# How to call C libraries from C++

Lisa Lippincott

```
int listener = socket( AF_INET6, SOCK_STREAM, 0 );
if ( listener == -1 )
    throw std::system_error( errno, std::system_category() );

try{
    if ( listen( listener, 1 ) )
        throw std::system_error( errno, std::system_category() );
    sockaddr_in6 listeningAddress;
    socklen_t listeningAddressLength = sizeof( listeningAddress );
    if ( getsockname( listener,
                      reinterpret_cast<sockaddr*>( &listeningAddress ),
                      &listeningAddressLength ) )
        throw std::system_error( errno, std::system_category() );
    char listeningAddressString[ INET6_ADDRSTRLEN ];
    if ( !inet_ntop( AF_INET6, &listeningAddress.sin6_addr,
                               listeningAddressString,
                               sizeof( listeningAddressString ) ) )
        throw std::system_error( errno, std::system_category() );
```

```
int listener = socket( AF_INET6, SOCK_STREAM, 0 );
if ( listener == -1 )
    throw std::system_error( errno, std::system_category() );

try{
    if ( listen( listener, 1 ) )
        throw std::system_error( errno, std::system_category() );
    sockaddr_in6 listeningAddress;
    socklen_t listeningAddressLength = sizeof( listeningAddress );
    if ( getsockname( listener,
                      reinterpret_cast<sockaddr*>( &listeningAddress ),
                      &listeningAddressLength ) )
        throw std::system_error( errno, std::system_category() );
    char listeningAddressString[ INET6_ADDRSTRLEN ];
    if ( !inet_ntop( AF_INET6, &listeningAddress.sin6_addr,
                               listeningAddressString,
                               sizeof( listeningAddressString ) ) )
        throw std::system_error( errno, std::system_category() );
```

# Don’t use an OO framework.

They have poor economics.

Design is expensive.

Documentation is expensive.

Learning is expensive.

Adapting existing code is expensive.

# Don’t redesign the library.

Write design rules, and apply them uniformly to the library.

# Match the original library

type for type,
function for function,

except for the exceptions.

# Design rules for names

Choose a pithy namespace name (“Po7”).

When a name exists in the library, use that name in the namespace.
(Use lowercase when the original name is a macro.)

Use unique_* for ownership, *_cast for unsafe conversions.

Otherwise, try to fit in with the library.

# Design rules for types

Use existing types for unitless numbers.

Where the library uses a numeric type for something else, create a wrapper type, with sensible overloaded operators.

Use existing structure types wherever possible.
Bring them into the namespace with using-declarations.

Where necessary, refine existing types into templates.

Create ownership types for resources.

# Design rules for conversions

Use Wrap< WrappedType >( unwrapped ) for wrapping.

Use Unwrap( wrapped ) for unwrapping.

Use Seize< SeizedType >( released ) to seize ownership.

Use Release( seized ) to release ownership.

Use Make< TypeToMake >( parameters… ) for safe explicit conversions and for building structures.

Use various *_cast functions for unsafe conversions.

# Design rules for parameters

Pass the same parameters as the library function, except:

Skip any output parameters.

Pass by value or reference, not by pointer.

Pass wrapped types, or when passing ownership, ownership types.

If the value of a parameter changes the type of the function,
refine the function into a template.

Provide default arguments and overloads where sensible.

# Design rules for results

Return the same results as the library function, except:

Throw errors instead of returning them.

Return outputs where the library uses an output parameter.

Skip outputs that are made redundant by type safety.

Return wrapped types or ownership types.

