Chapter 3
Using the ACE Logging Facility

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

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

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

The ACE mechanisms allow us to enable and disable these statements at compile time. When compiled in, they can also be enabled and disabled at will at runtime. Thus, you don't have to pay for the overhead—in either CPU cycles or disk space—under normal conditions. But if a problem arises, you can easily cause copious amounts of debugging information to be recorded to assist you in

finding and fixing it. It is an unfortunate fact that many bugs will never appear until the program is in the hands of the end user.

In this chapter, we cover how to

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

3.1 Basic Logging and Tracing

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

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

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

The *`formatting-args`* parameter is a `printf()`-like set of format conversion operators and formatting arguments for insertion into the output. The complete set of formatting directives is described in Table 3.2. One might wonder why `printf()`-like formatting was chosen instead of the more natural—to C++ coders—C++ iostream-style formatting. In some cases, it would have been easier to correctly log certain types of information with type-safe insertion operators. However, an important factor in the logging facility's design is the ability to effectively "no-op" the logging statements at compile time. Note that the `ACE_DEBUG`

Table 3.1. `ACE_Log_Msg` Logging Severity Levels

Severity Level	Meaning
`LM_TRACE`	Messages indicating function-calling sequence
`LM_DEBUG`	Debugging information
`LM_INFO`	Messages that contain information normally of use only when debugging a program
`LM_NOTICE`	Conditions that are not error conditions but that may require special handling
`LM_WARNING`	Warning messages
`LM_ERROR`	Error messages
`LM_CRITICAL`	Critical conditions, such as hard device errors
`LM_ALERT`	A condition that should be corrected immediately, such as a corrupted system database
`LM_EMERGENCY`	A panic condition, normally broadcast to all users

and `ACE_ERROR` invocations require two sets of parentheses. The outer set delimits the single macro argument. This single argument comprises all the arguments, and their enclosing parentheses, needed for a method call. If the preprocessor macro `ACE_NDEBUG` is defined, the `ACE_DEBUG` macro will expand to a blank line, ignoring the content of the inner set of parentheses. Achieving this same optimization with insertion operators would have resulted in a rather odd usage:

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

Similarly, many of the formatting tokens, such as `%I`, would have been awkward to implement and overly verbose to use:

```
ACE_DEBUG((debug_info<<ACE_Log_Msg::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 embedded real-time systems, you really *do* want the code to simply not exist.

Unlike `ACE_DEBUG` and `ACE_ERROR`, which cause output where the macro is placed, `ACE_TRACE` causes 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 and exited. The ACE_TRACE macro accepts a single character string rather than a set of formatting directives. Because C++ doesn't have a handy way to dump a stack trace, this can be very useful indeed.

Let's take a look at a simple application:

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

void foo (void);

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

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

  return 0;
}

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

  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%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 full set of output-producing macros is listed in Table 3.3.

You can use ACE_DEBUG to print just about any arbitrary string you want, and the many format directives listed in Table 3.2 can also be modified with printf()-style modifiers for length, precision, and fill adjustments. (See a printf() reference for details on the modifiers.) In the preceding example, 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 preceding code, you should get something like this:

Table 3.2. ACE Logging Format Directives

Code	Argument Type	Displays
A	ACE_timer_t	Floating-point number; long decimal number if platform doesn't support floating point
a	—	Aborts the program after displaying output
c	char	Single character
C	char*	Character string (narrow characters)
i,d	int	Decimal number
I	—	Indents output according to the nesting depth, obtained from ACE_Trace::get_nesting_indent()
e,E, f,F, g,G	double	Double-precision floating-point number
l	—	Line number where logging macro appears
M	—	Text form of the message severity level
m	—	Message corresponding to errno value, as done by strerror(), for example
N	—	File name where logging macro appears
n	—	Program name given to ACE_Log_Msg::open()
o	int	Octal number
P	—	Current process ID
p	ACE_TCHAR*	Specified character string, followed by the appropriate errno message, that is, as done by perror()
Q	ACE_UINT64	Decimal number
r	void (*)()	Nothing; calls the specified function
R	int	Decimal number
S	int	Signal name of the numbered signal
s	ACE_TCHAR*	Character string: narrow or wide, according to ACE_TCHAR type
T	—	Current time as hour:minute:sec.usec
D	—	Timestamp as month/day/year hour:minute:sec.usec
t	—	Calling thread's ID (1 if single threaded)

Table 3.2. ACE Logging Format Directives (Continued)

Code	Argument Type	Displays
u	int	Unsigned decimal number
w	wchar_t	Single wide character
W	wchar_t*	Wide-character string
x,X	int	Hexadecimal number
@	void*	Pointer value in hexadecimal
%	N/A	Single percent sign: "%"

