27.7 — Function try blocks

ALEX SEPTEMBER 11, 2023

Try and catch blocks work well enough in most cases, but there is one particular case in which they are not sufficient. Consider the following example:

```
#include <iostream>

class A
{
private:
    int m_x;
public:
    A(int x) : m_x{x}
    {
        if (x <= 0)
            throw 1; // Exception thrown here
    }
};

class B : public A
{
public:
    B(int x) : A{x}
    {
        // What happens if creation of A fails and we want to handle it here?
    }
};

int main()
{
    try
    {
        B b{0};
    }
    catch (int)
    {
        std::cout << "Oops\n";
    }
}
```

In the above example, derived class B calls base class constructor A, which can throw an exception. Because the creation of object b has been placed inside a try block (in function main()), if A throws an exception, main's try block will catch it. Consequently, this program prints:

```
Oops
```

But what if we want to catch the exception inside of B? The call to base constructor A happens via the member initialization list, before the B constructor's body is called. There's no way to wrap a standard try block around it.

In this situation, we have to use a slightly modified try block called a **function try block**.

Function try blocks

Function try blocks are designed to allow you to establish an exception handler around the body of an entire function, rather than around a block of code.

The syntax for function try blocks is a little hard to describe, so we'll show by example:

```
#include <iostream>

class A
{
private:
    int m_x;
public:
    A(int x) : m_x{x}
    {
        if (x <= 0)
            throw 1; // Exception thrown here
    }
};

class B : public A
{
public:
    B(int x) try : A{x} // note addition of try keyword here
    {
    }
    catch (...) // note this is at same level of indentation as the function itself
    {
                // Exceptions from member initializer list or constructor body are caught here

                std::cerr << "Exception caught\n";

                throw; // rethrow the existing exception
    }
};

int main()
{
    try
    {
        B b{0};
    }
    catch (int)
    {
        std::cout << "Oops\n";
    }
}
```

When this program is run, it produces the output:

```
Exception caught
Oops
```

Let's examine this program in more detail.

First, note the addition of the "try" keyword before the member initializer list. This indicates that everything after that point (until the end of the function) should be considered inside of the try block.

Second, note that the associated catch block is at the same level of indentation as the entire function. Any exception thrown between the try keyword and the end of the function body will be eligible to be caught here.

When the above program runs, variable `b` begins construction, which calls B's constructor (which utilizes a function try). B's constructor calls A's constructor, which then raises an exception. Because A's constructor does not handle this exception, the exception propagates up the stack to B's constructor, where it is caught by the function-level catch of B's constructor. The catch block prints "Exception caught", and then rethrows the current exception up the stack, which is caught by the catch block in `main()`, which prints "Oops".

Limitations on function catch blocks

With a regular catch block (inside a function), we have three options: We can throw a new exception, rethrow the current exception, or resolve the exception (by either a return statement, or by letting control reach the end of the catch block).

A function-level catch block for a constructor must either throw a new exception or rethrow the existing exception -- they are not allowed to resolve exceptions! Return statements are also not allowed, and reaching the end of the catch block will implicitly rethrow.

A function-level catch block for a destructor can throw, rethrow, or resolve the current exception via a return statement. Reaching the end of the catch block will implicitly rethrow.

A function-level catch block for other functions can throw, rethrow, or resolve the current exception via a return statement. Reaching the end of the catch block will implicitly resolve the exception for non-value (void) returning functions and produce undefined behavior for value-returning functions!

The following table summarizes the limitations and behavior of function-level catch blocks:

Function type	Can resolve exceptions via return statement	Behavior at end of catch block
Constructor	No, must throw or rethrow	Implicit rethrow
Destructor	Yes	Implicit rethrow
Non-value returning function	Yes	Resolve exception
Value-returning function	Yes	Undefined behavior

Because such behavior at the end of the catch block varies dramatically depending on the type of function (and includes undefined behavior in the case of value-returning functions), we recommend never letting control reach the end of the catch block, and always explicitly throwing, rethrowing, or returning.

Best practice

Avoid letting control reach the end of a function-level catch block. Instead, explicitly throw, rethrow, or return.

In the program above, if we had not explicitly rethrow the exception in the function-level catch block of the constructor, control would have reached the end of the function-level catch, and because this was a constructor, an implicit rethrow would have happened instead. The result would have been the same.

Although function level try blocks can be used with non-member functions as well, they typically aren't because there's rarely a case where this would be needed. They are almost exclusively used with constructors!

Function try blocks can catch both base and the current class exceptions

In the above example, if either A or B's constructor throws an exception, it will be caught by the try block around B's constructor.

We can see that in the following example, where we're throwing an exception from class B instead of class A:

```
#include <iostream>

class A
{
private:
    int m_x;
public:
    A(int x) : m_x{x}
    {
    }
};

class B : public A
{
public:
    B(int x) try : A{x} // note addition of try keyword here
    {
        if (x <= 0) // moved this from A to B
            throw 1; // and this too
    }
    catch (...)
    {
                std::cerr << "Exception caught\n";

                // If an exception isn't explicitly thrown here, the current exception will be implicitly rethrown
    }
};

int main()
{
    try
    {
        B b{0};
    }
    catch (int)
    {
        std::cout << "Oops\n";
    }
}
```

We get the same output:

```
Exception caught
Oops
```

Don't use function try to clean up resources

When construction of an object fails, the destructor of the class is not called. Consequently, you may be tempted to use a function try block as a way to clean up a class that had partially allocated resources before failing. However, referring to members of the failed object is considered undefined behavior since the object is "dead" before the catch block executes. This means that you can't use function try to clean up after a class. If you want to clean up after a class, follow the standard rules for cleaning up classes that throw exceptions (see the "When constructors fail" subsection of lesson 27.5 -- Exceptions, classes, and inheritance (https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/)).

Function try is useful primarily for either logging failures before passing the exception up the stack, or for changing the type of exception thrown.

Next lesson
27.8 Exception dangers and downsides

Back to table of contents

Previous lesson
27.6 Rethrowing exceptions