for example, three 256-byte buffers of data, the connected peer application will receive 768 bytes of data in the same order they were transmitted but may receive the data in any number of separate chunks, without any guarantee of where the breaks between chunks will be, if any. Conversely, if a UDP peer sends three 256-byte datagrams, the receiving peer will receive anywhere from zero to all three of them. Any datagram that is received will, however, be the complete 256-byte datagram sent; none will be broken up or coalesced. Therefore, UDP transmissions are more record oriented, whereas with TCP, you need a way to extract the streamed data correctly, referred to as unmarshaling.

2. UDP makes no guarantees about the arrival or order of data. Whereas TCP guarantees that any data received is precisely what was sent and that it arrives in order, UDP makes only best-effort delivery. As hinted at earlier, three 256 byte datagrams sent may not all be received. Any that are received will be the complete, correct datagram that was sent; however, datagrams may be lost or reordered in transit. Thus, although UDP relieves you of the need to marshal and unmarshal data on a stream of bytes, you are responsible for any needed reliability that your protocol and/or application requires.

3. Whereas TCP is a one-to-one connection between two peers, UDP offers several modes of operation: unicast, broadcast, and multicast. Unicast is a one-to-one operation, similar to TCP. In Broadcast mode, each datagram sent is broadcast to every listener on the network or subnetwork the datagram is broadcast on. This mode requires a broadcastable network medium, such as Ethernet. Because it must be processed by each station on the attached network, broadcast network traffic can cause network traffic problems and is generally frowned on. The third mode—multicast—solves the traffic issue of broadcast. Interested applications must join multicast groups that have unique IP addresses. Any datagram sent to a multicast group is received only by those stations subscribed to the group. Thus, multicast has the one-to-many nature of broadcast without all the attendant traffic issues.

We'll look at brief examples using UDP in the three addressing modes. Note that all the UDP classes we'll look at can be used with the ACE Reactor framework and that the I/O UDP classes can be used as the peer stream template argument with the ACE_Svc_Handler class template. The unicast mode ACE_SOCK_CODgram class can also produce a handle that's usable with the Proactor framework's ACE_Asynch_Read_Dgram and ACE_Asynch_Write_Dgram I/O factory classes.

9.1.1 Unicast Mode

Let's see an example of how to send some data on a unicast UDP socket. For this use case, ACE offers the ACE_SOCK_Dgram class:

```
#include "ace/OS.h"
#include "ace/Log_Msg.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Dgram.h"

int send_unicast (const ACE_INET_Addr &to)
```

9.2 Intrahost Communication

The classes described in this section can be used for intrahost communication only. They can offer some simplicity over interhost communications, owing to simplified addressing procedures. Intrahost IPC can also be significantly faster than interhost IPC, owing to the absence of heavy protocol layers and network latency, as well as the ability to avoid relatively low bandwidth communications channels. However, some interhost communications facilities, such as TCP/IP sockets, also work quite well for intrahost application because of improved optimization in the protocol implementations. TCP/IP sockets are also the most commonly available IPC mechanism across a wide variety of platforms, so if portability is a high concern, TCP/IP sockets can simplify your code greatly. The bottom line in IPC mechanism selection is to weigh the options, maybe do your own performance benchmarks, and decide what's best in your particular case. Fortunately, ACE's IPC mechanisms offer very similar programming interfaces, so it's relatively easy to exchange them for testing.

9.2.1 Files

The ACE_FILE_IO and ACE_FILE_Connector classes implement file I/O in a way that allows their use in the Acceptor-Connector framework, albeit only with the Connector side. The associated addressing class is ACE_FILE_Addr, which encapsulates the pathname to a file.

9.2.2 Pipes and FIFOs

Pipes and FIFOs are UNIX mechanisms. FIFOs are also sometimes referred to as named pipes but are not the same thing as Windows Named Pipes. Following are the more common classes in this area. Most are platform specific, so check the reference documentation for full details.

- ACE_FIFO_Recv, ACE_FIFO_Send, ACE_FIFO_Recv_Msg, and ACE_FIFO_Send_Msg work with UNIX/POSIX FIFOs in both stream and message mode. There is no addressing class; specify the FIFO name to the particular data transfer class you need.

- ACE_Pipe provides a simple UNIX/POSIX pipe. Although the pipe is distinctly a UNIX/POSIX capability, ACE_Pipe emulates it on Windows, using loopback TCP/IP sockets. It works in a pinch, but be aware of the difference. If you must grab one of the pipe handles and use it for low level I/O, ACE_OS::read() and ACE_OS::write() will not work on Windows, although it will most everywhere else, because it's a socket handle on Windows; use ACE_OS::recv() and ACE_OS::send() if you must use one of these handles for low-level I/O on Windows. As with the FIFO classes, there's no addressing class, as pipes don't have names or any other addressing method.

- ACE_SPIPE_Acceptor, ACE_SPIPE_Connector, ACE_SPIPE_Stream, and ACE_SPIPE_Addr follow the scheme for easy substitution in the Acceptor-Connector framework. Beware, though; these classes wrap the STREAMS pipe facility on UNIX/POSIX, where it's available, and Named Pipes on Windows. The two aren't really the same, but they program similarly, and if you need to use one on a given platform, you simply need to know to use ACE_SPIPE.

9.2.3 Shared Memory Stream

This set of classes comprises ACE_MEM_Acceptor, ACE_MEM_Connector, ACE_MEM_Stream, and ACE_MEM_Addr. The classes in the shared memory stream facility fits into the Acceptor-Connector framework but uses shared memory—memory-mapped files, actually—for data transfer. This can result in very good performance because data isn't transferred but rather is placed in memory that's shared between processes.

As you may imagine, synchronizing access to this shared data is where the tricky parts of this facility enter. Also, because there's no way to use select() on one of these objects, the ACE_MEM_Stream class can adapt its

9.3 Summary

Interprocess Communication (IPC) is an important part of many applications and is absolutely foundational to networked applications. Today's popular operating environments offer a wide range of IPC mechanisms, accessible via varying APIs, for both interhost and intrahost communication.

ACE helps to unify the programming interfaces to many disparate IPC types, as well as avoid the accidental complexity associated with programming at the OS API level. This uniformity of class interfaces across ACE's IPC classes makes it fairly easy to substitute them to meet changing requirements or performance needs.

Part III: Process and Thread Management

Chapter 10. Process Management

Chapter 11. Signals

Chapter 12. Basic Multithreaded Programming

Chapter 13. Thread Management

Chapter 14. Thread Safety and Synchronization

Chapter 15. Active Objects

Chapter 16. Thread Pools

Chapter 10. Process Management

Processes are the primary abstraction an operating system uses to represent running programs. Unfortunately, the meaning of the term process varies widely. On most general-purpose operating systems, such as UNIX and Windows, a process is seen as a resource container that manages the address space and other resources of a program. This is the abstraction that is supported in ACE. Some operating systems, such as VxWorks, do not have processes at all but instead have one large address space in which tasks run. The ACE process classes are not pertinent for these operating systems.

In this chapter, we first explain how to use the simple ACE_Process wrapper class to create a process and then manage child process termination. Next, we discuss how you can protect globally shared system resources from concurrent access by one or more processes, using special mutexes for process-level locking. Finally, we look at the high-level process manager class that offers integration with the Reactor framework.

10.1 Spawning a New Process

ACE hides all process creation and control APIs from the user in the ACE_Process wrapper class. This wrapper class allows a programmer to spawn new processes and subsequently wait for their termination. You usually use one ACE_Process object for each new process and are allowed to set several options for the child process:

- Setting standard I/O handles
- Specifying how handle inheritance will work between the two processes
- Setting the child's environment block and command line
- Specifying security attributes on Windows or set uid/gid/euid on UNIX.

For those of you from the UNIX world, the spawn() method does not have semantics similar to the fork() system call but is instead similar to the system() function available on most UNIX systems. You can force ACE_Process to do a simple fork() but in most cases are better off using ACE_OS::fork() to accomplish a simple process fork, if you need it.

Spawning a process using the ACE_Process class is a two-step process.

1. Create a new ACE_Process_Options object specifying the desired properties of the new child process.
2. Spawn a new process using the ACE_Process::spawn() method.

In the next example, we illustrate creating a slave process and then waiting for it to terminate. To do this, we create an object of type Manager, which spawns another process running the same example program, albeit with different options. Once the slave process is created, it creates an object of type Slave, which performs some artificial work and exits. Meanwhile, the master process waits for the slave process to complete before it too exits.

Because the same program is run as both the master and the slave, the command line arguments are used to distinguish which mode it is to run in; if there are arguments, the program knows to run in slave mode and otherwise runs in master mode:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  if (argc > 1)    // Slave mode
    {
      Slave s;
      return s.doWork ();
    }

  // Else, Master mode
  Manager m (argv[0]);
  return m.doWork ();
}
```

The Manager class has a single public method that is responsible for setting the options for the new slave process, spawning it, and then waiting for its termination:

10.2 Using the ACE_Process_Manager

Besides the relatively simple ACE_Process wrapper, ACE also provides a sophisticated process manager, ACE_Process_Manager, which allows a user, with a single call, to spawn and wait for the termination of multiple processes. You can also register event handlers that are called back when a child process terminates.

10.2.1 Spawning and Terminating Processes

The spawn() methods available in the ACE_Process_Manager class are similar to those available with ACE_Process. Using them entails creating an ACE_Process_Options object and passing it to the spawn() method to create the process. With ACE_Process_Manager, you can additionally spawn multiple processes at once, using the spawn_n() method. You can also wait for all these processes to exit and correctly remove all the resources held by them. In addition, you can forcibly terminate a process that was previously spawned by ACE_Process_Manager.

The following example illustrates some of these new process manager features:

```
#include "ace/Process_Manager.h"

static const int NCHILDREN = 2;

int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  if (argc > 1)     // Running as a child.
    {
      ACE_OS::sleep (10);
    }
  else             // Running as a parent.
    {
      // Get the processwide process manager.
      ACE_Process_Manager* pm = ACE_Process_Manager::instance ();

      // Specify the options for the new processes
      // to be spawned.
      ACE_Process_Options options;
      options.command_line (ACE_TEXT ("%s a"), argv[0]);

      // Spawn two child processes.
      pid_t pids[NCHILDREN];
      pm->spawn_n (NCHILDREN, options, pids);

      // Destroy the first child.
      pm->terminate (pids[0]);

      // Wait for the child we just terminated.
      ACE_exitcode status;
      pm->wait (pids[0], &status);

      // Get the results of the termination.

#if !defined(ACE_WIN32)
      if (WIFSIGNALED (status) != 0)
        ACE_DEBUG ((LM_DEBUG,
                    ACE_TEXT ("%d died because of a signal ")
                    ACE_TEXT ("of type %d\n"),
                    pids[0], WTERMSIG (status)));
#else
      ACE_DEBUG
        ((LM_DEBUG,
          ACE_TEXT ("The process terminated with exit code %d\n"),
          status));
#endif /*ACE_WIN32*/
```

10.3 Synchronization Using ACE_Process_Mutex

To synchronize threads, you need synchronization primitives, such as mutexes or semaphores. (We discuss these primitives in significant detail in Section 12.2.) When executing in separate processes, the threads are running in different address spaces. Synchronization between such threads becomes a little more difficult. In such cases, you can either

- Create the synchronization primitives that you are using in shared memory and set the appropriate options to ensure that they work between processes
- Use the special process synchronization primitives that are provided as a part of the ACE library

ACE provides a number of process-scope synchronization classes that are analogous to the thread-scope wrappers discussed in Chapter 14. This section explains how you can use the ACE_Process_Mutex class to ensure synchronization between threads running in different processes.

ACE provides, in the form of the ACE_Process_Mutex class, for named mutexes that can be used across address spaces. Because the mutex is named, you can recreate an object representing the same mutex by passing in the same name to the constructor of ACE_Process_Mutex.

In the next example, we create a named mutex: GlobalMutex. We then create two processes that cooperatively share an imaginary global resource, coordinating their access by using the mutex. The two processes both do this by creating an instance of the GResourceUser, an object that intermittently uses the globally shared resource.

We use the same argument-length trick that we used in the previous example to start the same program in different modes. If the program is started to be the parent, it spawns two child processes. If started as a child process, the program gets the named mutex GlobalMutex from the OS by instantiating an ACE_Process_Mutex object, passing it the name GlobalMutex. This either creates the named mutex—if this is the first time we asked for it—or attaches to the existing mutex—if the second process does the construction. The mutex is passed to a resource-acquirer object that uses it to ensure protected access to a global resource.

Again, note that even though we create separate ACE_Process_Mutex objects—each child process creates one—they both refer to the same shared mutex. The mutex itself is managed by the operating system, which recognizes that both mutex instances refer to the same GlobalMutex:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  if (argc > 1)       // Run as the child.
    {
      // Create or get the global mutex.
      ACE_Process_Mutex mutex ("GlobalMutex");

      GResourceUser acquirer (mutex);
      acquirer.run ();
    }
  else                  // Run as the parent.
    {
      ACE_Process_Options options;
      options.command_line ("%s a", argv[0]);
      ACE_Process processa, processb;

      pid_t pida = processa.spawn (options);
      pid_t pidb = processb.spawn (options);

      ACE_DEBUG ((LM_DEBUG,
```

10.4 Summary

In this chapter, we introduced the ACE classes that support process creation, life-cycle management, and synchronization. We looked at the simple ACE_Process wrapper and the sophisticated ACE_Process_Manager. We also looked at synchronization primitives that can be used to synchronize threads that are running in separate processes.

Chapter 11. Signals

Signals act as software interrupts and indicate to the application such asynchronous events as a user pressing the interrupt key on a terminal, a broken pipe between processes, job control functions, and so on. To handle signals, a program can associate a signal handler with a particular signal type. For example, for the interrupt key signal, you can set up a handler that instead of terminating the process, first asks whether the user wishes to terminate. In most cases, your application will want to handle most error signals so that you can either gracefully terminate or retry the current task.

Once the signal is raised, the associated signal handler is invoked in signal context, which is separate the interrupted main execution context. After the signal handler returns, execution continues back in the main context from wherever it happened to be right before the signal was received, as if the interruption never occurred.

Windows provides minimal support for signals for ANSI (American National Standards Institute) compatibility. A minimal set of signals is available, but even fewer are raised by the operating system. Therefore, the usefulness of signals is generally somewhat limited for Windows programmers.

In Chapter 7, we explained how you can use the ACE Reactor framework to handle signal events, along with several other event types. Here, we talk about how to use the signal-handling features of ACE, independent of the Reactor framework. This comes in handy in the following situations.

- You do not want to add the extra complexity of the reactor, as it isn't needed.
- The signal-handling actions aren't associated with any of your event handlers.
- You are developing a resource-constrained system in which you cannot afford to waste any memory.

In this chapter, we explain how ACE makes it easy to set up one or more handlers for a signal type. We start by looking at the simple ACE_Sig_Action wrapper, which calls a user-specified handler function when a signal occurs. We then look at the higher-level ACE_Sig_Handler class, which invokes an ACE_Event_Handler::handle_signal() callback when the specified signal occurs. Finally, we talk about the ACE_Sig_Guard class, which allows you to guard certain sections of your code from signal interruptions.

11.1 Using Wrappers

The POSIX sigaction() call enables a programmer to associate an action, such as the execution of a callback function, with the occurrence of a particular signal. ACE provides a wrapper around the sigaction() call. This wrapper provides a type-safe interface for signal registration. ACE also provides a type-safe wrapper to one of its argument types, sigset_t. This type represents a collection of signals and is discussed in detail in Section 7.3.2.

