

# 601.220 Intermediate Programming

Spring 2023, Day 35 (April 19th)

# Today's agenda

- Day 35 recap questions
- Exercise 35

## Reminders/Announcements

- Final project due by 11pm on **Friday, April 28th**

→ My office hours tomorrow (Thurs 4/20)  
will be 12:15 - 2:15

## Day 35 recap questions

- ❶ What is the difference between an unscoped and a scoped enum?
- ❷ Why do we use exceptions?
- ❸ 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?
- ❹ In the case of multiple matching `catch` blocks for a thrown exception, which one actually catches the exception?
- ❺ 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 type adds the enum members to the current namespace.

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

## Unscoped vs. scoped enum types

```
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.



## Error handling 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 need to be concerned whether this function returned true or false.



## Error handling with exceptions

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

The called can just assume that either the function will either succeed completely, or will 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.)*

## Using the read\_input function

```
// without exceptions
std::vector<double> data_vec;
if (!read_data(in, 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(in, 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;  
  
// 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 of a try 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.)

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

## Exercise 35

- Practice throwing and catching exceptions
- Talk to us if you have questions!

# Notes

# Notes



# Notes

# Notes

# Notes