Exceptionally Bad:
The Story on the Misuse of Exceptions and How to Do Better

PETER MULDOON

Cppcon
The C++ Conference
2023
October 01 - 06

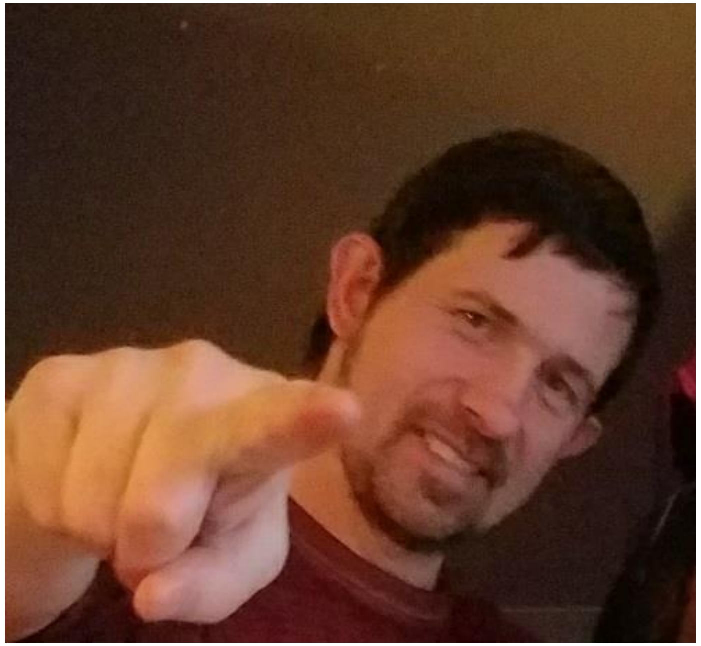
Exceptions in C++

Better Design Through Analysis of Real World Usage

PETER MULDOON

Cppcon
The C++ Conference
2023
October 01 - 06

Who Am I



- **Starting using C++ professionally in 1991**
- **Professional Career**
  - **Systems Analyst & Architect**
  - **21 years as a consultant**
  - **Bloomberg Ticker Plant Engineering Lead**
- **Talks focus on practical Software Engineering**
  - **Based in the real world**
  - **Take something away and be able to use it**

Questions

#include <slide_numbers>

Where will we be going ?

- Talk will be about Exceptions as seen in C++
- Look at original goals/ideals of exception handling
- Look at the mechanics of throw/catch
- Look at Exceptions in both their use and design as seen in real code
- Better thinking on the handling of error use cases
- Better thinking on the design of an Exception class

Warning

May be controversial

Foreword

Things to note about Exceptions in C++

- There is no exception class attribute/keyword
- No formal definition of what an Exception is
  - What characteristics denotes an Exception

Ergo, *Usage* is the primary characteristic of Exceptions

Exceptional Ideals

What were Exceptions meant to give us ?

The following ideals evolved for C++ exception handling:

[1] Type-safe transmission of arbitrary amounts of information from a throw-point to a handler.

[2] No added cost (in time or space) to code that does not throw an exception.

[3] A guarantee that every exception raised is caught by an appropriate handler.

[4] A way of grouping exceptions so that handlers can be written to catch groups of exceptions as well as individual ones.

[5] A mechanism that by default will work correctly in a multi-threaded program.

[6] A mechanism that allows cooperation with other languages, especially with C.

[7] Easy to use.

[8] Easy to implement.

Most of these ideals were achieved, others ([3], [8]) were considered too expensive or too constraining and were only approximated.

The Design and Evolution of C++ by Bjarne Stroustrup, Copyright © 1994

Exceptional Ideals

What do Exceptions now give us ?

