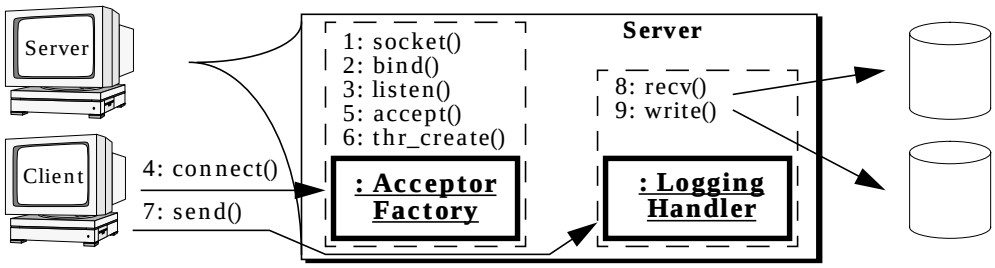
Wrapper Facade

The *Wrapper Facade* design pattern encapsulates low-level functions and data structures within more concise, robust, portable, and maintainable object-oriented class interfaces.

Example Consider you are developing a server for a distributed logging service that can handle multiple clients concurrently using a connection-oriented protocol like TCP [Ste90]. When a client wants to log data, it must first send a connection request. The logging server accepts connection requests using an *acceptor factory*, which listens on a network address known to clients.

When a connection request arrives from a client, the acceptor factory accepts the client's connection and creates a socket handle that represents this client's connection endpoint. This handle is passed up to the logging server, which spawns a thread and waits in this thread for logging requests to arrive on the connected socket handle. Once a client is connected, it can send logging requests to the server. The server receives these requests via the connected socket handles, processes the logging requests, and writes the requests to a log file.



A common way to develop the logging server is to use low-level C language functions and data structures for threading, synchronization and network communication, for example by using Solaris threads [EKBF+92] and the socket [Ste97] network programming API.

However, if the logging server is expected to run on multiple platforms, such as Solaris *and* Win32, the data structures you need may differ in type, and the functions may differ in their syntax and seman-

tics. As a result, the implementation will contain code that handles the differences in the Solaris and Win32 operating system APIs for sockets, mutexes, and threads. For example, if written with C APIs, the logging server implementation and the logging handler function, which runs in its own thread, will likely contain many `#ifdefs`:

```
#if defined (_WIN32)
   #include <windows.h>
   typedef int ssize_t;
#else
   // The following typedef is platform-specific.
   typedef unsigned int UINT32;
   #include <thread.h>
   #include <unistd.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <memory.h>
#endif /* _WIN32 */

// Keep track of number of logging requests.
static int request_count;

// Lock to protect request_count.
#if defined (_WIN32)
   static CRITICAL_SECTION lock;
#else
   static mutex_t lock;
#endif /* _WIN32 */

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

// Port number to listen on for requests.
static const int logging_port = 10000;

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// Entry point that processes logging records for
// one client connection.
#if defined (_WIN32)
   u_long
#else
   void *
#endif /* _WIN32 */
logging_handler (void *arg) {
   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
   int h = reinterpret_cast <int> (arg);
#endif /* _WIN32 */

   for (;;) {
#if defined (_WIN32)
    ULONG len;
#else
    UINT32 len;
#endif /* _WIN32 */

   // Ensure a 32-bit quantity.
   char log_record[LOG_RECORD_MAX];

   // The first <recv> reads the length
   // (stored as a 32-bit integer) of
   // adjacent logging record. This code
   // does not handle "short-<recv>s".
   ssize_t n = recv (h,
                     reinterpret_cast <char *> (&len),
                     sizeof len, 0);
   // Bail out if we're shutdown or
   // errors occur unexpectedly.
   if (n <= sizeof len) break;
   len = ntohl (len);
   if (len > LOG_RECORD_MAX) break;

   // The second <recv> then reads <len>
   // bytes to obtain the actual record.
   // This code handles "short-<recv>s".
   for (ssize_t nread = 0; nread < len; nread += n) {
       n = recv (h, log_record + nread,
                 len - nread, 0);
       // Bail out if an error occurs.
       if (n <= 0) return 0;
   }

#if defined (_WIN32)
   EnterCriticalSection (&lock);
#else
   mutex_lock (&lock);
#endif /* _WIN32 */
   // Execute following two statements
   // in a critical section to avoid
   // race conditions and scrambled
   // output, respectively.
   // Count # of requests
   ++request_count;
   if (write_record (log_record, len) == -1)
       break;

#if defined (_WIN32)
   LeaveCriticalSection (&lock);
#else
   mutex_unlock (&lock);
#endif /* _WIN32 */
   }

#if defined (_WIN32)
   closesocket (h);
#else
   close (h);
#endif /* _WIN32 */
   return 0;
}

// Main driver function for the server.
int main (int argc, char *argv[]) {
   struct sockaddr_in sock_addr;

   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET acceptor;
#else
   int acceptor;
#endif /* _WIN32 */
   // Create a local endpoint of communication.
   acceptor = socket (PF_INET, SOCK_STREAM, 0);
   // Set up the address to become a server.
   memset (reinterpret_cast<void *> (&sock_addr),
           0, sizeof sock_addr);
   sock_addr.sin_family = AF_INET;
   sock_addr.sin_port = htons (logging_port);
   sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);
   // Associate address with endpoint.
   bind (acceptor, reinterpret_cast<struct sockaddr *>
           (&sock_addr), sizeof sock_addr);
   // Make endpoint listen for connections.
   listen (acceptor, 5);

   // Main server event loop.
   for (;;) {
       // Handle UNIX/Win32 portability.
#if defined (_WIN32)
       SOCKET h;
       DWORD t_id;
#else
       int h;
       thread_t t_id;
#endif /* _WIN32 */
       // Block waiting for clients to connect.
       h = accept (acceptor, 0, 0);
       // Spawn a new thread that runs the <server>
       // entry point.
#if defined (_WIN32)
       CreateThread (0, 0,
            LPTHREAD_START_ROUTINE(&logging_handler),
            reinterpret_cast <void *> (h), 0, &t_id);
#else
       thr_create
           (0, 0, logging_handler,
            reinterpret_cast <void *> (h),
            THR_DETACHED, &t_id);
#endif /* _WIN32 */
   }
   return 0;
}
```