```
(1024) calling main in file `Simple1.cpp' on line 7
   Hi Mom
   (1024) calling foo in file `Simple1.cpp' on line 18
      Howdy Pardner
   (1024) leaving foo
   Goodnight
(1024) leaving main
```

The compile-time values of three configuration settings control whether the logging macros produce logging method calls: ACE_NTRACE, ACE_NDEBUG, and ACE_NLOGGING. These macros are all interpreted as "not." For example, ACE_NTRACE is "not tracing" when its value is 1. To enable the configuration area, set the macro to 0. ACE_NTRACE usually defaults to 1 (disabled), and the others default to 0 (enabled). Table 3.3 shows which configuration setting controls each logging macro. 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.

When deciding which features to enable, be aware that ACE_TRACE output is conditional on both the ACE_NTRACE and ACE_NDEBUG configuration settings. The reason is that the ACE_TRACE macro, when enabled, expands to instantiate an ACE_Trace object. The ACE_Trace class's constructor and destructor use ACE_DEBUG to log the entry and exit messages. They're logged at the LM_TRACE severity level, so that level also must be enabled at runtime to show any tracing output; it is enabled by default.

3.2 Enabling and Disabling Logging Severities

Consider this slightly modified code:

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

void foo(void);

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

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

  return 0;
}

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

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

The following output is produced:

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

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

We've also revealed a little more about how ACE's logging facility works. The `ACE_Log_Msg` class implements the log message formatting capabilities in ACE. ACE automatically maintains a thread-specific singleton instance of the `ACE_Log_Msg` class for each spawned thread, as well as the main thread. `ACE_LOG_MSG` is a shortcut for obtaining the pointer to the thread's singleton

Table 3.3. ACE Logging Macros

Macro	Function	Disabled by
`ACE_ASSERT(test)`	Much like the `assert()` library call. If the test fails, an assertion message including the file name and line number, along with the test itself, will be printed and the application aborted.	`ACE_NDEBUG`
`ACE_HEX_DUMP ((level, buffer, size [,text]))`	Dumps the `buffer` as a string of hex digits. If provided, the optional `text` parameter will be printed prior to the hex string. The `op_status`[a] is set to 0.	`ACE_NLOGGING`
`ACE_RETURN(value)`	No message is printed, the calling function returns with `value`; `op_status` is set to `value`.	`ACE_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`.	`ACE_NLOGGING`
`ACE_ERROR((level, string, ...))`	Sets the `op_status` to –1 and logs the `string` at the requested `level`.	`ACE_NLOGGING`
`ACE_DEBUG((level, string, ...))`	Sets the `op_status` to 0 and logs the `string` at the requested `level`.	`ACE_NLOGGING`
`ACE_ERROR_INIT( value, flags )`	Sets the `op_status` to `value` and the logger's option flags to `flags`. Valid flags values are defined in Table 3.5.	`ACE_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.	`ACE_NLOGGING`

Table 3.3. ACE Logging Macros (Continued)

Macro	Function	Disabled by
`ACE_TRACE(string)`	Displays the file name, line number, and `string` where `ACE_TRACE` appears. Displays "Leaving 'string'" when the `ACE_TRACE`-enclosing scope exits.	`ACE_NTRACE`

a. Many of the macros in this table refer to `op_status`. This internal variable is used to keep the logging framework aware of the program state, that is, the "operation status." By convention, a value of 0 indicates good. Anything else is considered an error or exception state. Also see Table 3.4.

instance. All the ACE logging macros use `ACE_LOG_MSG` to make method calls on the correct `ACE_Log_Msg` instance. There is seldom a reason to instantiate an `ACE_Log_Msg` object directly. ACE automatically creates a new instance for each thread spawned, keeping each thread's logging output separate.

We can use the `ACE_Log_Msg::priority_mask()` method to set the logging severity levels we desire output to be produced for: All the available logging levels are listed in Table 3.1. Each level is represented by a mask, so the levels can be combined. Let's look at the complete signature of the `priority_mask()` methods:

```
/// Get the current ACE_Log_Priority mask.
u_long priority_mask (MASK_TYPE = THREAD);