- Easy to read the happy path
  - Lack of clutter in code
- Decouples error reporting from error handling
- Exceptions can’t be passively ignored *
- Simpler way for handling constructor and operator errors
- Modern and “recommended” way of error handling in the language
  - Endorsed by stdlib

Exceptional Ideals

```
enum MyStatus { good, not_good,  bad};

MyStatus applyX(MyClass& out);

int main() {
  bool stay_active = true;
  while (stay_active) {
    MyClass me{.name_="Pete", .age_ = 21};
    if(me.notValid()) {
      std::cout << "error : Object invalid" << me << std::endl;
      exit(-1);
    }
    MyStatus rc = applyX(me);
    if(rc != good)
      std::cout << "error : " << rc << std::endl;
    …
  }
}
```

Function signature

Clutter

Exceptional Ideals

```cpp
enum MyStatus { good, not_good,  bad};

// Throws MyStatus on error
void apply(MyClass& out);

int main() {
  bool stay_active = true;
  while (stay_active) {
    try {
     MyClass me{"Pete", 21};
     apply(me);
    } catch (const MyStatus& s) {
     std::cout << "error : " << s << std::endl;
    } catch (...) {
       std::cout << "Unknown error : " << std::endl;
    }
    …
  }
}
```

Function signature changed

Happy Path (clean)

Error handling

Exception Swallower

Exceptional Ideals

What do exceptions now give us ?

- Easy to read the happy path
    - Lack of clutter in code
- Decouples error reporting from error handling
- Exceptions can’t be passively ignored
- Simpler way for handling constructor and operator errors
- Modern and “recommended” way of error handling in the language
    - Endorsed by stdlib

Most applications need the exception swallower - to prevent being taken down

Exceptional Definitions

What is an Exception ?

- Mechanism/Behavior-based view
  - Something that's thrown
  - Then gets caught (hopefully)

How exceptions work

Throwing Exceptions :

- The memory for the exception object is allocated in an unspecified way
  - Uses the heap even for an int - *expensive*
  - Outside of the regular efficient stack return of variables
- Separate stack unwinding mechanism - *expensive*
  - 2 types of unwinding
    - Frame-based
    - Table-based

Catching exceptions :

- exception handler is from a point previously passed by the execution
- Use RTTI to do polymorphic catching of exceptions - *expensive*
  - Not specified in standard but always used in practice
  - Exception hierarchies are typically long chains of class hierarchies
- The implementation may then deallocate the memory for the exception object, any such deallocation is done in an unspecified way – *expensive*

How exceptions work

If an Exception is thrown

- During stack unwinding, your program is terminated.
- Caught by a matching handler, the exception is no longer in flight unless rethrown.
- If no matching handler is found, the function std::terminate is invoked; whether or not the stack is unwound before this invocation of std::terminate is implementation-defined

Exception hygiene

- Throw by value
- Catch by (const) reference
- Rethrow using throw with no arguments
- Catch handlers with derived classes placed before catch handlers with base classes

Exception safety is given in terms of the three exception guarantees that a function can provide:

- *No-throw guarantee*
- *Basic guarantee*
- *Strong guarantee*

Exceptional Definition

What is an Exception ?

- Mechanism/Behavior-based view
  - Something that's thrown
  - That gets caught (hopefully)
- Philosophy-based view
  - Rare unhappy path events
  - Non-trivial errors

Exceptional Philosophy

Question 1 :

When is it appropriate to use an Exception ?

- Run-time errors where a function cannot do what is advertised
  - Guidelines : Throw an exception to signal that a function can't perform its assigned task
- Constructor / operator failures
- Undefined/unrecoverable system state ?
  - Memory corruption
  - Memory exhaustion

> Trying to provide facilities that allow a single program to recover from all errors is misguided and leads to error-handling strategies so complex that they themselves become a source of errors.

The Design and Evolution of C++ by Bjarne Stroustrup, Copyright © 1994

Exceptional Philosophy

Question 1 (contd) :

What are the use cases that Exceptions were designed for ?

- Program has encountered a serious error, and getting out of this control flow to prevent data corruption or other damage is more important than trying to continue onward.
- Program has encountered a situation that cannot be dealt with properly because it does not have enough context but the function that called it (or something further up the call stack) should know how to handle the error – you hope

If you're using Exceptions for something else, there’s likely a better way

Exceptional Philosophy

Question 2 :

Are Exceptions for very rare events only ?