Even moving platform-specific *declarations* into separate configuration header files does not resolve the problems. For example, `#ifdef`s that separate the use of platform-specific APIs, such as the thread creation calls, *still* pollute application code. Likewise, `#ifdef`s may also be required to work around compiler bugs or lack of features in certain compilers. The problems with different semantics of these APIs also remain. Moreover, adding a new platform still requires application developers to modify and update the platform-specific declarations, whether they are included directly into application code or separated into configuration files.

Context Applications that access services provided by low-level functions and data structures.

Problem Applications are often written using low-level operating system functions and data structures, such as for networking and threading, or other libraries, such as for user interface or database programming. Although this is common practice, it causes problems for application developers by failing to resolve the following *problems*:

- *Verbose, non-robust programs*. Application developers who program directly to low-level functions and data structures must repeatedly rewrite a great deal of tedious software logic. In general, code that is tedious to write and maintain often contains subtle and pernicious errors.

  ➥ The code for creating and initializing an acceptor socket in the `main()` function of our logging server example is prone to errors. Common errors include failing to zero-out the `sock_addr` or not using `htons` on the `logging_port` number [Sch92]. In particular, note how the lock will not be released if the `write_record()` function returns `-1`. ❑

- *Lack of portability*. Software written using low-level functions and data structures is often non-portable between different operating systems and compilers. Moreover, it is often not even portable to program to low-level functions across different versions of the same operating system or compiler due to the lack of release-to-release compatibility [Box97].

  ➥ Our logging server implementation has hard-coded dependencies on several non-portable native operating system threading and

network programming C APIs. For instance, `thr_create()`, `mutex_lock()` and `mutex_unlock()` are not portable to Win32 platforms. Although the code is designed to be quasi-portable—it also compiles and runs on Win32 platforms—there are various subtle portability problems. In particular, there will be resource leaks on Win32 platforms since there is no equivalent to the Solaris `THR_DETACHED` feature, which spawns a 'detached' thread whose exit status is not stored by the threading library. ❑

- *High maintenance effort*. C and C++ developers typically achieve portability by explicitly adding conditional compilation directives into their application source code using `#ifdef`s. However, using conditional compilation to address platform-specific variations at *all points of use* makes it hard to maintain and extend the application. In particular, the *physical design* complexity [Lak95] of the application software becomes very high since platform-specific implementation details are scattered throughout the application source files.

  ➥ The readability and maintainability of the code in our logging server example is impeded by the `#ifdef`s that handle Win32 and Solaris portability. For instance, several `#ifdef`s are required to handle differences in the type of a socket on Win32 and Solaris. In general, developers who program to low-level C APIs like these must have intimate knowledge of many operating system idiosyncrasies to write and maintain their code. ❑

- *Lack of cohesion*. Low-level functions and data structures are not encapsulated into cohesive modules supported by programming language features, such as classes, namespaces, or packages. This lack of cohesion makes it unnecessarily hard to understand the 'scope' of the low-level APIs, which increases the effort required to learn the underlying abstraction.

  ➥ The Socket API is particularly hard to learn since the several dozen functions in the Socket library lack a uniform naming convention. Non-uniform naming makes it hard to determine the scope of the Socket API. In particular, it is not immediately obvious that `socket()`, `bind()`, `listen()`, `connect()`, and `accept()` are related. Other low-level network programming APIs address this problem by prepending a common prefix before each function, such as the `t_` prefixed before each function in the TLI API. ❑

As a result of these drawbacks, developing applications by programming directly to low-level functions and data structures is rarely an effective design choice for non-trivial application or system software.

Solution Avoid accessing low-level functions and data structures directly. For each set of related low-level functions and data structures, create one or more *wrapper facade* classes that encapsulate these functions and data structures within more concise, robust, portable, and maintainable methods provided by the wrapper facade interface.

Structure There are two participants in the Wrapper Facade pattern.

Functions are existing low-level functions and data structures that provide a cohesive service.

