27.2 — Basic exception handling

In the previous lesson on the need for exceptions, we talked about how using return codes causes your control flow and error flow to be intermingled, constraining both. Exceptions in C++ are implemented using three keywords that work in conjunction with each other: **throw**, **try**, and **catch**.

Throwing exceptions

We use signals all the time in real life to note that particular events have occurred. For example, during American football, if a player has committed a foul, the referee will throw a flag on the ground and whistle the play dead. A penalty is then assessed and executed. Once the penalty has been taken care of, play generally resumes as normal.

In C++, a **throw statement** is used to signal that an exception or error case has occurred (think of throwing a penalty flag). Signaling that an exception has occurred is also commonly called **raising** an exception.

To use a throw statement, simply use the throw keyword, followed by a value of any data type you wish to use to signal that an error has occurred. Typically, this value will be an error code, a description of the problem, or a custom exception class.

Here are some examples:

```
throw -1; // throw a literal integer value
throw ENUM_INVALID_INDEX; // throw an enum value
throw "Can not take square root of negative number"; // throw a literal C-style
(const char*) string
throw dX; // throw a double variable that was previously defined
throw MyException("Fatal Error"); // Throw an object of class MyException
```

Each of these statements acts as a signal that some kind of problem that needs to be handled has occurred.

Looking for exceptions

Throwing exceptions is only one part of the exception handling process. Let’s go back to our American football analogy: once a referee has thrown a penalty flag, what happens next? The players notice that a penalty has occurred and stop play. The normal flow of the football game is disrupted.

In C++, we use the **try** keyword to define a block of statements (called a **try block**). The try block acts as an observer, looking for any exceptions that are thrown by any of the statements within the try block.

Here's an example of a try block:

```
try
{
    // Statements that may throw exceptions you want to handle go here
    throw -1; // here's a trivial throw statement
}
```

Note that the try block doesn't define HOW we're going to handle the exception. It merely tells the program, "Hey, if any of the statements inside this try block throws an exception, grab it!".

Handling exceptions

Finally, the end of our American football analogy: After the penalty has been called and play has stopped, the referee assesses the penalty and executes it. In other words, the penalty must be handled before normal play can resume.

Actually handling exceptions is the job of the catch block(s). The **catch** keyword is used to define a block of code (called a **catch block**) that handles exceptions for a single data type.

Here's an example of a catch block that will catch integer exceptions:

```
catch (int x)
{
    // Handle an exception of type int here
    std::cerr << "We caught an int exception with value" << x << '\n';
}
```

Try blocks and catch blocks work together -- a try block detects any exceptions that are thrown by statements within the try block, and routes them to a catch block with a matching type for handling. A try block must have at least one catch block immediately following it, but may have multiple catch blocks listed in sequence.

Once an exception has been caught by the try block and routed to a catch block for handling, the exception is considered handled, and execution will resume as normal after the catch block.

Catch parameters work just like function parameters, with the parameter being available within the subsequent catch block. Exceptions of fundamental types can be caught by value, but exceptions of non-fundamental types should be caught by const reference to avoid making an unnecessary copy (and, in some cases, to prevent slicing).

Just like with functions, if the parameter is not going to be used in the catch block, the variable name can be omitted:

```
catch (double) // note: no variable name since we don't use it in the catch block
below
{
    // Handle exception of type double here
    std::cerr << "We caught an exception of type double" << '\n';
}
```

This can help prevent compiler warnings about unused variables.

No type conversion is done for exceptions (so an int exception will not be converted to match a catch block with a double parameter).

