14.1 — The need for exceptions

BY ALEX ON OCTOBER 4TH, 2008 | LAST MODIFIED BY ALEX ON JULY 18TH, 2017

In the previous lesson on **handling errors**, we talked about ways to use assert(), cerr(), and exit() to handle errors. However, we punted on one further topic that we will now cover: exceptions.

When return codes fail

When writing reusable code, error handling is a necessity. One of the most common ways to handle potential errors is via return codes. For example:

```
int findFirstChar(const char* string, char ch)
{
    // Step through each character in string
    for (int index=0; index < strlen(string); ++index)
        // If the character matches ch, return its index
        if (string[index] == ch)
            return index;

    // If no match was found, return -1
    return -1;
}
```

This function returns the index of the first character matching ch within string. If the character can not be found, the function returns -1 as an error indicator.

The primary virtue of this approach is that it is extremely simple. However, using return codes has a number of drawbacks which can quickly become apparent when used in non-trivial cases:

First, return values can be cryptic -- if a function returns -1, is it trying to indicate an error, or is that actually a valid return value? It's often hard to tell without digging into the guts of the function.

Second, functions can only return one value, so what happens when you need to return both a function result and an error code? Consider the following function:

```
double divide(int x, int y)
{
    return static_cast<double>(x)/y;
}
```

This function is in desperate need of some error handling, because it will crash if the user passes in 0 for parameter y. However, it also needs to return the result of x/y. How can it do both? The most common answer is that either the result or the error handling will have to be passed back as a reference parameter, which makes for ugly code that is less convenient to use. For example:

```
#include <iostream>

double divide(int x, int y, bool &success)
{
    if (y == 0)
    {
        success = false;
        return 0.0;
    }

    success = true;
    return static_cast<double>(x)/y;
}

int main()
{
    bool success; // we must now pass in a bool value to see if the call was successful
    double result = divide(5, 3, success);

```

```
    if (!success) // and check it before we use the result
        std::cerr << "An error occurred" << std::endl;
    else
        cout << "The answer is " << result << '\n';
}
```

Third, in sequences of code where many things can go wrong, error codes have to be checked constantly. Consider the following snippet of code that involves parsing a text file for values that are supposed to be there:

```
    std::ifstream setupIni("setup.ini"); // open setup.ini for reading
    // If the file couldn't be opened (e.g. because it was missing) return some error enum
    if (!setupIni)
        return ERROR_OPENING_FILE;

    // Now read a bunch of values from a file
    if (!readIntegerFromFile(setupIni, m_firstParameter)) // try to read an integer from the file
        return ERROR_READING_VALUE; // Return enum value indicating value couldn't be read

    if (!readDoubleFromFile(setupIni, m_secondParameter)) // try to read a double from the file
        return ERROR_READING_VALUE;

    if (!readFloatFromFile(setupIni, m_thirdParameter)) // try to read a float from the file
        return ERROR_READING_VALUE;
```

We haven't covered file access yet, so don't worry if you don't understand how the above works -- just note the fact that every call requires an error-check and return back to the caller. Now imagine if there were twenty parameters of differing types -- you're essentially checking for an error and returning ERROR_READING_VALUE twenty times! All of this error checking and returning values makes determining *what* the function is trying to do much harder to discern.

Fourth, return codes do not mix with constructors very well. What happens if you're creating an object and something inside the constructor goes catastrophically wrong? Constructors have no return type to pass back a status indicator, and passing one back via a reference parameter is messy and must be explicitly checked. Furthermore, even if you do this, the object will still be created and then has to be dealt with or disposed of.

Finally, when an error code is returned to the caller, the caller may not always be equipped to handle the error. If the caller doesn't want to handle the error, it either has to ignore it (in which case it will be lost forever), or return the error up the stack to the function that called it. This can be messy and lead to many of the same issues noted above.

To summarize, the primary issue with return codes is that the error handling code ends up intricately linked to the normal control flow of the code. This in turns ends up constraining both how the code is laid out, and how errors can be reasonably handled.

Exceptions

Exception handling provides a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code. This allows more freedom to handle errors when and how ever is most useful for a given situation, alleviating many (if not all) of the messiness that return codes cause.

In the next lesson, we'll take a look at how exceptions work in C++.


14.2 -- Basic exception handling


Index


13.x -- Chapter 13 comprehensive quiz

Share this: