

## Day 35 (Wed 04/19)

- day 35 recap questions
- exercise 35

## Announcements/reminders

- \* - HW7 is due this evening by 11pm
- Final project
  - You should have access to your team repository (let me know about any issues)
  - Due by 11pm on Friday 4/29
  - You may submit by 11pm on Monday 5/2 with no penalty
  - Contributions survey is due by 11pm on Monday 5/2
- Friday (4/22) will be a final project work day
  - Great chance to get help!

1. What is the difference between an unscoped and a scoped enum?
2. Why do we use exceptions?
3. What keyword is used to generate an exception? What keyword indicates that the block of code may generate an exception? What keyword indicates what should be done in the case of an exception?
4. In the case of multiple matching catch blocks for a thrown exception, which one actually catches the exception?
5. How do you get the message associated with an exception?

1. What is the difference between an unscoped and a scoped enum?

An unscoped enum adds the enum members to the current namespace.

The members of a scoped enum are placed in the namespace of the enum type.

E.g.:

```
enum Color {  
    RED, GREEN, BLUE  
};
```

```
// ...elsewhere in the program...  
Color c = BLUE;
```

```
enum class Color {  
    RED, GREEN, BLUE  
};
```

```
// ...elsewhere in the program...  
Color c = Color::BLUE;
```

Scoped enumerations are generally preferred because they do not "pollute" the namespace they're in

## 2. Why do we use exceptions?

Exceptions help us separate

- where in the program error conditions might occur, from
- where in the program it makes sense to handle the error conditions

By using exceptions, we can write functions with the attitude that they will succeed.

If an error condition arises, we can throw an exception.

Exceptions allow us to only handle error conditions in the specific points in the program where we are prepared to deal with them, and not clutter the rest of the program with complicated and hard-to-test error handling paths.

Without exceptions:

```
// Read an integer, then read that many double values and add them  
// to the given vector.
```

```
bool read_input(std::istream &in, std::vector<double> &v) {  
    int n;  
    if (!(in >> n)) { return false; }  
    for (int i = 0; i < n; i++) {  
        double val;  
        if (!(in >> val)) { return false; }  
        v.push_back(val);  
    }  
    return true;  
}
```

The caller, the caller's caller, etc. now needs to be concerned whether this function returned true or false.

With exceptions:

```
// Read an integer, then read that many double values and add them  
// to the given vector.
```

```
void read_input(std::istream &in, std::vector<double> &v) {  
    int n;  
    if (!(in >> n)) throw std::runtime_error("failed to read num elements"); }  
    for (int i = 0; i < n; i++) {  
        double val;  
        if (!(in >> val)) { throw std::runtime_error("failed to read a value"); }  
        v.push_back(val);  
    }  
}
```

The caller can just assume that the function will either (1) succeed, or (2) throw an exception.

IT IS NO LONGER THE CALLER'S RESPONSIBILITY TO HANDLE THE POSSIBILITY OF FAILURE. (Unless the caller wants to handle a failure.)

Without exceptions:

```
std::vector<double> data_vec;  
if (!read_data(data_vec)) {  
    // what are we supposed to do if the data can't be read successfully?  
    // this might not be a good place to report an error to the user  
}
```

With exceptions:

```
std::vector<double> data_vec;  
→ read_data(data_vec);  
// If we get here, we know the data was read successfully!  
// If an exception was thrown, it is *our caller's* problem
```

3. What keyword is used to generate an exception? What keyword indicates that the block of code may generate an exception? What keyword indicates what should be done in the case of an exception?

```
// generate an exception  
throw exception_object; // replace exception_object with an exception object  
                        // to throw
```

```
// handle an exception  
try {  
    // ... this is code that might throw an exception ...  
} catch (exception_type &ex) {  
    // ... handle the possibility that an exception_type exception was thrown ...  
}
```

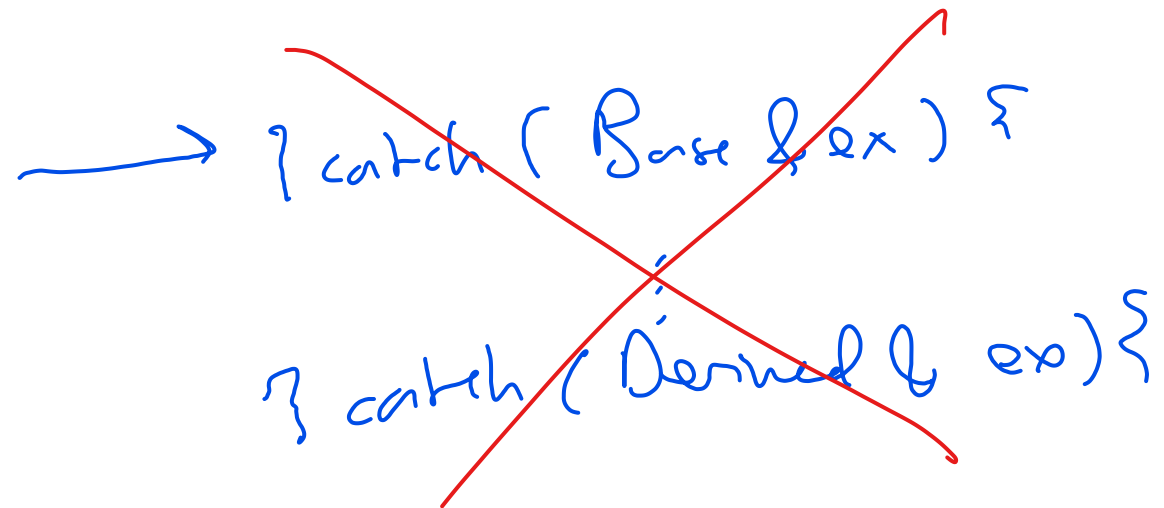


4. In the case of multiple matching catch blocks for a thrown exception, which one actually catches the exception?

The catch clauses are checked in order. The first one that matches the type of the thrown exception is the one that is executed.

So, you should order your catch clauses in the order from most-specific (derived exception classes) to most-general (base exception classes.)

If your program defines custom exception types, it's a good idea to have them inherit from one of the "standard" exception classes (e.g., `std::runtime_error`).



A handwritten diagram illustrating an incorrect catch order. A blue arrow points to two lines of C++ code: `{ catch ( Base & ex ) {` and `{ catch ( Derived & ex ) {`. A large red 'X' is drawn over the entire code block, indicating that this order is wrong. The code is written in blue ink.

## 5. How do you get the message associated with an exception?

The standard exception classes (derived from `std::exception`) have a virtual member function called `what()` which returns a `std::string`.

This string is a text message describing the reason for the exception.

The constructors for the standard exception classes accept a string value to set this message. E.g.

```
throw new std::runtime_error("Couldn't open input file");
```









