

# Catch As Catch Can: Systematic Error Handling and Exception Safety in Mixed C++/Objective-C

Mac Murrett and Andrew Pontious  
mmurrett@vanteon.com and apontious@vanteon.com

## Abstract

*C++ and Objective-C can be used in the same codebase, but their error-handling mechanisms don't mix well without some extra effort. This means careful resource allocation on both sides and translation code at every boundary between them. Sound like too much work? This paper describes the Membrane C++/Objective-C library, which makes these steps as easy as possible – often simple one-liners – while still both allowing for great flexibility and encouraging rigorous and systematic error-handling policies.*

## Error Different

### **C++ Error Handling**

Error handling in C++ has improved tremendously in the last several years.

Older C++ code, especially code ported from C or written using older compilers and libraries, often relies on return values for errors. Such values are unstandardized and difficult to propagate. And in dealing with them piecemeal, it is easy to leave a program in an unforeseen and indeterminate state.

C++ exceptions can solve these problems, but only if they are used comprehensively. Strategies to do so have come into their own, but require new code-writing habits that many have found difficult to adopt.

### **Objective-C Error Handling**

Objective-C has its own error handling mechanisms and traditions.

Return values of type `id` are widely used; a `nil` value signifies an error. Nested messages can be employed safely even if any of the inner ones could return this error value, because messages sent to `nil` just return `nil` themselves. This makes error propagation easier.

In addition, Objective-C has its own form of exceptions. No objects are released by raising an Objective-C exception, but as long as all affected objects are within an

autorelease pool any of whose parent pools are eventually released, there will be no memory leak.

The Cocoa framework takes care of this by allocating and releasing an autorelease pool each time an event is handled.

## Not Playing Nice

In any application (bundle, etc.) that contains both C++ and Objective-C code, each language's code is handled by its respective runtime.

That is the extent of their cooperation, however. There is no barrier preventing one runtime from affecting the other, nor is there any synchronization between those effects.

When a C++ exception is thrown, it doesn't stop at the beginning of the next Objective-C block; it propagates through both C++ and Objective-C sections until the next C++ catch statement or until it reaches the end of the application. This may bypass normal Cocoa memory-handling mechanism like disposal of autorelease pools (though any nested pools will be released if an outermost pool is released later).

Likewise when an `NSException` is raised, it doesn't stop at the next C++ statement, nor does it invoke the normal C++ runtime's "stack unwinding" (invocation of object destructors) as it exits C++ code blocks. This is no surprise; raising an `NSException` is really just a direct invocation of C's `setjmp/longjmp` mechanism, which likewise does not invoke C++ destructors.

The sample project "Exception Propagation"<sup>1</sup> demonstrates these side effects.

## From Objective-C to C++

### *Translating NSExceptions*

Unlike in C++, in Objective-C exceptions all inherit from one class, `NSException`, since only that class has the `raise` method that can invoke the exception mechanism.

The type of the exception is traditionally determined by looking at its name.

To use the Membrane library's `NSException` translation mechanism, you'll need to bracket your Objective-C code block with the following macros, defined in `nsexception_translation.h`, like this:

```
NSEXCEPTION_TRANSLATOR_DURING
```

---

<sup>1</sup> All projects are available on the MacHack conference CD and have versions for Project Builder 2.1 (Dec '02 Tools) and CodeWarrior 8.3.

```
/* Objective-C code block */
NSEXCEPTION_TRANSLATOR_HANDLER
```

These macros should be used instead of the `NS_DURING` and `NS_HANDLER/NS_ENDHANDLER` macros; as a convenience, they call these Apple-defined macros for you.

Behind the scenes, the first macro creates an instance called “translator” of the class `nsexception_translator`.

This class has two parts. First, it has a list of translation rules, encapsulated in instances of the class `nsexception_translation`. We will discuss them below.

## ***The Default Thrower Function***

But an `nsexception_translator` is useful even without any rules, because it also has a default “thrower function” which is invoked if there is no other match. Thrower functions take an `NSError` pointer as their single argument and create and throw a C++ type or object. The default thrower function, unpaired with any evaluator logic, translates *any* `NSError` into what the default thrower function throws.

