The ACE Programmer's Guide®

Stephen D. Huston

James CE Johnson

Umar Syyid

ADDISON–WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Library of Congress Catalog-in-Publication Data

Henning, Michi
Advanced CORBA® Programming with C++ / Michi Henning, Steve Vinoski.
p. cm. — (Addison-Wesley professional computing series)
Includes bibliographical references and index.
ISBN 0-201-37927-9
1. C++ (Computer program language) 2. CORBA (Computer architecture)
I. Vinoski, Steve. II. Title. III. Series.
QA76.73.C153 H458 1999
005.13’3—dc21

98–49077
CIP

ISBN 0-201-37927-9
Text printed on recycled and acid-free paper.
5 6 7 8 9 10—CRS—0302010099
Fifth printing, April 2001

Preface

Once upon a time, in the land of fruits and nuts, there was a grad student named Doug. Doug thought writing software that worked across computer networks was really cool. He loved to see how fast he could make his programs run, and the blinking lights on the network hubs were like little strobes that kept time with the heavy metal music drowning out the computer lab's air conditioning fans. As fun as this all was, Doug ran into three problems:

1. Trying to rearrange his code to try out new combinations of processes, threads, and communications mechanisms was very tedious. It was very boring and error prone to rip the code apart, mix the pieces around, and try to put it all back together.
2. Doug was spending so much time wrestling with the subtle little quirks of network programming that it was taxing the patience of his girlfriend and his Ph.D. advisor.
3. Doug's work was so successful, he kept getting new systems to work with, and each one was a little different (well, some were a lot different) than the one before.

All this started to be a real pain in the neck, so 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 templatize 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 different

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. The lights went blinkety-blink all the time. Doug could spend more time at Muscle Beach with his girlfriend. Life was good again. And thus was born ACE - the ADAPTIVE Communication Environment.

Now at this point, Doug could have grabbed his Ph.D. and some venture funding and taken the world of class libraries by storm. Then we'd all be paying serious money for the use of ACE, and that'd be it. But Doug wasn't just out to make a pile of money. Nope. World domination was the goal, and everyone (well, apparently not everyone) knows that the best way to get people to go your way is not to force them, but to seduce them. And software engineers (well, many anyway) are seduced by being able to see the source code, make it better, and see their name up in proverbial lights. So Doug took ACE down the Open Source road, before Open Source was cool. In the Open Source model of development, the source code is freely (as in freedom, and often, as in this case, in price) available. If it doesn't work right, you can fix it. If you want to add some new functionality, you can. If you want to know how to use it correctly, read the source. Many people like to read books instead, though, so here we are.

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. You can contribute fixes and additions to ACE (and get your name "up in lights" in the THANKS file!). If you'd like more information on Open Source Software in general, please visit The Open Source Page on the web at http://www.opensource.org/.

ACE's Benefits

The story about Doug in the previous section is real, and Doug's problems reflect lessons learned by all of us who've written networking programs – especially portable, high-performance networking programs:

- Multi-platform, portable 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.
- Many system-provided APIs were designed for C. Most APIs are written in terms of C API bindings, because C is common, and the API is useable from a variety of languages. If you're considering working with ACE, you've already committed to using C++ though, and the little tricks which seemed so slick with C (like 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 hard 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.

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 man pages, the ACE toolkit is designed using a layered architecture composed of the following layers:

- **OS Adaptation Layer.** The OS abstraction layer provides wrapper functions for most common system-level operations. This layer provides a common base of system functions across all of the platforms ACE has been ported to. Where the native platform does not provide the desired function, it is emulated if possible; if the functionality 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 [3]. 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. Applica-

tions 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 [2] [5]. Frameworks emphasize the integration and collaboration of application-specific and application-independent components. By doing this, frameworks enable larger-scale reuse of software compared with 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 semi-complete 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, reuseable 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 there is usually more than one way to do any given task in ACE, depending on your needs and design constraints. This book often show multiple ways to perform the same task using different layers in ACE. Is this correct? Do we talk bottom up or top down?

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 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 template classes. You should also have been exposed to basic OS 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 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. There are many source code examples to 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 (NOTE: this is a proposal and may be changed). Do we have a CD? The included CD-ROM also includes a copy of ACE's source kit, installable versions of ACE pre-built for a number of popular platforms, and complete reference for the classes we discuss in this book, and a few other, more esoteric ones.

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

- Chapter 1 describes common features and information you will need to better understand ACE and its use, as well as making your programs easier to develop and maintain.
- Chapter 2 describes how to build ACE, how to include ACE in your applications's source code, and how to use ACE's portable build scheme for building your own applications.
- Chapter 3 describes ACE's facilities for logging information at run time and for tracing program execution.
- Chapter 6 shows how to use ACE's TCP/IP Socket wrapper facades for basic TCP/IP networking.
- Chapter 7 explains how to react to events from many sources, including I/O, timers, and signals.
- Chapter 8 explains asynchronous I/O and how to perform multiple I/O operations at once using a single thread.
- Chapter 9 gives an overview of the I/O wrapper facades ACE provides in addition to the Socket wrappers.
- Appendix A contains a compatibility matrix that shows the availability of ACE features by platform. The vast majority of ACE is completely portable across platforms, but some platforms do not provide the functionality necessary for all features. This appendix provides a valuable, easy to use way to make sure that your designs will work on all of your intended platforms.
- Appendix B shows how to port ACE to a new platform.

Conventions Used in This Book

All ACE classes begin with `ACE_`. Sometimes we refer to patterns, instead of the classes they implement, without the prefix. An example is the Reactor pattern, implemented by the `ACE_Reactor` class.

All 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.

All method parameters are prefixed with '_'

The typical stuff about fonts & such

Acknowledgments

We are very grateful to the following people who provided very helpful comments on manuscript drafts: Alain Decamps, John Fowler, Kelly F. Hickel, Robert Kindred, Sven Köster, Jaroslaw Nozderko, Wojtek Pilorz, Edward Thompson.

Part I

ACE Basics

Chapter 1
Introduction to ACE

ACE is a rich and powerful toolkit. To help you make the best use of it, this chapter covers some foundational concepts, including the differences between class libraries, patterns, and frameworks. Then we'll cover one of ACE's less glamorous but equally useful aspects—its facilities for smoothing out the differences between different operating systems, hardware architectures, C++ compilers, and C++ run-time support environments. It is important to understand these aspects of ACE and get used to seeing them used before we dive into the rest of ACE's capabilities.

1.1 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 stems 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 makes it hard 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 micro-architectures.

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 micro-architectural elements of frameworks 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 class libraries in the following ways:

- *Frameworks define "semi-complete" 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 like 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 run-time.* 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 like Reactor and Observer. The "inversion of control" in the run-time 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 utilize 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 utilize class libraries to perform basic tasks such 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 semi-complete applications. ACE, in its classes and frameworks, implements many useful design patterns.

1.2 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 due 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." [1] 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 which 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 between 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 where 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.

Data type differences are a common area of porting difficulty that experienced multi-platform 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 below in the discussion of compilers.

1.2.1 The OS Adaptation Layer

ACE's OS adaptation layer provides the lowest level of functionality and forms the basis of ACE's very wide range of portability. This layer uses the Wrapper Facade [2]and Façade [1] patterns to shield you from platform differences. The Wrapper pattern forms a relatively simple wrapper around a function, and ACE uses it to unify the programming interfaces for common system functions where small differences in APIs and semantics are smoothed over. The Façade 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` function creates a new thread with a caller-specified set of attributes (scheduling policy, priority, state, etc.). The native system calls to do all that's required during thread creation vary widely between platforms in form, semantics, and the combination and order of calls needed. The Façade pattern allows the presentation of one consistent interface across all platforms to which ACE has been ported.

For relatively complex and obviously non-portable actions such as creating a thread, you would of course think to use the ACE OS functions (well, at least until you read about the higher level ACE classes and frameworks later in the book). But what about other functions which are often taken for granted, such as `printf()` and `fseek()`? Even when you need to perform some 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, since 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 of 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 which 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` man page.

As you might imagine, there is quite a lot of conditionally compiled code in ACE, and that is especially true 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 them 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 they may conflict with another vendor's macros. Rather than use a complicated combination of ACE-defined and vendor-supplied macros, a set of ACE-defined macros is used extensively, and is sometimes augmented by vendor-supplied macros, though this is relatively rare. The setting of all the ACE-defined compile-time macros is done in one file, `ace/config.h` (we'll look at this file in Chapter 2). There are many varieties of this configuration file supplied with ACE, matching all of the platforms ACE has been ported to. If you obtain a pre-built version of ACE for installation in your environment, you most likely will never need to read or change this file. But sometime when you're up for some adventure, read through it anyway for a sampling of the range of features and issues ACE handles for you.

1.3 Smoothing the Differences Between C++ Compilers

We bet you're wondering why it's so important to smooth over the differences between compilers since the C++ standard has finally settled down. There are a number of reasons.

- Compiler vendors are at different 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 just broken and ACE works around the problems.

There are many items of compiler capability (and disability) adjustment used to build ACE properly, but from a usage point of view, there are three primary areas of compiler differences which ACE helps you work with, or around:

1. Templates, both use and instantiation
2. Data types, both hardware and compiler
3. Run-time initialization and shutdown

1.3.1 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 will introduce you to class templates. If templates are a new feature to you, we suggest 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 [6]*[1]. 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 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 both in your project, and across projects you will be able to share code with. For example, let's say 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, 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. This class could be written as:

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

template <class T>

void max_tracker::track_this (const T val)
{
  if (val > this->max_val_)
    this->max_val_ = val;
  return;
}
```

Then when you want to track the maximum temperature of your home in integer degrees, you declare and use one of these objects:

```
max_tracker<int>  temperature_tracker;

// Get a new temperature reading and track it
```

1. See chapter 13 for a discussion of templates.

```
int temp_now = Thermometer.get_temp ();
temperature_tracker.track_this (temp_now);
```

The use of the template class would cause the compiler to generate code for the `max_tracker` class, substituting the `int` type for `T` in the class template. You could also declare a `max_tracker<float>` object and the compiler would instantiate another set of code to use floating point values. This magic of instantiating class template code brings us to the first area ACE helps you with—getting the needed classes instantiated correctly.

1.3.2 Template Instantiation

Many modern C++ compilers automatically "remember" which template instantiations your code needs and generate them as a part of the final link phase. In these cases, you're all set and don't need to do anything else. Or are you? What if you need to port your system to another platform where the compiler isn't so accommodating? Or maybe your compiler is really slow when doing its magical template auto-instantiation and you'd like to control it yourself, speeding up the build process? There are a variety of directives and methods that various compilers provide to specify template instantiation. While the intricacies of preprocessors and compilers make it extremely difficult to provide one simple statement for specifying instantiation, ACE provides a boilerplate set of source lines you should use to control explicit template instantiation. The best way to explain it is to show its use. In the example code above, this code would be added at the end of the source file:

```
#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)
template class max_tracker<int>;
#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)
#pragma instantiate max_tracker<int>
#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

The above code covers the compilers ACE has been ported to which customarily use explicit template instantiation for one reason or another. If you have more than one class to instantiate, you add more `template class` and `#pragma instantiate` lines to cover them.

1.3.3 Use of Defined Types in Classes That Are Template Arguments

A number of ACE classes define types as traits of that class. For example, the `ACE_SOCK_Stream` class defines `ACE_INET_Addr` as the `PEER_ADDR` type.

The other stream-abstracting classes also define a PEER_ADDR type internally to abstract the type of addressing information needed for that class. This use of traits will be covered in more detail in Chapter 4. For now, though, view it as an example of this problem: if a template class can be used with one of these stream-type classes as a template argument, the template class may very well need to know and use the addressing type defined in the stream class. You'd think that it could just be accessed as if the class were used by itself. But not so—some compilers disallow that. ACE works around the limitation and keeps your code compiler-neutral by defining a set of macros for the cases in ACE where the information is required together. In the case of the ACE_SOCK_Stream class's usage, when the addressing type is also needed, ACE defines ACE_SOCK_STREAM, which expands to include the addressing type for compilers that don't support typedefs in classes used as template arguments. It expands to just ACE_SOCK_Stream for compilers that do support it. There are a many uses of this tactic in ACE. They will all be noted for you at the points in the book where their use comes up.

1.3.4 Data Types

Programs you must port to multiple platforms need a way to avoid implicit reliance on the hardware size of compiler types and the relationships between them. For example, on some hardware/compiler combinations, an `int` is 16 bits, some it is 32 bits[2]. On some platforms, a `long int` is the same as an `int`, which is the same length as a pointer, and on others they're all different sizes. ACE provides a set of type definitions for use where the size of a data value really matters, and it must be the same on all platforms your code runs on. They are listed in Table 1.1 on page 9, with some other definitions that are useful in multiple platform development. If you use these types, your code will use properly-sized types on all ACE-supported platforms.

2. It wouldn't surprise us if there were other possible values too, like 8 bits or 64 bits. The point is, don't count on a particular size.

Table 1.1. Data types defined by ACE

`ACE_INT16, ACE_UINT16`	16 bit integer, signed and unsigned
`ACE_INT32, ACE_UINT32`	32 bit integer, signed and unsigned
`ACE_UINT64`	64 bit unsigned integer; on platforms without native 64-bit integer support, ACE provides a class to emulate it.
`ACE_SIZEOF_CHAR`	Number of bytes in a character
`ACE_SIZEOF_WCHAR`	Number of bytes in a wide character
`ACE_SIZEOF_SHORT`	Number of bytes in a `short int`
`ACE_SIZEOF_INT`	Number of bytes in an `int`
`ACE_SIZEOF_LONG`	Number of bytes in a `long int`
`ACE_SIZEOF_LONG_LONG`	Number of bytes in a `long long int`. On platforms without native long long support, this is 8, and `ACE_UINT64` is `ACE_U_LongLong`.
`ACE_SIZEOF_VOID_P`	Number of bytes in a pointer
`ACE_SIZEOF_FLOAT`	Number of bytes in a `float`
`ACE_SIZEOF_DOUBLE`	Number of bytes in a `double`
`ACE_SIZEOF_LONG_DOUBLE`	Number of bytes in a `long double`
`ACE_BYTE_ORDER`	Has a value of either `ACE_BIG_ENDIAN` or `ACE_LITTLE_ENDIAN`

1.3.5 Run-time Initialization and Rundown

One particular area of platform difference and incompatibility is run-time initialization of objects, and the associated destruction of those objects at program rundown. This is particularly true when multiple threads are involved since there is no compiler-added access serialization to the time-honored method of automatic initialization and destruction of run-time data—static objects. So remember this ACE saying: *Statics are Evil!* Fortunately, ACE provides a very portable solution to this problem in the form of three related classes:

ACE_Object_Manager, ACE_Cleanup, and ACE_Singleton (a thread-safe implementation of the Singleton pattern [1]).

- ACE_Object_Manager. As its name implies, this class manages objects. There is a single instance of this class in the ACE library and when initialized, it instantiates a set of ACE objects that are needed to properly support the ACE internals; it also destroys them at rundown time. ACE programs can make use of the Object Manager by registering objects that must be destroyed. The Object Manager destroys all of the objects registered with it at rundown time, in reverse order of their registration.
- ACE_Cleanup. This class provides the interface that ACE_Object_Manager uses to manage object lifecycles. Each object to be registered with the Object Manager must be derived from ACE_Cleanup.
- ACE_Singleton. The Singleton pattern is used to provide a single instance of an object to a program and provide a global way to access it. It is sort of like a global object in that it's not hidden from any part of the program. However, the instantiation and deletion of the object is under control of your program, not the platform's run-time environment. ACE's singleton class also adds thread safety to the basic Singleton pattern using the Double-Checked Locking pattern [1].The double-checked lock insures that only one thread initializes the object in a multithreaded system.

The two most common ways to be sure that your objects are properly cleaned up at program rundown are described below. Which one you use depends on the situation. If you want to create a number of objects of the same class and be sure that each is cleaned up, use the ACE_Cleanup method. If you want one instance of your object accessible using the Singleton pattern, use the ACE_Singleton method.

ACE_Cleanup Method

To be able to create a number of objects of a class and have them be cleaned up by the ACE Object Manager, derive your class from ACE_Cleanup. That class has a cleanup() method that the Object Manager calls to do the cleanup. To tell the Object Manager about your object (so it knows to clean it up) you make one simple call (probably from your object's constructor):

```
ACE_Object_Manager::at_exit (this);
```

ACE_Singleton Method

Use the `ACE_Singleton` class to create a single instance of your object (and only one). It uses the Adapter pattern [1] to turn any ordinary class into a singleton optimized with the Double-Checked Locking optimization pattern. `ACE_Singleton` is a template class with this definition:

```
template <class TYPE, class ACE_LOCK>
class ACE_Singleton : public ACE_Cleanup
```

`TYPE` is name of the class you are turning into a singleton. `ACE_LOCK` is the type of lock the Double-Checked Locking pattern implementation uses to serialize checking when the object needs to be created. For code that operates in a multi-threaded environment, you should use `ACE_Recursive_Thread_Mutex`. If you are writing code without threads, you should use `ACE_Null_Mutex`. That provides the proper interface but does not actually lock anything. This would be useful if your main factor in choosing the Singleton pattern is to provide the "instantiate when needed" semantics of the Singleton as well as having the Object Manager clean up the object. As an example, assume you have a class named `SystemController` and you want a single instance of it. You would define your class and then:

```
typedef ACE_Singleton<SystemController,
ACE_Recursive_Thread_Mutex> TheSystemController;
```

When you need access to the instance of that object, you get a pointer to it from anywhere in your system by calling

```
TheSystemController::instance().
```

Object Manager Ground Rules

As we've seen, the ACE Object Manager is quite a useful and powerful object. There are only two rules you need to remember in order to successfully make use of this facility:

1. Never call `exit()` directly.
2. Make sure the Object Manager is initialized successfully.

The first is easy. Remember all of the registrations with the Object Manager to request proper cleanup at rundown time? If you call exit, your program will terminate very directly, and without the Object Manager having a chance to clean anything up. Instead, have your main function simply do a `return` to end. If your program encounters some fatal condition inside a function call tree, either return, passing error indications, or throw an exception that your main function

will catch so it can cleanly return after cleaning up. In a pinch, call `ACE_OS::exit()` instead of the "naked" exit. That will insure the Object Manager gets a chance to clean up.

Object Manager initialization is a very important concept to understand. Once you understand it, you can often forget about it (for a number of platforms). The Object Manager is, effectively, a Singleton. Since it is created before any code has a chance to create threads, it doesn't need the same protection that Singletons generally require. And, what object would the Object Manager register with for cleanup? The cleanup chain has to end somewhere, and this is where. Remember the saying "Statics are evil!" Well, on some platforms, statics are not completely evil, and the Object Manager is actually a static object, initialized and destroyed properly by the platform's run-time environment, even when used in combination with shared libraries, some of which are possibly loaded and unloaded dynamically. On these platforms[3], you can usually ignore the rule for properly initializing the Object Manager.[4] On the others, however, the Object Manager needs to be initialized and terminated explicitly. For use cases where you are writing a regular C++ application that has the standard `int main (int argc, char *argv[])` entry point, ACE magically redefines your main function and inserts its own which instantiates the Object Manager on the run-time stack then calls your main function. When your main function returns, the Object Manager is run down as well (this is an example of why your code should not call exit—you'd bypass the Object Manager cleanup).

If your application does not have a standard main function, but needs to initialize and run down the Object Manager, you need to call two functions:

1. `ACE::init()` to initialize the Object Manager before any other ACE operations are performed.
2. `ACE::fini()` to run down the Object Manager after all of your ACE operations are complete. This call will trigger the cleanup of all objects registered with the Object Manager.

Cases where this may be necessary are Windows programs that have a WinMain function rather than the standard main function, and libraries that make use of ACE but the users of the library do not. For libraries, it is advantageous to add initialize and finalize functions to the library's API and have those functions call

3. The `config.h` file for these does *not* define `ACE_HAS_NONSTATIC_OBJECT_MANAGER`.
4. No, Windows is definitely not one of those platforms.

`ACE::init()` and `ACE::fini()`, respectively. In case you're writing a Windows DLL and thinking you can make the calls from `DllMain`, don't go there. It's been tried.

You can also build ACE on Windows with #define ACE_HAS_NONSTATIC_OBJECT_MANAGER 0 before including ace/config-win32.h. This removes the need to explicitly call ACE::init() and ACE::fini() but may cause problems when using dynamic services. Should this be moved to chap 2?

1.4 Using Both Narrow and Wide Characters

Developers outside the U.S. 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 subset of UCS that uses 2 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 below to use different character types without changing APIs:

`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_TEXT(str)`	Defines the string literal `str` correctly based on `ACE_USES_WCHAR`.

`ACE_TEXT_CHAR_TO_TCHAR(str)`	Converts a `char *` string to `ACE_TCHAR` format, if needed.
`ACE_TEXT_ALWAYS_CHAR(str)`	Converts an `ACE_TCHAR` string to `char *` format, if needed.

For applications to use wide characters, ACE must be built with the `ACE_HAS_WCHAR` configuration setting. 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.

ACE also supplies two string classes, `ACE_CString` and `ACE_WString`, that hold narrow and wide characters, respectively. These classes are analogous to the standard C++ `string` class, but can be configured to use custom memory allocators and are more portable. `ACE_TString` is a typedef for one of the two string types depending on the `ACE_USES_WCHAR` configuration setting.

1.5 Summary

This chapter introduced some foundational concepts and techniques that you'll need to work with ACE.

Chapter 2
How to Build ACE and Use it 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 you can not only see the source, you can 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. There are basically two types:

- Release—2 numbers (e.g., 5.3). These are stable and well-tested. Recommended for product development unless you need to add a new feature to ACE in support of your project.
- Beta—3 numbers (e.g., 5.3.4). These 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 been well-tested yet.

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". These are fixes made to a Release version. 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 (we'll assume it's 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 lots of other useful files and information. Some useful and interesting files in `ACE_wrappers` includes are:

- VERSION—Says 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—A detailed list of all changes made to ACE in reverse chronological order. The ChangeLog gets very long over time, so it's periodically moved to the ChangeLogs directory and a new ChangeLog started. Riverace writes release notes for each new ACE version which summarize the more

noteworthy changes and additions to ACE, but if you're curious about details, they're all in the ChangeLog.

- THANKS—Contains a list of 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.
- `docs`—Contains information related to various aspects of ACE such as its coding guidelines (should you wish to contribute code, it should follow these guidelines)
- `examples`—Contains many examples of how to use ACE classes and frameworks. The examples are arranged by category to make it easier to find what you need. Example code from this book is located in `examples/APG`.
- `include/makeinclude`—Contains all of the ACE build system files.
- `tests`—Contains ACE's regression test suite. If you build ACE, it's a good idea to build and run the tests as well (using the `run_test.pl` Perl script). Even if you don't need to run the tests, these programs also contain many examples of ACE usage.

You should define an environment variable named `ACE_ROOT` to reference the full path to the top-level `ACE_wrappers` directory. You'll need this environment variable when building both ACE and your applications.

2.3 How to Build ACE

Why would you want to (re)build ACE?

- ACE has been ported to many platforms; there may not be a prebuilt kit 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. Allows selection of debugging capabilities, to build with threads or not, etc. This file is not used with Microsoft Visual C++ since all command, feature, and option settings are contained in the Visual C++ project files.

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

The procedure for building ACE is:

1. Obtain the ACE source kit, if you don't already have it. You can get the kit from http://www.riverace.com or viahttp://deuce.doc.wustl.edu/Download.html.
2. The kit comes in a few formats, most commonly a gzip'd 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`. It 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"
   ```

   Table 2.1 lists the platform-specific configuration files for popular platforms. The full set of files is contained in `$ACE_ROOT/ace`.
5. (Skip this step if using Microsoft Visual C++) Create the platform_macros.GNU file in `$ACE_ROOT/include/makeinclude/`

`platform_macros.GNU`. It must include one of the platform-specific files supplied by ACE. For example, the following could be used for building on Linux:

```
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

Note that this is a GNU make directive, not a C++ preprocessor directive, so there's no leading '#' and use of an environment variable is legal.

Table 2.1 lists the platform-specific macros files for popular platforms. The full set of files is contained in `$ACE_ROOT/include/makeinclude`.

6. Build ACE. The method depends on your compiler.
   - Microsoft Visual C++ 6—Start Visual C++ and open the workspace file `$ACE_ROOT/ace/ace.dsw`, select the desired configuration, and build.
   - Microsoft Visual C++ .NET—Start Visual C++ and open the solution file `$ACE_ROOT/ace/ace.sln`, select the desired configuration, and build.
   - Borland C++Builder—Set your current directory to `$ACE_ROOT\ace`, set the BCCHOME environment variable to the home directory where C++Builder is installed; use the C++Builder make command. For example: **`make -i -f Makefile.bor -DDEBUG=1 all`**
   - All others—Set your current directory to `$ACE_ROOT/ace`, and use the GNU make command: **`make [options]`**. Table 2.2 lists the options, which can also be placed in the platform_macros.GNU file before including the platform-specific file.

Table 2.1. Configuration Files for Popular ACE Platforms

OS/Compiler	config.h file	platform_macros.GNU file
AIX 4.3/Visual Age C++	config-aix-4.x.h	platform_aix_ibm.GNU
HP-UX 11/aC++	config-hpux-11.00.h	platform_hpux_aCC.GNU
Linux	config-linux.h	platform_linux.GNU
Solaris 8/Forte 6	config-sunos5.8.h	platform_sunos5_sunc++.GNU
Windows	config-win32.h	N/A

Table 2.2. GNU make Options

Option	Description
debug=1|0	Enable or disable debugging in the built library or program. Default is enabled (1).
optimize=1|0	Turn compiler optimization on or off. Default is off (0).
buildbits=*bits*	Explicitly select, for example, 32-bit or 64-bit build target. Default is the compiler's default for the build machine. This option works for AIX, Solaris and HP-UX.
exceptions=1|0	Enable exception handling. Default is platform-specific, but usually enabled (1).
inline=1|0	Enable or disable inlining of many of ACE's methods. Default is platform-specific, but usually enabled (1).
templates=*model*	Specify how templates are instantiated. Most common values for *model* are **automatic**, the default for compilers that support it well, and **explicit**, requiring source code directives to explicitly instantiate needed templates.
static_libs=1|0	Build and use static libraries. Default is to not build static libraries (0).

2.4 How to Including ACE in Your Applications

This book describes many things you can do with ACE, and how to do them. To include ACE in your programs, there are 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 the ACE header files are located correctly, you must include $ACE_ROOT in the compiler's include search path (usually this done 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 using the -L option.

2.5 How to Build Your Applications

The build 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 on page 23.

If you have a program called `hello_ace` which 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 it 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, then linking all of 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:

```
SHLIB   = libSLD.$(SOEXT)
FILES   = Source1 Source2 Source3
LSRC    = $(addsuffix .cpp,$(FILES))
LIBS   += $(ACELIB)
BUILD   = $(VSHLIB)
#-------------------------------------------------------------
#       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.lib.GNU
include $(ACE_ROOT)/include/makeinclude/rules.local.GNU

#----------------------------------------------------------
#       Local targets
#----------------------------------------------------------
ifeq ($(shared_libs),1)
ifneq ($(SHLIB),)
CPPFLAGS     += -DSLD_BUILD_DLL
endif
endif
```

This Makefile builds libSLD.so (or whatever the shared library suffix is for the build platform) by compiling Source1.cpp, Source2.cpp, and Source3.cpp then linking them using the appropriate commands for the build platform.

The last section of the previous Makefile example conditionally adds a preprocessor macro named `SLD_BUILD_DLL` when building a shared library. This is related to the need to declare library functions as "exported" and is most needed on Windows. The next section discusses this further.

2.5.1 Import/Export Declarations and DLLs

Windows has specific rules for explicitly importing and exporting symbols in DLLs. Developers with a UNIX background may not have encountered these rules in the past, but they are important for managing symbol usage in DLLs on Windows. The rules are:

- When building a DLL, each symbol that should be visible from outside the DLL must have the declaration `__declspec (dllexport)` to specify that the symbol (or class members) are to be exported from the DLL for use by other programs or DLLs.
- When declaring the use of a symbol that your code is importing from another DLL, your declaration of the symbol (or class) must include `__declspec (dllimport)`.

Thus, depending on whether a particular symbol or class is being imported from a DLL or being built into a DLL for export to other users, the declaration of the symbol must be different. When symbol declarations reside in header files, as is

most often the case, some scheme is needed to declare the symbol correctly in either case.

ACE makes it easy to conform to these rules by supplying a script that generates the necessary import/export declarations and a set of guidelines for using them successfully. To ease porting, the following procedure can be used on all platforms that ACE runs on:

1. Select a concise mnemonic for each DLL to be built.
2. Run the `$ACE_ROOT/bin/generate_export_file.pl` Perl script, specifying the DLL's mnemonic on the command line. The script will generate a platform-independent header file and write it to the standard output. Redirect the output to a file named *`mnemonic`*`_export.h`.
3. Include the generated file in each DLL source file that declares a globally visible class or symbol.
4. To use in a class declaration, insert the keyword *`mnemonic`*`_Export` between the `class` keyword and the class name.
5. When compiling the source code for the DLL, define the macro *`mnemonic`*`_BUILD_DLL` (`SLD_BUILD_DLL` in the previous example).

Following this procedure results in the following behavior on Windows:

- Symbols decorated using the above guidelines will be declared using `__declspec (dllexport)` when built in their DLL
- When referenced from components outside the DLL, the symbols will be declared `__declspec (dllimport)`.

If you choose a separate mnemonic for each DLL and use them consistently, it will be straightforward to build and use DLLs across all OS platforms.

2.5.2 Important Notes for Microsoft Visual C++ Users

As mentioned above, the GNU make scheme is not used with Microsoft Visual C++. All needed settings are recorded in Visual C++'s project files. Because there's no common "include" file that a project can reuse settings from, each project must define the correct settings for each build configuration (debug vs. release, DLL vs. LIB, MFC vs. non-MFC). Settings important for proper ACE usage are described below. These descriptions are valid for Visual C++ version 6, and are all accessible via the Project>Settings... menu.

- C/C++ tab

- Code Generation category: Choose a multithreaded version of the run-time library.
- Preprocessor category: You should include $ACE_ROOT in the “Additional include directories” field. Some users choose to include $ACE_ROOT in the Tools>Options... menu on the Directories tab. This is acceptable if you always use one version of ACE for all projects. It is more flexible, however, to specify the ACE include path in each project.

- Link tab
  - Input category: Include the proper ACE library file in the “Object/library modules” field. Unlike the POSIX platforms, the name of the ACE library is different for various build configurations. Table 2.3 on page 25 lists the ACE library file names.
  - Input category: Include the path to the ACE library link file in the “Additional library path” field. This will generally be $ACE_ROOT/ace. If you’re linking with an ACE DLL, the export library file (.LIB) is in $ACE_ROOT/ace, and the DLL itself is in $ACE_ROOT/bin. Do not include $ACE_ROOT/bin in the “Additional library path” field. Instead, either include $ACE_ROOT/bin in your PATH environment variable, or copy the DLL file to a location that is part of the PATH.

Table 2.3. ACE Library File Names for Visual C++

Configuration	Filename
DLL Debug	`aced`
DLL Release	`ace`
Static Library Debug	`acesd`
Static Library Release	`aces`
MFC DLL Debug	`acemfcd`
MFC DLL Release	`acemfc`

Chapter 3
Using The ACE Logging Facility

Every program needs to display diagnostics of some sort: error messages, debugging output, etc. Traditionally we might use a number of `printf()` calls or `cerr` statements in our application in order to get an idea of what's going on at runtime. ACE provides us with a method of doing this while at the same time giving us great control over how much of the information is actually printed and to where it goes.

Why is it important to have a convenient way to create debug statements in the first place? 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. Such statements are useful long after an application is considered to be "bug-free". The ACE mechanisms allow us to enable and disable these statements at will at runtime. Thus, you don't have to pay for the overhead (in either cycles or disk space) under normal conditions but if a problem arises you can easily cause copious amounts of debugging information to be created to assist you in finding and fixing it. It is an unfortunate fact of our industry that many bugs will never appear until our programs are in the hands of the end-user.

In this chapter we will cover:

- Basic logging and tracing techniques including setting the "logging level"
- Customization of the logging mechanics
- How to direct the output
- Capturing log messages before they're output

- The distributed ACE logging service
- A simple Log Manager class
- Output and logging level configuration using the Logging Strategy

3.1 Basic Logging and Tracing

There are three commonly used macros 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 and 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 are LM_DEBUG and LM_ERROR. *formatting-args* is a printf()-like[1] set of format conversion operators and formatting arguments for insertion into the output.

Unlike ACE_DEBUG and ACE_ERROR, which cause output where the macro is placed, ACE_TRACE causes a one line of debug information to be printed at the point of the ACE_TRACE statement and another when its enclosing scope is exited. Therefore, placing an ACE_TRACE statement at the beginning of a function or method provides a trace of when that function or method is entered

1. One might wonder why printf-like formatting was chosen instead of the more natural (to C++ coders) cout style formatting. If the preprocessor macro ACE_NDEBUG is defined, the ACE_DEBUG macro will insert only a blank line into your code (rather than several lines of code which create output). Achieving this same optimization with cout style would have resulted in a rather odd usage: ACE_DEBUG(debug_info << "Hi Mom" << endl). Similarly, many of the tokens (such as %I) would have been awkward to implement: ACE_DEBUG(debug_info << nested_indent << "Hi Mom" << endl). One could argue-away the compile-time optimization by causing ACE_NDEBUG to put the debug output stream object into a no-op mode. That may be sufficient for some platforms but for others (such as embeded real-time systems) you really *do* want the code to simply not exist.

and exited. Since C++ doesn't have a handy way to dump a stack trace this can be very useful indeed[2].

Let's first take a look at a very simple application:

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

void foo (void);

int main(int, char *[])
{
  ACE_TRACE("main");

  ACE_DEBUG ((LM_INFO, "%IHi Mom\n"));
  foo();
  ACE_DEBUG ((LM_INFO, "%IGoodnight\n"));

  return 0;
}

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

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

Our first step is always to include the `Log_Msg.h` header file. It defines many helpful macros (including `ACE_DEBUG` and `ACE_ERROR`) to make your life easier. The set of output-producing macros is listed in Table 3.1. Most of these macros resolve at compile time to either empty air or very useful lines of code depending on the compile-time value of three configuration settings, `ACE_NTRACE`, `ACE_NDEBUG` and `ACE_NLOGGING`. This allows you to sprinkle your code with as little or as much debug information as you want and then turn it on or off when compiling.

`ACE_NTRACE`, `ACE_NDEBUG` and `ACE_NLOGGING` are used to disable the debug and logging macros. By “disable” we mean that if `ACE_NLOGGING` is

2. Be mindful of the fact that ACE_TRACE output is conditional on both ACE logging output at the LM_TRACE level and ACE_NDEBUG being defined as 0 (turn off no debugging) when the ACE_TRACE-using code is compiled.

defined the ACE_DEBUG macro, for instance, will resolve to a no-op. Table 3.1. lists the various macros and how each are disabled.

ACE_DEBUG is very much like good old printf(). You can use ACE_DEBUG to print just about any arbitrary string you want and there are many format characters (listed in Table 3.2) you can use to enhance your debug output. In the case above we've used %I so that the ACE_DEBUG messages are nicely indented along with the ACE_TRACE messages.

If you compile and execute the code above you should get something very much like this:

```
(1024) calling main in file `listing05-1.cpp' on line 7
   Hi Mom
   (1024) calling foo in file `listing05-1.cpp' on line 18
      Howdy Pardner
   (1024) leaving foo
   Goodnight
(1024) leaving main
```

Consider this slightly modified code:

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

void foo(void);

int main(int, char *[])
{
  ACE_TRACE("main");

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

  foo ();

  ACE_DEBUG ((LM_DEBUG, "%IGoodnight\n"));

  return 0;
}

void foo(void)
{
```

```
  ACE_TRACE ("foo");

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

and the output it produces:

```
(1024) calling main in file `listing05-2.cpp' on line 7
      Howdy Pardner
   Goodnight
```

In our second example we've used `ACE_LOG_MSG->priority_mask()` to change the level of logging produced. `ACE_LOG_MSG` is a shortcut for obtaining the pointer to a thread-specific singleton instance of the `ACE_Log_Msg` class that implements ACE's logging functionality. ACE maintains one `ACE_Log_Msg` instance for each thread spawned. Although you can instantiate more of these objects if you like for specialized purposes, the `ACE_TRACE` and `ACE_DEBUG` macros always use the thread-specific singleton obtained via `ACE_LOG_MSG`. By invoking methods on an `ACE_Log_Msg` instance, we can change its behavior in several ways.

We can use the `ACE_Log_Msg::priority_mask()` method to set the bits for the logging levels we desire (all of the available logging levels are listed inTable 3.3). The bits can be set at either a process-wide or instance-specific level (since ACE maintains an instance for each of your threads, this is often referred to as the "thread-specific" level). This can be handy if you're writing a multithreaded application since you may have a worker thread that you really don't care to see debug information for. By default, all bits are set at the process level and none at the instance level. That's why we specified the `ACE_Log_Msg::PROCESS` parameter to our call of `priority_mask()`. Of course you can use any message level at any time. By default all types are printed. However, take care to specify a reasonable level for each of your messages and then at runtime you can use the priority_mask() method to enable or disable messages you're interested in. This allows you to easily "over-instrument" your code and then enable just the things that are useful at any point in time.

If you want to set your logging level on a per-thread basis you might do something like this:

```
ACE_LOG_MSG->priority_mask (0, ACE_Log_Msg::PROCESS);

ACE_LOG_MSG->priority_mask (LM_DEBUG | LM_NOTICE,
                            ACE_Log_Msg::THREAD);
```

In addition to their simple usage (e.g. - %s) the format specifiers can also handle the standard printf modifiers for precision, width, justification, etc. (e.g. - %5.3s). For more detail on these please consult your system documentation.

3.1.1 Log_Msg Details

ACE_Log_Msg has a rich set of operators for recording the current "state" of your application. Table 3.4 summarizes some of the more commonly used functions. Most methods have both accessor and mutator signatures (e.g. -- *int op_status(void)* and *void op_status(int status)*)

.

Table 3.1. ACE Logging Macros

Macro	Function	Disabled By
`ACE_ASSERT(test)`	Very much like the assert() library call. If the test fails an assertion message including the filename and line number along with the test itself will be printed and the application aborted.	`NDEBUG`
`ACE_HEX_DUMP(( level, buffer, size [, text ] ))`	Dumps the buffer as a string of hex digits. The optional `text' parameter will be printed prior to the hex string if provided. The op_status[a] is set to 0.	`NLOGGING`
`ACE_RETURN(value)`	No message is printed, the calling function returns with `value'. op_status is set to ‘value’.	`NLOGGING`
`ACE_ERROR_RETURN( (level, string, ...), value)`	Logs the string at the requested level. The calling function then returns with `value'. op_status is set to ‘value’.	`NLOGGING`
`ACE_ERROR((level, string, ...))`	Sets the op_status to -1 and logs the string at the requested level.	`NLOGGING`

Table 3.1. ACE Logging Macros

`ACE_DEBUG((level, string, ...))`	Sets the op_status to 0 and logs the string at the requested level.	`NLOGGING`
`ACE_ERROR_INIT( value, flags )`	Sets the op_status to `value' and the logger's option flags to `flags'. The flags are defined below.	`NLOGGING`
`ACE_ERROR_BREAK(( level, string, ...))`	Invokes ACE_ERROR() followed by a `break'. Use this to display an error message and exit a while or for loop for instance.	`NLOGGING`
`ACE_TRACE(string)`	Displays the filename, line number and ‘string’ where ACE_TRACE appears. Displays “Leaving ‘string’” when the ACE_TRACE-enclosing scope exits.	`NTRACE`

a. Many of the macros in the above table refer to *op_status*. This is an internal variable used to keep the logging framework aware of the program state (e.g. -- the “operation status”). By convention, a value of ‘0’ indicates good. Anything else is considered an error or exception state.

Table 3.2. ACE_Log_Msg Formatting Specifiers

A	print an ACE_timer_t value (which could be either double or ACE_UINT32.)
a	abort the program at this point abruptly.
c	consumes and prints a character.
C	print a character string.
d	consumes an `int` and formats it as a decimal number.
I	indents output according to the nesting depth (obtained from ACE_Trace::get_nesting_indent)
G	consumes and prints a double.
l	substitutes the line number where an error occurred.
M	substitutes the name of the message priority.
m	substitutes the message corresponding to errno value, e.g., as done by strerror()
N	substitutes the file name where the error occurred.
n	substitutes the name of the program (or “” if not set)
o	consumes an `int` and formats it as an octal number.
P	substitutes the current process id.
p	consumes a char * and prints a text string followed by the appropriate errno message from sys_errlist, e.g., as done by perror()
Q	consumes an `ACE_UINT64` and formats it as a decimal number.
r	consumes a pointer and uses it as a function pointer; calls the function.
R	substitutes the return status.
S	consumes an `int`, treating it as a signal number; formats the signal name.
s	consumes a char *, displays a character string.

Table 3.2. ACE_Log_Msg Formatting Specifiers (Continued)

T	substitutes the current time in hour:minute:sec.usec format.
D	substitutes a timestamp in month/day/year hour:minute:sec.usec format.
t	substitutes the calling thread's id (1 if single-threaded).
u	consumes an int, formats it as unsigned int.
w	consumes a wchar_t, prints a wide character.
W	consumes a wchar_t *print a wide character string.
x	consumes an int, print as a hex number
@	consumes a void *, formats the pointer in hexadecimal
%	substitutes a single percent sign, "%"

Table 3.3. ACE_Log_Msg Logging Severity Levels

LM_TRACE	Messages indicating function-calling sequence.
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.
LM_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LM_EMERGENCY	A panic condition. This is normally broadcast to all users.

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	Filename in which the message was generated.
msg	A message to be sent to the log output target.
inc	Increment nesting depth. Returns the previous value.
dec	Decrement 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 thread. The tracing status of the ACE_LOG_MSG singleton determines whether or not an ACE_Trace instance's constructor/destructor provide generates log messages.
priority_mask	Get/Set the set of priorities (thread or process level) for which messages will be logged.
log_priority_enabled	Return true if the requested priority is enabled.
set	Used by ACE_ASSERT to set the line number, filename, op_status and several other characteristics all at once.
conditional_set	Used by a number of the logging macros to conditionally set the line number, filename, etc... if the requested priority is enabled.

3.2 Customizing the ACE Logging Macros

In most cases people will use the standard ACE tracing and logging macros shown above. However, there may be times when there is a need to customize their behavior. Or you might want to create wrapper macros in anticipation of future customization.

3.2.1 Wrapping ACE_DEBUG

There may come a time when you want to ensure that all of your LM_DEBUG messages contain a particular text string so that you can easily grep for them in your output file. Or maybe you want to ensure that every one of them is prefixed with the handy "%I" string so that they indent properly. If you lay the groundwork at the beginning of your project and encourage your coders to use your macros then it will be very easy to implement these kinds of things in the future.

Here are a set of macro definitions that wrap the ACE_DEBUG() macro in a handy way. Notice how we've guaranteed that every message will be properly indented and we've prefixed each message to make for easy grep'ing of the output.

```
#define DEBUG_PREFIX       "DEBUG%I"
#define INFO_PREFIX        "INFO%I"
#define NOTICE_PREFIX      "NOTICE%I"
#define WARNING_PREFIX     "WARNING%I"
#define ERROR_PREFIX       "ERROR%I"
#define CRITICAL_PREFIX    "CRITICAL%I"
#define ALERT_PREFIX       "ALERT%I"
#define EMERGENCY_PREFIX   "EMERGENCY%I"

#define DEBUG(FMT, ...)        \
        ACE_DEBUG(( LM_DEBUG,  \
                    DEBUG_PREFIX FMT \
                    __VA_ARGS__))
#define INFO(FMT, ...)        \
        ACE_DEBUG(( LM_INFO,  \
                    INFO_PREFIX FMT \
                    __VA_ARGS__))
#define NOTICE(FMT, ...)        \
        ACE_DEBUG(( LM_NOTICE,  \
                    NOTICE_PREFIX FMT \
                    __VA_ARGS__))
#define WARNING(FMT, ...)        \
        ACE_DEBUG(( LM_WARNING,  \
                    WARNING_PREFIX FMT \
```

```
                    __VA_ARGS__))
#define ERROR(FMT, ...)        \
        ACE_DEBUG(( LM_ERROR,  \
                    ERROR_PREFIX FMT \
                    __VA_ARGS__))
#define CRITICAL(FMT, ...)        \
        ACE_DEBUG(( LM_CRITICAL,  \
                    CRITICAL_PREFIX FMT \
                    __VA_ARGS__))
#define ALERT(FMT, ...)        \
        ACE_DEBUG(( LM_ALERT,  \
                    ALERT_PREFIX FMT \
                    __VA_ARGS__))
#define EMERGENCY(FMT, ...)        \
        ACE_DEBUG(( LM_EMERGENCY,  \
                    EMERGENCY_PREFIX FMT \
                    __VA_ARGS__))
```

Of course it would be more useful if each of our prefixes was surrounded by an #ifdef to allow them to be overridden but we leave that as an exercise to the reader.

Using these macros instead of the usual ACE_DEBUG macros is, as expected, easy to do:

```
#include "Log_Msg.h"

void foo (void);

int main(int, char *[])
{
  ACE_TRACE("main");

  DEBUG ("Hi Mom\n");

  foo ();

  DEBUG ("Goodnight\n");

  return 0;
}

void foo (void)
{
```

```
  ACE_TRACE ("foo");

  DEBUG ("Howdy Pardner\n");
}
```

Our output is nicely indented and prefixed as requested:

```
(1024) calling main in file `listing10-1.cpp' on line 7
DEBUG   Hi Mom
   (1024) calling foo in file `listing10-1.cpp' on line 20
DEBUG      Howdy Pardner
   (1024) leaving foo
DEBUG   Goodnight
(1024) leaving main
```

The __VA_ARGS__ trick works fine for the GNU preprocessor but may not be available everywhere else so be sure to read your compiler's documentation before committing yourself to this particular approach. If there isn't something similar available to you then there's a slightly less elegant approach:

```
#define DEBUG       LM_DEBUG,     "DEBUG%I"
#define INFO        LM_INFO,      "INFO%I"
#define NOTICE      LM_NOTICE,    "NOTICE%I"
#define WARNING     LM_WARNING,   "WARNING%I"
#define ERROR       LM_ERROR,     "ERROR%I"
#define CRITICAL    LM_CRITICAL,  "CRITICAL%I"
#define ALERT       LM_ALERT,     "ALERT%I"
#define EMERGENCY   LM_EMERGENCY, "EMERGENCY%I"
```

Which could be used something like this:

```
  ACE_DEBUG ((DEBUG "Hi Mom\n"));

  ACE_DEBUG ((DEBUG "Goodnight\n"));
```

and produces exactly the same output at the expense of slightly less attractive code.

3.2.2 ACE_Trace

Let's look now at creating a variant of ACE_TRACE() that will display the line number at which a function exits. The default ACE_Trace object implementation

doesn't do this and doesn't provide an easy way for us to extend it so, unfortunately, we have to create our own object from scratch. However, we can cut and paste from the ACE_Trace implementation in order to give ourselves a head-start. Consider this simple object:

```
class Trace
{
public:
  Trace (const ACE_TCHAR *prefix,
         const ACE_TCHAR *name,
         int line,
         const ACE_TCHAR *file)
    {
      this->prefix_ = prefix;
      this->name_   = name;
      this->line_   = line;
      this->file_   = file;

      ACE_Log_Msg *lm = ACE_LOG_MSG;
      if (lm->tracing_enabled ()
          && lm->trace_active () == 0)
        {
          lm->trace_active (1);
          ACE_DEBUG ((LM_TRACE,
                      ACE_TEXT ("%s%*s(%t) calling %s in file `%s'
")
                      ACE_TEXT (" on line %d\n"),
                      this->prefix_,
                      Trace::nesting_indent_ * lm->inc (),
                      ACE_LIB_TEXT (""),
                      this->name_,
                      this->file_,
                      this->line_));
          lm->trace_active (0);
        }
    }

  void setLine (int line)
    {
      this->line_ = line;
    }

  ~Trace (void)
    {
      ACE_Log_Msg *lm = ACE_LOG_MSG;
```

```
      if (lm->tracing_enabled ()
          && lm->trace_active () == 0)
        {
          lm->trace_active (1);
          ACE_DEBUG ((LM_TRACE,
                      ACE_TEXT ("%s%*s(%t) leaving %s in file `%s'
")
                      ACE_TEXT (" on line %d\n"),
                      this->prefix_,
                      Trace::nesting_indent_ * lm->dec (),
                      ACE_LIB_TEXT (""),
                      this->name_,
                      this->file_,
                      this->line_));
          lm->trace_active (0);
        }
    }

private:
  enum { nesting_indent_ = 3 };

  const ACE_TCHAR *prefix_;
  const ACE_TCHAR *name_;
  const ACE_TCHAR *file_;
  int line_;
};
```

Trace is a very simplified version of ACE_Trace. Since our focus is printing a modified function exit message we chose to leave out some of the more esoteric ACE_Trace functionality. We did, however, include a prefix parameter to the constructor so that each entry/exit message can be prefixed (before indentation) if you want. In an ideal world you would simply use the priority_mask() method of ACE_Log_Msg to select the messages you're interested in. On the other hand, if you're asked to do a post-mortem analysis of a massive, all-debug-enabled logfile the prefixes can be quite handy.

With our new Trace object available to us we can now create a set of very simple macros that will use this new object to implement function tracing in our code:

```
#define TRACE_PREFIX       "TRACE "
#if (ACE_NTRACE == 1)
#    define TRACE(X)
#    define TRACE_RETURN(V)
```

```
#else
#    define TRACE(X)                              \
             Trace ____ (TRACE_PREFIX,             \
                         ACE_LIB_TEXT (X),         \
                         __LINE__,                 \
                         ACE_LIB_TEXT (__FILE__))

#    define TRACE_RETURN(V)                       \
             do { ____.setLine(__LINE__); return V; } while (0)

#    define TRACE_RETURN_VOID()                   \
             do { ____.setLine(__LINE__); } while (0)
#endif
```

The addition of the two TRACE_RETURN* macros is how our Trace object's destructor will know to print the line number at which the function exits. Each of these uses the convenient setLine() method to set the current line number before allowing the Trace instance to go out of scope, destruct and print our message. A simple example of using our new object:

```
#include "Log_Msg.h"

void foo (void);

int main(int, char *[])
{
  TRACE ("main");

  DEBUG ("Hi Mom\n");

  foo ();

  DEBUG ("Goodnight\n");

  TRACE_RETURN (0);
}

void foo (void)
{
  TRACE ("foo");
```

```
  DEBUG ("Howdy Pardner\n");

  TRACE_RETURN_VOID ();
}
```

And the output it produces:

```
TRACE (1024) calling main in file `listing10-2.cpp' on line 7
DEBUG   Hi Mom
TRACE    (1024) calling foo in file `listing10-2.cpp' on line 20
DEBUG      Howdy Pardner
TRACE    (1024) leaving foo in file `listing10-2.cpp' on line 24
DEBUG   Goodnight
TRACE (1024) leaving main in file `listing10-2.cpp' on line 15
```

Although the output is a bit wordy, we were successful in our original intent of printing the line number with at which each function returns. While that may seem like a small thing for a trivial program consider the fact that very few useful programs are actually trivial. If you are trying to understand the flow of a legacy application it may very well be worth your time to liberally instrument it with TRACE() and TRACE_RETURN() macros to get a feel for the paths taken. Of course, training yourself to use TRACE_RETURN() may take some time but in the end you will have a much better idea of how the code flows.

3.3 Selecting the Output Destination

In this section we'll discuss output to three common and expected targets: the standard error channel (or stderr), the system logger (or syslog) and a programmer-specified output stream. Anyone familiar with Unix will know these immediately and anyone else will have likely heard of them. In addition we will discuss techniques for directing our output to an ostream object.

3.3.1 STDERR

Output to stderr/cerr is so common that it is, in fact, the default target for all ACE logging, tracing and debugging. Our examples to date have taken advantage of this. There may be times when you want to direct your output not only to stderr but to one of the other targets available to you. In these cases you will have to explicitly include stderr in your choices:

```
int main(int, char *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, "%IHi Mom\n"));

  ACE_LOG_MSG->set_flags (ACE_Log_Msg::STDERR);

  foo ();
```

If you choose this second approach, it may be necessary to invoke clr_flags() to disable any other output destinations. Everything after the set_flags() will be redirected to STDERR until you invoke clr_flags() to prevent it.

3.3.2 syslog

Most modern operating systems support the notion of a “system logger”. The implementation details range from a library of function calls to network daemon. The general idea is that all applications direct their logging activity to the syslog which will, in turn, direct it to the correct file or files. The system administrator has the opportunity to configure syslog such that different “classes” and “levels” of logging get directed to different destinations. Such an approach provides a good combination of scalability and configureability.
To use the system logger you would do something like this:

```
int main (int, char *argv[])
{
  ACE_LOG_MSG->open (argv[0], ACE_Log_Msg::SYSLOG, "syslogTest");
```

Although one would think we could use the `set_flag()` method to enable syslog output after the `ACE_Log_Msg` instance has been opened, unfortunately that isn't the case. Likewise, if you want to quit sending output to syslog a simple `clr_flag()` won't do the trick. In order to communicate with the system logger, `ACE_Log_Msg` must perform a set of initialization procedures that are only done in the `open()` method. Part of the initialization requires the program

name that will be recorded in syslog (this is the 3rd argument above). If we don't do this when our program starts then we will have to do it later in order to get the behavior we expect from invoking `set_flags()`. In a similar manner, the `open()` method will properly close down any existing connection to the system logger if it is invoked without the `ACE_Log_Msg::SYSLOG` flag.

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

void foo (void);

int main(int, char *argv[])
{
  // This will be directed to stderr (the default ACE_Log_Msg
  // behavior).

  ACE_TRACE ("main");

  ACE_DEBUG ((LM_DEBUG, "%IHi Mom\n"));

  // Everything from foo() will be directed to the system logger

  ACE_LOG_MSG->open (argv[0], ACE_Log_Msg::SYSLOG, "syslogTest");

  foo ();

  // Now we reset the log output to default (stderr)

  ACE_LOG_MSG->open (argv[0]);

  ACE_DEBUG ((LM_INFO, "%IGoodnight\n"));

  return 0;
}

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

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

Although it may seem strange to invoke `ACE_LOG_MSG->open()` more than once in your application, there is really nothing at all wrong with it. Think of it as

more of a `reopen()'. Before we end this chapter we will create a simple LogManager class to help hide some of these kinds of details.

The syslog facility has its own associated configuration details regarding its idea of logging facilities which are different from ACE's logging severity levels. The default log facility for ACE's syslog backend is LOG_USER. This value can be changed at compile time by changing the ACE_DEFAULT_SYSLOG_FACILITY config.h setting. Please consult the syslog man page for details on how to configure the logging destination for the specified facility.

3.3.3 ostream

ostreams are the preferred way to handle output to files (and other targets) in C++. They provide enhanced functionality over the printf family of functions and actually make for more readable code. The msg_ostream() method of ACE_Log_Msg lets us provide an ostream on which the logger will write our information.

```
ACE_LOG_MSG->msg_ostream (output, 1);
ACE_LOG_MSG->set_flags (ACE_Log_Msg::OSTREAM);
ACE_LOG_MSG->clr_flags (ACE_Log_Msg::STDERR);
```

Even though we specify a cout-style ostream on which the log output will be written we still use printf-style formatters in our ACE_DEBUG messages for the reasons discussed on page 28.

Note that it's perfectly safe to select OSTREAM as output (either via open() or set_flags()) and then generate output before you invoke msg_ostream(). If you do so, the output will simply disappear because there is no ostream assigned. Also notice that we have used the two-argument version of msg_ostream(). This not only sets the ostream for the Log_Msg instance to use, it also tells the Log_Msg whether or not to assume ownership of the pointer. In our example we have specified `1' indicating that the Log_Msg should assume ownership and delete the ostream instance when the Log_Msg is deleted. The single-argument version of msg_ostream() doesn't specify its default behavior with regard to ownership so it pays to be explicit in your wishes.

3.3.4 All Together

We can now easily combine all of these techniques and distribute our logging information among all three choices:

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

int main(int, char *argv[])
{
  ACE_LOG_MSG->open (argv[0]);

  // Output to default destination (stderr)

  ACE_TRACE ("main");

  ACE_OSTREAM_TYPE *output =
        new std::ofstream ("ostream.output.test");

  ACE_DEBUG ((LM_DEBUG, "%IThis will go to STDERR\n"));

  ACE_LOG_MSG->open (argv[0], ACE_Log_Msg::SYSLOG, "syslogTest");
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::STDERR);

  ACE_DEBUG ((LM_DEBUG, "%IThis will go to STDERR & syslog\n"));

  ACE_LOG_MSG->msg_ostream (output, 0);
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::OSTREAM);

  ACE_DEBUG ((LM_DEBUG,
              "%IThis will go to STDERR, syslog & an ostream\n"));

  ACE_LOG_MSG->clr_flags (ACE_Log_Msg::OSTREAM);

  delete output;

  return 0;
}
```

Beware of a subtle bug waiting to get you when you use an ostream. Notice that before we deleted the ostream instance *output* we first cleared the OSTREAM flag on the Log_Msg instance. Remember that the ACE_TRACE(“main”) still has to write its final message when the trace instance goes out of scope at the end of main(). If we delete the ostream without removing the OSTREAM flag then the Log_Msg will dutifully attempt to write that final message on a deleted ostream instance and your program will most likely crash.

3.4 Using Callbacks

To this point we've been content to give our logging output to ACE_DEBUG and ACE_TRACE and let them direct it to whatever destination we've chosen. For most cases that will be just fine. What if, though, we want to do something with that output ourselves? Is there a way we can inspect or even modify the logging output before it reaches its final destination? Of course there is. That's where ACE_Log_Msg_Callback comes in.

Using the callback object is quite easy. The msg_callback() method will allow you to register a callback object with an ACE_Log_Msg instance. You will still need to use set_flags() to tell the instance to direct 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 the log() method of ACE_Log_Msg is invoked. As it turns out, that is exactly what happens when ACE_TRACE and ACE_DEBUG are used.

There are some important caveats to remember when using the callback approach. These are documented in ACE_Log_Msg_Callback.h but bear repeating here: ACE_LOG_MSG is a shorthand access to the ACE_Log_Msg singleton which is actually a per-thread singleton, not a per-process singleton. Singletons, per-thread and per-process, will be covered later in the text. What is important to remember now is that if you register a callback object in one thread it won't be used by any other thread in your application. Also, the callback object will not be inherited by any threads you create. So, if you're going to be using callback objects with multi-threaded applications you need to take special care that each thread is given its appropriate callback instance.

Also, like the OSTREAM caveat, be sure that you don't delete a callback instance that might still be used by the Log_Msg instance.
And now a simple Log_Msg_Callback implementation:

```
#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
{
public:
  void log (ACE_Log_Record &log_record)
    {
```

```
      log_record.print ("", 0, cerr);
      log_record.print ("", ACE_Log_Msg::VERBOSE, cerr);
    }
};
```

And the program that uses it:

```
#include "ace/Log_Msg.h"
#include "Callback-1.h"

int main(int, char *[])
{
  Callback *callback = new Callback;

  ACE_LOG_MSG->set_flags (ACE_Log_Msg::MSG_CALLBACK);
  ACE_LOG_MSG->clr_flags (ACE_Log_Msg::STDERR);
  ACE_LOG_MSG->msg_callback (callback);

  ACE_TRACE ("main");

  ACE_DEBUG ((LM_DEBUG, "%IHi Mom\n"));

  ACE_DEBUG ((LM_INFO, "%IGoodnight\n"));

  return 0;
}
```

And the output it creates:

```
(1024) calling main in file `listing20-1.cpp' on line 12
Apr  6 15:28:23.606 2003@@15859@LM_TRACE@(1024) calling main in fi
le `listing20-1.cpp' on line 12
   Hi Mom
Apr  6 15:28:23.607 2003@@15859@LM_DEBUG@   Hi Mom
   Goodnight
Apr  6 15:28:23.607 2003@@15859@LM_INFO@   Goodnight
(1024) leaving main
Apr  6 15:28:23.608 2003@@15859@LM_TRACE@(1024) leaving main
```

The first log_record.print() simply prints the message we've always seen. The second, however, uses the VERBOSE flag to provide much more information. Both direct their output to stderr/cerr.

Not very exciting really but once you have access to the Log_Record instance you have control to do anything you want. Let's take a look at a bit more of the information contained in the Log_Record:

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

class Callback : public ACE_Log_Msg_Callback
{
public:
  void log (ACE_Log_Record &log_record)
    {
      cerr << "Log Message Received:" << endl;

      cerr << "\tType:        " <<
        ACE_Log_Record::priority_name
           (ACE_Log_Priority (log_record.type ())) <<
        endl;

      cerr << "\tLength:      " << log_record.length () << endl;

      const time_t epoch = log_record.time_stamp ().sec ();
      cerr << "\tTime_Stamp:  " << ACE_OS::ctime (&epoch)
           << flush;

      cerr << "\tPid:         " << log_record.pid () << endl;

      ACE_TCHAR *data =
            ACE_OS_String::strdup (log_record.msg_data ());

      ACE_OS_String::memmove (
            (void *)log_record.msg_data (),
            ">> ", 3);

      ACE_OS_String::memmove (
            (void *)(log_record.msg_data () + 3),
                           data,
                           ACE_OS::strlen (data) + 1);
      delete data;

      cerr << "\tMsgData:     " << log_record.msg_data ()
           << endl;
    }
};
```

And the output it creates:

```
Log Message Received:
        Type:        LM_TRACE
        Length:      88
        Time_Stamp:  Sun Apr  6 15:28:23 2003
        Pid:         15866
        MsgData:     >> (1024) calling main in file `listing20-2.c
pp' on line 12

Log Message Received:
        Type:        LM_DEBUG
        Length:      40
        Time_Stamp:  Sun Apr  6 15:28:23 2003
        Pid:         15866
        MsgData:     >>    Hi Mom

Log Message Received:
        Type:        LM_INFO
        Length:      40
        Time_Stamp:  Sun Apr  6 15:28:23 2003
        Pid:         15866
        MsgData:     >>    Goodnight

Log Message Received:
        Type:        LM_TRACE
        Length:      48
        Time_Stamp:  Sun Apr  6 15:28:23 2003
        Pid:         15866
        MsgData:     >> (1024) leaving main
```

As you can see, we have quite a bit of access to the Log_Record internals. We're not limited to changing just the message text. We can, in fact, change any of the values we want. Whether or not that makes any sense is up to your application. Table 3.5 lists the attributes of Log_Record and what they mean.

