throw expression

Signals an erroneous condition and executes an error handler.

Syntax

throw *expression*	(1)
throw	(2)

Explanation

See try-catch block for more information about *try* and *catch* (exception handler) blocks

1) First, copy-initializes the *exception object* from *expression*

- This may call the move constructor for rvalue expression
- This may also call the move constructor for lvalue expressions if they name local variables or function or catch-clause parameters whose scope does not extend past the innermost enclosing try-block (if any), by same two-step overload resolution as in return statement (since C++17)
- The copy/move may be subject to copy elision
- Even if copy initialization selects the move constructor, copy initialization from lvalue must be well-formed, and the destructor must be accessible (since C++14)

then transfers control to the exception handler with the matching type whose compound statement or member initializer list was most recently entered and not exited by this thread of execution.

2) Rethrows the currently handled exception. Abandons the execution of the current catch block and passes control to the next matching exception handler (but not to another catch clause after the same try block: its compound-statement is considered to have been 'exited'), reusing the existing exception object: no new objects are made. This form is only allowed when an exception is presently being handled (it calls `std::terminate` if used otherwise). The catch clause associated with a function-try-block must exit via rethrowing if used on a constructor.

See `std::terminate` and `std::unexpected` for the handling of errors that arise during exception handling.

The exception object

The exception object is a temporary object in unspecified storage that is constructed by the `throw` expression.

The type of the *exception object* is the static type of *expression* with top-level cv-qualifiers removed. Array and function types are adjusted to pointer and pointer to function types, respectively. If the type of the exception object would be an incomplete type, an abstract class type, or pointer to incomplete type other than pointer to (cv-qualified) void, the throw-expression is a compile-time error. If the type of *expression* is a class type, its copy/move constructor and destructor must be accessible even if copy elision takes place.

Unlike other temporary objects, the exception object is considered to be an lvalue argument when initializing the catch clause parameters, so it can be caught by lvalue reference, modified, and rethrown.

The exception object persists until the last catch clause exits other than by rethrowing (if not by rethrowing, it is destroyed immediately after the destruction of the catch clause's parameter), or until the last `std::exception_ptr` that references this object is destroyed (in which case the exception object is destroyed just before the destructor of `std::exception_ptr` returns.

Stack unwinding

Once the exception object is constructed, the control flow works backwards (up the call stack) until it reaches the start of a try block, at which point the parameters of all associated `catch` blocks are compared, in order of appearance, with the type of the exception object to find a match (see try-catch for details on this process). If no match is found, the control flow continues to unwind the stack until the next `try` block, and so on. If a match is found, the control flow jumps to the matching `catch` block.

As the control flow moves up the call stack, destructors are invoked for all objects with automatic storage duration constructed, but not yet destroyed, since the corresponding try-block was entered, in reverse order of completion of their constructors. If an exception is thrown from a destructor of a local variable or of a temporary used in a return statement, the destructor for the object returned from the function is also invoked. (since C++14)

If an exception is thrown from a constructor or (rare) from a destructor of an object (regardless of the object's storage duration), destructors are called for all fully-constructed non-static non-variant (until C++14) members and base classes, in reverse order of completion of their constructors. Variant members of union-like classes are only destroyed in the case of unwinding from constructor, and if the active member changed between initialization and destruction, the behavior is undefined. (since C++14)

If a delegating constructor exits with an exception after the non-delegating constructor successfully completed, the destructor for this object is called. (since C++11)

If the exception is thrown from a constructor that is invoked by a new-expression, the matching deallocation function is called, if available.

This process is called *stack unwinding*.

If any function that is called directly by the stack unwinding mechanism, after initialization of the exception object and before the start of the exception handler, exits with an exception, `std::terminate` is called. Such functions include destructors of objects with automatic storage duration whose scopes are exited, and the copy constructor of the exception object that is called (if not elided) to initialize catch-by-value arguments.

If an exception is thrown and not caught, including exceptions that escape the initial function of `std::thread`, the main function, and the constructor or destructor of any static or thread-local objects, then `std::terminate` is called. It is implementation-defined whether any stack unwinding takes place for uncaught exceptions.

Notes

When rethrowing exceptions, the second form must be used to avoid object slicing in the (typical) case where exception objects use inheritance:

```
try {
    std::string("abc").substr(10); // throws std::length_error
} catch(const std::exception& e) {
    std::cout << e.what() << '\n';
//  throw e; // copy-initializes a new exception object of type std::exception
    throw;   // rethrows the exception object of type std::length_error
}
```

The throw-expression is classified as prvalue expression of type `void`. Like any other expression, it may be a sub-expression in another expression, most commonly in the conditional operator:

```
double f(double d)
{
    return d > 1e7 ? throw std::overflow_error("too big") : d;
```

```
}
int main()
{
    try {
        std::cout << f(1e10) << '\n';
    } catch (const std::overflow_error& e) {
        std::cout << e.what() << '\n';
    }
}
```

Keywords

throw

Example

Run this code

```
#include <iostream>
#include <stdexcept>

struct A {
    int n;
    A(int n = 0): n(n) { std::cout << "A(" << n << ") constructed successfully\n"; }
    ~A() { std::cout << "A(" << n << ") destroyed\n"; }
};

int foo()
{
    throw std::runtime_error("error");
}

struct B {
    A a1, a2, a3;
    B() try : a1(1), a2(foo()), a3(3) {
        std::cout << "B constructed successfully\n";
    } catch(...) {
        std::cout << "B::B() exiting with exception\n";
    }
    ~B() { std::cout << "B destroyed\n"; }
};

struct C : A, B {
    C() try {
        std::cout << "C::C() completed successfully\n";
    } catch(...) {
        std::cout << "C::C() exiting with exception\n";
    }
    ~C() { std::cout << "C destroyed\n"; }
};
```

```
int main () try
{
    // creates the A base subobject
    // creates the a1 member of B
    // fails to create the a2 member of B
    // unwinding destroys the a1 member of B
    // unwinding destroys the A base subobject
    C c;
} catch (const std::exception& e) {
    std::cout << "main() failed to create C with: " << e.what();
}
```

Output:

```
A(0) constructed successfully
A(1) constructed successfully
A(1) destroyed
B::B() exiting with exception
A(0) destroyed
C::C() exiting with exception
main() failed to create C with: error
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 1866 (https://wg21.cmeerw.net/cwg/issue1866)	C++14	variant members were leaked on stack unwinding from constructor	variant members destroyed
CWG 1863 (https://wg21.cmeerw.net/cwg/issue1863)	C++14	copy constructor was not required for move-only exception objects when thrown, but copying allowed later	copy constructor required
CWG 2176 (https://wg21.cmeerw.net/cwg/issue2176)	C++14	throw from a local variable dtor could skip return value destructor	function return value added to unwinding

See also

- copy elision
- try-catch block
- noexcept specifier
- exception specifications

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/language/throw&oldid=115201"