

## 14.6 — Rethrowing exceptions

BY ALEX ON FEBRUARY 5TH, 2017 | LAST MODIFIED BY ALEX ON MAY 6TH, 2018

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:

```
1 Database* createDatabase(std::string filename)
2 {
3     try
4     {
5         Database *d = new Database(filename);
6         d->open(); // assume this throws an int exception on failure
7         return d;
8     }
9     catch (int exception)
10    {
11        // Database creation failed
12        // Write an error to some global logfile
13        g_log.logError("Creation of Database failed");
14    }
15
16    return nullptr;
17 }
```

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:

```
1 int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2 {
3     assert(d);
4
5     try
6     {
7         return d->getIntValue(table, key); // throws int exception on failure
8     }
9     catch (int exception)
10    {
11        // Write an error to some global logfile
12        g_log.logError("doSomethingImportant failed");
13
14        // However, we haven't actually handled this error
15        // So what do we do here?
16    }
17 }
```

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.

```
1 int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2 {
```

```

3     assert(d);
4
5     try
6     {
7         return d->getIntValue(table, key); // throws int exception on failure
8     }
9     catch (int exception)
10    {
11        // Write an error to some global logfile
12        g_log.logError("doSomethingImportant failed");
13
14        throw 'q'; // throw char exception 'q' up the stack to be handled by caller of getIntValueFromD
15    atabase()
16    }
17 }

```

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:

```

1  int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2  {
3      assert(d);
4
5      try
6      {
7          return d->getIntValue(table, key); // throws int exception on failure
8      }
9      catch (int exception)
10     {
11         // Write an error to some global logfile
12         g_log.logError("doSomethingImportant failed");
13
14         throw exception;
15     }
16 }

```

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:

```

1  int getIntValueFromDatabase(Database *d, std::string table, std::string key)
2  {
3      assert(d);
4
5      try
6      {
7          return d->getIntValue(table, key); // throws Derived exception on failure
8      }
9      catch (Base &exception)
10     {
11         // Write an error to some global logfile
12         g_log.logError("doSomethingImportant failed");
13
14         throw exception; // Danger: this throws a Base object, not a Derived object
15     }
16 }

```

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 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:

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      Base() {}
6      virtual void print() { std::cout << "Base"; }
7  };
8
9  class Derived: public Base
10 {
11 public:
12     Derived() {}
13     virtual void print() { std::cout << "Derived"; }
14 };
15
16 int main()
17 {
18     try
19     {
20         try
21         {
22             throw Derived();
23         }
24         catch (Base& b)
25         {
26             std::cout << "Caught Base b, which is actually a ";
27             b.print();
28             std::cout << "\n";
29             throw b; // the Derived object gets sliced here
30         }
31     }
32     catch (Base& b)
33     {
34         std::cout << "Caught Base b, which is actually a ";
35         b.print();
36         std::cout << "\n";
37     }
38
39     return 0;
40 }
```

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:

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      Base() {}
6      virtual void print() { std::cout << "Base"; }
```

```

7   };
8
9   class Derived: public Base
10  {
11  public:
12      Derived() {}
13      virtual void print() { std::cout << "Derived"; }
14  };
15
16  int main()
17  {
18      try
19      {
20          try
21          {
22              throw Derived();
23          }
24          catch (Base& b)
25          {
26              std::cout << "Caught Base b, which is actually a ";
27              b.print();
28              std::cout << "\n";
29              throw; // note: We're now rethrowing the object here
30          }
31      }
32      catch (Base& b)
33      {
34          std::cout << "Caught Base b, which is actually a ";
35          b.print();
36          std::cout << "\n";
37      }
38
39      return 0;
40  }

```

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.*



[14.7 -- Function try blocks](#)



[Index](#)



[14.5 -- Exceptions, classes, and inheritance](#)

Share this:



Facebook



Twitter



G+ Google



Pinterest