Table 3.5. Log_Record Attributes

Attribute	Description
type	The log record type from Table 3.3
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 which created the log record instance
msg_data	The textual message of the log record
msg_data_len	Length of the msg_data attribute

3.5 The Logging Client and Server Daemons

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

The 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 actual server with a huge number of connections from clients 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 can do a number of very interesting and useful things. All of that will be covered later in

the book and 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 right now or wait and read it later if you want.
To start the server you need to first create a file *server.conf* with the following content:

```
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"
```

Note that the lines above are wrapped for readability. Your *server.conf* should contain only two lines, each beginning with the word *dynamic*. The first line defines the *logging strategy* to write the log output to standard error and the output stream attached to a file named *foobar*. It also requests verbose log messages instead of a more terse format. The logging strategy section later in this chapter discusses more ways to use this service.

The second line of *server.conf* causes the server to listen for client connections at TCP port 20009[3] on all network interfaces available on your computer.
You can now start the server with:

```
$ACE_ROOT/netsvcs/servers/main -f server.conf
```

The next step is to create the configuration file for the client proxy and start it. The file should look something like this:

```
dynamic Client_Logging_Service Service_Object * netsvcs:_make_ACE_
Client_Logging_Acceptor() active "-p 20009 -h localhost"
```

Again, that's all on one line. The important parts are *-p 20009* which tells the proxy which TCP port the server will be listening to (this should match the *-p* value in your *server.conf*) and *-h localhost* that sets the hostname where the logging server is executing. For our simple test we are executing both client and server on the same system. In the real world you will most likely have to change *localhost* to the name of your real logging server.

3. There is nothing particularly magic about the port 20009. However, there are a standard set of ports typically used by the ACE examples and tests. Throughout this text we have tried to maintain consistency with that set.

Although we provide the port on which the server is listening, we did not provide a port value for clients of the proxy. This value is known as the "logger key" and defaults to *localhost:20012*. If you want your client proxy to listen at a different address you an specify that with the *-k* parameter in your client.conf. You can now start the client proxy with:

```
$ACE_ROOT/netsvcs/servers/main -f client.conf
```

Using the logging service in one of our previous examples is trivial:

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

int main(int, char *argv[])
{
  ACE_LOG_MSG->open (argv[0],
                     ACE_Log_Msg::LOGGER,
                     ACE_TEXT_CHAR_TO_TCHAR(
               ACE_DEFAULT_LOGGER_KEY));

  ACE_TRACE ("main");

  ACE_DEBUG ((LM_DEBUG, "%IHi Mom\n"));

  ACE_DEBUG ((LM_INFO, "%IGoodnight\n"));

  return 0;
}
```

Like the syslog example, we must use the `open()` method when we want to use the logging service; `set_flags()` isn't sufficient. Notice also the `open()` parameter `ACE_TEXT_CHAR_TO_TCHAR (ACE_DEFAULT_LOGGER_KEY)`. This is nothing more than *localhost:20012* with appropriate casts to make the compiler happy.

By design, the client proxy will only listen for connections at localhost:20012[4]. Remember that *localhost* is not a public network interface, only applications running on your physical system can access it. By specifically listening only on the *localhost* interface prevents remote applications from connecting to the local client proxy. This may seem strange at first but it really

4. In fact, if your platform supports stream pipes (such as some versions of Unix) they will be used instead of a TCP socket.

makes sense. If you allow a bunch of remote applications to connect to your local client proxy then you're defeating the purpose of the distributed design.

Of course if you have a really good reason for listening on a public interface you can use the *-k* parameter in your *client.conf* something like this:

```
dynamic ... “-p 20009 -k myhost.com:20012”
```

To summarize: On every machine on which you want to use the logging service you must execute an instance of the client proxy. Each of these is configured to connect to a single instance of the logging server somewhere on your network. Then, of course, you execute that server instance on the appropriate system.

For the truly adventurous your application can actually communicate directly with the logging server instance. This approach has two problems: 1) your program is now more complicated because of the connection & logging logic and 2) you run the risk of overloading the server instance because you've removed the scaling afforded by the client proxies.
However, if you still want your application to talk directly to the logging server, here's a way to do so:

```
#include "ace/Log_Msg.h"
#include "Callback-3.h"

int main (int, char *[])
{
  Callback *callback = new Callback;

  ACE_LOG_MSG->set_flags (ACE_Log_Msg::MSG_CALLBACK);
  ACE_LOG_MSG->clr_flags (ACE_Log_Msg::STDERR);
  ACE_LOG_MSG->msg_callback (callback);

  ACE_TRACE ("main");

  ACE_DEBUG ((LM_DEBUG, "%IHi Mom\n"));

  ACE_DEBUG ((LM_INFO, "%IGoodnight\n"));

  return 0;
}
```

This looks very much like our previous callback example. We use the callback hook to capture the Log_Record instance that contains our message. Our new Callback object then sends that to the logging server:

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

#define LOGGER_PORT 20009

class Callback : public ACE_Log_Msg_Callback
{
public:
  Callback ()
    {
      this->logger_ = new ACE_SOCK_Stream;
      ACE_SOCK_Connector connector;
      ACE_INET_Addr addr (LOGGER_PORT, ACE_DEFAULT_SERVER_HOST);

      if (connector.connect (*(this->logger_), addr) == -1)
        {
          delete this->logger_;
          this->logger_ = 0;
        }
    }

  virtual ~Callback ()
    {
      if (this->logger_)
        {
          this->logger_->close ();
        }
      delete this->logger_;
    }

  void log (ACE_Log_Record &log_record)
    {
      if (!this->logger_)
        {
          log_record.print ("", ACE_Log_Msg::VERBOSE, cerr);
          return;
        }

      size_t len = log_record.length();
      log_record.encode ();
```

```
      if (this->logger_->send_n ((char *) &log_record, len) == -1)
        {
          delete this->logger_;
          this->logger_ = 0;
        }
    }

private:
  ACE_SOCK_Stream *logger_;
};
```

We've introduced some things here that you won't read about for a bit but you should have a rough idea of what we're doing. The gist of it is that the callback object's constructor opens a socket to the logging service. The log() method then sends the Log_Record instance to the server via the socket. Since several of the Log_Record's attributes are numeric, we must use the encode() method to ensure that they are in a network-neutral format before sending them. This will prevent much confusion in the case where the host executing your application has different byte-ordering than the host executing your logging server.

3.6 A LogManager Class

In the sections above we've learned how to direct the logging output to several places. We even discovered that you can change your mind at runtime and direct it 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 char *prog_name = "");
  void redirectToSyslog (const char *prog_name = "");
  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 main(int, char *[])
{
  LOG_MANAGER->redirectToStderr ();

  ACE_TRACE ("main");

  LOG_MANAGER->redirectToSyslog ();

  ACE_DEBUG ((LM_INFO, "%IHi Mom\n"));

  foo ();

  LOG_MANAGER->redirectToDaemon ();

  ACE_DEBUG ((LM_INFO, "%IGoodnight\n"));

  return 0;
}

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

  LOG_MANAGER->redirectToFile ("output.test");

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

“But wait”, you say. “Where did LOG_MANAGER come from?” In a future chapter we will learn about the magic of the ACE_Singleton template. That's what we're using behind LOG_MANAGER. In fact, we've already used a singleton in the other examples where we invoked ACE_LOG_MSG. All will be revealed later, for now it is enough to know that a singleton gives you quick access to a

single instance of an object anywhere in your application. To declare our singleton we add to our header file:

```
typedef ACE_Singleton<LogManager, ACE_Null_Mutex> LogManagerSingle
ton;
#define LOG_MANAGER LogManagerSingleton::instance()
```

and to define it we must add to our cpp file:

```
#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)
 template class ACE_Singleton<LogManager, ACE_Null_Mutex>;
#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)
 pragma instantiate ACE_Singleton<LogManager, ACE_Null_Mutex>;
#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

Our log manager implementation is a straight-forward application of the things we learned earlier in this chapter.

```
LogManager::LogManager ()
  : log_stream_ (0), output_stream_ (0)
{
}

LogManager::~LogManager ()
{
  if (log_stream_)
    log_stream_->close ();
  delete log_stream_;
}

void LogManager::redirectToSyslog (const char *prog_name)
{
  ACE_LOG_MSG->open (prog_name, ACE_Log_Msg::SYSLOG, prog_name);
}

void LogManager::redirectToDaemon (const char *prog_name)
{
  ACE_LOG_MSG->open (prog_name, ACE_Log_Msg::LOGGER,
                     ACE_TEXT_CHAR_TO_TCHAR (ACE_DEFAULT_LOGGER_KE
Y));
}

void LogManager::redirectToOStream (ACE_OSTREAM_TYPE *output)
{
  output_stream_ = output;
```

```
  ACE_LOG_MSG->msg_ostream (this->output_stream_);

  ACE_LOG_MSG->clr_flags (ACE_Log_Msg::STDERR | ACE_Log_Msg::LOGGE
R);
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::OSTREAM);
}

void LogManager::redirectToFile (const char *filename)
{
  log_stream_ = new std::ofstream ();
  log_stream_->open (filename , ios::out | ios::app);
  this->redirectToOStream (log_stream_);
}

void LogManager::redirectToStderr (void)
{
  ACE_LOG_MSG->clr_flags (ACE_Log_Msg::OSTREAM | ACE_Log_Msg::LOGG
ER);
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::STDERR);
}

ACE_Log_Msg_Callback *redirectToCallback (ACE_Log_Msg_Callback * c
allback)
{
  ACE_Log_Msg_Callback *previous =
    ACE_LOG_MSG->msg_callback (callback);

    if (callback == 0)
      {
        ACE_LOG_MSG->clr_flags (ACE_Log_Msg::MSG_CALLBACK);
      }
    else
      {
        ACE_LOG_MSG->set_flags (ACE_Log_Msg::MSG_CALLBACK);
      }

    return previous;
}
```

It's primary limitation is the assumption that output will only go to one place at a time. For our trivial examples that may be sufficient but for a real application it could be a problem. Modifying the LogManager to overcome this should be a fairly easy task and we leave that to the reader.

3.7 Using the Logging Strategy

Thus far, all of our choices about what to log and where to send the output have been done “in code”. For small applications that may be sufficient but once an application is in use by many people it is no longer reasonable to require a recompile to change the logging options. We could, of course, provide parameters or a configuration file to our application but then we 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 Manager 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[5] and will be covered later in the text.

The following sample application uses the file above:

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

5. 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!

```
    ACE_DEBUG ((LM_INFO, "%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 alternate using *-f someFile*. In either case, the file's content would be something like what is shown above which tells the logging service to configure ACE_LOG_MSG to direct the output to both stderr and the file log.out.

Be careful that you open the service config object as shown above rather than with the default parameters. If the final parameter is not `1' then the open() method will restore the logging flags to their pre-open values. Since the logging services 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.

Note that by default all logging levels are enabled at a process-wide level. If you specify *-p INFO* in your config file you will probably be surprised when you get other logging levels also. To get what you want be sure to use the disable flags (e.g. -- ~INFO) as well.

One of the most powerful features of the logging strategy is the ability to rotate the application's logfiles when they reach a specified size. Use the *-m* parameter to set the size and the *-N* parameter to set the maximum number of files to keep. Authors of long-running applications will appreciate this since it will go a long way towards preventing rampant disk consumption.

f	Pass in the flags (such as OSTREAM, STDERR, LOGGER, VERBOSE, SILENT, VERBOSE_LITE) used to control logging.
i	The interval (in seconds) at which the logfile size is sampled (default is 0, i.e., do not sample by default).
k	Set the logging key.
m	The maximum logfile size in Kbytes.
n	Set the program name for the %n format specifier.
N	The maximum number of logfiles that we want created.

o	Specifies that we want the standard logfiles ordering (fastest processing in). Default is not to order logfiles.
p	Pass in the process-wide priorities to either enable (e.g., DEBUG, INFO, WARNING, NOTICE, ERROR, CRITICAL, ALERT, EMERGENCY) or to disable (e.g., ~DEBUG, ~INFO, ~WARNING, ~NOTICE, ~ERROR, ~CRITICAL, ~ALERT, ~EMERGENCY).
s	Specify the filename used when OSTREAM is specified as an output target.
t	Pass in the thread-wide priorities to either enable (e.g., DEBUG, INFO, WARNING, NOTICE, ERROR, CRITICAL, ALERT, EMERGENCY) or to disable (e.g., ~DEBUG, ~INFO, ~WARNING, ~NOTICE, ~ERROR, ~CRITICAL, ~ALERT, ~EMERGENCY).
w	Cause the logfile to be wiped out, both on startup and on reconfiguration.

The possible values for -p and -t are the same as those used by ACE_Log_Msg without the LM_ prefix. Any can be prefixed with ~ to omit that log level from the output. Multiple flags can be OR'd (|) together as needed.

3.8 Conclusion

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 Run-Time Information

Most applications offer ways for users to direct or alter their run-time behavior. Two of the most common approaches are:

- Accepting command-line arguments and options. These are often used for information that can reasonably change on each application invocation. For example, the host name to connect to during an FTP 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 ports a server should listen on, or whether or not to enable various logging severity levels.

Any information that can reasonably change at run time should be made available to an application at run time—not built into the application itself. This allows the information to change without having to rebuild and redistribute the application. In this chapter, we'll 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. It 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. `ACE_Get_Opt` is also easier to use than `getopt()` since the option definition string and argument vector are only passed once to the constructor, rather than to each iterator call.

`ACE_Get_Opt` can parse two kinds of options:

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

For example, the following code processes command line options for the HAStatus program. The HAStatus program offers a 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. It illustrates the two things 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 the option requires an argument. In the example above, **`-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()` (or the `get_opt()` method) 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 `opt_arg()` method. `operator()` returns `EOF` when all the options have been processed.

When processing the `argv` elements, `ACE_Get_Opt` takes an option's argument from the remaining characters of the current `argv` element or the next `argv` element as needed. Optional arguments, however, must be in the same element as the option character. The behavior when a required argument is missing depends on the first character of the short options definition string. If it is a `':'` (as in our example), `get_opt()` returns a `':'` when a required argument is missing. Otherwise, it returns `'?'`.

Short options that don't take arguments can be grouped together on the command line after the leading **`-`**, but in that case, only the last short option in the group can take an argument. A `'?'` is returned if the short option is not recognized.

Since short options are defined as integers, long options that wouldn't normally have a meaningful short option equivalent can designate non-alphanumeric values for the corresponding short option. These non-alphanumerics cannot appear in the argument list or in short options definition string, but can be returned and processed efficiently in a `switch` statement. The following two lines of code could be added to the previous example. They illustrate two ways to register a long option without a corresponding short option.

```
cmd_opts.long_option (ACE_TEXT ("cool_option"));
cmd_opts.long_option (ACE_TEXT ("the_answer"), 42);
```

The first call to `long_option()` adds a long option **`--cool_option`** that will cause a 0 to be returned from `get_opt()` if **`--cool_option`** is spec-

ified on the command line. The second is similar, but specifies that the integer value 42 will be returned from `get_opt()` when **`--the_answer`** is found on the command line. The following shows the additions that would be made to the switch block in the example on page 68:

```
case 0:
  ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("Yes, very cool.\n")));
  break;

case 42:
  ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("the_answer is 42\n")));
  break;
```

When the user supplies long options on the command line, each one can be abbreviated as long it is unambiguous. Therefore, **`--cool_option`** could be abbreviated as short as **`--coo`** (any shorter would also match **`--config`**).

If an `argv` element of **`--`** is encountered, it signifies the end of the option section and `get_opt()` returns `EOF`. If the `opt_ind()` method returns a value that's less than the number of command line elements (`argc`), there are more elements that haven't been parsed.

That's the basic use case; however, `ACE_Get_Opt` can do a lot more. The extended capabilities are accessed by specifying values for the defaulted arguments in the constructor. The complete signature for the constructor is:

```
ACE_Get_Opt (int argc,
             ACE_TCHAR **argv,
             const ACE_TCHAR *optstring,
             int skip_args = 1,
             int report_errors = 0,
             int ordering = PERMUTE_ARGS,
             int long_only = 0);
```

Start parsing at an arbitrary index

`ACE_Get_Opt` can be directed to start processing the argument vector at an arbitrary point specified by the `skip_args` parameter. The default value is 1, which causes `ACE_Get_Opt` to skip `argv[0]` (traditionally, the program name) when parsing a command line passed to `main()`. When using `ACE_Get_Opt` to parse options received when initializing a dynamic service (dynamic services are discussed in Chapter 19) `skip_args` is often specified as 0, because arguments passed to services initialized via the ACE Service Configurator framework start in `argv[0]`. `skip_args` can also be set to any other value that's less than the

value of `argc` to skip previously processed arguments, or arguments that are already known.

Report errors while parsing

By default, `ACE_Get_Opt` is silent about parsing errors; it simply returns the appropriate value from `operator()` (or the `get_opt()` method), allowing your application to handle and report errors in the most sensible way. If, however, you'd rather have `ACE_Get_Opt` display an error message when it detects an error in the specified argument vector, the constructor's `report_errors` argument should be non-zero. In this case, `ACE_Get_Opt` will use `ACE_ERROR` with the `LM_ERROR` severity to report the error. See Chapter 3 for a discussion of ACE's logging facility, including the `ACE_ERROR` macro.

Alternate long option specification

If `"W;"` is included in the options definitions string, `ACE_Get_Opt` treats **`-W`** similarly to **`--`**. For example, **`-W foo`** will be parsed the same as **`--foo`**. This can be useful when manipulating argument vectors to change parameters into long options by inserting an element with **`-W`** instead of inserting **`--`** on an existing element.

Long options only

If the `long_only` parameter to the `ACE_Get_Opt` constructor is non-zero, command line tokens that begin with a single `-` are checked as long options. For example, in the program on page 68, if the `long_only` argument were set to 1, the user could type either **`--config`** or **`-config`**.

4.1.1 Understanding Argument Ordering

Some applications require you to specify all options at the beginning of the command line, while others allow you to mix options and other non-option tokens (such as file names). `ACE_Get_Opt` supports selection of use cases defined by enumerators defined in `ACE_Get_Opt`. One of these values can be passed as the constructor's `ordering` parameter. It accepts the following values:

- `ACE_Get_Opt::PERMUTE_ARGS`: As the argument vector is parsed, the elements are dynamically rearranged so that those with valid options (and their arguments) appear at the front of the argument vector, in their original relative ordering. Non-option elements are placed after the option elements. They can be processed by some other part of your system, or processed as

known non-options (e.g., file names). When `operator()` returns `EOF` to indicate the end of options, `opt_ind()` returns the index to the first non-option element in the argument vector. This is the default ordering mode.

- `ACE_Get_Opt::REQUIRE_ORDER`: The argument vector is not reordered and all options and their arguments must be at the front of the argument vector. If a non-option element is encountered, `operator()` returns `EOF`; `opt_ind()` returns the index of the non-option element.
- `ACE_Get_Opt::RETURN_IN_ORDER`: The argument vector is not reordered. Any non-option element causes `operator()` to return 1 and the actual element is accessible via the `opt_arg()` method. This mode is useful for situations in which options and other arguments can be specified in any order and in which the relative ordering makes a difference. This is a situation where it may be useful to parse options, examine non-options, and continue parsing after the non-options using the `skip_args` argument to specify the new starting point.

As mentioned above, the argument ordering can be changed by specifying one of the above enumerators for the `ACE_Get_Opt` constructor's `ordering` parameter. However, the argument ordering can also be changed using two other mechanisms. Specifying a value for the constructor takes least precedence. The other two methods both override the constructor value, and are listed below in increasing order of precedence.

1. If the POSIXLY_CORRECT environment variable is set, the ordering mode is set to `REQUIRE_ORDER`.
2. A `+` or `-` character at the beginning of the options string. `+` changes the ordering mode to `REQUIRE_ORDER`; `-` changes it to `RETURN_IN_ORDER`. If both are at the start of the options string, the last one is used.

4.2 How to Access Configuration Information

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

- `ACE_Configuration_Heap` is available on all platforms. It 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.
- `ACE_Configuration_Win32Registry` is available only on Windows. It 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 below uses the `config_file` command-line argument read in the example on page 68. 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 needs to be imported from the configuration file. We'll look at configuration storage in Section 4.2.2.

Because our example application keeps the settings for each subsystem in a separate section, it opens the HAStatus section. The ListenPort value is read and used to set the TCP port number in the `listen_addr_` member variable.

4.2.1 Configuration Sections

Configuration data is organized hierarchically in sections, analogous to a file-system directory tree. Each configuration object contains a *root section* that has no name, similar to the filesystem root in UNIX. All other sections are created hierarchically beneath the root section and are named by the application. Sections can be nested to an arbitrary depth.

4.2.2 Configuration Backing Stores

The `ACE_Configuration_Win32Registry` class accesses the Windows registry directly. Therefore, it acts as a wrapper around the Windows API, so Windows manages the data and all access to it. Although it is possible to use a memory-mapped allocation strategy with `ACE_Configuration_Heap`, the resultant file contents are the in-memory format of the configuration and not a human-readable form. Therefore, configuration information is usually saved in a file. ACE offers two classes for importing data from and exporting data to a file:

1. `ACE_Registry_ImpExp` uses a text format that includes type information with each value. This allows type information to be preserved across export/import, even on machines with different byte orders. This is the class used in the previous example to import configuration data from the configuration file specified on the program's command line.

2. `ACE_Ini_ImpExp` uses the older Windows "INI" file format which does not have type information associated with the values. Therefore, configuration data exported using `ACE_Ini_ImpExp` is always imported as string data, regardless of the original type.

Both classes use text files; however, they are not interchangeable. Therefore, you should choose a format and use it consistently. It is usually best to use `ACE_Registry_ImpExp` when possible because it retains type information. `ACE_Ini_ImpExp` is most useful when your application must read existing .INI files over which you have no control.

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 it is necessary, though, 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 argc/argv argument vector in order to use `ACE_Get_Opt`. `ACE_ARGV` is a good class for this use.

Let's say that a program obtains its options from a string wants to parse the string 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 argc, ACE_TCHAR *argv[])
{
  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 above, the value of the HOSTNAME environment variable is substituted where $HOST-NAME 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. One shortcoming in this feature is that it only substitutes when an environment variable name is present by itself in a token. For example, if the cmdline literal above 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 example above also uses the skip_args parameter on the ACE_Get_Opt constructor. Whereas the argument vector passed to the main() program entrypoint 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.

Chapter 5
ACE Containers

Robust container classes are one of the most useful tools one can obtain from a toolkit. Although today the Standard Template Library (STL) with it's 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 completely 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 and it was had to be dropped from the ACE distribution. In its place, the ACE developers created a separate set of containers that are used internally within the library and are also exported for client development use. Although these containers are not as elegant as the STL containers they provide high performance and in many cases have a footprint much smaller then 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 due to footprint issues
- You need to use ACE's special purpose memory allocators (described in Chapter 17) due to performance or predictability issues

This chapter will briefly review container concepts and then go on to discuss the various template-based containers available in ACE, including both sequence type containers and associative containers. The object containers in ACE are special purpose for example message queues, timer data structures etc. and are covered in various chapters of this book. Finally, we round up the chapter with a discussion on some of the allocators that are available with ACE that plug right into the containers.

5.1 Container Concepts

Let's start this chapter off with a brief review of a few container concepts that are applicable to C++. General-purpose containers can be designed and built using different 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 programmer 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 different 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 are containers that use the C++ templates facility. This facility 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 only allow insertion of People objects into the list.

This is in sharp contrast to the C way of creating reusable lists of typeless pointers (i.e., 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 (e.g., in Java the java.lang.Object type is often used). This allows you to insert any subtype into the container. This means that you could conceptually have a single list that included both People and Car objects. This is probably not desired and can occur accidently. Such errors are usually

determined at run time when you use object containers. Typeless containers offer even less error protection because any type can usually be added to the container. On the other hand when a template containers is instantiated you explicitly specify what type of objects are allowed in the container.

We briefly discussed templates and explicit template instantiation in Section 1.3.2 on page 7. If you are unfamiliar with these concepts and have not gone over them in this book it might be a good idea to refresh your memory by rereading that section. One important facet of templates that we skipped over 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. When a user requests `Dynamic_Array<void*>` she 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 such as `ACE_Hash<>` and `ACE_Equal_To<>` 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 are containers that support insertion and deletion of a class of object types. For those of you who 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 that have been built for specfic uses (such as the `ACE_Message_Queue` class). As mentioned earlier, 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 are used to iterate through a container and 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 are dependent on the type of container and iterator. Some iterators only allow forward iteration while others allow bidirectional iteration. Similarily, constant iterators allow only read access to values.

On many containers ACE provides two iterator API's: one that to a certain degree follows the C++ standard and a second that is an older ACE proprietary API.

The standard C++ library includes a set of algorithms that operate using the standard iterator types. If ACE containers supported the standard C++ iterator concept then you could use the standard C++ template algorithms with an ACE based container. Unfortunately as of now very few ACE containers support enough of the standard C++ API to be directly used with the the standard algorithms.

5.2 Sequence Containers

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

5.2.1 Double Linked List

Doubly linked lists maintain both forward and reverse links within the sequence. This allows efficient forward and reverse traversal within the sequence; however, you cannot randomly access elements. Therefore, you will find the iterator concept very handy. Lets go through an example that illustrates these features as they are provided by the double linked list in ACE, `ACE_DLList<>`.

Since `ACE_DLList<>` is a template based container we need to specify in advance what kind of elements are 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;
private:
        static int count_;
public:
        DataElement()                       {count_++;}
```

```
        DataElement(int data): data_(data) {count_++;}
        DataElement(const DataElement&e)   {data_=e.getData();coun
t_++;}
        operator=(const DataElement&e)     {data_=e.getData();}
        ~DataElement()                     {count_--;}
        int getData() const                {return data_;}
        void setData(int val)              {data_ = val;}
        static int numOfActiveObjects()    {return count_;}
private:
        int data_;
};
```

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

To make things easy let's start off by creating a convenient typedef to represent our doubly linked list of `DataElemen`t objects.

```
#include "ace/Containers.h"
#include "DataElement.h"

typedef ACE_DLList<DataElement> MyList;
//create a new type of list that can only store
//data elements
```

Next we get to our test class.

```
class ListTest
{
public:
  int run();
  //create and run list with elements on the
  //heap and the stack

  void displayList(MyList & list);
  //iterate through the list displaying
  //the elements to stdout.

  void destroyList(MyList& list);
  //destroy the elements on the list
};
```

This simple test will take in our list and will then perform insertion, deletion and iteration operations on it when the public `run()` method is called. Lets see what happens when `run()` gets called.

```
int
ListTest::run()
{
  ACE_TRACE("ListTest::run");

  MyList list1;
  //create a list

  for(int i=0; i< 100; i++)
    {
      DataElement *element;
      ACE_NEW_RETURN(element, DataElement(i), -1);
      list1.insert_tail(element);
    }
  //insert the elements

  this->displayList(list1);
  //iterate through and display to output

  MyList list2;
  list2 = list1;
  //create a copy of list1

  this->displayList(list2);
  //iterate over the copy and display to output

  this->destroyList(list2);
  //get rid of the copy list and all it's elements
  //since both lists had the *same* elements
  //this will cause list1 to contain pointers that
  //point to data elements that have already been destroyed!

  ACE_DEBUG((LM_DEBUG, "# of live objects: %d\n",
             DataElement::numOfActiveObjects()));

  //the lists themselves are destroyed here. Note that the
  //list destructor will destory copies of whatever data the
  //list contained. Since in this case the list contained
  //copies of pointers to the data elements these are the
```

```
  //only thing that gets destroyed here.

  return 0;
}
```

This method first creates `list1` and populates it with 100 data elements. Notice the way data population is done here. Even though we created the template type as `ACE_DLList<DataElement>` we are actually inserting pointers to the elements, instead of the elements themselves. That is, the values are not stored in the container only pointers to the values are. For those of you who have worked with standard C++ containers, you will find this behaviour unexpected. This means that when the list goes out of scope the data elements will still live on the heap and it is your responsiblity to ensure that you delete them or you will have a leak. Such a container is commonly refered to as a reference container as it only stores pointers to the values that you insert in it. Standard C++ containers are mostly value containers. That is, they store copies of whatever value you store in them. Most ACE containers are also value containers. `ACE_DLList` is an exception rather than the rule.

After populating the list we use the list assignment operator to create a copy of `list1` called `list2`. We iterate through `list2` just to make sure everything is correctly copied over from `list1`. Remember both containers only contain pointers to the data elements. To illustrate this we destroy all of the data elements by passing `list2` to the `destroyList()` method. As expected, since both `list1` and `list2` pointed to the same data elements, both lists will now contain invalid pointers. We use our `DataElement::numOfActiveObjects()` method to determine how many active data elements there are at this point and it dutifully reports that there are no active elements.

Let's take a look at the display method where we have used iterators to go through a provided list.

```
void
ListTest::displayList(MyList& list)
{
  ACE_TRACE("ListTest::displayList");

  ACE_DEBUG((LM_DEBUG, "Forward iteration\n"));
  ACE_DLList_Iterator<DataElement> iter(list);
  while(!iter.done())
    {
        ACE_DEBUG((LM_DEBUG, "%d:", iter.next()->getData()));
        iter++;
```

```
    }
  ACE_DEBUG((LM_DEBUG, "\n"));

  ACE_DEBUG((LM_DEBUG, "Reverse Iteration \n"));
  ACE_DLList_Reverse_Iterator<DataElement> riter(list);
  while(!riter.done())
    {
      ACE_DEBUG((LM_DEBUG, "%d:", riter.next()->getData()));
      riter++;
    }
  ACE_DEBUG((LM_DEBUG, "\n"));
}
```

Here we get our first taste of an iterator class. Remember that you can think of iterators as generalizations of the C++ pointer concept. In this case we use an iterator that starts from the beginning of the list and starts displaying each element one by one. The ACE iterator has methods that let us advance forward in the sequence (`operator++`), get the current element we are pointing to (`next()`) and determine that we have reached the end of the sequence (`done()`). The example also uses a reverse iterator that starts from the end and goes backwards to the beginning of the sequence. You will notice that not all containers support both reverse and forward iteration. For example, it makes sense to have a forward iterator in a stack or queue but reverse iteration should not be (and isn't) possible.

The iterators in ACE do not support the same interface as the standard C++ iterators (although this is one of the things on ACE's TODO list). Unfortunately we will see that ACE iterators are not as consistent as we would want and support slightly different APIs. Both these issues make it impossible for you to use the ACE container types with the standard C++ generic algorithms. You will also notice that although some containers do offer nested type definitions for their available iterators (some of them are even like standard C++) many do not.

5.2.2 Stacks

Stacks are LIFO sequences. That is, the last element inserted (pushed) is always the first one that is extracted (popped). ACE provides both dynamic and static stacks. Static stacks have a fixed size and are therefore cheaper to use. Two stacks of this type are provided: `ACE_Bounded_Stack` and `ACE_Fixed_Stack`. Dynamic stacks allocate memory on every insertion and release this memory on every extraction. `ACE_Unbounded_Stack` is of this variety.

Let's look at an example that exercises each one of these stack types, starting with an `ACE_Bounded_Stack` (that is bound to a fixed number of elements when it is created). Internally an `ACE_Bounded_Stack` is implemented as a dynamically allocated array of elements. Insertions are *O(1)* and obviously the constant value here is very small (especially if copying the element is a cheap operation such as when you are pushing pointers onto the stack).

```
int StackExample::runBoundedStack()
{
   ACE_TRACE("StackExample::runBoundedStack");

   ACE_DEBUG((LM_DEBUG, "Using a bounded stack\n"));
   ACE_Bounded_Stack<DataElement> bstack1(100);
   {
   DataElement elem[100];
   for(int i =0; i<100; i++)
      {
         elem[i].setData(i);
         bstack1.push(elem[i]);
         //push the element on the stack
      }
   }// the element array is constrained to this scope.

   ACE_Bounded_Stack<DataElement> bstack2(100);
   bstack2 = bstack1;
   //make a copy!

   for(int j =0; j<100; j++)
      {
         DataElement elem;
         bstack2.pop(elem);

         ACE_DEBUG((LM_DEBUG, "%d:",
       elem.getData()));
      }

       return 0;
}
```

A bounded stack is fixed in size at runtime. This is achieved by passing in the size as an argument to the constructor of the stack. Here we create a stack that can contain at most 100 data elements. We then create 100 data elements on the stack that are then copied into `bstack1`. Notice that the `ACE_Bounded_Stack<>`

template is more standard-like than the last sequence we looked at. Here when we specify that the stack will contain `DataElement` values, that is exactly what the stack expects. Here the stack actually creates a copy of each element that we insert into it (i.e., it is a value container). If we had created an `ACE_Bounded_Stack<DataElement*>` then only the pointers would be copied on insertion.

Next we proceed to use the assignment operator to copy `bstack1` elements into another bounded stack, `bstack2`. Note that even though the `elem[]` array and its elements are destroyed once we leave the marked scope, `bstack1` still has copies of all the destroyed elements.

We then pop and remove all elements from `bstack2`. Notice that `bstack2` contains a copy of `bstack1` therefore even at this point another copy of the elements exist in `bstack1`. However, these elements are released when the destructor for `bstack1` is called on exit from this function.

Now let's take a quick look at fixed and unbounded stacks.

```
int StackExample::runFixedStack()
{
   ACE_TRACE("StackExample::runFixedStack");

   ACE_DEBUG((LM_DEBUG,
        "\nUsing a fixed stack and a heap\n"));
   ACE_Fixed_Stack<DataElement*, 100> fstack;
   for(int k =0; k<100; k++)
      {
         DataElement* elem;
         ACE_NEW_RETURN(elem, DataElement(k), -1);
         fstack.push(elem);
         //push the element on the stack
      }

   for(int l =0; l<100; l++)
      {
         DataElement* elem;
         fstack.pop(elem);

         ACE_DEBUG((LM_DEBUG, "%d:",
       elem->getData()));

         delete elem;
      }

   return 0;
```

```
}
int StackExample::runUnboundedStack()
{
   ACE_TRACE("StackExample::runUnboundedStack");

   ACE_DEBUG((LM_DEBUG,
        "\nUsing an unbounded stack and the heap\n"));
   ACE_Unbounded_Stack<DataElement*> ustack;
   for(int m = 0; m<100; m++)
      {
         DataElement *elem;
         ACE_NEW_RETURN(elem, DataElement(m), -1);

         ustack.push(elem);
         privateStack_.push(elem);
         //push the element on the stack
      }

   //Oddly enough you can actually iterate through
   //an unbounded stack!
   //This is this way because underneath the covers
   //the unbounded stack is a linked list.

   ACE_Unbounded_Stack_Iterator<DataElement*>
        iter(ustack);
   for(iter.first();
        !iter.done(); iter.advance())
      {
         DataElement** elem;
         iter.next(elem);
         ACE_DEBUG((LM_DEBUG,
              "%d:", (*elem)->getData()));
         delete (*elem);
      }
   //This will cause the elements in the private stack to also
   //disappear!

   return 0;
}
```

The first method is similar to the previous except that it uses a fixed stack whose size was predetermined at compile time. Internally the fixed stack uses an array whose size is fixed to be the value of the second template parameter of

ACE_Fixed_Stack, this preempts the initial heap allocation that is required with a bounded stack.

To make things a little interesting, this method inserts pointers to the elements instead of the elements themselves. This makes the copies less expensive as copying a pointer typically requires a single machine instruction.

Finally we come to the ACE_Unbounded_Stack class. When you use ACE_Unbounded_Stack, you do not have to have a predetermined notion of how many elements will be inserted on the stack. Internally ACE_Unbounded_Stack uses a linked list representation and both push and pop operations are *O(1)*. An interesting side-affect of the implementation is that an unbounded stack allows you to iterate through it, although you probably should not be using the iterators.

5.2.3 Queues

Queues are FIFO sequences. That is, they allow element insertion at the tail of the sequence but elements are removed from the head. ACE provides an ACE_Unbounded_Queue class that provides this functionality. The ACE queue implementation allows insertion both at the head and tail of the queue though elements are always extracted from the head.

The next example illustrates creating elements on the stack and the heap and then putting them on our queue. Let's start by looking at value insertions on the stack.

```
int QueueExample::runStackUnboundedQueue()
{
  ACE_TRACE("QueueExample::runStackUnboundedQueue");

  ACE_Unbounded_Queue<DataElement> queue;
  int i;
  for(i=0; i<10; i++)
    {
      DataElement elem[10];
      elem[i].setData(9-i);
      queue.enqueue_head(elem[i]);
    }

  for(i=0;i<10; i++)
    {
        DataElement elem[10];
        elem[i].setData(i+10);
        queue.enqueue_tail(elem[i]);
```

```
    }

  for(ACE_Unbounded_Queue_Iterator<DataElement> iter(queue);
      !iter.done();
      iter.advance())
    {
      DataElement *elem;
      iter.next(elem);
      ACE_DEBUG((LM_DEBUG, "%d:", elem->getData()));
    }

   return 0;
}
```

This is fairly similar to the previous example. First we insert a couple of elements on the front of the queue and then another few on the tail on the queue. Note that most queue implementations only allow insertion at the tail end and not on both ends like this. Insertion either on the head or tail of the queue is an *O(1)* operation. After completing the insertions we iterate through the queue with an iterator. Since both the queue and the elements are created on the stack, they will all be released when the method returns. Although we don't illustrate it here, you can only dequeue elements from the head of a queue using the `dequeue_head()` operation.

```
int QueueExample::runHeapUnboundedQueue()
{
  ACE_TRACE("QueueExample::runHeapUnboundedQueue");

  ACE_Unbounded_Queue<DataElement*> queue;
  for(int i=0; i<20; i++)
    {
      DataElement *elem;
      ACE_NEW_RETURN(elem, DataElement(i), -1);
        queue.enqueue_head(elem);
    }

  for(ACE_Unbounded_Queue_Iterator<DataElement*> iter
        = queue.begin();
        !iter.done();
        iter.advance())
    {
      DataElement **elem;
      iter.next(elem);
```

```
      ACE_DEBUG((LM_DEBUG,
        "%d:", (*elem)->getData()));

      delete (*elem);
    }

  return 0;
}
```

In this case the elements are allocated on the heap, and we only keep pointers to these elements within the bounded queue container. As we iterate through the array for display we also delete these elements. Note that we have not dequeued them from the queue—the unbounded queue contains invalid pointers at this point. We use the `deque_head()` method to actually remove these pointers from the queue. This also serves to illustrate that queues, like stacks (and the standard C++ library containers), always copy whatever you pass into them. In this example pointers are copied into the container and are not released even though the actual elements are destroyed.

5.2.4 Arrays

Although arrays are not sequences and are supported directly by the C++ language, ACE provides a safe wrapper type that performs checked access and offers useful features such as copy and comparison semantics.

The following simple example illustrates the use of the `ACE_Array` class and its features.

```
#include "ace/Containers.h"
#include "DataElement.h"

int main(int argc, char*argv[])
{
   ACE_UNUSED_ARG(argc);
   ACE_UNUSED_ARG(argv);

   ACE_Array<DataElement*> arr(10);

   for(int i=0; i < 10; i++)
   {
      DataElement* elem;
      ACE_NEW_RETURN(elem,
        DataElement(i), -1);
```

```
      //allocate the memory
      arr[i] = elem;
   }
   //insert elements


   DataElement *elem =0;
   ACE_ASSERT(arr.set(elem, 11)==-1);
   ACE_ASSERT(arr.get(elem,11)==-1);
   //checked access

   ACE_Array<DataElement*> copy = arr;
   ACE_ASSERT(copy == arr);
   //make a copy

   ACE_Array<DataElement*>::ITERATOR
        iter(arr);
   while(!iter.done())
   {
      DataElement** data;
      iter.next(data);
      ACE_DEBUG((LM_DEBUG,
        "%d\n", (*data)->getData()));
      iter.advance();

      delete (*data);
   }
   //walk through display the elements and exit.

   return 0;
}
```

5.2.5 Sets

A set is a sequence which does not allow duplicate entries. ACE includes two different types of sets, a bounded set and an unbounded set. The bounded set is fixed in size where the unbounded set is a dynamic structure that will grow as you add elements to it. To determine whether two elements are equal the set collection uses the equality operator on the elements that are inserted into it. If you are inserting pointers into the collection this will work automatically, otherwise you must provide an equality comparison operator for the type you are inserting into the set.

The following example illustrates the use of both a bounded and an unbounded set. We start by creating a bounded set of `DataElement` objects on the stack in the `runBoundedSet()` method. 100 objects are created and then copied into `bset` by value, that is, the collection stores copies of the `DataElement` objects. After this we do a `find()` on two random elements, remove two elements and then try to find them.

```
int SetExample::runBoundedSet()
{
  ACE_TRACE("SetExample::runBoundedSet");

  ACE_DEBUG((LM_DEBUG, "Using a bounded set\n"));
  ACE_Bounded_Set<DataElement> bset(100);

  DataElement elem[100];
  for(int i =0; i<100; i++)
    {
      elem[i].setData(i);

      bset.insert(elem[i]);
      if(bset.insert(elem[i])==-1)
        ACE_DEBUG((LM_ERROR,
          "%p\n", "insert set"));
      //insert two copies of the same element..
      //this isn't allowed!!!
    }
  ACE_DEBUG((LM_DEBUG,
        "%d\n",
        DataElement::numOfActiveObjects()));

  DataElement elem1(10), elem2(99);
  if(!bset.find(elem1) && !bset.find(elem2))
    ACE_DEBUG((LM_INFO,
               "The element's %d and %d are in the set!\n",
               elem1.getData(), elem2.getData()));

  for(int j =0; j<50; j++)
    {
      bset.remove(elem[j]);
      //remove these element from the set

      ACE_DEBUG((LM_DEBUG, "%d:",
                 elem[j].getData()));
    }
```

```
  if((bset.find(elem[0])==-1) && (bset.find(elem[49])==-1))
    ACE_DEBUG((LM_INFO,
               "The element's %d and %d are NOT in the set!\n",
               elem[0].getData(), elem[99].getData()));


  return 0;
}
```

The unbounded set is used in the `runUnboundedSet()` method. We start off by creating an unbounded set of `DataElement*`, that is, the set will keep copies of the pointers instead of copies of the elements themselves. We then insert 100 elements that we create on the heap, find two of them randomly, iterate through the collection and delete all the elements from the heap. We do not actually remove the pointers from the unbounded set. Since the set itself is on the stack, it's destructor will remove the copies of the pointers that it has created. Note that we are using the iterator to walk through the collection and delete the elements from the heap and not to remove the entries from the set which would be incorrect as the iterator is invalid once the contents of the underlying collection change.

```
int SetExample::runUnboundedSet()
{
  ACE_TRACE("SetExample::runUnboundedSet");

  ACE_DEBUG((LM_DEBUG,
        "\nUsing an unbounded set and the heap\n"));
  ACE_Unbounded_Set<DataElement*> uset;
  for(int m = 0; m<100; m++)
    {
      DataElement *elem;
      ACE_NEW_RETURN(elem, DataElement(m), -1);
      uset.insert(elem);
      //add the element to the set
    }
  DataElement deBegin(0), deEnd(99);
  if(!uset.find(&deBegin) && !uset.find(&deEnd))
    ACE_DEBUG((LM_DEBUG, "Found the elements\n"));

  //Iterate and destroy the elements in the set
  ACE_DEBUG((LM_DEBUG, "Deleting the elements\n"));
  ACE_Unbounded_Set_Iterator<DataElement*> iter(uset);
  for( iter= uset.begin();
```

```
        iter!=uset.end(); iter++)
    {
      DataElement* elem = (*iter);
      ACE_DEBUG((LM_DEBUG, "%d:", elem->getData()));
      delete elem;
    }

  return 0;
}
```

5.3 Associative Containers

Associative containers support efficient retrieval of elements based on keys instead of positions within the container. Examples of associative containers include 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 as the `ACE_Map_Manager<>` class template. This class maps a key type to a value type. Therefore, insertions take two parameters: 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 there is no space for the new entry) a new allocation takes place and all elements are copied into the new array. When copying takes place can be controlled with the ACE_HAS_LAZY_MAP_MANAGER compile flag. If this flag is set then 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 can be *O(n)*. Retrievals are always a linear *O(n)* operation, though for faster operations with an average case retrieval complexity of *O(1)* the ACE_Hash_Map_Manager<> template is provided.

ACE_Map_Manager requires that the key element 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 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.

```
class KeyType;
bool operator==(const KeyType&, const KeyType&);
//forward dec.

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 off in the MapExample::run() method where we create a 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 which are then passed by reference into the ACE_Map_Manager, which creates and stores copies of the objects. After performing the binding we then proceed to find() the objects by key and display them, once again creating the required keys on the stack. The value is returned by reference. Note that this is a reference to the copy that is

maintained by the map. We then go ahead and iterate through the collection in the forward and reverse direction, remove all the elements and then iterate again showing that there are no elements left.

```
int Map_Example::run()
{
  ACE_TRACE("Map_Example::run");

  for(int i=0; i<100; i++)
    map_.bind(i, DataElement(i));
  //Corresponding KeyType objects are created on the fly.

  ACE_DEBUG((LM_DEBUG,"Map has \n"));
  for(int j=0; j<100; j++)
    {
      DataElement d;
      map_.find(j,d);
      ACE_DEBUG((LM_DEBUG, "%d:", d.getData()));
    }
  ACE_DEBUG((LM_DEBUG, "\n"));

  this->iterate_forward();
    //iterate in the forward direction

  this->iterate_reverse();
  //iterate in the other direction

  this->remove_all();
  //remove all elements from the map

  this->iterate_forward();
  //iterate in the forward direction

  return 0;
}
```

The iteration itself is straightforward. Note that the map does support standard-style nested type defintions for `iterator` and `reverse_iterator`. It also supports the standard-style `begin()` and `end()` methods that return iterators to the begining and ending of the map. The only thing to note here is that when you dereference the iterator you get a `ACE_Map_Manager<EXT_ID, INT_ID,ACE_Lock>::ENTRY` which is type defined to be an `ACE_Map_Entry<EXT_ID, INT_ID>`. If you look at this in the header file

(`Map_Manager.h`) you will see that each entry has an `int_id_` and `ext_id_` attribute that you can use to get to the actual values. In the example we are using `int_id_` to show the values that are stored in the map.

```
void Map_Example::iterate_forward()
{
  ACE_TRACE("Map_Example::iterate_forward");

  ACE_DEBUG((LM_DEBUG, "Forward iteration\n"));
  for(ACE_Map_Manager<KeyType, DataElement, ACE_Null_Mutex>::itera
tor
        iter=map_.begin();
      iter!=map_.end();
        iter++)
    ACE_DEBUG((LM_DEBUG, "%d:", (*iter).int_id_.getData()));
    ACE_DEBUG((LM_DEBUG,"\n"));
}

void Map_Example::iterate_reverse()
{
  ACE_TRACE("Map_Example::iterate_reverse");

  ACE_DEBUG((LM_DEBUG, "Reverse iteration\n"));
  for(ACE_Map_Manager<KeyType, DataElement, ACE_Null_Mutex>::rever
se_iterator
        iter=map_.rbegin();
       iter!=map_.end();
       iter++)
    ACE_DEBUG((LM_DEBUG, "%d:", (*iter).int_id_.getData()));
  ACE_DEBUG((LM_DEBUG,"\n"));
}
```

Note that we do not use the iterators to go through the map when we are removing elements. This is because changing the collection (by removing or adding elements) invalidates the iterator, thus making it impossible to iterate and remove or add at the same time. However, as we mentioned earlier if the map is specified to be of type ACE_HAS_LAZY_MAP_MANAGER then this behavior is supported, that is, deletions can occur during iteration. In the example, however, we choose to use the convenient `unbind_all()` method to remove all entries in the map.

```
void Map_Example::remove_all()
{
  ACE_TRACE("Map_Example::remove_all");

  map_.unbind_all();
  //note that we can't use the iterators here
  //as they are invalidated after deletions
  //or insertions
}
```

Using Specialization

Besides overloading the equality operator to satisfy the `ACE_Map_Manager`'s requirement for comparable key types, you can also use template specialization. In this case you can specialize the `ACE_Map_Manager::equal()` method to provide for the required comparision by the key type. This looks something like the following;

```
class KeyType
{
public:
        KeyType():val_(0){};
        KeyType(int i): val_(i) {};
        KeyType(const KeyType& kt) {this->val_ = kt.val_;};
        operator int() const { return val_;};
private:
        int val_;
};

ACE_TEMPLATE_SPECIALIZATION
int
ACE_Map_Manager<KeyType, DataElement, ACE_Null_Mutex>::equal (cons
t KeyType& r1,
                                                    const KeyType &r2
)
{
        return (r1 == r2);
}
```

Locking

In the discussion so far we have glossed over the third argument to the `ACE_Map_Manager<>` template, the lock type. `ACE_Map_Manager` supports thread safe operations on it. In cases where multiple threads will be accessing your map, you will want to protect it by using an `ACE_Thread_Mutex` instead of an `ACE_Null_Mutex` as used in the example above. Note that thread safety in this sense means that the internal data structure itself will remain consistent, in most cases this is not the only guarantee you need. For example, if two threads were able to bind the same entry (an entry with the same key) without any other synchronization you would never be sure what entry actually existed in the map. In most cases you will find a need for coarser grained locking than the fine grained locks available on the map itself. In these cases you may want to use `ACE_Null_Mutex` with the map and provide your own external locking behaviour.

5.3.2 Hash Maps

The `ACE_Hash_Map_Manager` is an implementation of a hash map data structure. This allows for best case insertions and removals of *O(1)* and a worst case of *O(n)*. The performance of the data structure is dependent on the hashing function and the number of buckets you create for the hash, both of which are tunable in this implementation. Note that the hash map is a value container, that is, it will create copies of each element that you bind into it. Therefore, copy semantics should be valid on the copied in entity.

To illustrate lets build a simple map between integer keys and `DataElement` values. The hashing function is specified as the third template parameter to the map, in the form of a functor. The fourth parameter is an equality functor and the final parameter is the lock type that the map will use to ensure consistency. ACE comes with several useful hash functors that are predefined on different data types. In this case we use `ACE_Hash<int>` as the hash functor and `ACE_Equal_To<int>` as the equality functor, both of which are predefined in `Functor.h`. To make things a little easier (and more succint) we create a new template class that leaves out the key and data types but plugs everything else in.

```
template<class EXT_ID, class INT_ID>
class Hash_Map :
  public ACE_Hash_Map_Manager_Ex<EXT_ID, INT_ID,
  ACE_Hash<EXT_ID>, ACE_Equal_To<EXT_ID>, ACE_Null_Mutex>
{
};
//Little helper class
```

Next you want to specify a bucket size that makes sense for your map. This can be a difficult decision because you want to balance creating a map that is too big and will remain mostly empty with having a map that is too small, degrading the find operation to *O(n)*. In our contrived example we know that we will only have 100 unique keys (and our hash function is perfect, you will always get a different hash value for each key) we open our map with exactly 100 entries knowing that we will have *O(1)* performance.

```
Hash_Map_Example::Hash_Map_Example()
{
  ACE_TRACE("Hash_Map_Example::Hash_Map_Example");

  map_.open(100);
}
```

The rest of the example is similar to the previous one where we insert elements, find them and then iterate through them using the provided iterators and is not something we need to go over again (remember that dereferencing an iterator will return an ENTRY type (i.e, a `ACE_Hash_Map_Entry<>`). The `ACE_Hash_Map_Manager` and `ACE_Map_Manager` support an interface that is very similar although they are not type substitutable with each other.

Defining our functors

In most cases to use the `ACE_Hash_Map_Manager` effectively you must define your own hashing function for your key types (remember the efficiency of your searches is directly related to the ability of your hash function to ensure that all your items go into different buckets). This can be done by specializing the `ACE_Hash<>` functor. We change our previous example by using `KeyType` as the external identifier for the map. We then specialize the `ACE_Hash<>` functor for the `KeyType`, in this case simply returning the underlying integer value to provide a nice distribution.

```
class KeyType
{
public:
   KeyType():val_(0){};
        KeyType(int i): val_(i) {};
        KeyType(const KeyType& kt) {this->val_ = kt.val_;};
        operator int() const { return val_;};
private:
        int val_;
};
//key type that we are going to use

ACE_TEMPLATE_SPECIALIZATION
class ACE_Hash<KeyType>
{
public:
   u_long operator()(const KeyType kt) const
    {
      int val = kt;
      return (u_long)val;
    }
};
//specialize the hash functor

ACE_TEMPLATE_SPECIALIZATION
class ACE_Equal_To<KeyType>
{
  public:
     int operator()(const KeyType& kt1,
               const KeyType& kt2) const
     {
        int val1 = kt1;
        int val2 = kt2;
        return (val1 == val2);

     }
};
//specialize the equal functor
```

The `ACE_TEMPLATE_SPECIALIZATION` macro expands out correctly for your compiler (if your compiler supports standard template specialization it will just be `template<>` otherwise it will expand out to nothing).

5.3.3 Self Adjusting Binary Tree

A self adjusting binary tree provides for an average and worst case complexity for lookups of *O(lg N)*, thus providing a better worst case search time than a hash map (which in the worst case is *O(N)*). Insertions and deletions also have an *O(lg N)* time complexity, which is worse than the hash map. ACE implements a Red Black Tree, a certain type of self adjusting binary search tree, that keeps an extra "color" attribute for each tree node.

The ACE implemention of the tree conforms to the same interface that we have previously seen for the map classes in ACE, thus it is relatively simple to substitute one ACE based map data structure with another based on how they perform with particular data sets. This implementation is a value container, that is, it will create copies of each element that you bind into it (copy semantics should be valid on the copied in entity).

The following example illustrates the use of the Tree , in a manner very similar to what we have seen previously. Items are inserted, found, iterated over in both directions, removed and then once again iterated over. Once again we cannot use the iterator to remove elements as the tree does not allow you to delete the element you are iterating over during iteration (i.e., the iterator is invalidated). However, the tree does allow you to perform additions and deletions on the collection (as long as you don't delete the current item the iterator is pointing to).

```
int Tree_Example::run()
{
  ACE_TRACE("Tree_Example::run");

  DataElement *d  = 0;
  for(int i=0; i<100; i++)
  {
    ACE_NEW_RETURN(d, DataElement(i), -1);
    int result = tree_.bind(i, d);
    if(result!=0)
        ACE_ERROR_RETURN((LM_ERROR, "%p\n", "Bind"), -1);
  }

  ACE_DEBUG((LM_DEBUG,"Using find: \n"));
  for(int i=0; i<100; i++)
    {
      tree_.find(i,d);
      ACE_DEBUG((LM_DEBUG, "%d:", d->getData()));
      }
  ACE_DEBUG((LM_DEBUG, "\n"));
```

```
  this->iterate_forward();
  //use the forward iterate

  this->iterate_reverse();
  //use the reverse iterator

  ACE_ASSERT(this->remove_all()!=-1);
  //removeAll elements from the tree

  this->iterate_forward();
  //iterate through once again

  return 0;
}

void Tree_Example::iterate_forward()
{
  ACE_TRACE("Tree_Example::iterate_forward");

  ACE_DEBUG((LM_DEBUG,"Forward Iteration \n"));
  for(Tree<int, DataElement*>::iterator iter=tree_.begin();
      iter != tree_.end(); iter++)
    ACE_DEBUG((LM_DEBUG, "%d:", (*iter).item()->getData()));
  ACE_DEBUG((LM_DEBUG, "\n"));

}

void Tree_Example::iterate_reverse()
{
  ACE_TRACE("Tree_Example::iterate_reverse");

  ACE_DEBUG((LM_DEBUG,"Reverse Iteration \n"));
  for(Tree<int, DataElement*>::reverse_iterator iter=tree_.rbegin(
);
      iter != tree_.rend(); iter++)
    ACE_DEBUG((LM_DEBUG, "%d:", (*iter).item()->getData()));
  ACE_DEBUG((LM_DEBUG, "\n"));

}

int Tree_Example::remove_all()
{
  ACE_TRACE("Tree_Example::remove_all");

  ACE_DEBUG((LM_DEBUG, "Removing elements\n"));
  for(int i=0; i<100; i++)
```

```
  {
      DataElement * d = 0;
      int result = tree_.unbind(i, d);
      if(result != 0)
        ACE_ERROR_RETURN((LM_ERROR, "%p\n", "Unbind"), -1);

      ACE_ASSERT(d!=0);
      delete d;
  }
  //note that we can't use the iterators here
  //as they are invalidated after deletions
  //or insertions

 return 0;
}
```

In this example, unlike the previous examples, we are keeping pointers in the collection instead of the complete values. This make it neccessary for us to delete the items before we unbind them (we can't just call `unbind_all()`). ACE provides a nice `unbind()` method that returns the value to the caller as the unbind happens. We use this method to remove the elements from the map and then delete them.

The `ACE_RB_Tree<>` template uses the `ACE_Less_Than<>` functor to provide a binary ordering between key elements that are inserted into the tree. ACE provides several specializations for this functor; however, in most cases you will create your own specialization. For example, for the `KeyType` class the `ACE_Less_Than<>` functor would be specialized as;

```
class KeyType
{
public:
        KeyType():val_(0){};
        KeyType(int i): val_(i) {};
        KeyType(const KeyType& kt) {this->val_ = kt.val_;};
        operator int() const { return val_;};
private:
        int val_;
};
//same key type.

ACE_TEMPLATE_SPECIALIZATION
class ACE_Less_Than<KeyType>
{
```

```
   public:
      int operator() (const KeyType k1, const KeyType k2)
      {
         return k1 < k2;
      }
};
```

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 you can use one of the allocators that are 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.

For those of you who are familiar with allocators in the standard C++ library, you will notice several big differences in the way allocators work in ACE:

1. Allocators are passed in during object instantiation and 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 wish for 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.

5.4.1 ACE_Allocator

The `ACE_Allocator` interface is the interface 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 out of the box which are described in the table below.

Table 5.1. Allocators available in ACE

Allocator	Description
`ACE_New_Allocator`	An allocator that allocates memory 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 optimzed fashion. Memory is never deallocated.
`ACE_Cached_Allocator`	A fixed-size allocator that allocates a well defined number of fixed sized blocks of memory. These blocks are returned on allocation and returned to a free list on deallocation.
`ACE_Dynamic_Cached_Allocator`	A version of the cached allocator that allows you to specify the size and number of the cached blocks at run time instead of at compile time.

Lets change one of our previous stack examples and make it use the cached allocator instead of the default allocator. This allocator will preallocate a specified number of fixed size blocks of memory that are handed out on subsquent allocation calls. For example, if we were to create a cached allocator with 10 blocks of 1024 bytes each and then do a `malloc()` call for 4 bytes we would get a complete 1K block in response. You have to be careful when using this special purpose allocator, never ask for more then a block's worth of memory in a single `malloc()` call and be careful that the allocator hasn't run out of memory. Cached allocators come in real handy when you have a predictable memory allocation scenario where you know what the upper bounds are on memory usage and need predictable high speed allocation.

To have our stack use a cached allocator instead of the heap all we have to do is choose the allocator we want to use, create it specifying the appropriate block size and number of blocks and then pass it to the stack during construction.

```
int StackExample::run()
{
   ACE_TRACE("StackUser::run");

   ACE_Allocator *
        allocator = 0;

   size_t block_size =
         sizeof(ACE_Node<DataElement>);
   ACE_NEW_RETURN(allocator,
        ACE_Dynamic_Cached_Allocator<ACE_Null_Mutex>(100 + 1, bloc
k_size),
                  -1);

   ACE_DEBUG((LM_DEBUG, "\n# of live objects %d\n",
          DataElement::numOfActiveObjects()));

   ACE_ASSERT(this->runUnboundedStack(allocator) != -1);

   ACE_DEBUG((LM_DEBUG, "\n# of live objects %d\n",
          DataElement::numOfActiveObjects()));


    delete allocator;

   return 0;
}
```

Here we first determine the size of the blocks that we want to add to the cached allocator. We know that the stack is really a circular list of `ACE_Node<T>`, thus each time we push an element on the stack the container allocates an `ACE_Node<DataElement>`. Thus our block size should be equal or greater than this size. Second, we know we will only be pushing at most 100 elements on the stack, so the allocator must have at least that many blocks (+1 for the head element which is allocated as soon as the container is constructed).

```
int StackExample::runUnboundedStack(ACE_Allocator* allocator)
{
   ACE_TRACE("StackExample::runUnboundedStack");

   ACE_Unbounded_Stack<DataElement>
        ustack(allocator);
   //pass in an allocator during construction.
```

```
   for(int m = 0; m< 100; m++)
      {
         DataElement elem(m);
         int result = ustack.push(elem);
         if(result == -1)
            ACE_ERROR_RETURN((LM_ERROR,
                "%p\n", "Push Next Element"), -1);
      }

   void* furtherMemory = 0;
   furtherMemory =
      allocator->malloc(sizeof(ACE_Node<DataElement>));
   ACE_ASSERT(furtherMemory == 0);
   ACE_DEBUG((LM_DEBUG, "%p\n", "No memory.."));
   //No memory left..

   DataElement e;
   for(int m = 0; m < 10; m++)
        ustack.pop(e);
   //free up some memory in the allocator

   furtherMemory =
        allocator->malloc(sizeof(ACE_Node<DataElement>));
   ACE_ASSERT(furtherMemory != 0);

   return 0;
}
```

Next we pass our new allocator off to the stack during construction. We proceed to push 100 elements onto the stack. At this point there should be no more memory left in our allocator. To confirm this we try to allocate another node from it. As expected, no more memory is allocated. We then `pop()` off a few elements and find that the allocator once again has memory to hand out.

5.4.2 ACE_Malloc

Besides `ACE_Allocator`, ACE also has a general purpose allocator interface called `ACE_Malloc`. This allocator is much fancier than `ACE_Allocator` and allows you to allocate memory using techniques such as System V shared memory or memory mapped files (e.g., `mmap()` based allocation). In cases where you want to allocate your containers using these techniques, you can use an adapter (`ACE_Allocator_Adapter`) to adapt the `ACE_Malloc` template class to the

`ACE_Allocator` interface. We will talk about `ACE_Malloc` in detail when we get to our discussion of shared memory in Chapter 17.

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 features such as memory allocation and synchronization. Many of the ACE containers provide standard-like features making it easier for you to switch between container types easily.

Part I

Interprocess Communication

Chapter 6
Basic TCP/IP Socket Use

This chapter will introduce you to basic TCP/IP programming using the ACE toolkit. We will begin by creating simple clients 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 very rich set of wrapper facades encapsulating many forms of interprocess communication (IPC). Where possible, they present a common API allowing you to interchange one for another without restructuring your entire application. This is an application of the Strategy pattern [1] 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 arranged in sets of:

- 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 will use the ACE's Sockets-wrapping family of classes. These are:

- `ACE_SOCK_Connector`
- `ACE_INET_Addr`
- `ACE_SOCK_Stream`

- `ACE_SOCK_Acceptor`

The top three are used for clients, the bottom three for servers. Each one abstracts a bit of the low level mess of traditional socket programming. All together, they create a very easy to use, typesafe mechanism for creating distributed applications. We won't show you everything they can do but what we will show covers about 80% of the things you'll normally need to do.

6.1 A Simple Client

In BSD Sockets programming you have used a number of low-level operating system calls such as `socket()`, `connect()` and so forth. Programming directly to the Sockets API is troublesome due to *accidental complexities* [4] such 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 different communication families and modes of communication. Again, the compiler can offer no help in diagnosing improper usage.
- 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 both easier and more portable.

Borrowing a page from Stevens' venerable *UNIX Network Programming* we will start by creating a very simple client with just 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);
```

And 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);
```

Ok, so that's pretty simple and you're probably asking yourself "why are they teaching me what I already know?" Well, we do assume that you already know more than a little about network programming and we don't really want to go over all of that in this book. However, we're now going to show you The ACE Way of solving this same problem and we wanted you to have the traditional solution fresh in your mind.
First we'll create the equivalent of a sockaddr_in structure:

```
ACE_INET_Addr srvr (50000, ACE_LOCALHOST);
```

`ACE_INET_Addr` is a member of the `ACE_Addr` family of objects. Some (but not all) classes in that family are `ACE_UNIX_Addr`, `ACE_SPIPE_Addr` and `ACE_FILE_Addr`. Each of these objects represents a concrete implementation of the `ACE_Addr` baseclass and each knows how to handle the details of addressing in their domain. We've used the most commonly used constructor of `ACE_INET_Addr` which takes an `unsigned short` port number and a `char[]` hostname and internally creates the appropriate `sockaddr_in` (or `sockaddr_in6`, for IPv6) structure.

There are a number of other useful ACE_INET_Addr constructors defined in INET_Addr.h. Reading through the ACE_INET_Addr documentation you will very likely find one or more that are useful to your application.

Once you have the `ACE_INET_Addr` constructed appropriately it's time to use that address to get your socket connected. ACE represents a connected TCP socket with the `ACE_SOCK_Stream` object. It's called that because a TCP connection represents a virtual connection or "stream" of bytes (as opposed to the connection-less datagrams you get with UDP sockets). In order to actively connect an `ACE_SOCK_Stream` to a server we use an `ACE_SOCK_Connector` and the `ACE_INET_Addr` already constructed:

```
ACE_SOCK_Connector connector;
ACE_SOCK_Stream peer;

if (-1 == connector.connect (peer, srvr))
  ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "connect"), 1);
```

The connect() method is provided with the thing to connect and the place to which it should be connected. It then attempts to establish that relationship. If successful, the ACE_SOCK_Stream is placed into a connected state and we can use it to communicate with the server. At this point you can begin communicating.[1]

```
peer.send_n ("uptime\n", 7);
bc = peer.recv (buf, sizeof(buf));

write (1, buf, bc);

peer.close ();
```

`ACE_SOCK_Stream` implements the ACE_SOCK API which provides a number of methods for communicating with a remote process. We first use the `send_n()` method to send exactly seven bytes of data (our 'uptime' request) to the server. In your own network programming you have probably experienced "short writes". That is, you attempt to write a number of bytes to the remote but because of network buffer overflow or congestion or any number of other reasons not all of your bytes are transmitted. You must then move your data pointer and

1. The *write(1,...)* should send the output to the standard output (e.g. -- console) for your operating system. However, that may mean something completely different for an embedded application. The point? Beware of portability issues at all times and try to avoid things like *write(1,...)*.

send the rest. You continue doing this until all of the original bytes are sent. This happens so often that ACE provides you with the `send_n()` method call. It simply internalizes all of these retries so that it doesn't return to you until either it has sent everything or it has failed while trying.

The `recv()` method we've used above is the simplest available. It will read up to *n* bytes from the peer and place them into the designated buffer. Of course, if you know exactly how many bytes to expect then you're going to have to deal with "short reads". ACE has solved this for you with the `recv_n()` method call. Like `send_n()`, you tell it exactly how much to expect and it will take care of ensuring that they all are read before control is returned to your application.

Other send and receive methods allow you to set a timeout or change the socket IO flags. Still other methods will do crazy things for you like allocating a read buffer on your behalf or will even use overlapped I/O if your operating system supports such things.

Here, back to back, are both the traditional and ACE versions in their entirety:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main (int argc, char * argv [])
{
    int fd;

    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);

    fd = socket (AF_INET, SOCK_STREAM, 0);

    assert (fd >= 0);

    assert(
        connect (fd,
```

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

    int bc;
    char buf[64];
    memset (buf, 0, sizeof(buf));

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

    write (1, buf, bc);

    close (fd);

    exit (0);
}


#include "ace/INET_Addr.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Connector.h"
#include "ace/Log_Msg.h"

int main (int argc, char *argv[])
{
    ACE_INET_Addr srvr (50000, ACE_LOCALHOST);

    ACE_SOCK_Connector connector;
    ACE_SOCK_Stream peer;

    if (-1 == connector.connect (peer, srvr))
      ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "connect"), 1);

    int bc;
    char buf[64];

    peer.send_n ("uptime\n", 7);
    bc = peer.recv (buf, sizeof(buf));

    write (1, buf, bc);

    peer.close ();

    return (0);
}
```

Adding Robustness

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 `set()` method 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 non-trivial. 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.[2]

Another handy thing we can do with `ACE_INET_Addr` is convert its current value into a printable string. If you want your application to report progress back to its user then this may be something you would be interested in:

```
addr.set ("HAStatus", ACE_LOCALHOST);
if (addr.addr_to_string (peerAddress,
                         sizeof(peerAddress), 0) == 0)
  {
    ACE_DEBUG ((LM_DEBUG,
                "(%P|%t) Connecting to %s\n",
                peerAddress));
  }
```

2. ACE_OS::last_error() simply returns *errno* on Unix and Unix-like systems. For Win32, however, it uses the GetLastError() function. To increase portability of your application you should get in the habit of using ACE_OS::last_error().

The `addr_to_string()` method requires a buffer in which to place the string and the size of that buffer. It takes an optional third parameter specifying the format of the string it creates. Your options are (0) ip-name:port-number and (1) ip-number:port-number. If the buffer is large enough for the result it will be filled and null-terminated appropriately and the method will return zero. If the buffer is too small the method will return -1 indicating an error.

Now lets turn our attention to `ACE_SOCK_Connector`. The first thing we should probably worry about is checking the result of the `connect()` attempt. In our earlier example we simply `assert()` that it returns something other than the ubiquitous -1 failure code. You probably don't want your application to do this if you fail to connect to the server process. Even if your application will exit under those conditions it should at least provide some sort of warning to the user before doing so. In some cases, you may even choose to pause a while 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_Connectoryou 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 cool and you'll probably impress your friends but be absolutely certain that you check errno after constructing the connector or you will have one nasty bug to track down.[3]

```
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, "%p\n", "status"), 100);
```

Don't fret if you don't want to use the active constructors but you 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 the server

3. You'll notice in this example that we explicitly use ACE_OS::last_error(0) to set the last error value to zero before invoking the ACE_SOCK_Connector constructor. Although we would like to assume that the system calls below the ACE level do this for us, it is good form to do it ourselves "on purpose."

may be slow to respond you may want to timeout your connection attempt and either retry or exit.

```
ACE_SOCK_Connector logConnector;
ACE_Time_Value timeout (10);
ACE_SOCK_Stream log;
if (logConnector.connect (log, addr, &timeout) == -1)
  {
    if (ACE_OS::last_error() == ETIME)
      {
        ACE_DEBUG ((LM_DEBUG,
                    "(%P|%t) Timeout while "
                    "connecting to log server\n"));
      }
    else
      {
        ACE_ERROR ((LM_ERROR,
                    "%p\n",
                    "log"));
      }
    return (101);
}
```

In most client applications you will let the operating system choose your local port. ACE represents this value as `ACE_Addr::sap_any`. In some cases, however, you may want to choose your own port value. A peer-to-peer application, for instance, may behave this way. As always, ACE provides a way. Simply create your `ACE_INET_Addr` and provide it as the fourth parameter to `connect()`. If some other process is or might be listening on that port then give a non-zero value to the fifth parameter and the "reuse" socket option will be invoked for you.

```
ACE_SOCK_Connector logConnector;
ACE_INET_Addr local (4200, ACE_LOCALHOST);
if (logConnector.connect (log, addr, 0, local) == -1)
  {
    ...
```

Here we've chosen to set the port value of our local endpoint to 4200 and "bind" to the loopback network interface. Some server applications (rsh comes to mind) look at the port value of the client that has connected to them and will refuse the

connection if it is not in a specified range. This is a somewhat insecure way of securing an application but can be useful in preventing spoofs if combined with other techniques.

Still more you want the connector to handle for you? Reading through the ACE_SOCK_Connector documentation again we find that you can set quality of service parameters on your connection or even setup an asynchronous, nonblocking connection. We will explore some of these advanced features in Chapter 8.

Finally, we come to `ACE_SOCK_Stream`. We've already talked about the basic send and receive functionality. As you might suspect, both support the ability to timeout long-running operations. Like the `connect()` method of `ACE_SOCK_Connector`, we simply need to provide an `ACE_Time_Value` with our desired timeout.

```
ACE_Time_Value sendTimeout (0, 5);
if (status.send_n ("uptime\n", 7, &sendTimeout) == -1)
  {
    if (ACE_OS::last_error() == ETIME)
      {
        ACE_DEBUG ((LM_DEBUG,
                    "(%P|%t) Timeout while "
                    "sending query to status server\n"));
      }
```

And, of course, we want to find out what the status server has to say in return:

```
ssize_t bc ;
ACE_Time_Value recvTimeout (0, 1);
if ((bc = status.recv (buf, sizeof(buf), &recvTimeout)) == -1)
  {
    ACE_ERROR ((LM_ERROR,
                "%p\n",
                "recv"));
    return (103);
  }

log.send_n (buf, bc);
```

If you've worked with the low-level socket API then you may have come across the `readv()` and `writev()` system calls. When you use the `read()` and `write()` you have to work with contiguous data areas. With `readv()` and `writev()` you can use an array of `iovec` structures[4]. This will allow you to

send an arbitrary amount of non-contiguous data to the peer or receive peer data into a similar non-contiguous area. ACE doesn't let you down here, `ACE_SOCK_Stream` has `send()` and `recv()` signatures that allow you to work with an array of `iovec` structures. Here again we see how ACE will make your transition from traditional network programming to object-oriented network programming much smoother.

If we wanted to use an `iovec` to send our original uptime query to the server it might look something like this:

```
iovec send[4];
send[0].iov_base = ACE_const_cast (ACE_TCHAR *, "up");
send[0].iov_len  = 2;
send[1].iov_base = ACE_const_cast (ACE_TCHAR *, "time");
send[1].iov_len  = 4;
send[2].iov_base = ACE_const_cast (ACE_TCHAR *, "\n");
send[2].iov_len  = 1;

peer.sendv (send, 3);
```

Of course this is a contrived and not very realistic example, your real `iovec` array wouldn't likely be created this way at all. Consider the case where you have a table of commands to send to a remote server. You could construct a "sentence" of requests by a cleverly built iovec array.

```
iovec query[3];
addCommand (query, UPTIME);
addCommand (query, HUMIDITY);
addCommand (query, TEMPERATURE);
peer.sendv (query, 3);
```

4. iovec and the writev()/readv() system calls were introduced in the 4.3 BSD operating system. Reference *UNIX Network Programming* by W. Richard Stevens here. They are most commonly used when you need to send data from or receive data into non-contiguous buffers. The common example is sending a header and associated data which are already in separate buffers. With a standard write() system call you would have to use two calls to send each buffer individually. This is unacceptable if you want both to be written together atomically. Alternatively you could copy both into a single, larger buffer and use one call but this has drawbacks both in the amount of memory used and the time required. writev() (and, thus, ACE_SOCK_Stream's sendv() method) will atomically send all entries of the iovec array. readv() simply does the reverse of writev() by filling each buffer in turn before moving on to the next.

Imagine that `addCommand()` populates the query array appropriately from a global set of commands indexed by the `UPTIME`, `HUMIDITY`, and `TEMPERATURE` constants. You've now done a couple of very interesting things: you are no longer coding the command strings into the body of your application and you've begun the process of defining macros that will allow you to have a more robust conversation with the status server.

Receiving data with an `iovec` is pretty straight forward as well. Simply create your array of `iovec` structures in whatever manner makes the most sense to your application. We'll take the easy route here and just allocate some space. You might choose to point to an area of a memory-mapped file or a shared memory segment or some other interesting place.

```
iovec receive[2];
receive[0].iov_base = new char [32];
receive[0].iov_len  = 32;
receive[1].iov_base = new char [64];
receive[1].iov_len  = 64;

bc = peer.recvv (receive, 2);
```

Still, regardless of where the `iov_base` pointers point to, you have to do something with the data that gets stuffed into them.

```
for (int i = 0; i < 2 && bc > 0; ++i)
  {
    write (1, receive[i].iov_base,
           bc > receive[i].iov_len ? receive[i].iov_len : bc);
    bc -= receive[i].iov_len;
    delete receive[i].iov_base;
  }
```

There's one more thing we'd like to show you with the `iovec` approach. If you want, you can let the `recvv()` method call allocate the receiving data buffer for you, filling in the `iovec` with the pointer and length. It will figure out how much data is available and allocate a buffer just that big. This can be quite handy when you're not sure how much data the remote is going to send you but you're pretty sure it will all fit into a reasonably sized space and you'd like for that space to be contiguous. You're still responsible for freeing the memory to prevent leaks.

```
peer.send_n ("uptime\n", 7);
iovec response;
peer.recvv (&response);
write (1, response.iov_base, response.iov_len);
delete [] response.iov_base;
```

6.2 Building a Server

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

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,
                     "%p\n",
                     "acceptor.open"),
                    100);
```

The acceptor takes care of the underlying details like `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:
(listing15-1-2 should be here... there's a problem with the code generation that we'll fix later.)

This method creates a basic BSD-style socket. The most common usage will be as shown in the example above 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 there is already a connection on that port. If your server is not likely to service new connection requests rapidly then you may also want to adjust the `backlog` parameter.

Once you have an address defined and have opened the acceptor to listen for new requests you want to actually wait for those connection requests to arrive. This is done with the `accept()` method that closely mirrors the `accept()` system call.

```
        if (acceptor.accept (peer) == -1)
          {
            ACE_DEBUG ((LM_DEBUG,
                        "(%P|%t) Failed to accept client connectio
n\n"));
            return (100);
           }
```

This usage will block until a connection attempt is made. A better approach is to apply a timeout

```
        if (acceptor.accept (peer, &peer_address, &timeout, 0) ==
-1)
          {
            if (ACE_OS::last_error() == EINTR)
              ACE_DEBUG ((LM_DEBUG,
                          "(%P|%t) Interrupted while "
                          "waiting for client connection\n"));
            else
              if (ACE_OS::last_error() == ETIMEDOUT)
                ACE_DEBUG ((LM_DEBUG,
                            "(%P|%t) Timeout while "
                            "waiting for client connection\n"));
          }
```

If no client connects in the specified time you can at least print a message to the logfile to let the administrator know that your application is still open for business. As we learned in Chapter 3, we can easily turn those things off if they become a nuisance.

Regardless of which approach you take, a successful return will provide you a valid peer object initialized and representing a connection to the client. It is worth noting that by default the `accept()` method will restart itself if it is interrupted by a Unix signal such as `SIGALRM`. That may or may not be appropriate for your application. In the example above we have chosen to pass *0* as the fourth param-

eter (*restart*) of the accept() method. This will cause accept() to return -1 and ACE_OS::last_error() to return EINTR if the action is interrupted.

Once you have accepted a connection it is generally a good idea to make sure that its valid. While a *0* return from accept() indicates successful acceptance of the peer connection, you may want to go further and verify that the peer's port number falls within a required range or apply other security/sanity checks to the connection attributes.

```
else if (peer_address.get_port_number () == 0)
  {
    ACE_DEBUG ((LM_DEBUG,
                "(%P|%t) Invalid address for client\n"));
  }
```

It is also wise to check that the connection handle associated with the peer object is useful:

```
else if (peer.get_handle () == ACE_INVALID_HANDLE)
  {
    ACE_ERROR ((LM_ERROR,
                "%p\n",
                "accept"));
  }
```

Now we've used all the tricks at our disposal to ensure a good and timely connection from our clients. We begin by setting a timeout of ten seconds. If no client connects in that time then `accept()` will return -1 and `errno` will be `ETIMEDOUT`.

If the `accept()` succeeds we then check the address object that represents the client's endpoint of the connection. The port value will never be zero for a valid connection. Like rsh you may also want to verify that the port value is within a certain range.

Our final check is to ensure that the peer's handle value is valid. We haven't talked about handles before but if you've ever written code for Windows you probably have an idea of what they are. Simply put, a handle is an opaque chunk of native data that represents the connection. You will almost never care what this value is other than to compare it to `ACE_INVALID_HANDLE` which represents, surprisingly enough, an invalid handle. Certainly if your peer object's handle is invalid the connection to that peer cannot be worth using.

Now that we have accepted the client connection and validated it we can begin to work with it. At this point the distinction between client and server begins to blur because you just start sending and receiving data. In some applications the server will send first and in others the client will do so. How your application behaves depends on your requirements specification. For our purposes we will assume that the client is going to send a request which the server will simply echo back. As our examples become more robust in future chapters we will begin to process those requests into useful actions.

```
char buffer[4096];
ssize_t bytes_received;

while ((bytes_received =
        peer.recv (buffer, sizeof(buffer))) != -1)
  {
    peer.send_n (buffer, bytes_received);
  }

peer.close ();
```

As the server is currently written it will only process one client connection and request then exit. If we wrap a simple while loop around everything following the `accept()` we can service multiple clients but only one at a time. Such a server isn't very realistic but its important that you see the entire example. In a future chapter we will return to this simple server and enhance it with the ability to handle multiple, concurrent clients.

```
#include "ace/INET_Addr.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/Log_Msg.h"

int main (int, char * argv [])
{

    ACE_INET_Addr port_to_listen ("HAStatus");
    ACE_SOCK_Acceptor acceptor;

    if (acceptor.open (port_to_listen, 1) == -1)
      {
        // Why not ACE_ERROR_RETURN???
        ACE_ERROR ((LM_ERROR,
                    "%p\n",
```

```
                    "acceptor.open"));
        return( 100 );
      }


    while (1)
      {
        ACE_SOCK_Stream peer;
        ACE_INET_Addr peer_address;
        ACE_Time_Value timeout (10, 0);

        /*
         * Basic acceptor usage
         *
        if (acceptor.accept (peer) == -1)
          {
            ACE_DEBUG ((LM_DEBUG,
                        "(%P|%t) Failed to accept client connectio
n\n"));
            return (100);
          }
         *
         */

        if (acceptor.accept (peer, &peer_address, &timeout, 0) ==
-1)
          {
            if (ACE_OS::last_error() == EINTR)
              ACE_DEBUG ((LM_DEBUG,
                          "(%P|%t) Interrupted while "
                          "waiting for client connection\n"));
            else
              if (ACE_OS::last_error() == ETIMEDOUT)
                ACE_DEBUG ((LM_DEBUG,
                            "(%P|%t) Timeout while "
                            "waiting for client connection\n"));
          }
        else if (peer_address.get_port_number () == 0)
          {
            ACE_DEBUG ((LM_DEBUG,
                        "(%P|%t) Invalid address for client\n"));
          }
        else if (peer.get_handle () == ACE_INVALID_HANDLE)
          {
            ACE_ERROR ((LM_ERROR,
                        "%p\n",
```

```
                        "accept"));
          }
        else
          {
            char buffer[4096];
            ssize_t bytes_received;

            while ((bytes_received =
                    peer.recv (buffer, sizeof(buffer)-1)) != -1)
              {
                buffer[bytes_received] = 0;
                peer.send_n (buffer, bytes_received+1);
              }

            peer.close ();
          }
      }

    return (0);
}
```

6.3 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 of your traditionally-coded, single-threaded network applications to a true C++ object-oriented implementation.

By using the ACE objects you can create more maintainable and portable applications. Because you're working at a high-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 catch more errors at compile time rather than run time.

Chapter 7
ACE_Reactor

The purpose of the ACE_Reactor is, quite simply, to react to events. Events of interest to us are recurring timers, POSIX signals, I/O operations and Win32 handle signaling.

Many traditional applications will handle such things by creating new processes or threads. This is particularly popular in servers needing to handle multiple simultaneous client connections. While OK in many circumstances, the overhead of process or thread creation and maintenance can be unacceptable in others. Context switching, memory requirements and so forth can quickly cripple an application executing in an environment of limited resources.

Another popular approach is the use of the select(), poll() and WaitForMultipleObjects() systemcalls. These are excellent alternatives that allow us to handle many events with only one process or thread. Writting portable applications that use these can be quite challenging, however and that's where the ACE_Reactor helps us out.

At its core, depending on the operating system on which it is deployed, the ACE_Reactor is implemented in terms of either select(), poll() or WaitForMultipleObjects()[1]. In true ACE fasion, the ACE_Reactor provides us with a unified API wrapped around the specific implementation. This insulates us from the myriad of details we would otherwise have to know to write a portable application capable of responding to timers, signals, IO events and Win32 handle signaling.

The objects we'll visit in this chapter are:

- ACE_Reactor
- ACE_Event_Handler
- ACE_Sig_Set
- ACE_Time_Value
- ACE_Svc_Handler<>
- ACE_Acceptor<>

7.1 Basic structure

In our first listing we have a simple application using the ACE_Reactor:

```
/**
 * Reactor Listing 01
 *
 * Basic structure of an application using ACE_Reactor
 */

#include <ace/Reactor.h>
#include <ace/Event_Handler.h>

class MyTimerHandler : public ACE_Event_Handler
{
public:
    MyTimerHandler()
            : ACE_Event_Handler()
    {
    }

    int handle_timeout (const ACE_Time_Value &current_time,
                        const void * = 0)
    {
        time_t epoch = ((timespec_t)current_time).tv_sec;
```

1. In fact, there are nine reactor implementations available at the time of this writting. The API declared by ACE_Reactor has proven flexible enough to allow for easy integration with third-party toolkits such as the Xt framework of the X-Window system and the Win32 COM/DCOM framework.

```
        ACE_DEBUG(( LM_INFO,
                    "handle_timeout: %s\n",
                    ACE_OS::ctime(&epoch) ));

        return 0;
    }
};

class MySignalHandler : public ACE_Event_Handler
{
public:
    MySignalHandler()
            : ACE_Event_Handler()
    {
    }

    int handle_signal (int signum, siginfo_t * = 0,
                       ucontext_t * = 0)
    {
        ACE_DEBUG(( LM_INFO,
                    "handle_signal: Caught signal %d\n",
                    signum ));

        if( signum == SIGINT )
        {
            ACE_Reactor::instance()->end_reactor_event_loop();
        }
        return 0;
    }
};

class MyConnectionHandler : public ACE_Event_Handler
{
public:
    MyConnectionHandler()
            : ACE_Event_Handler()
    {
    }

    int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE)
    {
        // ...
    }
};

int main( int argc, char **argv )
```

```
{
    ACE_UNUSED_ARG(argc);
    ACE_UNUSED_ARG(argv);

    MyTimerHandler * timer = new MyTimerHandler();
    ACE_Time_Value initialDelay( 3 );
    ACE_Time_Value interval( 5 );
    ACE_Reactor::instance()->schedule_timer( timer,
                                             0,
                                             initialDelay,
                                             interval );

    MySignalHandler * signal = new MySignalHandler();
    ACE_Sig_Set signalSet(1);
    ACE_Reactor::instance()->register_handler( signalSet,
                                               signal );

    MyConnectionHandler * connection = new MyConnectionHandler();
    ACE_Reactor::instance()->register_handler(
        connection,
        ACE_Event_Handler::READ_MASK );

    ACE_Reactor::instance()->run_reactor_event_loop();

    return 0;
}
```

Here we have created three simple derivatives of the ACE_Event_Handler. The ACE_Event_Handler is the glue that binds your application logic to the reactor framework. The manner in which your class is connected to the reactor determines which methods on your class the reactor will invoke.

For instance, we have used *schedule_timer()* to connect an instance of *MyTimerHandler* to the reactor. When the timer is fired, the reactor is obligated to invoke the *handle_timeout()* method of the object. Likewise, our *MySignalHandler* instance is connected via a *register_handler()* method taking an *ACE_Sig_Set* instance ensuring that the *handle_signal()* method will be called.

In the sections that follow we will be exploring each of these usecases in a bit more detail. Before we get there, however, lets break down the example above into a bird's eye view of how to use the ACE_Reactor:

1. Create a derivative of ACE_Event_Handler

```
class MyHandler : public ACE_Event_Handler
{
public:
    MyHandler(
        // ...
        )
            : ACE_Event_Handler()
    {
        // ...
    }
```

2. Define the method or methods that will be invoked by ACE_Reactor

```
    int handle_timeout (const ACE_Time_Value &current_time,
                        const void * = 0)
    {
        // ...
    }

    // or

    int handle_signal (int signum, siginfo_t * = 0,
                       ucontext_t * = 0)
    {
        // ...
    }

    // or

    int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE)
    {
        // ...
    }

    // or ...
```

3. Create an instance of your derivative, generally in *main()*

```
    MyHandler * handler = new MyHandler();
```

4. Connect your handler to the reactor

```
    ACE_Time_Value initialDelay( 3 );
    ACE_Time_Value interval( 5 );
    ACE_Reactor::instance()->schedule_timer( handler,
                                             0,
                                             initialDelay,
                                             interval );

    // or

    ACE_Sig_Set signalSet(1);
    ACE_Reactor::instance()->register_handler( signalSet,
                                               handler );

    // or

    ACE_Reactor::instance()->register_handler( handler,
                                               ACE_Event_Handler::
READ_MASK );

    // or ...
```

5. Fire up the reactor

```
    ACE_Reactor::instance()->run_reactor_event_loop();
```

7.2 Signals

Responding to signals on POSIX systems generally involves providing the *signal()* systemcall 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[2]. The 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 an

2. For a very long time this was about the only way to do anything asynchronously in Unix.

ACE_Event_Handler derivative with your code in its *handle_signal()* method then register an instance of your object with one of the two appropriate *register_handler()* methods.

7.2.1 Catching one signal

For our first example lets consider a very simple requirement: Write an application that will print the text *Hello World* when it receives the INTterrupt signal (typically generated by CTRL-C at the keyboard).

1. Define the event handler

```
class MySignalHandler : public ACE_Event_Handler
{
public:
    MySignalHandler()
            : ACE_Event_Handler()
    {
    }

    int handle_signal (int signum,
                       siginfo_t * = 0,
                       ucontext_t * = 0)
    {
        ACE_DEBUG(( LM_INFO,
                    "Hello World\n" ));

        return 0;
    }
};
```

2. Create and register the event handler

```
    MySignalHandler * handler = new MySignalHandler();
    ACE_Reactor::instance()->register_handler( SIGINT,
                                               handler );

    ACE_Reactor::instance()->run_reactor_event_loop();
```

Because our design requires us to only catch SIGINT, we haven't put any error checks into the *handle_signal()* method to ensure that we are, in fact, getting the signal we expect.

7.2.2 Catching two signals with one event handler

On POSIX-compliant platforms you can send an application the SIGTSTP signal[3] to put it to sleep. From the shell this is generally done with the CTRL-Z keystroke which is then typically followed by a command to resume the command as a background process. Let's append to our design requirement now: Enhance the signal catching program to prevent backgrounding by printing a message instead of sleeping.

We can reuse the same signal handler and have it check the *signum* parameter or we can create a new event handler derivative instead. For such a simple requirement we will extend our current handler and register it twice with the reactor: once for each signal we want to catch.

1. Define our new event handler

```
class MySignalHandler : public ACE_Event_Handler
{
public:
    MySignalHandler()
            : ACE_Event_Handler()
    {
    }

    int handle_signal (int signum,
                       siginfo_t * = 0,
                       ucontext_t * = 0)
    {
        switch( signum )
        {
            case SIGINT:
                ACE_DEBUG(( LM_INFO,
                            "Hello World\n" ));
                break;

            case SIGTSTP:
                ACE_DEBUG(( LM_INFO,
                            "No Stopping!\n" ));
                break;
        }
```

3. Not to be confused with SIGSTOP which is unblockable.

```
        return 0;
    }
};
```

2. Register it with the reactor

```
MySignalHandler * handler = new MySignalHandler();

ACE_Reactor::instance()->register_handler( SIGINT,
                                           handler );

ACE_Reactor::instance()->register_handler( SIGTSTP,
                                           handler );

ACE_Reactor::instance()->run_reactor_event_loop();
```

7.2.3 Using ACE_Sig_Set

If we want to catch still more signals we can continue along this path but our registration is going to start getting ugly pretty quickly. There is another *register_handler()* method that takes an entire set of signals instead of just one. We will explore that as we extend again to intercept the next popular method of stopping a program (*kill -HUP pid*).

1. Defining our ever-expanding handler

```
class MySignalHandler : public ACE_Event_Handler
{
public:
    MySignalHandler()
            : ACE_Event_Handler()
    {
    }

    int handle_signal (int signum,
                       siginfo_t * = 0,
                       ucontext_t * = 0)
    {
        switch( signum )
        {
            // ...
```

```
            case SIGHUP:
                ACE_DEBUG(( LM_INFO,
                            "You can hangup but "
                            "I won't go away!\n" ));
                break;
        }

        return 0;
    }
};
```

2. A new way to register with the reactor

```
ACE_Sig_Set signalSet;
signalSet.sig_add( SIGINT );
signalSet.sig_add( SIGTSTP );
signalSet.sig_add( SIGHUP );

MySignalHandler * handler = new MySignalHandler();

ACE_Reactor::instance()->register_handler( signalSet,
                                           handler );

ACE_Reactor::instance()->run_reactor_event_loop();
```

The advantages of using ACE_Sig_Set instead of multiple calls to *register_handle()* may not be immediately apparent but as a rule of thumb it is easier to work with a set (or collection) of things than it is to work with a number of individual items. For instance, you can create the signal set in some initialization routine, pass it around for a while and then feed it to the reactor.

In addition to sig_add() there are also:

- *sig_del()* to remove a signal from the set
- *is_member()* to determine if a signal is in the set
- *empty_set()* to remove all signals from the set
- *fill_set()* to fill the set with all known signals.

7.2.4 Introducing *siginfo_t*

Thus far we have ignored the *siginfo_t* parameter of the *handle_signal()* method. Before the POSIX standard we could only use the *signal()* systemcall to register a simple callback function to be invoked on receipt of a signal. That function's only parameter is the signal which caused it to be invoked. With the advent of the POSIX standard, however, we gained the *sigaction()* family of systemcalls which allow us much more flexibility. Most interesting in this is the *siginfo_t* structure instance provided to the signal handling function (and subsequently to our event handler's *handle_signal()* method). One could easily write an entire chapter on the things *siginfo_t* can provide. We will touch on a few examples here and encourage the reader to study the *sigaction()* manpage for more detail on the data available in the *siginfo_t* structure.

The meaning of the information in the *siginfo_t* structure varies depending on the actual signal caught. In our next example we switch on the value of the signal to determine what information is relevant.

1. In order to print more interesting debug messages we define an array of character strings mapped to signal numbers. On a Linux 2.4 system you can find the signal descriptions in /usr/include/bits/siginfo.h

```
const char * signames [] =
{
    "no such signal",
    "Hangup (POSIX).",
    "Interrupt (ANSI).",
    "Quit (POSIX).",
    // ...
```

2. We begin our signal handling method by using our *signames* array to display a description of the signal received. If the *siginfo_t* structure was not provided we exit.

```
int handle_signal (int signum,
                   siginfo_t * siginfo = 0,
                   ucontext_t * = 0)
{
    const char * signame = signames[signum];

    ACE_DEBUG(( LM_INFO,
                "Received signal [%s]\n",
                signame));
```

```
if( siginfo == 0 )
{
    ACE_DEBUG(( LM_INFO,
                "No siginfo_t available for "
                "signal [%s]\n",
                signame));
    return 0;
}
```

3. *si_signo*, *si_errno* and *si_code* are available for all signals received. We already printed the signal number above but now we will print the *errno* value. We also print the process id and user id which sent the signal. These values are only valid for POSIX.1b signals but our simple example doesn't make that distinction.

   The *si_code* member variable provides us with information about why the signal was sent. Some values are signal-specific, others are relevant to all signals. Here we will look at some of this latter set and print a description of the reason code.

```
ACE_DEBUG(( LM_INFO,
            "errno for this signal is %d [%s]\n",
            siginfo->si_errno,
            strerror(siginfo->si_errno)));
ACE_DEBUG(( LM_INFO,
            "signal was sent by process %d"
            " / user %d\n",
            siginfo->si_pid,
            siginfo->si_uid));

switch( siginfo->si_code )
{
    case SI_TIMER:
        ACE_DEBUG(( LM_INFO,
                    "Timer expiration\n"));
        break;

    case SI_USER:
        ACE_DEBUG(( LM_INFO,
                    "Sent by kill, sigsend or raise\n"));
```

```
            break;

        case SI_KERNEL:
            ACE_DEBUG(( LM_INFO,
                        "Sent by kernel\n"));
            break;

        // ...
    };
```

4. As our example continues we now have to inspect the signal value before we can determine which parts of *siginfo_t* are applicable. The *si_address* attribute, for instance, is only valid for SIGILL, SIGFPE, SIGSEGV AND SIGBUS. For SIGFPE (floating-point exception) we will display a description of *si_code* along with *si_address* indicating both why the signal was raised and the memory location wich raised it.

```
switch( signum )
{
    case SIGFPE:
        switch( siginfo->si_code )
        {
            case FPE_INTDIV:
            case FPE_FLTDIV:
                ACE_DEBUG(( LM_INFO,
                            "Divide by zero at 0x%x\n",
                            siginfo->si_addr));
                break;

            case FPE_INTOVF:
            case FPE_FLTOVF:
                ACE_DEBUG(( LM_INFO,
                            "Numeric overflow at 0x%x\n",
                            siginfo->si_addr ));
                break;

            // ...
        }
        break;
```

5. As our outer *switch()* continues we may also display similar details for SIGSEGV (segmentation violation).

```
        case SIGSEGV:
            switch( siginfo->si_code )
            {
                // ...
            };

            break;
```

6. Our rather lengthy *handle_signal()* concludes by displaying the appropriate *siginfo_t* attributes when our process receives notification of a child processes termination.

```
        case SIGCHLD:
            ACE_DEBUG(( LM_INFO,
                        "A child process has exited\n"));
            ACE_DEBUG(( LM_INFO,
                        "The child consumed %l/%l time\n",
                        siginfo->si_utime,
                        siginfo->si_stime));
            ACE_DEBUG(( LM_INFO,
                        "and exited with value %d\n",
                        siginfo->si_status));
            break;

        // ...
    }

    return 0;
}
```

7. Our *main()* routine is the same as ever but we have inserted a small snippet of code that will create a short-lived child process allowing us to test our SIGCHLD handling code.

```
ACE_Sig_Set signalSet;
signalSet.fill_set();

MySignalHandler * handler = new MySignalHandler();

ACE_Reactor::instance()->register_handler( signalSet,
                                           handler );
```

```
    // ...
    int childPid = ACE_OS::fork();
    if( childPid == 0 ) // Parent process
    {
    // ...
    }

    ACE_Reactor::instance()->run_reactor_event_loop();
```

There are more attributes of *siginfo_t* than we have shown here. For information about these as well as which attributes apply to each signal please refer to the *sigaction* documention of your operating system.

7.2.5 Introducing *ucontext_t*

The final parameter of the *handle_signal()* method is *ucontext_t*. This structure contains the execution context (eg -- CPU state, FPU registers, etc...) of the application from just before the signal was raised. A thorough discussion of this topic is very much beyond the scope of this chapter. The reader is encouraged to consult the operating system and vendor documentation for details on the exact content and manner of interpreting the *ucontext_t* structure.

7.3 Timers

There may be times when 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()* method 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;
}
```

timerTask() will create a child process and return it's 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 non-zero 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()* creates the timer-handling process with an initial delay of three seconds and an interval of five seconds. Thus, the function *foo* will be invoked three seconds after the *timerTask()* is called and every five seconds thereafter.

main() then 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;
}
```

foo() isn’t very complicated. For sake of illustration we simply print the current time at each invocation.

As you might surmise, there are some problems with this approach:

- It inherently non-portable. Using the low-level *fork()* and *kill()* calls won’t work on every operating system or platform.
- It is resource intensive if you have more than a few timers. Each *timerTask* invocation creates a new process (or thread if you choose that implementation). If your application requires many timers you will quickly run into limitations.
- There is no control over the timer task after its creation other than to kill it. No chance to suspend or resume the timer or alter the timing interval.

7.3.1 handle_timeout

Our alternative is to use an ACE_Event_Handler derivative to handle timeout events from the ACE_Reactor. This easily addresses our three obvious shortcomings above.

To use the reactor in this way we first create an event handler with more or less the functionality of the previous example’s *foo()* function:

```
class MyTimerHandler : public ACE_Event_Handler
{
public:
    int handle_timeout (const ACE_Time_Value &current_time,
                        const void * = 0)
    {
        time_t epoch = ((timespec_t)current_time).tv_sec;

        ACE_DEBUG(( LM_INFO,
                    "handle_timeout: %s\n",
                    ACE_OS::ctime(&epoch) ));
```

```
        return 0;
    }
};
```

We know that all handlers registered with the reactor share the same process/thread. As such, the actual time at which *handle_timeout()* is invoked might not be the actual time at which the timer "went off". That is, the interval timer for a handler may expire while another event handler is in the middle of one of its *handle_** methods. The *current_time* parameter is the time that our event handler was selected for dispatching rather than the actual current system time.[4]

Registering your timer handler with the reactor is straight-forward:

```
MyTimerHandler * timer = new MyTimerHandler();
ACE_Time_Value initialDelay( 3 );
ACE_Time_Value interval( 5 );
ACE_Reactor::instance()->schedule_timer( timer,
                                         0,
                                         initialDelay,
                                         interval );
```

Like our non-ACE example we've set an initial delay of three seconds and an interval of five seconds. To create a one-shot timer simply omit the *interval* paramter.

7.3.2 State data

The second parameter to the *handle_timeout()* method can be used to pass state data into your event handler. Consider a timeout handler used for monitoring temperature sensors. For purposes of illustration let's assume that our design requires us to use a single handler to monitor multiple sensors possibly at different intervals.

4. If you use the thread-pool reactor (ACE_TP_Reactor) the handlers actually share a pool of threads rather than just one. *current_time* is still important, however, because you will likely register more handlers than the number of threads in the pool. Because of this shared nature, if you have timers that must be handled at exactly the timeout interval then a dedicated thread or process might be your only option.

We begin by defining a TemperatureSensor object to represent each physical sensor we wish to query:

```
class TemperatureSensor
{
public:
    TemperatureSensor( const char * location )
            : location_(location),
              count_(0),
              temperature_(0.0)
        // ...
    {
    }

    const char * location()
    {
        return this->location_;
    }

    int querySensor()
    {
        // ...
        return ++this->count_;
    }

    float temperature()
    {
        return this->temperature_;
    }

private:
    const char * location_;
    int count_;
    float temperature_;
    // ...
};
```

The details have been omitted since we're more interested in talking about the reactor than communicating with remote sensors. *querySensor()* is the primary method of the object. It will contact the sensor to get current temperature information and store that in the member variable *temperature_*.

Our design requires us to use a single event handler to query all of the sensors. To accomodate this, the *handle_timeout()* method will expect it's second parameter to be a pointer to a TemperatureSensor instance.[5]

```
class TemperatureQueryHandler : public ACE_Event_Handler
{
public:
    TemperatureQueryHandler()
            : ACE_Event_Handler(),
              counter_(0),
              averageTemperature_(0.0)
        // ...
    {
    }

    int handle_timeout (const ACE_Time_Value &current_time,
                        const void * arg)
    {
        time_t epoch = ((timespec_t)current_time).tv_sec;

        TemperatureSensor * sensor =
            (TemperatureSensor *)arg;

        int queryCount = sensor->querySensor();

        this->updateAverageTemperature( sensor );

        ACE_DEBUG(( LM_INFO,
                    "%s\t"
                    "%d/%d\t"
                    "%.2f/%.2f\t"
                    "%s\n",
                    sensor->location(),
                    ++this->counter_,
                    queryCount,
                    this->averageTemperature_,
                    sensor->temperature(),
                    ACE_OS::ctime(&epoch) ));
```

5. If our design allowed us to use a handler instance per sensor the handler would likely have a TemperatureSensor member variable. Part of the reason for the one-handler design, however, is to make it easy to maintan an average temperature. Of course, the primary reason for our design is to illustrate the use of *handle_timeout()'s* second parameter.

```
        return 0;
    }

private:
    void updateAverageTemperature( TemperatureSensor * sensor )
    {
        // ...
    }

    int counter_;
    float averageTemperature_;
};
```

Our new *handle_timeout()* method begins by casting the opaque *arg* parameter to a TemperatureSensor pointer. Once we have the TemperatureSensor pointer we can use the instance's *querySensor()* method to query the physical device. Before printing out some interesting information about the sensor and the event handler we update the average temperature value.

As you can see, customizing the event handler to expect state data in the *handle_timeout()* method is quite easy. Registering the handler with state data is likewise easily done. First, of course, we must create the handler:

```
TemperatureQueryHandler * temperatureMonitor =
    new TemperatureQueryHandler();
```

Next we can register the handler to monitor the kitchen temperature:

```
TemperatureSensor * sensorOne =
    new TemperatureSensor( "Kitchen" );
ACE_Reactor::instance()->schedule_timer( temperatureMonitor,
                                         sensorOne,
                                         initialDelay,
                                         intervalOne );
```

We can then register the same handler instance with another TemperatureSensor instance to monitor the Foyer:

```
    TemperatureSensor * sensorTwo =
        new TemperatureSensor( "Foyer" );
    ACE_Reactor::instance()->schedule_timer( temperatureMonitor,
                                             sensorTwo,
                                             initialDelay,
                                             intervalTwo );
```

7.3.3 Using the TimerId

The return value of *schedule_timer()* is an opaque value known as the timer id. With this value you can reset the timer's interval or cancel a timer alltogether. In our interval reset example we have created a signal handler which, when invoked, will increase the timer's interval.

```
class SigintHandler : public ACE_Event_Handler
{
public:
    SigintHandler( long timerId, int currentInterval )
            : ACE_Event_Handler(),
              timerId_(timerId),
              currentInterval_(currentInterval)
    {
    }
    int handle_signal (int,
                       siginfo_t * = 0,
                       ucontext_t * = 0)
    {
        ACE_DEBUG(( LM_INFO,
                    "Resetting interval of timer %d to %d\n",
                    this->timerId_,
                    ++this->currentInterval_));

        ACE_Time_Value newInterval( this->currentInterval_ );

        ACE_Reactor::instance()->reset_timer_interval(
            this->timerId_, newInterval );

        return 0;
    }
```

```
private:
    long timerId_;
    int currentInterval_;
};
```

Our SigintHandler's constructor is given the timerId of the timer we wish to reset as well as the current interval. Each time the resetter's *handle_signal()* is called it will increase the interval by one second.

We schedule the to-be-reset handler as before but now we keep the return value of *schedule_timer()*:

```
MyTimerHandler * handler = new MyTimerHandler();
long timerId =
    ACE_Reactor::instance()->schedule_timer( handler,
                                             0,
                                             initialDelay,
                                             interval );
```

We can then provide this timerId value to our SigintHandler instance:

```
SigintHandler * handleSigint =
    new SigintHandler( timerId, 5 );
ACE_Reactor::instance()->register_handler( SIGINT,
                                           handleSigint );
```

Another thing you can do with the timerId is cancel a timer. To illustrate this we will modify (and rename) our SigintHandler to cancel the scheduled timer when SIGTSTP is received.

```
class SignalHandler : public ACE_Event_Handler
{
public:
    SignalHandler( long timerId, int currentInterval )
            : ACE_Event_Handler(),
              timerId_(timerId),
              currentInterval_(currentInterval)
    {
    }
    int handle_signal (int sig,
                       siginfo_t * = 0,
                       ucontext_t * = 0)
    {
```

```
        if( sig == SIGINT )
        {
            ACE_DEBUG(( LM_INFO,
                        "Resetting interval of timer %d to %d\n",
                        this->timerId_,
                        ++this->currentInterval_));

            ACE_Time_Value newInterval( this->currentInterval_ );

            ACE_Reactor::instance()->reset_timer_interval(
                this->timerId_, newInterval );
        }
        else if( sig == SIGTSTP )
        {
            ACE_DEBUG(( LM_INFO,
                        "Canceling timer %d\n",
                        this->timerId_));

            ACE_Reactor::instance()->cancel_timer( this->timerId_
);
        }

        return 0;
    }

private:
    long timerId_;
    int currentInterval_;
};
```

As before, SIGINT will increase the interval of the timer. SIGTSTP (typically ^Z) will cause the timer to be canceled. To use this functionality we simply register a SignalHandler instance with the reactor twice:

```
    SignalHandler * mutateTimer =
        new SignalHandler( timerId, 5 );
    ACE_Reactor::instance()->register_handler( SIGINT,
                                               mutateTimer );
    ACE_Reactor::instance()->register_handler( SIGTSTP,
                                               mutateTimer );
```

7.4 I/O

One of the most common uses of the reactor framework is to handle network I/O. A simple server scenario requires two event handlers: one to process incoming connection requests and a second to process a client connection. When designed this way your application will have N+1 event I/O-dedicated handlers registered with the reactor (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.

7.4.1 Accepting Connections

The first thing a server must be able to do is to accept a connection request from a potential client. In our Sockets chapter we used an ACE_SOCK_Acceptor instance. We'll be using that again here but this time it will be wrapped up in an event handler. By doing this we can have our application accept connections and process client requests seemingly at the same time.

Our event handler begins much like the others we've seen so far:

```
class ConnectionRequestHandler : public ACE_Event_Handler
{
public:
    ConnectionRequestHandler()
            : ACE_Event_Handler()
    {
        // ...
    }
```

The next part of our event handler is a simple *open()* method where we prepare the ACE_SOCK_Acceptor member variable (*acceptor_*) for work. From our original non-reactor based server we know that the acceptor's *open()* method requires an ACE_INET_Addr so we provide that as a parameter to our handler's *open()*.

```
    int open( ACE_INET_Addr & addr )
    {
        if( this->acceptor_.open( addr, 1 ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                "%p\n",
```

```
                                    "acceptor open"), -1);

    return 0;
}
```

In addition to the address at which we hope to receive connection requests, we've also given the acceptor's *open()* method a *reuse_addr* flag set to true. Setting the flag this way will allow our acceptor to open even if there are already some sockets connected at our designated listen port. This is generally what you will want to do because even sockets in the *FIN_WAIT* state can prevent an acceptor from opening successfully.

Before we can continue there is an important housekeeping method we must define on our event handler: *get_handle()*. Recall that under the covers the reactor framework will use *select()*, *poll()*, *WaitForMultipleObject()* or some similar operating system function. These low-level functions know nothing of higher level constructs such as ACE_Event_Handler and its derivatives. Thus, the event handler must provide some mechanism for getting the low-level (and to us opaque) handle from the event handler instance. Our solution is to simply forward the request on to our ACE_SOCK_Acceptor instance:

```
ACE_HANDLE get_handle(void) const
{
    return this->acceptor_.get_handle();
}
```

We now come to the *handle_input()* method. When we get to *main()* in a little bit we will register our ConnectionRequestHandler with the reactor using *ACE_Event_Handler::READ_MASK*[6]. This registration will tell the reactor that we wish to be notified any time the handle (from *get_handle()*) associated with the event handler has some data to be read. Such registration will cause the event handler's *handle_input()* method to be invoked. In the case of an ACE_SOCK_Acceptor, any attempt by a client to connect is construed as "data to be read".

6. There are several other mask values defined in the ACE_Event_Handler class a couple of which we will explore in this chapter. The reader is encouraged to consult the documentation for the other masks and their usage.

```
int handle_input( ACE_HANDLE )
{
    ClientHandler * client;
    ACE_NEW_RETURN( client, ClientHandler(), -1 );
```

Similar to the way a signal handler can be registered for multiple signals, an I/O handler can be registered for multiple I/O handles. In our simple example our ConnectionRequestHandler will only concern itself with one ACE_SOCK_Acceptor. Therefore, we will ignore the ACE_HANDLE parameter of our *handle_input()* method.

The first thing our *handle_input()* method does is to create a ClientHandler instance. ClientHandler is also an event handler and will be used to process the input from connected clients. Its definition follows in Processing Input on page 160.

In our non-reactor based server (Section 6.2 on page 125) we saw how to use the *accept()* method of the ACE_SOCK_Acceptor to accept an incoming connection request. Our event-handler approach must do the same thing. Our ClientHandler instance, however, is an event handler, not an ACE_SOCK_Stream which can be fed directly into the *accept()* method. To get around this issue we have defined a *peer()* method on the ClientHandler which will return a reference to the underlying ACE_SOCK_Stream. In doing so we have mirrored the framework's use of the *get_handle()* method where concrete data types need to be made available between high-level constructs.

```
    if( this->acceptor_.accept( client->peer() ) == -1 )
        ACE_ERROR_RETURN (( LM_ERROR,
                            "%p\n",
                            "accept"), -1);
```

If we succeed in our attempt to accept the client's connection we will then notify the ClientHandler that it should get ready to do some work. This is done through the *open()* method. We will rely on the *open()* method to register the ClientHandler instance with the reactor.

```
    if( client->open() == -1 )
        ACE_ERROR_RETURN (( LM_ERROR,
                            "%p\n",
                            "client open"), -1);

    return 0;
}
```

If everything has gone well until this point our *handle_input()* will return the typical zero. If something has gone badly we will have returned -1. A -1 return from *handle_input()* signals the reactor that the event handler no longer wishes to be invoked. The reactor will then invoke the handler's *handle_close()* method to take care of any necessary cleanup. We will omitt *handle_close()* on our ConnectionRequestHandler because we can easily take care of its cleanup in *main()*. We will, however, revisit this interesting method in our ClientHandler.

7.4.2 Processing Input

In the section above our acceptor handler creates an instance of the mysterious ClientHandler object. In the same way ConnectionRequestHandler is an event handler wrapped around an ACE_SOCK_Acceptor, the ClientHandler is an event handler wrapped around an ACE_SOCK_Stream. We begin the ClientHandler in the usual way:

```
class ClientHandler : public ACE_Event_Handler
{
public:
    ClientHandler()
            : ACE_Event_Handler()
    {
        // ...
    }
```

The constructor is really quite simple. We could even omitt it and let the compiler create a default for us but it is generally good style to go ahead and stub it in in case you need it later.

The *open()* method comes next. This is where we prepare the ClientHandler instance for interaction with the client. In this very simple example our only task is to register with the reactor. We again register using the *READ_MASK* so that we will be notified when there is data to be read on the socket. Thus, when the client sends data our *handle_input()* method will be invoked.

```
    int open( void )
    {
        return
            ACE_Reactor::instance()->register_handler(
                this, ACE_Event_Handler::READ_MASK );
    }
```

As with the ConnectionRequestHandler we must make the opaque *ACE_HANDLE* value available to the reactor framework. Recall that this what will be registered with the underlying *select(), poll(), WaitForMultipleObjects()* or equivalent operating system function. Our *peer_* member variable is an ACE_SOCK_Stream and we can simply forward the *get_handle()* method call on to it.

```
ACE_HANDLE get_handle(void) const
{
    return this->peer_.get_handle();
}
```

Now that we have all of the housekeeping duties out of the way we can get to the interesting part: *handle_input()*. We begin our *handle_input()* method by defining a buffer into which we will read some data and a place to keep the byte-count[7].

```
int handle_input( ACE_HANDLE )
{
    char buf[64];
    int bytesReceived;
```

The second stage of *handle_input()* is the most critical. This is where we receive the data sent by the client. Using the *recv()* method of the underlying ACE_SOCK_Stream, we read up to 63 bytes from the client. The reactor will only invoke *handle_input()* if something has happened on the socket. With that in mind, we know that if we don't receive at least one byte of data then the stream is not in a usable state.

```
    if( (bytesReceived =
         this->peer_.recv( buf, sizeof(buf)-1 )) < 1 )
    {
        ACE_DEBUG(( LM_INFO,
                    "ClientHandler handle_input: "
                    "Received %d bytes. Leaving.\n",
```

7. If you are using the basic reactor (e.g. -- not a multi-threaded reactor) then your event handler will be more efficient if the buffer and byte counter are member variables. This would prevent them being creating on the stack with each call to *handle_input()*. If you are using a multi-threaded reactor you might consider putting these things into thread-specific storage.

```
                    bytesReceived
                    ));
        return -1;
    }
```

If the stream is in a bad state then our handler will display an appropriate message and return -1 to the reactor. A return value of -1 from any of the *handle_*()* methods informs the reactor that something bad has happened and the event handler can no longer be used. The reactor will then invoke *handle_close()* on the event handler[8] so that the handler may perform any necessary cleanup.

If the stream is not in a bad state then we perform a quick null-termination of the data and print it in our debug message. We then return zero to tell the reactor that we are ready to do more work.

```
    buf[bytesReceived] = 0;
    ACE_DEBUG(( LM_INFO,
                "ClientHandler handle_input: %s\n",
                buf
                ));

    return 0;
}
```

By convention, *handle_close()* is responsible for any cleanup that should be done by the event handler. In our simple example a ClientHandler is only ever created by the ConnectionRequestHandler. Furthermore, it is always created there via the *new* operator, never on the stack. As long as these assumptions are not violated we can safely delete the event handler's instance from within it's *handle_close()* method.[9]

```
int handle_close( ACE_HANDLE,
                  ACE_Reactor_Mask)
{
    ACE_DEBUG(( LM_INFO,
```

8. One of the event handler mask values is *DONT_CALL*. If you provide this flag to the reactor's *register_handler()* method when registering the event handler then the handler's *handle_close()* method will not be invoked for cleanup.

9. The ACE_Dynamic class can be used to determine whether or not an object has been dynamically created or created on the stack. Using this technique we can then invoke *delete this* conditionally and not need to worry about dangerous assumptions.

```
                    "ClientHandler handle_close\n"
                    ));

        delete this;

        return 0;
    }
```

We wrap up our ClientHandler with the *peer()* method used by our ConnectionRequestHandler's *handle_input()* method.

```
    ACE_SOCK_Stream & peer()
    {
        return this->peer_;
    }

private:
    ACE_SOCK_Stream peer_;
};
```

Our only member variable is the ACE_SOCK_Stream instance. The sock stream is the concrete representation of the connected peer just as it was in the previous chapter.

7.4.3 Using ACE_Svc_Handler<>

When we derive directly from ACE_Event_Handler for purposes of handing a connection we find ourselves writting a lot of code that isn't directly relevant to our application. While the *open()*, *peer()*, *handle_close()* and such methods are all required the real functionality is centered in the *handle_input()* method. Recognizing this, the ACE team created teh ACE_Svc_Handler<> template to handle much of these housekeeping duties for us.[10] Our use of ACE_Svc_Handler<> begins with the class signature and constructor:

```
class ClientHandler :
    public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
public:
```

10.ACE_Svc_Handler<> also does much more as we'll see in Chapter 19

```
typedef ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH> super;

  ClientHandler()
          : super()
  {
      // ...
  }
```

The ACE_Svc_Handler<> is quite flexible and supports a thread-safe locking mechanism should we require it. In our simple server we only intend to have one thread so we can use the ACE_NULL_SYNCH macro. This will cause the ClientHandler to use no-op locking objects thus removing unnecessary overhead. The ACE_SOCK_STREAM macro provides ACE_Svc_Handler<> with the necessary objects to use an ACE_SOCK_Stream in our connection.[11]

For convenience we've creatd the *super* typedef. This gives us much more readable code when initializing the baseclass in our constructor or invoking a baseclass method from our derivative.

Technically speaking we don't need to define the constructor since it doesn't do anything useful. But, again, it is good practice to stub it in (along with the destructor) in case we need it in the future. You might, for instance, want to add some ACE_DEBUG() calls in the constructor/destructor during development.

The remainder of our class is devoted entirely to the *handle_input()* method. This is, in fact, exactly like the previous version:

```
int handle_input( ACE_HANDLE )
{
    char buf[64];
    int bytesReceived;

    if( (bytesReceived =
         this->peer_.recv( buf, sizeof(buf)-1 )) < 1 )
    {
        ACE_DEBUG(( LM_INFO,
                    "ClientHandler handle_input: "
                    "Received %d bytes. Leaving.\n",
                    bytesReceived
```

11. You might think that all we need to do is use ACE_SOCK_Stream for the templat e parameter. However, internally, the template uses a nested class of ACE_SOCK_Stream. Some compilers can handle this, some can't. the ACE_SOCK_STREAM macro is conditionally set based on compiler capabilities such that everything works out right.

```
                        ));
            return -1;
        }

        buf[bytesReceived] = 0;
        ACE_DEBUG(( LM_INFO,
                    "ClientHandler handle_input: %s\n",
                    buf
                    ));

        return 0;
    }
};
```

As you can see, use of the ACE_Svc_Handler<> greatly simplifies our event handler and allows us to focus on the actual problem we need to solve rather than getting distracted by all of the connection-management issues.

7.4.4 Using ACE_Acceptor<>

Similar to the original ClientHandler, our current ConnectionRequestHandler is entirely focused on managing the connection with the peer. There is nothing at all in the ConnectionRequestHandler that has anything to do with our application logic. The ACE team comes to our rescue again by providing the ACE_Acceptor<> template. This template provides one-hundred percent of the functionality we previously coded by hand.

```
typedef
    ACE_Acceptor<ClientHandler,ACE_SOCK_ACCEPTOR>
    ConnectionRequestHandler;
```

We had to introduce ACE_Svc_Handler<> before ACE_Acceptor<> because the latter depends on some method signatures provided by the former. Together, the two make it easy to create a powerful server while you, the programmer, gets to focus entirely on the problem logic rather than spending time on the connection handling logic.

7.4.5 Details: *main()*

Now that we have all of the building blocks we can quite easily complete our server with a simple *main()*:

```
int main( int, char ** )
{
    ConnectionRequestHandler * acceptor =
        new ConnectionRequestHandler();

    ACE_INET_Addr addr("HAStatus");
    if( acceptor->open(addr) == -1 )
        return 100;

    ACE_Reactor::instance()->register_handler(
        acceptor, ACE_Event_Handler::READ_MASK );

    ACE_Reactor::instance()->run_reactor_event_loop();

    delete acceptor;

    return 0;
}
```

We first create a ConnectionRequestHandler instance and open it to listen on the *HAStatus* port.[12] If the *open()* succeeds we register the object instance with the reactor. In registration we provide the *ACE_Event_Handler::READ_MASK* flag which requests that the reactor invoke the event handler's *handle_input()* any time there is data available on the underlying handle.

When working with templates there is one last chore we must do before our application can be considered complete. We must give the compiler some help in applying the template code to create the real classes:

```
#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)
  template class
      ACE_Acceptor<ClientHandler, ACE_SOCK_ACCEPTOR>;
  template class
      ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;
#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)
  pragma instantiate
```

12. *HAStatus* is presumably defined in /etc/services or the Win32 equivalent. We could alternatively use any of the other ACE_INET_Addr constructors.