/// Set the ACE_Log_Priority mask, returns original mask.
u_long priority_mask (u_long, MASK_TYPE = THREAD);
```

The first version is used to read the severity mask; the second changes it and returns the original mask so it can be restored later. The second argument must be one of two values, reflecting two different scopes of severity mask setting:

1. `ACE_Log_Msg::PROCESS`: Specifying `PROCESS` retrieves or sets the processwide mask affecting logging severity for all `ACE_Log_Msg` instances.
2. `ACE_Log_Msg::THREAD`: Each `ACE_Log_Msg` instance also has its own severity mask, and this value retrieves or sets it. `THREAD` is technically a misnomer, as it refers to the `ACE_Log_Msg` instance the method is invoked on, and you can create `ACE_Log_Msg` instances in addition to those that

ACE creates for each thread. However, that is a relatively rare thing to do, so we usually simply refer to ACE_Log_Msg instances as thread specific.

When evaluating a log message's severity, ACE_Log_Msg examines both the processwide and per instance severity masks. If either of them has the message's severity enabled, the message is logged. By default, all bits are set at the process level and none at the instance level, so all message severities are logged. To make each thread decide for itself which severity levels will be logged, set the processwide mask to 0 and allow each thread set its own per instance mask. For example, the following code disables all logging severities at the process level and enables LM_DEBUG and LM_NOTICE severities in the current thread only:

```
ACE_LOG_MSG->priority_mask (0, ACE_Log_Msg::PROCESS);
ACE_LOG_MSG->priority_mask (LM_DEBUG | LM_NOTICE,
                            ACE_Log_Msg::THREAD);
```

A third mask maintained by ACE_Log_Msg is important when you start setting individual severity masks on ACE_Log_Msg instances. The per instance default mask is used to initialize each ACE_Log_Msg instance's severity mask. The per instance default mask is initially 0 (no severities are enabled). Because each ACE_Log_Msg instance's severity mask is set from the default value when the instance is created, you can change the default for groups of threads before spawning them. This puts the logging policy into the thread-spawning part of your application, alleviating the need for the threads to set their own level, although each thread can change its ACE_Log_Msg instance's mask at any time. Consider this example:

```
ACE_LOG_MSG->priority_mask (0, ACE_Log_Msg::PROCESS);
ACE_Log_Msg::enable_debug_messages ();
ACE_Thread_Manager::instance ()->spawn (service);
ACE_Log_Msg::disable_debug_messages ();
ACE_Thread_Manager::instance ()->spawn_n (3, worker);
```

We'll learn about thread management in Chapter 13. For now, all you need to know is that ACE_Thread_Manager::spawn() spawns one thread and that ACE_Thread_Manager::spawn_n() spawns multiple threads. In the preceding example, the processwide severity mask is set to 0 (all disabled). This means that each ACE_Log_Msg instance's mask controls its enabled severities totally. The thread executing the service() function will have the LM_DEBUG severity enabled, but the threads executing the worker() function will not.

The complete method signatures for changing the per instance default mask are:

```
static void disable_debug_messages
    (ACE_Log_Priority priority = LM_DEBUG);

static void enable_debug_messages
    (ACE_Log_Priority priority = LM_DEBUG);
```

Our example used the default argument, `LM_DEBUG`, in both cases. Even though the method names imply that `LM_DEBUG` is the only severity that can be changed, you can also supply any set of legal severity masks to either method. Unlike the `priority_mask()` method, which replaces the specified mask, the `enable_debug_messages()` and `disable_debug_messages()` methods add and subtract, respectively, the specified severity bits in both the calling thread's per instance mask and the per instance default severity mask.

Of course, you can use any message severity level at any time. However, take care to specify a reasonable level for each of your messages; then at runtime, you can use the `priority_mask()` method to enable or disable messages you're interested in. This allows you to easily overinstrument your code and then enable only the things that are useful at any particular time.

`ACE_Log_Msg` has a rich set of methods for recording the current state of your application. Table 3.4 summarizes the more commonly used functions. Most methods have both accessor and mutator signatures. For example, there are two `op_status()` methods:

```
int op_status(void);
void op_status(int status));
```

Although the method calls are most often made indirectly via the ACE logging macros, they are also available for direct use.

3.3 Customizing the ACE Logging Macros

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

Table 3.4. Commonly Used `ACE_Log_Msg` Methods

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

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

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

3.3.1 Wrapping ACE_DEBUG

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

The following macro definitions wrap the `ACE_DEBUG` macro in a handy way. Note how we've guaranteed that every message will be properly indented,

and we've prefixed each message to make searching for specific strings in the output easier.