The following example illustrates the use of wrappers to register callback functions for a few signals:

```
#include "ace/Signal.h"

// Forward declarations.
static void my_sighandler (int signo);
static void register_actions ();

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_TRACE (ACE_TEXT ("::main"));

  ::register_actions ();    // Register actions to happen.

  // This will be raised immediately.
  ACE_OS::kill (ACE_OS::getpid(), SIGUSR2);

  // This will pend until the first signal is completely
  // handled and returns, because we masked it out
  // in the registerAction call.

  ACE_OS::kill (ACE_OS::getpid (), SIGUSR1);

  while (ACE_OS::sleep (100) == -1)
    {
      if (errno == EINTR)
        continue;
      else
        ACE_OS::exit (1);
    }
  return 0;
}
```

When we enter the program, we first register actions, using the ACE-provided ACE_Sig_Action class. Then we explicitly raise certain signals, using the ACE_OS::kill() method. In this case, we are asking for SIGUSR2 to be sent to the current process, followed by SIGUSR1; both of these signal types are meant to be user definable.

Our signal handler function follows:

```
static void my_sighandler (int signo)
{
  ACE_TRACE (ACE_TEXT ("::my_sighandler"));

  ACE_OS::kill (ACE_OS::getpid (), SIGUSR1);

  if (signo == SIGUSR1)
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("Signal SIGUSR1\n")));
  else
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("Signal SIGUSR2\n")));

  ACE_OS::sleep (10);
}
```

11.2 Event Handlers

Bundled above the type-safe C++ wrappers for sigaction and sigset_t is an object-oriented event handler-based signal registration and dispatching scheme. This is available through the ACE_Sig_Handler class, which allows a client programmer to register ACE_Event_Handler-based objects for callback on the occurrence of signals. The ACE_Event_Handler type is central to the ACE Reactor framework, but there is no registration with a reactor when used with ACE_Sig_Handler.

As client programmers, we must subclass from ACE_Event_Handler and implement the callback method handle_signal():

```
class MySignalHandler : public ACE_Event_Handler
{
public:
  MySignalHandler (int signum) : signum_(signum)
  { }

  virtual ~MySignalHandler()
  { }

  virtual int handle_signal (int signum,
                             siginfo_t * = 0,
                             ucontext_t * = 0)
  {
    ACE_TRACE (ACE_TEXT ("MySignalHandler::handle_signal"));

    // Make sure the right handler was called back.
    ACE_ASSERT (signum == this->signum_);

    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("%S occured\n"), signum));
    return 0;
  }

private:
  int signum_;
};
```

We start by creating a signal handler class that publicly derives from the ACE_Event_Handler base class. We need to implement only the handle_signal() method here, as our event handler can handle only "signal"-based events. The siginfo_t and ucontext_t are also passed back to the handle_signal() method, in addition to the signal number, when the signal event occurs. This is in accordance with the new sigaction() signature; we will discuss the siginfo_t and ucontext_t types in Section 11.2.1 and Section 11.2.2, respectively.

```
int ACE_TMAIN (int, ACE_TCHAR *[])
{
  MySignalHandler h1 (SIGUSR1), h2 (SIGUSR2);
  ACE_Sig_Handler handler;
  handler.register_handler (SIGUSR1, &h1);
  handler.register_handler (SIGUSR2, &h2);

  ACE_OS::kill (ACE_OS::getpid (), SIGUSR1);
  ACE_OS::kill (ACE_OS::getpid (), SIGUSR2);


  int time = 10;
  while ((time = ACE_OS::sleep (time)) == -1)
    {
      if (errno == EINTR)
```

11.3 Guarding Critical Sections

Signals act a lot like asynchronous software interrupts. These interrupts are usually generated external to the application—not the way we have been generating them, using kill(), to run our examples. When a signal is raised, the thread of control in a single-thread application—or one of threads in a multithreaded application—jumps off to the signal-handling routine, executes it, and then returns to regular processing. As in all such scenarios, certain small regions of code can be treated as critical sections, during the execution of which you do not want to be interrupted by any signal.

ACE provides a scope-based signal guard class that you can use to disable or mask signal processing during the processing of a critical section of code, although some signals, such as SIGSTOP and SIGKILL, cannot be masked:

```
class MySignalHandler : public ACE_Event_Handler
{
public:
  virtual int handle_signal (int signo,
                             siginfo_t * = 0,
                             ucontext_t * = 0)
  {
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("Signal %d\n"), signo));
    return 0;
  }
};

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  MySignalHandler sighandler;
  ACE_Sig_Handler sh;
  sh.register_handler (SIGUSR1, &sighandler);

  ACE_Sig_Set ss;
  ss.sig_add (SIGUSR1);
  ACE_Sig_Guard guard (&ss);
  {
    ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT ("Entering critical region\n")));
    ACE_OS::sleep (10);
    ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT ("Leaving  critical region\n")));
  }

  // Do other stuff.

  return 0;
}
```

Here, an ACE_Sig_Guard is used to protect the critical region of code against all the signals that are in the ACE_Sig_Set ss. In this case we have added only the signal SIGUSR1. This means that the scoped code is "safe" from SIGUSR1 until the scope closes.

To test this, we ran this program and used the UNIX kill(1) utility to send the process SIGUSR1 (signal number 16):

```
$ SigGuard&
[1]        15191
$ Entering critical region
$kill 16 15191
$ Leaving critical region
$ Signal   16
```

11.4 Signal Management with the Reactor

As we mentioned in the beginnning of this chapter, the reactor is a general-purpose event dispatcher that can handle the dispatching of signal events to event handlers in a fashion similar to ACE_Sig_Handler. In fact, in most reactor implementations, ACE_Sig_Handler is used behind the scenes to handle the signal dispatching. It is important to remember that signal handlers are invoked in signal context, unlike the other event types that are handled by the Reactor framework. You can, however, transfer control back to a defined point of control in user context using reactor notifications. For more on this topic, see Section 7.4.

11.5 Summary

In this chapter, we looked at how signals, which are asynchronous software interrupts, are handled portably with ACE. We explained how to use the low-level sigaction and sigset_t wrappers to set up our own signal handler functions. We then discussed how ACE supports object-based callbacks with the ACE_Event_Handler abstract base class. Stacking signal handlers for a single signal number was also introduced. Finally, we showed you how to protect regions of code from being interrupted by signals by using the ACE_Sig_Guard class.

Chapter 12. Basic Multithreaded Programming

Multithreaded programming has become more and more of a necessity in today's software. Whereas yesterday, most general-purpose operating systems provided programmers with only user-level thread libraries, most of today's operating systems provide real preemptive multitasking. As the prices of multiprocessor machines fall, their use has also become prevalent. The scale of new software also appears to be growing. Whereas formerly, concurrent users numbered in the hundreds or thousands, today that number has jumped tenfold. Development of real time software has always required the use of thread priorities; with the surge of intelligent embedded devices, many more developers are getting involved in this field. All these factors combined make it more and more important for developers to be familiar with threading concepts and their productive use.

This chapter introduces the basics of using threads with the ACE toolkit. Because ACE provides so much in the area of threads, we have divided the discussion on threads into two chapters. This chapter covers the basics: creating new threads of control, safety primitives that ensure consistency when you have more than one thread accessing a global structure, and event and data communication between threads. Chapter 13 discusses more advanced topics, such as thread scheduling and management.

12.1 Getting Started

By default, a process is created with a single thread, which we call the main thread. This thread starts executing in the main() function of your program and ends when main() completes. Any extra threads that your process may need have to be explicitly created. To create your own thread with ACE, all you have to do is create a subclass of the ACE_Task_Base class and override the implementation of the virtual svc() method. The svc() method serves as the entry point for your new thread; that is, your thread starts in the svc() method and ends when the svc() method returns, in a fashion similar to the main thread.

You will often find yourself using extra threads to help process incoming messages for your network servers. This prevents clients that do not require responses from blocking on the network server, waiting for long-running requests to complete. In our first example, we create a home automation command handler class, HA_CommandHandler, that is responsible for applying long-running command sequences to the various devices that are connected on our home network. For now, we simulate the long-running processing with a sleep call. We print out the thread identifier for both the main thread and the command handler thread, using the ACE_DEBUG() macro's %t format specifier, so that we can see the two threads running in our debug log:

```
#include "ace/Task.h"

class HA_CommandHandler : public ACE_Task_Base
{
public:
  virtual int svc (void)
  {
    ACE_DEBUG
      ((LM_DEBUG, ACE_TEXT ("(%t) Handler Thread running\n")));
    ACE_OS::sleep (4);
    return 0;
  }
};

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_DEBUG
    ((LM_DEBUG, ACE_TEXT ("(%t) Main Thread running\n")));

  HA_CommandHandler handler;
  int result = handler.activate ();
  ACE_ASSERT (result == 0);

  handler.wait ();
  return 0;
}
```

To start the thread, you must create an instance of HA_CommandHandler and call activate() on it. Before doing this, we print out the main thread's identifier so that we can compare both child and main identifiers in our output debug log.

After activating the child thread, the main thread calls wait() on the handler object, waiting for its threads to complete before continuing and falling out of the main() function. Once the child thread completes the svc() method and exits, the wait() call in the main thread will complete, control will fall out of the main() function, and the process will exit. Why does the main thread have to wait for the child thread to complete? On many platforms, once the main thread returns from the main() function, the C runtime sees this as an indication that the process is ready to exit and destroys the entire running process, including the child thread. If we allowed this to happen, the program might exit before the child thread ever got scheduled and got a chance to execute.

The output shows the two threads—the main thread and the child command handler thread—running:

12.2 Basic Thread Safety

One of the most difficult problems you deal with when writing multithreaded programs is maintaining consistency of all globally available data. Because you have multiple threads accessing the same objects and structures, you must make sure that any updates made to these objects are safe. What safety means in this context is that all state information remains in a consistent state.

ACE provides a rich array of primitives to help you to achieve this goal. We cover a few of the most useful and commonly used primitives in the next few sections and continue coverage on the rest of these components in Chapter 14.

12.2.1 Using Mutexes

Mutexes, the simplest protection primitive available, provide a simple acquire(), release() interface. If successful in getting the mutex, the acquiring thread, acquire(), continues forward; otherwise, it blocks until the holder of the mutex releases it by using release().

As shown in Table 14.1, ACE provides several mutex classes. ACE_Mutex can be used as a lightweight synchronization primitive for threads and as a heavyweight cross-process synchronization primitive.

In the next example, we add a device repository to our home automation example. This repository contains references to all the devices connected to our home network, as well as the interface to apply command sequences to the various devices connected to our home network. Let us suppose that only one thread can make updates in the repository at a time, without causing consistency problems.

The repository creates and manages an ACE_Thread_Mutex object as a data member that it uses to ensure the consistency constraint. This is a common idiom that you will find yourself using on a regular basis. Whenever it calls the update_device() method, a thread first has to acquire the mutex before continuing forward, as only one thread can have the mutex at a time; at no point will two threads simultaneously update the state of the repository. It is important that release() be called on the mutex so that other threads can acquire the repository mutex and update the repository after the first thread is done. When the repository is destroyed, the destructor of the mutex will ensure that it properly releases all resources that it holds:

```
class HA_Device_Repository
{
public:
  HA_Device_Repository ()
  { }

  void update_device (int device_id)
  {
    mutex_.acquire ();
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%t) Updating device %d\n"),
                device_id));
    ACE_OS::sleep (1);
    mutex_.release ();
  }
private:
  ACE_Thread_Mutex mutex_;
};
```

To illustrate the mutex in action, we modify HA_CommandHandler to call update_device() on the repository and then create two handler tasks that compete with each other, trying to update devices in the repository at the same time:

12.3 Intertask Communication

When writing multithreaded programs, you will often feel the need for your tasks to communicate with one another. This communication may take the form of something simple, such as one thread informing another that it is time to exit, or something more complicated, such as communicating threads passing data back and forth.

In general, intertask communication can be divided into two broad categories:

1. State change or event notifications, whereby only the event occurrence needs to be communicated, but no data is passed between the two threads
2. Message passing, whereby data is passed between the two threads, possibly forming a work chain, in which the first thread processes the data and then passes it along to the next for further processing

12.3.1 Using Condition Variables

A thread can use condition variables to communicate a state change, an event arrival, or the satisfaction of another condition to other interested threads. A condition variable is always used in conjunction with a mutex. These variables also have a special characteristic in that you can do a timed block on the variable. This makes it easy to use a condition variable to manage a simple event loop. We show an example of this when we talk about timers in Chapter 20.

We can easily change our command handler example to use a condition variable to coordinate access to the device repository instead of using a mutex for protection. We start by modifying the repository so that it can record which task currently owns it:

```
class HA_Device_Repository
{
public:
  HA_Device_Repository() : owner_(0)
  { }

  int is_free (void)
    { return (this->owner_ == 0); }

  int is_owner (ACE_Task_Base* tb)
    { return (this->owner_ == tb); }

  ACE_Task_Base *get_owner (void)
    { return this->owner_; }

  void set_owner (ACE_Task_Base *owner)
    { this->owner_ = owner; }

  int update_device (int device_id);

private:
  ACE_Task_Base * owner_;
};
```

Next, we modify the command handler such that it uses a condition variable (waitCond_) and mutex (mutex_), to coordinate access to the repository. Both the condition variable and the mutex are created on the main() thread stack and are passed to the command handlers during construction.

To use a condition variable, you must first acquire the mutex, check whether the system is in the required state—the

12.4 Summary

The basic high-level ACE threading components provide an easy-to-use object interface to multithreaded programming. In this chapter, we explained how to use the ACE_Task objects to create new threads of control, how to use ACE_Mutex and ACE_Guard to ensure consistency, and how to use ACE_Condition and the ACE_Message_Queue, ACE_Message_Block, and ACE_Task message-passing constructs to incorporate communication between threads.

Chapter 13. Thread Management

Previously, we introduced the ACE_Task family and thread creation. This chapter goes into the details of how you can create various types of threads in varying states in accordance with your requirements. We also discuss how running threads can be managed: suspending and resuming threads, creating and waiting on threads that are in "thread groups," canceling running threads, and incorporating start-up and exit hooks for the threads that you create.

13.1 Types of Threads

In the previous chapter, we glossed over the fact that you can create threads with special attributes by passing various flags to the ACE_Task_Base::activate() method. These attributes define, among other things, how the new thread will be created, scheduled, and destroyed. Table 13.1 lists all the possible thread creation flags that control the assignment of various attributes to the new thread.

Because the ACE threads library tries to hide a wide array of operating systems, you may find that some of these flags to do not work exactly as you may expect them to work on your particular operating system. For example, if your particular OS does not support round-robin scheduling, don't expect ACE to do magic to make that work for you. However, on most major OSs, you will find that everything works as expected.

Table 13.1. Thread Creation Flags

Flag	Description
THR_CANCEL_DISABLE	Do not allow this thread to be canceled.
THR_CANCEL_ENABLE	Allow this thread to be canceled.
THR_CANCEL_DEFERRED	Allow for deferred cancellation only.
THR_BOUND	Create a thread that is bound to a kernel-schedulable entity.
THR_NEW_LWP	Create a kernel-level thread.
THR_DETACHED	Create a detached thread.
THR_SUSPENDED	Create the thread but keep the new thread in suspended state.
THR_DAEMON	Create a daemon thread.
THR_JOINABLE	Allow the new thread to be joined with.
THR_SCHED_FIFO	Schedule the new thread using the FIFO policy, if available.
THR_SCHED_RR	Schedule the new thread using a round-robin scheme, if available.
THR_SCHED_DEFAULT	Use whatever default scheduling scheme is available on the operating system.