```
      ACE_Acceptor<ClientHandler, ACE_SOCK_ACCEPTOR>;
  pragma instantiate
      ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;
#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

Because different compilers behave differently we must use different techniques for template instantiation depending on what the compiler is capable of. This can get tedious quickly but is necessary when creating a truly portable application.

7.4.6 A Reactor-based Client

A reactor can also be used on the client side of a connection if your application lends itself to such a thing. In this section we've contrived a reasonably simple client to illustrate this architecture.

Our Client object, like our most clever ClientHandler, derives from ACE_Svc_Handler<>. This allows us to focus on the task at hand without worrying too much about the housekeeping details. It begins very much like the ClientHandler:

```
class Client :
    public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
public:
    typedef
        ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
        super;

    enum {
        MAX_ITERATIONS = 7
    };

    Client()
            : super(),
              iterations_(0),
              bytes_sent_(-1),
              bytes_to_send_(-1)
    {
        // ...
    }
```

The *iterations_* member and *MAX_ITERATIONS* constant are used to give the client a clean way to exit. We'll see that in a moment. Member variables *bytes_sent_* and *bytes_to_send_* are used by our *handle_output()* method and will be explained in detail later.

The ACE_Svc_Handler<> *open()* method is really designed to work nicely with the ACE_Acceptor<>. Since we're not doing that here we have to override *open()* to do something more appropriate to the client-side of things.

```
int open( const ACE_INET_Addr & addr )
{
    ACE_SOCK_Connector connector;
    if( connector.connect( this->peer(), addr ) == -1 )
    {
        ACE_ERROR_RETURN(( LM_ERROR,
                           "%p\n",
                           "connect"), -1 );
    }

    return 0;
}
```

Our specialized *open()* only needs to use the ACE_SOCK_Connector to connect the embedded ACE_SOCK_Stream (from our template parameter) to the server.

In our contrived example we've chosen to use *handle_output()* to send data to the server. When an event handler is registered using the *ACE_Event_Handler::WRITE_MASK* flag the reactor framework will invoke *handle_output()* whenever the associated handle (taken from the event handler's *get_handle()* method) is available for sending data. The operative word here is *whenever*. This can lead to surprising and unpleasant behavior if you think that the reactor will only invoke *handle_output()* once and then wait for you to write some data!

The *handle_output()* begins with the appropriate signature and a brief message:

```
int handle_output( ACE_HANDLE )
{
    ACE_DEBUG(( LM_INFO,
                "Client handle_output\n"
                ));
```

We then inspect the *bytes_to_send_* member variable. If it has a value less than zero we take that to be a special case which begins our data transmission. The *output_buffer_* variable is given some interesting data, *bytes_to_send_* is given the length of the data block and *bytes_sent_* is initialized to zero.

```
if( this->bytes_to_send_ < 0 )
{
    ACE_OS::sprintf( this->output_buffer_,
                     "Iteration %d\n",
                     ++this->iterations_ );
    this->bytes_to_send_ =
        ACE_OS::strlen( this->output_buffer_ );
    this->bytes_sent_ = 0;
}
```

The next section of *handle_output()* checks to see if the entire buffer has been sent. That is, are the number of *bytes_sent_* equal to (or greater than) the number of *bytes_to_send_*. If so we reset our two counters to their original values and inform the reactor framework that we no longer wish to have *handle_output()* called when it would have otherwise done so. The *DONT_CALL* mask tells the framework to not call the *handle_close()* method. That is important in this case because, as you'll see in a moment, the same handler instance is also registered to receive notifications when data has been sent by the server *and* as a scheduled timer[13].

```
if( this->bytes_sent_ >= this->bytes_to_send_ )
{
    this->bytes_to_send_ = -1;
    this->bytes_sent_ = -1;

    this->reactor()->remove_handler(
        this,
        ACE_Event_Handler::WRITE_MASK |
        ACE_Event_Handler::DONT_CALL );

    return 0;
}
```

13.This particular example is rather contrived and is only intended to serve as an example of the mechanics of the reactor. It is not necessarily intended to be used as an architectural example.

If the buffer has not yet been completely sent to the server then our next action is to send a byte of data. We first create a pointer into the output buffer offset by the number of bytes already sent. We then use the *send()* method of the underlying ACE_SOCK_Stream to send the byte at that location. *send()* will return the number of bytes sent or *-1* on error. If the server is too busy and cannot take our data *send()* will return zero.

```
        char * buf =
            this->output_buffer_ +
            this->bytes_sent_;

        int bytes_sent = this->peer().send( buf, 1 );
```

If there was an error (other than flow-control) sending the data then we set *bytes_sent_* equal to *bytes_to_send_* which will cause the next invocation of *handle_ouput()* to think the process is complete. We also print an error message and return. Note that we return 0 so that *handle_output()* will be called again. If we returned *-1* at this point the reactor would invoke *handle_close()* and that is not at all what we want.

```
        if( bytes_sent < 0 )
        {
            this->bytes_sent_ = this->bytes_to_send_;

            ACE_ERROR_RETURN(( LM_ERROR,
                               "%p\n",
                               "send"), 0 );
        }
```

If everything went well with the send we increment our *bytes_sent_* counter and return from *handle_output()*.

```
        this->bytes_sent_ += bytes_sent;

        return 0;
    }
```

In a more realistic example we might use this approach to send a huge (e.g. - many, many megabytes) quantity of data. In that situation we would ask *send()* to send as much of the remaining data as possible knowing that it will send what it can and return us that byte count. A sample application along these lines would

likely involve mapping a file to some memory location and treating its contents as an in-memory buffer.

In *main()* our event handler will be registered using both *READ_MASK* and *WRITE_MASK* flags. As soon as the server is ready, *handle_output()* will be fired and the datastream sent to the server. The server will "process" the data and send the results back to the client where *handle_input()* will pickup the "processed" data.

```
int handle_input( ACE_HANDLE )
{
    char buf[64];
    int bytesReceived;

    if( (bytesReceived =
         this->peer_.recv( buf, sizeof(buf)-1 )) < 1 )
    {
        ACE_DEBUG(( LM_INFO,
                    "Client handle_input: "
                    "Received %d bytes. Leaving.\n",
                    bytesReceived
                    ));
        return -1;
    }

    buf[bytesReceived] = 0;
    ACE_DEBUG(( LM_INFO,
                "Client handle_input: %d: %s\n",
                bytesReceived,
                buf
                ));

    // Re-enable handle_output in three seconds.
    ACE_Time_Value one_shot( 3 );
    this->reactor()->schedule_timer(
        this, 0, one_shot );

    return 0;
}
```

Once *handle_input()* has received the data it applys a null-terminator and displays the message. At this point in our program flow *handle_output()* has likely been disabled because it already sent all of its data. In order to start another itera-

tion a one-shot timer is scheduled using this event handler instance. The *handle_timeout()* method will re-register the event handler using the *WRITE_MASK* to enable *handle_output()* thus starting the whole cycle over once more.

```
int handle_timeout( const ACE_Time_Value &, const void * )
{
    if( this->iterations_ >= MAX_ITERATIONS )
    {
        this->reactor()->end_event_loop();
    }
    else
    {
        this->reactor()->register_handler(
            this, ACE_Event_Handler::WRITE_MASK );
    }

    return 0;
}
```

The *iterations_* member variable is used to break the reactor's loop and allow the program to exit gracefully. If we had simply exited any of the *handle_*()* methods with *-1* we wouldn't have left the application running because the reactor loop will continue even if there are no handlers registered.

The remainder of the client is the *main()* function and template instantiation instructions:

```
int main( int, char ** )
{
    Client * client = new Client();

    ACE_INET_Addr addr("HAStatus");
    if( client->open(addr) == -1 )
        return 100;

    ACE_Reactor::instance()->register_handler(
        client,
        ACE_Event_Handler::READ_MASK |
        ACE_Event_Handler::WRITE_MASK
        );

    ACE_Reactor::instance()->run_reactor_event_loop();

    return 0;
```

```
}

#if defined (ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION)
  template class
      ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;
#elif defined (ACE_HAS_TEMPLATE_INSTANTIATION_PRAGMA)
  pragma instantiate
      ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>;
#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

To illustrate this particular approach we've made a few modifications to the server's original ClientHandler. Unlike our previous server, this version returns its input to the peer. This gives our client application's Client object something to look forward to.

```
class ClientHandler :
    public ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
{
public:
    typedef
        ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_NULL_SYNCH>
        super;

    ClientHandler()
            : super()
    {
        ACE_DEBUG(( LM_INFO,
                    "ClientHandler ctor\n"
                    ));
    }

    // ...

    int handle_input( ACE_HANDLE )
    {
        char buf[64];

        int bytes_received =
            this->peer().recv( buf, sizeof(buf)-1 );

        if( bytes_received < 1 )
        {
            ACE_DEBUG(( LM_INFO,
                        "ClientHandler handle_input: "
                        "Received %d bytes. Leaving.\n",
```

```
                    bytes_received
                    ));
            return -1;
        }

        buf[bytes_received] = 0;
        ACE_DEBUG(( LM_INFO,
                    "ClientHandler handle_input: %s\n",
                    buf
                    ));

        const char * return_buf =
            this->process_data( buf );

        int bytes_sent =
            this->peer_.send_n( return_buf,
                                strlen(return_buf) );

        ACE_DEBUG(( LM_INFO,
                    "ClientHandler handle_timeout: %s\n",
                    buf ));

        if( bytes_sent < 0 )
            ACE_ERROR_RETURN(( LM_ERROR,
                               "%p\n",
                               "send"), -1 );

        return 0;
    }

    // ...
};
```

7.5 Win32 Handle Signaling

7.6 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. You may even create your own extension of one of the existing implementations. For example, to use the thread-pool reactor implementation your application would first create the reactor instance:

```
ACE_TP_Reactor * reactor = new ACE_TP_Reactor(...);
```

Then tell the singleton to use this instance rather than the default:

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

The ‘1’ parameter allows the singleton to delete the ACE_TP_Reactor instance at program termination time. This is a good idea to prevent memory leaks and allow for a clean shutdown.

7.6.1 Select Reactor

The ACE_Select_Reactor is the default implementation used on POSIX and POSIX-like systems. The *select()* or *poll()* systemcall is ultimately used on these systems to wait for activity.

7.6.2 WFMO Reactor & Message WFMO Reactor

On Win32 systems the ACE_WFMO_Reactor is the default implementation used by the reactor framework. The Win32 function *WaitForMultipleObjects()* is used in place of *select()* or *poll()*. If your application will be a COM/DCOM server then you shoud use the ACE_Msg_WFMO_Reactor instead because of its ability to react on Windows messages.

7.6.3 Thread Pool Reactor

The ACE_TP_Reactor extends the ACE_Select_Reactor to allow it to operate in multiple threads at the same time. The TP_Reactor doesn’t create the threads, you are still responsible for that. Once you have your threads running you invoke the typical

```
ACE_Reactor::instance()->run_reactor_event_loop()
```

in one or more of your threads. One of the threads will take ownership of the handle-set and wait for an event while the others wait on it. When activity occurs, the owning thread will pass ownership to another thread and process the activity. This pattern continues until the reactors are all shutdown at which point the threads (and program) can exit.

7.6.4 Priority Reactor

The ACE_Priority_Reactor is another extension of the ACE_Select_Reactor. This implementation takes advantage of the *priority()* method on the ACE_Event_Handler class. When an event handler is registered with this reactor it is placed into a priority-specific bucket. When events take place they are dispatched in their priority order. This allows higher-priority events to be processed first.

7.6.5 GUI Integrated Reactors

Recognizing the need to write reactor-based GUI applications, the ACE community has created several reactor extensions for use with the X-Window system. Each of these extends the ACE_Select_Reactor to work with a specific toolkit. By using these reactors your GUI application can remain single-threaded yet still respond to both GUI events (e.g. - button presses) and your own application events.

QuickTime Reactor

The ACE_QtReactor extends both the ACE_Select_Reactor and the QuickTime library's QObject class. Rather than using *select()* or *poll()*, the *QtWaitForMultipleEvents()* function is used.

FastLight Reactor

The ACE_FlReactor integrates with the FastLight toolkit's *Fl::wait()* method.

Tk Reactor

The ACE_TkReactor provides reactor functionality around the popular Tcl/Tk library. The underlying Tcl/Tk method used is *Tcl_DoOneEvent()*.

Xt Reactor

Last, but not least, is the ACE_XtReactor which integrates with the X Toolkit library using *XtWaitForMultipleEvents()*.

7.7 Summary

The reactor framework is a very powerful and flexible system for creating a seemingly multi-threaded application without incurring the overhead of multiple threads. At the same time, you can use the reactor in a multi-threaded application and have the best of both worlds. A single reactor instance can easily handle activity of timers, signals and I/O events. Handling of I/O is not limited simply to sockets as shown in this chapter. A reactor-based application can handle I/O from anything that can provide an ACE_HANDLE representation: named pipes, UNIX-domain sockets, UDP sockets, serial and parallel IO 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:

- **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.
- **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.

Asynchronous I/O has been in use for many years on OS platforms such 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. It 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 since 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 because:

- Threading policy is 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, and further complexity.
- Synchronization performance penalty. Overhead due 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.

Asynchronous I/O, also known as proactive I/O, is a more scalable way to alleviate reactive I/O bottlenecks without necessarily introducing the complexity and overhead of multithreading. 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. There are two distinct steps in the proactive I/O model:

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 in any of the following situations:

- The IPC mechanisms in use (e.g. Windows named pipes) require it.
- The application can benefit significantly from parallel I/O operations.
- Reactive model limitations (limited handles or performance) prevent its use.

8.2 How to Send and Receive Data

The procedure for sending and receiving data asynchronously is a bit different than when using synchronous transfers. We'll look at an example and then 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 classes that are highly related, 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 and you can use it to keep some context as we progress through the chapter.

The following shows the declaration of a class that receives data using the Proactor framework and echoes it back to the sender. It introduces the primary classes involved in initiating and completing I/O requests on a connected TCP/IP socket.

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

class HA_PROACTIVE_Export 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_;
  ACE_Asynch_Write_Stream writer_;
};
```

This example begins by including the necessary header files for the Proactor framework classes that this example uses. These are:

- `ACE_Service_Handler`: the target class for creation of new service handlers in the Proactor framework, similar to the role played by `ACE_Svc_Handler` in the Acceptor-Connector framework
- `ACE_Handler`: the parent class of `ACE_Service_Handler`, which defines the interface for handling asynchronous I/O completions via the Proactor framework. `ACE_Handler` is analogous to `ACE_Event_Handler` in the Reactor framework.
- `ACE_Asynch_Read_Stream`: I/O factory class for initiating read operations on a connected TCP/IP socket

- `ACE_Asynch_Write_Stream`: I/O factory class for initiating write operations on a connected TCP/IP socket
- `Result`: each I/O factory class defines a nested `Result` class to contain the result of each operation the factory initiates. Since the initiation and completion of each asynchronous I/O operation are separate and distinct events, some mechanism is needed to "remember" the operation parameters and relay them, along with the result, to the completion handler.

8.2.1 Setting up the Handler and Initiating I/O

When a TCP connection is opened, the handle of the new socket should be passed to the handler object (in this example's case, `ACE_Service_Handler`) because it is a convenient point of control for the socket's lifetime and is most often the class from which I/O operations are invoked. When using the Proactor framework's asynchronous connection establishment classes (we'll look at these in Section 8.3), the `ACE_Service_Handler::open()` hook method is called when a new connection is established. Our example's `open()` hook follows:

```
void
HA_Proactive_Service::open (ACE_HANDLE h, ACE_Message_Block&)
{
  this->handle (h);
  if (this->reader_.open (*this) != 0 ||
      this->writer_.open (*this) != 0   )
    {
      ACE_ERROR ((LM_ERROR, ACE_TEXT ("%p\n"),
                  ACE_TEXT ("HA_Proactive_Service open")));
      delete this;
      return;
    }

  // Do something with this 1024 magic number.
  ACE_Message_Block *mb;
  ACE_NEW_NORETURN (mb, ACE_Message_Block (1024));
  if (this->reader_.read (*mb, mb->space ()) != 0)
    {
      ACE_ERROR ((LM_ERROR, ACE_TEXT ("%p\n"),
                  ACE_TEXT ("HA_Proactive_Service begin read")));
      mb->release ();
      delete this;
      return;
    }
```

```
  // mb is now controlled by Proactor framework.
  return;
}
```

Right at the beginning, the new socket's handle is saved using the inherited `ACE_Handler::handle()` method. This stores the handle in a convenient place for, among other things, access by the `HA_Proactive_Service` destructor shown on page 182. This is part of the socket handle's lifetime management implemented in this class.

After storing the socket handle, this method initializes the `reader_` and `writer_` I/O factory objects in preparation for initiating I/O operations. The complete signature of the `open()` method on both classes is:

```
int open (ACE_Handler &handler,
          ACE_HANDLE handle = ACE_INVALID_HANDLE,
          const void *completion_key = 0,
          ACE_Proactor *proactor = 0);
```

This first argument represents the completion handler for operations initiated by the factory object. The Proactor framework will call back to this object when I/O operations initiated via the factory object complete. That's why the handler object is referred to as a *completion handler*. In our example, `HA_Proactive_Service` is a descendant of `ACE_Handler` and will be the completion handler for both read and write operations, so `*this` is the handler argument. All other arguments are defaulted. Since we don't pass a handle value, the I/O factories will call back to `HA_Proactive_Service::handle()` to obtain the socket handle. This is another reason we stored the handle value immediately on entry to `open()`.

The `completion_key` argument is only used on Windows. The `proactor` argument is also defaulted. In this case, a process-side singleton `ACE_Proactor` object will be used. If a specific `ACE_Proactor` instance is needed, then the `proactor` argument must be supplied.

The last thing our `open()` hook method does is initiate a read operation on the new socket by calling the `ACE_Asynch_Read_Stream::read()` method. The signature for `ACE_Asynch_Read_Stream::read()` is:

```
int read (ACE_Message_Block &message_block,
          size_t num_bytes_to_read,
          const void *act = 0,
          int priority = 0,
          int signal_number = ACE_SIGRTMIN);
```

The most obvious difference between asynchronous read (and write, as we'll see) operations and their synchronous counterparts is that an `ACE_Message_Block` is specified for the transfer rather than a buffer pointer or `iovec`. This makes buffer management easier since you can take advantage of `ACE_Message_Block`'s capabilities and integration with other parts of ACE such as `ACE_Message_Queue`. When a read is initiated, data is read into the block starting at the block's write pointer, since the read data will be written into the block.

8.2.2 Completing I/O Operations

When the read completes, the Proactor framework calls the `handle_read_stream()` hook method. Our example's hook method is shown below:

```
void
HA_Proactive_Service::handle_read_stream
  (const ACE_Asynch_Read_Stream::Result &result)
{
  ACE_Message_Block &mb = result.message_block ();
  if (!result.success () || result.bytes_transferred () == 0)
    {
      mb.release ();
      delete this;
    }
  else
    {
      if (this->writer_.write (mb, mb.length ()) != 0)
        {
          ACE_ERROR ((LM_ERROR, ACE_TEXT ("%p\n"),
                      ACE_TEXT ("starting write")));
          mb.release ();
        }
      else
        {
          ACE_Message_Block *new_mb;
          ACE_NEW_NORETURN (new_mb, ACE_Message_Block (1024));
          this->reader_.read (*new_mb, new_mb->space ());
        }
    }
  return;
}
```

The passed-in `ACE_Asynch_Read_Stream::Result` refers to the object holding the results of the read operation. Each I/O factory class defines its own `Result` class to hold the results of operations initiated via that class. The message block used in the operation is referred to via the `message_block()` method. The Proactor framework automatically advances the block's write pointer to reflect the added data if there was any. The `handle_read_stream()` method above first checks to see if either the operation failed or completed successfully but read 0 bytes (as in synchronous socket reads, a 0-byte read indicates the peer has closed its end of the connection). If either of these cases is true, the message block is released and the handler object deleted. The handler's destructor will close the socket.

If the read operation read any data, we do two things:

1. Initiate a write operation to echo the received data back to the peer. Since the Proactor framework has already updated the message block's write pointer, we can simply use the block as-is. The read pointer is still pointing to the start of the data, and a write operation uses the block's read pointer to read data out of the block and write it on the socket.
2. Allocate a new `ACE_Message_Block` and initiate a new read operation to read the next set of data from the peer.

When the write operation completes, the the Proactor framework calls the following `handle_write_stream()` method:

```
void
HA_Proactive_Service::handle_write_stream
    (const ACE_Asynch_Write_Stream::Result &result)
{
  result.message_block ().release ();
  return;
}
```

Regardless of whether or not the write completed successfully, the message block that was used in the operation is released. If there is a broken socket, the previously initiated read operation will also complete with an error and `handle_read_stream()` will clean up the object and socket handle. More importantly, notice that the same `ACE_Message_Block` object was used to read data from the peer and echo it back. After it has been used for both oeprations, it is released.

The example presented in this section illustrates the following principles and guidelines for using asynchronous I/O in the ACE Proactor framework:

ACE_Message_Block is used for all transfers.

All read and write transfers use `ACE_Message_Block` rather than other types of buffer pointers and counts. This enables ease of data movement around other parts of ACE such as queueing data to an `ACE_Message_Queue` or other frameworks that reuse `ACE_Message_Queue` such as the ACE Task framework or the ACE Streams framework. Using the common message block class makes it possible for the Proactor framework to automatically update the block's read and write pointers as data is transferred, alleviating you of this tedious task. The class(es) involved in initiating and completing I/O operations mutually agree on how the blocks are allocated, statically or dynamically; however, it is generally more flexible to allocate the blocks dynamically.

Cleanup has very few restrictions, but must be managed carefully.

In the example above, the usual response to an error condition is to delete the handler object. After working with the ACE Reactor framework and its rules for event handler registration and cleanup, this "just delete it" simplicity may seem odd. Remember that there are no explicit handler registrations with the Proactor as there are with the Reactor framework.[1] The only connection between the Proactor and the completion handler object is an outstanding I/O operation. Therein lies a very important restriction on completion handler cleanup. If there are any outstanding I/O operations that refer to an `ACE_Handler` object, that object must not be deleted. When the I/O operation(s) complete, the Proactor framework will issue callback(s) to the handler and it must be valid or your program's behavior will be undefined and almost surely not correct.

8.3

Each I/O factory class offers a `cancel()` method that can be used to attempt to cancel any outstanding I/O operations. Not all operations can be canceled, however. Different operating systems offer different levels of support for canceling operations, sometimes varying with I/O type on the same system. For example, many disk I/O requests that haven't started to execute can be canceled, but many socket operations cannot. Sometimes closing the I/O handle on which

1. Use of timers in the Proactor framework does require cleanup, however. The cleanup requirements for timer use in the Proactor framework as similar to those for the Reactor framework.

the I/O is being performed will abort an I/O request, and sometimes not. It's often a good idea to keep track of the number of outstanding I/O requests and wait for them all to complete before destroying a handler.

Establishing Connections

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

- `ACE_Asynch_Acceptor` to initiate passive connection establishment.
- `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 our `HA_Proactive_Service` class in the example above) to handle the new connection. The `ACE_Service_Handler` class is the base class of all asynchronously-connected services in the ACE Proactor framework. It 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 straight-forward in its default case, and adds two hooks that you can extend its capabilities with. Let's look at an example using one of the hooks:

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

class HA_PROACTIVE_Export 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 ACE_Asynch_Acceptor. 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 to see that 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 =
    reinterpret_cast<struct in_addr*, remote.get_addr ()>;
  struct in_addr *local_addr =
    reinterpret_cast<struct in_addr*, local.get_addr ()>;
  if (inet_netof (local_addr) == net_inetof (remote_addr))
    return 0;
  return -1;
}
```

This check is fairly simple, and only work for IPv4 networks, but is an example of the hook's use. The handle of the newly-accepted socket is available via the `ACE_Asynch_Accept::Result::accept_handle()` method, so it is possible to do more involved checks that require data exchange. For example, an SSL handshake could be added at this point.

The other hook method available via `ACE_Asynch_Acceptor` is a protected virtual method, `make_handler()`. The Proactor framework calls this method to obtain a `ACE_Service_Handler` object to service the new connection. The default implementation is, essentially:

```
template <class HANDLER>
class ACE_Asynch_Acceptor : public ACE_Handler
...
protected:
  virtual HANDLER *make_handler (void)
      { return new HANDLER; }
```

If your application requires a different way of obtaining a handler, you should override the `make_handler()` hook method. For example, a singleton handler could be used, or you could keep a list of handlers in use.

To following shows how we use the HA_Proactive_Acceptor class we just described:

```
ACE_INET_Addr listen_addr;     // Set up with listen port
HA_Proactive_Acceptor aio_acceptor;
if (0 != aio_acceptor.open (listen_port,
                            0,     // bytes_to_read
                            0,     // pass_addresses
                            ACE_DEFAULT_BACKLOG,
                            1,     // reuse_addr
                            0,     // proactor
                            1))    // validate_new_connection
  ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                     ACE_TEXT ("acceptor open")), 1);
```

To initialize the acceptor object and begin accepting connections, call the open() method. The only required argument is the first, the address to listen on. The backlog and reuse_addr parameters are the same as for ACE_SOCK_Acceptor, and the default proactor argument selects the process's singleton instance. The non-zero validate_new_connection argument indicates that the framework should call the handler's validate_connection() method when accepting a new connection, as we discussed above.

The bytes_to_read argument can specify a number of bytes to read immediately on connection acceptance. This is not unversally supported by underlying protocol implementations, and is very seldom used. If used, though, it would be what causes data to be available in the message block passed to ACE_Service_Handler::open(), as we saw in our example on page 184.

The pass_addresses argument is of some importance if your handler requires the local and peer addresses when running the service. The only portable way to obtain the local and peer addresses for asynchronously-established connections is to implement the ACE_Service_Handler::addresses() hook method, and pass a non-zero value as the pass_addresses argument to ACE_Asynch_Acceptor::open().

Actively establishing connections is very similar to passively accepting them. The hook methods are similar. The following could be used to actively establish a connection and instantiate a HA_Proactive_Service object to service the new connection:

```
ACE_INET_Addr peer_addr;    // Set up peer addr
ACE_Asynch_Connector<HA_Proactive_Service> aio_connect;
aio_connect.connect (peer_addr);
```

8.4 The ACE_Proactor Completion Demultiplexer

The `ACE_Proactor` class drives completion handling in the ACE Proactor framework. It waits for completion events that indicate one or more operations started by the I/O factory classes have completed, demultiplexes those events to the associated completion handler, and dispatches the appropriate hook method on the 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 ();
```

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.

8.6 Other I/O Factory Classes

The I/O factory classes listed above are used for asynchronous I/O on many different types of IPC endpoints. An I/O handle from any ACE IPC class, such as `ACE_SOCK_Stream` or `ACE_FILE_IO`, may be used with the I/O factory classes .

- `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 Integrating Proactor and Reactor Events (Windows)

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 using the `ACE_WFMO_Reactor`, 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 . The `ACE_WIN32_Proactor` class uses a I/O completion port internally to manage its event dispatching. The I/O completion port's handle is signaled when one or more I/O operations complete, and this handle is the link between the reactor and the proactor. To make use of this link, instantiate an `ACE_WIN32_Proactor` object with 2nd arg 1 (this allows the `ACE_WIN32_Proactor` object's completion port handle to be made available via its `get_handle()` method) and an `ACE_Proactor` object with the `ACE_WIN32_Proactor` as its implementation. Register the `ACE_WIN32_Proactor`'s handle with the desired `ACE_Reactor` object. The reactor's event loop will then react to the completion port's handle being signaled, and call back to the registered proactor object which will then dispatch handlers for all completed asynchronous I/O operations. The following code shows the steps for creating a `ACE_Proactor` as described, making it the singleton, and registering it with the singleton reactor.

```
ACE_WIN32_Proactor proactor_impl (0, 1);
ACE_Proactor proactor (&proactor_impl);
ACE_Proactor::instance (&proactor);
ACE_Reactor::instance ()->register_handler (&proactor_impl,
proactor_impl.get_handle ());
```

Be cautious with this arrangement, because there is no way to unregister the proactor handle from the reactor. This means that the reactor instance that has the proactor handle registered must be shut down before the proactor is destroyed.

Chapter 9
Other IPC Types

9.1 The Other IPC Wrappers in ACE

So far, all the IPC usage and examples in this book 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. Like the TCP/IP Sockets wrappers, most of the others offer an interface compatible with using them in the ACE Acceptor-Connector framework (`ACE_Acceptor`, `ACE_Connector`, and `ACE_Svc_Handler` classes).

9.2 Interhost IPC Wrappers

The classes we describe in this section can be used for both interhost and intrahost communication. The IPC mechanisms these classes wrap are designed for interhost communication. However, intrahost communications is a very simplified host-to-host communication situation, and these mechanisms all work perfectly fine for communication between collocated entities.

9.2.1 UDP/IP

UDP is a datagram-oriented protocol that operates over IP. Therefore, like TCP/IP, it uses IP addressing. Also like TCP, datagrams are demultiplexed within each IP address using a port number. UDP port numbers have the same range as TCP port numbers, but they 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`.

There are three primary differences you should consider when deciding whether or not to use UDP communication:

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; however, it 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, when using UDP, there is more of a record-oriented nature to the transmissions, 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 above, 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 require.
3. Whereas TCP is a one-to-one connection between two peers, UDP offers three different modes of operation:
   - Unicast, which is one-to-one operation, similar to TCP.
   - Broadcast, in which 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 broadcast network traffic must be processed by each station on the attached network, it can cause network traffic problems and is generally frowned upon.

- Multicast, which 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 of the attendant traffic issues.

We'll look at brief examples using UDP in the three addressing modes. Note that all of the UDP classes we'll look at can be used with the ACE Reactor framework and the I/O UDP classes can be used as the peer stream template argument with the `ACE_Svc_Handler` class template.

Unicast

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)
{
  const char *message = "this is the message!\n";
  ACE_INET_Addr my_addr (ACE_static_cast (u_short, 10101));
  ACE_SOCK_Dgram udp (my_addr);
  ssize_t sent = udp.send (message,
                           ACE_OS_String::strlen (message) + 1,
                           to);
  udp.close ();
  if (sent == -1)
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("send")), -1);
  return 0;
}
```

You'll notice two differences from our earlier examples using TCP.

1. You just open and use `ACE_SOCK_Dgram`. There is no need for an acceptor or connector.
2. You need to explicitly specify the peer's address when sending a datagram.

These are a result of UDP's datagram nature. There is no formally established connection between any two peers—you obtain the peer's address and send

directly to that address. Although our simple example above sends one datagram and closes the socket, the same socket could be used to send and receive many datagrams, from any mixture of different addresses. Thus, even though UDP unicast mode is one-to-one, there is no one fixed peer. It simply means that each datagram is directed from the sending peer to one other.

If the application you are writing specifies a sending port number (e.g., your application is designed to receive datagrams at a known port) you must set that information in an `ACE_INET_Addr` object that specifies your local addressing information. If there is no fixed port, you can pass `ACE_Addr::sap_any` as the address argument to `ACE_SOCK_Dgram::open()`.

There are two ways to obtain the destination address to send a datagram to. First, you can use a well-known or configured IP address and port number, similar to the way you obtain the peer address when actively connecting a TCP socket using `ACE_SOCK_Connector`. This is often the way a client application addresses a known service. The second method, however, is often used in UDP server applications. Since each `ACE_SOCK_Dgram` object can send and receive datagrams from any number of peers, there isn't one fixed address to send to. In fact, the destination can vary with every sent datagram. To accomodate this use case, the sender's address is available with every received datagram. The following example shows how to obtain a datagram sender's address and echo the received data back to the sender.

```
void echo_dgram (void)
{
  ACE_INET_Addr my_addr (ACE_static_cast (u_short, 10102));
  ACE_INET_Addr your_addr;
  ACE_SOCK_Dgram udp (my_addr);
  char buff[BUFSIZ];
  size_t buflen;
  if (0 < udp.recv (buff, buflen, your_addr))
    udp.send (buff, buflen, your_addr);
  udp.close ();
  return;
}
```

The third argument in our usage of ACE_SOCK_Dgram::recv() receives the address of the datagram sender. We use the address to correctly echo the data back to the original sender. Again, for simplicity, the example uses and closes the UDP socket. This is also a reminder that `ACE_SOCK_Dgram` objects do not close the underlying UDP socket when the object is destroyed. The socket must be explic-

itly closed before destroying the ACE_SOCK_Dgram object or a handle leak will result.

You may ask, though, isn't this a lot of trouble for cases where an application uses UDP but always exchanges data with a single peer? Yes, it is. For cases where all communication takes place with a single peer, ACE offers the ACE_SOCK_CODgram class (connection-oriented datagram). There is no formal connection established at the UDP level; however, the addressing information is set when the object is opened so it need not be specified on every data transfer operation. There is still no need for a acceptor or connector class as with UDP. The following example briefly shows how to open an ACE_SOCK_CODgram object.

```
#include "ace/SOCK_CODgram.h"
// ...
  const ACE_TCHAR *peer = ACE_TEXT ("other_host:8042");
  ACE_INET_Addr peer_addr (peer);
  ACE_SOCK_CODgram udp;
  if (0 != udp.open (peer_addr))
    ACE_ERROR ((LM_ERROR, ACE_TEXT ("%p\n"), peer));

  // ...

  if (-1 == udp.send (buff, buflen))
    ACE_ERROR ((LM_ERROR, ACE_TEXT ("%p\n"), ACE_TEXT ("send")));
```

The example specifies UDP port 8042 at host other_host as the peer to always send data to. If the open() succeeds, the send() operations need not specify an address. All sent data will be directed to the prespecified address.

Broadcast

In broadcast mode, the destination address must still be specified for each send operation. However, the UDP port number part is all that changes between sends because the IP address part is always the IP broadcast address which is a fixed value. The ACE_SOCK_Dgram_Bcast class takes care of supplying the correct IP broadcast address for you; you need only specify the UDP port number to broadcast to. The following is an example.

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

```
int send_broadcast (u_short to_port)
{
  const char *message = "this is the message!\n";
  ACE_INET_Addr my_addr (ACE_static_cast (u_short, 10101));
  ACE_SOCK_Dgram_Bcast udp (my_addr);
  ssize_t sent = udp.send (message,
                           ACE_OS_String::strlen (message) + 1,
                           to_port);
  udp.close ();
  if (sent == -1)
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("send")), -1);
  return 0;
}
```

The `ACE_SOCK_Dgram_Bcast` class is a subclass of `ACE_SOCK_Dgram`, so all datagram receive operations are similar to those in the unicast examples.

Multicast

Multicast is a UDP mode that involves a group of network nodes called a multicast group. The underlying OS-supplied protocol software manages multicast groups using specialized protocols. The OS directs the group operations based on applications' requests to join (subscribe) or leave (unsubscribe) a particular multicast group. Once an application has joined a group, all datagrams sent on the joined socket are sent to the multicast group without specifying the destination address for each send operation.

Each multicast group has a separate IP address. Multicast addresses are IP class D addresses, which are separate from the class A, B, and C addresses that individual host interfaces are assigned. Applications define and assign class D addresses specific to the application.

The following shows an example of how to join a multicast group and transmit a datagram to the group using the `ACE_SOCK_Dgram_Mcast` class:

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

int send_multicast (const ACE_INET_Addr &mcast_addr)
{
  const char *message = "this is the message!\n";
  ACE_SOCK_Dgram_Mcast udp;
```

```
  if (-1 == udp.join (mcast_addr))
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("join")), -1);

  ssize_t sent = udp.send (message,
                           ACE_OS_String::strlen (message) + 1);
  udp.close ();
  if (sent == -1)
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("send")), -1);
  return 0;
}
```

As with `ACE_SOCK_Dgram_Mcast`, `ACE_SOCK_Dgram_Mcast` is a subclass of `ACE_SOCK_Dgram`; therefore, recv() methods are inherited from `ACE_SOCK_Dgram`.

9.3 Intrahost Communication

The classes we describe in the section can be used for intrahost communication only. They can offer some simplicity over interhost communications due to simplified addressing procedures. Intrahost IPC can also be significantly faster than interhost IPC due 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 weight the options, maybe do your own performance benchmarks, and decide on 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.

Part II

Process and Thread Management

Chapter 10
Process Management

Processes are the primary abstraction used by an operating system to represent running programs. Unfortunately the meaning of the term "process" varies widely between operating systems. 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 do not have processes at all, but instead have one large address space in which *Tasks* run (such as VxWorks). The ACE process classes are not included for these operating systems.

In this chapter we will cover the following topics:

- How to use the simple `ACE_Process` wrapper class to create a process and then manage child process termination.
- 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 will take a look at the high level process manager class that offers integration with the Reactor we have learned to love so much.

10.1 Spawning a new Process

ACE hides all process creation and control API's 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 are allowed to set several options for the child process including

- 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 you are better off using `ACE_OS::fork()` to accomplish this if you need it.
Spawning a process using the `ACE_Process` class is a two step process

- Create a new `ACE_Process_Options` object specifying the desired properties of the new child process.
- 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, that 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 also exits.

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

```
int main(int argc, char *argv[])
{
  ACE_UNUSED_ARG(argv);

  if(argc > 1)//slave mode
  {
      Slave s;
      s.doWork();
  }
  else       //master mode
  {
      Manager m(argv[0]);
      m.doWork();
```

```
  }

  return 0;
}
```

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

```
class Manager
{
public:

  Manager(const char* program_name)
  {
    ACE_TRACE("Manager::Manager");

    ACE_OS::strcpy(programName_, program_name);
  }

  int doWork()
  {
    ACE_TRACE("Manager::doWork");

    this->setupOptions();
    //setup options

    ACE_Process process;
    pid_t pid
          =process.spawn(this->options_);
    if(pid== -1)
       ACE_ERROR_RETURN((LM_ERROR,
                "%p\n", "spawn"),-1);
    //spawn the new process

    if(process.wait()==-1)
      ACE_ERROR_RETURN((LM_ERROR,
         "%p\n", "wait"), -1);
    //wait forever for my child to return

    this->dumpRun();
    //dump whatever happend
```

```
  return 0;

}
```

After the options for the process are set up we create an `ACE_Process` object on the stack. The process object is then used to spawn a new process based on the process options passed in. This uses `execvp()` on UNIX and `CreateProcess()` on Windows. Once the child process has been spawned successfully the parent process uses the `wait()` method to wait for the child to finish and exit. The `wait()` method collects the exit status of the child process and avoids zombie processes on UNIX. On Windows it causes the closing of the process and thread HANDLEs that `CreateProcess()` created. Once the slave returns the master prints out the activity performed by the slave and the master to the standard output stream.

Lets take a closer look at how we set up the options for the child process.

```
void setupOptions()
{
  ACE_TRACE("Manager::setupOptions");

  this->options_.command_line("%s 1", this->programName_);
  this->setStdHandles();
  this->setUserID();
  this->setEnvVariable();

}
void setStdHandles()
{
  ACE_TRACE("Manager::setStdHandles");

  ACE_OS::unlink("output.dat");

  this->outputfd_
    = ACE_OS::open("output.dat", O_RDWR | O_CREAT);

  int result =
          this->options_.set_handles(ACE_STDIN,
                                     ACE_STDOUT,
                                     this->outputfd_);
  ACE_ASSERT(result == 0);
  //setup the input, output and error handles
}
```

```
  void setEnvVariable()
  {
    ACE_TRACE("Manager::setEnvVariables");

    this->options_.setenv("PRIVATE_VARIABLE=/that/seems/to/be/
it");
    //setup an environment variable
  }
```

First we set the command line to be the same program name as the current program (since the child and parent processes are both represented by the same program) with a single argument. The extra argument indicates that the program is to run in slave mode. After this we set the standard input, output and error handles for the child process. The input and output handles will be shared between the two processes whereas the stderr handle for the child is set to point to a newly created file, `output.dat`. We also show how to set up an environment variable in the parent process that the child process should be able to see and use.

For UNIX runs we also set the effective user ID we want the child process to start off running as. We discuss this in a little detail later.

To reiterate, the Manager goes through the following steps;

- Set up process options including the command line, environment variables and I/O streams.
- Spawn the child process.
- Wait for the child to exit.
- Display the results of the child process run.

Now lets take a quick look at the how the Slave runs. The Slave class has a single method that exercises the validity of the input, output, and error handles, the environment variable that was created and displays its own and its parent's process ID.

```
class Slave
{
public:
  Slave()
  {
        ACE_TRACE("Slave::Slave");
  }
```

```
int doWork()
{
  ACE_TRACE("Slave::doWork");

  ACE_DEBUG((LM_INFO,
             "(%P) started at %T, it's parent is %d\n",
             ACE_OS::getppid()));

  this->showWho();
  //check who this is running as

  ACE_DEBUG((LM_INFO,
      "(%P) the private environment is %s\n",
      ACE_OS::getenv("PRIVATE_VARIABLE")));
  //check if the enviroment got setup as we wanted

  char str[128];
  ACE_OS::sprintf(str,
      "(%d) Enter your command\n", ACE_OS::getpid());
  ACE_OS::write(ACE_STDOUT,
      str, ACE_OS::strlen(str));

  this->readLine(str);
  ACE_DEBUG((LM_DEBUG,
      "(%P) Executed your command: %s",
     str));
  //check out the I/O handles

  return 0;
}
```

After the slave process is created the `doWork()` method is called. This method does the following:

1. Checks what the effective user ID of the program is
2. Check and print the private environment variable PRIVATE_VARIABLE
3. Asks the user for a string command
4. Reads the string command from standard input
5. Prints the string back out again to the standard error stream (remember all `ACE_DEBUG()` messages are set to go to the standard error stream by default).

After the master determines that the slave has completed and exited it displays the output that the slave generated in the debug log. Since the stderr stream was set to the file `output.dat` all the Manager needs to do is dump this file. Notice that when we set the set the stderr handle for the child we kept a reference to it in the Master that we did not close. We can use this open handle to do our dump. Since the file handle was shared between the slave and the master we first seek back to the beginning of the file and then move forward.

```
int dumpRun()
{
  ACE_TRACE("Manager::dumpRun");

  if(ACE_OS::lseek(this->outputfd_, 0, SEEK_SET)==-1)
    ACE_ERROR_RETURN((LM_ERROR, "%p\n", "lseek"),-1);

  char buf[1024];
  int length = 0;
  while( (length = ACE_OS::read(this->outputfd_,
              buf, sizeof(buf) -1 )) > 0)
  {
    buf[length] = 0;
    ACE_DEBUG((LM_DEBUG, "%s\n", buf));
  }
  //read the contents of the error stream writen
  //by the child and print it out.

  ACE_OS::close(this->outputfd_);

  return 0;
}
```

10.1.1 Security Parameters

As we mentioned earlier, `ACE_Process` allows you to specify the effective, real and group IDs that you want the child process to run with. Continuing with the previous example, the following code illustrates setting the effective user ID of the child process to be the user ID of the user nobody. Of course for this to run on your system you must make sure that there is a nobody account and that the user running the program has permission to perform the effective user id switch.

```
  int setUserID()
  {
    ACE_TRACE("Manager::setUserID");
#if !defined(WIN32)
    passwd* pw =
       ACE_OS::getpwnam("nobody");
    //get the real user id of nobody

    this->options_.seteuid(pw->pw_uid);
    //ask the child to run with this euid.
    //this works if the "correct" UNIX permissions
    //are setup before the run
#endif
    return 0;
  }
```

Note that this code only works for those systems that have a UNIX-like notion of these IDs. If you are using a Windows system you can specify the `SECURITY_ATTRIBUTES` for the new process and its primary thread; however, you cannot use `ACE_Process` to spawn a process for client impersonation.

10.1.2 Using ACE_Process Hook Methods

ACE_Process provides several callback hook methods that allow you to customize the spawn() process further. The first hook prepare(ACE_Process_Options&) is called back right after you call spawn(). If this method returns a value less then zero then the spawn() is aborted. You can use this method to modify the options or abort process creation.

The second hook method parent(pid_t child) is called back immediately after the fork() call on UNIX platforms (or the CreateProcess()) call on Win32). In addition to the parent hook method, there is also a child(pid_t parent) hook that is available on UNIX platforms. This occurs before the subsquent exec() call that occurs if you do not specify ACE_Process_Options::NO_EXEC in the creation flags for the process options (the default case). At this point the new enviroment, including handles and working directory are not set. This method is not called back on Win32 platforms as there is no concept of a fork() and then exec() here.

10.2 Synchronization

To synchronize threads with each other you need synchronization primitives such as mutexes or semaphores. We discuss these primitives in significant detail in Section 12.2. However, when the threads are executing in separate processes then they are actually running in different address spaces. Synchronization between such threads becomes a little harder. In cases like these you have two options:

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

This section goes over how you can use the latter scheme to ensure synchronization between threads running in different processes.

10.2.1 Mutexes

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

In the next example we create a named mutex called “GlobalMutex”. We then proceed to create two processes that cooperatively share an imaginary global resource, coordinating their access using the mutex. They 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 just spawns two child processes. If started up as a child process it 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.

Once again note that even though we create two 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 main(int argc, char *argv[])
{
  if(argc > 1) //run as the child
  {
     ACE_Process_Mutex mutex("GlobalMutex");
     //create or get the global mutex

     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,
        "Spawned processes with pids %d:%d \n",
                 pida, pidb));

      if(processa.wait()==-1)
        ACE_ERROR_RETURN((LM_ERROR, "%p\n",
                          "Error in process wait"), -1);

      if(processb.wait()==-1)
        ACE_ERROR_RETURN((LM_ERROR, "%p\n",
                          "Error in process wait"), -1);
  }

  return 0;
}
```

The GResourceUser class represents a user of an unspecified global resource that is protected by a gmutex. When the ResourceAcc**::**run() method of this object is called it intermittently acquires the global mutex, work with the global resource

and then releases the mutex. Since it release the resource between runs this gives the second process a chance to acquire it between runs.

```
class GResourceUser
{
public:
  GResourceUser(ACE_Process_Mutex &mutex)
        :gmutex_(mutex)
  {
    ACE_TRACE("GResourceUser::GResourceUser");
  }

  void run()
  {
    ACE_TRACE("GResourceUser::run");

    int count = 0;
    while(count++ < 10)
    {
     int result
        = this->gmutex_.acquire();
      ACE_ASSERT(result == 0);


      ACE_DEBUG((LM_DEBUG,
        "(%P| %t) has the mutex\n"));

      //
      //Access Global resource
      //

      ACE_OS::sleep(1);

      result = this->gmutex_.release();
      ACE_ASSERT(result == 0);
    }
  }

private:
   ACE_Process_Mutex &gmutex_;
};
```

The results from running the program shows the two process competing with each other to acquire the shared resource.

```
Spawned processes with pids 1808:1004
(1808| 1188) has the mutex
(1004| 1816) has the mutex
(1808| 1188) has the mutex
(1004| 1816) has the mutex
(1808| 1188) has the mutex
(1004| 1816) has the mutex
```

For UNIX users the ACE_Process_Mutex maps to a System V shared semaphore. Unlike most resources, these semaphores are not automatically released once all references to it are destroyed. Therefore be careful when using this class and make sure that the destructor of the `ACE_Process_Mutex` is called (even in the case of abnormal exits).

10.3 Using the ACE_Process_Manager

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

10.3.1 Spawning and Terminating Processes

The `spawn()` methods for `ACE_Process_Manager` are similar to that 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* of 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 main(int argc, char*argv[])
{
  if(argc > 1) //child
  {
      ACE_OS::sleep(10);
  }
  else //parent
  {
      ACE_Process_Manager* pm
           = ACE_Process_Manager::instance();
      //get the process wide process manager with space for

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

      pid_t pids[NCHILDREN];
      pm->spawn_n(NCHILDREN, options, pids);
      //spawn three child processes

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

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

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

      pm->wait(0);
      //wait for the all (only one left) of the
      //children to exit
```

```
    }

    return 0;
}
```

The `ACE_Process_Manager` is used to spawn `NCHILDREN` child processes (note that once again we spawn the same example). Once the child processes start they immediately fall asleep. The parent process then explicitly terminates the first child using the `terminate()` method. This should cause the child process to abort immediately. The only argument to this call is the process ID of the process that you wish to terminate. If you pass in the process ID 0 then the process manager waits for any of it's managed processes to exit. On UNIX platforms that does not work as well as one would hope and you may end up collecting the status for a process that is not managed by your process manager, for more on this see the ACE documentation. Immediately after issuing the termination call on the child process, the parent uses the process manger to do a blocking `wait()` on the exit of that child. Once the child exits, the `wait()` call returns with the termination code of the child process.

Note that on UNIX systems, `ACE_Process_Manager` issues a signal to terminate the child once you invoke the terminate() method. You can observe this by examining the termination status of the process.

10.3.2 Event Handling

After completing the wait on the first process the parent process waits for all the rest of its children by using another blocking `wait()` call on the process manager. To indicate this we pass a 0 timeout value to the `wait()` method. Note that you can also specify a timeout value after which the blocking wait will return. If the wait is unsuccessful and a timeout does occur the method returns 0.

In the previous example, we showed you how a parent can block waiting for all of its children to complete. Most of the time you will find that your main process has other work to do besides waiting for terminating children, especially if you have implemented a traditional network server that spawns child processes to handle network requests. In this case you will want to keep the main process free to handle further requests besides reaping your child processes.

To handle this use case the process manager's exit handling routines have been designed to work in conjunction with the ACE Reactor framework. This next

example illustrates how you can set up a termination callback handler that is called back whenever a process is terminated.

```
class DeathHandler: public ACE_Event_Handler
{
public:
  DeathHandler(): count_(0)
  {
    ACE_TRACE("DeathHandler::DeathHandler");
  }

  virtual int handle_exit(ACE_Process * process)
  {
    ACE_TRACE("DeathHandler::handle_exit");

    ACE_DEBUG((LM_DEBUG,
        "Process %d exited with exit code %d\n",
        process->getpid(), process->return_value()));

    if(++count_ == NCHILDREN)
        ACE_Reactor::instance()->end_reactor_event_loop();



    return 0;
  }
private:
  int count_;
};
```

In the program above we create a subclass of `ACE_Event_Handler`, DeathHandler that is used to handle process termination events for all the NCHILDREN processes that are spawned by the process manager. When a process exits, the reactor synchronously invokes the `handle_exit()` method on the event handler passing in a pointer to the `ACE_Process` object representing the process that has just exited. Under the hood on UNIX platforms, the process manger is registered to receive the SIGCHLD signal. On receipt of the signal it asks the reactor to synchronously notify it of the process termination. On Windows platforms the reactor receives an event notification using the process handle with `WaitForMultipleObjectsEx()` and in turn calls the process manager that invokes the DeathHandler::handle_exit() method.

```
int main(int argc, char*argv[])
{
  if(argc > 1)//child
    return 0;
  //exit immediately
  else //parent
  {
    ACE_Process_Manager
    pm(10, ACE_Reactor::instance());
    //instantiate a process manager with space for
    //10 processes.

    DeathHandler handler;
    //create a process termination handler

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

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

    for(int i=0; i<NCHILDREN; i++)
      pm.register_handler(&handler, pids[i]);
    //
register a handler to be called back when these processes exit

    ACE_Reactor::run_event_loop();
    //run the reactive event loop waiting for events to occur
  }

  return 0;
}
```

10.4 Conclusion

In this chapter we introduced the ACE classes that support process creation, life cycle management and synchronization. This included a look 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 asynchronous events to the application such as; a user typing the interrupt key on a terminal, a broken pipe between processes, job control functions etc. To handle signals a program can associate a signal handler with a particular signal type. For example you can setup a handler for the interrupt key signal that instead of just terminating the process, first asks the user whether he wishes to terminate or not. 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 occurs the associated signal handler is invoked asynchronously. After the signal handler returns execution continues from wherever it happened to be right before the signal was received, as if the interruption never occurred.

Win32 provides minimal support for signals for ANSI compatibility. A minimal set of signals is available and even less are actually raised by the operating system. Therefore, generally for Win32 programmers the usefulness of signals is somewhat limited.

In Chapter 7 we talked about how you can use the ACE_Reactor to handle signal events along with several other event types. Here we specifically talk about how to use the signal handling features of ACE independent of the ACE_Reactor. This comes in handy when either you do not want to add the extra complexity of the reactor, since it isn't needed or you are developing a resource constrained system where you cannot afford to wase any memory.

In this chapter we will go through how ACE makes it easy to setup one or more handlers for a signal type. We will start off by looking at the simple ACE_Sig_Action wrapper that calls a user specified handler function when a signal occurs. We will then look at the higher level ACE_Sig_Handler class that will invoke an ACE_Event_Handler::handle_signal when the specified signal occurs. Finally we will talk about the ACE_Sig_Guard guard class, that allows you to guard certain sections of your code from signal interruptions.

11.1 Using Wrappers

The sigaction() call provides a mechanism by which a programmer can associate an action (i.e., the execution of a call back function) with the occurrence of a particular signal. ACE provides a wrapper around the sigaction() call. This provides a type safe interface for signal registration. In addition to the sigaction wrapper, ACE also provides a typesafe wrapper to one of it's argument type's, sigset_t. This type represents a collection of signals and is discussed in detail in Section 7.2.3 on page 141.

The following example illustrates the use of wrappers to register call back functions for a few signals.

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

static void my_sighandler(int signo);
static int  register_actions();
//forward decl.

int main(int argc, char *argv[])
{
        ACE_TRACE("::main");
        ::register_actions();
        //register certain actions to happen

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

        ACE_OS::kill(ACE_OS::getpid(), SIGUSR1);
        //this will pend until the first signal is completely
        //handled and returns, because we masked it out
        //in the registerAction call.
```

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

The first thing we do when we enter the program is register actions using the ACE provided ACE_Sig_Action class. Then we explicitly cause certain signals to be generated 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 these signal types are meant to be user definable).

```
static int register_actions()
{
        ACE_TRACE("::register_actions");

        ACE_Sig_Action sa(my_sighandler);

        ACE_Sig_Set ss;
        ss.sig_add(SIGUSR1);
        sa.mask(ss);
        //make sure you specify that SIGUSR1 will
        //be masked out during the signal callback's
        //execution.

        sa.register_action(SIGUSR1);
        sa.register_action(SIGUSR2);
        //register the same handler function for these
        //two signals.
}
```

This is our signal handler call back function. Take note of the signature of the method. It returns a void and is passed an integer argument representing the signal that occurred. Under the current model for signals only one signal handler function can be associated for any particular signal number. This means you can’t have multiple functions be called back when a single signal event occurs. You can of course associate the same signal handler function for multiple signal events (as we have done in this example).

In this particular handler once a signal is received we sleep for 10 seconds before returning to regular program execution. Never do something like this in actual programs. In general signal handler functions should be very light and should return immediately, otherwise you will block execution of the rest of your program. In general signal handler functions are used to indicate to the application the occurrence of a signal. The work that must be done due to the signal occurs after the signal handler has returned.

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

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

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

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

Here we create an ACE_Sig_Action object on the stack passing in the address of a signal handler function that will be called back. We don't actually register for the callback in the constructor instead we defer that and do a series of register_action() calls, registering our call back function for SIGUSR1 and SIGUSR2. We also keep the old sigaction information for SIGUSR2 in old_sa (although we don't do anything with it).

Before registration we also create an ACE_Sig_Set container and add SIGUSR1 to it. We then add this set of signals as a mask to the our sigaction object. When the sigaction registers it's action it passes in this mask to the OS, informing it to automatically disable these signals during the execution of the signal handler function. This is why even though we explicitly raise SIGUSR1 during the execution of the signal handler we don't actually see it being executed until after the handler returns.

Be careful that you specify the mask before you register your action with the OS using the register_action() call, otherwise the OS will be unaware of this specification.

Notice that you do not need to keep the ACE_Sig_Action object around after you register the action you want for a particular signal. If you wish to change the

disposition of a certain signal you setup previously you can create another ACE_Sig_Action object and do this.

11.1.1 Interrupted System Calls

You probably noticed the weird loop that we executed around the ACE_OS::sleep() in the main function of the previous example. The reason we needed to do this is because certain blocking system calls, such as read(), write(), ioctl() etc. become unblocked on the occurrence of a signal. Once these calls become unblocked they return with errno set to EINTR (interrupted by a signal). This can be painful to deal with in programs that make use of signals. Therefore most implementations that provide the sigaction() call allow you to specify a flag SA_RESTART to cause signal-interrupted calls to automatically continue transparent to the programmer. Be careful though, every version of UNIX and UNIX like OS's deals with which calls are and are not restarted in a different manner, check your OS documentation for details on this.

In general however most SVR4 UNIX platforms provide sigaction() and the SA_RESTART flag. Using ACE you can call the ACE_Sig_Action::flags() method to set SA_RESTART or you can pass it in as an argument to the constructor of a sigaction.

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. This class allows a client programmer to register ACE_Event_Handler based objects for callback on the occurrence of signals. The ACE_Event_Handler type is used extensively in ACE (see Section 7.1 on page 134).

The ACE_Event_Handler can be treated as an abstract base class (in reality it's methods are virtual not pure virtual and are defined with empty bodies) that client programmers must sub-class and implement. It has different handle_eventXXX() type methods that are called back depending on the event type that occurs. For example the relevant method for signals is appropriately named handle_signal().

As client programmers we must sub-class 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("MySignalHandler::handle_signal");

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

    ACE_DEBUG((LM_DEBUG, "%d occured\n", signum));
  }
private:
  int signum_;
};
```

We start off by creating a signal handler class that publicly derives from the ACE_Event_Handler base class. We only need to implement the handle_signal() method here since our event handler is designed only to be able to handle “signal” based events. The siginfo_t and uncontext_t are also passed back to the handle_signal() method in addition to the signal number when the signal event occurs. This in accordance with the new sigaction() signature, for more information on these types and the information that they represent see Section 7.2.4 on page 143 and Section 7.2.5 on page 147.

```
int main()
{
  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)
        continue;
      else
        {
          ACE_ERROR_RETURN((LM_ERROR,
                            "%p\n", "sleep"),-1);
        }
    }
}
```

fter the program starts up we create two signal event handler objects on the stack and register them to be called back for different signals. Registration occurs with the help of the ACE_Sig_Handler class. This class allows you to pass in a signal number and a pointer to an ACE_Event_Handler object that you want to be called back when the signal occurs. You can also use the ACE_Sig_Handler::register_handler() method to pass in a new ACE_Sig_Action that contains the mask and flags you want setup for the action that is associated with the signal you are currently registering for. This method can also be used to return the old event handler and old ACE_Sig_Action struct.

After we register our event handlers to be called back we raise two signals that are caught in their respective event handlers and after sleeping for a while exit the program

11.2.1 Stacking Signal Handlers

In certain cases you may want to register more then one signal handler function for a particular signal event. In previous sections we mentioned that the UNIX model of signals only allows a single handler to be called back when a signal occurs. You can use the ACE_Sig_Handlers class to modify this behavior. This class provides for the stacking of signal handler objects for a particular signal event. By making a simple modification to the previous example main we get;

```
int main()
{
  MySignalHandler h1(SIGUSR1), h2(SIGUSR1);

  ACE_Sig_Handlers handler;
  handler.register_handler(SIGUSR1, &h1);
  handler.register_handler(SIGUSR1, &h2);

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

```
  int time=10;
  while((time = ACE_OS::sleep(time))==-1)
    {
      if(errno==EINTR)
        continue;
      else
        {
          ACE_ERROR_RETURN((LM_ERROR,
                            "%p\n", "sleep"),-1);
        }
    }
}
```

The only thing that we do differently here is we create an ACE_Sig_Handlers object instead of an ACE_Sig_Handler object. This allows us to register multiple handlers for the same signal number. When the signal occurs all handlers are automatically called back.

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(), just to run our examples). When a signal is raised the thread of control in a single thread application (or one of threads in a multi-threaded application) jumps off to the signal handling routine, executes it and then returns to regular processing. As in all such scenarios there are certain small regions of code that 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. Note that some signals such as SIGSTOP and SIGKILL cannot be masked.