```
#define DEBUG_PREFIX       ACE_TEXT ("DEBUG%I")
#define INFO_PREFIX        ACE_TEXT ("INFO%I")
#define NOTICE_PREFIX      ACE_TEXT ("NOTICE%I")
#define WARNING_PREFIX     ACE_TEXT ("WARNING%I")
#define ERROR_PREFIX       ACE_TEXT ("ERROR%I")
#define CRITICAL_PREFIX    ACE_TEXT ("CRITICAL%I")
#define ALERT_PREFIX       ACE_TEXT ("ALERT%I")
#define EMERGENCY_PREFIX   ACE_TEXT ("EMERGENCY%I")
#define MY_DEBUG(FMT, ...)     \
        ACE_DEBUG(( LM_DEBUG,  \
                    DEBUG_PREFIX FMT \
                    __VA_ARGS__))
#define MY_INFO(FMT, ...)      \
        ACE_DEBUG(( LM_INFO,  \
                    INFO_PREFIX FMT \
                    __VA_ARGS__))
#define MY_NOTICE(FMT, ...)     \
        ACE_DEBUG(( LM_NOTICE,  \
                    NOTICE_PREFIX FMT \
                    __VA_ARGS__))
#define MY_WARNING(FMT, ...)     \
        ACE_DEBUG(( LM_WARNING,  \
                    WARNING_PREFIX FMT \
                    __VA_ARGS__))
#define MY_ERROR(FMT, ...)      \
        ACE_DEBUG(( LM_ERROR,  \
                    ERROR_PREFIX FMT \
                    __VA_ARGS__))
#define MY_CRITICAL(FMT, ...)      \
        ACE_DEBUG(( LM_CRITICAL,  \
                    CRITICAL_PREFIX FMT \
                    __VA_ARGS__))
#define MY_ALERT(FMT, ...)      \
        ACE_DEBUG(( LM_ALERT,  \
                    ALERT_PREFIX FMT \
                    __VA_ARGS__))
#define MY_EMERGENCY(FMT, ...)     \
        ACE_DEBUG(( LM_EMERGENCY,  \
                    EMERGENCY_PREFIX FMT \
                    __VA_ARGS__))
```

Of course, it would be more useful if each of our prefixes were 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 "Trace.h"

void foo (void);

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_TRACE (ACE_TEXT ("main"));
  MY_DEBUG (ACE_TEXT ("Hi Mom\n"));
  foo ();
  MY_DEBUG (ACE_TEXT ("Goodnight\n"));
  return 0;
}

void foo (void)
{
  ACE_TRACE (ACE_TEXT ("foo"));
  MY_DEBUG (ACE_TEXT ("Howdy Pardner\n"));
}
```

Our output is nicely indented and prefixed as requested:

```
(1024) calling main in file `Wrap_Macros.cpp' on line 11
DEBUG   Hi Mom
   (1024) calling foo in file `Wrap_Macros.cpp' on line 20
DEBUG      Howdy Pardner
   (1024) leaving foo
DEBUG   Goodnight
(1024) leaving main
```

The `__VA_ARGS__` trick works fine for recent versions of the GNU C/C++ 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 something similar isn't available to you, you can use a slightly less elegant approach:

```
#define MY_DEBUG      LM_DEBUG,     ACE_TEXT ("DEBUG%I")
#define MY_INFO       LM_INFO,      ACE_TEXT ("INFO%I")
#define MY_NOTICE     LM_NOTICE,    ACE_TEXT ("NOTICE%I")
#define MY_WARNING    LM_WARNING,   ACE_TEXT ("WARNING%I")
#define MY_ERROR      LM_ERROR,     ACE_TEXT ("ERROR%I")
#define MY_CRITICAL   LM_CRITICAL,  ACE_TEXT ("CRITICAL%I")
#define MY_ALERT      LM_ALERT,     ACE_TEXT ("ALERT%I")
#define MY_EMERGENCY  LM_EMERGENCY, ACE_TEXT ("EMERGENCY%I")
```

This approach could be used something like this:

```
ACE_DEBUG ((MY_DEBUG ACE_TEXT ("Hi Mom\n")));

ACE_DEBUG ((MY_DEBUG ACE_TEXT ("Goodnight\n")));
```

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

3.3.2 ACE_Trace

We will now create an `ACE_TRACE` variant 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 class:

```
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_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_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 simplified version of `ACE_Trace`. Because 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 following method to select the messages you're interested in: `ACE_Log_Msg::priority_mask()`. On the other hand, if you're asked to do a postmortem analysis of a massive, all-debug-enabled log file, the prefixes can be quite handy.

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

```
#define TRACE_PREFIX       ACE_TEXT ("TRACE ")

#if (ACE_NTRACE == 1)
#    define TRACE(X)
#    define TRACE_RETURN(V)
#    define TRACE_RETURN_VOID()
#else
#    define TRACE(X)                              \
            Trace ____ (TRACE_PREFIX,             \
                        ACE_TEXT (X),             \
                        __LINE__,                 \
                        ACE_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 `TRACE_RETURN` and `TRACE_RETURN_VOID` macros is how our `Trace` object's destructor will know to print the line number at which the function exits. Each of these macros 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.