You can OR the flags together and pass them as the first argument to the activate() method. For example, the following call would cause the thread to be created as a kernel-schedulable thread, with a default scheduling scheme, and start in the suspended state, and it would be possible to join with this thread:

13.2 Priorities and Scheduling Classes

One important reason to use threads is to have various parts of your application run at different priorities. Many applications require such characteristics. In our home automation example, we can incorporate priorities to create two HA_CommandHandler instances that run at differing priorities. The high-priority command handler receives critical commands that must be executed before any of the noncritical commands are; for example, changing the temperature on the oven could be more critical than recording your favorite TV program. If you were using strict priority-based scheduling, this would mean that the low-priority handler would run only if the application was not busy processing critical commands in the high-priority handler thread.

Each OS defines priorities in a proprietary fashion. For example, Solaris defines priorities to range from 0–127, with 127 being the highest priority. Windows specifies a range of 0–15, with 15 being the highest, and VxWorks has 0–255, with 0 being the highest priority.

Most general-purpose operating systems offer a single time-shared scheduling class. Time-shared schedulers attempt to enforce fairness by allowing the highest priority thread to execute for a finite period of time called a time slice. If a higher-priority thread becomes runnable before the end of the executing thread's time slice, the lower-priority thread is preempted and the higher-priority thread begins its time slice.

Some operating systems offer a real-time scheduling class. Threads in the real-time scheduling class are always at a higher priority than any thread in the time-shared class, if the operating system supports a time-shared scheduling class. There are two scheduling policies offered in the real-time scheduling class.

1. Round-robin, where a time quantum specifies the maximum time a thread can run before it's preempted by another real-time thread with the same priority.
2. First-in, first-out (FIFO), where the highest-priority thread can run for as long as it chooses, until it voluntarily yields control or is preempted by a real-time thread with an even higher priority.

Table 13.2. Thread Priority Macros

Macro	Meaning
ACE_THR_PRI_OTHER_MIN	Minimum priority for the time-shared scheduling class
ACE_THR_PRI_OTHER_DEF	Default priority for the time-shared scheduling class
ACE_THR_PRI_OTHER_MAX	Maximum priority for the time-shared scheduling class
ACE_THR_PRI_RR_MIN	Minimum priority for the real-time scheduling class with the round-robin policy
ACE_THR_PRI_RR_DEF	Default priority for the real-time scheduling class with the round-robin policy
ACE_THR_PRI_RR_MAX	Maximum priority for the real-time scheduling class with the round-robin policy
ACE_THR_PRI_FIFO_MIN	Minimum priority for the real-time scheduling class with the FIFO policy

13.3 Thread Pools

So far, we have been creating an ACE_Task subclass instance and then activating it, causing a single thread of control to be created and start from the svc() method. However, the activate() method allows multiple threads to be started at the same time, all starting at the same svc() entry point. When creating a group of threads in a task, ACE internally assigns to all the threads in a task a group identifier, available through the grp_id() accessor, that can be used for subsequent management operations on the group. All the created threads share the same ACE_Task object but have their own separate stacks.

This is a handy way to create a simple pool of worker threads that share the same message queue. All the threads wait on the queue for a message to arrive. Once a message arrives, the queue unblocks a single thread that gets the message and continues to process it. Of course, when doing something like this, you must either ensure that the worker threads do not share any data with one another or use property safety constructs in the right places. The advantage is that you can improve message-processing throughput on a multiprocessor or may be able to improve latency on a uniprocessor, especially if the message processing is not compute bound. You must be careful that you don't make things worse by having all your worker threads lock on a single resource; the locking overhead may be high enough to obviate any performance advantage you gain because of the threads.

In the next example, we spawn multiple threads inside a single ACE_Task object. These threads display their identifiers by using the ACE_Log_Msg %t format specifier and then block, waiting for messages on the underlying message queue:

```
class HA_CommandHandler : public ACE_Task<ACE_MT_SYNCH>
{
public:
  virtual int svc (void)
  {
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%t) starting up \n")));
    ACE_Message_Block *mb;
    if (this->getq (mb) == -1)
      return -1;
    // ... do something with the message.
    return 0;
  }
};
```

We specify as the second argument to the activate() method the number of threads to be created: in this case, 4. This will cause four new threads to start at the HA_CommandHandler::svc() entry point:

```
int ACE_TMAIN (int, ACE_TCHAR *[])
{
  HA_CommandHandler handler;

  // Create 4 threads.
  handler.activate (THR_NEW_LWP | THR_JOINABLE, 4);
  handler.wait ();
  return 0;
}
```

The output shows us that four distinct threads start up and then block inside the single svc() method of the HA_CommandHandler subclass:

```
(2996) starting up
```

13.4 Thread Management Using ACE_Thread_Manager

So far, we have been using the ACE_Task family of classes to help us create and manage tasks. We used the wait() management call to wait for handler threads to complete so that the main thread can exit after the handler threads. In addition, we saw the suspend() and resume() methods, available to suspend and resume threads that are run within tasks. In most cases, you will find the ACE_Task interface to be rich enough to provide all the management functionality you need. However, ACE also provides a behind-the-scenes class, ACE_Thread_Manager, to help manage all ACE-created tasks, such as providing operations that suspend, resume, and cancel all threads in all tasks. The manager provides a rich interface that includes group management, creation, state retrieval, cancellation, thread start-up and exit hooks, and so on. The thread manager is also tightly integrated with the ACE Streams framework, which we cover in Chapter 18.

ACE allows a programmer to register an unlimited number of exit functions, or exit functors, that will be automatically called when a thread exits. These exit handlers can be used to do last-second cleanup or to inform other threads of the imminent termination of a thread. One of the nice things about the exit handlers is that they are always called, even if the thread exits forcefully with a call to the low-level ACE_Thread::exit() method, which causes a thread to exit immediately.

To set up an exit functor, you create a subclass of ACE_At_Thread_Exit, implement the apply() method, and then register an instance of this class with the ACE_Thread_Manager. In this case, we want the ExitHandler functor to be called when our home automation command handler thread shuts down. The exit functor then sends out commands to all the devices on the home network, informing them that the network has closed down:

```
#include "ace/Task.h"
#include "ace/Log_Msg.h"

class ExitHandler : public ACE_At_Thread_Exit
{
public:
  virtual void apply (void)
  {
    ACE_DEBUG ((LM_INFO, ACE_TEXT ("(%t) is exiting \n")));

    // Shut down all devices.
  }
};
```

The registration of the exit functor, with the thread manager, must occur within the context of the thread whose exit functor this is. In this case, we want the functor to be applied when the home automation handler thread exits, so we make sure that the exit functor is registered with the thread manager as soon as the command handler thread starts. We get a pointer to the thread manager by calling the task's thr_mgr() accessor and then register the exit handler, using the at_exit() method:

```
class HA_CommandHandler : public ACE_Task_Base
{
public:
  HA_CommandHandler(ExitHandler& eh) : eh_(eh)
  { }
  virtual int svc (void)
  {
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%t) starting up \n")));

    this->thr_mgr ()->at_exit (eh_);

    // Do something.
```

13.5 Signals

Chapter 11 is devoted to the topic of signals and their use within the ACE toolkit. This section explains how signals are done differently in a multithreaded application and a single-threaded application, respectively.

First, each thread has its own private signal mask, which by default is inherited during creation. This means that all the threads that are created by the main thread inherit its signal mask.

On the other hand, the signal handlers are global to the process. ACE maintains this invariant when you use it to handle a signal; that is, you can have only a single signal handler for a particular signal type.

13.5.1 Signaling Threads

You can explicitly send targeted synchronous signals to threads by using the thread manager. ACE_Thread_Manager allows you to send a signal to all threads, to a thread group or task, or to a particular thread. As ACE_Task inherits from ACE_Event_Handler, the best place to handle the signal is to use the built-in handle_signal() method and use one of the previously discussed signal dispatchers. In the next example, we use the ACE_Sig_Handler dispatcher and the handle_signal() method to illustrate sending signals to all threads in a thread group. We start by creating a routine message-processing task that implements its handle_signal() event-handling method:

```
class SignalableTask : public ACE_Task<ACE_MT_SYNCH>
{
public:
  virtual int handle_signal (int signum,
                             siginfo_t *   = 0,
                             ucontext_t * = 0)
  {
    if (signum == SIGUSR1)
      {

        ACE_DEBUG ((LM_DEBUG,
                    ACE_TEXT ("(%t) received a %S signal\n"),
                    signum));
        handle_alert ();
      }
    return 0;
  }

  virtual int svc (void)
  {
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%t) Starting thread\n")));

    while (1)
      {
        ACE_Message_Block* mb;
        ACE_Time_Value tv (0, 1000);
        tv += ACE_OS::time (0);
        int result = this->getq (mb, &tv);
        if (result == -1 && errno == EWOULDBLOCK)
          continue;
        else
          process_message (mb);
      }

      return 0;
    }

  void handle_alert ();
  void process_message (ACE_Message_Block *mb);
};
```

13.6 Thread Start-Up Hooks

ACE provides a global thread start-up hook that can be used to intercept the call to the thread start-up function. This allows a programmer to perform any kind of initialization that is globally applicable to all the threads being used in your application. This is an especially good spot to add thread-specific data to a thread (see Section 14.3).

To set up a start-up hook, you must first subclass ACE_Thread_Hook and implement the virtual start() method. This method is passed a pointer to the thread's entry function, and a void* argument that must be passed to this function. In most cases, you will execute your special start-up code and then call the passed entry point with the supplied argument. In our case, we add a special security context to thread-specific storage. We use this context to record the current client that is using our command handler thread, which helps us to determine whether the client has permission to execute specified commands. After setting up the security context, we execute the specified thread function:

```
class HA_ThreadHook : public ACE_Thread_Hook
{
public:
  virtual ACE_THR_FUNC_RETURN start (ACE_THR_FUNC func, void* arg)
  {
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT("(%t) New Thread Spawned\n")));

    // Create the context on the thread's own stack.
    ACE_TSS<SecurityContext> secCtx;
    // Special initialization.
    add_sec_context_thr (secCtx);

    return (*func) (arg);
  }

  void add_sec_context_thr (ACE_TSS<SecurityContext> &secCtx);
};
```

After creating your new start-up hook, you need to set it as the new start-up hook. To do this, you simply call the ACE_Thread_Hook::thread_hook() static method, passing in an instance of the new hook:

```
int ACE_TMAIN (int, ACE_TCHAR *[])
{
  HA_ThreadHook hook;
  ACE_Thread_Hook::thread_hook (&hook);

  HA_CommandHandler handler;
  handler.activate ();
  handler.wait();
  return 0;
}
```

13.7 Cancellation

Except for cooperative cancellation, cancellation is best avoided unless you can justify a real need for it. With that said, let's delve into what cancellation is and why we consider it to be a feature best shunned.

Cancellation is a way by which you can zap current running threads into oblivion. No thread exit handlers will be called; nor will thread-specific storage be released. Your thread will simply cease to exist. In certain instances, cancellation may be a necessary evil: to exit a long-running compute-bound thread or to terminate a thread that is blocked on a blocking system call, such as I/O. In most cases, cancellation will make sense when the application is terminating. In other cases, it is difficult to ensure that proper thread termination occurs with cancellation. Cooperative cancellation, which we discuss in Section 13.7.1, is the only cancellation mode that does not suffer from these ungraceful-exit problems.

Cancellation has several modes:

- Deferred cancelability. When a thread is in this mode, all cancels are deferred until the thread in question reaches the next cancellation point. Cancellation points are well-defined points in the code at which either the thread has blocked, as on an I/O call, or an explicit cancellation point is coded by using the ACE_Thread_Manager::testcancel() method. This mode is the default for applications built with ACE.
- Cooperative cancellation. In this mode, threads are not really canceled but instead have their state marked as canceled within the instance of the ACE_Thread_Manager that spawned it. You can call ACE_Thread_Manager::testcancel() to determine whether the thread is in the canceled state; if it is, you can choose to exit the thread. This mode also gets around most of the nasty side effects that come with regular cancellation. If you wish to build portable applications, it is best to stick with this cancellation mode.
- Asynchronous cancelability. When a thread is in this mode, cancels can be processed at any instant. Threads run in this mode can be difficult to manage. You can change the cancel state of any thread from enabled to disabled to ensure that the threads are not canceled when executing critical sections of code. You can also use cleanup handlers, which are called when a thread is canceled, to ensure that program invariants are maintained during cancellation. ACE does not support POSIX cleanup handler features, as many operating systems that ACE runs on do not support them. If you are running on a POSIX platform, look for these features if you really need to use cancellation.
- Disabled. Cancellation can be totally disabled for a thread by using the ACE_Thread::disablecancel() call.

13.7.1 Cooperative Cancellation

Let's first look at an example of cooperative cancellation. We start by creating a task that first makes sure that is has not been canceled and then does a timed wait of 1 ms for messages to arrive on its queue. If no messages arrive and it times out, the task checks whether it was canceled again; if it has, the thread exits explicitly. Using this scheme, the thread will notice that it has been canceled within 1 ms of cancellation, assuming that message processing is a short process. The testcancel() method of the thread manager requires the thread identifier of the thread whose cancel state is being checked. We can easily obtain this by using the ACE_Thread_Manager::thr_self() method or the low-level ACE_Thread::self() method:

```
class CanceledTask : public ACE_Task<ACE_MT_SYNCH>
{
public:

  virtual int svc (void)
```

13.8 Summary

In this chapter, we explained how to create various types of threads by using the ACE_Task::activate() method with varying parameters. We also used the ACE_Thread_Manager to help us better manage the threads that we create, going over signal handling, exit handlers, and thread cancellation using the manager. Finally, we illustrated how you can use ACE_Thread_Hook to better control thread start-up in your applications.

Chapter 14. Thread Safety and Synchronization

We covered the basics of both thread safety and synchronization in Chapter 12, explaining how you can use mutexes and guards to ensure the safety of your global data and how condition variables can help two threads communicate events between each other synchronizing their actions. This chapter extends that discussion. In the safety arena, you will get to see readers/writer locks, the atomic operation wrapper, using tokens for fine-grain locking on data structures, and an introduction to recursive mutexes. We also introduce semaphores and barriers as new synchronization mechanisms.

14.1 Protection Primitives

To ensure consistency, multithreaded programs must use protection primitives around shared data. Some examples of protection were given in Chapter 12, where we introduced mutexes and guards. Here, we consider the remaining ACE-provided protection primitives.

Table 14.1 lists the primitives that are available in ACE. All these primitives have a common interface but do not share type relationships and therefore cannot be polymorphically substituted at runtime.

