Bjarne fix your freaking language!

August 04, 2025 • Jonathan Marler

About 23 minutes into his talk about Safe C++ [1], Bjarne shows a slide with this code:

```
void f(const char* p)           // unsafe, naive use
{
    FILE *f = fopen(p, "r");    // acquire
    // use f
    fclose(f);                  // release
}
```

He shows this code to demonstrate how easy it is to miss resource cleanup. Any code that exits the function inside `// use f` that doesn't also call `fclose` results in a leak. He provides a `C++` equivalent with RAII to avoid this footgun:

```
class File_handle {    // belongs in some support library
    FILE *p;
public:
    File_handle(const char *pp, const char *r)
        { p = fopen(pp, r); if (p == 0) throw File_error(pp, r); }
    File_handle(const string& s, const char *r)
        { p = fopen(s.c_str(), r); if (p == 0) throw File_error(pp, r); }
    ~File_handle() { fclose(p); } // destructor
    // copy operations
    // access functions
};


void f(string s)
{
    File_handle fh { s, "r"};     // now: ifstream fh{s}
    // use fh
}
```

This sample is great at showing the benefits of RAII, but introduces some problems when it comes to error handling. In this example, Bjarne elects to throw an exception to propagate any error from calling `fopen`. Unfortunately, C++ exceptions have 3 problems:

1. Correctness: you don't know if the exception type you've caught matches what the code throws
2. Exhaustiveness: you don't know if you've caught all exceptions the code can throw
3. RAISI: try/catch requires a new scope which usually means you need RAISI (Resource Acquisition is Second Initialization)

In most applications it's expected that failing to open a file is a normal error that should be handled in some way other than throwing an exception all the way up the stack. Let's assume the proper way to handle the error in this case is to report it to stderr and return from the function. Let's see how that looks in the original C code:

```
void f(const char* p)
{
    FILE *f = fopen(p, "r");
    if (f == NULL) {
        fprintf(stderr, "failed to open file '%s', error=%d\n", p, errno);
        return;
    }
    // use f
    fclose(f);
}
```

Most systems-level programmers can look at this code and verify the error has been "caught". Almost all C functions that return a pointer reserve `NULL` for the error case. This method isn't perfect, relying on special values to detect errors is a common source of bugs, but let's contrast this with the C++ example:

```
void f(string s)
{
    try {
        File_handle fh { s, "r"};
        // use fh
    } catch (const File_error& e) {
        fprintf(stderr, "failed to open file '%s', error=%d\n", s.c_str(), errno);
        return;
    }
}
```

This example introduces a few problems. The first is that our error message may not be correct. It's possible that the exception we've caught was not introduced by opening this file, and, the errno may not reflect the errno at the time `fopen` was called. To fix the first problem, we can limit the code inside our `try/catch` to just the code that opens the file, let's adjust it to do so:

```
void f(string s)
{
    File_handle fh;
    try {
        fh = File_handle(s, "r");
    } catch (const File_error& e) {
        fprintf(stderr, "failed to open file '%s', error=%d\n", s.c_str(), errno);
        return;
    }
    // use fh
}
```

Unfortunately this won't compile because the `File_handle` type has no default constructor. Because we can only catch an exception inside a scope, we need to enhance our `File_handle` type to cover this transient state of existence before it's initialized. In other words, RAII is no longer good enough, now we need RAISI. We need to introduce our `File_handle` object in the outerscope with an initial “null” state, then really initialize it a second time inside the `try/catch` scope.

One way to accomplish this is to wrap `File_handle` in `optional`, then update all our `//use fh` code to use `fh.value()` or `*fh` instead of just `fh`. Another way is to enhance `File_handle` itself to support a `null state`, which would look something like this:

```
// new constructor
File_handle() : p(nullptr) { }

// enhanced destructor
~File_handle() { if (p) fclose(p); }
};
```

The second problem with our error message is that we don’t know if `errno` is correct. Alot of things have occurred between the time that our call to `fclose` failed in the contructor and the exception was caught. To handle this, we can enhance the `File_error` class provided to us by Bjarne by also having it store the errno at the time it’s thrown:

```
// enhanced constructor
File_handle(const char *pp, const char *r)
    { p = fopen(pp, r); if (p == 0) throw File_error(pp, r, errno); }
};
```

Now we have the tools to report a correct error message. However, if you look at the original C code and the C++ code side-by-side, it doesn’t look pretty. It’s more noisy than the original C example which generates backpressure in changes that attempt to implement proper error handling like this.

The bigger problem with our `C++` example is that it provides no guarantees about whether we’ve actually caught all the exceptions that could occur when opening a file. Any nested function/operator/object used inside our File_handle constructor has the ability to throw any other exception type that we haven’t accounted for, and now instead of an error message, our program unintentionally crashes. This scenario is unlikely in this particular example, but, you can imagine how this problem gets exponentially worse once you start sprinkling exceptions throughout your code base. It becomes impossible to know whether your code handles all possible exceptions in the places you need to. In contrast, using return values for error handling localizes the problem only to the function you are calling. With return values, it’s much more difficult for a change to a function multiple levels down in the call stack to introduce a new error state that you can’t catch/handle.

This is the general problem with RAII in C++, how do you pair it with error handling that developers will use without introducing the problems we’ve discussed? One technique you can use today is to do your error handling before your RAII. The example above could be written like this instead:

```
class File_handle {
    FILE *p;
public:
    File_handle(FILE* p) : p(p) { }
    ~File_handle() { if (p) fclose(p); }
};


void f(string s)
{
    File_handle fh(fopen(p, "r"));
    if (!fh.p) {
        fprintf(stderr, "failed to open file '%s', error=%d\n", s.c_str(), errno);
        return;
    }
    // use fh
}
```

I find that using this technique typically results in these benefits:

- your classes get smaller/simpler
- you no longer need to define exception types
- the code that knows how to handle the error has the context it needs to report it properly

Since we've removed the abstraction that `File_handle` calls `fopen`, we now call it ourselves and know that we can get further error information through `errno` without propagating it through a new `File_error` object. The general principle here is that anything that can fail, you do outside of a constructor. By avoiding the need to handle errors inside a constructor, you avoid having the introduce exceptions and the subsequent problems.

Another technique to address these problems is to provide a "side-channel" in your constructors for reporting errors. In fact, some of the std library does this. The function `std::filesystem::rename` takes a reference to an `error_code` which can be used to report errors. Here's what out code could look like with that:

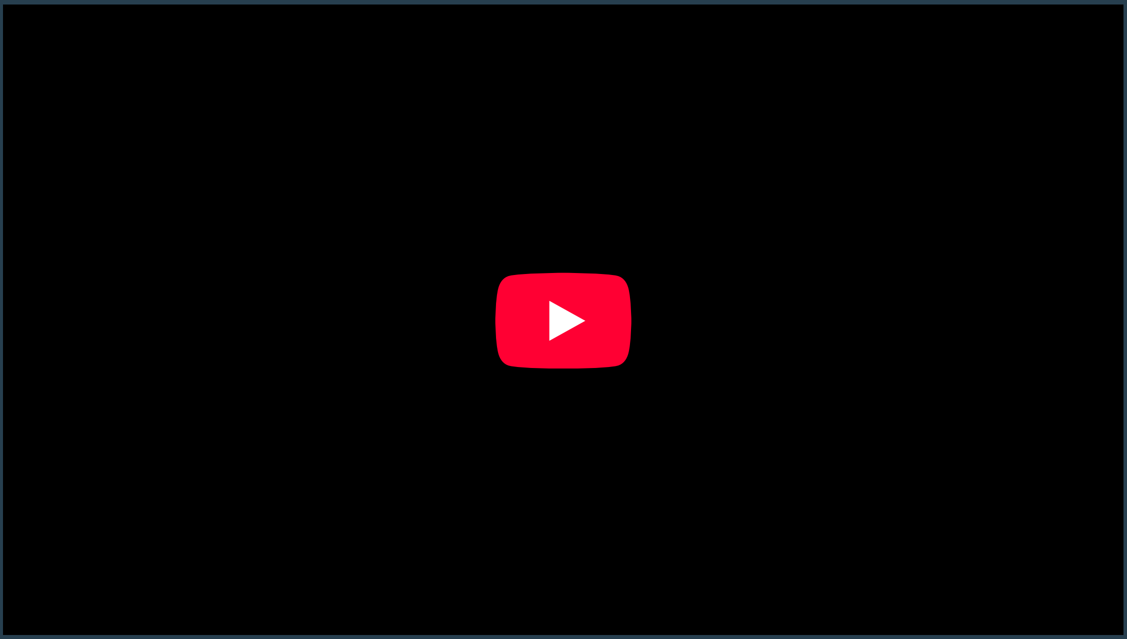
```
error_code error_code_from_errno(int);

class File_handle {
    FILE *p;
public:
    File_handle(const char *pp, const char *r, error_code& ec)
        { p = fopen(pp, r); if (p == 0) ec = error_code_from_errno(errno); }
    File_handle(const string& s, const char *r, error_code& ec)
        { p = fopen(s.c_str(), r); if (p == 0) ec = error_code_from_errno(errno); }
    ~File_handle() { if (p) fclose(p); }
};

void f(string s)
{
    std::error_code open_error;
    File_handle fh(s, "r", open_error);
    if (open_error) {
        fprintf(stderr, "failed to open file '%s', error=%d\n", s.c_str(), open_error.value());
        return;
    }
    // use fh
}
```

Now that we’ve ditched exceptions we’ve avoided the main problem with them. We no longer need RAISI and we are confident we’ve caught/handled the error. It’s still a little more noisy than the C code but it’s less noisy than the original one that used exceptions.

Now that we’ve looked at a couple practical techniques, let’s dream about how C++ could look. C++ exceptions actually have a few more problems that I havent mentioned. I highly recommend watching Herb Sutter’s talk about them here:



There’s also a paper you can read here:

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0709r4.pdf

... go on to talk about removing the required scope to avoid RAISI .. then go on to talk about compile-time enforcement of exception handling...