This is a simple example using our new object:

```
#include "Trace.h"

void foo (void);

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

  MY_DEBUG (ACE_TEXT ("Hi Mom\n"));
  foo ();
  MY_DEBUG (ACE_TEXT ("Goodnight\n"));

  TRACE_RETURN (0);
}

void foo (void)
{
  TRACE (ACE_TEXT ("foo"));
  MY_DEBUG (ACE_TEXT ("Howdy Pardner\n"));
  TRACE_RETURN_VOID ();
}
```

It produces the following output:

```
TRACE (1024) calling main in file `Trace_Return.cpp' on line 11
DEBUG   Hi Mom
TRACE    (1024) calling foo in file `Trace_Return.cpp' on line 22
DEBUG      Howdy Pardner
TRACE    (1024) leaving foo in file `Trace_Return.cpp' on line 24
DEBUG   Goodnight
TRACE (1024) leaving main in file `Trace_Return.cpp' on line 17
```

Although the output is a bit wordy, we succeeded in our original intent of printing the line number at which each function returns. Although that may seem like a small thing for a trivial program, consider the fact that few useful programs are trivial. If you are trying to understand the flow of a legacy application, it may 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.

3.4 Redirecting Logging Output

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

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

3.4.1 Standard Error Stream

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

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

or

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

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

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

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

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

Table 3.5. Valid `ACE_Log_Msg` Flags Values

Flag	Meaning
`STDERR`	Write messages to STDERR
`LOGGER`	Write messages to the local client logger daemon (see Section 3.6)
`OSTREAM`	Write messages to the assigned output stream
`MSG_CALLBACK`	Write messages to the callback object (see Section 3.5)
`VERBOSE`	Prepends program name, timestamp, host name, process ID, and message priority to each message
`VERBOSE_LITE`	Prepends timestamp and message priority to each message
`SILENT`	Do not print messages at all
`SYSLOG`	Write messages to the system's event log
`CUSTOM`	Write messages to the user-provided back end: an advanced usage topic not discussed in this book

3.4.2 System Logger

Most modern operating systems support the notion of a system logger. The implementation details range from a library of function calls to a network daemon. The general idea is that all applications direct their logging activity to the system logger, which will, in turn, direct it to the correct file(s) or other configurable destination(s). For example, UNIX system administrators can configure the UNIX syslog facility so that different classes and levels of logging get directed to different destinations. Such an approach provides a good combination of scalability and configurability.

To use the system logger, you would do something like this:

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

Although one would think that we could use the `set_flags()` method to enable syslog output after the `ACE_Log_Msg` instance has been opened, that isn't

the case, unfortunately. Likewise, if you want to quit sending output to syslog, a simple `clr_flags()` 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 done only in the `open()` method. Part of the initialization requires the program name that will be recorded in syslog: (the third argument). If we don't do this when our program starts, we will have to do it later, in order to get the behavior we expect from invoking `set_flags()`. Similarly, the `open()` method will properly close down any existing connection to the system logger if invoked without the `ACE_Log_Msg::SYSLOG` flag:

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

void foo (void);

int ACE_TMAIN (int, ACE_TCHAR *argv[])
{
  // This will be directed to stderr (the default ACE_Log_Msg
  // behavior).
  ACE_TRACE (ACE_TEXT ("main"));

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

  // Everything from foo() will be directed to the system logger
  ACE_LOG_MSG->open
    (argv[0], ACE_Log_Msg::SYSLOG, ACE_TEXT ("syslogTest"));
  foo ();

  // Now we reset the log output to default (stderr)
  ACE_LOG_MSG->open (argv[0]);
  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IGoodnight\n")));

  return 0;
}

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

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

Although it may seem strange to invoke `ACE_LOG_MSG->open()` more than once in your application, nothing is 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.