The factory-installed default thrower throws a `std::runtime_error` instance whose “what” variable is set to the `NSError`’s name, which can be moderately useful.

Here’s an example:

```
NSEXCEPTION_TRANSLATOR_DURING

    NSFileHandle *file = [NSFileHandle
                          fileHandleForReadingAtPath:
                          @"/private/var/tmp/console.log"];

    // writing to read-only file raises
    // NSErrorOperationException
    [file writeData:[NSData dataWithBytes:"7 bytes" length:7]];

NSEXCEPTION_TRANSLATOR_HANDLER
```

Run this example, located in the “Example #1” portion of the sample project “`NSError Translations`”, to see the default macros in action.

To change the default thrower function, you can use a mutator function on `nsexception_translator` called `set_default_thrower()`. Remember, the translator instance created by the macro is called “translator”, so after the `DURING` macro you would add the line:

```
translator.set_default_thrower(/*something*/);
```

But there's also an easier approach if that's the only modification you want to make to the translator. You can use the even longer-named macro

```
NSEXCEPTION_TRANSLATOR_DURING_DEFAULT_THROWER(/*something*/)
```

which requires the same parameter as the mutator function, a thrower function pointer or functor object instance. Here's an example:

```
throw_unsigned_int(NSException *nsexception)
{
    throw [[nsexception name] length];
}
```

```
NSEXCEPTION_TRANSLATOR_DURING_DEFAULT_THROWER(throw_unsigned_int)
```

```
/*same code as before*/
```

```
NSEXCEPTION_TRANSLATOR_HANDLER
```

In this example, Example #2 in "NSException Translations", the default thrower function is changed to `throw_unsigned_int()`, which does what it says. This means that the same `NSException` that we translated into a `std::runtime_error` before now gets translated into an unsigned int.

Note this modified functionality, apart from the thrower function definition, still only requires a one-line macro at the beginning of each code block.

## ¿Habla usted español?

Now that we've taken care of the default case, let's talk about translations.

Every `nsexception_translation` instance pairs an evaluator function with a thrower function.

Thrower functions we already know about. An evaluator function takes an `NSException` pointer and returns a `bool`.

The translator goes through its list of translations and invokes the evaluator function on each. The first one whose evaluator function returns true has its thrower function invoked.

The most common evaluation is the straight comparison of the name of the `NSException` to another string; it's so common that `nsexception_translation` has a special constructor that takes an `NSString` and makes the necessary evaluator function for you.

Here's an example:

```
NSEXCEPTION_TRANSLATOR_DURING
```

```
    translator.add_translation(nsexception_translation(  
                                NSFileHandleOperationException,  
                                throw_unsigned_int));  
  
    /*same code as before*/
```

```
NSEXCEPTION_TRANSLATOR_HANDLER
```

In this example, Example #3 in “NSExcption Translations”, a translation is added that, upon finding an “NSFileHandleOperationException” NSExcption, calls the same thrower function we used above, `throw_unsigned_int()`.

You can also define and use your own evaluator functions, allowing for more flexible criteria. The following example demonstrates this.

```
void has_userinfo(NSExcption* nsexception)  
{  
    return  
        ([nsexception userInfo] != nil &&  
         [[nsexception userInfo] count] > 0);  
}
```

```
NSEXCEPTION_TRANSLATOR_DURING
```

```
    translator.add_translation(nsexception_translation(  
                                has_userinfo,  
                                throw_unsigned_int));
```

```
    NSDictionary *userInfo =  
        [NSDictionary dictionaryWithObjectsAndKeys:  
            @"hi", @"hi key", nil];  
    NSExcption *exception =  
        [NSExcption exceptionWithName:@"name"  
            reason:nil  
            userInfo:userInfo];  
    [exception raise];
```

```
NSEXCEPTION_TRANSLATOR_HANDLER
```

In this example, Example #4 in “NSExcption Translations”, we define our own evaluator function, `has_userinfo()`, which only returns true if there is a `userInfo` dictionary with entries in it. You'll see in the example project that `throw_unsigned_int()` has been conditionally defined to throw the number of entries in the dictionary if it has any.