```
class MySignalHandler:
        public ACE_Event_Handler
{
public:
        virtual handle_signal(int signo,
                siginfo_t * s=0, ucontext_t* = 0)
        {
```

```
            ACE_DEBUG((LM_DEBUG,
               "Signal %d\n", signo));
        }
};

int main(int argc, char *argv[])
{

        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,
                    "Entering critical region\n"));
                ACE_OS::sleep(10);
                ACE_DEBUG((LM_DEBUG,
                    "Leaving  critical region\n"));
                //Signal Guarded region of code
        }
        //do other stuff..
}
```

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 only added the signal SIGUSR1. This means that the scoped code is “safe” from SIGUSR1 until the scope closes.

To test this out we ran this program and used the UNIX kill(1) utility to send the process SIGUSR1.

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

Notice that even though we sent the signal SIGUSR1 before the program had left the scope block of the critical region it was received and processed after we had left the critical region. This happens because while in the critical region we were guarded against SIGUSR1, and thus the signal remains pending on the process. Once we leave the critical section scope the guard is released and the pending signal is delivered to our process.

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 handled asynchronously, unlike the other event types that are handled by the Reactor. You can however convert asynchronous signal calls to synchronous notifications, for more on this topic and the Reactor see .

11.5 Conclusion

In this chapter we looked at how signals, which are asynchronous software interrupts, are handled portably with ACE. We learned how to use the low level sigaction and sigset_t wrappers to setup our own callback signal handler functions. We then learned 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

Multi-threaded programming has become more and more of a necessity in today's software. Whereas yesterday most general-purpose operating systems only provided programmers with user level thread libraries today you will find that most provide real pre-emptive multi-tasking. As the prices of multi-processor machines fall their use has also become prevalent. The scale of new software also appears to be growing. Whereas yesterday we had hundreds or thousands of concurrent users, today that number has jumped ten fold. Development of real-time software has always required the use of thread priorities and with the surge of intelligent embedded devices many more developers are getting involved in this field. All of these factors combined make it more and more important for developers to be familiar with threading concepts and their productive use.

This chapter will introduce you to the basics of using threads with the ACE toolkit. ACE provides a lot in the area of threads, therefore we have divided the discussion on threads into two separate chapters. This chapter will cover the basics and will include a discussion on;

- creating new threads of control,
- learning about safety primitives that ensure consistency when you have more then one thread accessing a global structure,
- event and data communication between threads.

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. With ACE all you have to do to create your own thread is create a sub-class of the `ACE_Task_Base` class and overide the implementation of the virtual `svc()` method. The `svc()` method serves as the entry point for your new thread, i.e., 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 this first example we will create a home automation command handler thread, `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()` macros `%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, "(%t) Handler Thread running\n"));

        return 0;
    }
};


int main(int argc, char *argv[])
{
    ACE_DEBUG((LM_DEBUG, "(%t) Main Thread running\n"));
    HA_CommandHandler handler;
    int result =
        handler.activate(THR_NEW_LWP |
                         THR_JOINABLE |
```

```
                        THR_SUSPENDED);
    ACE_ASSERT(result == 0);

    ACE_DEBUG((LM_DEBUG,
        "(%t) The current thread count is %d\n",
        handler.thr_count()));
    ACE_DEBUG((LM_DEBUG,
        "(%t) The group identifier is %d\n",
        handler.grp_id()));

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

To actually start up the thread you must first 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 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 it's threads to complete before continuing and falling out of the main function. Once the child thread runs to completion, i.e., completes the svc() method the wait()call will unblock the main thread which 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? Because, once the main thread returns from the main() function the C run time sees this as an indication that the process is ready to exit and destorys the entire running process, including the handler child thread. If we would allow this to happen the program might exit before the child thread ever got scheduled and got a chance to execute.

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

class HA_CommandHandler: public ACE_Task_Base
{
public:
    virtual int svc(void)
    {
        ACE_DEBUG((LM_DEBUG, "(%t) Handler Thread running\n"));

        return 0;
    }
};
```

The output shows the two threads running i.e., the main thread and the child command handler thread.

```
(496) Main Thread running
(3648) Handler Thread running
```

12.2 Basic Thread Safety

One of the hardest problems you deal with when writing multi-threaded programs is maintaining consistency of all globally available data. Since 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 will 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 *Thread Synchronization and Safety.*

12.2.1 Using Mutexes

Mutexes are the simplest protection primitive available and provides a simple `acquire()`, `release()` interface. If the `acquire()`'ing thread is succesful in getting the mutex it continues forward, otherwise it blocks until the holder of the mutex `release()`'s it.

As shown in the table above, ACE provides several mutex classes. ACE_Mutex can be used as light weight synch. primitive for threads and as a heavy weight cross process synch. primitive,

In the next example we add a device repository to our home automation example, that contains references to all the devices connected to our home network. This repository also contains the interface to applying 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 membr that it uses to ensure the aforementioned consistency constraint. This is a common idiom that you will find yourself using on a regular basis. Whenever a thread calls the `update_device()` method it first has to acquire the mutex

before it can continue forward, since only one thread can actually have the mutex at a time, at no point will two threads simulatenosuly 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()
        :mutex_()
    {
    }

    void update_device(int device_id)
    {
        mutex_.acquire();
            ACE_DEBUG((LM_DEBUG,
                    "(%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 our handler 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.

```
class HA_CommandHandler: public ACE_Task_Base
{
public:
    enum {NUM_USES = 10};

    HA_CommandHandler(HA_Device_Repository& rep)
        :rep_(rep)
    {
    }
```

```
    virtual int svc(void)
    {
        ACE_DEBUG((LM_DEBUG, "(%t) Handler Thread running\n"));

        for(int i=0; i < NUM_USES; i++)
            this->rep_.update_device(i);

        return 0;
    }

private:
    HA_Device_Repository & rep_;

};

int main(int argc, char *argv[])
{
    HA_Device_Repository rep;

    HA_CommandHandler handler1(rep);
    HA_CommandHandler handler2(rep);

    handler1.activate();
    handler2.activate();

    handler1.wait();
    handler2.wait();


    return 0;
}
```

The output from this program shows the two handler threads competing to update devices in the repository. You may notice that on your platform one thread may hang onto the repository until it done before it lets go or that the threads run amok amongst each other with no particular order as to which thread uses the toaster. You can ensure strict ordering (if that is what you need) by using an `ACE_Token`. But be aware that although tokens support strict ordering and are recursive they are slower and heavier then mutexes.

```
(3768) Handler Thread running
(3768) Updating device 0
(1184) Handler Thread running
```

```
(1184) Updating device 0
(3768) Updating device 1
(1184) Updating device 1
(3768) Updating device 2
(1184) Updating device 2
(3768) Updating device 3
(1184) Updating device 3
(3768) Updating device 4
```

12.2.2 Using Guards

In many cases we find that exceptional conditions cause deadlock in otherwise perfectly working code. The reasons for this vary but this usually happens when we overlook an exceptional path and forget to unlock a mutex. Let's illustrate this with a piece of code;

```
int
HA_Device_Repository::update_device(int device_id)
{
    mutex_.acquire();
    ACE_DEBUG((LM_DEBUG,
        "(%t) Updating device %d\n",
        device_id));

    ACE_NEW_RETURN(object, Object, -1);
    //allocate a new object.
    //..
    //use the object

    mutex_.release();
}
```

You can spot the problem here pretty easily, the `ACE_NEW_RETURN()` macro returns when an error occurs thereby preventing the lock from being released and we have a nasty deadlock. In real code, it becomes harder to spot such returns and having multiple `release()` calls all over your code can quickly become very ugly.

ACE provides a convenience class that solves this problem. The `ACE_Guard` set of classes and macros help simplify your code and prevent against the deadlock situations we described above. The guard is based on the familiar C++ idiom of using the constructor and destructor calls for resource acquisition and release.

The Guard classes `acquire()` an underlying lock when they are constructed and release the lock when they are destroyed. Using a guard on your stack you are always assured that the lock will be released no matter what pathological path your code may wind through.

Using a guard instead of explicit calls to acquire and release the mutex the previous snippet of code becomes

```
int
HA_Device_Repository::update_device(int device_id)
{
   ACE_Guard<ACE_Thread_Mutex> guard(this->mutex_);
   //construct a guard specifying the type of the mutex as
   //a template parameter and passing in the thread mutex
   //object as a paramter

   ACE_NEW_RETURN(object, Object, -1);
   //this can throw an exception that is not caught here
   //..
   //use the object
   //..
   //guard is destroyed, automatically releasing the lock.
}
```

All we do here is create an `ACE_Guard<>` on the stack. This automatically acquires and releases the mutex on function entry and exit, just what we want. The `ACE_Guard<>` template class takes the lock type as it's template parameter. It also requires you to pass in a lock object that the guard will operate on.

ACE provides a rich variety of guards;

Table 12.1. ACE Guard Categories

Table 1:

Guard	Description
`ACE_Guard<T>`	Uses the `acquire()` and `release()` methods of the supplied lock during guard creation and destruction. Thus if you use an `ACE_Thread_Mutex` for T you get the semantics of the `acquire()` and `release()` for this mutex type.
`ACE_Read_Guard<T>`	Ensures that `acquire_read()` is used for acquisition instead of the regular `acquire().`
`ACE_Write_Guard<T>`	Ensures that `acquire_write()` is used for acquisition instead of the regular `acquire().`
`ACE_TSS_Guard<T>`	Allocates the Guard on the heap and keeps a reference to it in thread specific storage. This ensures that the lock is always released even if the thread exits explicitly using `ACE_Thead::exit().`
`ACE_TSS_Read_Guard<T>`	Read version of a Thread Specific Guard.
`ACE_TSS_Write_Guard<T>`	Writer version of a Thread Specific Guard.

Most of these guards are self explanatory, the others we will cover in the advanced section of this book.

Guard Macros

ACE also provides a set of convient macros, much like the `ACE_NEW_RETURN`, macros that you can use to allocate guards on the stack. These macros perform error checking on the underlying `acquire()` and `release()` calls and return a supplied error value if anything fails.

Guards macros that do not return values;

- **ACE_GUARD** (LockType, StackObjectName, LockObject)

- **ACE_WRITE_GUARD** (LockType, StackObjectName, LockObject)
- **ACE_READ_GUARD** (LockType, StackObjectName, LockObject)

  Guard macros that do have return values;
- **ACE_GUARD _RETURN** (LockType, StackObjectName, LockObject, ReturnValue)
- **ACE_WRITE_GUARD_RETURN** (LockType, StackObjectName, LockObject, ReturnValue)
- **ACE_READ_GUARD_RETURN** (LockType, StackObjectName, LockObject, ReturnValue)

```
int
HA_Device_Repository::update_device(int device_id)
{
   ACE_GUARD_RETURN(ACE_Thread_Mutex, mon, mutex_, -1);

   ACE_NEW_RETURN(object, Object, -1);
   //use the object
   //...
}
```

Here the guard return macro is used with the device repository, if there is an error the macro exits the function return –1, otherwise it creates an `ACE_Guard<ACE_Thread_Mutex>` instance called mon on the stack.. In cases where the mutex protected method does not return a value the not return value guards should be used in conjunction with the `errno` facility.

12.3 Intertask Communication

Writing multithreaded programs you will often feel the need for your tasks to communicate with each other. This communication may take the form of something as simple as one thread informing the other that it is time to exit or something more complicated where the communicating threads are actually passing data back and forth.

In general intertask communication can be divided into two broad categories

- State change or event notifications, where only the event occurrence needs to be communicated but no data is passed between the two threads.

- Message passing, where data is passed between the two threads, possibly forming a work chain where the first threads processes the data then passes it along to the next for further processing.

12.3.1 Using Condition Variables

Condition Variables can be used to communicate a state change, an event arrival or the satisfaction of some condition to other threads and are often used for as a mechanism to achieve the afore mentioned state change communications. 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 will show you an example of this when we talk about Timers later in this book.

We can easily change our command handler example to use a condition variable to co-ordinate access to the device repository, instead of using a mutex for protection. We modify the repository so that it is able to record which task currently owns it.

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

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

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

    ACE_Task_Base * get_owner()
    {
        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 co-ordinate access to the repository, both of which are passed to the command handlers during construction (both are created on the main() thread stack.).

To use a condition variable you must first acquire the mutex, check whether the system state is in the required condition, if so perform the required action and then release the mutex. If the condition is not in the required state you must call `wait()` on the condition variable waiting for the system state to change. Once the system state changes, the state chaning thread signal()'s the condition variable waking up the waiting threads. The waiting threads wake up, check the system state again and if the state is still amenable performs the required action, otherwise falling asleep again.

In the command handler, the handler thread is wating for the `is_free()`condition to be become true on the repository. If this happens to be the case it succesfully acquires the repository and mark's itself as the owner, after which it free's the mutex. If any other competing handler tries to acquire the repository for update, at this instance the `is_free()` method will return false and the thread will block by calling `wait()` on the condition variable.

Once the successful thread is done updating it removes itself as the owner of the repository and call's `signal()` on the condition variable. This causes the blocked thread to wake up, check if the repository `is_free()`, and if so go on its merry way acquiring the repository for update.

You may notice that the blocking thread does not release the mutex before it fall's asleep on `wait()`, nor does it try to `acquire()` it once it wakes up. This is because the condition variable ensure the automatic release of the mutex right before falling asleep and acquisition of the mutex just before waking up.

```
int
HA_CommandHandler::svc()
{
    ACE_DEBUG((LM_DEBUG, "(%t) Handler Thread running\n"));
```

```
    for(int i=0; i < NUM_USES; i++)
    {
        mutex_.acquire();
        while(!this->rep_.is_free())
            this->waitCond_.wait();
        this->rep_.set_owner(this);
        mutex_.release();

        this->rep_.update_device(i);

        ACE_ASSERT(this->rep_.is_owner(this));
        this->rep_.set_owner(0);

        this->waitCond_.signal();
    }

    return 0;
}
```

As usual we create two handlers that compete with each other to acquire the repository, both are passed the condition variable, mutex and repository by reference when the handler is constructed.

The `ACE_Condition<>` class is a template and requires the type of mutex being used as it's argument, since in this case we are co-ordinating access between threads we use the `ACE_Thread_Mutex` type as it's argument the type constructing an `ACE_Conditon<ACE_Thread_Mutex>` type. The condition variable instance also keeps a reference to the mutex, so that it can automatically acquire and release it on the wait() call, as described earlier. This reference is passed to the condition variable during construction.

```
int main(int argc, char *argv[])
{
    HA_Device_Repository rep;
    ACE_Thread_Mutex rep_mutex;
    ACE_Condition<ACE_Thread_Mutex> wait(rep_mutex);

    HA_CommandHandler handler1(rep, wait, rep_mutex);
    HA_CommandHandler handler2(rep, wait, rep_mutex);

    handler1.activate();
    handler2.activate();
```

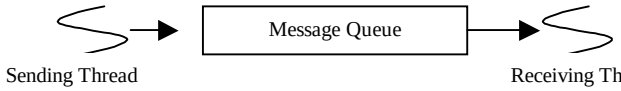
```
    handler1.wait();
    handler2.wait();


    return 0;
}
```

12.3.2 Message Passing

As we mentioned earlier message passing is often used to communicate data and event occurences between threads. This works by having the sender create a "message" that it enqueues on a message queue for the receiver to pick up. The receiver is either blocked or polling the queue waiting for new data to arrive. Once the data is in the queue it dequeues the data, uses it, and then goes back to waiting for new data on the queue.

The queue acts as a shared resource between the two threads and thus the enqueue and dequeue operations must be protected. It also would be handy if the queue supports a blocking dequeue call that unblocks when new data arrives. Finally, it would be convienient if each task object that we created came out of the box with a message queue attached to it, that way we don't have to have a global queue. Instead if a task has a reference to any other task it can send it messages.

Figure 12.1. Using a queue for communication



Fortunately, all of these features come out of the box with ACE. Up to this point we have been using `ACE_Task_Base` as the base class for all of our example threads. ACE also provides a facility to queue messages between threads that are derived from the `ACE_Task<>` template. By deriving your class from this template you automatically get an internal message queue of type `ACE_Message_Queue<>` which you can use with each of your tasks. The message queue provides a type safe interface, allowing you to enqueue messages that are instances of `ACE_Message_Block`.

Message Blocks

The `ACE_Message_Block` is an efficient data container that can be used to efficiently store and share messsages. You can think of the message block as an advanced data buffer that supports nice features such as reference counting and data sharing. Each message block contains two pointers, a `rd_ptr()` that points to the next byte to be read and a a `wr_ptr()`, that points to the next available empty byte. You can use these pointers to copy data into and get data out of the message block.

You can either use the `copy()` method to copy data into the message block

```
ACE_Message_Block *mb;
    ACE_NEW_RETURN(mb,
                ACE_Message_Block(128),
                -1);

const char * deviceAddr= "Dev#12";
mb->copy(deviceAddr, ACE_OS::strlen(deviceAddr)+1);
```

or you can use the `wr_ptr()` directly. When doing so you must move the `wr_ptr()` forward manually, so that the next write is at the end of the buffer.

```
ACE_Message_Block *mb;
ACE_NEW_RETURN(mb,
ACE_Message_Block(128),
                -1);

const char * commandSeq= "CommandSeq#14";
ACE_OS::sprintf(mb->wr_ptr(), commandSeq);
mb->wr_ptr(ACE_OS::strlen(commandSeq) +1);
//move the wr_ptr() forward in the buffer by the
//amount of data we just put in.
```

The `rd_ptr()` is similar to the write pointer, you can use it directly to get the data, but must be careful to move it forward by the number of bytes you have already read so that the next read access is at the right location in the buffer. Once you are done working with the message block you `release()` it, causing the reference count to be decremented, and if the count is 0 ACE will automatically release the memory that was allocated for the block.

```
ACE_DEBUG((LM_DEBUG,
        "Command Sequence --> %s\n",
        mb->rd_ptr()));
mb->rd_ptr(ACE_OS::strlen(mb->rd_ptr())+1);

mb->release();
```

Message blocks also include a type field, that can be set during construction or through the `msg_type()` modifier. The message type field comes in handy when you want to distinguish processing of the message based on it's type, or you wish to send a simple command notification. An example of the later is the use of `ACE_Messsage_Block::MB_HANGUP` message type to inform the message reciever that the source has shut down.

```
ACE_NEW_RETURN(mb,
                ACE_Message_Block(128, ACE_Message_Block::MB_HANGU
P),
                   -1);
//send a hangup notification to the reciever

mb->msg_type(ACE_Message_Block::MB_ERROR);
//send an error notification to the reciever
```

The message block also offers methods to `duplicate()` (create a shallow copy i.e., increment the reference count), and `clone()` (create a deep copy) of the message block.

Using the Message Queue

To illustrate the use of ACE_Task<> and it's underlying message queue we extend our previous automation handler example to include an `ACE_Svc_Handler<ACE_SOCK_STREAM, ACE_MT_SYNCH>`, called `Message_Receiver`, that recieves command messages, for the devices on the network, from TCP connected remote clients (for more on this see *Basic TCP/IP socket use*). The `Message_Reciever` on receipt of the commands first encapsulates them in message blocks and then proceeds to enqueue them on the command handler's message queue. The `HA_Command_Handler` derives from `ACE_Task<>` instead of `ACE_Task_Base`, thus inheriting the required message queue functionality. The command handler thread spends it's time waiting for command messages blocks to arrive on it's message queue, and on receiving these messages proceeds to process the commands.

The remote command messages have a simple header, `DeviceCommand-Header`, followed by a payload that consists of a null terminated command string;

```
struct DeviceCommandHeader
{
    int length_;
    int deviceId_;
};
```

The `Message_Receiver` service handler uses the `length_` field of the header to figure out the size of the payload. Once it knows the length of the payload is can create an ACE_Message_Block of the exact size i.e, the length of the payload + the length of the header. The handler then copies the header and payload into the message block, and enqueues it on the command handler task's message queue using the `ACE_Task<>::putq()` method (a reference to the `HA_Command_Handler` is kept by the `Message_Receiver`, which it receives on construction). Notice that if the device id read in from the header is negative we use it as an indication that the system needs to be shut down, to do this we send a shut down message to the `HA_Command_Handler`, more on this in a moment.

```
int
Message_Receiver::handle_input(ACE_HANDLE fd)
{
    ACE_UNUSED_ARG(fd);

    DeviceCommandHeader dch;
    if(this->read_header(&dch) < 0)
        return -1;

    if(dch.deviceId_ < 0)
    {
        this->handler_->putq(shut_down_message());
        return -1;
    }
    //handle shutdown


    ACE_Message_Block *mb;
    ACE_NEW_RETURN(mb, ACE_Message_Block(dch.length_ + sizeof dch)
, -1);
    mb->copy((const char*)&dch, sizeof dch);
    //copy the header

    if(this->copy_payload(mb, dch.length_) < 0)
```

```
        ACE_ERROR_RETURN((LM_ERROR, "%p\n", "Recieve Failure"), -1
);
    //copy the payload

    this->handler_->putq(mb);
    //pass it off to the handler thread

    return 0;
}
```

It is instructive to take a look at how the payload is copied in to the provided message block.. Here we provide the wr_ptr() directly to the ACE_SOCK_Stream::read_n() method that copies the data directly into the block. After the data is copied into the block we forward the wr_ptr() by the size of the payload, remember that we had moved the wr_ptr() for the message block forward by the sizeof the header before we made this call, therefore the message block now contains the header immediately followed by the payload.

```
int
Message_Receiver::copy_payload(ACE_Message_Block *mb, int payload_
length)
{
    int result =
        this->peer().recv_n(mb->wr_ptr(), payload_length);

    if(result <= 0)
    {
        mb->release();
        return result;
    }

    mb->wr_ptr(payload_length);

    return 0;
}
```

When the message receiver gets a command to shut down the system it creates a new message block that has no data in it but has the `ACE_Message_Block::MB_HANGUP` flag set. When the `HA_Command_Handler` receives a message, it first checks the type and if it is a hangup message it shuts down the system.

```
ACE_Message_Block *
Message_Receiver::shut_down_message()
{
    ACE_Message_Block *mb;
    ACE_NEW_RETURN(mb, ACE_Message_Block(0, ACE_Message_Block::MB_
HANGUP), 0);
    return mb;
}
```

On the other side of the fence the `HA_CommandHandler` thread blocks waiting for messages to arrive on it's queue by calling `getq()` on itself. Once a message arrives `getq()` will unblock, the handler then reads the messages and applies the received command to the device repository. Finally, it `release()`'s the message block, which will deallocate the used memory as the reference count to the block drops to zero.

As we said before the system uses a message of type `ACE_Message_Block::MB_HANGUP` to inform the server to shut down. On receiving a message that is of this type the handler stops waiting for incoming messages and shut's down the reactor, using the `ACE_Reactor::end_event_loop()` method. This causes the command handler process to shut down.

```
int
HA_CommandHandler::svc(void)
{
    while(1)
    {
        ACE_Message_Block *mb;
        ACE_ASSERT(this->getq(mb)!=-1);
        if(mb->msg_type() == ACE_Message_Block::MB_HANGUP)
        {
            mb->release();
            break;
        }
        else
        {
            DeviceCommandHeader * dch
                = (DeviceCommandHeader*)mb->rd_ptr();
            mb->rd_ptr(sizeof (DeviceCommandHeader) );

            char *payload =
                mb->rd_ptr();
```

```
            ACE_DEBUG((LM_DEBUG,
                "Message for device #%d with command payload of:\n
%s",
                dch->deviceId_, mb->rd_ptr()));

            this->rep_.update_device(dch->deviceId_, mb->rd_ptr())
;

            mb->release();
        }
    }

    ACE_Reactor::end_event_loop();

    return 0;
}
```

12.4 Summary

The basic high level ACE threading components provide an easy to use object interface to multi-threaded programming. In this chapter we went over 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 finally how to use `ACE_Condition` and the `ACE_Message_Queue`, `ACE_Message_Block`, `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 different types of threads in varying states in accordance with your requirements. We also get into how runing threads can be managed, this includes suspending and resuming threads, creating and waiting on threads that are in “thread groups”, cancelling running threads and incorporating startup 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 in 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.

The table below lists all the possible thread creation flags that control the assignment of different attributes to the new thread.;

Table 13.1. Thread Creation Flags

Flag	Description
THR_CANCEL_DISABLE	Do not allow this thread to be cancelled.
THR_CANCEL_ENABLE	Allow this thread to be cancelled.
THR_CANCEL_DEFFERED	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.

Since 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 OS. For example if your particular OS does not support round robin scheduling then don't expect ACE to do magic to make that work for you. However, on most major OS's you will find that everything works as expected.

The flags themselves can be OR'ed together and are passed as the first argument to the activate call. For example, the following call would cause the thread to be created as a kernel schedulable thread, with a default scheduling scheme, start in the suspended state and it would be possible to join with this thread.

```
handler.activate(THR_NEW_LWP|THR_SCHED_DEFAULT|
```

```
THR_SUSPENDED|THR_JOINABLE)
```

13.1.1 Scheduling scope

Most operating systems make the distinction between kernel level threads and user level threads. Kernel level threads are schedulable entities that the OS is aware of and schedules with a kernel level scheduler. On the other hand user level threads are lighter weight threads that are scheduled with a library-based scheduler within the processes address space. These threads allow for fast context switching but could block on each other. As a programmer you can *only* deal with user level threads. Your process is assigned a pool of kernel level threads that are used to schedule all your user level threads. You can explicitly bind a user level thread to a particular kernel level thread using the `THR_BOUND` flag. Doing this would cause the bound user level thread to be scheduled using the underlying kernel level thread. Figure 1 illustrates these concepts.

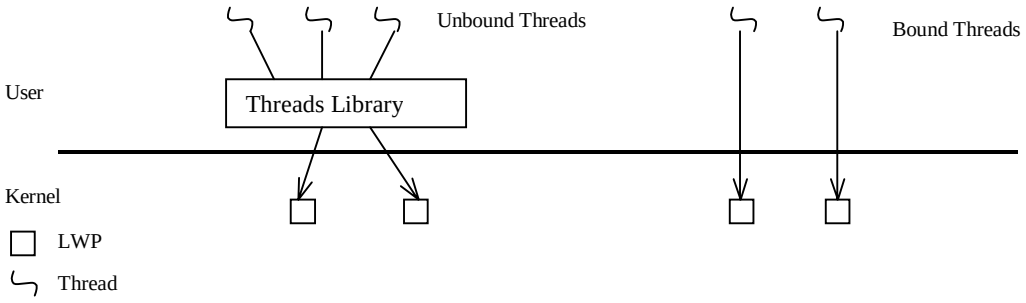


Figure 13.1. Kernel and User Leve Threads

If you specify that you want to create a thread with the `THR_NEW_LWP` flag set you are ensured that a new kernel level schedulable entity will also be created (i.e., the new user level thread is bound to a new kernel level thread). If you do not specify this flag the library will create the thread as a user level thread. The `ACE_Task_Base::activate()` method defaults to always creating a new kernel level thread.

13.1.2 Detached and Joinable Threads

Threads can be created as detached or joinable using the `THR_DETACHED` or `THR_JOINABLE` flags.

If you specify a thread as detached the OS thread library will automatically clean up all resources the thread held when it exits. Unfortunately, the exit status of this thread is not available for any other thread to look at. On the other hand if a thread is created as joinable another thread can *join* with it when it terminates.

We have been using the `THR_JOINABLE` feature in all of our previous examples, the main thread waits (or *joins*) with the child thread by calling `wait()` on the task object. As you may have guessed by default all `ACE_Task` threads are created as joinable, thereby making the `wait()` method possible. If several threads wait for the same thread to complete only one will actually join with it. The other waiting threads will unblock and return with a failure status. If you do specify a thread as being joinable you *must* join with it at some later point in your program. Thus with ACE it is manadatory for you to wait on each task to ensure proper cleanup. If you fail to do this you may leak system resources.

13.1.3 Initial Thread Scheduling State

Once the `ACE_Task_Base::activate()` method returns a new thread has been created by the OS. However, this does not say anything about the current running state of the new thread. The OS will use an internal OS specific scheduling scheme to decide when to actually run the new thread. So even after the thread has been created it may not actually be running when `activate()` returns.

You can control the initial running state of a thread to some degree by specifying the `THR_SUSPENDED` flag. This will cause the OS to create the new thread in a suspended state. This often comes in handy when you are activating a new thread and you do not want it to race for some resource that is currently held by the spawning thread.

```
HA_CommandHandler handler;
int result =
    handler.activate(THR_NEW_LWP |
                     THR_JOINABLE |
                     THR_SUSPENDED);
ACE_ASSERT(result == 0);

ACE_DEBUG((LM_DEBUG,
    "(%t) The current thread count is %d\n",
```

```
    handler.thr_count()));
ACE_DEBUG((LM_DEBUG,
    "(%t) The group identifier is %d\n",
    handler.grp_id()));

handler.resume();
handler.wait();
```

Here we `activate()` the thread, however since we specify the `THR_SUSPENDED` flag, the new thread is created in the suspended state and does not run. This allows us to display the current thread count (which is 1) and the automatically assigned group id of the handler task, without worrying about the child thread completing and altering the handlers state before we are done with our displaying fun. After we complete we call `resume()` on the handler, causing the thread be scheduled and run by the OS for the first time.

13.2 Priorities

One very 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 priorites to create two different `HA_CommandHandler` instances that run at differing priorities. The high priority command handler receives critical commands that must be executed before any of the non-critical commands are, for example, changing the temperature on the oven could be more critical then recording your favorite TV program.If you were using strict priority based scheduling this would mean that the low priority handler would only run if the application was not busy processing critical commands in the high priority handler thread.

Most general-purpose operating systems only exhibit weak real-time characteristics, i.e., a combination time-shared, real-time scheme is supported. Each OS defines priorities in a proprietary fashion, for example Solaris defines priorities to range from 0-127 with 127 being the highest priority, Win32 specifies a range of 0-15 with 15 being the highest and VxWorks has 0-255 with 0 being the highest priority. ACE does not handle these differences instead making it the programmers responsibility (read yours) to be sure that the range passed in to calls that set the priority work corretly on the particular platform you are dealing with.

To create tasks with different priorities you just specify the required priority as the fourth parameter to the `activate()` call. In this case we create two

instances of our home automation command handler, and give them the appropriate names “HighPriority” and “LowPriority” and `activate()` them with low and high priority values. We then proceed to pump 100 messages onto each handlers message queue, expecting to see the high priority handler process these messages before the low priority handler does.

```
int main(int argc, char *argv[])
{

    HA_CommandHandler hp_handler("HighPriority");
    hp_handler.activate(THR_NEW_LWP|
        THR_JOINABLE, 1, 1, HI_PRIORITY);

    HA_CommandHandler lp_handler("LowPriority");
    lp_handler.activate(THR_NEW_LWP|
        THR_JOINABLE, 1, 1, LO_PRIORITY);

    ACE_Message_Block mb;
    for(int i=0; i < 100; i++)
    {
        hp_handler.putq(&mb);
        lp_handler.putq(&mb);
    }

    hp_handler.wait();
    lp_handler.wait();

    return 0;
}
```

The handler on reciept of the message calls `process_message()`, that displays the tasks name and then simulates a compute bound task.

```
class HA_CommandHandler:
    public ACE_Task<ACE_MT_SYNCH>
{
public:
    HA_CommandHandler(const char* name)
    {
        name_ = name;
    }

    virtual int svc()
    {
        ACE_DEBUG((LM_DEBUG,
```

```
                    "(%t) starting up %s\n",
                    name_));

                ACE_OS::sleep(2);

                while(1)
                {
                    ACE_Message_Block *mb;

                    int result =
                        this->getq(mb);

                    process_message(mb);
                }

                return 0;
            }

            void process_message(ACE_Message_Block *mb)
            {
                ACE_DEBUG((LM_DEBUG,
                     "(%t) Processing message %s\n",
                      name_));
                for(int i =0 ; i < 100; i++);
                //simulate compute bound task.
            }

        private:
            const char* name_;
        };
```

The results are as expected, the high priority handler always gets to run before the low priority handler.

```
(2932) starting up HighPriority
(1116) starting up LowPriority
(2932) Processing message HighPriority
(2932) Processing message HighPriority
(2932) Processing message HighPriority
<continues 100 times>
(1116) Processing message LowPriority
(1116) Processing message LowPriority
(1116) Processing message LowPriority
```

```
<continues 100 times>
```

Once again we cannot emphasize enough that setting thread priorities that work correctly for your application is a task that you must take onto yourself. ACE does not provide much help in this area. Make sure that the values you pass to `activate()` work as you expect them to.

13.3 Thread Pools

Uptil this point we have been creating an `ACE_Task` sub-class 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 will internally assign a group identifier to all of the threads in a task, available through the `grp_id()` accesor, that can be used for subsequent management operations on the group. All of the created threads share the same `ACE_Task<>` object but have their own seperate stacks.

This is a handy way to create a simple pool of worker threads that share the same message queue. All of 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 each other or employ 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 of 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 using the `ACE_DEBUG %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()
    {
```

```
            ACE_DEBUG((LM_DEBUG, "(%t) starting up \n"));


            ACE_Message_Block *mb;

            int result =
                this->getq(mb);

            //... do something with the message.


            return 0;
        }
    };
```

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

```
int main(int argc, char *argv[])
{
    HA_CommandHandler handler;
    handler.activate( THR_NEW_LWP | THR_JOINABLE, 4);
    //create 4 threads

    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
(3224) starting up
(3368) starting up
(876) starting up
```

13.4 Thread Management

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(), resume()` methods that are 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, `ACE_Thread_Manager` class to help manage all ACE created tasks, for example providing operations that suspend, resume and cancel all threads in all tasks. The manager provides a rich interface that includes group management, creation, state retrieveal, cancellation, thread startup and exit hooks etc. The thread manager is also tightly integrated with the ACE `Streams` framework that we will cover later in this book.In the following sections we will take a look at some of the features that the ACE_Thread_Manager provides.

Thread Exit Handlers

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 (that causes a thread to exit immediately).

To setup an exit functor you first create a subclass of `ACE_At_Thread_Exit`, implementing 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()
  {
    ACE_DEBUG((LM_INFO, "(%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 since we want the functor to be applied when the home automation handler thread exits 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()` accesor 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()
    {
        ACE_DEBUG((LM_DEBUG, "(%t) starting up \n"));

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

        //do something.

        ACE_Thread::exit();
        //forcefully exit
```

```
        // NOT REACHED

        return 0;
    }
private:
    ExitHandler& eh_;
};
```

The exit handler and command handler objects are created on the stack of the `main()` function, as usual. An interesting twist here is that instead of using the `ACE_Task_Base::wait()` method, to wait for the handler thread to exit, we use the `ACE_Thread_Manager's wait()` method. The thread manager waits on all child threads, no matter what task they are associated with. This proves convenient when you have more then one executing task.

```
int main(int argc, char *argv[])
{
    ExitHandler eh;

    HA_CommandHandler handler(eh);
    handler.activate();

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

    return 0;
}
```

Notice that we treat the thread manager as a singleton object, this is because the ACE default is to manage all threads with this thread manager singleton. You can create and specify more then one thread manager, but in most cases you will find the default behaviour to be sufficient.

```
int main(int argc, char *argv[])
{
    ExitHandler eh;
    ACE_Thread_Manager tm;

    HA_CommandHandler handler(eh);
    handler.thr_mgr(&tm);
    handler.activate();
```

```
    tm.wait();

    return 0;
}
```

13.5 Signals

We have dedicated an entire chapter in this book on the topic of Signals and their use within the ACE framework. This section goes over the way signals are done differently in a multi-threaded application versus a single threaded application.

First of all each thread has it's own private signal mask which is inherited during creation by default. 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, i.e., you can only have a single signal handler for a particular signal type.

13.5.1 Signalling Threads

You can explicitly send targeted synchronous signals to threads using the thread manager. The manager allows you to send a signal to all threads, to a a thread group or task, or to a particular thread. Since all tasks derieve from `ACE_Event_Handler`, the best place to handle the signal is to use the inbuilt `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 of the threads in a thread group. We start off 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,
                    "(%t) received a %d signal\n" ,
                    signum));

                handle_alert();
            }

            return 0;
        }

        virtual int svc()
        {
            ACE_DEBUG((LM_DEBUG, "(%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);
    };
```

We first create the task and activate it with five new threads. We then register the task as the handler for `SIGUSR1` with an `ACE_Sig_Handler` dispatcher object. This will ensure that the `handle_signal()` method of the task is automatically called back when a signal of type `SIGUSR1` is received by a thread. Finally we use the thread managers `kill_grp()` method to send a `SIGUSR1`

signal to all of the threads within the grp. The group id we use is the group identifier for the threads in the `SignalableTask`, therefore our `kill_grp()` causes each one of the threads in the task to receive a `SIGUSR1` signal that they handle in their own context.

```
int main(int argc, char *argv[])
{
    SignalableTask handler;
    handler.activate(THR_NEW_LWP| THR_JOINABLE , 5);

    ACE_Sig_Handler sh;
    sh.register_handler(SIGUSR1, &handler);

    ACE_OS::sleep(1);

    ACE_Thread_Manager::instance() ->
        kill_grp(handler.grp_id(), SIGUSR1);

    handler.wait();

    return 0;
}
```

The output shows each thread receiving and then handling the signal in it's own context

```
(1026) Starting thread
(2051) Starting thread
(3076) Starting thread
(4101) Starting thread
(5126) Starting thread
(1026) received a 10 signal
(2051) received a 10 signal
(5126) received a 10 signal
(4101) received a 10 signal
(3076) received a 10 signal
```

13.5.2 Signalling Processes in Multithreaded programs

One difficult question to answer is when a signal is sent to a process which thread of control is used to actually handle the signal? The answer is of course that it depends (on many things in fact). If the signal is synchronous i.e., it is a signal that

was raised due to an error in execution of a thread (for example a SIGSEGV or a SIGFPE) then it's handled by the same thread that caused it be raised. On the other hand if it is asynchronous (for example SIGINT, SIGTERM etc.) then it can be handled by any thread running in the application.

These rules are of course affected by whatever platform you are running on. In general Win32 programmers do not use signals and UNIX or UNIX like platforms that support POSIX will support these rules. This includes general purpose and real-time operating systems.

We modify our previous example to send SIGUSR1, a synchronous signal, several times to the process instead of using the thread manager to send signals to particular threads. Since these signals are synchronous and are sent by the main thread we expect them to be handled by the main thread.

```
int main(int argc, char *argv[])
{
    ACE_DEBUG((LM_DEBUG, "(%t) Main thread \n"));
    SignalableTask handler;
    handler.activate(THR_NEW_LWP| THR_JOINABLE , 5);

    ACE_Sig_Handler sh;
    sh.register_handler(SIGUSR1, &handler);

    ACE_OS::sleep(1);

    for(int i=0; i < 5; i++)
        ACE_OS::kill(ACE_OS::getpid(), SIGUSR1);

    handler.wait();

    return 0;
}
```

The output (generated on Linux) illustrates how the signal is handled by the same thread (in this case the main thread) every single time.

```
(1024) Main thread
(1026) Starting thread
(2051) Starting thread
(3076) Starting thread
(4101) Starting thread
(5126) Starting thread
(1024) received a 10 signal
```

```
(1024) received a 10 signal
(1024) received a 10 signal
(1024) received a 10 signal
(1024) received a 10 signal
```

Signal handling in multiple threaded programs varies based on the platform and is plagued with inconsistencies, therefore it is our suggestion that if you do combine asynchronous signals with multi-threaded programs, take some time to understand the exact semantics on your particular platform.

13.6 Thread Startup Hooks

ACE provides a global thread startup hook that can be used to intercept the call to the thread startup function. This allows a programmer to perform any kind of initialization that is globally applicable to all the threads that are being used in your application. This is an especially good spot to add threads specific data to a thread (we talk about thread specific data in *Thread Safety and Synchronization*).

To setup a startup hook you must first subclass `ACE_Thread_Hook` and implement the virtual `start()` method. This method is passed a pointer to the user specified startup function, for the new thread, and a `void*` argument that must be passed to this function. In most cases you will execute your special startup code and then call the user specified entry point with the supplied arguments. 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, this helps us to determine whether the client has permission to execute specified commands. After setting up the security context we execute the user specified thread entry point.

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

        ACE_TSS<SecurityContext> secCtx;
        //create the context on the threads
```

```
        //own stack

        add_sec_context_thr(secCtx);
        //special initialization

        void *status
            = ((void*)((*func)(arg)));

        return status;
    }


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

After creating your new startup hook you need to set it as the new startup hook. To do this all you have to do to is call the `ACE_Thread_Hook::thread_hook()` static method passing in an instance of the new hook.

```
int main(int argc, char *argv[])
{
    HA_ThreadHook hook;
    ACE_Thread_Hook::thread_hook(&hook);

    HA_CommandHandler handler;
    handler.activate();


    handler.wait();

    return 0;
}
```

13.7 Cancellation

Cancellation, except for cooperative cancellation, is best avoided, unless you can justify a real need for it. With that off of our chest let's delve into what cancellation is and why we consider it to be a feature that is 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 just cease to exist. Talk about "ungraceful" exits. In certain instances cancellation may be a necessary evil, for example 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 below, is the only cancellation mode that does not suffer from these ungraceful exit problems.

There are several different *modes* of cancellation;

- **Deferred Cancelability:** When a thread is in this mode all cancels are deferred till 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 (for example on an I/O call) or an explicit `ACE_Thread_Manager::testcancel()` has been made. This mode is the default for applications built with ACE.
- **Cooperative cancellation:** If you wish to build portable applications it is best if you stick with this cancellation mode. In this mode thrads are not really cancelled but instead have their state marked as canceled within the `ACE_Thread_Manager`. You can call `ACE_Thread_Manager::testcancel()` to determine whether the thread is in the canceled state or not, if so you can choose to exit the thread. This mode also gets around most of the nasty side affects we talked about earlier that come with regular cancellation.
- **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 cancelled when executing critical sections of code. You can also use cleanup handlers, that are called when a thread is cancelled, to ensure 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 platform that is POSIX then look at for these features (i.e., 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

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

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

    virtual int svc()
    {
        ACE_DEBUG((LM_DEBUG, "(%t) starting up \n"));
        //do something.


        while(1)
        {
            if(this->thr_mgr()->
                testcancel(
                  this->thr_mgr()->thr_self()))
                ACE_Thread::exit();


            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
            {
                //do real work.
            }
        }
```

```
        return 0;
    }
};
```

The task itself can be cancelled from anywhere using the thread manager's cancel_task() method, in this case the main thread cancels the task after waiting for one second.

```
int main(int argc, char *argv[])
{

    CanceledTask task;
    task.activate();

    ACE_OS::sleep(1);

    ACE_Thread_Manager::instance()->
        cancel_task(&task);

    task.wait();

    return 0;
}
```

13.7.2 Asynchronous Cancellation

This next example illustrates how to use asynchronous cancelability to zap a thread in a tight compute loop (note that this example is platform dependent). Such a thread can't be cancelled in deferred cancellation mode (since there are no cancellation points), therefore we switch the cancel state of this thread to be asynchronous.

```
class CanceledTask:
public ACE_Task<ACE_MT_SYNCH>
{
public:
    virtual int svc()
    {
        ACE_DEBUG((LM_DEBUG,
            "(%t) Starting thread\n"));
        if(this->set_cancel_mode() < 0)
```

```
            return -1;

        while(1)
        {
            //put this thread in a compute loop.. no
            //cancellation points are available
        }
    }

    int set_cancel_mode()
    {
        cancel_state new_state;
        new_state.cancelstate =
                PTHREAD_CANCEL_ENABLE;
        new_state.canceltype =
                PTHREAD_CANCEL_ASYNCHRONOUS;
        //set the cancel state to asynchronous and enabled.

        if(ACE_Thread::setcancelstate(new_state, 0) == -1)
            ACE_ERROR_RETURN((LM_ERROR,
                "%p\n", "cancelstate"), -1);
        //set the cancellation state

        return 0;
    }
};
```

We cancel the thread using the thread manager `cancel_task()` method as we did before, however in this case we pass a second argument of true, indicating that we want the task to be cancelled asynchrounously (int other words we want it to be zapped).

```
int main(int argc, char *argv[])
{
    CanceledTask task;
    task.activate();

    ACE_OS::sleep(1);

    ACE_Thread_Manager::instance() ->
        cancel_task(&task, 1);
```

```
    task.wait();

    return 0;
}
```

13.8 Summary

In this chapter we went over how to create different types of threads 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 startup in your applications.

Chapter 14
Thread Safety and Synchronization

We have already covered the basics of both thread safety and synchronization in *Basic Multithreaded programming*. In that chapter we went over 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 what we have learned earlier. In the safety arena, you will get to see readers/writers locks, the atomic op wrapper, using tokens for fine grain locking on data structures and finally and introduction to recursive mutexes. Whereas we introduce semaphores, barriers and as new synchronization mechanisms

14.1 Protection Primitives

To ensure consistency, multithreaded programs must use protection primitives around shared data. We have already seen some examples of protection in *Basic Multithreaded Programming*, where we introduced mutexes and guards. Here we will go over the remaining ACE provided protection primitives.

The table below lists the primitives that are available in ACE. All of these primitives have a common interface but do not share type relationships and therefore cannot be polymorphically substituted at run-time.

Table 14.1. Protection Primitives in ACE

Primitive	Description
ACE_Mutex	Wrapper class around the mutual exclusion mechanism and are used to provide a simple and 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*" i.e., can be acquired multiple times by the same thread. Note that it must be released as many times as it was acquired to work correctly.
ACE_RW_Mutex	Wrapper class that encapsulates readers/writers locks. These are locks that are acquired differently for reading and writing, thus enabling multiple readers to read while no one is writing. These locks are non-recursive.
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 i.e., all threads that call acquire enter a FIFO queue and acquire the lock in order.
ACE_Atomic_Op<T>	Template wrapper that allows for *safe* arithmetic operations on **T**.
ACE_Guard<T>	Template based Guard class that takes the lock type as a 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 during construction.

14.1.1 Recursive Mutexes

Consider the scenario where a thread acquires the same mutex twice, what do you think will happen, will the thread block or not? If you think it won't you are wrong. In most cases the thread will block, deadlocking on itself! This is because in general mutexs are unaware who the acquiring thread is, thus making it impossible for the mutex to make the smart decision of not blocking.

If you are using a mutex from several inter-calling methods it sometimes become difficult to track whether a thread already has acquired a particular mutex or not. In the code below the logger will deadlock on itself once it recieves a critical log message, the guard has already acquired the mutex in the `log()` method and will then try to reacquire it from within `logCritical()`.

```
typedef ACE_Thread_Mutex MUTEX;
class Logger
{
public:
    void log(LogMessage* msg)
    {
        ACE_GUARD(MUTEX, mon, mutex_);
        if(msg->priority() == LogMessage::CRITICAL)
            logCritical(msg);
        //performing logging
    }

    void logCritical(LogMessage* msg)
    {
        ACE_GUARD(MUTEX, mon, mutex_);
        //performing logging
    }
private:
    MUTEX mutex_;
};

static Logger logger;

int main(int argc, char *argv[])
{
    CriticalLogMessage cm;
    logger.log(&cm);
    //will cause deadlock.

        return 0;
}
```

This can be avoided by using a recursive mutex. Recursive mutexes, unlike regular mutexes, allow the same thread to acquire the mutex multiple times without blocking on itself. To achieve this we just modify the initial typedef;

```
typedef ACE_Recursive_Thread_Mutex MUTEX;
```

The `ACE_Recursive_Thread_Mutex` has the exact same API as regular mutexes making them readily substitutable wherever a regular mutex is used.

14.1.2 Readers/Writers Locks

A readers/writer lock allows many threads to hold the lock simultaneously for reading, but only one thread can hold it for writing. This can provide efficiency gains especially if the data that is being protected is frequently read by multiple threads but is infrequently written, or written to by only a few threads. In most cases readers/writers lock are slower then mutexes, therefore it important for you to apply them only in situations where contention on reads is much higher then on writes.

ACE includes readers/writers lock in the form of the `ACE_RW_Mutex/ ACE_RW_Thread_Mutex` classes. To acquire these mutexes for reading you need to call `acquire_read(),` whereas to acquire the mutex for writing you must call `acquire_write()`.

Lets say that our home network includes a network discovery agent that keeps track of all the devices that are currently connected on the home network. The agent keeps a list of all devices that are currently on the network and clients can ask it whether a current device is currently present or not. Lets also say, for this examples sake, that this list could be very long and that devices are added or removed from the network infrequently. This gives us a good opportunity to apply a readers/writers lock, since the list is traversed, or read, much more then it is modified. We use the guard macros we saw in *Basic Multithreaded Programming*, to ensure that the `acquire_read()/ release()` and `acquire_write()/ release()` are called in pairs (i.e., we don't forget to call `release()` after we are done with the mutex).

```
class HA_DiscoveryAgent
{
public:
    void add_device(Device * device)
    {
```

```
        ACE_WRITE_GUARD(ACE_RW_Thread_Mutex, mon, rwmutex_);

        list_add_item_i(device);
    }

    void remove_device(Device * device)
    {
        ACE_READ_GUARD(ACE_RW_Thread_Mutex, mon, rwmutex_);

        list_remove_item_i(device);
    }

    int contains_device(Device * device)
    {
        ACE_READ_GUARD_RETURN(ACE_RW_Thread_Mutex, mon, rwmutex_,
-1);

        return list_contains_item_i(device);
    }

private:
    void list_add_item_i(Device * device);
    int list_contains_item_i(Device * device);
    void list_remove_item_i(Device* device);

private:
    DeviceList deviceList_;
    ACE_RW_Thread_Mutex rwmutex_;
};
```

14.1.3 Atomic Op. Wrapper

On most machine architectures changes to basic types are atomic, that is an increment of an integer variable does not require the use of synchronization primitives. However, on most multi-processor's this is not true and is dependent on the memory *ordering* properties of the machine. If the machine memory is *strongly ordered* then modifications made to memory on one processor are immediately visible to the other processor's, thus synchronization is not required around global variables, otherwise it is.

To help to achieve transparent synchronization around basic types ACE provides the `ACE_Atomic_Op<>` template wrapper. This class overloads all basic

operations and ensures that a synchronization guard is used before the operation is performed.

To illustrate the use of atomic op we will solve the classic producer/ consumer problem using busy waiting. To implement our solution we create a producer and a consumer task. Both tasks share a common produce buffer, the items that are produced are put into the buffer by the producer, the consumer in turn picks up these items. To ensure that the producer doesn't overflow the buffer and the consumer doesn't underflow we use `in` and `out` counters. Whenver the producer adds an item it increments the `in` counter by 1, similarily when the client consumes an item it increments the `out` counter by 1. The producer can only produce if `(in – out)` is not equal to the buffer size (i.e., the buffer isnt full), otherwise it must wait. The client can only consume if `(in – out)` is not zero (i.e., the buffer isn't empty). Lets also assume that the program is running on a multi-processor where memory is not *strongly ordered*. This means that if we use regular integers as our `in` and `out` counters the increments will **NOT** be consistent, the producer may increment the `in` count but the client will not see the increment and vice versa.To ensure consistency we first need to create a safe unsigned int type that we can use for the counters in our example;

```
typedef ACE_Atomic_Op<ACE_Thread_Mutex, unsigned int> SafeUInt;
```

The actual data passed between consumer and producer is an integer, that (lets imagine for the examples sake) must be safely incremented to be produced correctly. We create a safe int type as ;

```
typedef ACE_Atomic_Op<ACE_Thread_Mutex, int> SafeInt;
```

The producer and consumer tasks share the common production buffer and the in and out counter types, which we create on the main stack and pass by reference to each of the tasks. The producer uses a SafeInt, itemNo as the production variable, which it increments on each production and then adds to the buffer. We add a termination condition to each one of the threads such that after producing and consuming a fixed number of items both threads terminate gracefully. To read the actual value stored in all the atomic op types we use the `value()` accesor throughout the program.

```
class Producer
:public ACE_Task_Base
{
public:
    Producer(int *buf,
        SafeUInt &in,
        SafeUInt& out)
        :buf_(buf), in_(in), out_(out)
    {
    }
    int svc()
    {
        SafeInt itemNo = 0;
        while(1)
        {
            do
            {
            }while (in_.value() - out_.value() == Q_SIZE);
            //busy wait.


            itemNo++;
            buf_[in_.value() % Q_SIZE] = itemNo.value();
            in_++;

            ACE_DEBUG((LM_DEBUG, "Produced %d \n",
                    itemNo.value() ));

            if(check_termination(itemNo.value()))
                break;
        }

        return 0;
    }

    int check_termination(int item)
    {
        return (item == MAX_PROD);
    }
private:
    int * buf_;
    SafeUInt& in_;
    SafeUInt& out_;
};

class Consumer
```

```
:public ACE_Task_Base
{
public:
    Consumer(int *buf, SafeUInt &in, SafeUInt& out)
        :buf_(buf), in_(in), out_(out)
    {
    }
    int svc()
    {
        while(1)
        {
            int item;
            do
            {
            }while(in_.value() - out_.value() == 0);
            //busy wait.

            item = buf_[out_.value() % Q_SIZE];
            out_++;

            ACE_DEBUG((LM_DEBUG, "Consumed %d\n",
                item));

            if(check_termination(item))
                break;
        }

        return 0;
    }
    int check_termination(int item)
    {
        return (item == MAX_PROD);
    }
private:
    int * buf_;
    SafeUInt& in_;
    SafeUInt& out_;
};
```

The threads themselves are created as usual on the main stack.

```
int main(int argc, char *argv[])
{
    ACE_UNUSED_ARG(argc);
    ACE_UNUSED_ARG(argv);
```

```
    int shared_buf[Q_SIZE];
    SafeUInt in = 0;
    SafeUInt out = 0;

    Producer producer(shared_buf, in, out);
    Consumer consumer(shared_buf, in, out);

    producer.activate();
    consumer.activate();


    producer.wait();
    consumer.wait();


    return 0;
}
```

14.1.4 Token Management

Uptil this point we have been using synchronization and protection primitives on simplistic resources. However, in many cases global resources are more complex structures, such as records in a memory resident table or tree. Multiple threads act on individual records on the table by first obtaining the records, making sure that they have exclusive access to the record, making the desired modification and then releasing the record. A simplistic solution to this problem is to create a separate lock structure that maintains a lock for each record that is to be managed as a unit. However, many complex issues come into play such as how the lock structure is managed efficiently or how one avoids or detects deadlock situations, what happens if the record table also needs to be accessed by remote threads?

ACE provides a framework solution that solves all of the above problems, called the Token framework. Each record has associated with it an `ACE_Token` lock that internally maintains a strict FIFO ordering of threads that are waiting on the token. Before using a record you acquire the token and after the modification you release the token to the next waiter in line. Further more, a token manager is used to handle the tokens themselves, managing the creation, reference counting and deletion of tokens.

The following example illustrates the use of tokens to provide record level locking on a fixed size table of device records that are maintained for our auto-

mated household. The records are kept within an `HA_Device_Repository` that supports update operations. We will assume that a device can only be updated by one client at any particular instant of time. To enforce this invariant we create a token (an `ACE_Local_Mutex`) for every device which must first be acquired by a thread before it can perform an update. Once the token has been acquired the thread can go into the device table, obtain the appropriate device and update it. After completing the modification the token is released and the next thread in line can obtain the token.

```
class HA_Device_Repository
{
public:
    enum { N_DEVICES = 100 };

    HA_Device_Repository()
    {
        for(int i=0; i < N_DEVICES; i++)
            tokens_[i] = new ACE_Local_Mutex(0, 0, 1);
    }

    ~HA_Device_Repository()
    {
        for(int i=0; i < N_DEVICES; i++)
            delete tokens_[i];
    }

    int update_device(int device_id, char * commands)
    {
        this->tokens_[device_id]->acquire();

        Device * curr_device =
            this->devices_[device_id];
        internal_do(curr_device);

        this->tokens_[device_id]->release();

        return 0;
    }

    void internal_do(Device * device);

private:
```

```
    Device* devices_[N_DEVICES];
    ACE_Local_Mutex* tokens_[N_DEVICES];
    unsigned int seed_;
};
```

To illustrate the device repository in action we modify our `HA_CommandHandler` task so that it invokes `update_device()` on the repository using multiple threads.

```
class HA_CommandHandler:
public ACE_Task_Base
{
public:
    enum { N_THREADS = 5 };

    HA_CommandHandler(HA_Device_Repository &rep)
        :rep_(rep)
    {
    }

    int svc()
    {

        for(int i=0; i < HA_Device_Repository::N_DEVICES; i++)
            rep_.update_device(i, "");


        return 0;
    }
private:
    HA_Device_Repository &rep_;
};

int main(int argc, char *argv[])
{
    ACE_UNUSED_ARG(argc);
    ACE_UNUSED_ARG(argv);


    HA_Device_Repository rep;
    HA_CommandHandler handler(rep);

    handler.activate(THR_NEW_LWP| THR_JOINABLE,
        HA_CommandHandler::N_THREADS);
```

```
    handler.wait();

    return 0;
}
```

Deadlock Detection

One of the nice features that the ACE token management framework provides is deadlock detection, this comes in handy when initally designing your system to make sure that the algorithms you design do not cause deadlock.

To illustrate a deadlock situation and the ACE deadlock detection features we create two tasks, each running a single thread. Both tasks share two resources named “resource1” and “resource2”, that are protected by named tokens of the same name. Unlike the previous example where we pre-created and shared unnamed tokens, in this example we use the framework to manage the tokens for us by name. This allows both thread one and thread two to create an `ACE_Local_Mutex` named `resource1` secure in the knowledge that they are indeed sharing a single token with each other.The `mutex1` object acts as a proxy to a shared token for `resource1`. The shared token itself is reference counted and when both `mutex1` objects for thread one and two go out of scope the token is deleted.

```
class ThreadOne
    :public ACE_Task_Base
{
public:
    virtual int svc()
    {
        ACE_Local_Mutex mutex1("resource1",
            0, //deadlock detection enabled.
            1);//debugging enabled
        mutex1.acquire();

        ACE_OS::sleep(2);

        ACE_Local_Mutex mutex2("resource2",
            0, //deadlock detection enabled.
            1);//debugging enabled
        mutex2.acquire();

        return 0;
    }
```

```
};

class ThreadTwo
    :public ACE_Task_Base
{
public:
    virtual int svc()
    {
        ACE_Local_Mutex mutex2("resource2",
            0, //deadlock detection enabled.
            1);//debugging enabled
        mutex2.acquire();


        ACE_OS::sleep(2);

        ACE_Local_Mutex mutex1("resource1",
            0, //deadlock detection enabled.
            1);//debugging enabled
        mutex1.acquire();

        return 0;
    }
};

int main(int argc, char *argv[])
{
    ACE_UNUSED_ARG(argc);
    ACE_UNUSED_ARG(argv);

    ThreadOne t1;
    ThreadTwo t2;

    t1.activate();
    ACE_OS::sleep(1);
    t2.activate();

    t1.wait();
    t2.wait();

    return 0;
}
```

Resource acquisitions are forced to occur in the following order;

- Thread one acquires a lock on “resource1”
- Thread two acquire a lock on “resource2”
- Thread one blocks trying to acquire “resource2”
- Thread two blocks trying to acquire “resource1”

This order eventually leads to a deadlock situation where thread one is waiting for “`resource2`” and can’t get it because thread two has it and thread two wants “`resource1`” but can’t get it because thread one has it. Both threads end up waiting for each other forever. Notice that when we created the `ACE_Local_Mutex` we specified that we did not want deadlock detection to be ignored and wanted debugging on. This causes ACE to detect the deadlock and when it occurs the `acquire()` call on the local mutex fails setting errno to `EDEADLOCK`, enabling debugging causes the following output to be displayed when the program is run:

```
(3224) acquired resource1
(1192) acquired resource2
(3224) waiting for resource2, owner is /USYYID/612/1192, total wait-
ers == 2
(1192) Deadlock detected.
/USYYID/612/1192 owns resource2 and is waiting for resource1.
/USYYID/612/3224 owns resource1 and is waiting for resource2.
```

Note that although the Token framework can detect deadlock it cannot prevent it, an appropriate prevention scheme must be devised by the application programmer, this could for example include a try and back off strategy, i.e., if deadlock is detected back off releasing all your own locks and start acquiring them all over again.

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 example 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 their own sections.

You will find many instances where you want to make sure 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 vast array of primitives that are specifically designed for these purposes. We list all of them in Table **1.2**.

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`	Events are simple synchronization objects that are 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 zero (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 then 0. This happens when some other thread calls release on the semaphore. A semaphore count *will never be negative.* When using a semaphore you initialize the semaphore count to some positive 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 inbuilt `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 or in other words we implement a high water mark mechanism (the message queue already comes inbuilt with this functionality, we essentially reproduce it in an inefficient way for the sake of the example). To make things interesting the consumer task has multiple threads.

The `Producer` produces `ACE_Message_Block`'s that it enqueues on the consumers task's message queue. Each block contains an integer identifier as the only payload. When the producer is done producing it sends a hangup message to the consumer making it shutdown.