- Guidelines : Keep error handling separated from "ordinary code". C++ implementations tend to be pessimized based on the assumption that exceptions are rare.
- Microsoft - Use exception handling if the event doesn't occur often, that is, if the event is truly exceptional

How much of your flow (upper-bound) should be taking exception paths ?

- Does anyone measure this ?

How do I handle "frequent" minor errors ?

- How can 3rd party libraries know this relationship ?
- Error codes / state – saddled with inferior mechanism!

if you use Exceptions for normal situations, how do you locate unusual/serious problems ?

Exceptional Philosophy

Final Word :

> The following assumptions were made for the design:
> – Exceptions are used primarily for error handling.
> – Exception handlers are rare compared to function definitions.
> – Exceptions occur infrequently compared to function calls.
>
> **A History of C++: 1979–1991**
>
> *Bjarne Stroustrup*

Exceptional Philosophy

Conclusion :

Only use Exceptions for functional/recoverable errors

- Not for memory corruption/exhaustion problems

Prefer to keep exceptions as “Rare” as you can, meaning for serious, uncommon errors

Exceptional Definition

What is an Exception ?

- Behavior-based view
  - Something that's thrown
  - That gets caught (hopefully)
- Philosophy-based view
  - Rare unhappy path events
  - Non-trivial errors
- Usage-based view
  - What are people doing with exceptions in real code ?
  - What should people be doing with exceptions in real code ?

Exceptional Resource Management

Situation : Need to explicitly give back / release a resource acquired during an operation before leaving the function / exception block.

Exceptions can leave the function at any time potentially resulting in a leak

- Need to ensure cleanup before exiting

Pattern : Exception block catches the exception, releases a resource and usually rethrows the exception

Exceptional Resource Management

```
int process_file(const std::string& fileName_) {
  FILE * f = ::fopen(fileName_.c_str(), "r");
  …
  int rc;
  try  {
     do  {
        rc = printTaggedDataDumpEntries(OStream, f);
     } while (!::feof(f) && ::ftell(f) < file_length);
     return rc;
  }
  catch (const StreamingException& e)  {
     ::fclose(f);
     throw;
  }
  catch (const std::exception& e)  {
     ::fclose(f);
     throw;
  }
  ::fclose(f);
  return rc;
}
```

Bug

Resource release

Exceptional Resource Management

Use RAII :

```
struct FileManager {
    FileManager(const std::string& fileName_)
        : fptr{::fopen(fileName_.c_str(), "r")} {};
    ~FileManager() { ::fclose(fptr); }
    FILE* fptr;
};

int process_file(const std::string& fileName_) {
    FileManager fmgr{fileName_};
    FILE* f = fmgr.fptr;
    …
    int rc;
    do {
        rc = printTaggedDataDumpEntries(OStream, f);
    } while (!::feof(f) && ::ftell(f) < file_length);
    return rc;
}
```

```
class FilePtr {
    FILE* p;
public:
    FilePtr(const char* n, const char* a) { p = fopen(n,a); }
    FilePtr(FILE* pp) { p = pp; }
    ~FilePtr() { fclose(p); }

    operator FILE*() { return p; }
};
```

The Design and Evolution of C++ by Bjarne Stroustrup, Copyright © 1994

Exceptional Resource Management

Situation : Need to explicitly give back / release a resource acquired during an operation before leaving the function / exception block.

Exceptions can leave the function at any time resulting in a leak

- Cleanup resides in catch block with rethrow

Pattern : Exception block catches the exception, releases a resource and usually rethrows the exception

Conclusion : Use RAII over catch handlers for resource management

Exceptional Trier

Situation : I have a potentially temporary problem that retrying may solve.