Finally, you can replace all the explicit calls to `add_translation()` with the convenience macro `NSEXCEPTION_TRANSLATOR_ADD`, which is slightly shorter and easier to cut and paste (one word) than the function call and constructor name that it replaces. The below example, also in Example #5 of “NException Translations” demonstrates this:

```
NSEXCEPTION_TRANSLATOR_DURING
    NSEXCEPTION_TRANSLATOR_ADD(has_userinfo, throw_unsigned_int));

/*rest same as last example*/
```

## ***Clocking in at the Factory Function***

There’s one `DURING` macro we haven’t discussed yet:

```
NSEXCEPTION_TRANSLATOR_DURING_MAKE_FROM
```

This is the most sophisticated NException translation case, and should be used when you have a set of multiple translations you want to use repeatedly in your code, which is the best way to set things up for a large codebase.

First, define a factory function that returns a constant `nexception_translator` reference. Pass in the name of that function to this macro. At runtime, it will invoke the function and use the resulting translator.

Here’s a factory function example, Example #6 in “NException Translations”:

```
const nexception_translator& make_translator()
{
    static std::auto_ptr<const nexception_translator> ptr;
    if(!ptr.get())
    {
        std::auto_ptr<nexception_translator> temp_ptr(
            new nexception_translator());

        // add translations here
        temp_ptr->add_translation(
            nexception_translation(has_userinfo, throw_unsigned_int));

        ptr = temp_ptr;
    }
    return *ptr.get();
}

NSEXCEPTION_TRANSLATOR_DURING_MAKE_FROM(make_translator)

/*rest same as last example*/
```

This does the same thing as the previous example, add a translation from `NSExceptions` with a `userInfo` to unsigned longs, but it puts all the work in the factory function. The code needed per-incident is again only one line.

Also, because the factory function only ever makes one static, constant instance of the translator, the cost of its repeated invocation is lower at runtime than the previous examples.

If you're using the Membrane macros more than once, and you would be for any real project, we highly encourage factory functions over cut-and-paste.

## Translating Objective-C Return Values

### *Why We're Not Done*

If we stopped at the above translation mechanism, we'd still have a problem when wrapping Objective-C code in C++.

The problem is, in Apple's Objective-C frameworks, exceptions are relatively rare and almost always due to coding errors. For reporting runtime errors, Apple's Foundation and AppKit frameworks rely far more heavily on return values.

As stated in the introduction, in C++ return values are nonstandard and problematic, so ideally even these Objective-C return values should be translated into C++ exceptions. This is where the `throw_if()` function comes in.

### *throw\_if()*

`throw_if()` is loosely patterned on C++ Standard Library functions like `find_if()` and `count_if()`. It takes an Objective-C (or C or C++) expression as its first parameter and plugs that expression's return value into the predicate provided in the second parameter. If the predicate returns true, the thrower function in the third parameter is invoked.

Here's an example:

```
NSDictionary *dictionary = /*same as last example*/
throw_if([dictionary objectForKey:@"bye key"],
        std::bind1st(std::equal_to<id>(), nil));
```

In this example, Example #1 in the sample project "throw\_if", the result of the method `objectForKey` is evaluated by the `throw_if()` function before being passed along.

The predicate used for the evaluation is `std::equal_to()`, which normally takes two parameters. Via `std::bind1st()` we create a new predicate that only takes one and always compares that parameter to `nil`.

The thrower function parameter is absent, in which case `throw_if()` uses a default thrower function that throws `std::runtime_error`.