Before the producer can produce any items it must acquire the producer semaphore, `psema_`, that represents the maximum capacity of the consumers message queue, or the high water mark. The initial count value for `psema_` is set to the high water mark to begin with. Each time psema_ is acquire()'d this number is atomically decremented. Finally when this count becomes zero the producer cannot enqueue more elements on the consumers queue.

```
class Producer:
    public ACE_Task_Base
{
public:
    enum { MAX_PROD = 128 };

    Producer(ACE_Semaphore& psema, ACE_Semaphore& csema, Consumer
&consumer)
        :psema_(psema), csema_(csema), consumer_(consumer)
    {
    }

    int svc()
    {
        for(int i=0; i <= MAX_PROD; i++)
            produce_item(i);

        hang_up();

        return 0;
```

```
    }

    void produce_item(int item)
    {
        psema_.acquire();

        ACE_Message_Block *mb
            = new ACE_Message_Block( sizeof (int),
                      ACE_Message_Block::MB_DATA);
        ACE_OS::memcpy(mb->wr_ptr(), &item, sizeof item);
        mb->wr_ptr(sizeof (int) );

        this->consumer_.putq(mb);

        ACE_DEBUG((LM_DEBUG, "(%t) Produced %d\n",
                              item));

        csema_.release();
    }

    void hang_up()
    {
        psema_.acquire();

        ACE_Message_Block *mb
            = new ACE_Message_Block(0, ACE_Message_Block::MB_HANGU
P);

        this->consumer_.putq(mb);

        csema_.release();
    }

private:
    ACE_Semaphore& psema_;
    ACE_Semaphore& csema_;
    Consumer& consumer_;
};
```

The `csema_` semaphore on the other hand is initialized to zero in the beginning, its value is incremented (by calling `csema_.release()`) by the producer every time it produces a new element,. If the consumer threads are blocked on an `acquire()` of the csema_ variable one will wake up each time the producer calls `release()`, causing it to consume the new element

Once a consumer thread unblocks it consumes the element from the queue and then calls `release()` on `psema_`, the producer semaphore, as the consumption has freed up a space in the shared message queue. This increments the `psema_` count by 1 allowing the producer to continue and use up the free space in the queue.

```
class Consumer:
    public ACE_Task<ACE_MT_SYNCH>
{
public:
    enum { N_THREADS = 5};

    Consumer(ACE_Semaphore& psema, ACE_Semaphore& csema)
        :psema_(psema), csema_(csema), exit_condition_(0)
    {
    }

    int svc()
    {
        while(!is_closed())
            consume_item();

        return 0;
    }

    void consume_item()
    {
        csema_.acquire();

        if(!is_closed())
        {
            ACE_Message_Block *mb;
            this->getq(mb);

            if(mb->msg_type() == ACE_Message_Block::MB_HANGUP)
            {
                shutdown();
                mb->release();
                return;
            }
            else
            {
                ACE_DEBUG((LM_DEBUG,
                    "(%t) Consumed %d\n",
                        *((int*)mb->rd_ptr())));
```

```
            }

            mb->release();

            psema_.release();
        }

    }

    void shutdown()
    {
        exit_condition_ = 1;
        this->msg_queue()->deactivate();
        csema_.release(N_THREADS);
    }

    int is_closed()
    {
        return exit_condition_;
    }

private:
    ACE_Semaphore& psema_;
    ACE_Semaphore& csema_;
    int exit_condition_;
};
```

Shut down of the consumer task is a little complicated because of the multiple threads that are running in it. Since the producer only sends a single hangup message to the consumer task, only one thread will actually receive the hangup, the others are either blocked waiting for messages on the queue or are blocked on the consumer semaphore. Therefore the consumer thread that receives the hangup must mark the state of the task as closed and then wake up all the other threads so that they notice the closed condition and exit. The consumers shutdown() method does this by first setting the exit_condition_ to true and by first waking up all the threads on the message queue by deactivate() ing it and then decrementing the semaphore count by the number of consumer threads (thereby releasing all the threads waiting on the sempahore).

```
int main(int argc, char *argv[])
{
  ACE_UNUSED_ARG(argc);
  ACE_UNUSED_ARG(argv);
```

```
  ACE_Semaphore psem(5);
  ACE_Semaphore csem(0);

  Consumer consumer(psem, csem);
  Producer producer(psem, csem, consumer);


  producer.activate();
  consumer.activate(THR_NEW_LWP| THR_JOINABLE,
      Consumer::N_THREADS);


  producer.wait();
  consumer.wait();

  return 0;
}
```

The code here is pretty much boiler plate, except for the initialization of the two semaphores `psem` and `csem`. Once again notice that `psem` is initialized to the max number of allowable elements on the queue (i.e., 5) indicating that the producer can produce till the queue is full. On the other hand `csem` is initialized to 0 indicating that the consumer can't consume at all until the producer calls release on `csem`.

14.2.2 Using Barriers

Barriers have a role very similar to their name, a group of threads can use a barrier to collectively synchronize with each other. In essence each thread calls wait on the barrier when it has reached some well known state and then blocks waiting for all other participating threads to call wait indicating that they too have reached the mutual state. Once all the threads have reached the barrier all threads unblock and continue together.

The ACE barrier component is ACE_Barrier. The consturctor of this class takes a count of the number of threads that are synchronizing with the barrier as its first argument. The following example illustrates how barriers can be used to ensure that threads in our home automation commnd handler task startup and shutdown together. To achieve this we start off by creating two barriers, a startup_barrier and a shutdown_barrier. Each barrier is passed a count of the

number of threads running in the handler task. The barriers are then passed to the handler by reference.

```
int main(int argc, char *argv[])
{
  ACE_Barrier startup_barrier(HA_CommandHandler::N_THREADS);
  ACE_Barrier shutdown_barrier(HA_CommandHandler::N_THREADS);

  HA_CommandHandler handler(startup_barrier, shutdown_barrier);
  handler.activate(THR_NEW_LWP | THR_JOINABLE, HA_CommandHandler::
N_THREADS);

  handler.wait();

  return 0;
}
```

When a new handler thread is created it enters the `svc()` method and initializes itself by calling `initialize_handler()`, after which it calls `wait()` on the startup barrier. This blocks the thread on the barrier until all the other handler threads initialize themselves and also call `wait()` on the startup barrier. After completing startup the handler begins handling command requests, by calling `handle_command_requests()`. When the handler recieves the command to shut down the `handle_command_requests()` returns –1 and the threads block on the shutdown barrier. Once all threads reach this barrier they all exit together.

```
class HA_CommandHandler
    :public ACE_Task<ACE_MT_SYNCH>
{
public:
    enum { N_THREADS = 5 };

    HA_CommandHandler(
        ACE_Barrier& startup_barrier,
        ACE_Barrier &shutdown_barrier)
        :startup_barrier_(startup_barrier),
        shutdown_barrier_(shutdown_barrier)
    {
    }

    void initialize_handler();
    int handle_command_requests();

    int svc()
```

```
        {
            initialize_handler();

            startup_barrier_.wait();
            ACE_DEBUG((LM_DEBUG,
                    "(%t: %D) Started\n"));

            while(handle_command_requests() > 0);

            shutdown_barrier_.wait();
            ACE_DEBUG((LM_DEBUG,
                    "(%t: %D) Ended\n"));

            return 0;
        }
    private:
        ACE_Barrier& startup_barrier_;
        ACE_Barrier& shutdown_barrier_;
    };
```

For illustrative purposes we setup the `initialize_handler()` method and `handle_command_requests()` method to sleep for random time periods, thus ensuring each handler thread takes a different time period to initalize and to shut down. The barriers will ensure that they all startup and exit at the same time, we display the times (startup and shutdown) for each thread by using the `%D` current time stamp format specifier for the `ACE_DEBUG` macro. The resultant output shows that startup and shutdown times are indeed in synch for all threads;

```
(764: 08/12/2001 23.40.11.214000) Started
(720: 08/12/2001 23.40.11.214000) Started
(1108: 08/12/2001 23.40.11.214000) Started
(1168: 08/12/2001 23.40.11.214000) Started
(1080: 08/12/2001 23.40.11.214000) Started
(1168: 08/12/2001 23.40.11.334000) Ended
(720: 08/12/2001 23.40.11.334000) Ended
(764: 08/12/2001 23.40.11.334000) Ended
(1108: 08/12/2001 23.40.11.334000) Ended
(1080: 08/12/2001 23.40.11.334000) Ended
```

14.3 Thread Specific Storage

Unlike a process, 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. Such storage is called *Thread Specific* 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 multi-threaded programs first made their appearance 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 the `errno` in thread specific storage solved this. Logically the variable was global, but it was actually kept in TSS.

Thread specific storage should be used for all objects that you consider to part and parcel of a thread. ACE internally uses TSS 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 light weight.

The `ACE_Thread` class provides access to the low level OS TSS methods but in most cases you can avoid these tedious API's by using the `ACE_TSS<T>` template class. This class is very simple to use. All you must do is to pass it the data you want to be stored in TSS as its template parameter and use the `operator->()` method to access the data. 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 gt 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 that can hold any arbitrary, named, tidbits of information using an internal attribute map.

```
class ClientContext
{
public:
    void *get_attribute(const char*name);

    void set_attribute(const char*name,
                       void *value);
private:
        Map attributeMap_;
};
```

To stores this object in TSS all you need to do is wrap it within the ACE_TSS<T> template. The first access causes an instance of the type T 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()
        {
                ACE_thread_t tid =
                        this->thr_mgr()->thr_self();
                this->tss_ctx_->
                        set_attribute("thread_id", &tid);
                //set out identifier in TSS

                while(handle_requests() > 0);

                return 0;

        }

        int handle_requests()
        {
                ACE_thread_t *tid =
                        (ACE_thread_t*)this->tss_ctx_->
                           get_attribute("thread_id");

                ACE_DEBUG((LM_DEBUG, "(%t) TSS TID: %d \n",
                        *tid));
```

```
                //do work.
                return -1;
        }
private:
        ACE_TSS<ClientContext> tss_ctx_;
};
```

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 went over recursive mutexes lock using the `ACE_Recursive_Mutex/ ACE_Recursive_Thread_Mutex` classes, readers/writers locks with `ACE_RW_Mutex/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 sempahores 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 co-operate. We have seen various illustrations of thread cooperation in previous chapters, this included cooperation using condition variables, semaphores and message queues.

The ActiveObject Pattern has been specifically designed to provide an object-based solution to co-operative 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 and not in the context 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 may have methods that take a long time to execute. In this case 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 will go over the Active Object pattern and will illustrate how you can make this work in your applications. We will introduce several new components that are used to implement the pattern, these components will also come in handy when we talk about thread pools in the next chapter.

Main Thread

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

this->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.

 ..
}
```

Figure 15.1. Asynchronous log() Method

15.1 The Pattern

Let's start off by looking at Figure 15.1 which illustrates the behavior of an active Logger object.

Here the log() method is executed in the context of the logger thread and not in the context of the main thread. This of course introduces asynchronous method calls into the application. When the log() method returns in the main thread there is no guarantee that it has actually executed. Thus you can consider the Logger object as being *active*, i.e., when a client thread executes methods on the Logger the methods are executed by the private logger thread and not the main thread.

15.1.1 Pattern Participants

The Active Object Pattern is actually a combination of the Command pattern and the Proxy pattern. The participant objects in the pattern are illustrated in Figure 15.2 and listed in the table below:

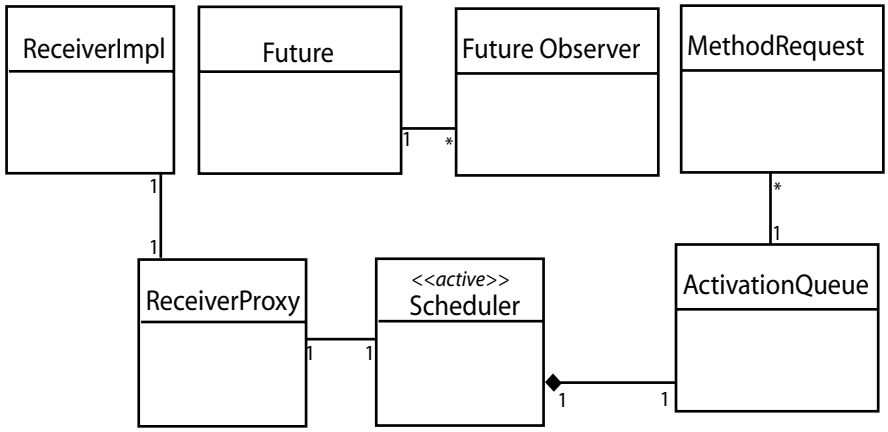


Figure 15.2. Active Object Pattern Participants

- Receiver Implementation: The actual receiever implementation. Note that this 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 Object: Proxy to the Receiver Object that will be used by the client to invoke requests. It is the responsibility of this class to convert method invocations to method requests and insert them on the activation queue. The Proxy 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 RecieverImpl the method request will need to have a reference to the Reciever Impl.
- 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.
- Future: A token returned to the client that can be redeemed at a later time to obtain the result of the asynchronous operation

- Future Observer: A callback object (Command Pattern) that is associated with a future and is automatically called when the return value of the asynchronous call is set in the future object, i.e., is called when the asynchronous operation has completed.

15.1.2 Collaborations

The collaborations between the various participants are illustrated by a sequence diagram in Figure 15.3.

When a client thread wishes to execute a method on the receiver object it first obtains a reference to the receiver's proxy object. Once it has a reference it can invoke any of the active methods on the receiver. In the diagram the client invokes active_operation() on the proxy.

The proxy then creates the corresponding concrete method request object, named ActiveOp and enqueues it on the activation queue. Note that the Scheduler, which encapsulates the private thread of the active object, has already issued a blocking dequeue() call on the activation queue and is waiting for method requests to arrive on the queue.

When the proxy enqueues the method request the scheduler thread wakes up, dequeues the method request object and invokes the call() method on it, which in turns invokes active_operation() on the ReceiverImpl object.

Note that to keep things simple the sequence diagram did not talk about futures. If active_operation() does not have a return value, you may not need to use futures. When the client invokes active_operation() on the proxy it immediately returns a future object to the client. The client can either block or poll on the future waiting for a result. Alternatively, you can attach a future observer to the future object. In the later case the observer is automatically called back (by the private thread of control owned by the active object) when the asynchronous operation completes and the result is available.

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

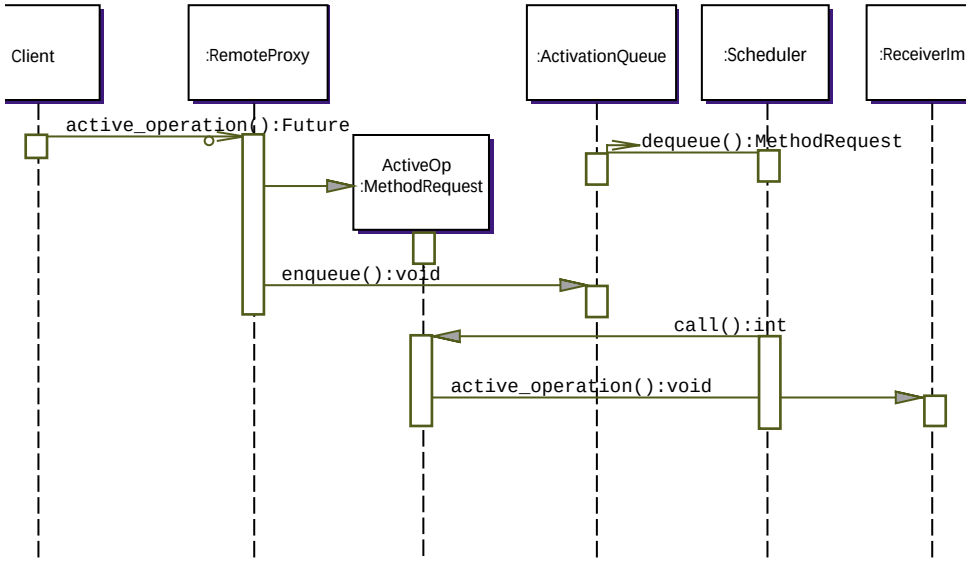


Figure 15.3. Active Object Collaborations

other network details providing clients with an easy to use representation of the controller (once again the proxy pattern).

```
class HA_ControllerAgent
{
  //A proxy to the real HA_Controller that is
  //on the network
public:
  HA_ControllerAgent()
  {
    ACE_TRACE("HA_ControllerAgent::HA_ControllerAgent");
    status_result_ = 1;
  }

  int status_update()
  {
    ACE_TRACE("HA_ControllerAgent::status_update");

    ACE_DEBUG((LM_DEBUG,
        "Obtaining a status_update in %t thread of control\n"));
```

```
    ACE_OS::sleep(2);
    return next_result_id();
    //simulate sending message to controller
    //and getting status
  }

private:
  int next_result_id()
  {
    ACE_TRACE("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. Notice 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 it's methods.

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

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

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

    return 0;
  }
```

```
private:
  HA_ControllerAgent& controller_;
  ACE_Future<int> returnVal_;
};
```

The StatusUpdate class is a method request Command object that encapsulates the activation record that is needed to perform a deferred call to the agent. As you will see in a second, the method request object is created when the client of the active object makes a request on the proxy. At a later point the active object obtains the method request and issues call() on it. This in turn causes the method request object to invoke the right method on the implementation object of the active object.

In this case, when a client uses a HA_ControllerAgentProxy to obtain status he causes the creation of a StatusUpdate object that keeps a reference to the real HA_ControllerAgent implementation object and a reference to the future object. Later, the scheduler will pick up the StatusUpdate object from it's queue in the thread of control of the active object and will issue call() on it. This invokes status_update() on the real implementation object.

```
class ExitMethod
  : public ACE_Method_Request
{
public:
  virtual int call()
  {
    return -1;
    //cause exit
  }
};
```

You will need to create one method request object for each method of your active object. To keep thing simple for the sake our our example we only show the HA_ControllerAgent as having two methods (which is not very realistic). The first is the status_update() method and the second is exit(). The ExitMethod class is the method request object that is used when the client programmer calls exit() on the HA_ControllerAgentProxy. Notice that in this case the call method just returns -1. We will see how we use this to cause the active object to shut down.

```
class Scheduler
  : public ACE_Task_Base
{
public:
```

```
  Scheduler()
  {
    ACE_TRACE("Scheduler::Scheduler");

    this->activate();
  }

  virtual int svc()
  {
    ACE_TRACE("Scheduler::svc");

    while(1)
      {
        // Dequeue the next method object
        auto_ptr<ACE_Method_Request>
          request(this->activation_queue_.dequeue ());
        // Dequeue the next method object

        if(request->call() == -1)
          break;
        //invoke the method request
      }

    return 0;
  }

  int enqueue(ACE_Method_Request* request)
  {
    ACE_TRACE("Scheduler::enqueue");

    return this->activation_queue_.enqueue(request);
  }
private:
  ACE_Activation_Queue activation_queue_;
};
```

The Scheduler is the heart of the active object. It is this class that “holds” the thread of control. This Scheduler derives from ACE_Task_Base that is very similar in nature to the SimpleThread class that we have been using in our previous examples. This class offers an activate() method that when called causes a new thread of control to start in the svc() method, much like a new thread of control would start in the start() method of classes that were derived from SimpleThread.

The Scheduler keeps an underlying activation queue on which method request objects are enqueued when the client requests service. The Scheduler uses it's thread to dequeue method objects from it's activation queue and then invoke the call() method on these objects. Thus whatever happens in call() happens in the context of the Scheduler's thread and not in the context of the client thread.

Also, notice that our Scheduler has been built such that if the call() method returns –1 the active object thread gracefully exits. This is exactly what happens if an ExitMethod object is placed on the activation queue by the client. Now you probably understand why we just returned –1 from the call method in the Exit-Method class, neat huh?

```
class HA_ControllerAgentProxy
{
  //This acts as a proxy to the
  //controller impl object.
public:
  ACE_Future<int> status_update()
  {
    ACE_TRACE("HA_ControllerAgentProxy::status_update");
    ACE_Future<int> result;

    this->scheduler_.
      enqueue(new StatusUpdate(this->controller_,result));
    //create and enqueue a method request
    //on the scheduler

    return result;
    //return future to the client
  }

  void exit()
  {
     ACE_TRACE("HA_ControllerAgentProxy::exit");

     this->scheduler_.
        enqueue(new ExitMethod());
  }

private:
  Scheduler scheduler_;
  HA_ControllerAgent controller_;
};
```

Finally we get to the proxy that is used by the client. The HA_ControllerAgentProxy aggregates both the scheduler and the agent implementation. The constructor of the Scheduler causes the active object going using the activate method from ACE_Task_Base.

When a client issues a request on the proxy all it does is create the corresponding method request object and pass it along to the scheduler. It is the scheduler's responsibility to use the method object and then destroy it (this is done using the C++ auto_ptr<> template class in this example).

In the case of the status_update() method the client is returned a future object. The future object holds a reference to the result that will be returned to the client. Next let's look at how the client can redeem the future for it's actual value.

```
int main(int argc, char *argv[])
{
  ACE_UNUSED_ARG(argc);
  ACE_UNUSED_ARG(argv);

  HA_ControllerAgentProxy controller;
  ACE_Future<int> results[10];

  for(int i=0 ; i < 10; i++)
    {
      ACE_DEBUG((LM_DEBUG, "%t calling status_update\n"));
      results[i] = controller.status_update();
    }

  ACE_OS::sleep(5);
  //do other work

  for(int j=0; j < 10; j++)
    {
      int result;
          results[j].get(result);

       ACE_DEBUG((LM_DEBUG,
                  "New status_update %d\n", result ));
    }
  //Get results..

  controller.exit();
  //cause the status_updateer threads to exit

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

```
  //wait for threads to exit

  return 0;
}
```

Here the client of the active agent object, uses it's status_update() method to issue requests to get the current status of the controller. The agent returns futures to the client which the client redeem's using the get() method of the future object. If the agent has not finished obtaining the status (which is a distinct possibility in our example) then the client will block on the get() method of the future until the results are obtained from the remote controller.

The ACE_Future class also you to do a timed waits for results that are returned using futures. You may need this feature to ensure that your clients do not block on lower priority methods.

15.2.1 Using Observers

Instead of blocking on the future and waiting for result's the client can attach a concrete ACE_Future_Observer to the ACE_Future returned by the proxy. To create a concrete observer just sub-class ACE_Future_Observer and implement the update() method. The update method takes a single argument, an ACE_Future<> class. Since our status_update() method returns an int the template is instantiated as ACE_Future<int> and the observer as ACE_Future_Observer<int>

```
class CompletionCallBack:
public ACE_Future_Observer<int>
{
public:
    CompletionCallBack(HA_ControllerAgentProxy& proxy)
        :proxy_(proxy)
    {
    }
    virtual void update(const ACE_Future<int> & future)
    {
        int result;
        ((ACE_Future<int>)future).get(result);

        ACE_DEBUG((LM_INFO,
            "(%t) New Status %d\n", result));
        if(result == 10)
            this->proxy_.exit();
```

```
    }
private:
    HA_ControllerAgentProxy& proxy_;
};
```

To attach the observer to a future all you need to do is call attach() on the future object when it is returned by the proxy.

```
int main(int argc, char *argv[])
{
  ACE_UNUSED_ARG(argc);
  ACE_UNUSED_ARG(argv);

  HA_ControllerAgentProxy controller;
  ACE_Future<int> results[10];
  CompletionCallBack cb(controller);

  for(int i=0 ; i < 10; i++)
    {
      ACE_DEBUG((LM_DEBUG, "(%t) calling status update\n"));
      results[i] = controller.status_update();
      results[i].attach(&cb);
    }

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

  return 0;
}
```

This makes the code a bit more simpler, now when a status_update() method completes we are assured that the observer's update() method is automatically called. This free's up the client thread of control to do other things instead of just waiting for the call to complete.

15.3 Summary

In this chapter we covered the active object pattern and all it's 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 which you wish to run in a seperate thread of control. We also saw how you can obtain the result of asynchronous operations, i.e., 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 will re-use the activation queue, future and task objects 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 process or multiple threads in our servers. When building a multi threaded server there are many different design options that are available including;

- spawning a new thread for each request
- spawning a new thread for each connection/session
- pre-spawning a managed pool of threads, i.e., creating a thread pool.

In this chapter we are going to explore this last option, i.e, Thread Pools in some detail. We will start by exploring what we mean by a thread pool and what advantages it provides over some of the other approaches to building multi-threaded servers. We then go into some detail about various different types of thread pools and their performance characteristics, illustrating a few types with examples pool that are built using the ACE components. Finally we will 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, instead of creating a new thread for each session or request and then destroying it, we pre-spawn 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, where a new thread is created for each request, if the server receives a large spurt of requests in a short period of time it 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.

There are actually several variants of the thread pool model, each with different performance characteristics;

- Half Sync/ Half Async model; in this model a single listener thread waits for requests to arrive and buffer them in a queue. This thread then hands off the work to a worker thread out of a separate set of worker threads.
- Leader/Followers model; in this model a single thread is elected as a leader and the rest are followers in the thread pool. When a request arrives the leader picks it up, elects 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

The Half Sync/Half Async model breaks the thread pool into three separate layers, the asynchronous layer which receives the asynchronous requests, the queueing layer which buffers the requests and the synchronous layer which contains several threads of control blocked on the queueing layer. When the layer indicates there is new data the synchronous layer handles the request synchronously in a separate thread of control.

There are several advantages to this model including

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

There are also some disadvantages to this model;

- Because a thread switch occurs at the queueing layer we must incur synchronization and context switch overhead, furthermore on a multi processor machine we may experience data copying and cache coherency overhead.
- We cannot keep any request information on the stack or in TSS as the request is actually processed in a different worker thread.

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

First lets take a 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("Manager::Manager");
    }

    int svc()
    {
        ACE_TRACE("Manager::svc");

        ACE_DEBUG((LM_INFO,
            "(%t) Manager started\n"));
```

```
    create_worker_pool();
    //create pool

    while(!done())
    {
        ACE_Message_Block *mb = NULL;
        ACE_Time_Value tv((long)MAX_TIMEOUT);
        tv += ACE_OS::time(0);
        if(this->getq(mb, &tv)  <0)
        {
            shut_down();
            break;
        }
        //get a message request.

        Worker* worker;
        {
            ACE_GUARD_RETURN(ACE_Thread_Mutex,
                worker_mon, this->workers_lock_, -1)

            while(this->workers_.is_empty())
                workers_cond_.wait();

            this->workers_.dequeue_head(worker);
        }
        //choose a worker.

        worker->putq(mb);
        //ask him to do the job.
    }

    return 0;
}

int shut_down();

int thread_id(Worker * worker);

virtual int return_to_work(Worker* worker)
{
    ACE_GUARD_RETURN(ACE_Thread_Mutex,
        worker_mon, this->workers_lock_, -1);
    ACE_DEBUG((LM_DEBUG,
        "(%t) Worker %d returning to work.\n",
        worker->thr_mgr()->thr_self()));
    this->workers_.enqueue_tail(worker);
```

```
        this->workers_cond_.signal();

        return 0;
    }

private:
    int create_worker_pool()
    {
        ACE_GUARD_RETURN(ACE_Thread_Mutex,
            worker_mon, this->workers_lock_, -1);
        for(int i=0; i < POOL_SIZE; i++)
        {
            Worker* worker;
            ACE_NEW_RETURN(worker, Worker(this), -1);
            this->workers_.enqueue_tail(worker);
            worker->activate();
        }

        return 0;
    }

    int done();

private:
    int shutdown_;
    ACE_Thread_Mutex workers_lock_;
    ACE_Condition<ACE_Thread_Mutex> workers_cond_;
    ACE_Unbounded_Queue<Worker* > workers_;
};
```

First of all note that the Manager has an ACE_Unbounded_Queue<> of Worker objects. This represents the worker thread pool. This thread pool is protected by the workers_lock_ mutex. When the Manager thread start's in it's svc() method, it first creates the underlying worker thread pool and then blocks on it's underlying message queue waiting for requests.

When a request does arrive it dequeues the request and then dequeue's the first worker off the worker pool and enqueues the request onto it's message queue. Notice that if there are no workers in the thread pool (perhaps because they are all busy processing messages) then we block on the worker_cond_ condition variable waiting for a thread to return to work. When a thread does return to work, it enqueues itself back on the worker queue and notifies the Manager by signaling

the condition variable. The Manager thread then wakes up and hands the new request off to the worker thread that has just returned to the pool.

```
class Worker: public ACE_Task<ACE_MT_SYNCH>
{
public:
    Worker(IManager* manager)
        :manager_(manager)
    {
    }

    virtual int svc()
    {
        thread_id_ = ACE_Thread::self();
        while(1)
        {
            ACE_Message_Block *mb = NULL;
            ACE_ASSERT(this->getq(mb)!=-1);
            if(mb->msg_type() ==
                ACE_Message_Block::MB_HANGUP)
            {
                ACE_DEBUG((LM_INFO,
                    "(%t) Got hangup shutting down\n"));
                break;
            }

            process_message(mb);
            //process the msg.

            this->manager_->return_to_work(this);
            //return to work.

        }

        return 0;
    }
```

The Worker Thread sits in an infinite loop waiting for a request to arrive on it's queue. When a request does arrive it picks it up and then processes it. Once the processing completes it returns back to it's manager, where it is once again enqueued in the worker thread pool.

16.2.1 Using an Activation Queue and Futures

In the last example we kept things simple by using simple ACE_Message_Block messages as the unit of work. We also assumed that the client thread that enqueues the work on the Manager's queue did not require any results. Although both of these conditions are often true there are situations where a more object oriented solution using method requests as the unit of work and futures as results is more appropriate.

In this next example we are going to add these features to our previous example. Let's begin with a new method request object that expects to return a string as the result of the operation.

```
class LongWork:
    public ACE_Method_Request
{
public:
    virtual int call()
    {
        ACE_TRACE(("LongWork::call"));
        ACE_DEBUG((LM_INFO,
            "(%t) Attempting long work task\n"));
        ACE_OS::sleep(1);

        char buf[1024];
        ::sprintf(buf, "(%d) Completed assigned task\n",
            ACE_Thread::self());

        ACE_CString* msg;
        ACE_NEW_RETURN(msg,
            ACE_CString(buf, ACE_OS::strlen(buf)+1),
            -1);
        result_.set(msg);

        return 0;
    }

    ACE_Future<ACE_CString*>& future()
    {
        ACE_TRACE("LongWork::future");
        return result_;
    }

    void attach(CompletionCallBack* cb)
    {
        result_.attach(cb);
```

```
    }

private:
    ACE_Future<ACE_CString*> result_;
};
```

Next, lets add a future observer that will automatically get called back when the result from the LongWork operation is available.

```
class CompletionCallBack:
    public ACE_Future_Observer<ACE_CString*>
{
public:
    virtual void update
        (const ACE_Future<ACE_CString*> & future)
    {
        ACE_CString* result;
        ((ACE_Future<ACE_CString*>)future).
            get(result);
        //block for the result.

        ACE_DEBUG((LM_INFO,
            "%s\n", result->c_str()));
        delete result;
    }
};
```

The Manager and Worker both need to be modified so that they use an ACE_Activation_Queue instead of an ACE_Message_Queue. Also, since neither Worker or Manager will be using the underlying message queue in ACE_Task<> they both will inherit from the lighter ACE_Task_Base class that does not include the message queue.

```
class Worker: public ACE_Task_Base
{
public:
    Worker(IManager* manager)
        :manager_(manager)
    {
    }

    int perform(ACE_Method_Request* req)
    {
        ACE_TRACE("Worker::perform");
```

```
        return this->queue_.enqueue(req);
    }

    virtual int svc()
    {
        thread_id_ = ACE_Thread::self();
        while(1)
        {
            ACE_Method_Request* request =
                this->queue_.dequeue();
            if(request == 0)
                return -1;
            int result
                = request->call();
            //invoke the request

            if(result == -1)
                break;

            this->manager_->return_to_work(this);
            //return to work.

        }

        return 0;
    }

    ACE_thread_t thread_id();

private:
    IManager* manager_;
    ACE_thread_t thread_id_;
    ACE_Activation_Queue queue_;
};
```

The worker thread has not changed much, except now instead of dequeueing an ACE_Message_Block the thread dequeues and then immediately invokes the method request object. Once the method request has returned the worker thread returns back to the worker thread pool that is held by the Manager.

```
class Manager: public ACE_Task_Base, IManager
{
public:
    enum {POOL_SIZE = 5, MAX_TIMEOUT = 5};
```

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

int perform(ACE_Method_Request *req)
{
    ACE_TRACE("Manager::perform");
    return this->queue_.enqueue(req);
}

int svc()
{
    ACE_TRACE("Manager::svc");

    ACE_DEBUG((LM_INFO, "(%t) Manager started\n"));

    create_worker_pool();
    //create pool when you get in the first time.

    while(!done())
    {
        ACE_Time_Value tv((long)MAX_TIMEOUT);
        tv += ACE_OS::time(0);
        ACE_Method_Request *request
            = this->queue_.dequeue(&tv);
        //get the next message
        if(request == 0)
        {
            shut_down();
            break;
        }
        //get a message request.

        Worker* worker
            = choose_worker();
        //choose a worker.

        worker->perform(request);
        //ask him to do the job.
    }

    return 0;
```

```
    }


    int shut_down();

    virtual int return_to_work(Worker* worker)
    {
        ACE_GUARD_RETURN(ACE_Thread_Mutex, worker_mon, this->worke
rs_lock_, -1);
        ACE_DEBUG((LM_DEBUG, "(%t) Worker %d returning to work.\n"
,
            ACE_Thread::self()));

        this->workers_.enqueue_tail(worker);
        this->workers_cond_.signal();

        return 0;
    }

private:
    Worker* choose_worker()
    {
        ACE_GUARD_RETURN(ACE_Thread_Mutex, worker_mon, this->worke
rs_lock_, 0)

        while(this->workers_.is_empty())
            workers_cond_.wait();

        Worker* worker;
        this->workers_.dequeue_head(worker);

        return worker;
    }

    int create_worker_pool()
    {
        ACE_GUARD_RETURN(ACE_Thread_Mutex, worker_mon, this->worke
rs_lock_, -1);
        for(int i=0; i < POOL_SIZE; i++)
        {
            Worker* worker;
            ACE_NEW_RETURN(worker, Worker(this), -1);
            this->workers_.enqueue_tail(worker);
            worker->activate();
        }
```

```
        return 0;
    }

    int done()
    {
        return (shutdown_ == 1);
    }

    int thread_id(Worker * worker)
    {
        return worker->thread_id();
    }

private:
    int shutdown_;
    ACE_Thread_Mutex workers_lock_;
    ACE_Condition<ACE_Thread_Mutex> workers_cond_;
    ACE_Unbounded_Queue<Worker* > workers_;
    ACE_Activation_Queue queue_;
};
```

The Manager class is also almost the same, the only difference being the use of the activation queue instead of the underlying message queue. The Manager also has a new public method perform() that can be used by clients to enqueue method requests onto the Manager's activation queue.

```
int main(int argc, char*argv[])
{
    ACE_UNUSED_ARG(argc);
    ACE_UNUSED_ARG(argv);

    Manager tp;
    tp.activate();

    ACE_Time_Value tv;
    tv.msec(100);
    //wait for a few seconds every time you send a message..
    CompletionCallBack cb;
    LongWork workArray[OUTSTANDING_REQUESTS];
    for(int i=0; i < OUTSTANDING_REQUESTS; i++)
    {
        workArray[i].attach(&cb);
        ACE_OS::sleep(tv);
        tp.perform(&workArray[i]);
    }
```

```
#if 0
    LongWork workArray[OUTSTANDING_REQUESTS];
    for(int i=0; i < OUTSTANDING_REQUESTS; i++)
    {
        ACE_OS::sleep(tv);
        tp.perform(&workArray[i]);
    }

    for(int j=0; j < OUTSTANDING_REQUESTS; j++)
    {
        ACE_CString* result;
        workArray[j].future().get(result);
        //block for the result.

        ACE_DEBUG((LM_INFO, "%s\n", result->c_str()));

        delete result;
    }
#endif

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

    return 0;
}
```

16.3 Leader/Followers Model

In the Leader/Followers Model a single group of thread is used to wait for new requests and to handle the request. In this model 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, then it promotes one of the followers to leader status and finally goes on to process the request that 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.

There are several advantages to the leader/followers model;

- performance improves as their is no context switch between threads. This also allows for keeping request data on the stack or in thread specific storage.

There are also some disadvantages to using this model;

- Not easy to handle bursty clients since there might not be an explicit queuing layer
- More complex to implement

Lets take a look at a simple example which implements the leader follower model thread pool. In this case there is a single ACE_Task<> (the LF_ThreadPool class) which encapsulates all the threads in the thread pool. Remember, there can only be one leader at any given time, the thread id of the current leader of the pool is maintained in current_leader_. Like our first example the LF_ThreadPool is given new work as an ACE_Message_Block on it's message queue, i.e., a unit of work is a message.

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

    virtual int svc();

    void shut_down()
    {
        shutdown_ = 1;
    }


private:

    int become_leader();

    Follower* make_follower();

    int elect_new_leader();

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

```
    }

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

    void process_message(ACE_Message_Block*mb);

    int done()
    {
        return (shutdown_ == 1);
    }

private:
    int shutdown_;
    ACE_thread_t current_leader_;
    ACE_Thread_Mutex leader_lock_;
    ACE_Unbounded_Queue<Follower*> followers_;
    ACE_Thread_Mutex followers_lock_;
    static long LONG_TIME;
};
```

Let’s take a look at the svc() method for the LF_ThreadPool.

```
int
LF_ThreadPool::svc()
{
    ACE_TRACE("LF_ThreadPool::svc");

    while(!done())
    {
        become_leader();
        //block until you are indeed a leader.

        ACE_Message_Block *mb = NULL;
        ACE_Time_Value tv(LONG_TIME);
        tv += ACE_OS::time(0);

        if(this->getq(mb, &tv) < 0)
        {
            if(elect_new_leader() == 0)
                break;
            continue;
```

```
        }
        //get a message.

        elect_new_leader();
        //elect a new leader.

        //do the blocking I/O call.
        process_message(mb);
    }

    return 0;
}
```

As each thread start's it first tries to become a leader by calling become_leader(). If the thread can't become a leader it blocks in the become_leader() method and will only return once it has indeed been elected as leader. Once a thread has been elected as the leader it blocks on the message queue until a message arrives. When the leader receives a new message it first elects a new leader by calling elect_new_leader() and then proceeds to process the incoming message. The newly elected leader will wake up from its previous become_leader() call and then block on the message queue waiting for the next request.

```
int
LF_ThreadPool::become_leader()
{
    ACE_TRACE("LF_ThreadPool::become_leader");

    ACE_GUARD_RETURN(ACE_Thread_Mutex, leader_mon, this->leader_lo
ck_, -1);
    if(leader_active())
    {
        Follower * fw
            = make_follower();
        {

            while(leader_active())
                fw->wait();
            //wait until told to do so..
        }

        delete fw;
    }
```

```
    ACE_DEBUG((LM_DEBUG, "(%t) Becoming the leader\n"));
    leader_active(ACE_Thread::self());
    //mark yourself as the active leader.

    return 0;
}

Follower*
LF_ThreadPool::make_follower()
{
    ACE_TRACE("LF_ThreadPool::make_follower");

    ACE_GUARD_RETURN(ACE_Thread_Mutex, follower_mon, this->followe
rs_lock_,0);
    Follower* fw;
    ACE_NEW_RETURN(fw, Follower(this->leader_lock_), 0);
    this->followers_.enqueue_tail(fw);

    return fw;
}
```

When a thread calls become_leader() it first checks to see if there is already a current active leader. If there is, it creates a new Follower object which is enqueued on the followers_ queue. The thread then calls wait on the new follower object. The Follower is a thin wrapper around a condition variable. Once a follower thread wakes up it is the newly elected leader and it returns in the svc() method as described previously.

```
int
LF_ThreadPool::elect_new_leader()
{
    ACE_TRACE("LF_ThreadPool::elect_new_leader");

    ACE_GUARD_RETURN(ACE_Thread_Mutex, leader_mon, this->leader_lo
ck_, -1);
    leader_active(0);

    if(!followers_.is_empty())
    {
        ACE_GUARD_RETURN(ACE_Thread_Mutex, follower_mon, this->fol
lowers_lock_, -1);
        Follower* fw;
        ACE_ASSERT(this->followers_.dequeue_head(fw)==0);
        //get the old follower..
```

```
        ACE_DEBUG((LM_ERROR, "(%t) Resigning and Electing %d\n", f
w->owner()));
        ACE_ASSERT(fw->signal() ==0);

        return 0;
    }
    else
    {
        ACE_DEBUG((LM_ERROR, "(%t) Oops no followers left\n"));
        return -1;
    }
    //wake up a follower
}
```

To elect a new leader the previous leader first changes the current_leader_ value to 0, indicating there is currently no active leader in the pool. It then proceeds to obtain a follower from the followers queue and then waking it up by calling signal() on it. The blocked follower thread then wakes up, notices there is no current leader and marks itself as the current leader.

When the leader thread finishes processing a message it once again calls the become_leader() method and tries to become the leader again. If it can't then it will become a follower and get it enqueued in the followers_ queue.

16.4 Thread Pools and the Reactor

The ACE_Reactor has several different implementations that you have seen previously in this book. Some of these implementations only allow 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 follower model for it's 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 actually waits for events. When an event occurs the leader elects a new leader and dispatches the event handlers that had been signalled. Before electing 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 the leader finishes the dispatch it resumes the handler in the reactor, making it available for dispatch once again.

The following example is a rehash from a test in the ACE directory that illustrates the ACE_TP_Reactor.

```
int
main (int argc, ACE_TCHAR *argv[])
{
  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;
}
```

Lets first start off by looking at how the ACE_TP_Reactor is setup. We create an ACE_TP_Reactor instance and set it up as the implementation class for the reactor by passing it to the constructor of the ACE_Reactor. We also set the global singleton instance of the ACE_Reactor to be the instance we create here. This makes it easy for us to obtain the reactor through the global singleton. After we do this we open an acceptor and activate two tasks, one to represent the server thread pool that will run the reactive event loop (ServerTP) and the other to represent clients that connect to the server (Client).

```
static int
reactor_event_hook (void *)
{
  ACE_DEBUG ((LM_DEBUG,
              "(%t) handling events ....\n"));

  return 0;
}

class ServerTP: public ACE_Task_Base
{
public:
    virtual int svc()
    {
        ACE_DEBUG((LM_DEBUG,
            "(%t) Starting and try to run the event loop\n"));

        int result =
            ACE_Reactor::instance ()->
                run_reactor_event_loop (&reactor_event_hook);

        if (result == -1)
            ACE_ERROR_RETURN ((LM_ERROR,
            "(%t) %p\n",
            "Error handling events"),
            0);

        ACE_DEBUG ((LM_DEBUG,
            "(%t) I am done handling events. Bye, bye\n"));
```

```
        return 0;
    }
};
```

Once the ServerTP is activated multiple threads start up in it's svc() method. Each thread immediately starts the reactor's event loop. Note that you can use any of the reactors event handling methods here. As we described earlier the reactor then takes charge, creating the thread pool, electing the leader and then dispatching events in multiple threads of control.

16.4.2 ACE_WFMO_Reactor

The ACE_WFMO_Reactor is available on the Win32 platform and uses the Win32 WaitForMultipleObjects() call to wait for events. It usage is similar to the ACE_TP_Reactor. There are a few noteworthy differences though

- The WFMO Reactor does **NOT** suspend and resume handlers the way the TP_Reactor does. This means that you have to ensure that state information in your event handlers are protected.
- The WFMO Reactor uses the concept of ownership to choose one thread that will handle time-outs. If none of the threads that are running the event loop are designated as the owner (using the owner() method) time-outs may not be handled properly.
- State changes to the WFMO reactor are not immediate they are delayed until the reactor reaches a "stable state" (under the covers when you make a change the leader thread is informed and this thread makes the state changes).

16.5 Summary

In this chapter we looked at Thread Pools. We first talked about a few different design models for thread pools, including the Leader/Followers and Half Sync/ Half Async model. We then implemented each of these using the ACE_Task<> and synchronization components. We also saw how to use ACE_Future, ACE_Activation_Queue and ACE_Future_Observer to implement pools that can execute arbitrary ACE_Method_Request's and then allow for clients to obtain results using futures. In the end we talked about the two reactor implementations that support thread pools, the ACE_TP_Reactor and the ACE_WFMO_Reactor.

Part III

Advanced ACE Usage

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 only used 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 just this technique to provide persistence for your configuration).

ACE provides several tools to assist in your shared memory adventures. For sharing memory between applications, it 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 we 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 that 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/deallocate.

Besides the allocators that we saw earlier, ACE 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 different memory pools, including pools that allocate shared memory and pools that allocate regular heap memory. The pools are detailed in the table below.

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 light weight 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.
`ACE_Local_Memory_Pool`	A memory pool that is based on the C++ new operator.
`ACE_Pagefile_Memory_Pool`	Make a memory pool that is based on “anonymous” memory regions allocated from the Win32 page file
`ACE_Sbrk_Memory_Pool`	Make a memory pool that is based on <sbrk(2)>.

As you can see, several pools are OS-specific and use 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 then you need to use an ACE provided macro for the pool type to instantiate the template. The macro names are provided in the table 17.2.

Table 17.2. Memory Pool macro names.

Class	Macro
ACE_MMAP_Memory_Pool	ACE_MMAP_MEMORY_POOL
ACE_Lite_MMAP_Memory_Pool	ACE_LITE_MMAP_MEMORY_POOL
ACE_Shared_Memory_Pool	ACE_SHARED_MEMORY_POOL
ACE_Local_Memory_Pool	ACE_LOCAL_MEMORY_POOL
ACE_Pagefile_Memory_Pool	ACE_PAGEFILE_MEMORY_POOL
ACE_Sbrk_Memory_Pool	ACE_SBRK_MEMORY_POOL

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 operations like `malloc()`, `calloc()`, and `free()`, `ACE_Malloc` also supports a map like 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 allows you to insert values in the underlying memory pool and associate these values 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, you guessed it, LIFO and FIFO order of insertion. We will see several examples that use this interface in the next few sections.

17.1.2 Memory Protection Interface

In certain cases the underlying shared memory pool allows you to change the memory protection level of individual pages in the memory pool. You could for example make certain pages read only, others read/write or executable. To specify protection, `ACE_Malloc` offers several `protect()` methods that allow you to specify the protection attributes for different regions of its underlying memory pool. Note that protection is only available with certain pools (for example the `ACE_MMAP_Memory_Pool`).

17.1.3 Sync Interface

If your memory pool is backed to a file (such as when using `ACE_MMAP_Memory_Pool`) then you need a way to flush out the mapping file to disk at the appropriate times in your program. To allow this `ACE_Malloc` includes a `sync()` method.

17.2 Persistence with ACE_Malloc

Lets 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`.

First let's start by creating our allocator;

```
#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;
```

We first create 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.

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

int main(int argc, char* argv[])
{
    ACE_UNUSED_ARG(argv);

    ACE_NEW_RETURN (g_allocator,
        ALLOCATOR (BACKING_STORE),
        -1);

    if(argc > 1)
    {
        showRecords();
    }
    else
        addRecords();

    g_allocator->sync ();

    delete g_allocator;

    return 0;
}
```

Next we go ahead and 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. The example actually 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 only looking 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 take a look at the record type that we are inserting into memory.

```
class Record
{
public:
    Record(int id1, int id2, char *name)
        :id1_(id1), id2_(id2), name_(0)
```

```
    {
        size_t len = ACE_OS::strlen (name) + 1;
        this->name_ = ACE_reinterpret_cast (char *,
            g_allocator->malloc (len));
        ACE_OS::strcpy (this->name_, name);
    }

    ~Record() { g_allocator->free(name_);}

    char* name()
    {
        return name_;
    }

    int id1()
    {
        return id1_;
    }

    int id2()
    {
        return id2_;
    }

private:
    int id1_;
    int id2_;
    char *name_;
};
```

The `Record` class has three data members: two simple integers and a char pointer, `name_`, that represents a string. This is where the tricky part comes in, if we just allocate the record in shared memory then we would allocate just enough space for the two integers and the pointer. The pointer itself would be pointing somewhere out into heap space. This is definitely not what we want, as the next time the application is run the pointer will have a value that points to heap space that does not exist anymore—a recipe for looming disaster.

To ensure that the value `name_` is pointing to is in the shared memory pool we explicitly allocate it using the shared memory allocator in the constructor.

```
int addRecords()
{
    char buf[32];
```

```
    for(int i =0; i < 10; i++)
    {
        ::sprintf(buf, "%s:%d", "Record", i);

        void * memory =
            g_allocator->malloc(sizeof (Record));
        if(memory == 0)
            ACE_ERROR_RETURN((LM_ERROR,
                "Unable to allocate memory\n"), -1);

        Record* newRecord =
            new (memory) Record(i, i+1, buf);
        //Allocate and place record

        if (g_allocator->bind (buf,
            newRecord) == -1)
            ACE_ERROR_RETURN ((LM_ERROR,
                    "bind failed\n"),
                    -1);
    }

    return 0;
}
```

To add a record we allocate enough memory for the record using the shared memory allocator and then use the placement `new` operator to "place" a new `Record` object into this memory. Next we bind the record into the allocator using it's map interface. The key is the name of the record and the value is the record itself.

```
void showRecords()
{
    ACE_DEBUG ((LM_DEBUG,
        "The following records were found.......\n\n"));

    {
        MALLOC_LIFO_ITERATOR
                iter (*g_allocator);

        for (void *temp = 0;
                    iter.next (temp) != 0;
                    iter.advance ())
        {
            Record *record =
```

```
                ACE_reinterpret_cast (Record *,
                                      temp);
            ACE_DEBUG ((LM_DEBUG,
                        "Record name: %s|id1:%d|id2:%d\n",
                        record->name(),
                ACE_reinterpret_cast (Record *,
                record->id1(),
                record->id2()));
        }
    }
}
```

Finally we illustrate LIFO iteration and the fact that the records were indeed persisted by iterating through them and displaying them on the screen. When we run the example again we get the following results;

```
The following records were found.......


Record name: Record:9|id1:9|id2:10
Record name: Record:8|id1:8|id2:9
Record name: Record:7|id1:7|id2:8
Record name: Record:6|id1:6|id2:7
Record name: Record:5|id1:5|id2:6
Record name: Record:4|id1:4|id2:5
Record name: Record:3|id1:3|id2:4
Record name: Record:2|id1:2|id2:3
Record name: Record:1|id1:1|id2:2
Record name: Record:0|id1:0|id2:1
```

17.3 Position Independent Allocation

In the last 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? Let me explain this with 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 it 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 it will not be there, instead it is mapped at 0x7e0001a0 in it's 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 we looked at worked. If the OS did not assign the same base address to the allocator then 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 only occur if you use the special `ACE_MMAP_Memory_Pool::NEVER_FIXED` option; we will talk more about this and other options later).

However, as usual, ACE comes to the rescue. ACE includes several classes that, when used in conjunction with each other, allow you to perform position-independent memory allocation. The way this works is that these classes calculate offsets from the current base address and store them into shared memory. So the allocator knows that the record is located at an offset of 0x01a0 from the base address instead of just knowing that the record is at 0x400001a0. Of course this comes with some overhead in terms of memory usage 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 the position-independent allocation classes.

```
#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 off by changing the typedef for our allocator, instead of using `ACE_Malloc` we use its base template 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)
        :id1_(id1), id2_(id2)
    {
        ACE_OS::strcpy (recName_, name);
        name_ = recName_;
    }

    char* name()
    {
        return name_;
    }

    int id1()
    {
        return id1_;
    }

    int id2()
    {
        return id2_;
    }

private:
    int id1_;
    int id2_;
    char recName_[128];
    ACE_Based_Pointer_Basic<char> name_;
};
```

We also need to change our `Record` class a little. Instead of using a raw pointer for the name, we first specify an array for the name called `recName_`. We

do this as it makes it easier to allocate the entire record in one operation, instead of first allocating a `Record` and then allocating the name. Next, we use an `ACE_Based_Pointer`, `name_`, that *points* back to recName_. This utility class calculates and keeps the offset of the name string instead of just keeping the raw pointer to the string. If the underlying memory region is mapped to a different address we will still get the right pointer for `name_` because `ACE_Based_Pointer` will recalculate the pointer for different base addresses.

This class also overloads several useful operators, including `()`, which for the most part allow you to treat `name_` as a regular pointer. Once again the class automatically recalculates the address for the pointer when it is mapped to a different base address.

```
#define BACKING_STORE "backing2.store"
//backing file where the data is kept

int main(int argc, char* argv[])
{
    ACE_UNUSED_ARG(argv);

    if(argc > 1)
    {
        ACE_MMAP_Memory_Pool_Options
            options(ACE_DEFAULT_BASE_ADDR,
            ACE_MMAP_Memory_Pool_Options::ALWAYS_FIXED);

        ACE_NEW_RETURN (g_allocator,
            ALLOCATOR (BACKING_STORE,
            BACKING_STORE,
            &options),
            -1);

        ACE_DEBUG((LM_DEBUG, "Mapped to base address %x\n",
            g_allocator->base_addr()));
        showRecords();
    }
    else
    {
        ACE_MMAP_Memory_Pool_Options options(0,
            ACE_MMAP_Memory_Pool_Options::NEVER_FIXED);

        ACE_NEW_RETURN (g_allocator,
            ALLOCATOR (BACKING_STORE,
            BACKING_STORE,
            &options),
```

```
            -1);
        ACE_DEBUG((LM_DEBUG, "Mapped to base address %x\n",
            g_allocator->base_addr()));
        addRecords();
    }


    g_allocator->sync ();

    delete g_allocator;

    return 0;
}
```

To illustrate that this works, we explicitly map the allocator to a different base address when we are adding records versus when we are just showing them. To achieve this we use the `ACE_MMAP_Memory_Pool_Options` class. This allows us to specify various options including the base address and whether we want the OS to map to a fixed address or not. Other options are described in the table below.

When we are adding records we ask the OS to map the address to any address it pleases; however when we are showing records we map to a specific fixed base address. When you run the example you can see that the allocator is mapped to a different address each time it is run, even though this occurs the program works flawlessly.

Table 17.3. ACE_MMAP_Memory_Pool_Options

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()`.
`guess_on_fault`	When a fault occurs try to guess the faulting address and remap/extend the mapped file to cover it.
`sa`	`LPSECURITY_ATTRIBUTES` on Windows
`file_mode`	File access mode to use when creating the backing file.

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 just 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. All you need to do is create the type by passing 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 interface compatibility issue, there is one other issue that 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 only valid in the original process that created the allocator and not any other processes that wishes to share the container. To overcome this problem you must provide the container with a valid memory allocator reference for all its operations. ACE overloads the 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.

Finally since most, if not all, ACE containers contain raw pointers, you cannot expect to use them between processes that map them to different base addresses. ACE currently does not include any container class that uses the position independent pointers we showed you earlier (although you could easily create one on your own). Therefore if you are going to use a container in shared memory you must make sure that you can map the entire container into all sharing processes at the same base address.

17.4.1 Hash Map

Now let's get down to an example. In the next example, we are going to put a hash map into shared memory. We will have a parent process add records into the hash table and have two other worker processes consume these records and remove them from the map. To ensure the map consistency we create an `ACE_Process_Mutex` and use it to serialize access to the map.

```
#include "ace/Hash_Map_With_Allocator_T.h"
#include "ace/Malloc_T.h"
#include "ace/PI_Malloc.h"
#include "ace/Process_Mutex.h"
#include "ace/Process.h"

#define BACKING_STORE "map.store"
#define MAP_NAME "records.db"

class Record;

typedef ACE_Allocator_Adapter<ACE_Malloc_T <ACE_MMAP_MEMORY_POOL,
```

```
                ACE_Process_Mutex, ACE_Control_Block> >
            ALLOCATOR;

typedef ACE_Hash_Map_With_Allocator<int, Record>
            MAP;

ACE_Process_Mutex coordMutex("Coord-Mutex");
```

We start by creating a few convenient type definitions and creating the coordination mutex globally. We create a position-independent allocator and use `ACE_Allocator_Adapter` to adapt the interface to `ACE_Allocator`. We define a hash map with simple integer keys and `Record` values.

Now let's take a quick look at a minor change to the `Record` class.

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

    ~Record()
    {
    }

    Record(const Record& rec)
        :id1_(rec.id1_), id2_(rec.id2_)
    {
        ACE_OS::strcpy(recName_, rec.name_);
        this->name_ = recName_;
    }

    Record(int id1, int id2, char *name)
        :id1_(id1), id2_(id2)
    {
        ACE_OS::strcpy(recName_, name);
        this->name_ = recName_;
    }

    char* name()
    {
        return recName_;
    }
```

```
    int id1()
    {
        return id1_;
    }

    int id2()
    {
        return id2_;
    }

private:
    int id1_;
    int id2_;
    char recName_[128];
    ACE_Based_Pointer_Basic<char> name_;
};
```

Notice that we have written a copy constructor for the `Record` class as the map requires this. During a copy we make a deep copy of the record name. This allows the container to safely delete the memory it allocates for a `Record` object during an unbind.

Next let's take a look at how we actually create and place the map into our allocator.

```
MAP* smap(ALLOCATOR * shmem_allocator)
{
    void *db = 0;

    if (shmem_allocator->find (MAP_NAME, db) == 0)
        return (MAP *) db;
    else
    {
        void *hash_map = 0;
        size_t hash_table_size = sizeof (MAP);
        hash_map = shmem_allocator->malloc (hash_table_size);

        if (hash_map == 0)
            return 0;

        new (hash_map) MAP (hash_table_size, shmem_allocator);

        if(shmem_allocator->bind (MAP_NAME, hash_map) == -1)
        {
            ACE_ERROR ((LM_ERROR,
```

```
                    "allocate_map\n"));

            shmem_allocator->remove ();
            return 0;
        }

        return (MAP*)hash_map;
    }
}
```

Since the map will actually be shared among processes we have written a little helper routine that helps us find the map if it already exists or creates a new one if it has not been created yet. This method assumes that the caller already has control of the coordinating mutex (otherwise you might leak a map).

First we look for the map in the allocator using the convenient `find()` method. If it is not found, we then go ahead and allocate memory for a new map and use the placement `new` operator to place a map in this memory. We then associate it with the key MAP_NAME so that in the future it will be found there by other processes.

```
int handle_parent(char *cmdLine)
{
    ACE_TRACE("::handle_parent");

   ALLOCATOR * shmem_allocator = 0;
   ACE_MMAP_Memory_Pool_Options
            options(ACE_DEFAULT_BASE_ADDR,
            ACE_MMAP_Memory_Pool_Options::ALWAYS_FIXED);

    ACE_NEW_RETURN(shmem_allocator,
        ALLOCATOR(BACKING_STORE, BACKING_STORE, &options), -1);

    MAP* map =
        smap(shmem_allocator);

    ACE_Process processa, processb;
    ACE_Process_Options poptions;
    poptions.command_line("%s a", cmdLine);
    {
        ACE_GUARD_RETURN(ACE_Process_Mutex, ace_mon,
            coordMutex, -1);

        ACE_DEBUG((LM_DEBUG,
            "(%P|%t) Map has %d entries\n",
```

```
                map->current_size()));

            ACE_DEBUG((LM_DEBUG,
                "In parent map is located at %x\n",
                    map));

            pid_t pida = processa.spawn(poptions);
            //then have the child show and eat them up.

            addRecords(map, shmem_allocator);
            //first append a few records.
        }


        {
            ACE_GUARD_RETURN(ACE_Process_Mutex, ace_mon,
                coordMutex, -1);

            addRecords(map, shmem_allocator);
            //add a few more records..

            ACE_DEBUG((LM_DEBUG,
                "(%P|%t) Parent finished adding, map has %d entries\n"
,
                map->current_size()));
            //let's see what's left.

            pid_t pidb = processb.spawn(poptions);
            //have another child try to eat them up.
            //let the children do their stuff..
        }


        processa.wait();
        processb.wait();

        //no one around anymore and don't want to
        //keep the data around anymore
        //safe to remove it.
        //!!This will remove the backing store.!!
        shmem_allocator->remove ();

        delete shmem_allocator;

        return 0;
    }
```

```
int main(int argc, char* argv[])
{
    if(argc == 1) //parent
        ACE_ASSERT(handle_parent(argv[0]) == 0);
    else
        ACE_ASSERT(handle_child() == 0);

    return 0;
}
```

When the program is started it will go into the handle_parent function. The first thing we do is create the shared memory allocator on the heap. Next we acquire the coordinating mutex and add a few records into the map and then start off a child process to process these records. Since we still hold the coordinating mutex the child process will not be able to process the records until the guard goes out of scope. After we release the mutex we once again try to acquire it and add further records, we also spawn off a second child to finish processing these records.

```
int addRecords(MAP * map, ALLOCATOR *shmem_allocator)
{
    ACE_TRACE("::addRecords");

    char buf[32];
    size_t mapLength = map->current_size();

    ACE_DEBUG((LM_DEBUG,
        "Adding 20 entries, map had %d entries\n",
        mapLength));

    for(int i = mapLength ; i < mapLength + 20; i++)
    {
        ACE_OS::sprintf(buf, "%s:%d", "Record", i);

        Record newRecord(i, i+1, buf);
        //allocate new record on stack

        ACE_DEBUG((LM_DEBUG,
            "Adding a record for %d\n", i));
        int result =
            map->bind(i,
                    newRecord, shmem_allocator);
        if(result == -1)
            ACE_ERROR_RETURN ((LM_ERROR,
```

```
            "bind failed\n"),-1);
    }

    return 0;
}
```

The `addRecords()` routine is simple enough. All we do is create a new record on the stack and then bind it into the underlying map. Notice that we use a special `bind()` method here that takes a pointer to the allocator in addition to the key value pair. As we explained earlier, we must specify the allocator whenever we use the hash map. The internal reference the hash map keeps is only valid in the process that creates the hash map (in this case the parent process is that process and this is not strictly necessary but you should always follow this rule to avoid errors).

When we `bind()` the record into the hash map, the map will use the allocator to create a copy of the record in shared memory. Now let's take a look at how the child process will process and then remove the record from the map.

```
int handle_child()
{
    ACE_TRACE("::handle_child");

    ACE_GUARD_RETURN(ACE_Process_Mutex, ace_mon, coordMutex, -1);

    ALLOCATOR * shmem_allocator = 0;
    ACE_MMAP_Memory_Pool_Options
            options(ACE_DEFAULT_BASE_ADDR,
            ACE_MMAP_Memory_Pool_Options::ALWAYS_FIXED);

    ACE_NEW_RETURN(shmem_allocator,
        ALLOCATOR(BACKING_STORE, BACKING_STORE, &options), -1);

    MAP * map = smap(shmem_allocator);

    ACE_DEBUG((LM_DEBUG,
        "(%P|%t) Map has %d entries\n",
        map->current_size()));

    ACE_DEBUG((LM_DEBUG,
            "In child map is located at %x\n",
                map));
```

```
    processRecords(map, shmem_allocator);

    shmem_allocator->sync();

    delete shmem_allocator;

    return 0;
}
```

Unlike the parent, the child first acquires the coordinating mutex and then creates the shared memory allocator. Since the child is actually going to dereference records that are created by the parent it must be sure that the underlying pool does not grow after it has created an allocator and mapped the pool into its address space. If not the following scenario could occur:

1. The child creates the allocator whose underlying map is, for example, 16K bytes in size.
2. The parent continues to add further records that cause the map to grow to 20K.
3. The child gets the coordinating mutex and starts accessing all the records in the map.
4. The chikd reaches a record that lies outside its mapping (i.e., between the 16K and 20K) region, it dereferences it and oops.. receives an address exception.
5. The parent process did not have to worry about this as it knows the child never causes the map to grow.

We have solved this problem by taking the rather draconian step of locking everyone else out of the entire map. ACE offers another solution that we will talk about in the next section.