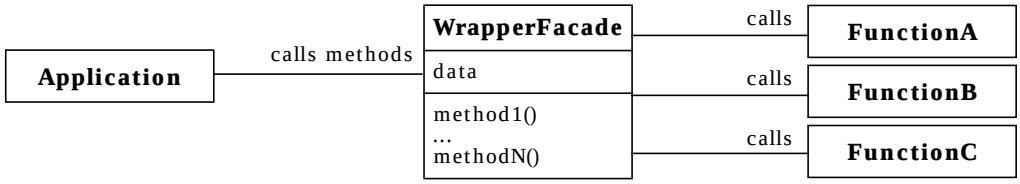
A *wrapper facade* is a set of one or more classes that encapsulate the functions and their associated data structures. These class(es) export a cohesive abstraction, that is they provide a specific kind of functionality, with each class representing a specific role in this abstraction. Application code can use the wrapper facade class(es) to access low-level functions and data structures correctly and uniformly.

Class Wrapper Facade	***Collaborator*** • Functions
Responsibility • Encapsulates low-level functions and data-structures into a cohesive abstraction	

Class Function	***Collaborator***
Responsibility • Provides a single low-level service	

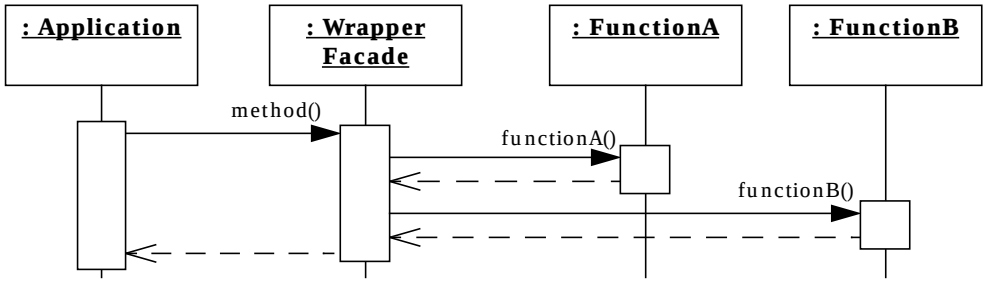
The methods in the wrapper facade class(es) generally forward client invocations to one or more of the low-level functions, passing the data structures as parameters. The data structures are typically hidden within the private portion of the wrapper facade and are not accessible to application code. The encapsulation of data types within strongly typed wrapper facade interfaces allows compilers to enforce type-safety.

The following UML class diagram illustrates the structure of Wrapper Facade.



Dynamics Collaborations in the Wrapper Facade pattern are straightforward:

- The application code invokes a method via an instance of the wrapper facade.
- The wrapper facade method forwards the request to one or more of the underlying functions that it encapsulates, passing along any internal data structures needed by the function(s).



Implementation This section describes how to implement the Wrapper Facade pattern in C++. The implementation described below is influenced by the reusable wrapper facade components provided in the ACE communication software framework [Sch97]. The Wrapper Facade pattern can be implemented in four steps:

1 *Identify the cohesive abstractions and relationships among existing functions*. Conventional APIs like Win32, POSIX, or X Windows that are implemented as individual functions and data structures provide many cohesive abstractions, such as mechanisms for network programming, synchronization and threading, and GUI event loop management. Due to the lack of data abstraction support in low-level languages like C, however, it is often not immediately obvious how these existing functions and data structures are related to each other.

The first step in applying the Wrapper Facade pattern, therefore, is to identify the cohesive abstractions and relationships among the lower

level functions and data structures in an existing API. In other words, we first define an 'object model' by clustering the existing low-level API functions and data structures into one or more classes, which hide the data representation from clients.

➥ In our logging server example, we start by carefully examining our original implementation. This implementation uses many low-level functions that actually provide several cohesive services, such as synchronization and network communication. For instance, the Solaris `mutex_lock()` and `mutex_unlock()` functions are associated with a mutex synchronization abstraction. Likewise, the `socket()`, `bind()`, `listen()`, and `accept()` functions play various roles in a network programming abstraction. ❑

2 *Cluster cohesive groups of functions into wrapper facade classes and methods*. This step can be decomposed into the following substeps:

2.1 *Create cohesive classes*. We start by defining one or more wrapper facade classes for each group of functions and data structures that are related to a particular abstraction. Several common criteria used to create cohesive classes include the following:

- Coalesce functions and data structures with high *cohesion* into individual classes, while minimizing unnecessary *coupling* between classes. Common examples of cohesive functions are ones that manipulate a common data structure, such as a socket, a file, or a signal set [Ste97].
- Determine the *common* and *variable* aspects [Cope98] in the underlying functions and data structures. Common variable aspects include synchronization mechanisms, memory managers, addressing formats, and operating system platform APIs. Whenever possible, variation in functions and data structures should be abstracted into classes that isolate the variation behind a uniform interface.

In general, if the original API contains a wide range of related functions it may be necessary to create several wrapper facade classes to properly separate concerns.

2.2 *Coalesce multiple individual functions into methods defined together in a class*. In addition to grouping existing functions into classes, it is often beneficial to combine multiple individual functions into a small-

er number of methods in each wrapper facade class. This coalescing can be used to ensure that a group of low-level functions are called in the appropriate order.

