20.3 — Exceptions, functions, and stack unwinding

ALEX AUGUST 25, 2021

In the previous lesson on 20.2 -- 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 outside of a try block

In the examples in the previous lesson, the throw statements were placed directly within a try block. If this were a necessity, exception handling would be of limited use.

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 due to the way exceptions propagate up the stack when thrown. 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!

Let's revisit for a moment what happens when an exception is raised. First, the program looks to see if the exception can be handled immediately (which means it was thrown inside a try block). If not, the current function is terminated, and the program checks to see if the function's caller will handle the exception. If not, it terminates the caller and checks the caller's caller. Each function is terminated in sequence until a handler for the exception is found, or until main() is terminated without the exception being handled. This process is called **unwinding the stack** (see the lesson on the stack and the heap if you need a refresher on what the call stack is).

Now, let's take a detailed look at how that applies to this program when an exception is raised from within mySqrt(). First, the program checks to see if the exception was thrown from within a try block inside the function. In this case, it was not. Then, the stack begins to unwind. First, mySqrt() terminates, and control returns to main(). The program now checks to see if we're inside a try block. We are, and there's a const char* handler, so the exception is handled by the try block within main().

To summarize, mySqrt() raised the exception, but the try/catch block in main() was the one who captured and handled the exception. Or, put another way, try blocks catch exceptions not only from statements within the try block, but also from functions that are called within the try block.

The most interesting part of the above program is that the mySqrt() function can throw an exception, but this exception is not immediately inside of a try block! 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 back up the stack, 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.

Another stack unwinding example

Here's another example showing stack unwinding in practice, using a larger stack. Although this program is long, it's pretty simple: main() calls first(), first() calls second(), second() calls third(), third() calls last(), and last() throws an exception.

```cpp
#include <iostream>

void last() // called by third()
{
    std::cout << "Start last\n";
    std::cout << "last throwing int exception\n";
    throw -1;
    std::cout << "End last\n";

}

void third() // called by second()
{
    std::cout << "Start third\n";
    last();
    std::cout << "End third\n";
}

void second() // called by first()
{
    std::cout << "Start second\n";
    try
    {
        third();
    }
    catch(double)
    {
         std::cerr << "second caught double exception\n";
    }
    std::cout << "End second\n";
}

void first() // called by main()
{
    std::cout << "Start first\n";
    try
    {
        second();
    }
    catch (int)
    {
         std::cerr << "first caught int exception\n";
    }
    catch (double)
    {
         std::cerr << "first caught double exception\n";
    }
    std::cout << "End first\n";
}

int main()
{
    std::cout << "Start main\n";
    try
    {
        first();
    }
    catch (int)
    {
         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 first
Start second
Start third
Start last
last throwing int exception
first caught int exception
End first
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 last() prints "last throwing int exception" and then throws an int exception. This is where things start to get interesting.

Because last() doesn't handle the exception itself, the stack begins to unwind. Function last() terminates immediately and control returns to the caller, which is third().

Function third() doesn't handle any exceptions, so it terminates immediately and control returns to second().

Function second() has a try block, and the call to third() is within it, so the program attempts to match the exception with an appropriate catch block. However, there are no handlers for exceptions of type int here, so second() terminates immediately and control returns to first(). Note that the integer exception is not implicitly converted to match the catch block handling a double.

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

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

Control returns to main(). Although main() has an exception handler for int, our exception has already been handled by first(), 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, third() didn't handle the exception thrown by last(). 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, second() didn't handle the exception either because it didn't have the right kind of catch block.

Third, once an exception is handled, control flow proceeds as normal starting from the end of the catch blocks. This was demonstrated by first() 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.

Next lesson
20.4 Uncaught exceptions and catch-all handlers

Back to table of contents

Previous lesson
20.2 Basic exception handling

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies:

POST COMMENT

DP N N FOUT

Newest