27.6 — Rethrowing exceptions

learncpp.com/cpp-tutorial/rethrowing-exceptions/

Occasionally you may run into a case where you want to catch an exception, but not want to (or have the ability to) fully handle it at the point where you catch it. This is common when you want to log an error, but pass the issue along to the caller to actually handle.

When a function can use a return code, this is simple. Consider the following example:

```
Database* createDatabase(std::string filename)
{
    Database* d {};

    try
    {
        d = new Database{};
        d->open(filename); // assume this throws an int exception on failure
        return d;
    }
    catch (int exception)
    {
        // Database creation failed
        delete d;
        // Write an error to some global logfile
        g_log.logError("Creation of Database failed");
    }

    return nullptr;
}
```

In the above code snippet, the function is tasked with creating a Database object, opening the database, and returning the Database object. In the case where something goes wrong (e.g. the wrong filename is passed in), the exception handler logs an error, and then reasonably returns a null pointer.

Now consider the following function:

```
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // throws int exception on failure
    }
    catch (int exception)
    {
        // Write an error to some global logfile
        g_log.logError("getIntValueFromDatabase failed");

        // However, we haven't actually handled this error
        // So what do we do here?
    }
}
```

In the case where this function succeeds, it returns an integer value -- any integer value could be a valid value.

But what about the case where something goes wrong with getIntValue()? In that case, getIntValue() will throw an integer exception, which will be caught by the catch block in getIntValueFromDatabase(), which will log the error. But then how do we tell the caller of getIntValueFromDatabase() that something went wrong? Unlike the top example, there isn't a good return code we can use here (because any integer return value could be a valid one).

Throwing a new exception

One obvious solution is to throw a new exception.

```
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // throws int exception on failure
    }
    catch (int exception)
    {
        // Write an error to some global logfile
        g_log.logError("getIntValueFromDatabase failed");

        throw 'q'; // throw char exception 'q' up the stack to be handled by caller
of getIntValueFromDatabase()
    }
}
```

In the example above, the program catches the int exception from getIntValue(), logs the error, and then throws a new exception with char value ‘q’. Although it may seem weird to throw an exception from a catch block, this is allowed. Remember, only exceptions thrown within a try block are eligible to be caught. This means that an exception thrown within a catch block will not be caught by the catch block it’s in. Instead, it will be propagated up the stack to the caller.

The exception thrown from the catch block can be an exception of any type -- it doesn’t need to be the same type as the exception that was just caught.

Rethrowing an exception (the wrong way)

Another option is to rethrow the same exception. One way to do this is as follows:

```
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // throws int exception on failure
    }
    catch (int exception)
    {
        // Write an error to some global logfile
        g_log.logError("getIntValueFromDatabase failed");

        throw exception;
    }
}
```

Although this works, this method has a couple of downsides. First, this doesn’t throw the exact same exception as the one that is caught -- rather, it throws a copy-initialized copy of variable exception. Although the compiler is free to elide the copy, it may not, so this could be less performant.

But significantly, consider what happens in the following case:

```
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // throws Derived exception on failure
    }
    catch (Base& exception)
    {
        // Write an error to some global logfile
        g_log.logError("getIntValueFromDatabase failed");

        throw exception; // Danger: this throws a Base object, not a Derived object
    }
}
```

In this case, getIntValue() throws a Derived object, but the catch block is catching a Base reference. This is fine, as we know we can have a Base reference to a Derived object. However, when we throw an exception, the thrown exception is copy-initialized from variable exception. Variable exception has type Base, so the copy-initialized exception also has type Base (not Derived!). In other words, our Derived object has been sliced!

You can see this in the following program:

```
#include <iostream>
class Base
{
public:
    Base() {}
    virtual void print() { std::cout << "Base"; }
};

class Derived: public Base
{
public:
    Derived() {}
    void print() override { std::cout << "Derived"; }
};

int main()
{
    try
    {
        try
        {
            throw Derived{};
        }
        catch (Base& b)
        {
            std::cout << "Caught Base b, which is actually a ";
            b.print();
            std::cout << '\n';
            throw b; // the Derived object gets sliced here
        }
    }
    catch (Base& b)
    {
        std::cout << "Caught Base b, which is actually a ";
        b.print();
        std::cout << '\n';
    }

    return 0;
}
```

This prints:

```
Caught Base b, which is actually a Derived
Caught Base b, which is actually a Base
```

The fact that the second line indicates that Base is actually a Base rather than a Derived proves that the Derived object was sliced.

Rethrowing an exception (the right way)

Fortunately, C++ provides a way to rethrow the exact same exception as the one that was just caught. To do so, simply use the throw keyword from within the catch block (with no associated variable), like so:

```
#include <iostream>
class Base
{
public:
    Base() {}
    virtual void print() { std::cout << "Base"; }
};

class Derived: public Base
{
public:
    Derived() {}
    void print() override { std::cout << "Derived"; }
};

int main()
{
    try
    {
        try
        {
            throw Derived{};
        }
        catch (Base& b)
        {
            std::cout << "Caught Base b, which is actually a ";
            b.print();
            std::cout << '\n';
            throw; // note: We're now rethrowing the object here
        }
    }
    catch (Base& b)
    {
        std::cout << "Caught Base b, which is actually a ";
        b.print();
        std::cout << '\n';
    }

    return 0;
}
```

This prints:

```
Caught Base b, which is actually a Derived
Caught Base b, which is actually a Derived
```

This throw keyword that doesn’t appear to throw anything in particular actually re-throws the exact same exception that was just caught. No copies are made, meaning we don’t have to worry about performance killing copies or slicing.

If rethrowing an exception is required, this method should be preferred over the alternatives.

Rule

When rethrowing the same exception, use the throw keyword by itself