# 27.5 — Exceptions, classes, and inheritance

ALEX SEPTEMBER 11, 2023

### Exceptions and member functions

Up to this point in the tutorial, you've only seen exceptions used in non-member functions. However, exceptions are equally useful in member functions, and even more so in overloaded operators. Consider the following overloaded [] operator as part of a simple integer array class:

```
int& IntArray::operator[](const int index)
{
    return m_data[index];
}
```

Although this function will work great as long as index is a valid array index, this function is sorely lacking in some good error checking. We could add an assert statement to ensure the index is valid:

```
int& IntArray::operator[](const int index)
{
    assert (index >= 0 && index < getLength());
    return m_data[index];
}
```

Now if the user passes in an invalid index, the program will cause an assertion error. Unfortunately, because overloaded operators have specific requirements as to the number and type of parameter(s) they can take and return, there is no flexibility for passing back error codes or Boolean values to the caller to handle. However, since exceptions do not change the signature of a function, they can be put to great use here. Here's an example:

```
int& IntArray::operator[](const int index)
{
    if (index < 0 || index >= getLength())
        throw index;

    return m_data[index];
}
```

Now, if the user passes in an invalid index, operator[] will throw an int exception.

---

### When constructors fail

Constructors are another area of classes in which exceptions can be very useful. If a constructor must fail for some reason (e.g. the user passed in invalid input), simply throw an exception to indicate the object failed to create. In such a case, the object's construction is aborted, and all class members (which have already been created and initialized prior to the body of the constructor executing) are destructed as per usual.

However, the class's destructor is never called (because the object never finished construction). Because the destructor never executes, you can't rely on said destructor to clean up any resources that have already been allocated.

This leads to the question of what we should do if we've allocated resources in our constructor and then an exception occurs prior to the constructor finishing. How do we ensure the resources that we've already allocated get cleaned up properly? One way would be to wrap any code that can fail in a try block, use a corresponding catch block to catch the exception and do any necessary cleanup, and then rethrow the exception (a topic we'll discuss in lesson 27.6 -- Rethrowing exceptions (https://www.learncpp.com/cpp-tutorial/rethrowing-exceptions/)). However, this adds a lot of clutter, and it's easy to get wrong, particularly if your class allocates multiple resources.

Fortunately, there is a better way. Taking advantage of the fact that class members are destructed even if the constructor fails, if you do the resource allocations inside the members of the class (rather than in the constructor itself), then those members can clean up after themselves when they are destructed.

Here's an example:

```
#include <iostream>

class Member
{
public:
    Member()
    {
        std::cerr << "Member allocated some resources\n";
    }

    ~Member()
    {
        std::cerr << "Member cleaned up\n";
    }
};

class A
{
private:
    int m_x {};
    Member m_member;

public:
    A(int x) : m_x{x}
    {
        if (x <= 0)
            throw 1;
    }

    ~A()
    {
        std::cerr << "~A\n"; // should not be called
    }
};


int main()
{
    try
    {
        A a{0};
    }
    catch (int)
    {
        std::cerr << "Oops\n";
    }

    return 0;
}
```

This prints:

```
Member allocated some resources
Member cleaned up
Oops
```

In the above program, when class A throws an exception, all of the members of A are destructed. m_member's destructor is called, providing an opportunity to clean up any resources that it allocated.

This is part of the reason that RAII (covered in lesson 19.3 -- Destructors (https://www.learncpp.com/cpp-tutorial/destructors/)) is advocated so highly -- even in exceptional circumstances, classes that implement RAII are able to clean up after themselves.

However, creating a custom class like Member to manage a resource allocation isn't efficient. Fortunately, the C++ standard library comes with RAII-compliant classes to manage common resource types, such as files (std::fstream, covered in lesson 28.6 -- Basic file I/O (https://www.learncpp.com/cpp-tutorial/basic-file-io/)) and dynamic memory (std::unique_ptr and the other smart pointers, covered in 22.1 -- Introduction to smart pointers and move semantics (https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/)).