Directing logging output to `SYSLOG` means different things on different platforms, according to what the platform's native "system logger" is and what it is capable of. If the runtime platform doesn't support any type of system logger, directing output to `SYSLOG` has no effect. The following platforms have `SYSLOG` support in ACE:

- Windows NT 4 and newer, such as Windows 2000 and XP: ACE directs `SYSLOG` output to the system's Event Log. The third argument to `ACE_Log_Msg::open()` is an `ACE_TCHAR*` character string. It is optional; if supplied, it replaces the program name as the event source name for recording events in the system's event log. The ACE message severities are mapped to Event Log severities, as shown in Table 3.6.
- UNIX/Linux: ACE directs `SYSLOG` output to the syslog facility. The syslog facility has its own associated configuration details about logging facilities, which are different from ACE's logging severity levels. ACE's syslog back end specifies the `LOG_USER` syslog facility by default. This value can be changed at compile time by changing the `config.h` setting `ACE_DEFAULT_SYSLOG_FACILITY`. Please consult the syslog man page for details on how to configure the logging destination for the specified facility.

Table 3.6. Mapping ACE Logging Severity to Windows Event Log Severity

ACE Severity	Event Log Severity

`LM_STARTUP`
`LM_SHUTDOWN`
`LM_TRACE`
`LM_DEBUG`
`LM_INFO`
	`EVENTLOG_INFORMATION_TYPE`

`LM_NOTICE` `LM_WARNING`	`EVENTLOG_WARNING_TYPE`

`LM_ERROR`
`LM_CRITICAL`
`LM_ALERT`
`LM_EMERGENCY`
	`EVENTLOG_ERROR_TYPE`

3.4.3 Output Streams

The preferred way to handle output to files and other targets in C++ is output streams (C++ `ostream` objects). They provide enhanced functionality over the `printf()` family of functions and usually result in more readable code. The `ACE_Log_Msg::msg_ostream()` method lets us provide an output stream on which the logger will write our information:

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

Note that it's perfectly safe to select `OSTREAM` as output—via either `open()` or `set_flags()`—and then generate logging output before you invoke `msg_ostream()`. If you do so, the output will simply disappear, because no `ostream` is assigned. Also note that we have used the two-argument version of `msg_ostream()`. This not only sets the `ostream` for the `ACE_Log_Msg` instance to use but also tells `ACE_Log_Msg` that it should assume ownership and delete the `ostream` instance when the `ACE_Log_Msg` object 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.

You may wonder why the stream type is `ACE_OSTREAM_TYPE` instead of simply `std::ostream`. This is another aspect of ACE that helps its portability to platforms of all sizes and capabilities. `ACE_OSTREAM_TYPE` can be defined with or without the `std` namespace declaration, and it can also be defined as `FILE` for platforms without any C++ iostream support at all, such as some embedded environments.

3.4.4 Combined Techniques

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

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

int ACE_TMAIN (int, ACE_TCHAR *argv[])
{
  // Output to default destination (stderr)
  ACE_LOG_MSG->open (argv[0]);
```

```
  ACE_TRACE (ACE_TEXT ("main"));

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

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

  ACE_LOG_MSG->open
    (argv[0], ACE_Log_Msg::SYSLOG, ACE_TEXT ("syslogTest"));
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::STDERR);
  ACE_DEBUG
    ((LM_DEBUG, ACE_TEXT ("%IThis goes to STDERR & syslog\n")));

  ACE_LOG_MSG->msg_ostream (output, 0);
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::OSTREAM);
  ACE_DEBUG ((LM_DEBUG,
              ACE_TEXT ("%IThis will go to STDERR, ")
              ACE_TEXT ("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`. Note that before we deleted the ostream instance `output`, we first cleared the `OSTREAM` flag on the `ACE_Log_Msg` instance. Remember that the `ACE_TRACE` for `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, `ACE_Log_Msg` will dutifully attempt to write that final message on a deleted `ostream` instance, and your program will most likely crash.

3.5 Using Callbacks

To this point, we've been content to give our logging output to `ACE_Log_Msg`, which formatted the messages and directed them to the configured logging sinks. For most cases, that will be fine. What if, though, we want to do something with that output ourselves? Can we inspect or even modify the logging output before it

reaches its final destination? Of course. That's where ACE_Log_Msg_Callback comes in.

