14.8 — Exception dangers and downsides

BY ALEX ON OCTOBER 26TH, 2008 | LAST MODIFIED BY ALEX ON AUGUST 16TH, 2017

As with almost everything that has benefits, there are some potential downsides to exceptions as well. This article is not meant to be comprehensive, but just to point out some of the major issues that should be considered when using exceptions (or deciding whether to use them).

Cleaning up resources

One of the biggest problems that new programmers run into when using exceptions is the issue of cleaning up resources when an exception occurs. Consider the following example:

```
try
{
    openFile(filename);
    writeFile(filename, data);
    closeFile(filename);
}
catch (FileException &exception)
{
    cerr << "Failed to write to file: " << exception.what() << std::endl;
}
```

What happens if WriteFile() fails and throws a FileException? At this point, we've already opened the file, and now control flow jumps to the FileException handler, which prints an error and exits. Note that the file was never closed! This example should be rewritten as follows:

```
try
{
    openFile(filename);
    writeFile(filename, data);
    closeFile(filename);
}
catch (FileException &exception)
{
    // Make sure file is closed
    closeFile(filename);
    // Then write error
    cerr << "Failed to write to file: " << exception.what() << std::endl;
}
```

This kind of error often crops up in another form when dealing with dynamically allocated memory:

```
try
{
    Person *john = new Person("John", 18, PERSON_MALE);
    processPerson(john);
    delete john;
}
catch (PersonException &exception)
{
    cerr << "Failed to process person: " << exception.what() << '\n';
}
```

If processPerson() throws an exception, control flow jumps to the catch handler. As a result, john is never deallocated! This example is a little more tricky than the previous one -- because john is local to the try block, it goes out of scope when the try block exits. That means the exception handler can not access john at all (its been destroyed already), so there's no way for it to deallocate the memory.

However, there are two relatively easy ways to fix this. First, declare john outside of the try block so it does not go out of scope when the try block exits:

```
Person *john = NULL;
try
```

```
{
    john = new Person("John", 18, PERSON_MALE);
    processPerson(john);
    delete john;
}
catch (PersonException &exception)
{
    delete john;
    cerr << "Failed to process person: " << exception.what() << '\n';
}
```

Because john is declared outside the try block, it is accessible both within the try block and the catch handlers. This means the catch handler can do cleanup properly.

The second way is to use a local variable of a class that knows how to cleanup itself when it goes out of scope (often called a “smart pointer”. The standard library provides a class called std::unique_ptr that can be used for this purpose. **std::unique_ptr** is a template class that holds a pointer, and deallocates it when it goes out of scope.

```
#include <memory>; // for std::unique_ptr

try
{
    Person *john = new Person("John", 18, PERSON_MALE);
    unique_ptr<Person> upJohn(john); // upJohn now owns john

    ProcessPerson(john);

    // when upJohn goes out of scope, it will delete john
}
catch (PersonException &exception)
{
    cerr << "Failed to process person: " << exception.what() << '\n';
}
```

We’ll talk more about smart pointers in the next chapter.

Exceptions and destructors

Unlike constructors, where throwing exceptions can be a useful way to indicate that object creation did not succeed, exceptions should *never* be thrown in destructors.

The problem occurs when an exception is thrown from a destructor during the stack unwinding process. If that happens, the compiler is put in a situation where it doesn’t know whether to continue the stack unwinding process or handle the new exception. The end result is that your program will be terminated immediately.

Consequently, the best course of action is just to abstain from using exceptions in destructors altogether. Write a message to a log file instead.

Performance concerns

Exceptions do come with a small performance price to pay. They increase the size of your executable, and they may also cause it to run slower due to the additional checking that has to be performed. However, the main performance penalty for exceptions happens when an exception is actually thrown. In this case, the stack must be unwound and an appropriate exception handler found, which is a relatively expensive operation.

As a note, some modern computer architectures support an exception model called zero-cost exceptions. Zero-cost exceptions, if supported, have no additional runtime cost in the non-error case (which is the case we most care about performance). However, they incur an even larger penalty in the case where an exception is found.

So when should I use exceptions?

Exception handling is best used when all of the following are true:

- The error being handled is likely to occur only infrequently.
- The error is serious and execution could not continue otherwise.
- The error cannot be handled at the place where it occurs.

- There isn't a good alternative way to return an error code back to the caller.

As an example, let's consider the case where you've written a function that expects the user to pass in the name of a file on disk. Your function will open this file, read some data, close the file, and pass back some result to the caller. Now, let's say the user passes in the name of a file that doesn't exist, or a null string. Is this a good candidate for an exception?

In this case, the first two bullets above are trivially met -- this isn't something that's going to happen often, and your function can't calculate a result when it doesn't have any data to work with. The function can't handle the error either -- it's not the job of the function to re-prompt the user for a new filename, and that might not even be appropriate, depending on how your program is designed. The fourth bullet is the key -- is there a good alternative way to return an error code back to the caller? It depends on the details of your program. If so (e.g. you can return a null pointer, or a status code to indicate failure), that's probably the better choice. If not, then an exception would be reasonable.


14.x -- Chapter 14 comprehensive quiz


Index


14.7 -- Function try blocks

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL C++, DESTRUCTORS, EXCEPTIONS, PROGRAMMING, TUTORIAL | PRINT THIS POST

26 comments to 14.8 — Exception dangers and downsides

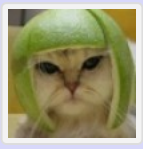
Sean Kelly
August 15, 2017 at 7:40 am · Reply

Hello Alex!

Just a question for clarification when you said:
"exception handling is best used for truly exceptional cases and catastrophic errors, not for routine error handling."

Are you talking about things that cause the program to crash specifically? Like trying to access memory that is out of range in an array? Where do you draw the line when it comes to using expectations rather than checks to fix problems?

Thanks!



Alex
August 16, 2017 at 10:27 am · Reply

I've updated the lesson to try to provide some additional guidance as to when use of exceptions is reasonable. Have a read and see if that helps clarify.

Sean Kelly
August 17, 2017 at 6:36 pm · Reply

Nice! I've given it a read and it makes more sense to me, thank you.