27.3 — Exceptions, functions, and stack unwinding

In the previous lesson on 27.2 -- Basic exception handling, we explained how throw, try, and catch work together to enable exception handling. In this lesson, we'll talk about how exception handling interacts with functions.

Throwing exceptions from a called function

In the prior lesson, we noted, "a try block detects any exceptions that are thrown by statements within the try block". In the corresponding examples, our throw statements were placed within a try block, and caught by an associated catch block, all within the same function. Having to both throw and catch exceptions within a single function is of limited value.

Of more interest is what happens if a statement inside a try block is a function call, and the called function throws an exception. Will the try block detect an exception thrown by a function that is called from within the try block?

The answer, fortunately, is yes!

One of the most useful properties of exception handling is that the throw statements do NOT have to be placed directly inside a try block. Instead, exceptions can be thrown from anywhere in a function, and those exceptions can be caught by the try block of the caller (or the caller's caller, etc…). When an exception is caught in this manner, execution jumps from the point where the exception is thrown to the catch block handling the exception.

Key insight

Try blocks catch exceptions not only from statements within the try block, but also from functions that are called within the try block.

This allows us to use exception handling in a much more modular fashion. We'll demonstrate this by rewriting the square root program from the previous lesson to use a modular function.

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

// 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 std::sqrt(x);
}

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

    try // Look for exceptions that occur within try block and route to attached
catch block(s)
    {
        double d = mySqrt(x);
        std::cout << "The sqrt of " << x << " is " << d << '\n';
    }
    catch (const char* exception) // catch exceptions of type const char*
    {
        std::cerr << "Error: " << exception << std::endl;
    }

    return 0;
}
```

In this program, we've taken the code that checks for an exception and calculates the square root and put it inside a modular function called mySqrt(). We've then called this mySqrt() function from inside a try block. Let's verify that it still works as expected:

```
Enter a number: -4
Error: Can not take sqrt of negative number
```

It does! When the exception is thrown within mySqrt(), there is no handler in mySqrt() to handle the exception. However, the call to mySqrt() (in main()) is within a try block that has an associated matching exception handler. Therefore execution jumps from the throw statement in mySqrt() to the top of the catch block in main() and then resumes.

The most interesting part of the above program is that the mySqrt() function can throw an exception, but does not handle this exception itself! This essentially means mySqrt() is willing to say, "Hey, there's a problem!", but is unwilling to handle the problem itself. It is, in essence, delegating the responsibility for handling the exception to its caller (the equivalent of how using a return code passes the responsibility of handling an error back to a function’s caller).

At this point, some of you are probably wondering why it’s a good idea to pass errors back to the caller. Why not just make MySqrt() handle its own error? The problem is that different applications may want to handle errors in different ways. A console application may want to print a text message. A windows application may want to pop up an error dialog. In one application, this may be a fatal error, and in another application it may not be. By passing the error out of the function, each application can handle an error from mySqrt() in a way that is the most context appropriate for it! Ultimately, this keeps mySqrt() as modular as possible, and the error handling can be placed in the less-modular parts of the code.

Exception handling and stack unwinding

In this section, we’ll take a look at how exception handling actually works when multiple functions are involved.

Related content

Before proceeding, see lesson 20.2 -- The stack and the heap if you need a refresher on call stacks and stack unwinding.

When an exception is thrown, the program first looks to see if the exception can be handled immediately inside the current function (meaning the exception was thrown within a try block inside the current function, and there is a corresponding catch block associated). If the current function can handle the exception, it does so.

If not, the program next checks whether the function’s caller (the next function up the call stack) can handle the exception. In order for the function’s caller to handle the exception, the call to the current function must be inside a try block, and a matching catch block must be associated. If no match is found, then the caller’s caller (two functions up the call stack) is checked. Similarly, in order for the caller’s caller to handle the exception, the call to the caller must be inside a try block, and a matching catch block must be associated.

The process of checking each function up the call stack continues until either a handler is found, or all of the functions on the call stack have been checked and no handler can be found.

If a matching exception handler is found, then execution jumps from the point where the exception is thrown to the top of the matching catch block. This requires unwinding the stack (removing the current function from the call stack) as many times as necessary to make the function handling the exception the top function on the call stack.

If no matching exception handler is found, the stack may or may not be unwound. We will talk more about this case in the next lesson (27.4 -- Uncaught exceptions and catch-all handlers).

When the current function is removed from the call stack, all local variables are destroyed as usual, but no value is returned.

Key insight

Unwinding the stack destroys local variables in the functions that are unwound (which is good, because it ensures their destructors execute).

Another stack unwinding example

To illustrate the above, let’s take a look at a more complex example, using a larger stack. Although this program is long, it’s pretty simple: main() calls A(), A() calls B(), B() calls C(), C() calls D(), and D() throws an exception.