```
int processRecords(MAP * map,
                   ALLOCATOR *shmem_allocator)
{
    ACE_TRACE("::processRecords");

    size_t mapLength = map->current_size();
    ACE_DEBUG ((LM_DEBUG,
        "(%P|%t) Found %d records\n\n", mapLength));

    int *todelete = new int[mapLength];
    int i =0;

    for(MAP::iterator iter = map->begin();
        iter != map->end();
```

```
    iter++)
{
    int key = (*iter).ext_id_;

    ACE_DEBUG((LM_DEBUG,
        "(%P|%t) [%d] Pre Processing %d:%x\n", i+1,
        key, &(*iter).ext_id_));

    todelete[i++] = key;
    //mark this message for deletion

    Record record;
    int result =
        map->find(key, record,
                    shmem_allocator);
    if(result == -1)
        ACE_DEBUG((LM_ERROR,
            "Could not find record for %d\n",
            key));
    //check out the find feature of the map.

    ACE_DEBUG ((LM_DEBUG,
        "Record name: %s|id1:%d|id2:%d\n",
         record.name(),record.id1(),record.id2()));
    //show for all to see.
}


for(int j=0; j < i ; j++)
{
    int result
        = map->unbind(todelete[j],
                shmem_allocator);

    if(result == -1)
        ACE_ERROR_RETURN((LM_ERROR,
            "Failed to unbind key %d\n", todelete[j]),
                -1);
    else
        ACE_DEBUG((LM_INFO,
            "Fully processed and removed %d\n",
            j));
}
//delete everything we processed
```

```
    delete [] todelete;

    return 0;
}
```

Processing the records involves iterating through the map, collecting all the record ids that need to removed and then `unbind()`'ing them. Notice that the `find()` and `unbind()` methods both take an additional allocator argument just like the `bind()` method did above.

Although this last example was a little more involved it illustrates several issues that you need to be aware of when you are using shared memory.

1. You must first realize that this example depends on the fact that all processes (parent and child) manage to map the shared memory pool to the same base address. If you cannot achieve this then the pointers in the hash map will be invalid.
2. If one process causes the underlying memory pool to grow (in this case the parent does this by calling `addRecords()`) this does not grow the pool in the other processes (in this case the children calling processRecords()). When the child accesses a record that is not in range, de referencing it will cause a fault.

17.4.2 Handling Pool Growth

You can handle the memory pool growth in three different ways.

1. You can make sure that it does not happen by allocating enough memory to start off with thus preventing the problem from occurring. If this is possible then this is perhaps the easiest, most portable and efficient thing to do.
2. You can make sure that the child process only maps the pool in and uses it when it is sure that the pool will not grow behind it's back, similar to the previous example. The problem here is that you lose all opportunities for concurrent read/write access to the entire pool.
3. Use your OS exception handling mechanism in conjunction with the ACE provided remap() functionality to remap the pool so that the faulting address is now mapped into the process. OS exception handling comes in two flavors, UNIX based signal handling, and Win32 structured exception handling.

For those of you who are lucky enough to be using a UNIX variant ACE will automatically install a signal handler for SIGSEGV for you. This will cause the

remapping to occur transparently. If you are using Win32 you will have to handle the exception yourself using structured exception handling.

In this next example we will show you how to handle pool growth using Win32 Structured Exception Handling. Also in this example we will extend the same idiom ACE uses to create `ACE_Hash_Map_With_Allocator` to build our own `Unbounded_Queue` that will work with shared memory. The program will start off and spawn two child processes. The parent will then add messages into a queue and the child processes will continuously dequeue these messages from the queue until they get a special termination message, at which point they will exit.

Let's start off by taking a look at the `Unbounded_Queue` type that we have created.

```
template <class T>
class Unbounded_Queue : public ACE_Unbounded_Queue<T>
{
public:
    typedef ACE_Unbounded_Queue<T> BASE;

    Unbounded_Queue(ACE_Allocator* allocator)
        :ACE_Unbounded_Queue<T>(allocator)
    {
    }

    int enqueue_tail (const T &new_item, ACE_Allocator* allocator)
    {
        this->allocator_ = allocator;
        return BASE::enqueue_tail(new_item);
    }

    int dequeue_head (T &item, ACE_Allocator* allocator)
    {
        this->allocator_ = allocator;
        return BASE::dequeue_head(item);
    }

    void delete_nodes (ACE_Allocator* allocator)
    {
        this->allocator_ = allocator;
        delete_nodes();
    }

};
```

This simple data type overloads the enqueue and dequeue methods to allow us to specify the shared memory allocator in each method. All we do is reassign the allocator pointer so that we are sure that the queue uses a valid memory allocator instead of using the invalid allocator pointer it has in shared memory.

```
int handle_parent(char *cmdLine)
{
    ALLOCATOR * shmem_allocator =0;
    ACE_MMAP_Memory_Pool_Options
            options(ACE_DEFAULT_BASE_ADDR,
            ACE_MMAP_Memory_Pool_Options::ALWAYS_FIXED);

    ACE_NEW_RETURN(shmem_allocator,
        ALLOCATOR(BACKING_STORE, BACKING_STORE, &options), -1);
    //create the allocator.

    ACE_Process processa, processb;
    ACE_Process_Options poptions;
    poptions.command_line("%s a", cmdLine);
    processa.spawn(poptions);
    processb.spawn(poptions);

    ACE_OS::sleep(2);
    //make sure the child does map
    //a partial pool in memory

    for(int i=0; i < 100; i++)
        sendRecord(i, shmem_allocator);
    sendRecord(-1, shmem_allocator);

    processa.wait();
    processb.wait();

    shmem_allocator->remove ();

    return 0;
}
```

When the parent starts it first allocates the shared memory pool and spawns off two child processes. At this point it waits for a bit, just to make sure that each one of the children has mapped in the pool at this point (where it has no queue nor messages on the queue). This ensures that after the parent adds messages to the queue the memory pool will need to grow. The parent then quickly places 100

records on the queue, sends a termination message and waits for the children to exit.

```
int handle_child()
{
    ALLOCATOR * shmem_allocator =0;
    ACE_MMAP_Memory_Pool_Options
            options(ACE_DEFAULT_BASE_ADDR,
            ACE_MMAP_Memory_Pool_Options::ALWAYS_FIXED);

    ACE_NEW_RETURN(shmem_allocator,
        ALLOCATOR(BACKING_STORE, BACKING_STORE, &options), -1);
    //create the allocator.

    g_shmem_allocator = shmem_allocator;

#if defined(WIN32)
    while(processWin32Record(shmem_allocator) != -1);
#else
    while(processRecord(shmem_allocator) != -1);
#endif

    return 0;
}
```

On startup, the child goes into the `handle_child()` method that creates the shared memory allocator and then loops through processing records. Note that we have elected to use polling here, in a realistic application you would probably want to place a condition variable in shared memory and use that to notify the child when it is appropriate to read.

If you are running on WIN32 we call a special process`Win32Record()` function that uses structured exception handling to handle the remapping case. Let's first take a quick look at `processRecord`().

```
int processRecord(ALLOCATOR* shmem_allocator)
{
    ACE_GUARD_RETURN(ACE_Process_Mutex, ace_mon, coordMutex, -1);

    QUEUE* queue =  squeue(shmem_allocator);
    if(queue == 0)
    {
        delete shmem_allocator;
        ACE_ERROR_RETURN((LM_ERROR,
            "Could not obtain queue"), -1);
```

```
    }

    if(queue->is_empty())
        return 0;
    //nothing to process.

    Record record;
    int result =
        queue->dequeue_head(record, shmem_allocator);
    if(result == -1)
    {
        ACE_ERROR_RETURN((LM_ERROR,
            "Failed to process message\n"), -1);
    }

    ACE_DEBUG ((LM_DEBUG,
        "(%P|%t) Processing record|name: %s|Record id1:%d|Record i
d2:%d\n",
        record.name(),record.id1(),record.id2()));

    if(record.id1() == -1)
        queue->enqueue_tail(record, shmem_allocator);

    return record.id1();
}
```

All we do here is get the queue, check to see if there are any messages on it and dequeue the message. If the message was a termination message we put it back on the queue so that any other children can pick it up and process it.

```
#if defined(WIN32)

int handle_remap(EXCEPTION_POINTERS *ep)
{
    ACE_DEBUG((LM_INFO,
        "Handle a remap\n"));

    DWORD ecode =
        ep->ExceptionRecord->ExceptionCode;

    if (ecode != EXCEPTION_ACCESS_VIOLATION)
        return EXCEPTION_CONTINUE_SEARCH;

    void *addr =
        (void *) ep->
```

```
            ExceptionRecord->ExceptionInformation[1];

    if (g_shmem_allocator->
            alloc ().memory_pool ().remap (addr) == -1)
        return EXCEPTION_CONTINUE_SEARCH;
#if __X86__
    // This is 80x86-specific.
    ep->ContextRecord->Edi =
        (DWORD) addr;
#elif __MIPS__
    ep->ContextRecord->IntA0 =
        ep->ContextRecord->IntV0 =
        (DWORD) addr;
    ep->ContextRecord->IntT5 =
        ep->ContextRecord->IntA0 + 3;
#endif /* __X86__ */
    return EXCEPTION_CONTINUE_EXECUTION;
}

int processWin32Record(ALLOCATOR * shmem_allocator)
{
    ACE_SEH_TRY
    {
        return processRecord(shmem_allocator);
    }
    ACE_SEH_EXCEPT (handle_remap (GetExceptionInformation ()))
    {
    }

    return 0;
}
#endif /*WIN32*/
```

Finally we get to processWin32Record(), here we place `process-Record()` in an SEH _try/_except clause (wrapped by ACE_SEH_TRY and ACE_SEH_EXCEPT). If a fault occurs the `handle_remap()` SEH selector is called. This checks to see if an EXCEPTION_ACCESS_VIOLATION occurred, if so it finds the faulting address and uses the allocator's remap() feature to map the faulting address into it's pool. Once the remap() returns we return EXCEPTION_CONTINUE_EXECUTION causing the program to continue normally.

17.5 Wrappers

Besides the more advanced features provided by `ACE_Malloc<>` and friends you may want to something much simpler, for example mapping files into memory is a common technique many web servers use to improve on the time it takes to send high hit rate file's back to client browsers.

ACE provides several wrapper classes that wrapper lower level shared memory primitives such as `mmap()`, `MapViewOfFileEx()`, System V shared memory segments etc. We will take a brief look at ACE_Mem_Map that is a wrapper around the memory mapping primitives of the OS.

In this next example, we will rewrite the classic Richard Stevens example of copying files using memory mapping. We will map both source and destination files into memory and then use a simple memcpy routine to copy source to destination.

```
int main(int argc, char *argv[])
{
    ACE_HANDLE srcHandle = ACE_OS::open(SOURCE, O_RDONLY);
    ACE_ASSERT(srcHandle != ACE_INVALID_HANDLE);

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

    ACE_Mem_Map destMapping(DEST,
            srcMapping.size(), O_RDWR|O_CREAT,
            ACE_DEFAULT_FILE_PERMS, PROT_RDWR, ACE_MAP_SHARED);
    ACE_ASSERT(destMapping.addr() != 0);

    ACE_OS::memcpy(destMapping.addr(),
        srcMapping.addr(),
        srcMapping.size());
    destMapping.sync();

    srcMapping.close();
    destMapping.close();

    return 0;
}
```

We create two ACE_Mem_Map instances that represent the memory mapping's 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 srcMapping. 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.

17.6 Conclusion

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

Chapter 18
ACE Streams Framework

The ACE_Streams framework is an excellent way to model processes consisting of a set of ordered steps. Each step in the process is implemented as an ACE_Task<> derivative. As each step is completed, the data is handed off to the next for continuation. Steps can be multi-threaded in order to increase throughput if the data lends itself to parallel processing.

In this chapter we will explore:

- ACE_Stream
- ACE_Module
- ACE_Task
- ***and others...***

18.1 ACE Streams Framework Overview

ACE's Streams framework is based on the design of Unix System V STREAMS framework although it is implemented at a higher level. ACE Streams provides the programmer with an object-oriented mechanism for processing an ordered set of actions.

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, 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 actual ACE_Stream instance and push the ordered list of modules onto it. You can then *put()* data onto the stream for processing and *get()* the results.

Figure 18.1. Diagram of an ACE_Stream

18.2 Using a One-Way Stream

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

- Answer incoming call
- Collect Caller Identification data
- Play outgoing message
- Collect incoming message
- Return recording device to the pool
- Encode collected message into a normalized form
- Save message and meta-data into message repository
- 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 up the work between different team members. Each person can focus on the implementation of their own step as long as the interfaces between each are well-defined.

18.2.1 main()

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

```
int main( int argc, char ** argv )
{
    RecordingDevice * recorder =
        RecordingDeviceFactory::instantiate( argc, argv );
```

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

```
    RecordingStream * recording_stream;
    ACE_NEW_RETURN( recording_stream,
                    RecordingStream,
                    -1);

    if( recording_stream->open() < 0 )
        ACE_ERROR_RETURN (( LM_ERROR,
                            "%p\n",
                            "RecordingStream->open() failed"), 0);
```

Our example next creates and opens an instance of RecordingStream. We will see in a minute that this is a simple derivative of ACE_Stream. If the *open()* fails we will print a brief message and exit with an error code.

```
    for(;;)
    {
        ACE_DEBUG(( LM_INFO,
                    "main - Waiting for incoming message\n" ));

        RecordingDevice * activeRecorder =
            recorder->wait_for_activity();

        ACE_DEBUG(( LM_INFO,
                    "main - Initiating recording process\n" ));

        recording_stream->record( activeRecorder );
    }

    return 0;
}
```

The final task of *main()* is to enter an infinite loop waiting for messages to arrive and recording them when they do. The Recording Device's *wait_for_activity()* will block until there is some activity (e.g. - RING) on the physical device. The Recording Device instance is then given to the Recording Stream to process our list of directives as described above.

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.

18.2.2 RecordingStream

We now take a look at the RecordingStream object. As mentioned, this is a simple derivative of ACE_Stream. We begin with the constructor which takes care of the details of initializing the baseclass.

```
class RecordingStream : public ACE_Stream<ACE_MT_SYNCH>
{
public:
    typedef ACE_Stream<ACE_MT_SYNCH> inherited;
    typedef ACE_Module<ACE_MT_SYNCH> Module;

    RecordingStream(void)
            : inherited()
    {
    }
```

A stream will always have at least two modules installed: head and tail. the default downstream task of the head module simply passes any data on down the stream. The default tail module, however, will treat any received data as an error. There are two ways our example can prevent this from happening:

- We can code our last task in the stream to not send data any further
- We can install a replacement for the task in the tail module

In order to prevent special conditions (which are generally a sign of a bad design) we will go with the second option. Therefore, our stream's *open()* method will create a Module with an instance of our EndTask object and install that at the tail of the stream.

```
    int open(void)
    {
        Module * endModule;
        ACE_NEW_RETURN( endModule,
                        Module(
                            "End Module",
                            new EndTask()
                            ),
                        -1 );

        this->inherited::open( (void *)0, (Module *)0, endModule )
;
```

Our design described eight steps in the process and we've created eight ACE_Task derivatives to implement those steps. In the next part of *open()* we create an instance of each of these eight objects and a Module to contain them.

```
Module * answerIncomingCallModule;
ACE_NEW_RETURN( answerIncomingCallModule,
                Module(
                    "Answer Incoming Call",
                    new AnswerIncomingCall()
                    ),
                -1 );

Module * collectCallerIdModule;
ACE_NEW_RETURN( collectCallerIdModule,
                Module(
                    "Collect Caller ID",
                    new CollectCallerId()
                    ),
                -1 );

Module * playOGMModule;
ACE_NEW_RETURN( playOGMModule,
                Module(
                    "Play Outgoing Message",
                    new PlayOutgoingMessage()
                    ),
                -1 );

Module * recordModule;
ACE_NEW_RETURN( recordModule,
                Module(
                    "Record Incoming Message",
                    new RecordIncomingMessage()
                    ),
                -1 );

Module * releaseModule;
ACE_NEW_RETURN( releaseModule,
                Module(
                    "Release Device",
                    new ReleaseDevice()
                    ),
                -1 );
```

```
Module * conversionModule;
ACE_NEW_RETURN( conversionModule,
                Module(
                    "Encode Message",
                    new EncodeMessage()
                    ),
                -1 );

Module * saveMetaDataModule;
ACE_NEW_RETURN( saveMetaDataModule,
                Module(
                    "Save Meta-Data",
                    new SaveMetaData()
                    ),
                -1 );

Module * notificationModule;
ACE_NEW_RETURN( notificationModule,
                Module(
                    "Notify Someone",
                    new NotifySomeone()
                    ),
                -1 );
```

The general design of the Stream framework is that an ACE_Stream contains a list of ACE_Module instances and each ACE_Module contains one or two ACE_Task instances. The tasks can be designated either downstream (e.g. -- invoked as data moves away from the stream's head) or upstream (e.g. -- invoked as data moves towards from the stream's head). Our instantiation of *collectCallerIdModule:*

```
Module * collectCallerIdModule;
ACE_NEW_RETURN( collectCallerIdModule,
                Module(
                    "Collect Caller ID",
                    new CollectCallerId()
                    ),
                -1 );
```

creates a Module (ACE_Module<ACE_MT_SYNC>) with the name "Collect Caller ID" and an instance of the CollectCallerId object which is a derivative of

ACE_Task (as required by the ACE_Module constructor signature). The CollectCallerId instance is installed as the module's downstream task. We don't provide an upstream task because our stream will store the recorded messages to disk and not expect anything to return back upstream to *main()*.

The final part of our RecordingStream's *open()* method now pushes the modules onto the stream in the correct order:

```
        if( this->push( notificationModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                ",
                                "Failed to push notificationModule
\n" ),
                               -1 );
        if( this->push( saveMetaDataModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                ",
                                "Failed to push saveMetaDataModule
\n" ),
                               -1 );
        if( this->push( conversionModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                ",
                                "Failed to push conversionModule\n
" ),
                               -1 );
        if( this->push( releaseModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                ",
                                "Failed to push releaseModule\n" )
,
                               -1 );
        if( this->push( recordModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                ",
                                "Failed to push recordModule\n" ),
                               -1 );
        if( this->push( playOGMModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                ",
                                "Failed to push playOGMModule\n" )
,
                               -1 );
        if( this->push( collectCallerIdModule ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
```

```
                              "",
                              "Failed to push "
                              "collectCallerIdModule\n" ),
                             -1 );
    if( this->push( answerIncomingCallModule ) == -1 )
        ACE_ERROR_RETURN (( LM_ERROR,
                              "",
                              "Failed to push "
                              "answerIncomingCallModule\n" ),
                             -1 );

    return 0;
}
```

Pushing the modules onto the stream in the correct order is very important. If your process is order-dependant and you push the first-used module first you'll be in for an unpleasant surprise! Why is it done this way? Well, it had to be done one way or the other and in the end it really doesn't matter. Just be sure that you remember "First Pushed, Last Used".

The remainder of our RecordingStream is the *record()* method. Whereas the constructor protected *main()* from the details of stream creation, *record()* protects it from direct interaction with the stream API.

```
int record( RecordingDevice * recorder )
{
    ACE_Message_Block * mb;
    ACE_NEW_RETURN( mb,
                    ACE_Message_Block( sizeof(Message) ),
                    -1 );

    Message * message = (Message *)mb->wr_ptr();
    mb->wr_ptr( sizeof(Message) );

    ACE_DEBUG(( LM_DEBUG,
                "RecordingStream::record() - "
                "message->recorder(recorder)\n" ));

    message->recorder(recorder);

    int rval = this->put( mb );

    ACE_DEBUG(( LM_DEBUG,
                "RecordingStream::record() - "
```

```
                    "this->put() returns %d\n", rval ));

        return rval;
    }
};
```

An ACE_Stream is, ultimately, a very fancy implementation of a linked list of ACE_Task objects. Each ACE_Task comes from the manufacturer with a built-in ACE_MessageQueue at no extra charge. That's primarily because the message queue is such an easy way to request work from a task[2].

To give our stream some work to do we create an ACE_Message_Block that will be *put()* into the first downstream task's message queue. Since our stream will be dealing with recording messages, we create the message block with enough space to contain our Message[3] object. We will basically be using the message block's data area to contain a Message object's contents.

Once our message is created we provide it with the RecordingDevice pointer so that it will be able to ask the physical device to do things during the message processing. There may be many RecordingDevice implementations, each able to talk to a different kind of device. The task of the recording stream, however, don't need to know any of these details and only need the baseclass (RecordingDevice) pointer to get the job done.

Finally, the stream's *put()* method is used to start the message down the stream by putting it on the first module's downstream task's message queue. Every stream has a head and tail module. You can actually provide these to the stream when you create it or you can take the defaults. In this case, we've taken the defaults. The default downstream task at the head of the stream simply forwards the message on to the next task.

---

2. Although this example doesn't take advantage of it, don't forget than an ACE_Task can be made to use many threads. Such tasks are allowed and, in fact, encouraged in a stream because they can greatly improve the stream's throughput

3. Message is a simple data object designed to hold a recorded message and information about that message. It's not a derivative of ACE_Message_Block. Sorry for any confusion this may be causing. Note also that we're doing a very dangerous thing here by treating the data area as a Message instance. We never actually *created* the Message object, therefore its constructor was never invoked. This is OK because it is a simple data object with no virtual functions. If it were more complex we would want to use the placement new operator to actually instantiate the Message into the message block's data area.

18.2.3 Tasks

Our example is really quite simple. Many of the tasks only need to delegate their action to the RecordingDevice instance. This is primarily because our design has required the RecordingDevice implementations to do all of the “dirty work” required to talk with the physical device. Our tasks serve only as the “glue” to ensure that things happen in the right order.

AnswerIncomingCall

Before we look at our baseclass lets look at the first task in our stream:

```
class AnswerIncomingCall : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "AnswerIncomingCall::process()\n"));

        if( message->recorder()->answer_call() < 0 )
            ACE_ERROR_RETURN((LM_ERROR,
                              "%p\n",
                              "AnswerIncomingCall"),
                             -1);

        return 0;
    }
};
```

The job of AnswerIncomingCall is to simply tell the recording device to “pickup the phone”. Recall that in *main()* we were only notified that “the phone is ringing” by the fact that the recording device’s *wait_for_activity()* unblocked. It is up to our stream’s first task to request that the recording device actually answer the incoming call. In a more robust application our stream may be in use by many recording devices at one time. Therefore we have chosen to put the recording device instance pointer into the message (in the recording stream’s *record()* method) that is passed to each task of the stream. AnswerIncomingCall uses the *recorder()* method to retrieve this pointer and then invokes *answer_call()* on it to tell the physical device to respond to the incoming call.

BasicTask

AnswerIncomingCall (and the tasks that follow) all posses a *process()* method in which they implement their required functionality. There's no mention of this method in the ACE_Task documentation... They also don't bother to move the Message they're given on to the next task in the stream. These things, and others, are handled by the common baseclass: BasicTask.

Given that our tasks are all derivatives of ACE_Task and that they expect to get their input from a message queue, it is easy to create a baseclass for all of them. This baseclass takes care of reading data from the queue, requesting a derivative to process it and passing it on to the next task in the stream. It also takes care of shutting down the tasks cleanly when the containing stream is shut down (either explicitly or implicitly by destruction).

```
class  BasicTask : public ACE_Task<ACE_MT_SYNCH>
{
public:
    typedef ACE_Task<ACE_MT_SYNCH> inherited;

    BasicTask(void)
            : inherited()
    {
    }

    virtual int open(void * = 0)
    {
        ACE_DEBUG(( LM_DEBUG,
                    "BasicTask::open()\n" ));

        return this->activate(THR_NEW_LWP|THR_JOINABLE,
                              this->desired_threads() );
    }
```

BasicTask extends ACE_Task and takes care of the housework our stream's tasks would have otherwise been bothered with. The virtual *open()* method activates the thread into one or more threads as required by the (also virtual) *desitred_threads()* method. For our simple example none of the tasks require more than one thread but we've provided two methods (overriding of *open()* or *desired_threads()*) of allowing customizing this if a task needs to do so.

We next provide a simple *put()* method that will put a message block onto the task's message queue. The *put()* method is used by the stream framework to move

messages along the stream. The *putq()* method could have been used instead but then the stream would be tied more closely to the task implementation. That is, the stream would be assuming that all tasks wish to use their message queue for communication. A *put()* method could just as well send the message to a file for the *svc()* method to pickup at a pre-defined interval.

```
int put (ACE_Message_Block *message,
         ACE_Time_Value *timeout)
{
    return this->putq(message, timeout);
}
```

Before we investigate *svc()* we need to take a look at the *close()*[4] method. When a stream is shut down via its *close()* method it will cause the *close*() of each task to be invoked with a *flags* value of *1*.

```
virtual int close(u_long flags)
{
    int rval = 0;

    if( flags == 1 )
    {

        ACE_Message_Block *hangup = new ACE_Message_Block();

        hangup->msg_type(ACE_Message_Block::MB_HANGUP);

        if ( this->putq(hangup->duplicate()) == -1)

        {
            ACE_ERROR_RETURN ((LM_ERROR,
                               "%p\n", "Task::close() putq"),
                              -1);
        }

        hangup->release();

        rval = this->wait();
```

4. There is actually a *module_close()* method that is better suited to shutting down a task when a stream is closed. Old-time users of ACE will most likely be familiar with the *close()* approach, however, which is why we've presented it here.

```
    }

    return rval;
}
```

When closed by the stream the task needs to take steps to ensure that all of its tasks are shutdown cleanly. We do this by creating a message block of type *MB_HANGUP*. Our *svc()* method will look for this and close down cleanly when it arrives. After enqueuing the hang-up request, *close()* then waits for all threads of the task to exit before returning. The combination of the hang-up message and *wait()* ensures that the stream will not shutdown before all of its tasks have had a chance to do so.

As with any other task, *svc()* is the workhorse of BasicTask. Here we get the message, ask for our derivatives to do whatever work is necessary, and then send the message on to the next task in the stream.

```
virtual int svc(void)
{
    ACE_Message_Block * message;

    for(;;)
    {
        ACE_DEBUG(( LM_DEBUG,
                    "BasicTask::svc() - "
                    "waiting for work\n" ));

        if (this->getq (message) == -1)
            ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "getq"), -1);
```

svc() consists of an infinite loop of the actions described above. The first action is to get a message from the queue. We use the simple form of *getq()* that will block until data becomes available.

```
        if (message->msg_type() ==
            ACE_Message_Block::MB_HANGUP)
        {
            if (this->putq(message->duplicate()) == -1)
            {
                ACE_ERROR_RETURN ((LM_ERROR,
                                   "%p\n",
                                   "Task::svc() putq"),
                                  -1);
```

```
    }

    message->release();

    break;
}
```

With the message in hand we check to see if it is the special hang-up message. If so, we put it back into the queue[5] so that peer threads of the task can also shut-down cleanly. We then exit the *for(;;)* loop and allow the task to end.

```
Message * recordedMessage =
    (Message *)message->rd_ptr();

if( this->process(recordedMessage) == -1 )
{
    message->release();
    ACE_ERROR_RETURN(( LM_ERROR,
                       "%p\n",
                       "process"),
                     -1);
}
```

With the message block in hand and a determination that the stream isn't being closed, we proceed to extract the read pointer from the message block and cast it into a Message pointer. Thus, we get back to the data that was originally put into the stream by the RecordingStream's *record()* method. The virtual *process()* method is now invoked to allow derivatives to do work on the message as required by the design specification.

```
ACE_DEBUG(( LM_DEBUG,
            "BasicTask::svc() - "
            "Continue to next stage\n" ));

if( this->next_step( message->duplicate() ) < 0 )
    ACE_ERROR_RETURN(( LM_ERROR,
                       "%p\n",
```

5. Astute readers may think there is a memory leak here. What happens when the last of the task's threads puts the message back into the queue? As it turns out, when message queue is destructed (which it will be when it's containing task is destructed) it will iterate through any remaining message blocks it contains and *release()* each of them.

```
                                    "put_next failed"),
                                  -1);

            message->release();
        }

        return 0;
    }
```

If all went well with *process()* we attempt to send the message on to the next module of the stream. Remember in our discussion of RecordingStream we mentioned the issue of the default downstream tail task and the need to create a replacement that would not treat input data as an error. The behavior of *next_step()* in BasicTask is to simply invoke *put_next()*:

```
protected:
    virtual int next_step( ACE_Message_Block * message_block )
    {
        return this->put_next( message_block->duplicate() );
    }
```

This will put the message block onto the next task's message queue for processing. Our custom EndTask will override this method to do nothing:

```
class EndTask : public BasicTask
{
protected:
    virtual int process( Message * message )
    {
        ACE_UNUSED_ARG( message );
        ACE_DEBUG(( LM_DEBUG,
                    "EndTask::process()\n"));
        return 0;
    }

    virtual int next_step( ACE_Message_Block * message_block )
    {
        ACE_UNUSED_ARG( message_block );
        ACE_DEBUG(( LM_DEBUG,
                    "EndTask::next_step() - end of the line.\n"));
        return 0;
    }

};
```

The nice thing about creating the custom EndTask is that we don't have to add any special code to any other task (or even the BasicTask baseclass) to handle the "end of stream" condition. This approach to special conditions is much more powerful and flexible than a host of *if* statements!

Now that we've seen how AnswerIncomingCall uses the BasicTask and how BasicTask itself is implemented we can move quickly through the rest of the tasks in the stream.

CollectCallerId

Like AnswerIncomingCall, CollectCallerId delegates its work to the RecordingDevice on which we expect to record the Message:

```
class CollectCallerId : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "CollectCallerId::process()\n"));

        CallerId * id =
            message->recorder()->retrieve_callerId();

        if( ! id )
            ACE_ERROR_RETURN((LM_ERROR,
                              "%p\n",
                              "CollectCallerId"),
                             -1);

        message->caller_id( id );

        return 0;
    }
};
```

retrive_callerId() returns an opaque CallerId object. The CallerId object contains some reference to the originator of the message. It could be a phone number, email address or even IP address depending on the physical device that is

taking the message for us. We store the CallerId in the Message for use later when we write the message's meta-data.

PlayOutgoingMessage

In this task object we retrieve an ACE_FILE_Addr pointing to an outgoing message appropriate to the recording device: an mp3 to be played through a voice modem, a text file to be sent over a socket, etc... The recorder's *play_message()* is then given the file for playing. If the recording device is smart enough, it could even convert a text message to audio data.

```
class PlayOutgoingMessage : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "PlayOutgoingMessage::process()\n"));

        ACE_FILE_Addr outgoing_message =
            this->get_outgoing_message(message);

        if( message->recorder()->play_message(
                outgoing_message ) < 0 )
            ACE_ERROR_RETURN((LM_ERROR,
                              "%p\n",
                              "PlayOutgoingMessage"),
                             -1);

        return 0;
    }

    ACE_FILE_Addr get_outgoing_message( Message * message )
    {
        // ...
    }
};
```

RecordIncomingMessage

Our survey of trivial objects continues with RecordIncomingMessage. The recorder is now asked to capture the incoming message and record it to a queue/

spool location where it can be processed later in the stream. The location of the message and its type are remembered by the Message so that later modules will be able to quickly locate the recorded data.

```
class RecordIncomingMessage : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "RecordIncomingMessage::process()\n"));

        ACE_FILE_Addr incoming_message =
            this->get_incoming_message_queue();

        MessageType * type =
            message->recorder()->record_message(
                incoming_message );

        if( ! type )
            ACE_ERROR_RETURN((LM_ERROR,
                              "%p\n",
                              "RecordIncomingMessage"),
                             -1);

        message->incoming_message( incoming_message, type );

        return 0;
    }

    ACE_FILE_Addr get_incoming_message_queue(void)
    {
        // ...
    }
};
```

ReleaseDevice

After a message has been collected we release the physical device. Remember that *main()* is operating in a different thread than the stream tasks. Thus, a new message can come into our application while we're finalizing the processing of the current one. In fact, the recording device could represent many physical chan-

nels and we could actually implement a system that has many simultaneous "current" calls. In such a system our BasicTask might instantiate each task into many threads instead of just one.

```
class ReleaseDevice : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "ReleaseDevice::process()\n"));

        message->recorder()->release();

        return 0;
    }
};
```

EncodeMessage

To keep life easy for other applications using recorded data, it is preferable to encode the three message types (text, audio, video) into a standard format. We might, for instance, require that all audio is encoded into "radio-quality" mp3.

```
class EncodeMessage : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "ReleaseDevice::process()\n"));

        ACE_FILE_Addr & incoming =
            message->addr();

        ACE_FILE_Addr addr=
            this->get_message_destination(message);

        if( message->is_text() )
            Util::convert_to_unicode( incoming, addr );
        else if( message->is_audio() )
            Util::convert_to_mp3( incoming, addr );
```

```
        else if( message->is_video() )
            Util::convert_to_mpeg( incoming, addr );

        message->addr( addr );

        return 0;
    }

    ACE_FILE_Addr get_message_destination( Message * message )
    {
        // ...
    }
};
```

get_message_destination() determines the final location of the encoded message. The Util object methods take care of encoding the message appropriately and placing the result into the final path. This final path is then added to the Message in case remaining tasks need to know what it is.

SaveMetaData

SaveMetaData is our most complex task yet weighs in at only a handful of lines. By now you should have an appreciation of how the streams framework (along with a little housekeeping) can eliminate all of the tedious coding and allow you to focus on your application logic!

The message's meta-data will describe the message for other applications. Included in this information is the path to the actual message, the type of message (text, audio or video) and other interesting bits. We've chosen to create a simple XML file to contain the meta-data and the ACE_IOStream<> framework to create this XML file. ACE_IOStream<> lets us turn any object that provides a few simple methods into an iostream-like object. ACE_FILE_IO, unfortunately, doesn't provide everything we need but we easily solve that with an adaptor to provide the missing methods.

```
class SaveMetaData : public BasicTask
{
protected:
    virtual int process( Message * message )
    {
        ACE_DEBUG(( LM_DEBUG,
                    "SaveMetaData::process()\n"));
```

```
        ACE_FILE_Addr & encoded = message->addr();

        ACE_CString path( message->addr().get_path_name() );
        path += ".xml";

        ACE_FILE_Connector connector;
        ACE_FILE_IO file;
        ACE_FILE_Addr addr( path.c_str() );

        if( connector.connect( file, addr ) == -1 )
            ACE_ERROR_RETURN (( LM_ERROR,
                                "%p\n",
                                "create meta-data file"), 0);

        file.truncate(0);

        this->write(file, "<Message>\n");
        // ...
        this->write(file, "</Message>\n");

        file.close();

        return 0;
    }

private:
  int write(ACE_FILE_IO & file, const char * str)
  {
    return file.send(str, strlen(str));
  }
};
```

NotifySomeone

Our final task is to notify someone that a new message has arrived. This may be as simple as logfile entry or something more complex such as sending the message as an attachment in an email to some interested party.

```
class NotifySomeone : public BasicTask
{
protected:

    virtual int process( Message * message )
    {
```

```
        ACE_DEBUG(( LM_DEBUG,
                    "NotifySomeone::process()\n"));

        // Format an email to tell someone about the
        // newly received message.
        // ...

        // Display message information in the logfile
        ACE_DEBUG(( LM_INFO,
                    "New message from %s "
                    " received and stored at %s\n",
                    message->caller_id()->string(),
                    message->addr().get_path_name()
                    ));

        return 0;
    }
};
```

18.2.4 Remainder

In this section we have seen that creating and using the ACE_Stream framework is very easy to do. The bulk of our sample code, in fact, had little or nothing to do with the framework itself. If we had included the answering system objects we would find that the actual stream interaction code was less than 10% of the total. This is as it should be. A good framework should do as much as possible and allow you to focus on your application.

18.3 A Bi-Directional Stream

In this section we will use a bi-directional 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.

[[A diagram here would be good]]

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

18.3.1 The CommandStream

For purposes of this example we have created a RecordingDevice which 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 some appropriate manner.

We begin by looking at the definition of our CommandStream object:

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

    CommandStream(void)
            : inherited()
    {
    }

    int open( ACE_SOCK_Stream * peer );

    Command * execute( Command * command );
};
```

We expect clients of the 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. It begins by invoking the superclass' *open()* method so that we don't have to duplicate that functionality:

```
int CommandStream::open( ACE_SOCK_Stream * peer )
{
    ACE_DEBUG(( LM_DEBUG,
                "CommandStream::open(peer)\n" ));
```

```
if( this->inherited::open(0) == -1 )
    ACE_ERROR_RETURN (( LM_ERROR,
                        "",
                        "Failed to open superclass\n"
                        ),
                      -1 );
```

In this example, like the last, we are not providing custom head/tail modules. If there is no task registered to handle a command it will reach the default tail module and cause an error. A more robust implementation would replace the default tail module with one that would return an error back upstream in much the same way that results will be returned upstream.

We now create a task-pair for each RecordingDevice command we intend to support:

```
CommandModule * answerCallModule;
ACE_NEW_RETURN( answerCallModule,
                AnswerCallModule( peer ),
                -1 );

CommandModule * retrieveCallerIdModule;
ACE_NEW_RETURN( retrieveCallerIdModule,
                RetrieveCallerIdModule( peer ),
                -1 );

CommandModule * playMessageModule;
ACE_NEW_RETURN( playMessageModule,
                PlayMessageModule( peer ),
                -1 );

CommandModule * recordMessageModule;
ACE_NEW_RETURN( recordMessageModule,
                RecordMessageModule( peer ),
                -1 );
```

At this point we don’t care about the specific tasks. Each CommandModule is a derivative of ACE_Module<> and knows what tasks need to be created to process the command it represents. This approach helps to decouple the module and the stream. As each module is constructed we provide it with a copy of the

pointer to the socket. We'll see later that the modules tasks are then able to fetch this pointer when they need to interact with the socket.

With the stream ready and the modules created we can now push each one onto the stream. Because this stream doesn't represent an ordered set of steps we can push them in any order. Because a Command must flow down the stream until it is encountered by a task-pair to process it would probably be wise to have the module responsible for the most-used command to be at the beginning of the stream. Our example is pretty-much a one-shot sequence of commands, though, so the order really doesn't matter at all to us.

```
if( this->push( answerCallModule ) == -1 )
    ACE_ERROR_RETURN (( LM_ERROR,
                        "",
                        "Failed to push %s\n",
                        answerCallModule->name()),
                      -1 );

if( this->push( retrieveCallerIdModule ) == -1 )
    ACE_ERROR_RETURN (( LM_ERROR,
                        "",
                        "Failed to push %s\n",
                        retrieveCallerIdModule->name()),
                      -1 );

if( this->push( playMessageModule ) == -1 )
    ACE_ERROR_RETURN (( LM_ERROR,
                        "",
                        "Failed to push %s\n",
                        playMessageModule->name()),
                      -1 );

if( this->push( recordMessageModule ) == -1 )
    ACE_ERROR_RETURN (( LM_ERROR,
                        "",
                        "Failed to push %s\n",
                        recordMessageModule->name()),
                      -1 );

ACE_DEBUG(( LM_DEBUG,
            "this->inherited::open"
```

```
                "(peer,cmdModule,ioModule)\n" ));

    return 0;
}
```

The final CommandStream method is *execute()*. A client of the stream will construct a Command instance and provide it to *execute()* for processing. *execute()* will send the command downstream and wait for a result to be returned.

```
Command * CommandStream::execute( Command * command )
{
    ACE_Message_Block * mb;
    ACE_NEW_RETURN( mb,
                    ACE_Message_Block( command ),
                    0 );

    if( this->put( mb ) == -1 )
        ACE_ERROR_RETURN (( LM_ERROR,
                            "",
                            "Failed to push command %d\n",
                            command->command_
                            ),
                          0 );

    this->get( mb );

    command =
        (Command *)mb->data_block();

    ACE_DEBUG((LM_DEBUG,
               "Command (%d) returns (%d)\n",
               command->command_,
               command->numeric_result_));

    return command;
}
```

The Command object is a derivative of ACE_Data_Block. This allows us to provide it directly to the ACE_Message_Block constructor. It also allows us to take advantage of the fact that the ACE_Stream framework (ACE_Message_Block in particular) will free the data block's memory at the appropriate time so that we don't have any memory leaks.

Once the message block is configured we start it down the stream with the *put()* method. We immediately invoke *get()* to wait for the return data to be given to us. The return data is then cast back to a Command instance and returned to the caller. The TextListener implementation of RecordingDevice uses *execute()* to implement each RecordingDevice command. We'll visit that in a few pages but we first need to take a look at the objects that actually do the work.

18.3.2 Supporting Objects and Baseclasses

Command

In order for all of this to work cleanly we need a bit of support structure. First and foremost is our Command object. The Command object is the interface between the clients of the CommandStream (such as TextListener) and the tasks that provide the actual implementation.

```
class Command : public ACE_Data_Block
{
public:
    // Result Values
    enum {
        PASS    = 1,
        SUCCESS = 0,
        FAILURE = -1
    };

    // Commands
    enum {
        UNKNOWN            = -1,
        ANSWER_CALL        = 10,
        RETRIEVE_CALLER_ID,
        PLAY_MESSAGE,
        RECORD_MESSAGE
    } commands;

    int flags_;
    int command_;

    void * extra_data_;

    int numeric_result_;
    ACE_CString result_;
};
```

The Command extends ACE_Data_Block so that we can take advantage of the auto-destruction provided by ACE_Message_Block. Recall that Command-Stream::execute() simply instantiates an ACE_Message_Block with the provided Command. The fact that Command is an ACE_Message_Block makes this trivial. In a more robust application we would create further derivatives of Command instead of using the generic public member variables.

CommandModule

CommandModule is the baseclass for all of the modules our CommandStream will be using. We got a hint of this in CommandStream::open() above. The CommandModule is nothing more than adaptor around its ACE_Module<> base-class so that we can (a) easily provide the socket to the module and (b) easily retrieve the socket. All necessary casting is handled internally by Command-Module so that its clients (the command-implementation tasks) can remain cast-free.

```
class CommandModule : public ACE_Module<ACE_MT_SYNCH>
{
public:
    typedef ACE_Module<ACE_MT_SYNCH> inherited;
    typedef ACE_Task<ACE_MT_SYNCH> Task;

    CommandModule( const ACE_TCHAR * module_name,
                   CommandTask * writer,
                   CommandTask * reader,
                   ACE_SOCK_Stream * peer );

    ACE_SOCK_Stream & peer(void);
};
```

The constructor has essentially the same signature as that of ACE_Module<> but with the more specific data types appropriate to our application:

```
CommandModule::CommandModule( const ACE_TCHAR *module_name,
                              CommandTask * writer,
                              CommandTask * reader,
                              ACE_SOCK_Stream * peer )
        : inherited( module_name,
                     writer,
                     reader,
```

```
                    peer
                    )
{
}
```

The *peer()* method makes use of the *arg()* method of ACE_Module() to retrieve it's optional data. The *arg()* result is then cast into an ACE_SOCK_Stream as expected by the clients of CommandModule:

```
ACE_SOCK_Stream & CommandModule::peer(void)
{
    ACE_SOCK_Stream * peer =
        (ACE_SOCK_Stream *)this->arg();

    return *peer;
}
```

CommandTask

The workhorse of our CommandStream framework is the CommandTask object. CommandTask extends ACE_Task<> and is the baseclass for all of the tasks that implement each command. CommandTask provides a *svc()* method that will:

- retrieve Command requests from its message queue
- determine if the task can process the Command
- decide where to pass the Command next

```
class CommandTask : public ACE_Task<ACE_MT_SYNCH>
{
public:
    typedef ACE_Task<ACE_MT_SYNCH> inherited;

    virtual ~CommandTask(void)
    {
    }

    virtual int open(void * = 0 );

    int put (ACE_Message_Block *message,
             ACE_Time_Value *timeout);

    virtual int svc(void);
```

```
    virtual int close(u_long flags);

protected:
    CommandTask(int command);

    virtual int process( Command * message );

    int command_;
};
```

Before we look at *svc()* in detail lets look at what the other methods are doing for us. The constructor simply initializes the ACE_Task<> baseclass and sets the *command_* attribute. This will be provided by a derivative and should be one of the enumerated values from Command. The *svc()* method will compare the *command_* attribute of an incoming Command to this value to determine whether or not a CommandTask derivative instance can process the requested command.

```
CommandTask::CommandTask(int command)
        : inherited(),
          command_(command)
{
}
```

The *open()* method is as we've seen before. It simply creates a new thread in which the task will execute. For this example we only need one thread per command.

```
int CommandTask::open(void *)
{
    ACE_DEBUG(( LM_DEBUG,
                "CommandTask::open()\n" ));

    return this->activate(THR_NEW_LWP|THR_JOINABLE,
                          1 );
}
```

The *put()* method is similarly familiar:

```
int CommandTask::put (ACE_Message_Block *message,
         ACE_Time_Value *timeout)
{
    return this->putq(message, timeout);
}
```

The *close()* method is yet another boiler-plate from our previous example:

```
int CommandTask::close(u_long flags)
{
    int rval = 0;

    if( flags == 1 )
    {

        ACE_Message_Block *hangup = new ACE_Message_Block();

        hangup->msg_type(ACE_Message_Block::MB_HANGUP);

        if ( this->putq(hangup->duplicate()) == -1)
        {
            ACE_ERROR_RETURN ((LM_ERROR,
                               "%p\n", "Task::close() putq"),
                              -1);
        }

        hangup->release();

        rval = this->wait();
    }

    return rval;
}
```

Next comes our virtual *process()* method. This baseclass implementation returns a failure code. The CommandTask derivatives are expected to override this method to implement the command they represent. *process()* will return Command::FAILURE if processing fails, Command::SUCCESS if it succeeds or Command::PASS if it chooses not to process the command. On failure or success the Command will be returned upstream to where the command stream's *execute*() method is blocking on *getq()*. On Command::PASS the Command instance will continue downstream. Any return from an upstream task will allow the Command to continue the upstream journey.

```
int CommandTask::process( Command * command )
{
    ACE_UNUSED_ARG(command);
    ACE_DEBUG(( LM_DEBUG,
                "CommandTask::process()\n"));
    return Command::FAILURE;
}
```

With the trivia behind us we can now take a look at *svc()*. We begin with the usual business of taking a message block off of the task's queue and checking for shutdown:

```
int CommandTask::svc(void)
{
    ACE_Message_Block * message;

    for(;;)
    {
        ACE_DEBUG(( LM_DEBUG,
                    "CommandTask::svc() - "
                    "%s waiting for work\n",
                    this->module()->name()
                    ));

        if (this->getq (message) == -1)
            ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "getq"), -1);

        if (message->msg_type() ==
            ACE_Message_Block::MB_HANGUP)
        {
            if (this->putq(message->duplicate()) == -1)
            {
                ACE_ERROR_RETURN ((LM_ERROR,
                                   "%p\n",
                                   "Task::svc() putq"),
                                  -1);
            }

            message->release();

            break;
        }
```

Now that we have a valid message block we extract its data block. We know the data block is actually a Command instance so we cast that back. If the command we're being asked to execute is not "ours" then the message block is sent on to the next module in the stream. Since this is a bi-directional stream the *put_next()* could be moving data up or down-stream but we don't care at this stage.

```
Command * command =
    (Command *)message->data_block();

ACE_DEBUG(( LM_DEBUG,
            "CommandTask::svc() - "
            "%s got work request %d\n",
            this->module()->name(),
            command->command_
            ));

if( command->command_ != this->command_ )
{
    this->put_next(
        message->duplicate() );
}
```

If the Command is our responsibility then we need to *process()* it. Or, more specifically, our derivative needs to process it. We provide a helping hand by setting the *numeric_result_* attribute to -1 if *process()* failed. This allows our derivatives' *process()* to simply return Command::FAILURE yet the command stream's client can inspect *numeric_result_*.

```
else
{
    int result = this->process( command );

    ACE_DEBUG(( LM_DEBUG,
                "CommandTask::svc() - "
                "%s work request %d result is %d\n",
                this->module()->name(),
                command->command_,
                result
                ));
```

```
if( result == Command::FAILURE )
{
    command->numeric_result_ = -1;
}
```

If the Command is intended for this task, the task's *process()* method can still decide to let someone else handle the processing. Perhaps a scenario will require two or more tasks to process the same Command. In any case, a Command::PASS return value will let the message continue along the stream:

```
else if( result == Command::PASS )
{
    this->put_next(
        message->duplicate() );
}
```

Any other return value must be success and we need to decide what should be done with the Command. If the current task is on the downstream side of the CommandStream (*is_writer()*) then we want to "turn-around" the Command and send it back to the stream head. This is simply done by putting the message block on our sibling task's message queue:

```
else // result == Command::SUCCESS
{
    if( this->is_writer() )
    {
        this->sibling()->putq(
            message->duplicate() );
    }
```

On the other hand, if the task is on the upstream side we want the Command to keep flowing upstream:

```
    else // this->is_reader()
    {
        this->put_next(
            message->duplicate()
            );
    }
} // result == ...
}     // command->command_ ?= this->command_
```

That completes the section where this task is processing a command. All that is left now is to release the message block taken off of the message queue and wrap-up the method.

```
        message->release();
    }   // for(;;)

    return 0;
}
```

18.3.3 Implementations

Answer Call

Now that we've seen the CommandModule and CommandTask baseclasses we can look at the specific implementations instantiated by CommandStream::open(). In the life cycle of a recording a call the first thing we must do is actually answer the call.

To implement the Answer Call function we've created three objects. AnswerCallModule is a CommandModule (ACE_Module<>) responsible for creating the upstream and downstream tasks:

The only method of AnswerCallModule is the constructor which simply provides the necessary information to the baseclass instance. The *peer* parameter is provided as "extra data" to the baseclass so that the *peer()* method can use the *arg()* method to return the socket to either of the tasks.

```
AnswerCallModule::AnswerCallModule( ACE_SOCK_Stream * peer )
        : CommandModule( "AnswerCall Module",
                         new AnswerCallDownstreamTask(),
                         new AnswerCallUpstreamTask(),
                         peer
                         )
{
}
```

The two tasks are responsible for handling the ANSWER_CALL command, therefore they must provide this value to the CommandTask constructor:

```
AnswerCallDownstreamTask::AnswerCallDownstreamTask(void)
        : CommandTask(Command::ANSWER_CALL)
{
}


AnswerCallUpstreamTask::AnswerCallUpstreamTask(void)
        : CommandTask(Command::ANSWER_CALL)
{
}
```

All that is left now is to implement the *process()* method that will answer the incoming connection request. The decision to put this on the downstream or upstream side is rather arbitrary and in the end doesn't really matter. We've taken the approach that any "active" action will go on the downstream side and any "passive" (or "receiving") action will go on the upstream side. Thus, we implement the connection acceptance on the downstream task:

```
int AnswerCallDownstreamTask::process( Command * command )
{
    ACE_DEBUG(( LM_DEBUG,
                "Answer Call (downstream)\n"));

    TextListenerAcceptor * acceptor =
        (TextListenerAcceptor *)command->extra_data_;

    CommandModule * module =
        (CommandModule*)this->module();

    command->numeric_result_ =
        acceptor->accept( module->peer() );

    acceptor->release();

    return Command::SUCCESS;
}
```

There are a few things worth noticing here. First, the Command instance's extra data is expected to be a TextListenerAcceptor instance. This sort of implied API could be avoided by creating derivatives of Command for each command verb.

Next of interest is the *module()* method. Any ACE_Task<> has access to this method for retrieving the ACE_Module<> in which the task is contained (if any). Because our module is actually a CommandModule, we cast the *module()* return. On our CommandModule we can then invoke the *peer()* method to get a reference to the socket.

Finally, we return Command::SUCCESS to tell CommandTask::svc() that we've processed the command and the data is ready to be sent upstream to the client.

CommandTask::svc() will then invoke *putq()* on the upstream task whose *process()* method has nothing important to do:

```
int AnswerCallUpstreamTask::process( Command * command )
{
    ACE_UNUSED_ARG(command);

    ACE_DEBUG(( LM_DEBUG,
                "Answer Call (upstream)\n"));

    return Command::SUCCESS;
}
```

Retrieve Caller Id

The action of retrieving caller id from a socket consists of gathering up the IP of the remote peer. Like the AnswerCallModule, the RetrieveCallerIdModule consists of nothing more than a constructor:

```
RetrieveCallerIdModule::RetrieveCallerIdModule(
    ACE_SOCK_Stream * peer )
        : CommandModule( "RetrieveCallerId Module",
                         new RetrieveCallerIdDownstreamTask(),
                         new RetrieveCallerIdUpstreamTask(),
                         peer
                         )
{
}
```

We consider this to be a "read" operation so we've implemented it on the upstream side of the stream:

```
RetrieveCallerIdUpstreamTask::RetrieveCallerIdUpstreamTask(void)
        : CommandTask(Command::RETRIEVE_CALLER_ID)
{
}
int RetrieveCallerIdUpstreamTask::process( Command * command )
{
    ACE_UNUSED_ARG(command);

    ACE_DEBUG(( LM_DEBUG,
                "Returning Caller ID data\n"));

    ACE_INET_Addr remote_addr;

    CommandModule * module =
        (CommandModule*)this->module();

    module->peer().get_remote_addr(remote_addr);

    ACE_TCHAR remote_addr_str[256];
    remote_addr.addr_to_string( remote_addr_str, 256 );

    command->result_ =
        ACE_CString( remote_addr_str );

    return Command::SUCCESS;
}
```

There should be nothing surprising here. We again use the *module()* method to get access to our CommandModule and it's *peer()*. The peer's address is returned in the *result_* attribute of the Command for our client's consumption.

Our downstream object is expectedly simple:

```
RetrieveCallerIdDownstreamTask::RetrieveCallerIdDownstreamTask(voi
d)
        : CommandTask(Command::RETRIEVE_CALLER_ID)
{
}
int RetrieveCallerIdDownstreamTask::process( Command * command )
{
    ACE_DEBUG(( LM_DEBUG,
```

```
                "Retrieving Caller ID data\n"));

    return Command::SUCCESS;
}
```

Play Message and Record Message

As you can see, our CommandModule and CommandTask derivatives are well-insulated from the mechanics of the ACE_Stream framework. As with our unidirectional stream example we have gone to some effort to ensure that the application programmer can focus on the task at hand (no pun intended) rather than worrying about the details of the underlying framework.

Play Message and Record Message each require an appropriate module constructor. Play Message is then implemented on the downstream side:

```
int PlayMessageDownstreamTask::process( Command * command )
{
    ACE_DEBUG(( LM_DEBUG,
                "Play Outgoing Message\n"));

    ACE_FILE_Connector connector;
    ACE_FILE_IO file;

    ACE_FILE_Addr * addr =
        (ACE_FILE_Addr *)command->extra_data_;

    if( connector.connect( file, *addr ) == -1 )
    {
        command->numeric_result_ = -1;
    }
    else
    {
        command->numeric_result_ = 0;

        CommandModule * module =
            (CommandModule*)this->module();

        char rwbuf[512];
        int rwbytes;
        while( (rwbytes = file.recv(rwbuf,512)) > 0 )
        {
            module->peer().send_n( rwbuf, rwbytes );
```

```
        }
    }

    return Command::SUCCESS;
}
```

And RecordMessage on the upstream side:

```
int RecordMessageUpstreamTask::process( Command * command )
{
    // Collect whatever the peer sends and write into the
    // specified file.
    ACE_FILE_Connector connector;
    ACE_FILE_IO file;

    ACE_FILE_Addr * addr =
        (ACE_FILE_Addr *)command->extra_data_;

    if( connector.connect( file, *addr ) == -1 )
        ACE_ERROR_RETURN (( LM_ERROR,
                            "%p\n",
                            "create file"),
                          Command::FAILURE
                          );
    file.truncate(0);

    CommandModule * module =
        (CommandModule*)this->module();

    int total_bytes = 0;
    char rwbuf[512];
    int rwbytes;
    while( (rwbytes = module->peer().recv(rwbuf,512)) > 0 )
    {
        total_bytes += file.send_n( rwbuf, rwbytes );
    }

    file.close();

    ACE_DEBUG(( LM_INFO,
                "RecordMessageUpstreamTask "
                "- recorded %d byte message\n",
```

```
                total_bytes ));

    return Command::SUCCESS;
}
```

18.3.4 Using the Command Stream

All of our component parts are now in place. We have an opaque CommandStream into which one can place a Command instance and retrieve a response. The CommandStream is built from a list of CommandModule derivatives each of which contain a pair of CommandTask derivatives. The actual program logic is implemented in the CommandTask derivatives and, for the most part, is immune to the details of the ACE_Stream framework.

Let's now take a look at how the TextListener implementation of RecordingDevice uses the CommandStream. Because we can concurrently accept connections on a socket and process established connections we actually have two RecordingDevice derivatives to implement our "socket recorder". The first, TextListenerAcceptor, implements only the *wait_for_activity()* method of RecordingDevice. That has nothing to do with the stream interaction so we won't go into the details. In short, TextListenerAcceptor::wait_for_activity() will wait for a connection request on the socket and return a new TextListener instance when that happens.

TextListener implements the other RecordingDevice interface using the CommandStream. We begin with the constructor as invoked by the acceptor:

```
TextListener::TextListener(TextListenerAcceptor * acceptor)
        : acceptor_(acceptor)
{
    ACE_DEBUG(( LM_DEBUG,
                "TextListener ctor\n"));

    ACE_NEW( this->command_stream_,
             CommandStream());

    this->command_stream_->open( &(this->peer_) );
}
```

The TextListenerAcceptor doesn't even *accept()* the incoming socket connection. It shouldn't, that's a job for the Answer Call task as shown in the previous

sub-section. Therefore... the acceptor's *wait_for_activity()* method provides a pointer to the acceptor object when the TextListener is created. This is held in a member attribute until ready to be used in the *answer_call()* method.

We've chosen not to implement an *open()* method for the TextListenerAcceptor to invoke. Thus, we create and initialize the CommandStream directly in the constructor. The *peer_* attribute is not yet connected but because it is instantiated we can safely provide a pointer to it to the command stream at this stage. In fact, we must provide it (API aside) so that the answer_call task can perform the connection.

```
int TextListener::answer_call(void)
{
    ACE_DEBUG(( LM_DEBUG,
                "TextListener::answer_call()\n"));

    Command * c = new Command();
    c->command_ = Command::ANSWER_CALL;

    c->extra_data_ = this->acceptor_;

    c = this->command_stream_->execute( c );

    ACE_DEBUG(( LM_DEBUG,
                "TextListener::answer_call() result is %d\n",
                c->numeric_result_));

    return c->numeric_result_;
}
```

To implement the *answer_call()* method the TextListener first creates a Command instance with the *command_* attribute set to Command::ANSWER_CALL. This tells the command stream what needs to be done. We also save the TextListenerAcceptor instance (*acceptor_*) as extra data on the Command. This satisfies the implied API of the AnswerCallDownstreamTask so that it can establish the actual connection.

The newly-created Command is then given to the stream for execution. As written, the *execute()* method will block until the command has been completed. For simplicity, *execute()* will return a Command instance identical to the one it was given[6] plus the return values.

The remainder of the TextListener methods follow this general pattern. In life cycle order they are:

retrieve_callerId

```
CallerId * TextListener::retrieve_callerId(void)
{
    ACE_DEBUG(( LM_DEBUG,
                "TextListener::retrieve_callerId()\n"));

    Command * c = new Command();
    c->command_ = Command::RETRIEVE_CALLER_ID;

    c = this->command_stream_->execute( c );

    CallerId * caller_id =
        new CallerId(
            c->result_
            );

    return caller_id;
}
```

play_message

```
int TextListener::play_message( ACE_FILE_Addr & addr )
{
    MessageType * type = Util::identify_message( addr );

    if( type->is_text() )
    {
        Command * c = new Command();
        c->command_ = Command::PLAY_MESSAGE;

        c->extra_data_ = &addr;

        c = this->command_stream_->execute( c );

        return c->numeric_result_;
    }
```

6. In fact, it will return the same instance in this particular implementation.

```
    ACE_FILE_Addr temp( "/tmp/outgoing_message.text" );
    ACE_FILE_IO * file;

    if( type->is_audio() )
        file = Util::audio_to_text( addr, temp );
    else if( type->is_video() )
        file = Util::video_to_text( addr, temp );

    int rval = this->play_message(temp);
    file->remove();

    return rval;
}
```

record_message

```
MessageType * TextListener::record_message( ACE_FILE_Addr & addr )
{
    Command * c = new Command();
    c->command_ = Command::RECORD_MESSAGE;

    c->extra_data_ = &addr;

    c = this->command_stream_->execute( c );

    if( c->numeric_result_ == -1 )
    {
        return 0;
    }

    return new MessageType( MessageType::RAWTEXT, addr );
}
```

release

The *release()* method is invoked by our RecordingStream framework when the recording process is complete. Because a new TextListener is instantiated for each recording we take this opportunity to free that memory:

```
void TextListener::release(void)
{
    delete this;
}
```

18.4 Summary

In this chapter we have investigated the ACE_Stream 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.