2.3 *Select the level of indirection*. Most wrapper facade classes simply forward their method calls directly to the underlying low-level functions. Thus, if the wrapper facade methods are inlined there may be no indirection overhead compared to invoking the low-level functions directly. To enhance extensibility, it is also possible to add another level of indirection by dispatching wrapper facade method implementations dynamically using virtual functions or some other form of polymorphism. In this case, the wrapper facade classes play the role of the *abstraction* class in the Bridge pattern [GHJV95].

2.4 *Determine where to encapsulate platform-specific variation*. Minimizing platform-specific application code is an important benefit of using the Wrapper Facade pattern. Thus, although wrapper facade class method *implementations* may differ across different operating system platforms they should provide uniform, platform-independent *interfaces*. There are two general strategies for determining how to encapsulate platform-specific implementation variation:

- One is to use conditional compilation to `#ifdef` the wrapper facade class method implementations. The use of `#ifdef`s is inelegant and tedious when littered throughout application code. It may be appropriate, however, if it's localized to a small number of platform-specific wrapper facade classes or files that are not visible to application developers. Moreover, when conditional compilation is used in conjunction with auto-configuration tools, such as GNU `autoconf`, platform-independent wrapper facades can be created with a single source file. As long as the number of variations supported in this file does not become unwieldy, the `#ifdef` strategy actually helps to localize the maintenance effort.
- A strategy for avoiding or minimizing `#ifdef`s altogether is to factor out different wrapper facade class implementations into separate directories, for example, one per platform. Language processing tools then can be configured to include the appropriate wrapper facade class into applications at compile-time. For example, each operating system platform can have its own directory containing implementations of platform-specific wrapper facades. To obtain a different implementation, a different include path could

be provided to the compiler. This strategy avoids the problems with `#ifdef`s described above because it physically decouples the various alternative implementations into separate directories.

Choosing a particular strategy depends largely on how frequently the wrapper facade interfaces and implementations change. For instance, if changes occur frequently, it is tedious to update the `#ifdef`s for each platform. Likewise, all files that depend on this file may need to be recompiled, even if the change is only necessary for one platform. Therefore, the larger the number of different APIs that must be supported, the less desirable the use of condition compilation.

Regardless of which strategy is selected, the burden of maintaining the wrapper facade implementations becomes the responsibility of the wrapper facade developers, rather than the application developers.

➥ For our logging server example, we define wrapper facade classes for mutexes, sockets, and threads to illustrate how each of the issues outlined above can be addressed.

Mutex wrapper facades. We first define a `Thread_Mutex` abstraction that encapsulates the Solaris mutex functions in a uniform and portable class interface:

```
class Thread_Mutex {
public:
     Thread_Mutex (void) { mutex_init (&mutex_, 0, 0); }
     ~Thread_Mutex (void) { mutex_destroy (&mutex_); }
     int acquire (void) { return mutex_lock (&mutex_); }
     int release (void) { return mutex_unlock (&mutex_); }

private:
     // Solaris-specific Mutex mechanism.
     mutex_t mutex_;

     // Disallow copying and assignment.
     Thread_Mutex (const Thread_Mutex &);
     void operator= (const Thread_Mutex &);
};
```

By defining a `Thread_Mutex` class interface, and then writing applications to use it, rather than the low-level native operating system C APIs, we can easily port our wrapper facade to other platforms. For

instance, the following Thread_Mutex implementation works on Win32:

```
class Thread_Mutex {
public:
     Thread_Mutex (void) { InitializeCriticalSection
                           (&mutex_); }
     ~Thread_Mutex (void) { DeleteCriticalSection
                           (&mutex_); }
     int acquire (void) {
          EnterCriticalSection (&mutex_);
          return 0;
     }
     int release (void) {
          LeaveCriticalSection (&mutex_);
          return 0;
     }
private:
     // Win32-specific Mutex mechanism.
     CRITICAL_SECTION mutex_;

     // Disallow copying and assignment.
     Thread_Mutex (const Thread_Mutex &);
     void operator= (const Thread_Mutex &);
};
```

As described earlier, we can support multiple operating system platforms simultaneously by using conditional compilation and #ifdefs in the Thread_Mutex method implementations. If conditional compilation is unwieldy due to the number of platforms that must be supported, one alternative is to factor out the different Thread_Mutex implementations into separate directories. In this case, language processing tools, like compilers and preprocessors, can be instructed to include the appropriate platform variant into our application at compile-time.

In addition to improving portability, our Thread_Mutex wrapper facade provides a mutex interface that is less error-prone than programming directly to the low-level Solaris and Win32 functions and data structures. For instance, we can use the C++ private access control specifier to disallow copying and assignment of mutexes, which is an erroneous use case that the less strongly-typed C programming API does not prevent.

Socket wrapper facades. Our next wrapper facade encapsulates the Socket API. This API is much larger and more expressive than the Solaris mutex API [Sch92]. Thus, we must define a group of related

wrapper facade classes to encapsulate sockets. We start by defining a typedef that handles UNIX/Win32 portability differences:

```
typedef int SOCKET;
#define INVALID_HANDLE_VALUE -1
```

Both SOCKET and INVALID_HANDLE_VALUE are defined in the Win32 API already. Naturally, we would either integrate this using #ifdefs or using separate platform-specific directories, as discussed earlier.

