WARWICK
THE UNIVERSITY OF WARWICK

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” or “thrown”) using the `throw` keyword followed by the object to be thrown

```
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 1: handling exceptions from std::stoul

- 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
```

https://en.cppreference.com/w/cpp/header/stdexcept

- 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;
}
```

Documenting exceptions

- The exception objects themselves can be documented just as any other class
- However, it is important to allow document which exceptions may be emitted by a given function
  - For example, our CaesarCipher constructor:

```
/**
 * Create a new CaesarCipher with the given key
 *
 * The string will be converted to an unsigned integer.
 * If the conversion fails an InvalidKey exception will be emitted.
 *
 * \param key the key to use in the cipher
 *
 * \exception InvalidKey will be emitted if the supplied string cannot be
 *                       successfully converted to a positive integer
 */
explicit CaesarCipher( const std::string& key );
```

Testing for exceptions

- You should include test cases in your unit tests for the exceptions that may or may not be emitted from a given function

- In the Catch framework there are the useful `REQUIRE_THROWS_AS` and `REQUIRE_NOTHROW` macros, which allow you to do just that

- So you can call a function with a configuration that should not throw and test that with `REQUIRE_NOTHROW`

- And you can call it with a configuration that you expect to throw a particular type of exception and test that it does so using `REQUIRE_THROWS_AS`

- See the Catch documentation for further details:

Exercise 2: 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 3: 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

Exercise 4: using exceptions in CaesarCipher

- We can also improve further the behaviour of the second `CaesarCipher` constructor (where the key is provided as a string)

- At present, if the string -> unsigned integer conversion fails, we simply set the key to a default value of 0 and print a warning message to say what has happened and what we've done about it

- But, again, 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://isocpp.org/guidelines
- Also see
  - http://exceptionsafecode.com

exceptions.cpp (~/tmp/day6) - VIM

```cpp
#include <exception>
#include <iostream>
#include <memory>

struct A {
  A() {std::cout << "[A::A()]\n";}
  ~A() {std::cout << "[A::~A()]\n";}
};

struct B {
  B() {std::cout << "[B::B()]\n";}
  ~B() {std::cout << "[B::~B()]\n";}
};

struct C {
  C() {std::cout << "[C::C()]\n";}
  ~C() {std::cout << "[C::~C()]\n";}
};


void somethingThatThrows() {
  A foo {};
  B bar {};
  auto baz = std::make_unique<C>();

  std::cout << "About to throw\n";
```

exceptions.cpp[cpp] [1/42][1]

Another example

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