Because the value is passed along, we could wrap the `throw_if()` function in some further function, as in this example, Example #2 in “`throw_if`”:

```
NSLog(@"Value for \"bye key\" is @",
      throw_if(/*same as previous*/));
```

What if you want to throw on everything *but* one particular value? Here’s an (admittedly trivial) example Example #3 in “`throw_if`”:

```
const char *string1 = "hi";
const char *string2 = "bye";

throw_if(std::strcmp(string1, string2),
        std::bind1st(std::not_equal_to<int>(), 0));
```

This example evaluates the result of the common C function `strcmp()`. Unlike the previous example, there is only one value for success: zero.

So for our predicate, we use `std::not_equal_to()`, and use the same trick as before to turn that two-parameter functor into a one-parameter functor.

You are not limited to these two predicates; see the C++ Standard Library header `functional` for more ideas. But since the “is equal to” and “is not equal to” cases are so frequent, we have provided specialized `throw_if()` variants for them.

Here are the same two examples as above, also Example #4 in “`throw_if`”, using the specialized versions of `throw_if()`.

```
throw_if_equal_to([dictionary objectForKey:@"bye key"], nil);
throw_if_not_equal_to(std::strcmp(string1, string2), 0);
```

And finally, here’s an example, Example #5 in “`throw_if`”, of a caller-supplied thrower function. See how `throw_if()` thrower functions take the return value type as their parameter, not an `NSException`:

```
enum Error{ kGreaterThan, kLessThan };
void throw_Error(int result)
{
    throw (result > 0 ? kGreaterThan : kLessThan);
}
```

```
throw_if_not_equal_to(std::strcmp(string1, string2),
                     0,
                     throw_Error);
```

The above example takes the result of `strcmp()` and changes it into the custom typedef `Error`.

## **Location, Location, Location**

Now that we know *how* to turn return values into C++ exceptions, *where* should you do so?

Since throwing C++ exceptions has the same effect on Objective-C code that raising `NSExcptions` does, you should follow the same guidelines in both cases: only use `throw_if()` if the code block can be interrupted without ill effects.

This means doing all the potentially dangerous steps on temporary, autoreleased objects. Only commit the changes to the real objects at the end, once you know no more errors are possible.<sup>2</sup>

How do you know when you've reached that stage? We've provided you with an answer; if it makes sense to wrap any more statements in `throw_if()` calls, then you're not out of danger yet.

## **From C++ to Objective-C**

Now that we can handle errors that should propagate from Objective-C code to C++ code, we should look at how to do the reverse: catch C++ exceptions and translate them into objects and values that any enclosing Objective-C code can use.

## **Translating C++ Exceptions**

### **Are You My Type? [Y/N]**

Unlike Objective-C, where every exception is guaranteed to be an instance of `NSExcption`, in C++ any type can be thrown: primitives like `int` and `bool`, custom typedefs, and full-fledged classes like `std::runtime_error`.

Unlike in our `nsexception_translator`, where different kinds of `NSExcptions` could be lumped together by writing the correct name-evaluator function, in C++ multiple types can be lumped together in the same catch block only if they are subclasses of the same superclass, or if you use an indiscriminant "default" catch block.

---

<sup>2</sup> For a more in-depth discussion of this in C++, see Herb Sutter's *Guru of the Week* column 59, "Exception-Safe Class Design, Part 1" at <http://www.gotw.ca/gotw/059.htm>.

```

try
{
    /* throw something */
}
catch(int integer_exception)
{ /* ... */ }
catch(bool boolean_exception)
{ /* ... */ }
catch(MyClass my_exception)
{ /* ... */ }
catch(...)
{ /* ... */ }

```

In the above example, integers and booleans are handled separately, and only MyClass and its subclasses otherwise singled out. Everything else is still caught, in the default block, but you can't evaluate or manipulate what was caught, because you don't even have any way to refer to it!

## ***Your Translator Has Arrived***

The C++ equivalent to `nsexception_translator` is called, not surprisingly, `exception_translator`.

Also, unsurprisingly, given the introduction above, you register translations by C++ type, where specified superclasses will also match their subclasses.