```
#include <iostream>

void D() // called by C()
{
    std::cout << "Start D\n";
    std::cout << "D throwing int exception\n";

    throw - 1;

    std::cout << "End D\n"; // skipped over
}

void C() // called by B()
{
    std::cout << "Start C\n";
    D();
    std::cout << "End C\n";
}

void B() // called by A()
{
    std::cout << "Start B\n";

    try
    {
        C();
    }
    catch (double) // not caught: exception type mismatch
    {
        std::cerr << "B caught double exception\n";
    }

    try
    {
    }
    catch (int) // not caught: exception not thrown within try
    {
        std::cerr << "B caught int exception\n";
    }

    std::cout << "End B\n";
}

void A() // called by main()
{
    std::cout << "Start A\n";

    try
    {
        B();
    }
    catch (int) // exception caught here and handled
    {
        std::cerr << "A caught int exception\n";
    }

    std::cout << "End A\n";
}

int main()
{
    std::cout << "Start main\n";

    try
    {
        A();
    }
    catch (int) // not called because exception was handled by A
    {
        std::cerr << "main caught int exception\n";
    }
    std::cout << "End main\n";

    return 0;
}
```

Take a look at this program in more detail, and see if you can figure out what gets printed and what doesn't when it is run. The answer follows:

```
Start main
Start A
Start B
Start C
Start D
D throwing int exception
A caught int exception
End A
End main
```

Let's examine what happens in this case. The printing of all the "Start" statements is straightforward and doesn't warrant further explanation. Function D() prints "D throwing int exception" and then throws an int exception. This is where things start to get interesting.

Because D() doesn't handle the exception itself, its callers (the functions up the call stack) are checked to see if one of them can handle the exception. Function C() doesn't handle any exceptions, so no match is found there.

Function B() has a two try blocks. The try block containing the call to C() has a handler for exceptions of type double, but that does not match our exception of type int (and exceptions do not do type conversion), so no match is found. The empty try block does have an exception handler for exceptions of type int, but this catch block is not considered a match because the call to C() is not within the associated try block.

A() also has a try block, and the call to B() is within it, so the program looks to see if there is a catch handler for int exceptions. There is! Consequently, A() handles the exception, and prints “A caught int exception”.

Because the exception has now been handled, control continues normally at the end of the catch block within A(). This means A() prints “End A” and then terminates normally.

Control returns to main(). Although main() has an exception handler for int, our exception has already been handled by A(), so the catch block within main() does not get executed. main() simply prints “End main” and then terminates normally.

There are quite a few interesting principles illustrated by this program:

First, the immediate caller of a function that throws an exception doesn’t have to handle the exception if it doesn’t want to. In this case, C() didn’t handle the exception thrown by D(). It delegated that responsibility to one of its callers up the stack.

Second, if a try block doesn’t have a catch handler for the type of exception being thrown, stack unwinding occurs just as if there were no try block at all. In this case, B() didn’t handle the exception either because it didn’t have the right kind of catch block.

Third, if a function has a matching catch block but the call to the current function did not occur within the associated try block, that catch block isn’t used. We also saw this in B().

Finally, once an exception is handled, control flow proceeds as normal starting from the end of the matching catch block. This was demonstrated by A() handling the error and then terminating normally. By the time the program got back to main(), the exception had been thrown and handled already -- main() had no idea there even was an exception at all!

As you can see, stack unwinding provides us with some very useful behavior -- if a function does not want to handle an exception, it doesn’t have to. The exception will propagate up the stack until it finds someone who will! This allows us to decide where in the call stack is the most appropriate place to handle any errors that may occur.

In the next lesson, we’ll take a look at what happens when you don’t capture an exception, and a method to prevent that from happening.