

C++ Exceptions

Tom Latham

(based on material from Ben Morgan)

Error Handling in mpags-cipher

- In mpags-cipher we had several cases where we needed to handle errors
 - Bad command line input
 - Invalid Cipher Key
- Errors were indicated using `bool` returns, but
 - That doesn't provide much information on the cause
 - Calling code can happily ignore the return value...

Throwing Exceptions

- An exception is nothing exceptional - it can be any object that is Copyable or Movable
- Exceptions are created (“raised, thrown”) using the throw keyword followed by the object to “throw”

```
int foo() {  
    ...  
    throw true;  
    ...  
    return 42;  
}
```

```
int main() {  
    int answer {foo()};  
    return 0;  
}
```

Exception Propagation

- A throw results in quite different behaviour to a return
- The thrown object is passed “up the stack” of calls until it is handled.
- When handled, the stack is “unwound” with destructors of any fully created objects invoked.
- If the exception is never handled, it passes out of main, resulting in an immediate termination.
- In this case, whether destructors are invoked is implementation defined.

Exception Propagation

```
int bar() {  
    BObject b {};  
    throw true;  
    return 1;  
}  
  
int foo() {  
    AObject a {};  
    bar();  
    return 42;  
}  
  
int main() {  
    int answer {foo()};  
    return 0;  
}
```

Stack Before throw:

BObject::BObject()
bar()
AObject::AObject()
foo()
main()

On Stack Unwind, call

BObject::~~BObject()
AObject::~~AObject()

Catching Exceptions

- To handle exceptions, we wrap code that may emit them in a try/catch block.
- The catch parts specify the types of exception object this block can handle (any others propagate further)

```
int main() {  
    try {  
        somethingThatMightThrow();  
    } catch (bool& e) { //Catch by reference to avoid slicing  
        std::cout << "Handling bool exception\n";  
    } catch (int& e) {  
        std::cout << "Handling int exception\n";  
    }  
    return 0;  
}
```

Exercise: handling exceptions from `std::stoul` function

- In one of the CaesarCipher constructors we convert the key from a string to an unsigned integer
- At present we do a prior check that each character in the string is a digit
- However, the `std::stoul` function will throw exceptions if the conversion doesn't work: http://en.cppreference.com/w/cpp/string/basic_string/stoul
 - NB we're not actually handling one of the cases at the moment! Try it out and see what happens...
- Remove the explicit check of the string and instead handle the two possible exceptions that could be thrown (use the code on the previous slide as a guide)

<stdexcept>

- Header that provides several generic concrete classes that inherit from the `std::exception` base class (itself defined in the `<exception>` header), e.g.

`std::logic_error`, `std::runtime_error`

- Best to implement exception types specific to the project, e.g. for `mpags-cipher`, could have:

```
class MissingArgument
class UnknownArgument
class InvalidKey
```

- In effect, we use the type to decide how to handle the error

Writing an exception class

- There is very little that needs to be written (you'll be glad to hear!)
- The most effective way to proceed is to derive from one of the existing standard library exception classes, then you just need to implement a constructor that delegates to that of the base class

```
class MissingArgument : public std::invalid_argument {  
    public:  
        MissingArgument( const std::string& msg ) :  
            std::invalid_argument(msg)  
        {  
        }  
};
```

Using a custom exception class

- We can then use our custom class by doing something like:

```
throw MissingArgument("-i/--infile requires a filename argument");
```

- We can then handle it as follows:

```
try {  
    processCommandLine(cmdLineArgs, settings);  
} catch ( const MissingArgument& e ) {  
    std::cerr << "[error] Missing argument: " << e.what() << std::endl;  
    return 1;  
}
```

Exercise: using exceptions in processCommandLine

- There are several problems that can occur when processing the command line arguments (in our processCommandLine function)
- At present we set a boolean flag to indicate an error, print an error message and return the boolean
- But this means that while the calling code can (optionally!) find out that something has gone wrong, it can't know what was the nature of the problem
- So let's remove the boolean return and instead throw custom exception objects to indicate the different problems
- Use the previous few slides to help you to implement this

Exercise: using exceptions in VigenereCipher

- There is also a potential problem in the `VigenereCipher`, where an empty key prevents this cipher from functioning
- At present we simply set the key to a default value "VIGENEREEXAMPLE" and print a warning message to say what has happened and what we've done about it
- But changing the key to a value that has not been requested is a bit unsatisfactory
- Instead, we can throw a custom exception object, `InvalidKey`, to indicate the problem, which the `main` function can catch and act on
- Use the previous few slides to help you to implement this

Traps and Pitfalls

- Though exceptions offer an easy error handling mechanism, their use does require a bit of care because of the stack unwinding
- For example, if you've new'd an object then throw, the object won't be deleted (memory leak)
 - Using Smart Pointers helps here!
- Exception Safety: ensuring that an object isn't corrupted when one of its member functions throws.

Further Reading

- The two best starting points for Exceptions in C++ are the Super FAQ and Core Guidelines:
 - <https://isocpp.org/faq>
 - <https://github.com/isocpp/CppCoreGuidelines>
- Also see
 - <http://exceptionsafecode.com>

```
1 #include <exception>
2 #include <iostream>
3 #include <memory>
4
5 struct A {
6     A() {std::cout << "[A::A()]\n";}
7     ~A() {std::cout << "[A::~~A()]\n";}
8 };
9
10 struct B {
11     B() {std::cout << "[B::B()]\n";}
12     ~B() {std::cout << "[B::~~B()]\n";}
13 };
14
15 struct C {
16     C() {std::cout << "[C::C()]\n";}
17     ~C() {std::cout << "[C::~~C()]\n";}
18 };
19
20
21 void somethingThatThrows() {
22     A foo {};
23     B bar {};
24     auto baz = std::make_unique<C>();
25
26     std::cout << "About to throw\n";
```

Example

<https://github.com/cpp-pg-mpags/mpags-cpp-extra>