For example, instead of this:

```
class Foo
private:
    int* ptr; // Foo will handle allocation/deallocation
```

Do this:

```
class Foo
private:
    std::unique_ptr<int> ptr; // std::unique_ptr will handle allocation/deallocation
```

In the former case, if Foo's constructor were to fail after ptr had allocated its dynamic memory, Foo would be responsible for cleanup, which can be challenging. In the latter case, if Foo's constructor were to fail after ptr has allocated its dynamic memory, ptr's destructor would execute and return that memory to the system. Foo doesn't have to do any explicit cleanup when resource handling is delegated to RAII-compliant members!

---

## Exception classes

One of the major problems with using basic data types (such as int) as exception types is that they are inherently vague. An even bigger problem is disambiguation of what an exception means when there are multiple statements or function calls within a try block.

```
// Using the IntArray overloaded operator[] above

try
{
    int* value{ new int{ array[index1] + array[index2]} };
}
catch (int value)
{
    // What are we catching here?
}
```

In this example, if we were to catch an int exception, what does that really tell us? Was one of the array indexes out of bounds? Did operator+ cause integer overflow? Did operator new fail because it ran out of memory? Unfortunately, in this case, there's just no easy way to disambiguate. While we can throw const char* exceptions to solve the problem of identifying WHAT went wrong, this still does not provide us the ability to handle exceptions from various sources differently.

One way to solve this problem is to use exception classes. An **exception class** is just a normal class that is designed specifically to be thrown as an exception. Let's design a simple exception class to be used with our IntArray class:

```
#include <string>
#include <string_view>

class ArrayException
{
private:
    std::string m_error;

public:
    ArrayException(std::string_view error)
        : m_error{ error }
    {
    }

    const std::string& getError() const { return m_error; }
};
```

Here's a full program using this class:

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

class ArrayException
{
private:
    std::string m_error;

public:
    ArrayException(std::string_view error)
        : m_error{ error }
    {
    }

    const std::string& getError() const { return m_error; }
};

class IntArray
{
private:
    int m_data[3]{}; // assume array is length 3 for simplicity

public:
    IntArray() {}

    int getLength() const { return 3; }

    int& operator[](const int index)
    {
        if (index < 0 || index >= getLength())
            throw ArrayException{ "Invalid index" };

        return m_data[index];
    }

};

int main()
{
    IntArray array;

    try
    {
        int value{ array[5] }; // out of range subscript
    }
    catch (const ArrayException& exception)
    {
        std::cerr << "An array exception occurred (" << exception.getError() << ")\n";
    }
}
```

Using such a class, we can have the exception return a description of the problem that occurred, which provides context for what went wrong. And since ArrayException is its own unique type, we can specifically catch exceptions thrown by the array class and treat them differently from other exceptions if we wish.

Note that exception handlers should catch class exception objects by reference instead of by value. This prevents the compiler from making a copy of the exception at the point where it is caught, which can be expensive when the exception is a class object, and prevents object slicing when dealing with derived exception classes (which we'll talk about in a moment). Catching exceptions by pointer should generally be avoided unless you have a specific reason to do so.

## Exceptions and inheritance

Since it's possible to throw classes as exceptions, and classes can be derived from other classes, we need to consider what happens when we use inherited classes as exceptions. As it turns out, exception handlers will not only match classes of a specific type, they'll also match classes derived from that specific type as well! Consider the following example:

```
#include <iostream>

class Base
{
public:
    Base() {}
};

class Derived: public Base
{
public:
    Derived() {}
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (const Base& base)
    {
        std::cerr << "caught Base";
    }
    catch (const Derived& derived)
    {
        std::cerr << "caught Derived";
    }

    return 0;
}
```

In the above example we throw an exception of type Derived. However, the output of this program is:

```
caught Base
```

What happened?

First, as mentioned above, derived classes will be caught by handlers for the base type. Because Derived is derived from Base, Derived is-a Base (they have an is-a relationship). Second, when C++ is attempting to find a handler for a raised exception, it does so sequentially. Consequently, the first thing C++ does is check whether the exception handler for Base matches the Derived exception. Because Derived is-a Base, the answer is yes, and it executes the catch block for type Base! The catch block for Derived is never even tested in this case.

In order to make this example work as expected, we need to flip the order of the catch blocks:

```
#include <iostream>

class Base
{
public:
    Base() {}
};

class Derived: public Base
{
public:
    Derived() {}
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (const Derived& derived)
    {
        std::cerr << "caught Derived";
    }
    catch (const Base& base)
    {
        std::cerr << "caught Base";
    }

    return 0;
}
```

This way, the Derived handler will get first shot at catching objects of type Derived (before the handler for Base can). Objects of type Base will not match the Derived handler (Derived is-a Base, but Base is not a Derived), and thus will "fall through" to the Base handler.

**Rule**

Handlers for derived exception classes should be listed before those for base classes.

The ability to use a handler to catch exceptions of derived types using a handler for the base class turns out to be exceedingly useful.

## std::exception

Many of the classes and operators in the standard library throw exception classes on failure. For example, operator new can throw std::bad_alloc if it is unable to allocate enough memory. A failed dynamic_cast will throw std::bad_cast. And so on. As of C++20, there are 28 different exception classes that can be thrown, with more being added in each subsequent language standard.

The good news is that all of these exception classes are derived from a single class called **std::exception** (defined in the <exception> header). std::exception is a small interface class designed to serve as a base class to any exception thrown by the C++ standard library.

Much of the time, when an exception is thrown by the standard library, we won't care whether it's a bad allocation, a bad cast, or something else. We just care that something catastrophic went wrong and now our program is exploding. Thanks to std::exception, we can set up an exception handler to catch exceptions of type std::exception, and we'll end up catching std::exception and all of the derived exceptions together in one place. Easy!

```
#include <cstddef> // for std::size_t
#include <exception> // for std::exception
#include <iostream>
#include <limits>
#include <string> // for this example

int main()
{
    try
    {
        // Your code using standard library goes here
        // We'll trigger one of these exceptions intentionally for the sake of the example
        std::string s;
        s.resize(std::numeric_limits<std::size_t>::max()); // will trigger a std::length_error or allocation exception
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception& exception)
    {
        std::cerr << "Standard exception: " << exception.what() << '\n';
    }

    return 0;
}
```

On the author's machine, the above program prints:

```
Standard exception: string too long
```

The above example should be pretty straightforward. The one thing worth noting is that std::exception has a virtual member function named **what()** that returns a C-style string description of the exception. Most derived classes override the what() function to change the message. Note that this string is meant to be used for descriptive text only -- do not use it for comparisons, as it is not guaranteed to be the same across compilers.



Sometimes we'll want to handle a specific type of exception differently. In this case, we can add a handler for that specific type, and let all the others "fall through" to the base handler. Consider:

```
try
{
     // code using standard library goes here
}
// This handler will catch std::length_error (and any exceptions derived from it) here
catch (const std::length_error& exception)
{
    std::cerr << "You ran out of memory!" << '\n';
}
// This handler will catch std::exception (and any exception derived from it) that fall
// through here
catch (const std::exception& exception)
{
    std::cerr << "Standard exception: " << exception.what() << '\n';
}
```

In this example, exceptions of type std::length_error will be caught by the first handler and handled there. Exceptions of type std::exception and all of the other derived classes will be caught by the second handler.

Such inheritance hierarchies allow us to use specific handlers to target specific derived exception classes, or to use base class handlers to catch the whole hierarchy of exceptions. This allows us a fine degree of control over what kind of exceptions we want to handle while ensuring we don't have to do too much work to catch "everything else" in a hierarchy.

---

## Using the standard exceptions directly

Nothing throws a std::exception directly, and neither should you. However, you should feel free to throw the other standard exception classes in the standard library if they adequately represent your needs. You can find a list of all the standard exceptions on cppreference (http://en.cppreference.com/w/cpp/error/exception).



std::runtime_error (included as part of the stdexcept header) is a popular choice, because it has a generic name, and its constructor takes a customizable message:

```
#include <exception> // for std::exception
#include <iostream>
#include <stdexcept> // for std::runtime_error

int main()
{
    try
    {
        throw std::runtime_error("Bad things happened");
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception& exception)
    {
        std::cerr << "Standard exception: " << exception.what() << '\n';
    }

    return 0;
}
```

This prints:

```
Standard exception: Bad things happened
```

---

## Deriving your own classes from std::exception or std::runtime_error

You can, of course, derive your own classes from std::exception, and override the virtual what() const member function. Here's the same program as above, with ArrayException derived from std::exception:

```
#include <exception> // for std::exception
#include <iostream>
#include <string>
#include <string_view>

class ArrayException : public std::exception
{
private:
    std::string m_error{}; // handle our own string

public:
    ArrayException(std::string_view error)
        : m_error{error}
    {
    }

    // std::exception::what() returns a const char*, so we must as well
    const char* what() const noexcept override { return m_error.c_str(); }
};

class IntArray
{
private:
    int m_data[3] {}; // assume array is length 3 for simplicity

public:
    IntArray() {}

    int getLength() const { return 3; }

    int& operator[](const int index)
    {
        if (index < 0 || index >= getLength())
            throw ArrayException("Invalid index");

        return m_data[index];
    }

};

int main()
{
    IntArray array;

    try
    {
        int value{ array[5] };
    }
    catch (const ArrayException& exception) // derived catch blocks go first
    {
        std::cerr << "An array exception occurred (" << exception.what() << ")\n";
    }
    catch (const std::exception& exception)
    {
        std::cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
    }
}
```

Note that virtual function what() has specifier noexcept (which means the function promises not to throw exceptions itself). Therefore, our override should also have specifier noexcept.

Because std::runtime_error already has string handling capabilities, it's also a popular base class for derived exception classes. std::runtime_error can take a C-style string parameter, or a std::string parameter.

Here's the same example derived from std::runtime_error instead:

```
#include <exception> // for std::exception
#include <iostream>
#include <stdexcept> // for std::runtime_error
#include <string>

class ArrayException : public std::runtime_error
{
public:
    // std::runtime_error takes a const char* null-terminated string.
    // std::string_view may not be null-terminated, so it's not a good choice here.
    // Our ArrayException will take a const std::string& instead,
    // which is guaranteed to be null-terminated, and can be converted to a const char*.
    ArrayException(const std::string& error)
        : std::runtime_error{ error } // std::runtime_error will handle the string
    {
    }

        // no need to override what() since we can just use std::runtime_error::what()
};

class IntArray
{
private:
    int m_data[3]{}; // assume array is length 3 for simplicity

public:
    IntArray() {}

    int getLength() const { return 3; }

    int& operator[](const int index)
    {
        if (index < 0 || index >= getLength())
            throw ArrayException("Invalid index");

        return m_data[index];
    }

};

int main()
{
    IntArray array;

    try
    {
        int value{ array[5] };
    }
    catch (const ArrayException& exception) // derived catch blocks go first
    {
        std::cerr << "An array exception occurred (" << exception.what() << ")\n";
    }
    catch (const std::exception& exception)
    {
        std::cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
    }
}
```

It's up to you whether you want create your own standalone exception classes, use the standard exception classes, or derive your own exception classes from std::exception or std::runtime_error. All are valid approaches depending on your aims.

---

### Exception classes should be copyable

When an exception is thrown, the object being thrown is typically a temporary or local variable that has been allocated on the stack. However, the process of exception handling may unwind the function, causing all variables local to the function to be destroyed. So how does the exception object being thrown survive stack unwinding?

When an exception is thrown, the compiler makes a copy of the exception object to some piece of unspecified memory (outside of the call stack) reserved for handling exceptions. That way, the exception object is persisted regardless of whether or how many times the stack is unwound.

This means that the objects being thrown generally need to be copyable (even if the stack is not actually unwound). Smart compilers may be able to perform a move instead, or elide the copy altogether in specific circumstances.

Here's an example showing what happens when we try to throw a Derived object that is not copyable:

```
#include <iostream>

class Base
{
public:
    Base() {}
};

class Derived : public Base
{
public:
    Derived() {}

    Derived(const Derived&) = delete; // not copyable
};

int main()
{
    Derived d{};

    try
    {
        throw d; // compile error: Derived copy constructor was deleted
    }
    catch (const Derived& derived)
    {
        std::cerr << "caught Derived";
    }
    catch (const Base& base)
    {
        std::cerr << "caught Base";
    }

    return 0;
}
```

When this program is compiled, the compiler will complain that the Derived copy constructor is not available, and halt compilation.


**Next lesson**
27.6 Rethrowing exceptions

**Back to table of contents**

**Previous lesson**
27.4 Uncaught exceptions and catch-all handlers