Table 14.1. Protection Primitives in ACE

Primitive	Description
ACE_Mutex	Wrapper class around the mutual-exclusion mechanism and used to provide a simple, efficient mechanism to serialize access to a shared resource. Similar in functionality to a binary sempahore. Can be used for mutual exclusion among both threads and processes.
ACE_Thread_Mutex	Can be used in place of ACE_Mutex and is specific for synchronization of threads.
ACE_Recursive_Thread_Mutex	Mutex that is recursive, that is, can be acquired multiple times by the same thread. Note that to work correctly, it must be released as many times as it was acquired.
ACE_RW_Mutex	Wrapper class that encapsulates readers/writer locks. These locks are acquired differently for reading and writing, thus enabling multiple readers to read while no one is writing. These locks are nonrecursive.
ACE_RW_Thread_Mutex	Can be used in place of ACE_RW_Mutex and is specific for synchronization of threads.
ACE_Token	Tokens are the richest and heaviest locking primitive available in ACE. The locks are recursive and allow for read and write locking. Also, strict FIFO ordering of acquisition is enforced; all threads that call acquire() enter a FIFO queue and acquire the lock in order.
ACE_Atomic_Op	Template wrapper that allows for safe arithmetic operations on the specified type.
ACE_Guard	Template-based guard class that takes the lock type as a template parameter.
ACE_Read_Guard	Guard that uses acquire_read() on a read/write guard during construction.
ACE_Write_Guard	Guard that uses acquire_write() on a read/write guard

14.2 Thread Synchronization

Synchronization is a process by which you can control the order in which threads execute to complete a task. We have already seen several examples of this; we often try to order our code execution such that the main thread doesn't exit before the other threads are done. In fact, the protection code in the previous section also has a similar flavor to it; we try to control the order of the threads as they access shared resources. However, we believe that synchronization and protection are sufficiently different to warrant separate sections.

In many instances, you want to make sure that one thread executes before the other or some other specific ordering is needed. For example, you might want a background thread to execute if a certain condition is true, or you may want one thread to signal another thread that it is time for it to go. ACE provides a number of primitives that are specifically designed for these purposes. We list them in Table 14.2.

Table 14.2. ACE Synchronization Primitives

Primitive	Description
ACE_Condition	A condition variable; allows signaling other threads to indicate event occurrence
ACE_Semaphore	A counting semaphore; can be used as a signaling mechanism and also for synchronization purposes
ACE_Barrier	Blocks all threads of execution until they all reach the barrier line, after which all threads continue
ACE_Event	A simple synchronization object that is used to signal events to other threads

14.2.1 Using Semaphores

A semaphore is a non-negative integer count that is used to coordinate access among multiple resources. Acquiring a semaphore causes the count to decrement, and releasing a semaphore causes the count to increment. If the count reaches 0—no resources left—and a thread tries to acquire the semaphore, it will block until the semaphore count is incremented to a value that is greater than 0. This happens when another thread releases on the semaphore. A semaphore count will never be negative. When using a semaphore, you initialize the semaphore count to a non-negative value that represents the number of resources you have.

Mutexes assume that the thread that acquires the mutex will also be the one that releases it. Semaphores, on the other hand, are usually acquired by one thread but released by another. It is this unique characteristic that allows one to use semaphores so effectively.

Semaphores are probably the most versatile synchronization primitive provided by ACE. In fact, a binary semaphore can be used in place of a mutex, and a regular semaphore can be used in most places where a condition variable is being used.

Once again, we implement a solution to the producer/consumer problem, this time using counting semaphores. We use the built-in ACE_Message_Queue held by the consumer as the shared data buffer between the producer and consumer. Using semaphores, we control the maximum number of messages the message queue can have at any particular instant. In other words, we implement a high-water-mark mechanism; the message queue already has this functionality built in; we essentially reproduce it in an inefficient way for the sake of the example. To make things interesting, the consumer task has multiple threads.

14.3 Thread-Specific Storage

When you create a thread, all you create is a thread stack, a signal mask, and a task control block for the new thread. The thread carries no other state information on creation. Nonetheless, it is convenient to be able to store state information that is specific to a thread: thread-specific storage (TSS),or thread local storage.

One classic example of the use of TSS is with the application global errno. The global errno is used to indicate the most recent error that has occurred within an application. When multithreaded programs first appeared on UNIX platforms, errno posed a problem. Different errors can occur within different threads of the application, so if errno is global, which error would it hold? Putting errno in thread-specific storage solves this problem. Logically, the variable is global but is kept in TSS.

Thread-specific storage should be used for all objects that you consider to be part and parcel of a thread. ACE uses TSS internally to store a thread-specific output stream for the purposes of logging. By doing this, no locks are required on any of the logging streams, as only the owning thread can access the TSS stream. This helps keeps the code efficient and lightweight.

The ACE_Thread class provides access to the low-level OS TSS methods, but in most cases, you can avoid these tedious APIs by using the ACE_TSS class template. This class is very simple to use. You simply pass it the data you want to be stored in TSS as its template parameter and then use the operator->() method to access the data when you need it. The operator->() creates and stores the data in TSS when called the first time. The destructor for ACE_TSS ensures that the TSS data is properly removed and destroyed.

One useful application of TSS is the addition of a context object that can hold information specific to the current client that is using the thread. For example, if you were to use a thread-per-connection model, you could keep connection-specific information within TSS, where you can get to it easily and efficiently. At least one database server we know keeps transactional context within TSS for each connection it hands out.

We can use the same idea to improve our home automation command handler. The handler keeps a generic-client context object in TSS. The context can then hold any arbitrary, named tidbit of information, using an internal attribute map:

```
// Client-specific context information.
class ClientContext
{
public:
  void *get_attribute (const char *name);
  void set_attribute (const char *name, void *value);

private:
  Map attributeMap_;
};
```

To store this object in TSS, all you need to do is wrap the object within the ACE_TSS template. The first access causes an instance of the type ClassContext to be created on the heap and stored within thread-specific storage. For illustrative purposes, we store the current thread ID in the context, only to obtain it at a later point and display it:

```
class HA_CommandHandler : public ACE_Task<ACE_MT_SYNCH>
{
public:
  virtual int svc (void)
  {
    ACE_thread_t tid = this->thr_mgr ()->thr_self ();
    // Set our identifier in TSS.
    this->tss_ctx_->set_attribute ("thread_id", &tid);
```

14.4 Summary

In this chapter, we covered some of the more advanced synchronization and protection primitives that are available from the ACE toolkit. We discussed recursive mutex locks, using the ACE_Recursive_Mutex and ACE_Recursive_Thread_Mutex classes, readers/writer locks with ACE_RW_Mutex and ACE_RW_Thread_Mutex, introduced the Token framework with ACE_Local_Mutex, and talked about enforcing atomic arithmetic operations on multiprocessors with the ACE_Atomic_Op wrapper as new protection primitives. We also introduced several new synchronization primitives, including counting semaphores with ACE_Semaphore and barriers with ACE_Barrier. Finally, we introduced thread-specific storage as an efficient means to get around protection with thread-specific data and the ACE_TSS wrapper.

Chapter 15. Active Objects

Threads need to cooperate with one another. Various illustrations of thread cooperation were given in previous chapters, including cooperation using condition variables, semaphores, and message queues.

The Active Object pattern has been specifically designed to provide an object-based solution to cooperative processing between threads. The pattern is based on the requirement that two Active Objects should communicate through what look like regular method calls, but these methods will be executed in the context of the receiver, not that of the invoker; hence the name Active Object. In other words, every object that is active has a private thread of control that is used to execute any methods that a client invokes on the object.

This comes in handy when you have an object whose method calls take a long time to complete: for example, a proxy to a remote service. You can make this proxy an Active Object and free up the main thread of control to do other things, such as interact with the user or run a GUI event loop.

This chapter examines the Active Object pattern and illustrates how you can make this pattern work in your applications. We also introduce several new components used to implement the pattern; these components will also be useful when we talk about thread pools (Chapter 16).

15.1 The Pattern

Figure 15.1 illustrates the behavior of an active Logger object. The main thread starts by creating a object and then activating it. The activate() call causes the Logger object to spawn its own private thread of control. Next, the main thread invokes the log() method. Here is where the Active Object magic steps in. The log() method is executed in the context of the Logger-spawned thread, not in the context of the main thread. Therefore, when it returns in the main thread, there is no guarantee that the log() method has completed execution or, for that matter, even started execution. In Active Object parlance, the Logger object is active.

Figure 15.1. Asynchronous log() method

Main Thread

```
//the main thread
int main()
{
ActiveLogger logger;
logger.activate();
//create an active
//logger object

logger.log(data);
//this occurs in the
//context of the
//logger thread, that
//returns immediately in
//main i.e., this is
 //an asynchronous call. ..
}
```

Logger Thread

```
//the logger thread..
Logger::log(const char* data)
{
 ..

 //logs the data in a separate
 //thread of control. Context
 //switch occurs here.


 ..
}
```

15.1.1 Pattern Participants

The Active Object pattern is a combination of the Command pattern and the Proxy pattern. The participants in the pattern, illustrated in Figure 15.2, are as follows:

- Receiver Implementation. The Receiver Implementation is independent of threading details.
- Scheduler. The Scheduler class has a reference to an Activation Queue on which the private thread of control used by the Active Object remains blocked until a new method request arrives. When a request arrives, the private thread wakes up, dequeues the request, and executes it.
- Proxy. Proxy to the Receiver object that will be used by the client to invoke requests. This class converts method invocations to method requests and inserts them on the activation queue. The Proxy object keeps a reference to the real Receiver Implementation object (Proxy pattern).
- Method Request. An object that encapsulates a method call in the form of an object (Command pattern). Each method on the Receiver interface will have one method request class associated with it. To invoke the method on the ReceiverImpl, the method request will need to have a reference to the ReceiverImpl.
- Activation Queue. A priority queue that is held by the Scheduler. All method requests are placed on this queue by the Proxy object and are picked up by the private thread of the Active Object.
-

15.2 Using the Pattern

In this example, we have a home automation controller on our home network. This controller is capable of providing status information about itself to any clients on the network. The client devices on the network use an agent object to talk to the controller. The agent object encapsulates the remoting protocol and other network details, providing clients with an easy-to-use representation of the controller: once again, the Proxy pattern:

```
class HA_ControllerAgent
{
  // Proxy to the HA_Controller that is on the network.
public:
  HA_ControllerAgent ()
  {
    ACE_TRACE
      (ACE_TEXT ("HA_ControllerAgent::HA_ControllerAgent"));
    status_result_ = 1;
  }

  int status_update (void)
  {
    ACE_TRACE (ACE_TEXT ("HA_ControllerAgent::status_update"));
    ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT ("Obtaining a status_update in %t ")
                ACE_TEXT ("thread of control\n")));
    // Simulate time to send message and get status.
    ACE_OS::sleep (2);
    return next_result_id ();
  }
private:
  int next_result_id (void)
  {
    ACE_TRACE (ACE_TEXT ("HA_ControllerAgent::next_cmd_id"));
    return status_result_++;
  }

  int status_result_;
};
```

Let's assume that it takes a while for the agent to talk to the controller, obtain the status, and get back. This seems like an ideal object to turn into an Active Object (Figure 15.4). Note that we will not change any code within the agent object, and it will therefore contain no details about how threads are going to be used to dispatch its methods:

```
class StatusUpdate : public ACE_Method_Request
{
public:
  StatusUpdate (HA_ControllerAgent& controller,
                ACE_Future<int>& returnVal)
    : controller_(controller), returnVal_(returnVal)
  {
    ACE_TRACE (ACE_TEXT ("StatusUpdate::StatusUpdate"));
  }

  virtual int call (void)
  {
    ACE_TRACE (ACE_TEXT ("StatusUpdate::call"));

    // status_update with the controller.
    this->returnVal_.set (this->controller_.status_update ());
    return 0;
```

15.3 Summary

In this chapter, we described the Active Object pattern and all its component participants. This pattern allows you to build an object whose methods are run asynchronously. This comes in handy when you have long-running I/O calls that you wish to run in a separate thread of control. We also illustrated how you can obtain the result of asynchronous operations; that is, you can either block on an ACE_Future object, or you can attach an Observer to the Future and receive a callback on method completion. In the next chapter, we reuse the Active Object pattern and the ACE Task framework to build thread pools.

Chapter 16. Thread Pools

Most network servers are designed to handle multiple client requests simultaneously. As we have seen, there are multiple ways to achieve this, including using reactive event handling, multiple processes, or multiple threads in our servers. When building a multithreaded server, many design options are available, including

- Spawning a new thread for each request
- Spawning a new thread for each connection/session
- Prespawning a managed pool of threads, or creating a thread pool

In this chapter, we explore thread pools in some detail. We start by defining what we mean by a thread pool and what advantages it provides over some of the other approaches to building multithreaded servers. We then go into some detail about various types of thread pools and their performance characteristics, illustrating a few types with example pools that are built using the ACE components. Finally, we look at two ACE_Reactor implementations that can use thread pools for concurrent event handling.

16.1 Understanding Thread Pools

In the thread pool model, we do not create a new thread for each session or request and then destroy it; rather, we prespawn all the threads we are going to use and keep them around in a pool. This bounds the cost of the resources that the server is going to use, so you as a developer always know how many threads are running in the server.

Contrast this with the thread-per-request model, in which a new thread is created for each request. If it receives a large spurt of requests in a short period of time, the server will spawn a large number of threads to handle the load. This will cause degradation in service for all requests and may cause resource allocation failures as the load increases. In the thread pool model, when a request arrives, a thread is chosen from the queue to handle the request; if there are no threads in the pool when the request arrives, it is enqueued until one of the worker threads returns to the pool.

The thread pool model has several variants, each with different performance characteristics:

- Half-sync/half-async model. In this model, a single listener thread asynchronously receives requests and buffers them in a queue. A separate set of worker threads synchronously processes the requests.
- Leader/followers model. In this model, a single thread is the leader and the rest are followers in the thread pool. When a request arrives, the leader picks it up, selects one of the followers as the new leader, and then continues to process the request. Thus in this case, the thread that receives the request is also the one that handles it.

16.2 Half-Sync/Half-Async Model

This model breaks the thread pool into three separate layers:

1. The asynchronous layer, which receives the asynchronous requests
2. The queueing layer, which buffers the requests
3. The synchronous layer, which contains several threads of control blocked on the queueing layer

When the queueing layer indicates that there is new data, the synchronous layer handles the request synchronously in a separate thread of control. This model has the following advantages.

- The queuing layer can help handle bursty clients; if there are no threads to handle a request, they are simply buffered in the queueing layer.
- The synchronous layer is simple and independent of any asynchronous processing details. Each synchronous thread blocks on the queueing layer, waiting for a request.

  This model also has disadvantages.
- Because a thread switch occurs at the queueing layer, we must incur synchronization and context switch overhead. Furthermore, on a multiprocessor machine, we may experience data copying and cache coherency overhead.
- We cannot keep any request information on the stack or in thread-specific storage, as the request is processed in a different worker thread.

Let's go through a simple example implementation of this model. In this example, the asynchronous layer is implemented as an ACE_Task subclass, Manager, which receives requests on its underlying message queue. Each worker thread is implemented by the Worker class, which is also an ACE_Task derivative. When it receives a request, the Manager picks a Worker object from the worker thread pool and enqueues the request on the Worker object's underlying ACE_Message_Queue. This queue acts as the queueing layer between the asynchronous Manager and the synchronous Worker class.

First, let's look at the Manager task:

```
class Manager: public ACE_Task<ACE_MT_SYNCH>, IManager
{
public:
  enum {POOL_SIZE = 5, MAX_TIMEOUT = 5};

  Manager ()
    : shutdown_(0), workers_lock_(), workers_cond_(workers_lock_)
  {
    ACE_TRACE (ACE_TEXT ("Manager::Manager"));
  }

  int svc (void)
```

16.3 Leader/Followers Model

In this model, a single group of threads is used to wait for new requests and to handle the request. One thread is chosen as the leader and blocks on the incoming request source. When a request arrives, the leader thread first obtains the request, promotes one of the followers to leader status, and goes on to process the request it had received. The new leader waits on the request source for any new requests while the old leader processes the request that was just received. Once the old leader is finished, it returns to the end of thread pool as a follower thread.