Next, we define an INET_Addr class that encapsulates the Internet domain address struct:

```
class INET_Addr {
public:
     INET_Addr (u_short port, long addr) {
          // Set up the address to become a server.
          memset (reinterpret_cast <void *> (&addr_),
                  0, sizeof addr_);
          addr_.sin_family = AF_INET;
          addr_.sin_port = htons (port);
          addr_.sin_addr.s_addr = htonl (addr);
     }
     u_short get_port (void) const { return
                                      addr_.sin_port; }
     long get_ip_addr (void) const { return
                                 addr_.sin_addr.s_addr; }
     sockaddr *addr (void) const {
          return reinterpret_cast <sockaddr *> (&addr_); }
     size_t size (void) const {return sizeof (addr_); }
     // ...

private:
     sockaddr_in addr_;
};
```

Note how the INET_Addr constructor eliminates several common socket programming errors. For instance, it zeros-out the sockaddr_in field and ensures the port and IP address are converted into network byte order by automatically applying the ntons() and ntonl() macros [Ste97].

The next wrapper facade class, SOCK_Stream, encapsulates the I/O operations, such as recv() and send(), that an application can invoke on a connected socket handle:

```
class SOCK_Stream {
public:
     // Default constructor.
     SOCK_Stream (void) : handle_ (INVALID_HANDLE_VALUE){}
```

```
        // Initialize from an existing HANDLE.
        SOCK_Stream (SOCKET h): handle_ (h) {}
        // Automatically close the handle on destruction.
        ~SOCK_Stream (void) {close (handle_); }
        void set_handle (SOCKET h) {handle_ = h; }
        SOCKET get_handle (void) const {return handle_; }
        // I/O operations.
        int recv (char *buf, size_t len, int flags = 0);
        int send (const char *buf, size_t len, int flags = 0);

        // ...
    private:
        // Handle for exchanging socket data.
        SOCKET handle_;
    };
```

Note how this class ensures that a socket handle is automatically closed when a SOCK_Stream object goes out of scope.

SOCK_Stream objects can be created by a connection factory, SOCK_Acceptor, which encapsulates *passive* connection establishment. The constructor of SOCK_Acceptor initializes the passive-mode acceptor socket to listen at the sock_addr address. Likewise, SOCK_Acceptor::accept() is a factory that initializes the SOCK_Stream parameter with the newly accepted connection:

```
    class SOCK_Acceptor {
    public:
        SOCK_Acceptor (const INET_Addr &sock_addr) {
            // Create a local endpoint of communication.
            handle_ = socket (PF_INET, SOCK_STREAM, 0);
            // Associate address with endpoint.
            bind (handle_, sock_addr.addr (),
                  sock_addr.size ());
            // Make endpoint listen for connections.
            listen (handle_, 5);
        };

        // Accept a connection and initialize the <stream>.
        int accept (SOCK_Stream &s) {
            s.set_handle (accept (handle_, 0, 0));
            if (s.get_handle () == INVALID_HANDLE_VALUE)
                return -1;
            else
                return 0;
        }
    private:
        // Socket handle factory.
        SOCKET handle_;
    };
```

Note how the constructor for the SOCK_Acceptor ensures that the low-level socket(), bind(), and listen() functions are always called together and in the right order.

A complete set of wrapper facades for sockets [Sch97] would also include a SOCK_Connector, which encapsulates the active connection establishment logic. Both the SOCK_Acceptor and SOCK_Connector can be used to configure the generic acceptor and connector classes described in Acceptor-Connector pattern (145) with concrete IPC mechanisms for passive and active connection establishment.

Thread wrapper facade. Our final wrapper facade encapsulates the functions and data structures related to multi-threading. Many threading APIs are available on different operating system platforms, including Solaris threads, POSIX Pthreads, and Win32 threads. These APIs exhibit subtle syntactic and semantic differences, for example, Solaris and POSIX threads can be spawned in 'detached' mode, whereas Win32 threads cannot. It is possible, however, to provide a Thread_Manager wrapper facade that encapsulates these differences within a uniform API.

The following Thread_Manager wrapper facade illustrates the spawn method implemented for Solaris threads:

```
class Thread_Manager {
public:
    int spawn (void *(*entry_point) (void *),
               void *arg,
               long flags,
               long stack_size = 0,
               void *stack_pointer = 0,
               thread_t *t_id = 0) {
        thread_t t;
        if (t_id == 0)
            t_id = &t;
        return thr_create (stack_size,
                           stack_pointer,
                           entry_point,
                           arg,
                           flags,
                           t_id);
    }
    // ...
};
```

The `Thread_Manager` can also provide methods for joining and canceling threads, as well. All these methods can be ported to many other operating system platforms. ❑

3 *Determine an error handling mechanism*. Low-level C function APIs typically use return values and integer codes, such as `errno`, that use thread-specific storage (119) to communicate errors back to their callers. This technique can be error-prone, however, if callers neglect to check the return status of their function calls. A more elegant way of reporting errors is to use exception handling. Many programming languages, such as C++ and Java, use exception handling as an error reporting mechanism. It is also used in some operating systems, such as Win32 [Cus93].