Exceptions can be caught and used as a retry counting mechanism

Kind of messy but neat

Pattern : Exception is in a loop and has a conditional rethrow

Exceptional Trier

```
// if problem throw
void connect(…);

bool try_connection(int max_retries) {
   bool connected = false;
   int retry_count = 0;
   while(!connected) {
      try {
         connect (…);
         connected = true;
      }
      catch(const std::runtime_error&) {
         if (retry_count++ > max_retries)
           throw;
      }
   }
   …
}
```

Conditional rethrow

Exceptional Trier

```
bool try_connection(int max_retries) {
   bool connected = false;
   int retry_count = 0;
   while(!connected) {
      try {
         connect (…);
         notify(connected = true);
      }
      catch(const std::runtime_error&) {
        if (retry_count++ > max_retries) {
           notify(connected);
           throw;
        }
      }
   }
   …
}
```

Conditional rethrow

Exceptional Trier

```
// if invalid return false
bool connect(...)

bool try_connection(int max_retries) {
  bool connected = false;
  int retry_count = 0;
  while (!connected && (retry_count++ < max_retries)) {
    connected = connect(...);
  }
  notify(connected);
  return connected;
}
```

Simple return value

Exceptional Trier

Situation : I have a potentially temporary problem that retrying my solve.

Exceptions can be caught and used as a retry counting mechanism

Kind of messy

Pattern : Exception is in a loop and has a conditional rethrow;

Conclusion : Use a return status for loop control

Exceptional Pass the buck

Situation : Any problem whatsoever, throw an exception.

Exceptions allow the thrower to offload the problem to the catcher

Users generally need catch blocks all throughout their code

All responsibility goes higher up the chain

- Right to the top – effectively transaction is wiped
- Needs local handling to mitigate transaction wiping

Pattern : External libraries, modular functionality expresses all problems via an Exception

Exceptional Pass the buck

Some Questions :

Is having try-catch blocks all throughout the code base OK ?

- Around multiple single function calls ?

Is having exception swallowers all throughout the code base OK ?

No, as part of the benefit of exceptions is the lack of these constructs throughout the code.

- Try - catch blocks are ugly
- Straight swap of try- catch blocks for if statements is *not* desirable
- Not efficient due to try - catch mechanics
- if you use exceptions for “normal situations”, how do you locate serious errors ?

Exceptional Pass the buck

A comment :

A notable exception to this agreement was Doug McIlroy, who stated that the availability of exception handling would make systems less reliable because library writers and other programmers will throw exceptions rather than try to understand and handle problems. Only time will tell to what extent Doug's prediction will be true.

The Design and Evolution of C++ by Bjarne Stroustrup, Copyright © 1994

Doug's influence on the development of both C and C++ cannot be overestimated. I cannot remember a single critical design decision in C++ that I have not discussed at length with Doug.

The Design and Evolution of C++ by Bjarne Stroustrup, Copyright © 1994

Exceptional Pass the buck

Situation : Any problem whatsoever, throw an exception.

Exceptions allow the thrower to offload the problem to the catcher

- Users generally need catch blocks all throughout their code
- All responsibility goes higher up the chain
  - Right to the top – effectively transaction wiped
  - Needs local handling to mitigate transaction wiping

Pattern : External libraries, modular functionality expresses all problems via an exception

Conclusion : Prefer to use exceptions only for unusual (rare) situations

Exceptional Logging

Situation : Exception thrower is missing needed context for an error

Exceptions when thrown can be missing vital information needed for error investigation.

1. Log error with extra information in handler and rethrow
   Logging reported deep in stack instead of at handler site
2. Repackage the Exception as a different type with additional information and rethrow
   Logging reported at handler site

This can often lead to thinking modes where the logging and the exception/error type are conflated

Pattern : Logging/repackaging the exception in the handler with extra information and then rethrow for error handling elsewhere

Exceptional Logging

```
enum class component_type { text, graphic, heading};

int apply(int key, component_type type) {
  try {
    …
    auto name = get_name_from_type(type);
    return transmit(name);
  }
  catch(const std::runtime_error& e) {
    std::cerr << "key : " << key << ", type : " << type << " - " << e.what() << std::endl;
    throw;
  }
}
```

Context Added

Exceptional Logging

```
enum class component_type { text, graphic, heading};

int apply(int key, component_type type) {
   try {
     …
     auto name = get_name_from_type(type);
     return transmit(name);
   }
   catch(const std::runtime_error& e) {
      MyException myexcept{key, type};
      myexcept.msg() = e.what();
      throw myexcept;
   }
}
```