The leader/followers model has the advantage that performance improves, as there is no context switch between threads. This also allows for keeping request data on the stack or in thread-specific storage.

This model also has some disadvantages.

- It is not easy to handle bursty clients, as there might not be an explicit queueing layer.
- This model is more complex to implement.

The following simple example implements the leader/followers thread pool model. In this case, a single ACE_Task (the LF_ThreadPool class) encapsulates all the threads in the thread pool. Remember, there can be only one leader at any given time; the thread ID of the current leader of the pool is maintained in current_leader_. As in our first example, the LF_ThreadPool is given new work as an ACE_Message_Block on its message queue; that is, a unit of work is a message:

```
class LF_ThreadPool : public ACE_Task<ACE_MT_SYNCH>
{
public:
  LF_ThreadPool () : shutdown_(0), current_leader_(0)
  {
    ACE_TRACE (ACE_TEXT ("LF_ThreadPool::TP"));
  }

  virtual int svc (void);

  void shut_down (void)
  {
    shutdown_ = 1;
  }

private:
  int become_leader (void);

  Follower *make_follower (void);

  int elect_new_leader (void);

  int leader_active (void)
  {
    ACE_TRACE (ACE_TEXT ("LF_ThreadPool::leader_active"));
    return this->current_leader_ != 0;
  }

  void leader_active (ACE_thread_t leader)
  {
    ACE_TRACE (ACE_TEXT ("LF_ThreadPool::leader_active"));
    this->current_leader_ = leader;
  }

  void process_message (ACE_Message_Block *mb);
```

16.4 Thread Pools and the Reactor

The ACE_Reactor has several implementations that you have seen previously in this book. Some of these implementations allow only a single owner thread to run the event loop and dispatch event handlers; others, however, allow you to have multiple threads run the event loop at once. The underlying reactor implementation builds a thread pool using these client-supplied threads and uses them to wait for and then dispatch events.

The two implementations that provide for this functionality are

- ACE_TP_Reactor
- ACE_WFMO_Reactor

16.4.1 ACE_TP_Reactor

The ACE_TP_Reactor uses the leader/followers model for its underlying thread pool. Thus, when several threads enter the event loop method, all but one become followers and wait in a queue. The leader thread waits for events. When an event occurs, the leader selects a new leader and dispatches the event handlers that had been signaled. Before selecting a new leader or dispatching events, the leader first suspends the handlers that it is about to call back. By suspending the handlers, it makes sure that they can never be invoked by two threads at the same time, thereby making our lives easier, as we don't have to deal with protection issues arising from multiple calls into our event handlers. Once it finishes the dispatch, the leader resumes the handler in the reactor, making it available for dispatch once again.

The following example is a rehash from an ACE test that illustrates the ACE_TP_Reactor:

```
int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_TP_Reactor sr;
  ACE_Reactor new_reactor (&sr);
  ACE_Reactor::instance (&new_reactor);

  ACCEPTOR acceptor;
  ACE_INET_Addr accept_addr (rendezvous);

  if (acceptor.open (accept_addr) == -1)
    ACE_ERROR_RETURN ((LM_ERROR,
                       ACE_TEXT ("%p\n"),
                       ACE_TEXT ("open")),
                      1);

  ACE_DEBUG ((LM_DEBUG,
              ACE_TEXT ("(%t) Spawning %d server threads...\n"),
              svr_thrno));

  ServerTP serverTP;
  serverTP.activate (THR_NEW_LWP | THR_JOINABLE, svr_thrno);

  Client client;
  client.activate ();

  ACE_Thread_Manager::instance ()->wait ();

  return 0;
}
```

16.5 Summary

In this chapter, we looked at thread pools. We first talked about a few design models for thread pools, including the leader/followers and half-sync/half-async models. We then implemented each of these, using the ACE_Task and synchronization components. We also saw how to use ACE_Activation_Queue, ACE_Future, and ACE_Future_Observer to implement pools that can execute arbitrary ACE_Method_Requests and then allow for clients to obtain results, using Futures. Finally, we talked about the two reactor implementations that support thread pools: the ACE_TP_Reactor and the ACE_WFMO_Reactor.

Part IV: Advanced ACE

Chapter 17. Shared Memory

Chapter 18. ACE Streams Framework

Chapter 19. ACE Service Configurator Framework

Chapter 20. Timers

Chapter 21. ACE Naming Service

Chapter 17. Shared Memory

Modern operating systems enforce address space protection between processes. This means that the OS will not allow two distinct processes to write to each other's address space. Although this is what is needed in most cases, sometimes you may want your processes to share certain parts of their address space. Shared memory primitives allow you to do just this. In fact, when used correctly, shared memory can prove to be the fastest interprocess communication mechanism between collocated processes, especially if large amounts of data need to be shared.

Shared memory is not used only as an IPC mechanism. Shared memory primitives also allow you to work with files by mapping them into memory. This allows you to perform direct memory-based operations on files instead of using file I/O operations. This comes in handy when you want to provide for a simple way to persist a data structure. (In fact, ACE_Configuration can use this technique to provide persistence for your configuration.)

ACE provides several tools to assist in your shared memory adventures. For sharing memory between applications, ACE provides a set of allocators that allow you to allocate memory that is shared. In fact, you can use a shared memory allocator with the containers discussed in Chapter 5 to create containers in shared memory. For example, you could create a hash map in shared memory and share it across processes.

ACE also provides low-level wrappers around the OS shared memory primitives. These wrappers can be used to perform memory-mapped file operations.

17.1 ACE_Malloc and ACE_Allocator

When we talked about containers in Chapter 5, we saw how we could supply special-purpose allocators to the containers. These allocators were of type ACE_Allocator and were usually passed in during container construction. The container then used the allocator to manage any memory it needed to allocate and deallocate.

ACE also includes another family of allocators, based on the ACE_Malloc class template. Unlike the ACE_Allocator family, which is based on polymorphism, the ACE_Malloc family of classes are template based. The ACE_Malloc template takes two major parameters: a lock type and a memory pool type. The lock is used to ensure consistency of the allocator when used by multiple threads or processes. The memory pool is where the allocator obtains and releases memory from/to. To vary the memory allocation mechanism you want ACE_Malloc to use, you need to instantiate it with the right pool type.

ACE comes with several memory pools, including those that allocate shared memory and those that allocate regular heap memory. The various pools are detailed in Table 17.1.

As you can see, several pools are OS specific; make sure that you use a pool that is available on your OS. For example, to create an allocator that uses a pool that is based on a memory-mapped file, you would do something like this:

```
typedef ACE_Malloc <ACE_MMAP_Memory_Pool, ACE_SYNCH_MUTEX> MALLOC;
```

If your compiler does not support nested typedefs for template classes, you need to use an ACE-provided macro for the pool type to instantiate the template. The macro names are provided in Table 17.2. Using a macro, the sample example would be:

```
typedef ACE_Malloc<ACE_MMAP_MEMORY_POOL, ACE_SYNCH_MUTEX> MALLOC;
```

17.1.1 Map Interface

Besides a memory allocation interface that supports such operations as malloc(), calloc(), and free(), ACE_Malloc also supports a maplike interface. This interface is very similar to the interface supported by ACE_Map_Manager and ACE_Hash_Map_Manager, which we talked about in Chapter 5. This interface allows you to insert values in the underlying memory pool and associate them with a character string key. You can then retrieve the values you stored, using your key. The map also offers LIFO and FIFO iterators that iterate through the map in LIFO and FIFO order of insertion. In the next few sections, we will see several examples that use this interface.

Table 17.1. Memory Pool Types

Memory Pool Name	Description
ACE_MMAP_Memory_Pool	A memory pool based on a memory mapped file
ACE_Lite_MMAP_Memory_Pool	A lightweight version of a pool that is based on a memory-mapped file
ACE_Shared_Memory_Pool	A memory pool that is based on System V shared memory

17.2 Persistence with ACE_Malloc

Let's look at a simple example of using a shared memory allocator based on the ACE_MMAP_Memory_Pool class. One of the nice properties of the ACE_MMAP_Memory_Pool allocator is that it can be backed up by a file. That is, whatever you allocate using this allocator is saved to a backing file. As we mentioned earlier, you can use this mechanism to provide a simple persistence mechanism for your data. We will use this and the map interface to insert several records into a shared memory allocator with an ACE_MMAP_Memory_Pool.

Let's start by creating a few easy-to-use types that define the allocator type and iterator on that type. We also declare a global pointer to the allocator we are going to create:

```
#include "ace/Malloc_T.h"

typedef ACE_Malloc<ACE_MMAP_MEMORY_POOL, ACE_Null_Mutex>
  ALLOCATOR;
typedef ACE_Malloc_LIFO_Iterator <ACE_MMAP_MEMORY_POOL,
                                  ACE_Null_Mutex>
  MALLOC_LIFO_ITERATOR;

ALLOCATOR  *g_allocator;
```

Next, we instantiate the shared memory allocator. The only option we pass to the constructor is the name of the backing file where we wish to persist the records we will be adding to the allocator:

```
// Backing file where the data is kept.
#define BACKING_STORE "backing.store"

int ACE_TMAIN (int argc, ACE_TCHAR *[])
{
  ACE_NEW_RETURN (g_allocator,
                  ALLOCATOR (BACKING_STORE),
                  -1);
  if (argc > 1)
    {
      showRecords ();
    }
  else
    {
      addRecords ();
    }

  g_allocator->sync ();
  delete g_allocator;
  return 0;
}
```

The example needs to be run twice. The first time, you need to run it with no command line arguments; in that case, it will add new records to the memory pool. The second time, you need to run with at least one argument—anything will do, as we are looking only at the number of arguments in the command line—in which case, it will iterate through the inserted records and display them on the screen.

Next, let's look at the record type that we are inserting into memory:

```
class Record
{
```

17.3 Position-Independent Allocation

In the previous example, we glossed over several important issues that arise when you are using shared memory. The first issue is that it may not be possible for the underlying shared memory pool of an allocator to be assigned the same base address in all processes that wish to share it or even every time you start the same process. What does all this mean? Here is an example.

Let's say that process A creates a shared memory pool that has a base address of 0x40000000. Process B opens up the same shared memory pool but maps it to address 0x7e000000. Process A then inserts a record into the pool at 0x400001a0. If process B attempts to obtain this record at this address, the record will not be there; instead, it is mapped at 0x7e0001a0 in its address space! This issue continues to get worse, as the record itself may have pointers to other records that are all in shared memory space, but none are accessible to process B, as the base address of the pool is different for each process.

In most cases, the operating system will return the same base address for a ACE_MMAP_Memory_Pool by default. This is why the previous example worked. If the OS did not assign the same base address to the allocator, the previous example would not work. Further, if the underlying memory pool needs to grow as you allocate more memory, the system may need to remap the pool to a different base address. This means that if you keep direct pointers into the shared region, they may be invalidated during operation. (This will occur only if you use the special ACE_MMAP_Memory_Pool::NEVER_FIXED option; we talk more about this and other options later.)

As usual, however, ACE comes to the rescue. ACE includes several classes that, when used together, allow you to perform position-independent memory allocation. These classes calculate offsets from the current base address and store them in shared memory. So the allocator knows that the record is located at an offset of 0x01a0 from the base address instead of knowing only that the record is at 0x400001a0. Of course, this comes with some overhead in terms of memory use and processing, but it allows you to write applications that you know will work with shared memory.

Let's modify our previous example to use position-independent allocation:

```
#include "ace/Malloc_T.h"
#include "ace/PI_Malloc.h"

typedef ACE_Malloc_T <ACE_MMAP_MEMORY_POOL,
                      ACE_Null_Mutex,
                      ACE_PI_Control_Block>
  ALLOCATOR;
typedef ACE_Malloc_LIFO_Iterator_T<ACE_MMAP_MEMORY_POOL,
                                   ACE_Null_Mutex,
                                   ACE_PI_Control_Block>
  MALLOC_LIFO_ITERATOR;

ALLOCATOR  *g_allocator;
```

We start by changing the typedef for our allocator. Instead of using ACE_Malloc, we use its base class, ACE_Malloc_T. This template includes one additional parameter, a control block type. A control block is allocated in the shared memory pool to provide bookkeeping information. Here, we specify that we want to use the position-independent control block, ACE_PI_Control_Block. This ensures that the find() and bind() operations will continue to work even if the underlying pool is mapped to different addresses in different runs or in different processes:

```
class Record
{
public:
  Record (int id1, int id2, char *name)
```

17.4 ACE_Malloc for Containers

As you know, you can specify special-purpose allocators for containers. The container then uses the allocator to satisfy its memory needs. Wouldn't it be nice if we could use an ACE_Malloc shared memory allocator with the container and allocate containers in shared memory? This would mean that the container would allocate memory for itself in shared memory. The problem, of course, is that the allocator needs to implement the ACE_Allocator interface, but ACE_Malloc doesn't.

No need to fret, though, because ACE includes a special adapter class for this purpose. ACE_Allocator_Adapter adapts an ACE_Malloc-based class to the ACE_Allocator interface. This class template is very easy to use. To create the adapter, simply pass in the appropriate ACE_Malloc type. For example:

```
typedef ACE_Malloc<ACE_MMAP_MEMORY_POOL, ACE_Null_Mutex> MALLOC;
typedef ACE_Allocator_Adapter<MALLOC> ALLOCATOR;
```

Besides the issue of interface compatibility, another issue crops up. Most of the container classes keep a reference to the allocator they use and use this reference for all memory operations. This reference will point to heap memory, where the shared memory allocator itself is allocated. Of course, this pointer is valid only in the original process that created the allocator and not any other processes that wish to share the container. To overcome this problem, you must provide the container with a valid memory allocator reference for all its operations. As an example, ACE overloads the ACE_Hash_Map container—the new class is ACE_Hash_Map_With_Allocator—to provide this, and you can easily extend the idea to any other containers you wish to use.

Table 17.3. ACE_MMAP_Memory_Pool_Options Attributes

Option Name	Description
base_address	Specifies a starting address for where the file is mapped into the process's memory space.
use_fixed_addr	Specifies when the base address can be fixed; must be one of three constants: • FIRSTCALL_FIXED: Use the specified base address on the initial acquire • ALWAYS_FIXED: Always use the specified base address • NEVER_FIXED: Always allow the OS to specify the base address
write_each_page	Write each page to disk when growing the map.
minimum_bytes	Initial allocation size.
flags	Any special flags that need to be used for mmap().

17.5 Wrappers

Besides the more advanced features provided by ACE_Malloc and friends, you may want to do something much simpler. For example, mapping files into memory is a common technique many web servers use to reduce the time it takes to send high hit-rate files back to client browsers.

ACE provides several wrapper classes that wrap lower-level shared memory primitives, such as mmap(), MapViewOfFileEx(), System V shared memory segments, and so on. We take a brief look at ACE_Mem_Map, which is a wrapper around the memory-mapping primitives of the OS.

