



27.7 — Function try blocks

👤 **ALEX¹** 🕒 **AUGUST 15, 2024**

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:

```
1  #include <iostream>
2
3  class A
4  {
5  private:
6      int m_x;
7  public:
8      A(int x) : m_x{x}
9      {
10         if (x <= 0)
11             throw 1; // Exception thrown here
12     }
13 };
14
15 class B : public A
16 {
17 public:
18     B(int x) : A{x}
19     {
20         // What happens if creation of A fails and we want to handle it
21         here?
22     }
23 };
24
25 int main()
26 {
27     try
28     {
29         B b{0};
30     }
31     catch (int)
32     {
33         std::cout << "Oops\n";
34     }
35 }
```

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:

```

1  #include <iostream>
2
3  class A
4  {
5  private:
6      int m_x;
7  public:
8      A(int x) : m_x{x}
9      {
10         if (x <= 0)
11             throw 1; // Exception thrown here
12     }
13 };
14
15 class B : public A
16 {
17 public:
18     B(int x) try : A{x} // note addition of try keyword here
19     {
20     }
21     catch (...) // note this is at same level of indentation as the function
22     itself
23     {
24         // Exceptions from member initializer list or
25         // from constructor body are caught here
26
27         std::cerr << "Exception caught\n";
28
29         throw; // rethrow the existing exception
30     }
31 };
32
33 int main()
34 {
35     try
36     {
37         B b{0};
38     }
39     catch (int)
40     {
41         std::cout << "Oops\n";
42     }
43 }
```

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".

Best practice

Use function try blocks when you need a constructor to handle an exception thrown in the member initializer list.

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:

```

1  #include <iostream>
2
3  class A
4  {
5  private:
6      int m_x;
7  public:
8      A(int x) : m_x{x}
9      {
10     }
11 };
12
13 class B : public A
14 {
15 public:
16     B(int x) try : A{x} // note addition of try keyword here
17     {
18         if (x <= 0) // moved this from A to B
19             throw 1; // and this too
20     }
21     catch (...)
22     {
23         std::cerr << "Exception caught\n";
24
25         // If an exception isn't explicitly thrown here,
26         // the current exception will be implicitly rethrown
27     }
28 };
29
30 int main()
31 {
32     try
33     {
34         B b{0};
35     }
36     catch (int)
37     {
38         std::cout << "Oops\n";
39     }
40 }

```

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/) (<https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/>)²).

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](#)

3



[Back to table of contents](#)

4



[Previous lesson](#)

27.6 [Rethrowing exceptions](#)


5

6



[B](#) [U](#) [URL](#) [INLINE CODE](#) [C++ CODE BLOCK](#) [HELP!](#)

Leave a comment...



 Name*


@ Email* | 

Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!

 Avatars from <https://gravatar.com/>⁸ are connected to your provided email address.

36 COMMENTS

Newest ▼



clarification on the term 'fuction'

🕒 August 8, 2023 11:09 am

Let's examine this program in more detail.

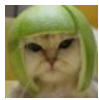
First, note the addition of the “try” keyword before the member initializer list. This indicates that everything after that point (until the end of the **function**) should be considered inside of the try block.

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.



Hello Alex, What do you mean by 'function' in those paragraphs? Do you mean constructor?

 Last edited 1 year ago by clarification on the term 'fucntion'

 0  Reply



Alex Author

 Reply to [clarification on the term 'fucntion'](#)⁹  August 12, 2023 10:58 am

Yes, constructors are functions (as are destructors).

 1  Reply



Bubun

 March 27, 2023 1:31 pm

"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."

Well, I tried...

```

1  class B : public A
2  {
3  public:
4      try
5      {
6          B(int x) : A{x} // note addition of try keyword here
7          {
8          }
9      }
10     catch (...) // note this is at same level of indentation as the function
11     itself
12     {
13         // Exceptions from member initializer list or constructor
14         body are caught here
15
16         std::cerr << "Exception caught\n";
17
18         throw; // rethrow the existing exception
19     }
20 };

```

Not only it is ugly but also indeed fails to compile. But I still don't understand why it is not possible

👍 0 ➡ Reply



Alex Author

🗨 Reply to [Bubun](#)¹⁰ ⌚ March 29, 2023 10:52 pm

It's not possible because that's not the syntax they decided on for doing function-level try. Probably because it makes the function hard to find.