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

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

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

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

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

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

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

A simple callback implementation follows:

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

class Callback : public ACE_Log_Msg_Callback
{
```

```
public:
  void log (ACE_Log_Record &log_record) {
    log_record.print (ACE_TEXT (""), 0, cerr);
    log_record.print (ACE_TEXT (""), ACE_Log_Msg::VERBOSE, cerr);
  }
};
```

The program that uses it follows:

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

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

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

  return 0;
}
```

The program creates this output:

```
(1024) calling main in file `Use_Callback.cpp' on line 12
Sep 24 12:35:02.829 2003@@22396@LM_TRACE@(1024) calling main in fi
le `Use_Callback.cpp' on line 12
   Hi Mom
Sep 24 12:35:02.830 2003@@22396@LM_DEBUG@   Hi Mom
   Goodnight
Sep 24 12:35:02.830 2003@@22396@LM_INFO@   Goodnight
(1024) leaving main
Sep 24 12:35:02.830 2003@@22396@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 the standard error stream.

Once you have access to the `ACE_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 `ACE_Log_Record`:

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

class Callback : public ACE_Log_Msg_Callback
{
public:
  void log (ACE_Log_Record &log_record)
    {
      cerr << "Log Message Received:" << endl;
      unsigned long msg_severity = log_record.type ();
      ACE_Log_Priority prio =
        ACE_static_cast (ACE_Log_Priority, msg_severity);
      const ACE_TCHAR *prio_name =
        ACE_Log_Record::priority_name (prio);
      cerr << "\tType:        "
           << ACE_TEXT_ALWAYS_CHAR (prio_name)
           << endl;

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

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

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

      ACE_CString data (">> ");
      data += ACE_TEXT_ALWAYS_CHAR (log_record.msg_data ());

      cerr << "\tMsgData:     " << data.c_str () << endl;
    }
};
```

The following output is created:

```
Log Message Received:
        Type:        LM_TRACE
        Length:      88
        Time_Stamp:  Wed Sep 24 12:35:09 2003
        Pid:         22411
        MsgData:     >> (1024) calling main in file `Use_Callback2
.cpp' on line 12

Log Message Received:
        Type:        LM_DEBUG
        Length:      40
        Time_Stamp:  Wed Sep 24 12:35:09 2003
        Pid:         22411
        MsgData:     >>    Hi Mom

Log Message Received:
        Type:        LM_INFO
        Length:      40
        Time_Stamp:  Wed Sep 24 12:35:09 2003
        Pid:         22411
        MsgData:     >>    Goodnight

Log Message Received:
        Type:        LM_TRACE
        Length:      48
        Time_Stamp:  Wed Sep 24 12:35:09 2003
        Pid:         22411
        MsgData:     >> (1024) leaving main
```

As you can see, we have quite a bit of access to the `ACE_Log_Record` internals. We're not limited to changing only the message text. We can, in fact, change any of the values we want. Whether that makes any sense is up to your application. Table 3.7 lists the attributes of `ACE_Log_Record` and what they mean.

3.6 The Logging Client and Server Daemons

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

Table 3.7. `ACE_Log_Record` Attributes

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

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

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

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

```
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 these lines 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`. This line also requests verbose log messages instead of a more terse format. (Section 3.8 discusses more ways to use this service.) The second line of `server.conf` causes the server to listen for client connections at TCP (Transmission Control Protocol) port 20009[1] 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 the proxy. The file could be named `client.conf` and 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`, which sets the host name 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.

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 its form and value change, depending on the capabilities of the platform the client logger is built on. On some platforms, it's a pipe; where that's not possible, it's a loopback TCP socket at address `localhost:20012`. If you want your client proxy to listen at a different address, you can specify that with the `-k` parameter in `client.conf`.

You can now start the client logger with:

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

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

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

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

int ACE_TMAIN (int, ACE_TCHAR *argv[])
{
  ACE_LOG_MSG->open (argv[0],
                     ACE_Log_Msg::LOGGER,
                     ACE_DEFAULT_LOGGER_KEY);

  ACE_TRACE (ACE_TEXT ("main"));

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

  return 0;
}
```

As with the syslog example, we must use the `open()` method when we want to use the logging service; `set_flags()` isn't sufficient. Note also the `open()` parameter `ACE_DEFAULT_LOGGER_KEY`. This has to be the same logger key that the client logger is listening at; if you changed it with the `-k` option in `client.conf`, you must specify the new value to `open()`.

To summarize: On every machine on which you want to use the logging service, you must execute an instance of the client logger. Each instance 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 communicate directly with the logging server instance. This approach has two problems:

1. Your program is now more complicated because of the connection and logging logic.
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 ACE_TMAIN (int, ACE_TCHAR *[])
{
  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 (ACE_TEXT ("main"));

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

  return 0;
}
```

This looks very much like our previous callback example. We use the callback hook to capture the `ACE_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_TEXT (""), 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. The gist of what we're doing is that the callback object's constructor opens a socket to the logging service. The `log()` method then sends the `ACE_Log_Record` instance to the server via the socket. Because several of the `ACE_Log_Record` attributes are numeric, we must use the `encode()` method to ensure that they are in a network-neutral format before sending them. Doing so will prevent much confusion if the byte ordering of the host executing your application is different from that of the host executing your logging server.

3.7 The LogManager Class

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

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

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

  // ...
};
```