In this next example, we rewrite the classic Richard Stevens example of copying files [8], using memory mapping. We map both source and destination files into memory and then use a simple memcpy() call to copy source to destination:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  ACE_HANDLE srcHandle = ACE_OS::open (argv[1], O_RDONLY);
  ACE_ASSERT(srcHandle != ACE_INVALID_HANDLE);

  ACE_Mem_Map srcMap (srcHandle, -1, PROT_READ, ACE_MAP_PRIVATE);
  ACE_ASSERT(srcMap.addr () != 0);

  ACE_Mem_Map destMap (argv[2],
                       srcMap.size (),
                       O_RDWR | O_CREAT,
                       ACE_DEFAULT_FILE_PERMS,
                       PROT_RDWR,
                       ACE_MAP_SHARED);
  ACE_ASSERT(destMap.addr () != 0);

  ACE_OS::memcpy (destMap.addr (),
                  srcMap.addr (),
                  srcMap.size ());
  destMap.sync ();

  srcMap.close ();
  destMap.close ();
  return 0;
}
```

We create two ACE_Mem_Map instances that represent the memory mappings of both the source and destination files. To keep things interesting, we map the source by explicitly opening the file ourselves in read-only mode and then supplying the handle to srcMap. However, we let ACE_Mem_Map open the destination file for us in read/write/create mode for us. The PROT_READ and PROT_RDWR specify the protection mode for the pages that are mapped into memory. Here, the source file memory-mapped pages can only be read from, whereas the destination pages can be both read and written to. Finally, we specify the sharing mode as ACE_MAP_PRIVATE for the source file and ACE_MAP_SHARED for the destination. ACE_MAP_PRIVATE indicates that if any changes are made to the in-memory pages, the changes will not be propagated back to the backing store or to any other processes that have the same file mapped into memory. ACE_MAP_SHARED implies that changes are shared and will be seen in the backing store and in other processes.

In many cases, it is not feasible to map an entire file into memory at once. Instead, you can use ACE_Mem_Map to map chunks of the file into memory. You can then operate on the chunk, release it when you are done with it, and map the next chunk. To do this, you must specify the size of the chunk you want to map and the offset into the file where the chunk will begin.[1]

[1] It is important that the offset you specify into the file be properly aligned. This alignment is platform specific and

17.6 Summary

In this chapter, we reviewed the ACE_Malloc family of allocators. In particular, we talked about the shared memory pools you can use with ACE_Malloc. First, we showed how you can use an ACE_Malloc allocator to build a simple persistence mechanism. We then showed how you can use position-independent pointers to build portable applications that will work no matter where an allocator pool is mapped to in virtual memory. Next, we showed how to adapt the ACE_Malloc class to the ACE_Allocator interface and use it with the containers that are supplied with ACE. We also identified the problems that come with dynamic shared memory pool growth and explained how you can tackle them. Finally, we mentioned a few of the wrapper classes that ACE provides to deal with shared memory.

Chapter 18. ACE Streams Framework

The ACE Streams framework implements the Pipes and Filters pattern described in [1]. The framework is an excellent way to model processes consisting of a set of ordered steps. Each step, or filter in Pipes and Filters terminology, in the process is implemented as an ACE_Task derivative. As each step is completed, the data is handed off to the next step for continuation, using the ACE_Task objects' message queues, or pipes in Pipes and Filters terminology. Steps can be multithreaded in order to increase throughput if the data lends itself to parallel processing. In this chapter, we explore the following classes: ACE_Stream, ACE_Module, ACE_Task, and ACE_Message_Block.

18.1 Overview

Another commonly known implementation of the Pipes and Filters pattern is the UNIX System V STREAMS framework. If you're familiar with System V STREAMS, you will recognize the concepts. The ACE Streams framework allows the flexible, dynamically configurable assembly of a set of modules into a stream through which information travels. Each module has an opportunity to manipulate the data in the stream and can modify it, remove it, or add to it before passing it to the next module in the stream. Data moves bidirectionally in the stream, and each module in the stream has a reader and a writer task—one for each data direction.

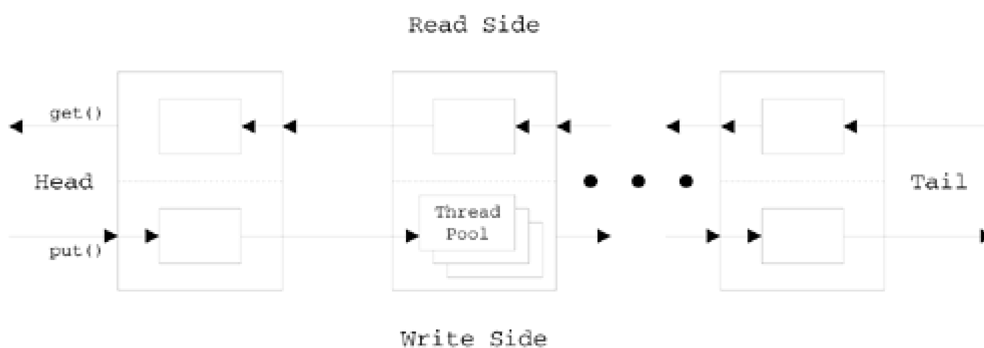
The first thing to do when using ACE Streams is to identify the sequence of events you wish to process. The steps should be as discrete as possible, with well-defined outputs and inputs. Where possible, identify and document opportunities for parallel processing within each step.

Once your steps are identified, you can begin implementing each as a derivative of ACE_Task. The svc() method of each will use getq() to get a unit of work and put_next() to pass the completed work to the next step. At this time, identify which tasks are "downstream" and which are "upstream" in nature. Downstream tasks can be thought of as moving "out of" your primary application as they move down the stream. For instance, a stream implementing a protocol stack would use each task to perform a different stage of protocol conversion. The final task would send the converted data, perhaps via TCP/IP, to a remote peer. Similarly, upstream tasks can be thought of as moving data "into" your primary application.

Module instances, or paired downstream and upstream tasks, can be created once the tasks are implemented. In our protocol conversion example, the downstream tasks would encode our data and the upstream tasks would decode it.

The final step is to create the ACE_Stream instance and push the ordered list of modules onto it. You can then use put() to move data onto the stream for processing and obtain the results with get(). See Figure 18.1.

Figure 18.1. Diagram of an ACE_Stream



18.2 Using a One-Way Stream

In this section, we look at an answering machine implementation that uses a one-way stream to record and process messages. Each module of the stream performs one function in the set of functions required to implement the system. The functions are

1. Answer incoming call
2. Collect Caller Identification data
3. Play outgoing message
4. Collect incoming message
5. Return recording device to the pool
6. Encode collected message into a normalized form
7. Save message and metadata into message repository
8. Send notification of received message

As defined, each step of the process is very specific in what it must do. This may seem a bit like overkill, but in a more complex application, it is an excellent way to divide the work among team members. Each person can focus on the implementation of his or her own step, as long as the interfaces between each step are well defined.

18.2.1 Main Program

We're going to work through this example from the top down; that is, we start with main() and then dig down into successively lower levels:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  RecordingDevice *recorder =
    RecordingDeviceFactory::instantiate (argc, argv);
```

Here, main() begins by using the RecordingDeviceFactory to instantiate a RecordingDevice instance based on command line parameters.[1] Our system may have many kinds of recording devices: voice modems, video phones, e-mail receivers, and so on. The command line parameters tell us which kind of device this instance of the application is communicating with.

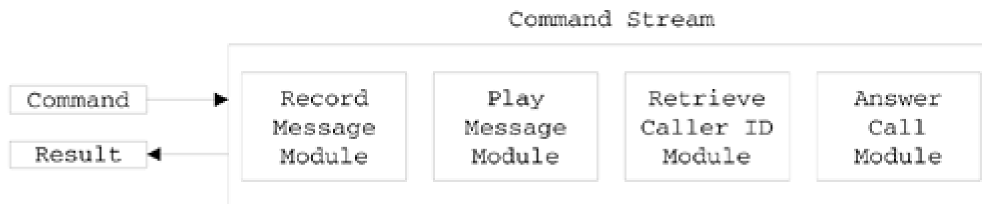
[1] Obviously, a more realistic application would be structured to read a list of devices from a configuration file and listen to several at one time.

```
RecordingStream *recording_stream;
```

18.3 A Bidirectional Stream

In this section, we use a bidirectional stream to implement a command stream. The general idea is that each module on the stream will implement one command supported by a RecordingDevice. The RecordingDevice will configure a Command object and place it on the stream; the first module capable of processing the Command will do so and return the results up the stream for the RecordingDevice to consume. Figure 18.2 shows the logical structure of the Command Stream.

Figure 18.2. Logical structure of the command stream



We will work this example from the inside out so that we can focus on the details of the CommandStream itself. The following subsections describe the stream, its tasks, and how the RecordingDevice derivative uses the stream.

18.3.1 CommandStream

For purposes of this example, we have created a RecordingDevice that will record a message delivered on a socket. Each task pair of our command stream will implement a RecordingDevice command by interacting with the socket in an appropriate manner.

We begin with the definition of our CommandStream class:

```
class CommandStream : public ACE_Stream<ACE_MT_SYNCH>
{
public:
  typedef ACE_Stream<ACE_MT_SYNCH> inherited;

  CommandStream () : inherited()  { }
  int open (ACE_SOCK_Stream *peer);
  Command *execute (Command *command);
};
```

We expect clients of CommandStream to do three things:

1. Instantiate the object.
2. Open the object and provide a socket.
3. Request execution of one or more commands.

The open() method is where the stream's modules are created and pushed onto the stream. The method begins by invoking the superclass's open() method so that we don't have to duplicate that functionality:

```
int CommandStream::open (ACE_SOCK_Stream *peer)
{
```

18.4 Summary

In this chapter, we have investigated the ACE Streams framework. An ACE_Stream is nothing more than a doubly linked list of ACE_Module instances, each of which contains a pair of ACE_Task derivatives. Streams are useful for many things, only two of which we've investigated here. The following list enumerates how one would use a stream.

1. Create one or more ACE_Task derivatives that implement your application logic.
2. If applicable, pair these tasks into downstream and upstream components.
3. For each pair, construct a module to contain them.
4. Push the modules onto the stream in a last-used/first-pushed manner.

Keep in mind the following when architecting your stream.

- Each task in the stream can exist in one or more threads. Use multiple threads when your application can take advantage of parallel processing.
- The default tail tasks will return an error if any data reaches them. If your tasks don't entirely consume the data, you should at least provide a replacement downstream tail task.
- You can provide your task's open() method with arbitrary data by passing it as the fourth parameter (args) of module's constructor or open() method.
- A task's open() method is invoked as a result of pushing its module onto the stream, not as a result of invoking open() on the module.
- ACE_Message_Block instances are reference counted and will not leak memory if you correctly use their duplicate() and release() methods.

One last thing: If you're wondering what happened to the protocol stack we mentioned, rest assured that we haven't forgotten it. The ACE source is distributed with online tutorials. Please check out tutorial 15 in the ACE_wrappers/docs/tutorials directory for a protocol stream implementation example.

Chapter 19. ACE Service Configurator Framework

As we've shown so far, ACE permits tremendous flexibility when you are designing your applications. You can exchange classes, change behavior by using different strategies and template arguments, and easily change services to use multiple threads, multiple processes, use the Reactor framework, and/or use the Proactor framework.

However, you often need the flexibility to configure your application differently at runtime to use different services. You might need to do this for the following reasons.

- Your customers or users need to be able to move certain services to other machines or processes to make better use of their network or other available resources.
- You may offer optional pieces of a system and don't want them to always take up space in your program or use up memory when they're not used.
- You may have services that you always use but that you want the users to be able to configure differently, with site-specific options, for example.

You may want to allow your users to make all these changes at will, dynamically, while your application is running. This allows you to write services that you don't decide how or if they'll be used until runtime. In this chapter, we look at the ACE Service Configurator framework, which gives you all this flexibility.

19.1 Overview

The ACE Service Configurator framework is an implementation of the Component Configurator pattern [5]. You can dynamically configure services and streams at runtime, whether they are statically linked into your program or the objects are dynamically loaded from shared libraries. You can configure both services—one object representing a service—and streams—assembling modules based on a configuration file rather than at compile time. Both the arrangement of the services/streams and configuration arguments similar to command line arguments can be specified. Further, you can add and remove services to a running program, suspend services, and resume them. All this can be done by using a configuration file or by making method calls on ACE_Service_Config with text lines that would otherwise be read from a configuration file. Runtime configuration is beneficial for the following reasons.

- Multiple types of services are linked into the program or are available in shared libraries, and the set of services to activate is deferred until runtime, enabling site- or configuration-specific sets of services to be activated.
- Different arguments can be passed into the service initialization. For example, the TCP port number for a service to listen on can be specified in the configuration file rather than compiled into the program.

19.2 Configuring Static Services

A static service is one whose code is already linked into the executable program. Configuring a static service is done for the same reasons as for a dynamic service. Additionally, statically linked programs are sometimes favored for simplicity or for security concerns. Following is an example of a statically configured service; note the code involved in initializing the service, stopping the service, and how those are controlled:

```
#include "ace/OS.h"
#include "ace/Acceptor.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Service_Object.h"
#include "ace/Svc_Handler.h"

class ClientHandler :
      public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
    // ... Same as previous examples.
};

class HA_Status : public ACE_Service_Object
{
  public:
    virtual int init (int argc, ACE_TCHAR *argv[]);
    virtual int fini (void);
    virtual int info (ACE_TCHAR **str, size_t len) const;

  private:
    ACE_Acceptor<ClientHandler, ACE_SOCK_ACCEPTOR> acceptor_;
    ACE_INET_Addr listen_addr_;
};
```

We're building the HA_Status service as a statically linked, configurable service. Using this technique, the service is not instantiated or activated until the ACE Service Configurator framework explicitly activates it, although all the code will already be linked into the executable program.

The preceding class declaration shows two important items that are central to developing configurable services.

1. Your service class must be a subclass of ACE_Service_Object. Remember that ACE_Task is derived from ACE_Service_Object and that ACE_Svc_Handler is a subclass of ACE_Task; therefore, ACE_Task and ACE_Svc_Handler are often used to implement configurable services.
2. Each service needs to implement the following hook methods:
   - virtual int init (int argc, ACE_TCHAR *argv[]), which is called by the framework when an instance of this service is initialized. If arguments were specified to the service initialization, they are passed via the method's arguments.
   - virtual int fini (void), which is called by the framework when an instance of the service is being shut down.
   - virtual int info (ACE_TCHAR **str, size_t len), which is optional. It is used for the service to report

19.3 Setting Up Dynamic Services

Dynamic services can be dynamically loaded from a shared library (DLL) when directed at runtime. They need not be linked into the program at all. This dynamic-loading capability allows for great flexibility of substitution at runtime, as the code for the service need not even be written when the program is. Existing services can be removed and new services added dynamically by directives in a service configuration file or programatically.

The procedure for writing a dynamic service is very similar to that for writing a static service. The primary differences are as follows.

- The service class(es) will reside in a shared library (DLL) instead of being linked into the main program. Therefore, when declaring the service class—derived from ACE_Service_Object, just as for static services—the proper DLL export declaration must be used. Section 2.5.1 describes these declarations.

- The only service-related macro needed for a dynamic service is ACE_FACTORY_DEFINE. The record-keeping macros used for static services are not needed. The record-keeping information for dynamic services is created dynamically at runtime as the services are configured.

The following example shows the same service used in our static-service example but adjusted to work as a dynamic service. First, the class declaration:

```
#include "ace/OS.h"
#include "ace/Acceptor.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Service_Object.h"
#include "ace/Svc_Handler.h"

#include "HASTATUS_export.h"

class ClientHandler :
      public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
    // ... Same as previous examples.
};

class HASTATUS_Export HA_Status : public ACE_Service_Object
{
  public:
    virtual int init (int argc, ACE_TCHAR *argv[]);

    virtual int fini (void);

    virtual int info (ACE_TCHAR **str, size_t len) const;

  private:
    ACE_Acceptor<ClientHandler, ACE_SOCK_ACCEPTOR> acceptor_;
    ACE_INET_Addr listen_addr_;
};
```

As you can see, the only difference between this version of the service and the static version is the addition of the HASTATUS_Export specification on the HA_Status class declaration and the inclusion of the HASTATUS_export.h header file, which defines the needed import/export specifications.

