14.4 — Uncaught exceptions, catch-all handlers, and exception specifiers

BY ALEX ON OCTOBER 25TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 19TH, 2017

By now, you should have a reasonable idea of how exceptions work. In this lesson, we'll cover a few more interesting exception cases.

Uncaught exceptions

In the past few examples, there are quite a few cases where a function assumes its caller (or another function somewhere up the call stack) will handle the exception. In the following example, mySqrt() assumes someone will handle the exception that it throws -- but what happens if nobody actually does?

Here's our square root program again, minus the try block in main():

```
#include <iostream>
#include <cmath> // for sqrt() function

// A modular square root function
double mySqrt(double x)
{
    // If the user entered a negative number, this is an error condition
    if (x < 0.0)
        throw "Can not take sqrt of negative number"; // throw exception of type const char*

    return sqrt(x);
}

int main()
{
    std::cout << "Enter a number: ";
    double x;
    std::cin >> x;

    // Look ma, no exception handler!
    std::cout << "The sqrt of " << x << " is " << mySqrt(x) << '\n';

    return 0;
}
```

Now, let's say the user enters -4, and mySqrt(-4) raises an exception. Function mySqrt() doesn't handle the exception, so the program stack unwinds and control returns to main(). But there's no exception handler here either, so main() terminates. At this point, we just terminated our application!

When main() terminates with an unhandled exception, the operating system will generally notify you that an unhandled exception error has occurred. How it does this depends on the operating system, but possibilities include printing an error message, popping up an error dialog, or simply crashing. Some OSes are less graceful than others. Generally this is something you want to avoid altogether!

Catch-all handlers

And now we find ourselves in a conundrum: functions can potentially throw exceptions of any data type, and if an exception is not caught, it will propagate to the top of your program and cause it to terminate. Since it's possible to call functions without knowing how they are even implemented (and thus, what type of exceptions they may throw), how can we possibly prevent this from happening?

Fortunately, C++ provides us with a mechanism to catch all types of exceptions. This is known as a **catch-all handler**. A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch.

If you recall from lesson 7.14 on **ellipses and why to avoid them**, ellipses were previously used to pass arguments of any type to a function. In this context, they represent exceptions of any data type. Here's an simple example:

```
#include <iostream>

int main()
```

```
{
    try
    {
        throw 5; // throw an int exception
    }
    catch (double x)
    {
        std::cout << "We caught an exception of type double: " << x << '\n';
    }
    catch (...) // catch-all handler
    {
        std::cout << "We caught an exception of an undetermined type\n";
    }
}
```

Because there is no specific exception handler for type int, the catch-all handler catches this exception. This example produces the following result:

```
We caught an exception of an undetermined type
```

The catch-all handler should be placed last in the catch block chain. This is to ensure that exceptions can be caught by exception handlers tailored to specific data types if those handlers exist. Visual Studio enforces this constraint -- I am unsure if other compilers do. (Per reader Lonami in the comments below, GCC does too).

Often, the catch-all handler block is left empty:

```
catch(...) {} // ignore any unanticipated exceptions
```

This will catch any unanticipated exceptions and prevent them from stack unwinding to the top of your program, but does no specific error handling.

Using the catch-all handler to wrap main()

One interesting use for the catch-all handler is to wrap the contents of main():

```
#include <iostream>
int main()
{

    try
    {
        runGame();
    }
    catch(...)
    {
        std::cerr << "Abnormal termination\n";
    }

    saveState(); // Save user's game
    return 1;
}
```

In this case, if runGame() or any of the functions it calls throws an exception that is not caught, that exception will unwind up the stack and eventually get caught by this catch-all handler. This will prevent main() from terminating, and gives us a chance to print an error of our choosing and then save the user's state before exiting. This can be useful to catch and handle problems that may be unanticipated.

Exception specifiers

This subsection should be considered optional reading because exception specifiers are rarely used in practice, are not well supported by compilers, and Bjarne Stroustrup (the creator of C++) considers them a failed experiment.

Exception specifiers are a mechanism that allows us to use a function declaration to specify whether a function may or will not throw exceptions. This can be useful in determining whether a function call needs to be put inside a try block or not.

There are three types of exception specifiers, all of which use what is called the **throw (...)** syntax.

First, we can use an empty throw statement to denote that a function does not throw any exceptions outside of itself:

```
1 int doSomething() throw(); // does not throw exceptions
```

Note that doSomething() can still use exceptions as long as they are handled internally. Any function that is declared with throw() is supposed to cause the program to terminate immediately if it does try to throw an exception outside of itself, but implementation is spotty.

Second, we can use a specific throw statement to denote that a function may throw a particular type of exception:

```
1 int doSomething() throw(double); // may throw a double
```

Finally, we can use a catch-all throw statement to denote that a function may throw an unspecified type of exception:

```
1 int doSomething() throw(...); // may throw anything
```

Due to the incomplete compiler implementation, the fact that exception specifiers are more like statements of intent than guarantees, some incompatibility with template functions, and the fact that most C++ programmers are unaware of their existence, I recommend you do not bother using exception specifiers.


14.5 -- Exceptions, classes, and inheritance


Index


14.3 -- Exceptions, functions, and stack unwinding

Share this:

C++ TUTORIAL C++, CATCH-ALL HANDLER, EXCEPTION SPECIFIERS, EXCEPTIONS, PROGRAMMING, TUTORIAL | PRINT THIS POST

23 comments to 14.4 — Uncaught exceptions, catch-all handlers, and exception specifiers

Rev
July 5, 2018 at 12:32 am · Reply

Just a comment about fact of exception specifiers. I by chance opened a library file of DevC++ and saw the creators added throw() to the constructors of the auto_ptr class. So what you said that not many programmers knew about it is not quite true ☺

Peter Baum
May 7, 2018 at 2:51 pm · Reply

I'm actually back at the end of chapter 6 and just finished my blackjack program. I ended up here because I needed exception handling for error checking purposes. In particular, gracefully catching conversion errors from stoi() and similar situations. It seems to me that the sections on exception handling are placed way too late in the lesson sequence. By the time a student reaches the end of chapter 6, they should be programming relatively sophisticated programs, and the basics of exception handling should certainly be taught before objects. You know... IMHO.