When returning more than one thing, use a tuple:
library result first, then output parameters in the original order.

```
auto listener = Po7::socket< Po7::af_inet6 >( Po7::sock_stream,
                                              Po7::socket_protocol_t() );
Po7::listen( *listener, 1 );
std::cout << "Listening on “
          << Po7::Make< std::string >( Po7::getsockname( *listener ) )
          << "\n";
auto acceptedTuple     = Po7::accept( *listener );
auto& connectedSocket  = std::get<0>( acceptedTuple );
auto& connectedAddress = std::get<1>( acceptedTuple );
std::cout << "Accepted a connection from “
          << Po7::Make< std::string >( connectedAddress ) << "\n";
Po7::close( std::move( listener ) );
Po7::send( *connectedSocket, greeting );
char buffer[ 256 ];
while ( std::size_t receivedLength = Po7::recv( *connectedSocket, buffer ) )
    std::cout.write( buffer,
                     static_cast< std::streamsize >( receivedLength ) );
Po7::close( std::move( connectedSocket ) );
```

```
auto listener = Po7::socket< Po7::af_inet6 >( Po7::sock_stream,
                                              Po7::socket_protocol_t() );
Po7::listen( *listener, 1 );
std::cout << "Listening on "
          << Po7::Make< std::string >( Po7::getsockname( *listener ) )
          << "\n";
auto acceptedTuple     = Po7::accept( *listener );
auto& connectedSocket  = std::get<0>( acceptedTuple );
auto& connectedAddress = std::get<1>( acceptedTuple );
std::cout << "Accepted a connection from "
          << Po7::Make< std::string >( connectedAddress ) << "\n";
Po7::close( std::move( listener ) );
Po7::send( *connectedSocket, greeting );
char buffer[ 256 ];
while ( std::size_t receivedLength = Po7::recv( *connectedSocket, buffer ) )
    std::cout.write( buffer,
                     static_cast< std::streamsize >( receivedLength ) );
Po7::close( std::move( connectedSocket ) );
```

# Applying rules to a library seems rather mechanical.

Can’t a computer do that?

# Mostly, a computer can do that.

But it needs more information:

Type mappings for wrapped types; deleters for seized types;

Parameter lists and result types for improved functions;

Separation of inputs into in, out, and in/out;

Separation of outputs into errors and results; and

Ways to test for failure and create exception objects.

```
enum class socket_domain_t: int {};

template <> struct Wrapper< socket_domain_t >
                          : PlusPlus::EnumWrapper< socket_domain_t > {};

const socket_domain_t af_inet   = socket_domain_t( AF_INET );
const socket_domain_t af_unix   = socket_domain_t( AF_UNIX );
const socket_domain_t af_unspec = socket_domain_t( AF_UNSPEC );
const socket_domain_t af_inet6  = socket_domain_t( AF_INET6 );
```

```
struct SocketTag
   {
    constexpr int operator()() const            { return -1; }
    static const bool hasEquality               = true;
    static const bool hasComparison             = true;
   };
using socket_t = PlusPlus::Boxed< SocketTag >;

template < socket_domain_t >
struct SocketInDomainTag
   {
    constexpr int operator()() const            { return -1; }
    static const bool hasEquality               = true;
    static const bool hasComparison             = true;
    constexpr operator SocketTag() const        { return SocketTag(); }
   };
template < socket_domain_t domain >
using socket_in_domain = PlusPlus::Boxed< SocketInDomainTag<domain> >;
```

```
struct SocketDeleter
   {
    using pointer = PlusPlus::PointerToValue< socket_t >;
    void operator()( pointer s ) const;
   };
using unique_socket = std::unique_ptr< const socket_t, SocketDeleter >;

template < socket_domain_t domain >
struct SocketInDomainDeleter
   {
    using pointer = PlusPlus::PointerToValue< socket_in_domain<domain> >;
    void operator()( pointer s ) const      { SocketDeleter()( s ); }
    operator SocketDeleter() const          { return SocketDeleter(); }
   };
template < socket_domain_t domain >
using unique_socket_in_domain
                  = std::unique_ptr< const socket_in_domain<domain>,
                                     SocketInDomainDeleter<domain> >;
```

```
unique_socket socket( socket_domain_t   domain,
                      socket_type_t     type,
                      socket_protocol_t protocol )
   {
    return Invoke( Result< unique_socket >() + FailsWhenFalse(),
                   ::socket,
                   In( domain, type, protocol ),
                   ThrowErrorFromErrno() );
   }

template < socket_domain_t domain >
unique_socket_in_domain< domain >
socket( socket_type_t type, socket_protocol_t protocol )
   {
    return domain_cast< domain >( socket( domain, type, protocol ) );
   }
```

# Questions?