Let's take a moment and enumerate 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

Some things to keep in mind when architecting your stream:

1. Each task of in the stream can exist in one or more threads. Use multiple threads when your application can take advantage of parallel processing.
2. 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.
3. You can provide your tasks' *open()* method with arbitrary data by passing it as the fourth parameter (*args*) of module's constructor or *open()* method.
4. 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.
5. ACE_Message_Block instances are reference counted and will not leak memory if you correctly use their *duplicate()* and *release()* methods.

Chapter 19
ACE Service Configurator

19.1 ACE Service Configurator Framework Overview

The ACE Service Configurator framework allows you to dynamically configure services and streams at run time, whether they are statically linked into your program or the objects are dynamically loaded from 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 of this can be done 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. Some reasons why run time configuration is beneficial:

- Multiple types of services are linked into the program (or available in shared libraries) and the set of services to actually activate is deferred until run time, 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. Why would anyone want to configure a static service? All the reasons above apply to static as well as dynamic services. Additionally, statically linked programs are sometimes favored for simplicity or for security concerns. Let's look at an example of a statically configured service, focusing on 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 staticly 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 class declaration above shows 2 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 `ACE_Svc_Handler` is a subclass of `ACE_Task`; therefore, `ACE_Task` and `ACE_Svc_Handler` are often used to implement configurable services.
2. There are three important hook methods for each service to implement:
   - `virtual int init (int argc, ACE_TCHAR *argv[])`—this hook method 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)`—this hook method is called by the framework when an instance of the service is being shut down.
   - `virtual int info (ACE_TCHAR **str, size_t len)`—this method is optional. It is used for the service to report information about itself when asked.

Let's look at the implementations of the `init()` and `fini()` hook methods for our example service:

```
int
HA_Status::init (int argc, ACE_TCHAR *argv[])
{
  static const ACE_TCHAR options[] = ACE_TEXT (":f:");
  ACE_Get_Opt cmd_opts (argc, argv, options, 0);
  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);
```

```
    }

  ACE_Configuration_Heap config;
  config.open ();
  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));

  if (this->acceptor_.open (this->listen_addr_) != 0)
    ACE_ERROR_RETURN
      ((LM_ERROR,
        ACE_TEXT ("HAStatus %p\n"), ACE_TEXT ("accept")),
       -1);

  return 0;
}
```

The `init()` method is called when an instance of the service is initialized. Its main purpose is to initialize an `ACE_Acceptor` to listen for service connection requests. It accepts a `-f` option to specify a configuration file to import to learn what TCP port to listen on for service requests. Note that the constructor for `ACE_Get_Opt` uses the `skip_args` parameter (4th) with a value of 0 to force option scanning to begin at the first `argv` token. When an `argc`/`argv` pair is

passed during service initialization, the arguments begin in `argv[0]` instead of `argv[1]` as in a `main()` program.

If the service initialization completes successfully, the `init()` hook should return 0. Otherwise it should return -1 to indicate an error. In an error situation, ACE will remove the service instance.

When a successfully-loaded service is to be shut down, the framework calls the `fini()` hook method:

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

The main responsibility of the `fini()` hook is to perform any cleanup actions necessary for removing the service instance. Upon return from the method, the service object itself will most often be deleted, so `fini()` must insure that all cleanup actions are complete before returning. In our example's case, the acceptor is closed to prevent further service connections from being accepted. Any existing service requests will be allowed to continue, since they have no coupling to the `HAStatus` object that's being destroyed.

Now let's look at how this service gets loaded and initialized. ACE keeps an internal repository of known static services that can be configured. Each static service must insert some bookkeeping information into this repository using some macros that ACE supplies for this purpose. The bookkeeping information for our example service is shown below. It is located in the .cpp file for the service:

```
ACE_FACTORY_DEFINE (ACE_Local_Service, HA_Status)

ACE_STATIC_SVC_DEFINE (HA_Status_Descriptor,
                       ACE_TEXT ("HA_Status_Static_Service"),
                       ACE_SVC_OBJ_T,
                       &ACE_SVC_NAME (HA_Status),
                       ACE_Service_Type::DELETE_THIS |
                         ACE_Service_Type::DELETE_OBJ,
                       0)  // Service not initially active

ACE_STATIC_SVC_REQUIRE (HA_Status_Descriptor)
```

The purposes of these three macros are described below.

1. `ACE_FACTORY_DEFINE(CLS,SERVICE_CLASS)` defines service factory and teardown functions that ACE will use to assist in creating and destroying the service object. `CLS` is the identifier your program/library is using for import/export declarations (see Section 2.5.1 on page 23). This can be the special symbol ACE_Local_Service in cases where there is no need to export the service factory function outside a DLL, as is the case with a static service that's linked into the main program. `SERVICE_CLASS` is the name of the `ACE_Service_Object`-derived class that's instantiated when the service is initialized.
2. `ACE_STATIC_SVC_DEFINE(SERVICE_VAR,NAME, TYPE,FACTORY,FLAGS,ACTIVE)` creates a static object that contains all of the information needed to register the service with the ACE Service Configurator repository. The arguments are:
   - `SERVICE_VAR`: A name for the static object that this macro creates.
   - `NAME`: This is a text string containing the name thge service will be identified as in the service configuration file below. It should not contain whitespace.
   - `TYPE`: This defines the type of object being registered. We'll look at streams in a future section, but whenever you're building a service, use `ACE_SVC_OBJ_T`.
   - `FACTORY`: This is a pointer to the factory function used to create the service object's instance. This is usually formed using the macro `ACE_SVC_NAME (SERVICE_CLASS)` which creates a reference to the factory function defined with `ACE_FACTORY_DEFINE` above.
   - `FLAGS`: These define the disposition of the service-related objects when the service is shut down. Unless you are hand-creating special objects, use `DELETE_THIS` and `DELETE_OBJ`.
   - `ACTIVE`: If this is 1, the service is actived when the program starts but can't be passed any arguments. If 0, the service is initialized by directive from the service configuration file.
3. `ACE_STATIC_SVC_REQUIRE(SERVICE_VAR)` insures that an instance of your service object is created and registered with the ACE Service Configurator when your program starts. `SERVICE_VAR` is the same name used with the `ACE_STATIC_SVC_DEFINE` macro.

Following these steps will insure that your service object is prepared and registered with the Service Configurator framework. However, it will not be acti-

vated or initialized. That step occurs at run time, usually by adding an entry for it to the service configuration file. The format of a line to configure a static service is simply:

```
static service-name [arguments]
```

service-name is the name of your service assigned in the `ACE_STATIC_SVC_DEFINE` macro. When the service configuration file is processed, your service's `init (int argc, ACE_TCHAR *argv[])` method will be called. The *arguments* string will have been separated into tokens before calling the method. The following directive can be used to initialize our example service:

```
static HA_Status_Static_Service "-f status.ini"
```

The following main program is typical of one that loads service(s) and executes a reactor event loop to drive all other program actions.

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

int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  ACE_STATIC_SVC_REGISTER (HA_Status_Descriptor);
  ACE_Service_Config::open
    (argc, argv, ACE_DEFAULT_LOGGER_KEY, 0);

  ACE_Reactor::instance ()->run_reactor_event_loop ();
  return 0;
}
```

Note the `ACE_STATIC_SVC_REGISTER` macro usage. Due to differences in platform handling for static objects, this is sometimes not required (the `ACE_STATIC_SVC_REQUIRE` in the service implementation file is enough). However, for best portability, you should use `ACE_STATIC_SVC_REGISTER` as well. If it's not needed, it's a no-op.

The `ACE_Service_Config::open()` call is the mechanism that actually configures any requested services. The command line options for the main program should be passed to `ACE_Service_Config::open()` to affect the service processing. By default, `open()` attempts to process a file in the current

directory named `svc.conf`. If the static service directive shown above for the example service were in such a file, the service would be initialized at this point.

The 4th parameter to the `open()` method is `ignore_static_svcs`. The default value is 1, so the ACE Service Configurator framework will ignore static services completely. Therefore, if your program will load any static services, you need to either pass 0 as the argument for `ignore_static_svcs` or pass **`-y`** on the command line, which has the same affect. The decision is simply to enable static services at compile time or defer the decision to program start time. The **`-y`** option is just one of those available for altering the behavior of the ACE Service Configurator at run time. Table 19.1 on page 425 lists all of the available options. Note that these options are not passed to individual services' `init()` hooks; they direct the processing of the Service Configurator itself.

Table 19.1. Service Configurator Command Line Options

Option	Meaning
-b	Directs that the program should become a daemon. Note that when this option is used, the process will be daemonized before the service configuration file(s) are read. During daemonization, (on POSIX systems) the current directory will be changed to "/" so the caller should either fully specify the file names, or execute a `chroot()` to the appropriate directory.
-d	Turn on debugging mode. When debugging mode is on, the ACE Service Configurator will log messages describing its actions as they occur.
-f	Specify a service configuration file to replace the default `svc.conf` file. Can be specified multiple times to use multiple files.
-k	Specifies the rendezvous point for the ACE distributed logging system.
-y	Explicitly enables the use of static services, overriding the `ignore_static_svcs` parameter.
-n	Explicitly disables the use of static services, overriding the `ignore_static_svcs` parameter.
-s	Specifies the signal number used to trigger a reprocessing of the configuration file(s). This is ignored on platforms that don't have POSIX signals, such as Windows.
-S	Specifies a service directive string to process. Enclose the string in quotes and escape any embedded quotes with a backslash. This option specifies service directives without the need for a configuration file.

19.3 Setting up Dynamic Services

Dynamic services are those that can be dynamically loaded from a shared library (DLL) when directed at run time. They need not be linked into the main program at all. This dynamic loading capability allows for great flexibility of substitution at run time since the code for hte service need not even be written when the main

program is. Existing services can be removed and new services can be 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:

- 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 all created dynamically at run time 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 that defines the needed import/export specifications.

The actual code for the service is exactly the same as that used in the static service example in this chapter. The record-keeping macros in the dynamic service have been reduced to simply:

```
ACE_FACTORY_DEFINE (HASTATUS, HA_Status)
```

Notice that we use `HASTATUS` in the first argument this time. The macro will expand this to `HASTATUS_Export`, matching the service class's import/export specification. The `ACE_FACTORY_DEFINE` macro generates a short factory function called `_make_HA_Status`. It also generated a function of this name in the static service example, but we didn't need to know it then. We'll need the name in this example, as we'll see shortly.

The main program for this example is also shorter than that used in the static service example:

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

int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  ACE_Service_Config::open (argc, argv);
  ACE_Reactor::instance ()->run_reactor_event_loop ();
  return 0;
}
```

There are no service record-keeping macros here. In fact, there is no trace of any specific service information at all. The set of services to load and use is determined completely outside of the program's code. There's certainly nothing to prevent you from using both static and dynamic services in the same program, but since this example isn't using static services, the arguments to `ACE_Service_Config::open()` that allow them have been left out.

Once again, the Service Configurator framework will attempt to load services as directed in the service configuration file (`svc.conf` by default). The following configuration file directive will load our example dynamic service:

```
dynamic HA_Status_Dynamic_Service Service_Object *
HA_Status:_make_HA_Status() "-f status.ini"
```

Since the record-keeping information is not compiled into the program as with static services, the service configuration directives contain the information needed to direct ACE to create the proper record-keeping information. `Service_Object` indicates a service (as opposed to a stream, which we'll look at soon). `HA_Status:_make_HA_Status()` indicates that a shared library named HA_Status should be dynamically loaded and the `_make_HA_Status()` factory function called to instantiate the service object. This is the factory function generated via the `ACE_FACTORY_DEFINE` macro. The complete syntax definition for configuring a dynamic service is:

```
dynamic ident Service_Object * lib-pathname : object-class [ac-
tive|inactive] [parameters]
```

```
dynamic ident Service_Object * lib-pathname : factory-func() [ac-
tive|inactive] [parameters]
```

Our example used the second variant.

The *`lib-pathname`* token illustrates a very useful feature of ACE with respect to portability. As you may know, different platforms use different conventions for shared library naming. For example, UNIX and Linux platforms prefix the name with `lib` and use a `.so` suffix (e.g. `libHA_Status.so`). Windows doesn't add a prefix, but may decorate the name with a trailing `d` if it's a debug version, and use a `.DLL` suffix (e.g. `HA_Statusd.DLL`). ACE knows how to properly deal with all these naming variants, as well as how to use various library-searching mechanisms appropriate to the run-time platform, such as the `LD_LIBRARY_PATH` environment variable used on many platforms.

Keep in mind that the proper `(PROG)_BUILD_DLL` preprocessor macro needs to be specified at compile time when building a service DLL on Windows. This procedure was explained in Section 2.5.1 but is very important for successfully building a service-containing DLL and bears repeating here.

19.4 Setting up Streams

Configuring streams 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, just as above. Generally 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|inac-
tive] [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 run time. 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 all together.

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 would initiate 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 and then the service object created by the factory function is deleted.

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

We have so far learned how service configuration directives are processed at program startup 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. Since 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 (Table 19.1 on page 425).
2. The program itself can call `ACE_Service_Config::reconfigure()` directly. This is the favored option on Windows (since 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.

The two other directives that can be used to affect a service while it is active are:

```
suspend service
resume service
```

These directives, suspend and resume a service, respectively. Exactly what suspending and resuming a service entails, however, are completely up to the service implementation and it is free to ignore the requests completely. The service implements the suspend and resume operations via two hook methods inherited from `ACE_Service_Object`:

```
virtual int suspend (void);
virtual int resume (void);
```

Appropriate actions to take in these hook methods are service-dependent but may include removing a handler from a reactor or suspending a thread. It's customary

to make the `resume()` hook method "undo" the actions of the `suspend()` hook, resuming normal operations for the service.

19.6 Configuring Services Without svc.conf

In some situations, it may be too restrictive to direct service (re)configuration operations completely 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 remove (const ACE_TCHAR svc_name[]);
```

Chapter 20
Timers

Timers are used in almost all types of software, but are especially useful when writing systems or network software. Most timers are implemented with the assistance of an underlying hardware timer. This hardware timer is accessed through your operating system interface that indicates timer expiration in an OS dependent fashion. Once the hardware timer expires it notifies the operating system that in turn will notify the calling application. The way all of this is achieved varies vastly on different machines/OS.

20.1 Timer Concepts

Most operating systems only provide for 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 time-out that the user is interested in. The head of the queue always contains the next time out 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 time-out value is obtained from the timer queue and set as the next time-out.

ACE has implementations for many of these different types of timer queues. In addition these queues directly support interval timers. These timers expire repeatedly after a specified interval has elapsed. ACE timer queues also offers dispatching facilities that can be used to dispatch specified handlers once an actually time out does occur. ACE is flexible in that you can either use the default event handler hierarchy based on ACE_Event_Handler to handle time-outs, or you can roll your own.

In this chapter we first introduce the timer queue facilities provided by ACE and we will ourselves incorporate them into a crude timer dispatcher. We will then show you some of the pre-built timer dispatchers that are a part of ACE and you can use directly in your applications. Finally we will change the event handler hierarchy to build our own private call back classes independent of ACE_Event_Handler.

20.2 Timer Queues

Let's start off with an introduction to the various timer queues that ACE provides. All ACE based timer queues derive from the abstract base class ACE_Timer_Queue. This means that they are run-time substitutable with one another as illustrated in Figure 20.1.

Each concrete timer queue uses a different algorithm for maintaining timing nodes in order each with different time complexity characteristics. These characteristics should drive your decision in moving from one timer queue mechanism to another one. Further, if none of ACE pre-built timers satisfy your requirements you can sub-class and create your own timer queue.

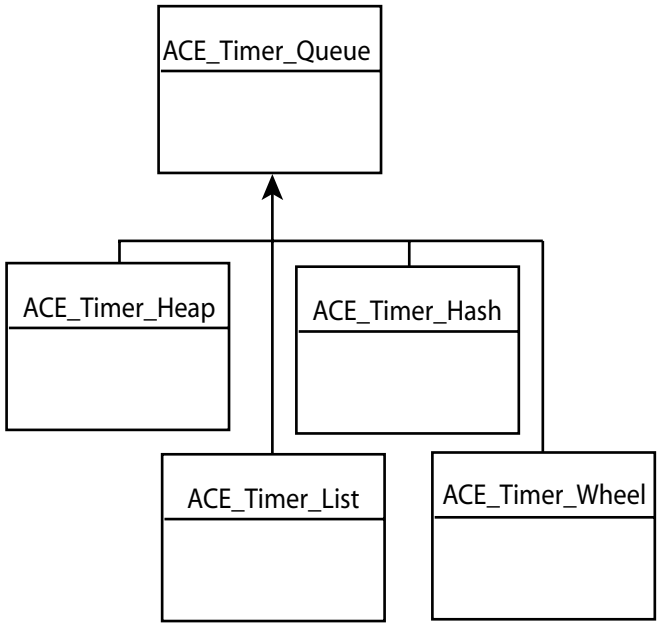


Figure 20.1. Timer Queue Class Hierarchy

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_timer()= O(lg n) cancel_timer()= O(lgn) current timer find= O(1)
ACE_Timer_List	The timers are stored in a doubly linked list	schedule_timer()= O(n) cancel_timer()= O(1) current timer find= O(1)
ACE_Timer_Hash	This structure used in this case is a variation on the timer wheel algorithm. The performance is highly dependent on the hashing function used.	schedule_timer()= Worst = O(n) Best = O(1) cancel_timer()= O(1) current timer find= O(1)
ACE_Timer_Wheel	The timers are stored in an array of "pointers to arrays" where each array being pointed to is sorted.	schedule_timer()= Worst =O(n) cancel_timer()= O(1) current timer find= O(1)

ACE_Timer_Heap avoids expensive memory allocation by pre-allocating 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 setup to allocate memory from a free list of timer nodes, the free list can be provided by the programmer.

20.2.1 Creating a Timer Dispatcher

In the following example we create a "timer dispatcher" (TimerDispatcher) that we use to register event handlers (CB) that are called when scheduled timers expire. The timer dispatcher contains two parts;

- An ACE Timer Queue

- A Timer Driver

We use the term “Timer Driver” to mean the mechanism that instigates or causes the indication of a time-out event. You can use various schemes for timer indication, including your private OS timer API, timed condition variables, timed semaphores, timed event loop mechanisms etc. In our example we use an ACE_Event object to “drive” the timer, which is illustrated in Figure 20.2.
ACE provides several pre-built timer dispatchers that are directly tied to OS timer API’s. These dispatchers can be used “out-of-the-box” to schedule and cancel timers. We have already seen one such general-purpose timer dispatcher, the ACE_Reactor. We will go over these pre-built dispatchers in greater detail later in this chapter.

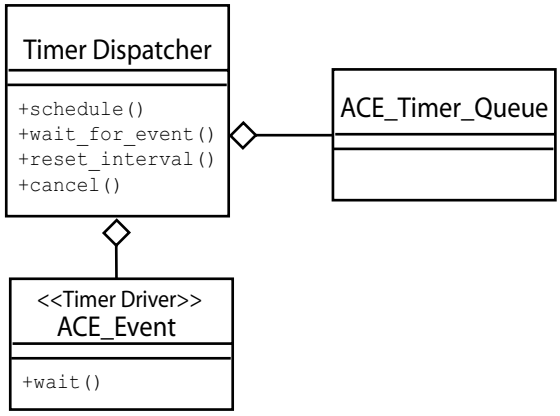


Figure 20.2. Timer Dispatcher Example Class Diagram

```
class Timer_Dispatcher
{
public:
  void wait_for_event();

  long schedule (EventHandler* cb, void * arg,
                 const ACE_Time_Value &abs_time,
                 const ACE_Time_Value &interval);

  int cancel (EventHandler* cb,
              int dont_call_handle_close = 1);
```

```
  int reset_interval(long timer_id,
                     const ACE_Time_Value &interval);

  void set (TimerQueue* timer_queue);

private:
  TimerQueue* timer_queue_;
  ACE_Event timer_;
};
```

The dispatcher allows the user to register a timer-event listener object that will be called back when a timer expires. This is done through the schedule() API. This method returns an identifier to the newly scheduled timer that can be used to change or cancel the timer. The event listener class must conform to the ACE event handling scheme i.e., it must be a sub-type of the ACE_Event_Handler abstract class and implement the handle_timeout() method.

A user can cancel a previously scheduled timer event through the cancel() method of the timer event dispatcher and reset the interval of an interval timer using the reset_interval() method. Both these methods uses the timer id returned by the schedule() method.

After scheduling the appropriate timers the dispatcher's wait_for_event() method must be called. This causes the dispatcher to wait for the underlying ACE_Event class to generate a timer event. Once this event occurs the dispatcher proceeds to callback all registered event handlers.

We create a singleton called Timer that makes our dispatcher globally visible within the application.

```
void Timer_Dispatcher::wait_for_event()
{
  ACE_TRACE("Timer_Dispatcher::wait_for_event");

  while(1)
    {
      ACE_Time_Value max_tv =
          timer_queue_->gettimeofday ();

      ACE_Time_Value *this_timeout
          = this->timer_queue_->calculate_timeout (&max_tv);

      if(*this_timeout == ACE_Time_Value::zero)
          this->timer_queue_->expire();
```

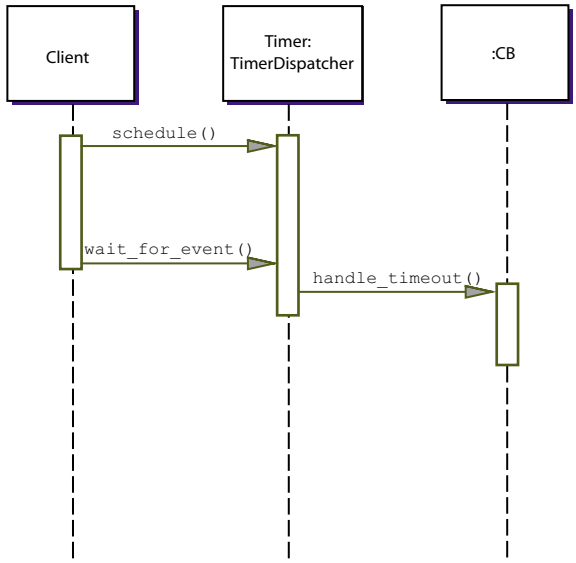


Figure 20.3. Timer Dispatcher Example Sequence Diagram

```
      else
      {
          ACE_Time_Value next_timeout = timer_queue_->gettimeofday
();
          next_timeout+= *this_timeout;
        //convert to absolute time.

        if(this->timer_.wait(&next_timeout) == -1 )
          this->timer_queue_->expire();
      }
    }
}
```

The wait_for_event() method uses the calculate_timeout() method on the timer queue to determine the next timer that is set to expire. The time value max_tv that is passed in to calculate_timeout() is compared to the next time-out on the queue and the greater of the two is returned. We pass this greater time-out, to the underlying timer, as the time for the next expiration. This behavior can be

used to allow for a maximum blocking time for the wait_for_event() method, similar to the time-out specified on the ACE_Reactor::handle_events() method.

The actual blocking call is made on the ACE_Event class that is acting as our timer. If a time-out does occur, the wait() call on ACE_Event returns –1. When this happens, we call expire() on the timer queue, that consequently calls handle_timeout() on all event handlers that were to expire at time < next_timeout. expire() returns the number of handlers that were actually dispatched.

```
long
Timer_Dispatcher::schedule(EventHandler* cb, void * arg,
                           const ACE_Time_Value &abs_time,
                     const ACE_Time_Value &interval)
{
  ACE_TRACE ("Timer_Dispatcher::schedule_timer");

  return this->timer_queue_->schedule(cb, arg,
                                      abs_time,
                                      interval);
}
```

The dispatchers schedule() method forwards the call to schedule() on the underlying timer queue. When a scheduled timer expires, the handle_timeout() method of cb is called with the argument arg. The time-out occurs in abstime expressed as an absolute time and not relative time (ACE users commonly make the mistake of assuming relative time instead of absolute time). If the value of interval is anything but ACE_Time_Value::zero then after the initial timer expiry, the timer will continue to expire at this interval over and over again. As we mentioned earlier this method returns a long timer id that is used in subsequent timer management calls, such as cancel() or reset_interval().

```
int
Timer_Dispatcher::cancel(EventHandler * cb,
                         int dont_call_handle_close)
{
  ACE_TRACE("Timer_Dispatcher::cancel");

  return timer_queue_->cancel(cb, dont_call_handle_close);
}
```

The cancel method causes all timers that are being handled by the cb callback object to be cancelled. If the value of dont_call_handle_close is 1 then the handle_close() method of cb is not automatically called back.

```
class CB :
  public ACE_Event_Handler
{
public:
  CB();

  void setID(long timerID);
  //set the timer id that is being handled
  //by this instance.

  long getID();
  //get the timer id

  virtual int handle_timeout(const ACE_Time_Value &tv, const void
*arg);
  //actually handle the timeout.

  virtual int handle_close(ACE_HANDLE handle,
                           ACE_Reactor_Mask close_mask);

private:
  long timerID_;
  int count_;
};
```

Finally, we get to the callback handler whose handle_timeout() method is called back when a timer expires. The callback method is passed two arguments; the absolute time at which the timer expired and a void pointer that was passed in through the dispatchers schedule() API.

```
int CB::handle_timeout(const ACE_Time_Value &tv, const void *arg)
{
  ACE_TRACE("CB::handle_timeout");

  int *val = ACE_static_cast(int*, arg);

  ACE_ASSERT((*val) == timerID_);


  ACE_DEBUG((LM_DEBUG, "Timer %d expiry being handled by thread id
 %t \n",
        timerID_));

  if(count_ == 5)
  {
```

```
        ACE_DEBUG((LM_DEBUG, "Reseting time interval for %d\n",
                timerID_));
        ACE_Time_Value interval(0L, 1000L);
        //new interval is 10 ms.

        ACE_ASSERT(Timer::instance()->reset_interval(timerID_, int
erval)!=-1);
  }
  if(count_++ == 10)
  {
        ACE_DEBUG((LM_DEBUG, "Canceling %d\n", timerID_));

        ACE_ASSERT(Timer::instance()->cancel(this) !=0);
  }
  //cancel this timer

  return 0;
}
```

In the handle_timeout() method we first assert that the argument passed in matches the identifier stored by the call back object. If this particular callback has been called more then 5 times we uses the dispatcher singleton, Timer, to reset the time interval for the interval timer. Similarly, if the handler has been called more then 10 times we cancel all timers that are handled by this event handler.

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

#include "ace/Timer_Heap.h"
#include "ace/Timer_Wheel.h"
#include "ace/Timer_Hash.h"
#include "ace/Timer_List.h"

#include "CB.h"
#include "TimerDispatcher.h"

int main(int argc, char *argv[])
{
  ACE_Timer_Queue* timer_queue;

#if defined(HEAP)
  ACE_NEW_RETURN(timer_queue, ACE_Timer_Heap, -1);
#elsif defined(HASH)
  ACE_NEW_RETURN(timer_queue, ACE_Timer_Hash, -1);
#elsif defined(WHEEL)
```

```
  ACE_NEW_RETURN(timer_queue, ACE_Timer_Wheel, -1);
#else
  ACE_NEW_RETURN(timer_queue, ACE_Timer_List, -1);
#endif

  Timer::instance()->set(timer_queue);
  //setup the timer queue

  CB cb[10];
  long args[10];
  for(int i=0; i < 10 ; i++)
  {
     long timerID = Timer::instance()->schedule(&cb[i], &args[i],
                timer_queue->gettimeofday() + (ACE_Time_Value)5, i
);

     cb[i].setID(timerID);
     //set the timerID state variable of the handler

     args[i] = timerID;
     //implicitly send the handler it's timer id
  }

  Timer::instance()->wait_for_event();
  //"run" the timer..

  return 0;
}
```

Since ACE provides many different types of concrete timer queues we decided to take them all out for a spin in this example. By defining the appropriate pre-processor symbol you can setup the timer dispatcher with an appropriate timer queue.

Once the timer dispatcher is setup we create 10 callback objects that we schedule to handle 10 different timers. We then perform a blocking call on the wait_for_event() method of the timer dispatcher.

20.3 Prebuilt Dispatchers

In the previous section we described at some length how you can create your own timer dispatcher. In this section we are going to give you an overview of some of the pre-built timer dispatchers that are a part of ACE

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.
This next example illustrates the use of an active timer where 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 Active Timer 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("CB::handle_timeout");

    int *val = ACE_static_cast(int*, arg);

    ACE_ASSERT((*val) == id_);

    ACE_DEBUG((LM_DEBUG, "Timer expiry being handled by thread id
```

```
%t \n"));
    return 0;
  }
private:
  int id_;
};
```

We start by creating a useful timer callback handler. Since the timer dispatcher is active, the handle_timeout() method, of the event handler, will be dispatched using the active timers private thread of control. We display this using the ACE_DEBUG() macros %t format specifier.

```
int main(int argc, char *argv[])
{
  ACE_DEBUG((LM_DEBUG, "the main thread %t has started \n"));

  ActiveTimer atimer;
  //Create an "active" timer.

  CB cb1(1);
  CB cb2(2);
  int arg1 = 1;
  int arg2 = 2;

  atimer.activate();
  //start the timer handling thread

  const ACE_Time_Value curr_tv = ACE_OS::gettimeofday();
  ACE_Time_Value interval = ACE_Time_Value(1, 1000);
  long tid1 = atimer.schedule(&cb1,&arg1,
        curr_tv + ACE_Time_Value(3L), interval);
  long tid2 = atimer.schedule(&cb2,&arg2,
        curr_tv + ACE_Time_Value(4L), interval);
  //schedule a timer to go off 4 seconds from now
  //with a time interval of 1.1 seconds

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

  return 0;
}
```

We start the program by creating and then activating the active timer. The activate() method of the timer queue actually gets the timer dispatchers private thread running.
After starting the dispatcher we schedule() a couple of timers and block the main thread on the ACE_Thread_Manager::wait() method. Note that in the case of the active timer dispatcher we did not have to run any event loop and the main thread was free to perform other functions.

20.3.2 Signal Timers

Another ACE provided timer dispatcher ACE_Async_Timer_Queue_Adapter<> uses UNIX signals to indicate timer expirations. This timer queue only works on platforms that support the interval timer signals. These signals are not supported on Win32.
As we discussed earlier in the book, signals are asynchronous software interrupts that are handled by using any of the application threads to handle the interrupt. This allows for asynchronous behavior in single threaded applications

```
#include "ace/Timer_Queue_Adapters.h"
#include "ace/Timer_Heap.h"

typedef ACE_Async_Timer_Queue_Adapter<ACE_Timer_Heap> Timer;
```

We start off by creating an asynch. timer dispatcher, specifying the underlying timer queue as ACE_Timer_Heap.

```
int main(int argc, char *argv[])
{

  Timer timer;
  //Create the timer such that it
  //blocks all signals when it goes off.


  CB cb(1);
  int arg = 1;
  timer.schedule(&cb, &arg, ACE_OS::gettimeofday() + (ACE_Time_Val
ue)5, 4);
  //schedule a timer to go off two seconds later
  //and then after every 4 seconds.
```

```
  while(1)
    {
      ACE_OS::sleep(2);
    }
  //don't let the main thread exit.

}
```

We run the example by creating the timer dispatcher and scheduling a timer. We then block the main thread of control on ACE_OS::sleep(). When a timer goes off a signal is raised and the handle_timeout() method is called back in signal context. Once the signal handler returns we once again block on the ACE_OS::sleep() call.

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 pre-existing 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.

20.4.1 Timer Queue Template Classes

Underneath the covers of the ACE timer queues that we have been using in all our previous examples, lie a couple of template based timer queue classes. These template classes allow the template instantiator to specify

- the type of callback handler that will be called back when a timer expires
- the up call manager that calls methods on the callback handler when events such as timer expiration or cancellation occur.
- A lock to ensure thread safety of the underlying timer queue

In fact, the ACE_Timer_Queue class is created by instantiating the ACE_Timer_Queue_T template type with the ACE_Event_Handler class as the callback handler type and an ACE internal up call handler class as the up call manager.

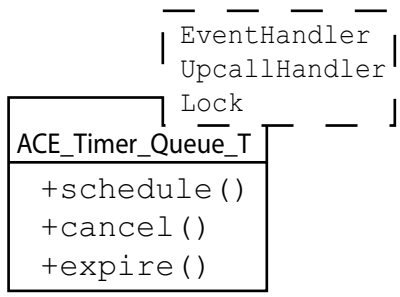


Figure 20.4. Timer Queue Template Classes

How the up call handler works

When an action is performed on any of the ACE timer queue classes a method of the up call manager is automatically called. This provides you, the application programmer, a hook to decide what should occur when the action occurs. This allows you to set up your own up call manager to call back your specific event handler class methods.

The sequence diagram below, illustrates how the timer queue, up call manager and a private callback class (PCB) would interoperate. When expire() is called the timeout() method of the supplied up call handler is invoked. In this case we call handleEvent() on our new call back class, whereas the ACE provided up call handler would have invoked handle_timeout(). Similarly when cancel() is invoked by the timer queue client, cancellation() is called on the up call handler and when the timer queue is deleted deletion() is called on the up call handler.

Lets' step through an example of creating our own up call handler. We start off by defining our event handler class.

```
class PCB
{
public:
  PCB();

  void setID(long timerID);
  //set the timer id that is being handled
  //by this instance.

  long getID();
  //get the timer id
```

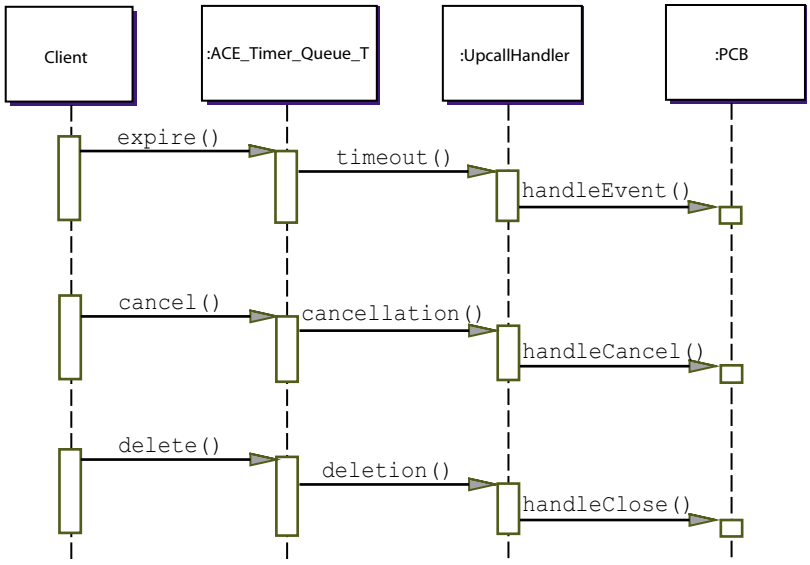


Figure 20.5. Upcall Handler Sequence Diagram

```
  virtual int handleEvent(const void *arg);
  //actually handle the timeout.

  virtual int handleCancel();
  //handle cancelation

  virtual int handleClose();
  //handle the close of the timeout

private:
  long timerID_;
  int count_;
};
```

This event handler has one handleEvent() method that we want to be called when a timer expires. Similarly we want handleCancel() to be called on timer expiration and handleClose() to be called when the timer queue is deleted (but the

handler is still alive). Each PCB is associated with a single timer the ID of which is stored within the PCB as timerID_.

After defining our call back class we move on to the up call handler.

```
class UpcallHandler;
//handle the upcalls here

typedef ACE_Timer_Queue_T<PCB*, UpcallHandler, ACE_Null_Mutex>
      PTimerQueue;

typedef ACE_Timer_Heap_T<PCB*, UpcallHandler, ACE_Null_Mutex>
      PTimerHeap;
//Create a special heap based timer that allows you
//to control exactly how timer events are handled.
```

First we need to instantiate the provided ACE timer queue templates with the appropriate parameters. We start off by creating the abstract base class PTimer-Queue and then create a single heap based concrete class PTimerHeap. By instantiating these templates we automatically create the base, derived relationship between these classes.

```
int
UpcallHandler::timeout (PTimerQueue& timer_queue,
                        PCB * handler,
                        const void * arg,
                        const ACE_Time_Value &cur_time)
{
    ACE_TRACE("UpcallHandler::timeout");

    return (*handler).handleEvent(arg);
    //call the functor
}

int
UpcallHandler::cancellation (PTimerQueue &timer_queue,
                                  PCB *handler)
{
    ACE_TRACE("UpcallHandler::cancellation");

    ACE_DEBUG((LM_DEBUG, "Handler %d has been cancelled\n",
        handler->getID()));

    return handler->handleCancel();
}
```

```
// This method is called when the timer is canceled
int
UpcallHandler::deletion (PTimerQueue &timer_queue,
                         PCB  *handler,
                         const void *arg)
{
    ACE_TRACE("UpcallHandler::deletion");

    ACE_DEBUG((LM_DEBUG, "Handler %d has been deleted\n",
        handler->getID()));

    return handler->handleClose();
}
```

When expire() is called on the new timer queue type that we have created (PTimerQueue) it knows that it needs to call our UpcallHandler handlers timeout(). We have set our upcall handler so that it delegates off to our own PCB class, specifically invoking the handleEvent() method each time the timer queue informs us of a timeout. We also pass back the args (asynchronous completion token) that were passed to the timer queue when this timer was scheduled. Notice that we ignore the cur_time value.

The cancellation() and deletion() methods are also called back at the appropriate times and once again we delegate the call of to handleCancel() and handleClose() as appropriate.

```
int main(int argc, char *argv[])
{
  PCB cb1, cb2;
  cb1.setID(1);
  cb2.setID(2);
  int arg1=1, arg2=2;

  PTimerQueue* timerQueue;
  ACE_NEW_RETURN(timerQueue, PTimerHeap(), -1);
  Timer::instance()->set(timerQueue);

  ACE_Time_Value tv = ACE_OS::gettimeofday();
  tv+=20L;

  Timer::instance()->schedule(&cb1, &arg1, tv, 1);
  Timer::instance()->schedule(&cb2, &arg2, tv, 2);
  //schedule two different timers to go off
```

```
  Timer::instance()->wait_for_event();
  //run the timer event loop forever.

  return 0;
}
```

Finally we get to the main program that creates two call back objects and schedules timers to go off with a timer dispatcher Timer. This dispatcher is similar to the one we used in previous sections but has been created to use an underlying PTimerQueue instead of the ACE_Timer_Queue. To setup the underlying queue for the dispatcher we have to pass in a concrete sub-class of PTimerQueue, in this case PTimerHash.

20.5 Summary

In this chapter we started 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 up call 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

21.1 ACE Naming Service Overview

The ACE_Naming_Service provides your application with a persistent key/value mapping mechanism. We can, for instance, use this to create a server-to-address mapping much like DNS but tailored to our application's needs. The naming service can exist for a single process, all processes on a single node or for many processes on a network of nodes.

21.2 The ACE_Naming_Context

The ACE_Naming_Context is the center of the Naming Serivce 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.

There are five things one will typically do with a naming context instance:

Table 21.1. Naming Context Actions

bind / rebind	Add/Replace a value to/in the context
unbind	Remove an entry from the context
resolve	Find a value in the context
list scalars	Fetch a list of names, values or types from the context
list entries	Fetch a list of name/value/type bindings from the context

In addition, you can also fetch and set the ACE_Name_Options instance used to configure the naming context instance. With the name options instance you can set various behaviors of the naming context such as:

Table 21.2. Name Options Attributes

TCP/IP Port	The TCP/IP port at which a client of a network mode naming service will connect
hostname	The hostname 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 filesystem where the persistent namespace data is kept
process name	The name of the current process. In process-local mode this is 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 Win32 environment can choose to use the Win32 registry

The naming service supports three contexts from which your application can choose:

- Process-only access
- System (or node) -only access
- Network-wide access

The first two options are essentially the same, differing only in your convention for accessing the persistent database of key/value pairs. When using the PROC_LOCAL context only applications of the same name (e.g. - argv[0] value) should access a particular database. When using the NODE_LOCAL context, any application on the local system is permitted access to the database.

The NET_LOCAL context requires you execute a server somewhere on your network. Clients can then query this server to set or get key/value pairs. This would be the most appropriate architecture for our hypothetical DNS lookalike.

Note that access to the persistent data may be implmented using a memory-mapped file. Attempting to access a network-mounted database in a local context could have unpleasant results. If you need multiple hosts accessing the same database, use the NET_LOCAL context.

One of the best features of the naming service is that it presents exactly the same API to your application regardless of which context you choose. This means that as your application expands you can switch from process to node to network mode simply by changing one line of code.

21.3 PROC_LOCAL - A single-process naming context

In our first example we will use the PROC_LOCAL context. The intention of this context is for a single application to access the database. By “single application” we mean a named application (e.g. - argv[0]), not necessarily a single instance of an application (e.g. - pid).

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.

21.3.1 main()

The first thing our application will do is create a helper object to manage the command line options. This isn't strictly necessary but it is a handy way to delegate that work to another object and keep *main()* clear of clutter:

```
int main( int argc, char ** argv )
{
    Temperature_Monitor_Options opt(argc, argv);
```

With that out of the way we can proceed to 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 that provides a few handy methods. We'll come to those details in a moment when we discuss the Naming_Context in detail.

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 name 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 provide 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();
```

Although the *monitor()* method is designed to execute forever, we still provide cleanup code for the naming context instance. A future version of the application may provide some sort of clean shutdown mechanism that would cause *monitor()* to exit and, if so, we won't have to worry about the context's cleanup.

```
    naming_context.close();

    return 0;
}
```

21.3.2 Naming_Context and Name_Binding

In the previous section we learned how to create and initialize a naming service instance. Table 21.1. lists the common things you will do with the naming context. Because the naming context is your primary interface to the ACE Naming Service it makes sense to extend the ACE_Naming_Context object to the specific needs of your application. In the example we will be storing temperatures and times (of failures and resets). To simplify our application code, therefore, our ACE_Naming_Context extension provides methods for storing (binding) these datatypes in the naming service database.

```
class Naming_Context : public ACE_Naming_Context
{
public:
    typedef ACE_Naming_Context inherited;

    int rebind (const char *name_in,
                const char *value_in,
                const char *type_in = "" )
    {
        return this->inherited::rebind(name_in,
                                       value_in,
                                       type_in);
    }

    int rebind (const char *name_in,
                float value_in,
                const char *type_in = "" )
    {
        char buf[BUFSIZ];
        ACE_OS::sprintf(buf, "%2f", value_in );
        return this->inherited::rebind(name_in,
                                       (const char *)buf,
                                       type_in);
    }

    int rebind (const char *name_in,
                int value_in,
                const char *type_in = "" )
    {
        char buf[BUFSIZ];
        ACE_OS::sprintf(buf, "%d", value_in );
        return this->inherited::rebind(name_in,
                                       (const char *)buf,
                                       type_in);
    }
```

The naming service database ultimately stores its keys and values as wide (two-byte) character strings[2]. If we have any other kind of information to store we must be able to represent it this way. Our helper functions simply use the *sprintf()*

2. The optional *type* is always stored as a narrow (one-byte) character string.

function to convert float (temperature) and int (time) values into a string before invoking the baseclass *rebind()* method.

Before we continue it is important to take a moment and discuss the difference between *bind()* and *rebind*(). The *bind()* method will only put a value into the database if there exists no value at the current key location. On success, *bind()* will return 0; on failure (such as "key already exists") *bind()* will return a non-zero value.

The *rebind()* method on the other hand will add a value if the key doesn't exist or replace an existing value if the key does exist. Like *bind()*, it will return zero on success and non-zero on failure but an attempt to add a value to an existing key will not be considered a failure.

Whether you use *bind()* or *rebind()* in your application is entirely up to you and the needs of your application. In our temperature monitoring scenario it is appropriate to replace any existing values so we use *rebind()* in all cases.

Returning to our Naming_Service object, we have one more value-added feature:

```
    Name_Binding * fetch( const char * name )
    {
        ACE_NS_WString value;
        char * type;

        if( this->resolve( name, value, type ) != 0 ||
            value.length() < 1)
        {
            return 0;
        }

        Name_Binding * rval =
            new Name_Binding(
                ACE_NS_WString(name),
                value,
                type);

        return rval;
    }
};
```

When you know the key (known as the *name* in naming service parlance) of an item stored in the naming context you will use the *resolve()* method to fetch the

value and type (if any). You do this by passing a reference to an ACE_WString and a reference to a character pointer[3]. If the requested key (name) exists, the ACE_Naming_Context will allocate space at the references you provide to contain the value and type. It is up to you to remember to deallocate this space so as not to cause a memory leak.

To avoid the need to remember to delete these instances and to generally make the example code easier to write, read and maintain, we've created a Name_Binding object to contain the resutls of a successful *resolve()* invocation. When the Name_Binding instance is destroyed it will take care of freeing the memory allocated by the *resolve()* call.

The name was chosen carefully because there actually is an ACE_Name_Binding used by the naming service to contain key/value/type tuples. Since our Name_Binding object mirrors this, the name "Name_Binding" made sense.

```
class Name_Binding
{
public:
    Name_Binding(ACE_Name_Binding * entry)
    {
        this->name_ = entry->name_.char_rep();
        this->value_ = entry->value_.char_rep();
        this->type_ = ACE_OS::strdup(entry->type_);
    }

    Name_Binding (const ACE_NS_WString &n,
                  const ACE_NS_WString &v,
                  const char *t)
    {
        this->name_ = n.char_rep();
        this->value_ = v.char_rep();
        this->type_ = ACE_OS::strdup(t);
    }

    ~Name_Binding()
    {
        delete this->name_;
        delete this->value_;
        delete this->type_;
```

3. You can also provide *resolve()* with a character pointer for the *value* parameter if you know that's what was stored in the naming service in the first place.

```
    }

    char * name() { return this->name_; }

    char * value() { return this->value_;}

    const char * type() { return this->type_;}

    int int_value() {
        return ACE_OS::atoi(this->value());
    }

private:
    char * name_;
    char * value_;
    const char * type_;
};

typedef auto_ptr<Name_Binding> Name_Binding_Ptr;
```

To convert an ACE_WString to a character pointer (which is what our application will ultimately want) you must use the *char_rep()* method which allocates memory. To keep things as straight-forward as possible we let that happen in the constructor and at the same time use *strdup()* to create a copy of the type. Our destructor can then delete all of the member variables safely with no special cases.

As our example grows in later secitons we will find a need to do the same memory mangement for an actual ACE_Name_Binding instance. A constructor accepting an ACE_Name_Binding that copies that binding's values is just the thing to handle this situation.

The *int_value()* method is a simple wrapper around the standard *atoi()* function. This isn't strictly necessary but it makes certain bits of our application code a little easier to read.

Finally, remember that *Name_Binding::fetch()* creates a new instance of Name_Binding. Somebody has to remember to free that instance so that the Name_Binding's destructor will free up the key/value/type tuple. The auto_ptr<> template makes it easy for us to create smart pointers that will these things for us.

21.3.3 The Temperature Monitor

Now that we have our Naming_Context and Name_Binding objects defined we can get into the core of the application and investigate the Temperature_Monitor

object. Temperature_Monitor will use both of these custom objects to interact with the underlying ACE_Naming_Context for managing the temperature and thermometer status.

The constructor accepts instances of Temperature_Monitor_Options and Naming_Context. The former provides us with runtime behavior configuration, while the latter is the access to our persistent name/value pairs:

```
Temperature_Monitor::Temperature_Monitor(
    Temperature_Monitor_Options & opt,
    Naming_Context & naming_context )
        : opt_(opt), naming_context_(naming_context)
{
}
```

We saw this in *main()* just before invocation of the *monitor()* method:

```
void Temperature_Monitor::monitor()
{
    this->thermometer_ =
        new Thermometer(this->opt_.thermometer_address());

    for(;;)
    {
        float temp = this->thermometer_->temperature();

        ACE_DEBUG ((LM_INFO, "Read temperature %.2f\n", temp ));

        if( temp >= 0)
        {
            this->record_temperature(temp);
        }
        else
        {
            this->record_failure();
        }

        ACE_OS::sleep( this->opt_.poll_interval() );
    }

    delete this->thermometer_;
}
```

monitor() is responsible for polling the physical thermometer at a periodic interval. If the physical device returns success, we log the value. If the temperature query fails we log that also and possibly reset it.

In *record_temperature()* method we finally begin interacting with the naming service itself. Our first task is to fetch any previous record of the current temperature:

```
void Temperature_Monitor::record_temperature(float temp)
{
    Name_Binding_Ptr current(
        this->naming_context_.fetch("current") );

    if( current.get() )
    {
        this->naming_context_.rebind( "previous",
                                      current->value() );
    }
```

Recall that *fetch()* is one of our "value added" methods in the Naming_Context extension of ACE_Naming_Context. Internally, *fetch()* invoke's the baeclass method *resolve()*. If *resolve()* locates the named value ("current" in this case) it will return a pointer to a new Name_Binding instance. By "wrapping" this pointer with a stack instance of Name_Binding_Ptr (a typedef based on auto_ptr<>) we can use the auto_ptr's behavior to ensure that the Name_Binding pointer is deleted when the *record_temperature()* method exits.

An auto_ptr's *get()* method will return the pointer it has ownership of. If that value is non-zero then we know that *fetch()* was successful and we can rebind the current temperature under the new name "previous". The name binding's *value()* method will return a character pointer to the data associated with the name we queried for. That can then be given directly to the *rebind()* method of the naming context. If a value already exists there it will be replaced, if not then the new value will be added.

Our next action is to save the new current value. We do that with another call to *rebind()*, this time overwritting the previous "current" value:

```
    this->naming_context_.rebind( "current", temp );
```

Finally, we clear out the various reset variables. After all, we're recording a successful temperature fetch so it doesn't make sense to have failure state information at this time.

```
    this->naming_context_.unbind( "lastReset" );
    this->naming_context_.unbind( "resetCount" );
}
```

To clear out these values we use the *unbind()* method. This works much like a database's *delete* keyword. Any subsequent attempt to *resolve()* the now-unbound name will result in a failure.

Now we take a moment to look at the *record_failure()* method which is invoked when *monitor()* fails to fetch a temperature from the thermometer device. The first thing here is to fetch the current failure status if any:

```
void Temperature_Monitor::record_failure()
{
    Name_Binding_Ptr lastReset(
        this->naming_context_.fetch("lastReset") );
    Name_Binding_Ptr resetCount(
        this->naming_context_.fetch("resetCount") );
```

As with *record_temperature()* we use the *fetch()* method to locate each named value we're interested in and wrap those results in on-the-stack Name_Binding_Ptr instances.

If a previous reset time was recorded we use the *int_value()* method of our extended name binding instance to find out when that reset took place. If there was no previous reset then we dummy-up the value as "right now".

```
    int now = ACE_OS::time();

    int lastResetTime;

    if( lastReset.get() )
    {
        lastResetTime = lastReset->int_value();
    }
    else
```

```
    {
        this->naming_context_.rebind( "lastReset", now );
        lastResetTime = now;
    }
```

We then compare the *lastResetTime* to the current time. Once this delta reaches some reasonable limit we reset the physical device:

```
    if( now - lastResetTime > this->opt_.reset_interval() )
    {
        this->reset_device(resetCount);
    }
}
```

Thus we're done with the *record_failure()* method and we move on to the *reset_device()* method:

```
void Temperature_Monitor::reset_device(
    Name_Binding_Ptr & resetCount)
{
    int number_of_resets = 1;

    if( resetCount.get() )
    {
        number_of_resets = resetCount->int_value() + 1;

        if( number_of_resets > this->opt_.excessive_resets() )
        {
            // ...
        }
    }

    this->thermometer_->reset();

    this->naming_context_.rebind( "lastReset",
                                  (int) ACE_OS::time() );

    this->naming_context_.rebind( "resetCount",
                                  number_of_resets );
}
```

The purpose of *reset_device()* is, unsurprisingly, to reset the physical device. If too many consecutive resets have been done then we may want to notify

someone of the situation. After the reset action we record the reset time and current “consecutive resets” count using another pair of *rebind()* methods.

The Temperature_Monitor object is the workhorse of our simple little application. Its job is to monitor a physical device and record its current “state” in the form of the current temperature or failure status. We’ve used the naming service to persist this information between executions of the application. This is particularly important with the failure data.

21.4 NODE_LOCAL Mode - Sharing a Naming Context

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 will modify the previous application to write the ten most recent successful results to a NODE_LOCAL naming service. A second application will then poll this naming context periodically to create a graph of the temperature history.

21.4.1 Saving shared data

We begin with a new version of *main()* where we create a second naming context instance in which we’ll store the shared information.

```
int main( int argc, char ** 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 );
        char * nargv[] = { argv[0] };
        name_options->parse_args(sizeof(nargv) / sizeof(char*) ,
                                 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, 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
") );

    if( current.get() )
    {
        this->naming_context_.rebind( "previous",
                                      current->value() );
    }

    this->record_history(temp);
```

```
    this->naming_context_.unbind( "lastFailure" );
    this->naming_context_.unbind( "lastReset" );
    this->naming_context_.unbind( "resetCount" );
}
```

As before, we save the previous value and store the current one as well as clearing the failure flags. Hidding in between is a call to *record_history()* where we'll use the shared context:

```
void Temperature_Monitor2::record_history(float temp)
{
    int now = (int) ACE_OS::time();

    this->shared_context_.rebind( "lastUpdate", now );

    Name_Binding_Ptr counter( this->shared_context_.fetch("counter
") );
    int counterValue = counter.get() ? counter->int_value() : 0;

    char name[BUFSIZ];
    ACE_OS::sprintf(name, "history[%d]", counterValue );

    char value[BUFSIZ];
    ACE_OS::sprintf(value, "%d|%.2f", now, temp );

    this->shared_context_.rebind( name, value );

    counterValue = ++counterValue % this->opt_.history_size();

    this->shared_context_.rebind( "counter", counterValue );
}
```

As you can see, storing data to the shared context is no different than storing it to the local context. The naming context keys are simple text strings so we build a clever string of the format *history[n]* to store our ten (or so) most recent temperatures. Likewise, the values we store must also be strings so we use the format *date*|*temperature* to store not just the temperature but also the date and time (in epoch seconds) at which the temperature was stored. Our cooperating application will be aware of these two clever ideas and use them to extract the data we've stored.

The remainder of the second example is identical to the first. In particular, the failure and reset logic work just as before. We now move on to look at our peer application which will read and process the temperature history.

21.4.2 Reading shared data

As you might expect by now, the *main()* of the temperature graphing application is quite similar to the *main()* of the temperature collector:

```
int main( int argc, char ** argv )
{
    Temperature_Grapher_Options opt(argc, argv);

    Naming_Context naming_context;
    ACE_Name_Options * name_options =
      naming_context.name_options();
    name_options->process_name( argv[0] );
    name_options->context( ACE_Naming_Context::NODE_LOCAL );
    naming_context.open( name_options->context() );

    Temperature_Grapher grapher(opt, naming_context);

    grapher.monitor();

    naming_context.close();

    return 0;
}
```

The grapher uses a simple *monitor()* method to poll the shared naming context from time to time:

```
void Temperature_Grapher::monitor()
{
    for(;;)
    {
        this->update_graph();

        ACE_OS::sleep( this->opt_.poll_interval() );
    }
}
```

update_graph() begins by checking to see that the shared naming context has a *lastUpdate* entry. If there is no such entry then the temperature monitor has not yet successfully queried the thermometer.

```
void Temperature_Grapher::update_graph()
{
    Name_Binding_Ptr lastUpdate( this->naming_context_.fetch("last
Update") );

    if( !lastUpdate.get() )
    {
        ACE_DEBUG ((LM_DEBUG, "No data to graph\n" ));
        return;
    }
```

Once we know there is data available for graphing we need to decide if its time to graph something:

```
    Name_Binding_Ptr lastGraphed( this->naming_context_.fetch("las
tGraphed") );

    if( lastGraphed.get() &&
        lastGraphed->int_value() == lastUpdate->int_value() )
    {
        ACE_DEBUG ((LM_DEBUG, "Data already graphed\n" ));
        return;
    }
```

The very last thing *update_graph()* will do is record the current time. If the time between the last graphing and the current time is less than desired we won't do anything.

Recall that in the monitor application that records temperatures we use the format *history[n]* for the name of each value we store. The grapher doesn't know all possible values of '*n*' so it doesn't know the exact name values to ask for via *fetch()* or *resolve()*. The naming context comes to our rescue here with the *list_name_entries()* method. This method of ACE_Naming_Context will provide us with a set of ACE_Name_Binding instances where each one's name matches the pattern provided to *list_name_entries()*[4].

```
    ACE_BINDING_SET set;
    if( this->naming_context_.list_name_entries (set, "history[")
!= 0)
    {
        ACE_DEBUG ((LM_INFO,
                    "Failed to locate anything to graph\n"));
        return;
    }
```

An ACE_BINDING_SET is an STL-like container of ACE_Name_Binding instances. In a moment we'll iterate over that list to extract the times and temperatures to graph. Each of those pairs will be put into a list of Graphable_Elements. A Graphable_Element is a simple application-specific extension of Name_Binding (which is itself a derivative of ACE_Name_Binding). We'll get into the details of Graphable_Element shortly but first let's see how it to iterate over the binding set to create each graphable element:

```
    Graphable_Element_List graphable;

    ACE_BINDING_ITERATOR set_iterator (set);

    for (ACE_Name_Binding *entry = 0;
         set_iterator.next (entry) !=0;
         set_iterator.advance())
    {
        Name_Binding binding(entry);

        ACE_DEBUG ((LM_DEBUG, "%s\t%s\t%s\n",
                    binding.type(),
                    binding.name(),
                    binding.value() ));

        Graphable_Element * ge = new Graphable_Element(entry);
        graphable.push_back( *ge );
    }
```

4. There are several other very useful methods that allow you to query the context for values or types. In each case you can receive a set of things (names, values or types) that match your pattern or a set of ACE_Name_Binding instances as we've done with *list_name_entries()*.

The Graphable_Element_List is a simple STL list<> of Graphable_Element instances. As we iterate through the set of ACE_Name_Binding instances we create a Name_Bindingwrapper for each[5] and display a line of debug information describing the data we retrieved.We then create a Graphable_Element instance from the ACE_Name_Binding instance and add it to our growing list of graphable elements.

Our final action is to creaaate a Graph instance and use it to draw a graph of the temperatures we've fetched:

```
    Graph g;
    g.graph(lastUpdate->value(), graphable);

    this->naming_context_.rebind("lastGraphed",
                                 lastUpdate->int_value());
}
```

As promised, we also store the *lastGrpahed* value so that we don't graph too oftent.

Our Graphable_Element derives from our previous Name_Binding and provides three important functions. First, It knows how to extract the time and temperature components from the *value* attribute of the underlying ACE_Name_Binding instance:

```
class Graphable_Element : public Name_Binding
{
public:
    Graphable_Element(ACE_Name_Binding * entry)
            : Name_Binding(entry)
    {
        sscanf( this->value(), "%d|%f",
                &this->when_, &this->temp_ );
    }
```

Second, it provides convenient access to these values for other parts of the application:

5. Remember the memory allocation issues that the Name_Binding wrapper solves for us.

```
    inline int when()
    {
        return this->when_;
    }

    inline float temp()
    {
        return this->temp_;
    }
```

This adaptation of one interface to another is very useful when you have an opaque-data storage mechanism such as the naming service. By taking this approach we can store any arbitrarily complex information in the naming service.

Third, the Graphable_Element provides a less-than operator to compare the time components of two graphable element instances. Considering that our graph widget might want to sort the graphable elements by time, this could be quite handy:

```
    inline bool operator<(Graphable_Element & other)
    {
        return this->when() < other.when();
    }
```

The Graphable_Element concludes with its private member data and an STL list<>.

```
private:
    int when_;
    float temp_;
};

typedef std::list<Graphable_Element> Graphable_Element_List;
```

A brief note about std::list<>... Recall that the temperature grabber's *update_graph()* method dynamically creates instances of Graphable_Element and adds them to the element list yet it never explicitly deletes these instances. Memory leak? Not so. When the std::list<> instance goes out of scope at the end of *update_graph()* it is freed from the stack. The list's destructor then iterates through the list and deletes each of the elements. Since our Graphable_Element is a derivative of our memory-conservative Name_Binding object, the space allo-

cated by the conversion of the name and value attributes from WString to char* is also cleaned up. Thus... no memory leaks.

In this section we've extended our rather simple example to use a naming context that can be shared between processes on the same system. We saw that sharing a naming context in this way is very simple and, in fact, no different than using the process-local naming context. We move on now to consider the network-local naming context and we'll see again that doing so has little effect on our application.

Appendix A
Compatibility Matrices

21.5 What Classes Work Where

Bibliography

References

1. Gamma, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
2. R. Johnson and B. Foote 1988. *Designing Reusable Classes*, Journal of Object-Oriented Programming, SIGS, June/July 1988, Vol. 1, Nr. 5, pgs 22-35.
3. Schmidt, et al. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* John Wiley & Sons, Ltd.
4. Schmidt, Douglas C. and Huston, Stephen D. 2002. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns.* Addison-Wesley.
5. Schmidt, Douglas C. and Huston, Stephen D. 2002. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks.* Addison-Wesley.
6. Stroustrup, Bjarne. 1997. *The C++ Programming Language, 3rd Edition.* Addison-Wesley.
7. Tanenbaum, A. 1996. *Computer Networks 3/e*. Prentice-Hall.

S