Here's an example, Example #1 in sample project "C++ Exception Translations":

```

exception_translator translator;
try
{
    translator.register_translation<std::bad_alloc>(
        @"NSOutOfMemoryException",
        @"There is no memory left",
        nil);

    // more C++ code that might
    // throw exceptions
}
catch(...)
{
    translator.translate_and_raise();
}

```

The `register_translation()` function requires the type that will be caught as the first "parameter" (actually the template type), and its remaining three parameters are all the inputs needed for making an `NSException`: an `NSString` for the name, an `NSString`

for the reason, and an NSDictionary for the userInfo. Note the reason and userInfo parameters are optional. If they are not provided, nil is used.

The translations from type to NSError occur within the translate\_and\_raise() call in the default catch block. “What?” you say? “I thought you didn’t know the type in a default catch block?”

What translate\_and\_raise() does via the magic of template metaprogramming is instantiate its own series of nested catch blocks automatically for you, right where you call it, and then simply re-throw the (unknown) exception within the most deeply nested block. Each block gets the chance to identify the exception by type, catch it again, and perform the correct translation, before control is passed to the enclosing block. That’s it! If you want to know more, see the exception\_translator.h and .mm files, though you can use these classes without knowing the particulars of their implementation.

As with the add\_translation() function in nsexception\_translator, you can add as many translations as you like, and they’re “called” in order.

## **More One-Liners**

And the similarities don’t end there, because just as with nsexception\_translator, we provide macros and a system whereby the translation layer can be reduced to a single line before and after the affected code block.

As with the NSError macros, we try to follow the form of the code we’re replacing. Instead of DURING and HANDLER, we’re replacing try and catch, so:

```
EXCEPTION_TRANSLATOR_TRY

    translator.register_translation<std::bad_alloc>(
        @"NSOutOfMemoryException",
        @"There is no memory left",
        nil);

    // more C++ code that might
    // throw exceptions

EXCEPTION_TRANSLATOR_CATCH
```

This is Example #2 in “C++ Exception Translations”.

## **Not Your Fault – Defaults**

Just as with nsexception\_translator, we provide a way for you to use a default value if the thrown exception doesn’t match any of the translations you’ve registered, a mutator method called set\_default\_nsexception(). And we call that method for you if you use the following macro (Example #3):

```

EXCEPTION_TRANSLATOR_TRY_DEFAULT( \
    @"NSOutOfMemoryException", \
    @"There is no memory left", nil)

// more C++ code that might
// throw exceptions

EXCEPTION_TRANSLATOR_CATCH

```

Example #4 shows the use of a factory function to create the translator. Otherwise the example is the same as above:

```

const exception_translator& make_translator()
{
    static std::auto_ptr<const exception_translator> ptr;
    if(!ptr.get())
    {
        std::auto_ptr<exception_translator> temp_ptr(
            new exception_translator());

        // add translations here
        temp_ptr->register_translation<std::bad_alloc>(
            @"NSOutOfMemoryException",
            @"There is no memory left");

        ptr = temp_ptr;
    }

    return *ptr.get();
}

EXCEPTION_TRANSLATOR_TRY_MAKE_FROM(make_translator)

// more C++ code that might
// throw exceptions

EXCEPTION_TRANSLATOR_CATCH

```

## Returning id for Exceptions

As stated earlier, NSEExceptions are sometimes of limited value, since Apple's frameworks so often deal with return values.

So the Membrane library also has a mechanism to translate C++ exceptions to the most common return type, id.

In fact, the class that allows you to do this, `exception_handler`, is a superset of the `exception_translator` functionality. It lets you register both NSEExceptions to be

thrown for particular C++ exception types or return values to be returned for a type. Use `register_translation()` for the former and `register_return_value()` for the latter.

Example #5 shows you how to use the return value functionality for the `init` method of the Objective-C class `MemoryThrower`:

```
@interface MemoryThrower : NSObject
{}
@end

@implementation MemoryThrower
-(id)init
{
    [super init];
    id value = self;

    exception_handler handler;
    handler.register_return_value<std::bad_alloc>(nil);
    try
    { throw std::alloc; }
    catch(...)
    {
        value = handler.translate_and_raise_or_return();
    }

    return value;
}
@end
```

In this example, any attempt to create a `MemoryThrower` instance will end in a return value of `nil`. This is what you want from a memory failure in an Objective-C class.