Putting throw, try, and catch together

Here’s a full program that uses throw, try, and multiple catch blocks:

```
#include <iostream>
#include <string>

int main()
{
    try
    {
        // Statements that may throw exceptions you want to handle go here
        throw -1; // here's a trivial example
    }
    catch (int x)
    {
        // Any exceptions of type int thrown within the above try block get sent here
        std::cerr << "We caught an int exception with value: " << x << '\n';
    }
    catch (double) // no variable name since we don't use the exception itself in the
catch block below
    {
        // Any exceptions of type double thrown within the above try block get sent
here
        std::cerr << "We caught an exception of type double" << '\n';
    }
    catch (const std::string&) // catch classes by const reference
    {
        // Any exceptions of type std::string thrown within the above try block get
sent here
        std::cerr << "We caught an exception of type std::string" << '\n';
    }

    std::cout << "Continuing on our merry way\n";

    return 0;
}
```

Running the above try/catch block would produce the following result:

```
We caught an int exception with value -1
Continuing on our merry way
```

A throw statement was used to raise an exception with the value -1, which is of type int. The throw statement was then caught by the enclosing try block, and routed to the appropriate catch block that handles exceptions of type int. This catch block printed the appropriate error message.

Once the exception was handled, the program continued as normal after the catch blocks, printing “Continuing on our merry way”.

Recapping exception handling

Exception handling is actually quite simple, and the following two paragraphs cover most of what you need to remember about it:

When an exception is raised (using **throw**), the compiler looks in the nearest enclosing **try** block (propagating up the stack if necessary to find an enclosing try block -- we'll discuss this in more detail next lesson) to see if any of the **catch** handlers attached to the try block can handle that type of exception. If so, execution jumps to the top of the catch block, the exception is considered handled.

If no appropriate catch handlers exist in the nearest enclosing try block, the compiler continues to look at subsequent enclosing try blocks for a catch handler. If no appropriate catch handlers can be found before the end of the program, the program will fail with an exception error.

Note that the compiler will not perform implicit conversions or promotions when matching exceptions with catch blocks! For example, a char exception will not match with an int catch block. An int exception will not match a float catch block. However, casts from a derived class to one of its parent classes will be performed.

That's really all there is to it. The rest of this chapter will be dedicated to showing examples of these principles at work.

Exceptions are handled immediately

Here's a short program that demonstrates how exceptions are handled immediately:

```
#include <iostream>

int main()
{
    try
    {
        throw 4.5; // throw exception of type double
        std::cout << "This never prints\n";
    }
    catch (double x) // handle exception of type double
    {
        std::cerr << "We caught a double of value: " << x << '\n';
    }

    return 0;
}
```

This program is about as simple as it gets. Here's what happens: the throw statement is the first statement that gets executed -- this causes an exception of type double to be raised. Execution *immediately* moves to the nearest enclosing try block, which is the only try block in

this program. The catch handlers are then checked to see if any handler matches. Our exception is of type double, so we're looking for a catch handler of type double. We have one, so it executes.

Consequently, the result of this program is as follows:

```
We caught a double of value: 4.5
```

Note that "This never prints" is never printed, because the exception caused the execution path to jump immediately to the exception handler for doubles.

A more realistic example

Let's take a look at an example that's not quite so academic:

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

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)
    {
        // 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*

        // Otherwise, print the answer
        std::cout << "The sqrt of " << x << " is " << std::sqrt(x) << '\n';
    }
    catch (const char* exception) // catch exceptions of type const char*
    {
        std::cerr << "Error: " << exception << '\n';
    }
}
```

In this code, the user is asked to enter a number. If they enter a positive number, the if statement does not execute, no exception is thrown, and the square root of the number is printed. Because no exception is thrown in this case, the code inside the catch block never executes. The result is something like this:

```
Enter a number: 9
The sqrt of 9 is 3
```

If the user enters a negative number, we throw an exception of type const char*. Because we're within a try block and a matching exception handler is found, control immediately transfers to the const char* exception handler. The result is:

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

By now, you should be getting the basic idea behind exceptions. In the next lesson, we'll do quite a few more examples to show how flexible exceptions are.

What catch blocks typically do

If an exception is routed to a catch block, it is considered "handled" even if the catch block is empty. However, typically you'll want your catch blocks to do something useful. There are four common things that catch blocks do when they catch an exception:

First, catch blocks may print an error (either to the console, or a log file) and then allow the function to proceed.

Second, catch blocks may return a value or error code back to the caller.

Third, a catch block may throw another exception. Because the catch block is outside of the try block, the newly thrown exception in this case is not handled by the preceding try block -- it's handled by the next enclosing try block.

Fourth, a catch block in main() may be used to catch fatal errors and terminate the program in a clean way.