What happens if my C++ exception handler itself raises an exception?

devblogs.microsoft.com/oldnewthing/20221021-00

October 21, 2022

Raymond Chen

Last time, we looked at what happens if your structured exception handler raises its own structured exception. Today, we'll look at the C++ version of the same question: What happens if my C++ exception handler itself raises an exception?

Let's look at this sample code fragment:

```
try
{
    Block1;
}
catch (Type1 ex1)
{
    Block2;
}
catch (Type2 const& ex2)
{
    Block3;
}
/* finally { destructors;
} */
```

If an exception is thrown out of `Block1`, C++ looks for a matching `catch` block,[1] and neither clause matches, then the search for a handler continues at the next outer scope. If no scope handles the exception, then the process terminates via `std::terminate`.

The bodies of the `catch` blocks are not in scope of the `try` statement, so if an exception is thrown by `Block2` or `Block3`, the search for a handler does not include the `catch (Type1 ex1)` or `catch (Type2 const& ex1)` clauses.

There is no `finally` clause in C++ `try` / `catch` statements, but the equivalent functionality is obtained by putting the desired cleanup code in destructors of objects declared in `Block1`. If control exits `Block1` due to an exception, and one of the objects in

that block throws an exception in its destructor, then the rules of C++ are that the process terminates immediately via `std::terminate`. This is different from Windows structured exceptions and C# exceptions.

In the code sample above, I've put these destructors in a pseudo-"finally" clause, just so I will have a place to annotate them.

Another difference from Windows structured exceptions is the case of an exception that occurs while performing type matching: If a `catch` clause captures the exception object by value, the exception object is constructed from the thrown object. And if that constructor throws an exception, the C++ standard says that the process terminates via `std::terminate`.[2] (This is also different from Windows structured exceptions.)

Here's an annotated version of the above discussion:

	Under consideration

`try`
`{`
`    Block1;`
`}`	

	`std::terminate`
`catch (Type1 ex1)`	
	Not considered

`{`
`    Block2;`
`}`	

	`std::terminate`
`catch (Type2 const& ex2)`	
	Not considered

`{`
`    Block3;`
`}`	

	It's complicated
`/* finally { destructors;` `} */`	

The "It's complicated" for the pseudo-finally clause applies because both behaviors are possible, depending on why the destructors are running.

- If `Block1` is exiting normally, then exceptions that occur in destructors are catchable by the `try` statement.
- If `Block1` is exiting due to an exception, then `std::terminate` is called.

Note that both cases can occur in the same `try` statement! Suppose the `Block1` runs to the final close-brace, and then it becomes time to run the destructors. Suppose there are two objects inside the `Block1` that require destruction. The first one to destruct throws an exception. This is an exception thrown during normal exit of `Block1`, so the exception is catchable. But before we try to catch that exception, we need to run the second destructor. If this second destructor also throws an exception, we are now in the case of a destructor throwing an exception during exception handling, and this results in `std::terminate`.

[1] This search is done sequentially, so it will try to match `Type1` first, and `Type2 const&` second. As a result, reordering your `catch` clauses can result in changes in behavior if the thrown object matches multiple `catch` clauses.

[2] In general, you should catch things by reference. This removes the possibility of exceptions during the construction of the `catch` argument, and it also avoids slicing if the thrown object is a derived class of the thing you're catching.

Raymond Chen

Follow