Context added via new exception type

Exceptional Logging

```
class MyException {
public:
   MyException(std::string str):err_str(std::move(str)){}

   std::string& what() noexcept {return err_str;}
   const std::string& what() const noexcept {return err_str;}
private:
   std::string err_str;
}
```

Mutable

Exceptional Logging

```
enum class component_type { text, graphic, heading};

int funca(int key, component_type type) {
   try {
      …
      auto name = get_name_from_type(type);
      return transmit(name);
   }
   catch(MyException& e) {
      e.what() += format_add_keytype(key, type);
      throw;
   }
}
```

Context added
Into current exception

Exceptional Logging

Situation : logging handled in the exception handler

Exceptions when thrown can be missing context needing for error investigation.

- Log error with extra information in handler and rethrow
- Repackaging Exception to a different type and rethrow
- Logging reported deep in stack instead of at handler site

This can often lead to thinking modes where the logging and the exception/error type are conflated

Pattern : Exception thrower is missing extra context for an error

Conclusion : Use an exception type with mutable logging

Exceptionally Hard to find

Situation : Searching for something and the item cannot be found.

Exceptions can be a “clean” way to indicate a function has failed to uphold its end of the bargain

Throw an unfound tantrum

Pattern : Exception thrower is a search function that returns the object when found and throws an exception when its missing

Exceptionally Hard to find

```
Order findOrder(unsigned int id) {
    auto it = orders.find(id);
    if(it == orders.end())
     throw MyException ("Order not found", id);
    return it->second;
}

bool funca(unsigned int id) {
    try {
     Order ord = findOrder(id);
     std::cout << “Order id : " << id << “, value : " << ord << std::endl;
     return true;
    }
    catch(MyException& e) {
     std::cerr << “Error Order missing : " << e.what() << std::endl;
    }
    return false;
}
```

Exceptionally Hard to find

```
std::optional<Order> findOrder(unsigned int id) {
    auto it = orders.find(id);
    if(it == orders.end())
      return std::nullopt;
    return it->second;
}

bool funca(unsigned int id) {
    std::optional<Order> opt_ord = findOrder(id);
    if(opt_ord) {
      std::cout << “Order id : " << id << “, value : " << opt_ord->value_ << std::endl;
      return true;
    }
    std::cerr << "Order not found for id : " << id << std::endl;
    return false;
}
```

Exceptionally Hard to find

```
std::optional<Order> findOrder(unsigned int id) {
    auto it = orders.find(id);
    if(it == orders.end())
      return std::nullopt;
    return it->second;
}

bool funca(unsigned int id) {
    std::optional<Order> opt_ord =
      findOrder(id).or_else([] -> std::optional<Order> {std::cerr << "Order not found"; return std::nullopt;});
    return opt_ord.has_value();
}
```

Exceptionally Hard to find

Situation : Search for something and cannot find it.

Exceptions can be a “clean” way to indicate function has failed to uphold its end of the bargain

Throw an unfound tantrum

Pattern : Exception thrower is a search function that returns object when found and throws an exception when its missing

Conclusion : Prefer to return an optional which indicates found(value)/not found(empty) especially if stack return is shallow

Exceptional Data Passing

Situation : Handler needs to perform specific actions based on data at the throw site

Exceptions can transfer information up the stack to the handler

More expensive than a traditional value return but easier to invoke

Pattern : Exception contains data used to perform handler actions (as opposed to logging and reset)

Exceptional Data Passing

```
struct ExceptionBadClient : std::runtime_error {
  using std::runtime_error::runtime_error;
  Msg msg_;
};

// Throws Exception on failure
Msg apply(const Info& data);

bool process(const Info& data) {
  try {
    Msg request = apply(data);
    return send(request);
  }
  catch(ExceptionBadClient & e) {
     std::cerr << "Error : " << e.what() << std::endl;
     return send_error(e.msg_);
  }
  return true;
}
```

exception data

Exceptional Data Passing

```
struct ExceptionBadClient : std::runtime_error {
  using std::runtime_error::runtime_error;
  Msg msg_;
};

using ExpectedProcessing = std::expected<Msg, ExceptionBadClient >;
ExpectedProcessing apply(const Info& data);

bool process(const Info& data) {
  ExpectedProcessing result = apply(data);
  if (result.has_value())
    return send(*result);
  std::cerr << "error : " << result.error().what() << std::endl;
  return send_error(result.error().msg_);
};
```