👍 2 ➡ Reply



yellowEmu

⌚ January 26, 2023 5:54 am

function try blocks don't work with lambdas :(

👍 0 ➡ Reply



Athena

⌚ December 26, 2022 10:42 pm

Small typo here

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

Should presumably be reach.

Side note, this guide has helped me progress from knowing nearly nothing about coding to here. Cheers and thank you for all your work.

👍 2 ➡ Reply



Alex Author

🗨 Reply to [Athena](#)¹¹ ⌚ January 1, 2023 5:43 pm

The missing 'r' has been located and returned to its intended position. Thanks!

👍 0 ➡ Reply



Waldo

⌚ September 26, 2022 4:26 pm

> with function-level try blocks, you must throw or rethrow an exception

Only in constructors and destructors

👍 0

➡ Reply

**Alex** Author👤 Reply to [Waldo](#)¹² ⌚ September 30, 2022 2:55 pm

And in value returning functions (otherwise UB will result). I've updated the lesson a bit to explicitly note that the catch blocks of non-value returning functions can resolve exceptions, but everyone else must throw or rethrow (either explicitly or implicitly), and added a table with behavioral summary info. Thanks for pointing out the issue!

👍 1

➡ Reply

**Waldo**👤 Reply to [Alex](#)¹³ ⌚ September 30, 2022 3:41 pm

> And in value returning functions (otherwise UB will result)

Values can be returned from function-try-block catch clauses:

```

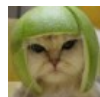
1 | #include <iostream>
2 | #include <stdexcept>
3 |
4 | int foo() try {
5 |     throw std::exception{};
6 | }
7 | catch (const std::exception&) {
8 |     return 69;
9 | }
10 |
11 | int main() { std::cout << foo() << '\n'; }
```

Destructors can return without rethrowing too (rethrowing is implicit when they don't). See <https://en.cppreference.com/w/cpp/language/function-try-block>

📄 Last edited 2 years ago by Waldo

👍 1

➡ Reply

**Alex** Author👤 Reply to [Waldo](#)¹⁴ ⌚ October 3, 2022 7:50 pm

That's what I get for trying to rewrite a significant portion of a lesson while sick.
:/ Thanks for the corrections!

👍 1

➡ Reply

**Waldo**👤 Reply to [Alex](#)¹⁵ ⌚ October 3, 2022 10:20 pm

Ah, that sucks. Hope you get well soon :)

The new section is great. I really like the idea of using a table there

👍 0

➡ Reply



Waldo

🕒 July 21, 2022 2:10 pm

> see the **"When constructor fail"** subsection of lesson

*"When constructors fail" (I checked)

👍 0

➡ Reply



JustABu

JustABug

🕒 June 25, 2022 1:16 am

... if either A or B's constructor throw an exception ... (throws, I assume)

👍 0

➡ Reply



JustABu

JustABug

🕒 June 25, 2022 1:06 am

`#include <iostream>` is missing in the first example.

👍 0

➡ Reply



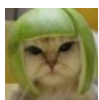
Armitage

🕒 April 11, 2022 7:14 am

Why you noted that level of indentation is the same as function itself it is about style or what?
I mean it's technically unnecessary.

👍 0

➡ Reply



Alex Author

➡ Reply to [Armitage](#)¹⁶ 🕒 April 11, 2022 12:00 pm

It's a matter of style, and doing it that way is idiomatic.

👍 3

➡ Reply



Berrie

🕒 March 17, 2021 3:01 pm

Feedback:

This chapter uses a lot of direct initializations instead of list initializations.

I'm also curious as to why you would want to change the type of an exception that is thrown. Does this have to do with precompiled code from the standard library that can not be changed or are there other reasons to rethrow exceptions in a different type.

It seems to be of much more use to throw the exception in the right type to start with.

👍 0

➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/>
3. <https://www.learncpp.com/cpp-tutorial/exception-dangers-and-downsides/>
4. <https://www.learncpp.com/>
5. <https://www.learncpp.com/cpp-tutorial/rethrowing-exceptions/>
6. <https://www.learncpp.com/function-try-blocks/>
7. <https://www.learncpp.com/cpp-tutorial/chapter-27-summary-and-quiz/>
8. <https://gravatar.com/>
9. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-585294>
10. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-578729>
11. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-575579>
12. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-573568>
13. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-573675>
14. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-573680>
15. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-573748>
16. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/#comment-567448>
17. <https://g.ezoic.net/privacy/learncpp.com>