There are several benefits to using exception handling as the error handling mechanism for wrapper facade classes:

- It is extensible. Modern programming languages allow the extension of exception handling policies and mechanisms via features that have minimal intrusion on existing interfaces and usage. For instance, C++ and Java use inheritance to define hierarchies of exception classes.
- It cleanly decouples error handling from normal processing. For example, error handling information is not passed explicitly to an operation. Moreover, an application cannot accidentally ignore an exception by failing to check function return values.
- It can be type-safe. In languages like C++ and Java exceptions are thrown and caught in a strongly-typed manner to enhance the organization and correctness of error handling code. In contrast to checking a thread-specific error value like `errno` explicitly, the compiler ensures that the correct handler is executed for each type of exception.

However, there are several drawbacks to the use of exception handling for wrapper facade classes:

- It is not universally available. Not all languages provide exception handling. For instance, some C++ compilers do not implement exceptions. Likewise, operating systems like Windows NT that provide proprietary exception handling services [Cus93] that must be supported by language extensions, which can reduce the portability of applications that use these extensions.

- It complicates the use of multiple languages. Since languages implement exceptions in different ways, or do not implement exceptions at all, it can be hard to integrate components written in different languages when they throw exceptions. In contrast, reporting error information using integer values or structures provides a more universal solution.
- It complicates resource management. Resource management can be complicated if there are multiple exit paths from a block of C++ or Java code [Mue96]. Thus, if garbage collection is not supported by the language or programming environment, care must be taken to ensure that dynamically allocated objects are deleted when an exception is thrown.
- It is potentially time and/or space inefficient. Poor implements of exception handling incur time and/or space overhead even when exceptions are *not* thrown [Mue96]. This overhead can be particularly problematic for embedded systems that must be efficient and have small memory footprints.

The drawbacks of exception handling are particularly problematic for wrapper facades that encapsulate kernel-level device drivers or low-level native operating system APIs, such as the mutex, socket, and thread wrapper facades described above, that must run portably on many platforms [Sch92]. For these types of wrapper facades, a more portable, efficient, and thread-safe way to handle errors is to define an error handler abstraction that maintains information about the success or failure of operations explicitly.

One widely used error handling mechanism for the system-level wrapper facades is to use the Thread-Specific Storage pattern (119). in conjunction with `errno`. This solution is efficient and portable, which is why we use this approach for most of the wrapper facade use cases in this book.

4 *Define related helper classes (optional).* Once low-level functions and data structures are encapsulated within cohesive wrapper facade classes, it often becomes possible to create other helper classes that further simplify application development. The benefits of these helper classes is usually apparent only after the Wrapper Facade pattern has been applied to cluster low-level functions and their associated data into classes.

➥ In our example, for instance, we can leverage the following `Guard` class that implements the C++ Scoped Locking idiom (141) [Str98], which ensures that a `Thread_Mutex` is released properly, regardless of how the methods's flow of control exits a scope:

```
template <class LOCK> class Guard {
public:
     Guard (LOCK &lock): lock_ (lock) { lock_.acquire (); }
     ~Guard (void) { lock_.release (); }
     // Other methods omitted...

private:
     // Hold the lock by reference to avoid
     // the use of the copy constructor...
     LOCK &lock_;
}
```

The Guard class applies the C++ Scoped Locking idiom whereby a constructor acquires resources and the destructor releases them within a scope, as follows:

```
{
     // Constructor of <mon> automatically
     // acquires the <mutex> lock.
     Guard<Thread_Mutex> mon (mutex);
     // ... operations that must be serialized.

     // Destructor of <mon> automatically
     // releases the <mutex> lock.
}
```

Since we use a *class* as the `Thread_Mutex` wrapper facade, we can easily substitute a different type of locking mechanism, while still reusing the `Guard`'s automatic locking/unlocking protocol. For instance, we can replace the `Thread_Mutex` class with a `Process_Mutex` class:

```
// Acquire a process-wide mutex.
Guard<Process_Mutex> mon (mutex);
```

It is much harder to achieve this degree of 'pluggability' if C functions and data structures are used instead of C++ classes. ❑

Example Resolved

The code below illustrates the `logging_handler()` function of the logging server after it has been rewritten to use our wrapper facades for mutexes, sockets, and threads described in *Implementation* section. For comparison with the original code we present it in a two-col-