Exceptional Class Design

```
struct ExceptionBadClient : std::runtime_error {
  using std::runtime_error::runtime_error;
  Msg msg_;
};
struct ExceptionBadOrder : std::runtime_error {
  using std::runtime_error::runtime_error;
  Msg msg_;
};
struct ExceptionFormatError : std::runtime_error {
  using std::runtime_error::runtime_error;
  FormatCode fcode_;
};
struct ExceptionUnknownTransaction : std::runtime_error {
  using std::runtime_error::runtime_error;
  Info info_;
};
```

Exceptional Class Design

```
template<typename DATA_T>
class MyException {
public:
  MyException(std::string str, DATA_T data)
   :err_str(std::move(str)),data_(std::move(data)){}

  DATA_T& data(){ return data_;}
  const DATA_T& data() const noexcept { return data_;}

  std::string& what(){return err_str;}
  const std::string& what() const noexcept {return err_str;}

private:
  std::string err_str;
  DATA_T data_;
};
```

Exceptional Data Passing

```
using ExceptionBadClient = MyException<Msg>;
```

```
using ExpectedProcessing = std::expected<Msg, ExceptionBadClient >;
ExpectedProcessing apply(const Info& data);

bool process(const Info& data) {
  ExpectedProcessing result = apply(data);
  if (result.has_value())
    return send(*result);
  std::cerr << "error : " << result.error().what() << std::endl;
  return send_error(result.error().data());
};
```

Exceptional Data Passing

Situation : Handler needs to perform specific actions based on data at the throw site

Exceptions can transfer information up the stack to the handler

More expensive than a traditional return but easy

Pattern : Exception contains data used to perform handler actions (as opposed to logging and reset)

Conclusion : if stack return is shallow prefer to return std::expected type of success data/failure data

Additional : Use a simple class template to transfer different failure data

Exceptional Control Flow

Situation : Handler needs to perform varying specific actions depending on the failure that occurs

Depending on the exception type that's caught, do something specific

- Frowned upon in theory but the very design of catch by type encourages this line of thinking in practice

Pattern : Many exception catch blocks on a single try and potentially each has specific functionality (as opposed to logging and reset)

Exceptional Control Flow

```
bool process(const Info& data) {
  try {
    Msg request = apply(data);
    return send(request);
  }
  catch(ExceptionOrderNotFound& e) {
    err_log(e.what());
    return send_error(e.msg_);
  }
  catch(ExceptionIllegalCurrency& e) {
    err_log(e.what());
    return send_error(e.msg_);
  }
  catch(ExceptionInvalidClientId& e) {
    err_log(e.what());
    return send_error(e.msg_);
  }
  return true;
}
```

Exceptional Control Flow

```
using ExceptionProcessing = std::variant<ExceptionOrderNotFound,
                                         ExceptionIllegalCurrency, ExceptionInvalidClientId>;

using ExpectedProcessing = std::expected<Msg, ExceptionProcessing>;
ExpectedProcessing apply(const Info& data);

bool process(const Info& data) {
  ExpectedProcessing result = apply(data);
  if (result.has_value())
    return send(*result);
  return std::visit([](auto& e){err_log(e.what()); return send_error(e.msg_);}, result.error());
}
```

Exceptional Control Flow

```
bool process(const Info& data) {
   try {
    Msg request = apply(data);
    return send(request);
   }
   catch(ExceptionOrderNotFound& e) {
    err_log(e.what());
    return send_warning(e.msg_);
   }
   catch(ExceptionIllegalCurrency& e) {
    err_log(e.what());
    return send_error(e.msg_);
   }
   catch(ExceptionInvalidClientId& e) {
    err_log(e.what());
    return handle_bad_client(e.msg_);
   }
   return true;
}
```

