20.4 — Uncaught exceptions and catch-all handlers

ALEX AUGUST 25, 2021

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. For this reason, the catch-all handler is also sometimes called an "ellipsis catch handler"

If you recall from lesson 11.12 -- 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 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():

```cpp
#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 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.

Next lesson

20.5 Exceptions, classes, and inheritance

Back to table of contents

Previous lesson

20.3 Exceptions, functions, and stack unwinding

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