umn table with the original code from the example section in the left column, and the new code in the right column:

```
#if defined (_WIN32)
   #include <windows.h>
   typedef int ssize_t;
#else
   // The following typedef is platform-specific.
   typedef unsigned int UINT32;
   #include <thread.h>
   #include <unistd.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <memory.h>
#endif /* _WIN32 */

// Keep track of number of logging requests.
static int request_count;

// Lock to protect request_count.
#if defined (_WIN32)
   static CRITICAL_SECTION lock;
#else
   static mutex_t lock;
#endif /* _WIN32 */

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

// Port number to listen on for requests.
static const int logging_port = 10000;

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// Entry point that processes logging records for
// one client connection.
#if defined (_WIN32)
   u_long
#else
   void *
#endif /* _WIN32 */
logging_handler (void *arg) {
     // Handle UNIX/Win32 portability.
  #if defined (_WIN32)
     SOCKET h = reinterpret_cast <SOCKET> (arg);
  #else
     int h = reinterpret_cast <int> (arg);
  #endif /* _WIN32 */

     for (;;) {
  #if defined (_WIN32)
     ULONG len;
  #else
     UINT32 len;
  #endif /* _WIN32 */

     // Ensure a 32-bit quantity.
     char log_record[LOG_RECORD_MAX];

     // The first <recv> reads the length
     // (stored as a 32-bit integer) of
     // adjacent logging record. This code
     // does not handle "short-<recv>s".
     ssize_t n = recv (h,
                       reinterpret_cast <char *> (&len),
                       sizeof len, 0);
     // Bail out if we're shutdown or
     // errors occur unexpectedly.
     if (n <= sizeof len) break;
     len = ntohl (len);
     if (len > LOG_RECORD_MAX) break;

     // The second <recv> then reads <len>
     // bytes to obtain the actual record.
     // This code handles "short-<recv>s".
     for (ssize_t nread = 0; nread < len; nread += n) {
         n = recv (h, log_record + nread,
                   len - nread, 0);
         // Bail out if an error occurs.
         if (n <= 0) return 0;
     }
```

```
#include "ThreadManager.h"
#include "ThreadMutex.h"
#include "Guard.h"
#include "INET_Addr.h"
#include "SOCKET.h"
#include "SOCK_Acceptor.h"
#include "SOCK_Stream.h"



// Keep track of number of logging requests.
static int request_count;



// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

// Port number to listen on for requests.
static const int logging_port = 10000;

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// Entry point that processes logging records for
// one client connection.



void *logging_handler (void *arg) {

   SOCKET h = reinterpret_cast <SOCKET> (arg);

   // Create a <SOCK_Stream> object.
   SOCK_Stream stream (h);


   for (;;) {
       UINT_32 len;



       // Ensure a 32-bit quantity.
       char log_record[LOG_RECORD_MAX];

       // The first <recv_n> reads the length
       // (stored as a 32-bit integer) of
       // adjacent logging record.  This code
       // handles "short-<recv>s".
       ssize_t n = stream.recv_n
           (reinterpret_cast <char *> (&len),
            sizeof len);
       // Bail out if we're shutdown or
       // errors occur unexpectedly.
       if (n <= 0) break;
       len = ntohl (len);
       if (len > LOG_RECORD_MAX) break;

       // The second <recv_n> then reads <len>
       // bytes to obtain the actual record.
       // This code handles "short-<recv>s".
       n = stream.recv_n (log_record, len);


       // Bail out if an error occurs
       if (n <= 0) break;
```

```
  #if defined (_WIN32)
      EnterCriticalSection (&lock);
  #else
      mutex_lock (&lock);
  #endif /* _WIN32 */
      // Execute following two statements
      // in a critical section to avoid
      // race conditions and scrambled
      // output, respectively.
      // Count # of requests
      ++request_count;
      if (write_record (log_record, len) == -1)
          break;


  #if defined (_WIN32)
      LeaveCriticalSection (&lock);
  #else
      mutex_unlock (&lock);
  #endif /* _WIN32 */
      }

  #if defined (_WIN32)
      closesocket (h);
  #else
      close (h);
  #endif /* _WIN32 */
      return 0;
  }
```

```
        {
            // Constructor of Guard auto-
            // matically acquires the lock.
            Guard<Thread_Mutex> mon (lock);

            // Execute following two statements
            // in a critical section to avoid
            // race conditions and scrambled
            // output, respectively.
            // Count # of requests
            ++request_count;
            if (write_record (log_record, len)
                == -1)
                break;

            // Destructor of Guard
            // automatically
            // releases the lock, regardless of
            // how we exit this block!
        }
    }

    // Destructor of <stream> automatically
    // closes down <h>.


    return 0;
}
```

Analogously to the `logging_handler()` function, we present a two-column table that compares the original code for the `main()` function with the new code using wrapper facades:

```
// Main driver function for the server.
int main (int argc, char *argv[]) {
    struct sockaddr_in sock_addr;

    // Handle UNIX/Win32 portability.
#if defined (_WIN32)
    SOCKET acceptor;
#else
    int acceptor;
#endif /* _WIN32 */
    // Create a local endpoint of communication.
    acceptor = socket (PF_INET, SOCK_STREAM, 0);
    // Set up the address to become a server.
    memset (reinterpret_cast<void *> (&sock_addr),
            0, sizeof sock_addr);
    sock_addr.sin_family = AF_INET;
    sock_addr.sin_port = htons (logging_port);
    sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);
    // Associate address with endpoint.
    bind (acceptor, reinterpret_cast<struct sockaddr *>
          (&sock_addr), sizeof sock_addr);
    // Make endpoint listen for connections.
    listen (acceptor, 5);

    // Main server event loop.
    for (;;) {
        // Handle UNIX/Win32 portability.
#if defined (_WIN32)
        SOCKET h;
        DWORD t_id;
#else
        int h;
        thread_t t_id;
#endif /* _WIN32 */
        // Block waiting for clients to connect.
        h = accept (acceptor, 0, 0);



        // Spawn a new thread that runs the <server>
        // entry point.
#if defined (_WIN32)
        CreateThread (0, 0,
             LPTHREAD_START_ROUTINE(&logging_handler),
             reinterpret_cast <void *> (h), 0, &t_id);
#else
        thr_create
            (0, 0, logging_handler,
             reinterpret_cast <void *> (h),
             THR_DETACHED, &t_id);
#endif /* _WIN32 */
    }
    return 0;
}
```

```
// Main driver function for the server.
int main (int argc, char *argv[]) {
    INET_Addr addr (port);

    // Passive-mode acceptor object.
    SOCK_Acceptor server (addr);
    SOCK_Stream new_stream;









    // Main server event loop.
    for (;;) {









        // Accept a connection from a client.
        server.accept (new_stream);

        // Get the underlying handle.
        SOCKET h = new_stream.get_handle ();

        // Spawn off a thread-per-connection.
        thr_mgr.spawn
            (logging_handler
             reinterpret_cast <void *> (h),
             THR_DETACHED);
    }




return 0;
}
```