## Sharing Resources

We've handled how to treat separate blocks of C++ and Objective-C code. But wait, there's more!

In any real-world Objective-C++ program, you don't just have separate code blocks. You have Objective-C objects owning pointers to C++ objects, and C++ objects owning pointers to Objective-C objects.

One way to deal with such embedded objects is to handle them per their native language. This means `retain` and `release` calls in the middle of C++ constructors and destructors, and bare `new` and `delete` calls in the middle of Objective-C `init` and `dealloc` methods. These methods can be error-prone, not to mention a maintenance nightmare.

Instead, we would like to use the native methods in each runtime; the autorelease pool in Objective-C, and smart pointers like `std::auto_ptr` and `boost::shared_ptr` in C++.

To do that, we need wrapper classes.

For Objective-C objects, the Membrane library provides a smart pointer called `shared_nsubject`. Here's an example:

```
shared_nsubject<NSString> foo([NSString stringWithCString:"foo"]);
```

`shared_nsubject`, like other C++ smart pointers, is created on the stack, and when it goes out of scope, it deletes its resource. But because this is an Objective-C object, it releases it rather than destructing it.

C++ classes can have `shared_nsubject` data members and know that, when their instances go out of scope, their Objective-C objects will also be released (unless retained by other Objective-C objects).

Note that because `release` messages will be sent (and `dealloc` methods invoked) upon C++ class destructor invocation, in order to keep with the C++ best practice of "no exceptions during destructors," no Objective-C class should raise during its `dealloc` method.

To access the Objective-C pointer, use the `get()` method. Here are some examples:

```
[foo.get() retain];  
[foo.get() cString];
```

The `get()` function returns a pointer of the same type that was used when the `shared_nsubject` was created.

To use Objective-C objects without worrying about type, see the `shared_id` class.

`shared_nsubject` also supports `std::swap()`.

How about in the other direction? For C++ objects, the Membrane library provides `NSAutoPtr` and the factory function `MakeNSAutoPtr()`. Here's an example:

```
std::auto_ptr<std::string> stringptr(new std::string("string"));  
  
NSAutoPtr *ns_stringptr(MakeNSAutoPtr(stringptr));  
[stringptr retain];
```

`MakeNSAutoPtr()` takes the C++ object by `std::auto_ptr`. This is for two reasons: it clarifies that `MakeNSAutoPtr()` is taking ownership of your object, and it ensures that the object is owned when `MakeNSAutoPtr()` is called. The latter point is important as `MakeNSAutoPtr()` might throw a C++ exception.

After calling `MakeNSAutoPtr()`, your `std::auto_ptr` will no longer own the pointer, regardless of whether creation of the `NSAutoPtr` was successful. Your pointer will get deleted at some point regardless of what happens.

`MakeNSAutoPtr()` returns an autoreleased `NSAutoPtr`, so be sure to retain it if you need to, as shown in the example above.

As the `NSAutoPtr` is likely to be released at autorelease time, your pointer is likely to be deleted at autorelease time. It is therefore *very* important that your pointer's destructor and operator `delete` do not throw.

To see a simple example of shared resources, open the sample project "Shared Resources". It contains two class, the C++ class `CppClass` and the Objective-C class `ObjCClass`. In this respect, it is similar to the original sample project mentioned in this paper, "Exception Propagation". Unlike that project, however, in this project the classes each own an instance of the other class.

Running that project will show you how the lifetime of the owned object is handled by its owner's language's means. The owned `ObjCClass` instance is released by a C++ smart pointer, and the `CppClass` is destructed via a `release` message to its wrapper Objective-C class's pointer.

## Conclusion

We've shown you lots of small examples about how Membrane works, but we think Membrane will shine especially with larger projects that need the kind of flexibility and discipline it can provide.

Please check out <http://www.umbar.com/membrane/> for more in-depth projects as well as SourceForge information.