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

```
void foo (void);

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

  return 0;
}
```

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

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

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

To deal with compilers that don't do automatic template instantiation, we must add the following 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>
#elif defined (__GNUC__) && (defined (_AIX) || defined (__hpux))
template ACE_Singleton<LogManager, ACE_Null_Mutex> *
  ACE_Singleton<LogManager, ACE_Null_Mutex>::singleton_;
#endif /* ACE_HAS_EXPLICIT_TEMPLATE_INSTANTIATION */
```

Our `LogManager` implementation is a straightforward application of the things discussed 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 ACE_TCHAR *prog_name)
{
  ACE_LOG_MSG->open (prog_name, ACE_Log_Msg::SYSLOG, prog_name);
}

void LogManager::redirectToDaemon (const ACE_TCHAR *prog_name)
{
  ACE_LOG_MSG->open (prog_name, ACE_Log_Msg::LOGGER,
                     ACE_DEFAULT_LOGGER_KEY);
}

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::LOGGER);
  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::LOGGER);
  ACE_LOG_MSG->set_flags (ACE_Log_Msg::STDERR);
}

ACE_Log_Msg_Callback *
LogManager::redirectToCallback (ACE_Log_Msg_Callback * callback)
{
  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;
}
```

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

3.8 Runtime Configuration with the ACE Logging Strategy

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

Consider the following file:

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

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

The following sample application uses the preceding file:

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
{
  if (ACE_Service_Config::open (argc,
                                argv,
                                ACE_DEFAULT_LOGGER_KEY,
                                1,
                                0,
                                1) < 0)
```

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

```
    ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                       ACE_TEXT ("Service Config open")),
                      1);
    ACE_TRACE (ACE_TEXT ("main"));
    ACE_DEBUG ((LM_NOTICE, ACE_TEXT ("%t%IHowdy Pardner\n")));
    ACE_DEBUG ((LM_INFO, ACE_TEXT ("%t%IGoodnight\n")));

    return 0;
}
```

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

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

Recall that, by default, all logging severity levels are enabled at a processwide level. If you specify `-p INFO` in your config file, you will probably be surprised when you get other logging levels also; they were already enabled by default. To get what you want, be sure to use the disable flags, such as `~INFO`, as well; these are listed in Table 3.8.

One of the most powerful features of the logging strategy is the ability to rotate the application's log files 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, as it will go a long way toward preventing rampant disk space consumption.

Table 3.8 lists all the ACE Logging Strategy options that can be specified and their values. The possible values for `-p` and `-t` are the same as those listed in Table 3.1, but without the `LM_` prefix. Any value can be prefixed with `~` to omit that log level from the output. Multiple flags can be OR'd (|) together as needed.

Table 3.8. ACE Logging Strategy Configuration Options

Option	Arguments and Meaning
`-f`	Specify `ACE_Log_Msg` flags (`OSTREAM`, `STDERR`, `LOGGER`, `VERBOSE`, `SILENT`, `VERBOSE_LITE`) used to control logging.
`-i`	The interval, in seconds, at which the log file size is sampled (default is 0; do not sample by default).
`-k`	Specify the rendezvous point for the client logger.
`-m`	The maximum log file size in Kbytes.
`-n`	Set the program name for the `%n` format specifier.
`-N`	The maximum number of log files to create.
`-o`	Request the standard log file ordering (keeps multiple log files in numbered order). Default is not to order log files.
`-p`	Pass in the processwide priorities to either enable (`DEBUG`, `INFO`, `WARNING`, `NOTICE`, `ERROR`, `CRITICAL`, `ALERT`, `EMERGENCY`) or to disable (`~DEBUG`, `~INFO`, `~WARNING`, `~NOTICE`, `~ERROR`, `~CRITICAL`, `~ALERT`, `~EMERGENCY`).
`-s`	Specify the file name used when `OSTREAM` is specified as an output target.
`-t`	Pass in the per instance priorities to either enable (`DEBUG`, `INFO`, `WARNING`, `NOTICE`, `ERROR`, `CRITICAL`, `ALERT`, `EMERGENCY`) or to disable (`~DEBUG`, `~INFO`, `~WARNING`, `~NOTICE`, `~ERROR`, `~CRITICAL`, `~ALERT`, `~EMERGENCY`).
`-w`	Cause the log file to be wiped out on both start-up and reconfiguration.

3.9 Summary

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