Note how the code in the right column fixes the various problems with the code shown in the left column. For instance, the destructors of `SOCK_Stream` and `Guard` will close down the socket handle and release the `Thread_Mutex`, respectively, regardless of how the blocks of code are exited. Likewise, this code is easier to understand, maintain, and port since it is much more concise and uses no platform-specific APIs.

Known Uses

Microsoft Foundation Classes (MFC). MFC provides a set of wrapper facades that encapsulate most of the low-level C Win32 APIs, focusing largely on providing GUI components that implement the Microsoft Document/Template architecture, which is a synonym for the Document-View architecture described in [POSA1].

The ACE framework. The mutex, thread, and socket wrapper facades described in the *implementation* section are based on components in the ACE framework [Sch97], such as the `ACE_Thread_Mutex`, `ACE_Thread_Manager`, and `ACE_SOCK*` classes, respectively.

Rogue Wave class libraries. Rogue Wave's `Net.h++` and `Threads.h++` class libraries implement wrapper facades for sockets, threads, and synchronization mechanisms on a number of operating system platforms.

ObjectSpace System<Toolkit>. Wrapper facades for sockets, threads, and synchronization mechanisms are also provided by the ObjectSpace System<Toolkit>.

Java Virtual Machine and Java class libraries. The Java Virtual Machine (JVM) and various Java class libraries, such as AWT and Swing, provide a set of wrapper facades that encapsulate many low-level native operating system calls and GUI APIs.

Siemens REFORM framework. The REFORM framework for hot rolling mill process automation [BGHS98] uses the Wrapper Facade pattern to shield the object-oriented parts of the system, such as material tracking and setpoint transmission, from a neural network for the actual process control. This neural network is programmed in C due to its algorithmic nature: it contains mathematical models which describe the physics of the automation process. The wrapper facades provide the views that the object-oriented parts of the framework need onto these functional models.

Consequences The Wrapper Facade pattern provides the following **benefits**:

Concise, cohesive, and robust programming interfaces. The Wrapper Facade pattern encapsulates many low-level functions and data structures within a more concise and cohesive set of object-oriented class methods. This reduces the tedium of developing applications, thereby decreasing the potential for programming errors. The use of encapsulation also eliminates a class of programming errors that occur when using untyped data structures, such as socket or file handles, incorrectly

Portability and maintainability. Wrapper facade classes can be implemented to shield application developers from non-portable aspects of low-level functions and data structures. Moreover, the Wrapper Facade pattern improves software structure by replacing an application configuration strategy based on *physical design* entities, such as files and `#ifdef`s, with *logical design* entities, such as base classes, subclasses, and their relationships [Lak95]. It is generally easier to understand and maintain applications in terms of their logical design rather than their physical design.

Modularity, reusability, and configurability. The Wrapper Facade pattern creates cohesive reusable class components that can be 'plugged' in and out of other components in a wholesale fashion using object-oriented language features like inheritance and parameterized types. In contrast, it is much harder to replace groups of functions without resorting to coarse-grained operating system utilities, such as linkers or file systems.

The Wrapper Facade pattern has the following **liability**:

Additional indirection. The Wrapper Facade pattern can incur additional indirection compared with using low-level functions and data structures directly. However, languages that support inlining, such as C++, can implement this pattern with no significant overhead since compilers can inline the method calls used to implement the wrapper facades. Thus, the overhead is often the same as calling the low-level functions directly

See Also The Wrapper Facade pattern is similar to the Facade pattern [GHJV95]. The intent of the Facade pattern is to simplify the interface for a subsystem. The intent of the Wrapper Facade pattern is more specific: it provides concise, robust, portable, maintainable, and

cohesive class interfaces that encapsulate low-level functions and data structures, such as the native operating system mutex, socket, thread, and GUI C language APIs. In general, facades hide complex class relationships behind a simpler API, whereas wrapper facades hide complex function and data structure relationships behind a richer class API. Moreover, wrapper facades provide building block components that can be 'plugged' into higher-level components.

The Wrapper Facade pattern can be implemented with the Bridge pattern [GHJV95] if dynamic dispatching is used to implement wrapper facade methods that play the role of the abstraction participant in the Bridge pattern.

The Layers pattern [POSA1] helps with organizing multiple wrapper facades into a separate layer component. This layer resides directly atop the operating system and shields an application from all low-level APIs that it uses.

Credits Thanks to Brad Appleton for extensive comments that greatly improved the form and content of the Wrapper Facade pattern description.