The code for the service is exactly the same as that used in the static service example in this chapter. The

19.4 Setting Up Streams

Configuring streams by using the configuration file involves a number of lines to specify the modules to be included in the stream and their relationships:

```
stream static|dynamic [modules]

stream ident [modules]

modules:
{ [[service-specification] [service-specification...]] }
```

A set of service specifications is given, listed one per line. Each successive service is created, has its init() method called with the arguments from the service specification, then is pushed onto the stream:

```
dynamic ident Module *

dynamic ident STREAM * lib-pathname : object-class [active|inactive] [parameters]
```

19.5 Reconfiguring Services During Execution

So far, we've looked primarily at how to write code for configurable services and initialize those services at runtime. The other side of configurable services is being able to remove them. This capability makes it possible to add, replace, and remove services without interrupting the program as a whole, including other services executing in the same process. It is also possible to suspend and resume individual services without removing or replacing them.

The directive to remove a service that was previously initialized, such as the two services we previously showed in this chapter, is:

```
remove service
```

where service is the service name used in either the static or dynamic directive used to initialize the service. For example, the following directive initiates the removal of the HA_Status_Dynamic_Service:

```
remove HA_Status_Dynamic_Service
```

When this directive is processed, the service's fini() method is called; then the service object created by the factory function is deleted:

```
int
HA_Status::fini (void)
{
  this->acceptor_.close ();
  return 0;
}
```

We have learned how service configuration directives are processed at program start-up via the ACE_Service_Config::open() method. So how do these configuration directives get processed after that point? As we saw in the example programs in this chapter, after the Service Configurator framework is initialized, control often resides in the Reactor framework's event loop. There are two ways to make ACE reprocess the same service configuration file—or set of files—used when ACE_Service_Config::open() was called.

1. On systems that have POSIX signal capability, send the process a SIGHUP signal. For this to work, the program must be executing the Reactor event loop. As this is often the case, the requirement is not usually a problem. Note that the signal number to use for this can be changed when the program starts, by specifying the -s command line option (see Table 19.1).
2. The program itself can call ACE_Service_Config::reconfigure() directly. This is the favored option on Windows, as POSIX signals are not available, and can also be used for programs that are not running the Reactor event loop. To make this more automatic on Windows, it's possible to create a file/directory change event, register the event handle with the ACE_WFMO_Reactor, and use the event callback to do the reconfiguration.

Both of these options will reprocess the service configuration file(s) previously processed via ACE_Service_Config::open(). Therefore, the usual practice is to comment out the static and/or dynamic service initialization directives and add—or uncomment, if previously added—the desired remove directives, save the file(s), and trigger the reconfiguration.

19.6 Using XML to Configure Services and Streams

ACE version 5.3 added a new, optional way to configure services and streams. It's based on XML service configuration files rather than the syntax we've seen so far. At some future ACE version, the XML form will become the standard configuration file syntax, and the syntax we've seen to this point will become optional but won't be removed. The document type definition (DTD) for the new language follows:

```
<!ELEMENT ACE_Svc_Conf (dynamic|static|suspend|resume
                          |remove|stream|streamdef)*>
<!ELEMENT streamdef ((dynamic|static),module)>
<!ATTLIST streamdef id IDREF #REQUIRED>
<!ELEMENT module (dynamic|static|suspend|resume|remove)+>
<!ELEMENT stream (module)>
<!ATTLIST stream id IDREF #REQUIRED>
<!ELEMENT dynamic (initializer)>
<!ATTLIST dynamic id ID #REQUIRED
                  status (active|inactive) "active"
                  type (module|service_object|stream)
                  #REQUIRED>
<!ELEMENT initializer EMPTY>
<!ATTLIST initializer init CDATA #REQUIRED
                      path CDATA #IMPLIED
                      params CDATA #IMPLIED>
<!ELEMENT static EMPTY>
<!ATTLIST static id ID #REQUIRED
                 params CDATA #IMPLIED>
<!ELEMENT suspend EMPTY>
<!ATTLIST suspend id IDREF #REQUIRED>
<!ELEMENT resume EMPTY>
<!ATTLIST resume id IDREF #REQUIRED>
<!ELEMENT remove EMPTY>
<!ATTLIST remove id IDREF #REQUIRED>
```

The syntax of this XML-based configuration language is different from the current service configuration language in ACE, but its semantics are the same. Although it's more verbose to compose, the ACE XML-based configuration file format is more flexible. For example, users can plug in customized XML event handlers to extend the behavior of the ACE Service Configurator framework without modifying the underlying ACE implementation. You can try this format out by adding the following to your ace/config.h file and rebuilding ACE:

```
#define ACE_HAS_XML_SVC_CONF
```

Do not worry about converting all your current configuration files. ACE provides the svcconf-convert.pl perl script to translate original-format files into the new XML format. The script is located in the ACE_wrappers/bin directory.

19.7 Configuring Services without svc.conf

In some situations, it may be too restrictive to direct service (re)configuration operations completely by using service configuration files. For example, a program may wish to instantiate or alter a service in direct response to a service request instead of waiting for an external event. To enable this direct action on the Service Configurator framework, the ACE_Service_Config class offers the following method:

```
static int process_directive (const ACE_TCHAR directive[]);
```

The argument to this method is a string with the same syntax as a directive you could place in a service configuration file. If you want to process a configuration file's contents, whether or not the Service Configurator framework has seen the file, you can pass the file specification to this method:

```
static int process_file (const ACE_TCHAR file[]);
```

ACE_Service_Config also offers methods to finalize, suspend, and resume individual named services:

```
static int remove (const ACE_TCHAR svc_name[]);

static int suspend (const ACE_TCHAR svc_name[]);

static int resume (const ACE_TCHAR svc_name[]);
```

19.8 Singletons and Services

Recall from Section 1.6.3 that the ACE_Object_Manager provides a useful facility for ensuring that objects can be cleaned up properly when your program runs down. The ACE_Singleton class is integrated with the Object Manager facility, and many application services use this feature to manage singleton lifetimes correctly. However, when using this facility, you must remember when the singleton objects are destroyed: at program rundown, when the ACE_Object_Manager does its cleanup work. Now consider what happens when a dynamically loaded service that makes use of ACE_Singleton is unloaded before the Object Manager cleans up singletons. Usually, very bad things happen. The code that was to run the destruction of the service's singleton is probably not mapped into your process when the ACE_Object_Manager cleanup happens, and you'll likely get an access violation.

To make the instantiate-when-needed, double-checked locking safety of ACE_Singleton available to dynamically loaded services without the danger of having the cleanup performed after the service is unloaded, ACE offers the ACE_Unmanaged_Singleton class, which is used exactly like ACE_Singleton, except that an unmanaged singleton is not registered with ACE_Object_Manager and is not automatically cleaned up at program termination. To delete an unmanaged singleton, you must call its close() method. You should do this from your service's fini() method, as that's the dynamic service equivalent of program termination.

19.9 Summary

The Service Configurator framework allows you to divide your system into individual services and then include them or not and configure them at runtime instead of when you build your application. Users can customize the behavior of your applications to suit their site's requirements and resources and can reconfigure dynamically, adding to the availability and flexibility of your applications.

In this chapter, we gave some background of the ACE Service Configurator framework, said why it is useful, and explained how to use it. We showed how to write code that can be dynamically loaded, including those pesky Windows-required import/export directives, and how to configure your services via a configuration file and also directly in code that can be executed on demand.

We also gave a glimpse into a new feature of ACE that will become the standard way of configuring services: the XML-based service configuration file. You can use this facility today by building it into ACE, and ACE also provides a conversion script to convert your existing ACE service configuration files to the new XML format.

Finally, we explained why you need to be careful with singletons and their interaction with the ACE Object Manager facility. With the ACE_Unmanaged_Singleton, you are completely free of the need to always build services into your applications at link time. Your job will be easier and your customers impressed.

Chapter 20. Timers

Timers are used in almost all types of software but are especially useful for writing systems or network software. Most timers are implemented with the assistance of an underlying hardware timer, accessed through your operating system interface, which indicates timer expiration in an OS-dependent fashion. Once the hardware timer expires, it notifies the operating system, which in turn will notify the calling application. Achieving all this varies greatly, depending on the machine/OS.

In this chapter, we introduce timers and how they are built with timer queues. We then discuss the timer queue facilities provided by ACE and incorporate them into a crude timer dispatcher. Next, we show you some of the prebuilt timer dispatchers that are a part of ACE and that you can use directly in your applications. Finally, we change the event handler hierarchy to build our own private callback classes independent of ACE_Event_Handler.

20.1 Timer Concepts

Most operating systems provide for only a few unique timers. However, in most network software, a large number of extremely efficient timers are needed. For example, it is plausible to have one timer associated with each packet that is sent and held within a network queue. In turn, it is also necessary that the process of timer setup and cancellation be extremely efficient.

Much research has taken place in this arena, resulting in the design of highly efficient data structures called timer queues. These ordered timer queues hold nodes that individually specify the next timeout that the user is interested in. The head of the queue always contains the next timeout value. This value is obtained and then fed back to the underlying hardware timer through an OS interface. Once the timer goes off, a new timeout value is obtained from the timer queue and set as the next timeout.

ACE has implementations for many of these types of timer queues. In addition, these queues directly support interval timers, which expire repeatedly after a specified interval has elapsed. ACE timer queues also offer facilities that can be used to dispatch specified handlers once timeout does occur. ACE is flexible in that to handle timeouts, you can either use the default event handler hierarchy, based on ACE_Event_Handler, or you can roll your own.

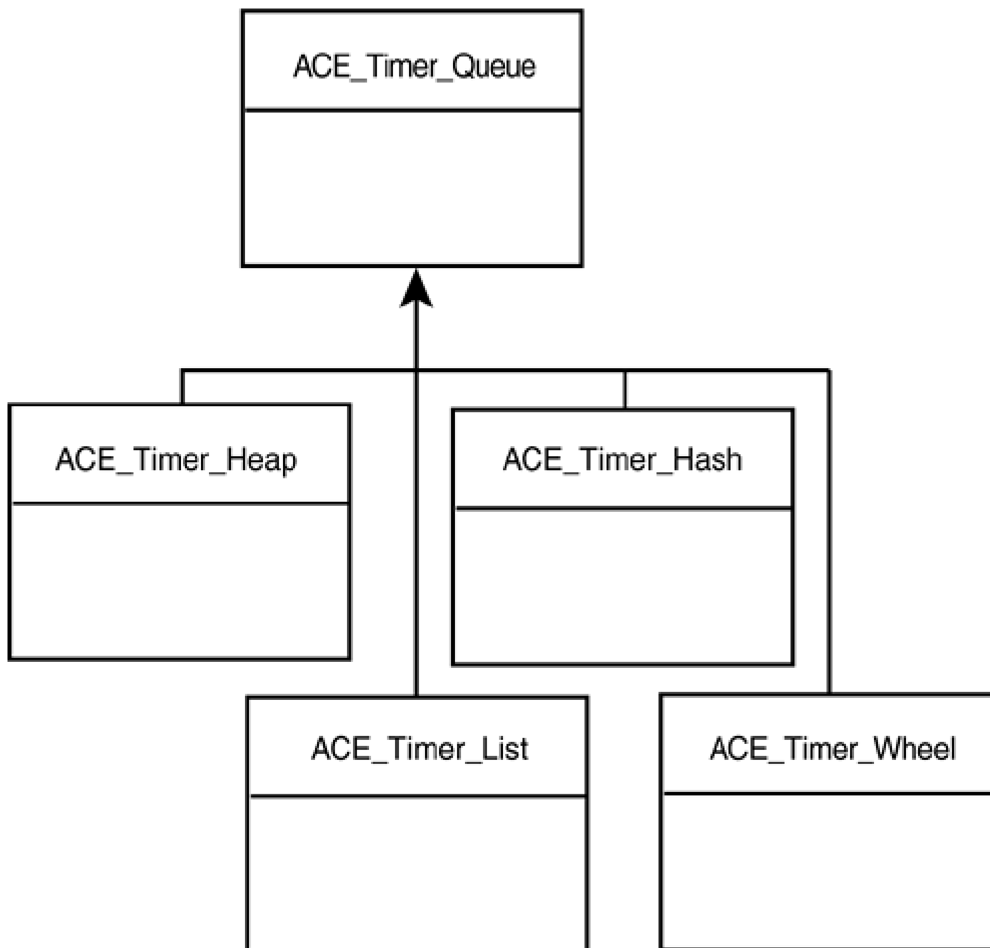
Table 20.1. Timer Storage Structures and Relative Performance

Timer	Description of Structure	Performance
ACE_Timer_Heap	The timers are stored in a heap implementation of a priority queue.	schedule: O(lg n) cancel: O(lg n) find: O(1)
ACE_Timer_List	The timers are stored in a doubly linked list.	schedule: O(n) cancel: O(1) find: O(1)
ACE_Timer_Hash	A variation on the timer wheel algorithm. The performance is highly dependent on the hashing function used.	schedule (worst case): O(n) schedule (best case): O(1) cancel: O(1) find: O(1)
ACE_Timer_Wheel	The timers are stored in an array of "pointers to arrays," whereby each array being pointed to is sorted.	schedule (worst case): O(n) schedule (best case): O(1) cancel: O(1) find: O(1)

20.2 Timer Queues

All ACE timer queues derive from the abstract base class ACE_Timer_Queue. This means that they are runtime substitutable with one another, as illustrated in Figure 20.1.

Figure 20.1. Timer queue class hierarchy



Each concrete timer queue uses a different algorithm for maintaining timing nodes in order, each with different time-complexity characteristics (see Table 20.1). These characteristics should drive your decision in moving from one timer queue mechanism to another one. Further, if none of ACE prebuilt timers satisfy your requirements, you can subclass and create your own timer queue.

Each timer data structure also differs in the way it manages memory. ACE_Timer_Heap avoids expensive memory allocation by preallocating memory for internal timer nodes. When using a timer heap, you can specify how many entries are to be created in the underlying heap array. Similarly, ACE_Timer_Wheel can be set up to allocate memory from a free list of timer nodes, the free list can be provided by the programmer. For specific details, refer to the ACE reference documentation.

In the following example, we create a TimerDispatcher, which we use to register event handlers (CB) that are called when scheduled timers expire. The timer dispatcher contains two parts: an ACE timer queue and a timer driver. We use the term timer driver to mean the mechanism that instigates, or causes, the indication of a timeout event. You can use various schemes for timer indication, including your private OS timer API, timed condition variables, timed semaphores, timed event loop mechanisms, and so on. In our example, we use an ACE_Event object to drive the timer, as illustrated in Figure 20.2.

Figure 20.2. Timer dispatcher example class diagram

20.3 Prebuilt Dispatchers

Having described at some length how you can create your own timer dispatcher, we now give you an overview of some of the prebuilt ACE timer dispatchers.

