27.4 — Uncaught exceptions and catch-all handlers

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

When a function throws an exception that it does not handle itself, it is making the assumption that a function somewhere down 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 std::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 looks to see if some function down the call stack will handle the exception. main() does not have a handler for this exception either, so no handler can be found.

When no exception handler for a function can be found, std::terminate() is called, and the application is terminated. In such cases, the call stack may or may not be unwound! If the stack is not unwound, local variables will not be destroyed, and any cleanup expected upon destruction of said variables will not happen!

Warning

The call stack may or may not be unwound if an exception is unhandled.

If the stack is not unwound, local variables will not be destroyed, which may cause problems if those variables have non-trivial destructors.

As an aside…

Although it might seem strange to not unwind the stack in such a case, there is a good reason for not doing so. An unhandled exception is generally something you want to avoid at all costs. If the stack were unwound, then all of the debug information about the state of the stack that led up to the throwing of the unhandled exception would be lost! By not unwinding, we preserve that information, making it easier to determine how an unhandled exception was thrown, and fix it.

When an exception is unhandled, 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!

Catch-all handlers

And now we find ourselves in a conundrum:

- Functions can potentially throw exceptions of any data type (including program-defined data types), meaning there is an infinite number of possible exception types to catch
- If an exception is not caught, your program will terminate immediately (and the stack may not be unwound, so your program may not even clean up after itself properly).
- Adding explicit catch handlers for every possible type is tedious, especially for the ones that are expected to be reached only in exceptional cases!

Fortunately, C++ also 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. For this reason, the catch-all handler is also sometimes called an “ellipsis catch handler”

If you recall from lesson 20.5 -- Ellipsis (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 must 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.

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

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

This will catch any unanticipated exceptions, ensuring that stack unwinding occurs up to this point and preventing the program from terminating, but does no specific error handling.

Using the catch-all handler to wrap main()

One common 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 handled, it will be caught by this catch-all handler. The stack will be unwound in an orderly manner (ensuring destruction of local variables). This will also prevent the program from terminating immediately, giving us a chance to print an error of our choosing and save the user's state before exiting.

The downside of this approach is that if an unhandled exception does occur, stack unwinding will occur, making it harder to determine why the unhandled exception was thrown in the first place. For this reason, using a catch-all handler in main is often a good idea for production applications, but disabled (using conditional compilation directives) in debug builds.

Best practice

If your program uses exceptions, consider using a catch-all handler in main, to help ensure orderly behavior when an unhandled exception occurs. Also consider disabling the catch-all handler for debug builds, to make it easier to identify how unhandled exceptions are occurring.