Exception Handling

Try / Catch / Duck!

(just kidding, there is no duck)

https://github.com/alexander-katrompas/exception-handling-cplusplus

Why Exception Handling versus “normal” flow-control error handling?

Cleaner and more logical code: In normal error handling there are if/else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and less maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

No Choice: Many functions/object/procedure throw exceptions, and you have to handle them, so exception handling is fundamentally necessary to ***all*** programming and ***all*** languages.

How do you know whether to use flow control or exception handling?

- **DO NOT** use try/catch if you ***know*** something could *and* will go wrong, use flow control to account for it. For example:
  - **Checking user input**. You *know* the user can enter something wrong, so account for it. For example you ask the user for an integer, but they enter a letter. It's up to you to test the input (if/then statements) and take appropriate action.
  - **Reading csv data from a file**. You *know* a csv file may have mismatched columns, so check it as you read it using flow control (if/then statements).
  - **Divide by Zero Error**. You *know* if you divide a number by another, the divisor might be 0. Check it and take action using flow control (if/then statements)so an error is not thrown.

How do you know whether to use flow control or exception handling?

- **DO** use try/catch if you ***don't*** know something will go wrong or what could go wrong, but you *know* an exception could be thrown. For example:
  - **Opening a file**: The file could have been moved or locked by another process for which you don't have access or knowledge. Place the file open operation in a try/catch.
  - **Allocating memory**: The OS may refuse to allocate memory for unknown reasons, so place the allocation in a try/catch.
  - **Another process or library** might throw an exception. This is by far the most common case. Someone else (or you) built a function/method/call that can throw an exception. When you use that function/method/call you must place it inside a try/catch block and handle the error appropriately.

Example

```
#include <iostream>
using namespace std;

void some_funtion() {
    /* Assume this function does some processing,
     * something goes wrong and an exception is thrown*/
    throw 1;
}

int main() {

    try {
        // call some function that might throw an int error
        some_funtion();
    } catch (int e) {
        // handle exception here
        cout << "The error thrown is: " << e << endl;
    }

    return 0;
}
```

Try/Catch: To Flow Control or not to Flow Control? *Not*!

- Try/Catch statements are ***not*** a flow control tool.
- In other words, Try/Catch statements are ***not*** fancy if statements (except in Python where it is considered “Pythonic” and debatable as good practice).
- Exception handling is just what it says, it’s handling an ***exception*** to the normal flow. *Do not use try/catch as flow control!*
- An *exception* means something went wrong that is an ***exception***, so handle it ***outside*** the normal flow of control.
- Most languages treat exceptions this way (Java, JavaScript, PHP, etc.)
- There are exceptions (pun intended). Python Try/Except statements *are* used for flow control. Whether this is a good idea or not in Python is debatable.

Failure to Catch

This program will terminate abnormally (notice the thrown error is char but the catch is for int).

You must catch all thrown exceptions explicitly or in a generic catch.

Implicit type conversion does ***not*** happen for primitive types in catches.

```
#include <iostream>
using namespace std;

void some_funtion() {
    /* Assume this function does some processing,
     * something goes wrong and an exception is thrown*/
    throw  'a';
}

int main() {

    try {
        some_funtion();
    } catch (int e) {
        // handle exception here
        cout << "The error thrown is: " << e << endl;
    }

    return 0;
}
```

Default Catch

Similar to many programming constructs like if/else and switch, try/catch has a default case (...)

This will work regardless of what is thrown.

```
#include <iostream>
using namespace std;

void some_funtion() {
    /* Assume this function does some processing,
     * something goes wrong and an exception is thrown*/
    throw  'a';
}

int main() {

    try {
        some_funtion();
    } catch (...) {
        // handle exception here
        cout << "An error was thrown." << endl;
    }

    return 0;
}
```

Many thrown exceptions are objects - Get used to it.

- A derived class exception should be caught before a base class exception.
- The C++ standard library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type.
- In C++, all exceptions are unchecked. The compiler doesn't check whether an exception is caught or not. For example, in C++, it is not necessary to specify all uncaught exceptions, although it's a recommended practice to do so.

More About Exception Handling

- Functions/Methods can choose which exceptions to handle: A function can throw many exceptions, but may choose to only handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
- In C++, primitives, ADTs, and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Catch derived class exceptions first!

Notice the derived class is caught first.

This will work to catch the derived class properly, and will also catch a base class if it is thrown.

```
#include <iostream>
using namespace std;

// these are "dummy" empty classes for demo purposes
class BaseError {};
class DerivedError : public BaseError {};

// this function will throw a derived object
void some_funtion() {
    DerivedError d;
    throw d;
}

int main() {
    try {
        some_funtion();
    } catch (DerivedError e) {
        // this will work properly
        cout << "The derived error was caught." << endl;
    } catch (BaseError e) {
        cout << "The base error was caught." << endl;
    }

    return 0;
}
```

Catch derived class exceptions first!

Notice the base class is caught first.

This will ***not*** work to catch the derived class, and will always report the base class was caught.

```
#include <iostream>
using namespace std;

// these are "dummy" empty classes for demo purposes
class BaseError {};
class DerivedError : public BaseError {};

// this function will throw a derived object
void some_funtion() {
    DerivedError d;
    throw d;
}

int main() {
    try {
        some_funtion();
    } catch (BaseError e) {
        cout << "The base error was caught." << endl;
    } catch (DerivedError e) {
        // this will NEVER execute!
        cout << "The derived error was caught." << endl;
    }

    return 0;
}
```