20.3.1 Active Timers

ACE provides an active-timer queue class that not only encapsulates the OS-based timer mechanism but also runs the timer event loop within its own private thread of control; hence the name active-timer queue. In this next example, we use an ACE_Event_Handler-based callback, very similar to the one we used in the previous example:

```
#include "ace/Timer_Queue_Adapters.h"
#include "ace/Timer_Heap.h"

typedef ACE_Thread_Timer_Queue_Adapter<ACE_Timer_Heap> ActiveTimer;
```

The ActiveTimer adapter allows you to specify any one of the concrete timer queues as the underlying timer queue for the active timer. In this case, we chose to use the ACE_Timer_Heap queue:

```
class CB : public ACE_Event_Handler
{
public:
  CB (int id) : id_(id) { }

  virtual int handle_timeout (const ACE_Time_Value &tv,
                              const void *arg)
  {
    ACE_TRACE (ACE_TEXT ("CB::handle_timeout"));

    const int *val = ACE_static_cast (const int*, arg);
    ACE_ASSERT((*val) == id_);
    ACE_DEBUG ((LM_DEBUG,
                ACE_TEXT ("Expiry handled by thread %t\n")));
    return 0;
  }
private:
  int id_;
};
```

We start by creating a useful timer callback handler. As the timer dispatcher is active, the handle_timeout() method of the event handler will be dispatched, using the active timer's private thread of control. We display this, using the ACE_DEBUG() macro's %t format specifier:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  ACE_DEBUG ((LM_DEBUG,
              ACE_TEXT ("the main thread %t has started \n")));

  // Create an "active" timer and start its thread.
  ActiveTimer atimer;
  atimer.activate ();

  CB cb1 (1);
  CB cb2 (2);
  int arg1 = 1;
```

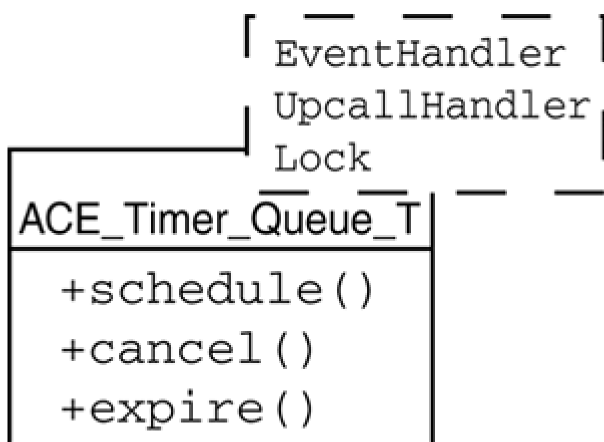
20.4 Managing Event Handlers

You will probably use the ACE_Event_Handler hierarchy almost exclusively for your timer needs unless you need to integrate with a preexisting event-handling hierarchy. In these cases, you can create or integrate with your own event-handling mechanisms. However, before we can go into how you can do this, it is necessary to understand the template types behind the ACE_Timer_Queue hierarchy of timer queues.

Underneath the covers of the ACE timer queues used in all our previous examples lie a couple of template-based timer queue classes (Figure 20.4). These template classes allow the template instantiator to specify

- The type of callback handler that will be called back when a timer expires
- The upcall manager, which calls methods on the callback handler when timer expiration or cancellation occurs
- A lock to ensure thread safety of the underlying timer queue

Figure 20.4. Timer queue template classes

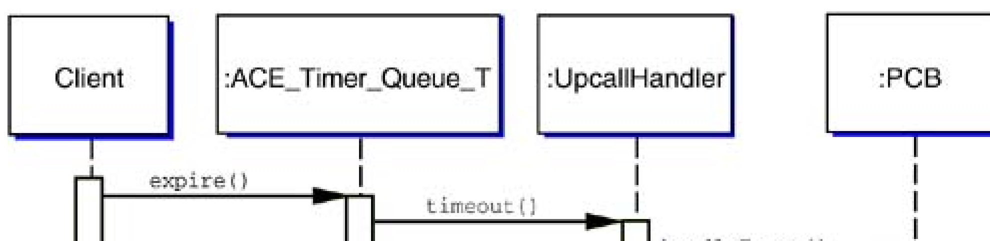


In fact, the ACE_Timer_Queue class is created by instantiating the ACE_Timer_Queue_T class template with the ACE_Event_Handler class as the callback handler type and an ACE internal upcall handler class as the upcall manager.

When an action is performed on any of the ACE timer queue classes, a method of the upcall manager is automatically called. This provides you, the application programmer, with a hook to decide what should happen when the action occurs. This allows you to set up your own upcall manager to call back your specific event handler class methods.

Figure 20.5 illustrates how the timer queue, upcall manager, and a private callback class (PCB) would interoperate. When the timer queue's expire() method is called, the timeout() method of the supplied upcall handler is invoked. In this case, we call handleEvent() on our new callback class, whereas the ACE-provided upcall handler would have invoked handle_timeout(). Similarly, when cancel() is invoked by the timer queue client, cancellation() is called on the upcall handler; when the timer queue is deleted, deletion() is called on the upcall handler.

Figure 20.5. Upcall handler sequence diagram



20.5 Summary

In this chapter, we began by talking about timers and how timer queues and timer drivers are combined to create timer dispatchers. We then created our own timer dispatcher that used ACE_Event as the underlying timer driver. Next, we took a look at the timer dispatchers that are provided as a part of the ACE framework. Finally, we decided to go deep down and replace the upcall handler and event handler classes that are used by the timer queues with our own home-grown event handler class.

Chapter 21. ACE Naming Service

The ACE Naming Service provides your application with a persistent key/value mapping mechanism, which can, for instance, create a server-to-address mapping much like DNS but tailored to your application's needs. The name space can cover a single process, all processes on a single node, or many processes on a network.

21.1 The ACE_Naming_Context

The ACE_Naming_Context class is the center of the Naming Service universe. Once you have an instance of a naming context, you can begin to provide it with key/value pairs and fetch data from it. In addition to the key/value pair, you can also provide a type value. The type does not augment the key in any way. That is, keys are unique within the entire naming context, not just within a type. However, you can use the type to group a set of related keys for later resolution.

A naming context instance is typically used to:

- Bind or rebind a key to a value in the naming context
- Unbind, or remove, an entry from the context
- Resolve, or find, an entry based on a key
- Fetch a list of names, values, or types from the context
- Fetch a list of name/value/type bindings from the context.

Table 21.1. Name Options Attributes

Attribute	Meaning
TCP port	The TCP port at which a client of a network mode naming service will connect.
Host name	The host name at which a client will find a network mode naming service.
Context type	Instructs the naming context to use a process, node, or network local database.
Namespace directory	A location in the file system where the persistent namespace data is kept.
Process name	The name of the current process; in process-local mode, the default database name.
Database	The name of the database to use if the default is not appropriate.
Base address	Allows you to get/set the address of the underlying allocator used for creating entries in the database.
Registry use	Applications in the Windows environment can choose to

21.2 A Single-Process Naming Context : PROC_LOCAL

We will use the PROC_LOCAL context to show a single application accessing the database. By "single application," we mean a named application, that is, having the same argv[0], not necessarily a single instance of an application.

The sample application will poll a thermometer device to request the current temperature. The current and previous temperatures will be stored in the naming context. In addition, a reset mechanism will be implemented such that the thermometer can be reset if there are too many successive failures. The naming context is used to record the first failure time, most recent failure time, number of resets, and so forth, so that the application can reset the thermometer intelligently and notify someone if necessary. By storing the values in the naming context, the data will be preserved even if the application is restarted.

First, our application creates a helper object to manage the command line options. This isn't strictly necessary but is a handy way to delegate that work to another object and keep main() clear of clutter:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  Temperature_Monitor_Options opt (argc, argv);
```

With that out of the way, we can create a Naming_Context instance and fetch the ACE_Naming_Options from it:

```
Naming_Context naming_context;

ACE_Name_Options *name_options = naming_context.name_options();
```

The Naming_Context is a simple derivative of ACE_Naming_Context and provides a few handy methods. We'll come to those details shortly.

Ordinarily, you would provide argc/argv directly to the parse_args()[1] method of the name context instance. Our sample application has opted to consume those values with its own options manager, however, so we must manually initialize the name options attributes:

[1] Because the Naming Service is generally considered an advanced topic, it will frequently be used alongside other advanced features of ACE. In particular, it is frequently used in applications that rely on the Service Configurator. In these situations, the context's parse_args() is fed from the svc.conf file's data. In order to keep the example focused and a little simpler, we've chosen not to use the Service Configurator.

```
char *naming_options_argv[] = { argv[0] };
name_options->parse_args
  (sizeof(naming_options_argv) / sizeof(char*),
  naming_options_argv);
name_options->context (ACE_Naming_Context::PROC_LOCAL);
naming_context.open (name_options->context ());
```

After setting the name options attributes, we give it to the context's open() method. Once the context has been opened, we're free to instantiate our temperature monitor object and turn control over to it:

```
Temperature_Monitor temperature_monitor (opt, naming_context);
temperature_monitor.monitor ();
```

21.3 Sharing a Naming Context on One Node: NODE_LOCAL

NODE_LOCAL mode is what you want to use when you have multiple applications on the same system needing to use a single naming context. In this section's example, we modify the previous application to write the ten most recent successful results to a NODE_LOCAL naming service. A second application then polls this naming context periodically to create a graph of the temperature history.

21.3.1 Saving Shared Data

We begin with a new version of main() to create a second naming context instance in which we'll store the shared information:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  Temperature_Monitor_Options opt (argc, argv);
  Naming_Context process_context;
  {
    ACE_Name_Options *name_options =
      process_context.name_options ();
    name_options->context (ACE_Naming_Context::PROC_LOCAL);
    ACE_TCHAR *nargv[] = { argv[0] };
    name_options->parse_args (sizeof(nargv) / sizeof(ACE_TCHAR*) ,
                              nargv);
    process_context.open (name_options->context ());
  }

  Naming_Context shared_context;
  {
    ACE_Name_Options *name_options =
      shared_context.name_options ();
    name_options->process_name (argv[0]);
    name_options->context (ACE_Naming_Context::NODE_LOCAL);
    shared_context.open (name_options->context ());
  }

  Temperature_Monitor2 temperature_monitor (opt,
                                            process_context,
                                            shared_context);
  temperature_monitor.monitor ();

  process_context.close ();
  shared_context.close ();

  return 0;
}
```

This is very much like the previous example's main(), even though we're creating two Naming_Context instances. We've taken the opportunity to initialize the instances in slightly different ways. Both are effective; choose the version that best fits your application's needs.

Next, we look at the modified Temperature_Monitor object. The monitor() loop is no different: Instantiate a Thermometer, fetch the temperature, and record success or failure. The record_temperature() method includes a new hook to record the temperature history:

```
void Temperature_Monitor2::record_temperature (float temp)
{
  Name_Binding_Ptr current
    (this->naming_context_.fetch ("current"));
```

21.4 Sharing a Naming Context across the Network: NET_LOCAL

At the end of Section 21.1, we made a comment about the API presented by the Naming Service. Specifically, the API used by clients of the Naming Service remains constant, regardless of the context—process, node, or network—chosen. In this section, we leverage that consistency to convert the two NODE_LOCAL programs from Section 21.3 to the NET_LOCAL context.

21.4.1 Starting the Naming Service Server

For NET_LOCAL mode, we must have a Naming Service instance executing somewhere in our network. The Naming Service is a part of ACE's network services (netsvcs) functionality. (The client and server logging servers we discussed in Chapter 3 are also part of netsvcs.) The service is dynamically loadable and can be loaded into any process. Possibly the easiest way to do so is to leverage the generic networking services application distributed with the ACE framework. In the netsvcs/servers directory is main.cpp, which can execute one or more of the ACE networking services specified in a svc.conf file. For our purposes, we need only the Naming Service, so we'll create a simple svc.conf:[6]

[6] Please refer to Section 19.3 for details on configuring dynamic services.

```
dynamic Name_Server Service_Object *
netsvcs:_make_ACE_Name_Acceptor() "-p 20012"
```

When executed, main will create a Naming Service instance listening on TCP/IP port 20012. Our modified temperature monitor and graphing applications will then look there for their naming service needs.

21.4.2 Modifying the Applications

Modifying the two applications from NODE_LOCAL to NET_LOCAL mode is easy. First, we'll modify the temperature monitor process:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  Temperature_Monitor_Options opt (argc, argv);

  Naming_Context process_context;
  {
    ACE_Name_Options *name_options =
      process_context.name_options ();
    name_options->context (ACE_Naming_Context::PROC_LOCAL);
    ACE_TCHAR *nargv[] = { argv[0] };
    name_options->parse_args (sizeof(nargv) / sizeof(ACE_TCHAR*),
                              nargv);
    process_context.open (name_options->context ());
  }

  Naming_Context shared_context;
  {
    ACE_Name_Options *name_options =
      shared_context.name_options ();
    name_options->process_name (argv[0]);
    name_options->context (ACE_Naming_Context::NET_LOCAL);
    shared_context.open (name_options->context ());
  }

  Temperature_Monitor2 temperature_monitor (opt,
                                            process_context,
```

21.5 Summary

The ACE Naming Service is an easy mechanism for storing and sharing name/value pairs. The architecture is such that it can easily keep up with the growth of your application as it grows from a single, stand-alone process to a distributed application running on several networked nodes.

Bibliography

1. F. Buschmann et al. 1996. Pattern-Oriented Software Architecture—A System of Patterns. Wiley.

2. T. Cormen et al. 2001. Introduction to Algorithms, Second Edition. MIT Press.

3. E. Gamma et al. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

4. R. Johnson and B. Foote. 1988. "Designing Reusable Classes," Journal of Object-Oriented Programming, SIGS, June/July 1988, 1(5): 22–35.

5. D. Schmidt et al. 2000. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. Wiley.

6. Schmidt, D. C., and S. D. Huston. 2002. C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns. Addison-Wesley.

7. Schmidt, D. C. and S. D. Huston. 2003. C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks. Addison-Wesley.

8. Stevens, W. R. 1992. Advanced Programming in the UNIX Environment. Addison-Wesley.

9. Stevens, W. R. 1994. TCP/IP Illustrated, Volume 1. Addison-Wesley.

10. Stevens, W. R. 1998. UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2d ed. Prentice-Hall.

11. Stroustrup, B. 1997. The C++ Programming Language, 3rd Edition. Addison-Wesley.

12. Tanenbaum, A. 1996. Computer Networks 3d ed. Prentice-Hall.

13. van Rooyen, M. "Alternative C++: A New Look at Reference Counting and Virtual Destruction in C++". In C++ Report 8(4), April 1996.

14. Vandevoorde, D., and N. M. Josuttis. 2003. C++ Templates: The Complete Guide. Addison-Wesley.

CD-ROM Warranty

Addison-Wesley warrants the enclosed disc to be free of defects in materials and faulty workmanship under normal use for a period of ninety days after purchase. If a defect is discovered in the disc during this warranty period, a replacement disc can be obtained at no charge by sending the defective disc, postage prepaid, with proof of purchase to:

Editorial Department
Addison-Wesley Professional
Pearson Technology Group
75 Arlington Street, Suite 300
Boston, MA 02116
Email: AWPro@awl.com

Addison-Wesley makes no warranty or representation, either expressed or implied, with respect to this software, its quality, performance, merchantability, or fitness for a particular purpose. In no event will Addison-Wesley, its distributors, or dealers be liable for direct, indirect, special, incidental, or consequential damages arising out of the use or inability to use the software. The exclusion of implied warranties is not permitted in some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have that vary from state to state. The contents of this CD-ROM are intended for personal use only.

More information and updates are available at: http://www.awprofessional.com/