14.5 — Exceptions, classes, and inheritance

BY ALEX ON OCTOBER 26TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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 moreso 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. While this is useful to indicate to the user that something went wrong, sometimes the better course of action is to fail silently and let the caller know something went wrong so they can deal with it as appropriate.

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. 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 not rely on said destructor to clean up any resources that have already been allocated. Any such cleanup can happen in the constructor prior to throwing the exception in the first place. However, even better, because the members of the class are destructed as per usual, if you do the resource allocations in the members themselves, 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. This gives m_member an opportunity to clean up any resources that were allocated.

This is part of the reason that RAII (reference: **8.7 -- Destructors**) is advocated so highly -- even in abnormal circumstances, classes that implement RAII properly should be able to clean up after themselves.

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>

class ArrayException
{
private:
    std::string m_error;

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

    const char* getError() { return m_error.c_str(); }
};
```

Here's a full program using this class:

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

class ArrayException
{
private:
    std::string m_error;

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

     const char* getError() { return m_error.c_str(); }
};

```

```
class IntArray
{
private:

    int m_data[3]; // assume array is length 3 for simplicity
public:
    IntArray() {}

    int getLength() { 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 (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, 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:

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

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

```

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

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

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

int main()
{
    try
    {
        throw Derived();
    }
    catch (Derived &derived)
    {
        cerr << "caught Derived";
    }
    catch (Base &base)
    {
        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++17, there are 25 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**. 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 (21+) of the derived exceptions together in one place. Easy!

```
#include <iostream>
#include <exception> // for std::exception
#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 example
                std::string s;
                s.resize(-1); // will trigger a std::length_error
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (std::exception &exception)
    {
        std::cerr << "Standard exception: " << exception.what() << '\n';
    }

    return 0;
}
```

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 (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 (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**.

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 <iostream>
#include <stdexcept>

int main()
{
    try
    {
        throw std::runtime_error("Bad things happened");
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (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

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

```
4
5   class ArrayException: public std::exception
6   {
7   private:
8       std::string m_error;
9
10  public:
11      ArrayException(std::string error)
12          : m_error(error)
13      {
14      }
15
16      // return the std::string as a const C-style string
17  //  const char* what() const { return m_error.c_str(); } // pre-C++11 version
18      const char* what() const noexcept { return m_error.c_str(); } // C++11 version
19  };
20
21  class IntArray
22  {
23  private:
24
25      int m_data[3]; // assume array is length 3 for simplicity
26  public:
27      IntArray() {}
28
29      int getLength() { return 3; }
30
31      int& operator[](const int index)
32      {
33          if (index < 0 || index >= getLength())
34              throw ArrayException("Invalid index");
35
36          return m_data[index];
37      }
38
39  };
40
41  int main()
42  {
43      IntArray array;
44
45      try
46      {
47          int value = array[5];
48      }
49      catch (ArrayException &exception) // derived catch blocks go first
50      {
51          std::cerr << "An array exception occurred (" << exception.what() << ")\n";
52      }
53      catch (std::exception &exception)
54      {
55          std::cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
56      }
57  }
```

In C++11, virtual function what() was updated to have specifier noexcept (which means the function promises not to throw exceptions itself). Therefore, in C++11 and beyond, our override should also have specifier noexcept.

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. All are valid approaches depending on your aims.