Utility

```
// Utility Class
template<typename... Ts>
struct Overloaded : Ts... {
  using Ts::operator()...;
};
template<typename... Ts>
Overloaded(Ts&&...) -> Overloaded<std::decay_t<Ts>...>;
```

Exceptional Control Flow

```
using ExceptionProcessing = std::variant<ExceptionOrderNotFound,
                                         ExceptionIllegalCurrency, ExceptionInvalidClientId>;
using ExpectedProcessing = std::expected<Msg, ExceptionProcessing>;

ExpectedProcessing apply(const Info& data);

bool process(const Info& data) {
  ExpectedProcessing result = apply(data);
  if (result.has_value())
    return send(*result);
  auto visitor = Overloaded{
    [&](ExceptionIllegalCurrency& ex) { err_log(e.what()); return send_error(ex.msg_);},
    [&](ExceptionOrderNotFound& ex) { err_log(e.what()); return send_warning(ex.msg_);},
    [&](ExceptionInvalidClientId& ex) { err_log(e.what()); return handle_bad_client(ex.msg_);}
  };
  return std::visit(visitor, result.error());
}
```

Exceptional Control Flow

Situation : Handler needs to perform varying specific actions depending on the failure that occurs

Depending on the exception type that’s caught, do something specific

- Frowned upon in general but the very design of catch by type encourages this line of thinking in practice

Pattern : Many exception catch blocks on a single try and potentially each has specific functionality (as opposed to logging and reset)

Conclusion : If stack return is shallow, prefer std::expected with an error type as a variant of exception types

Exceptional Hierarchies

Situation : I have lots of different error modes and want to use exception types to represent each

Exceptions can be many and varied, with the type indicating some categorization of the problem however the real information is (usually) in the message/data contained within.

Pattern : For a single try block, many exception catch handlers doing the same thing **OR** a single catch handler for the parent class

Exceptional Hierarchies

```
int main() {
  bool stay_active = true;
  while (stay_active) {
    try {
      ...
      check_even(ip_num);
      apply_subtraction(ip_num, subtractor);
      apply_division(ip_num, divisor);
      ...
    } catch (const std::logic_error& e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (const std::underflow_error& e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (const std::range_error& e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (const std::runtime_error & e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (...) {
      std::cout << "Unknown error : " << std::endl;
    }
  }
}
```

Many handlers

Exceptional Hierarchies

```
int main() {
  bool stay_active = true;
  while (stay_active) {
    try {
      ...
      check_even(ip_num);
      apply_subtraction(ip_num, subtractor);
      apply_division(ip_num, divisor);
      ...
    } catch (const std::underflow_error& e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (const std::range_error& e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (const std::runtime_error & e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (const std::logic_error& e) {
      std::cout << "error : " << e.what() << std::endl;
    } catch (...) {
      std::cout << "Unknown error : " << std::endl;
    }
  }
}
```

Many handlers

Exceptional Hierarchies

```
int main() {
    bool stay_active = true;
    while (stay_active) {
        try {
            ...
            check_even(ip_num);
            apply_subtraction(ip_num, subtractor);
            apply_division(ip_num, divisor);
            ...
        }
        catch (const std::exception& e) {
            std::cout << "error : " << e.what() << std::endl;
        }
        catch (...) {
            std::cout << "Unknown error : " << std::endl;
        }
        ...
    }
}
```

Single parent class handler

Exceptional Hierarchies

```
void check_even(int a) {
  if (a % 2 != 0)
    throw std::logic_error("Odd number given, even number required");
}
int apply_subtraction(unsigned int operand, unsigned int subtractor) {
  if (subtractor > operand)
    throw std::underflow_error("subtractor too large for operand");
  return operand - subtractor;
}
double apply_division(double operand, double divisor) {
  if (divisor == 0)
    throw std::range_error("zero divisor operation not allowed");
  return operand / divisor;
}
bool save_result(double operand) {
  if (operand < 0)
    throw std::runtime_error("illegal negative value");
  return true;
}
```

Many types but single function

Exceptional Hierarchies

```
void check_even(int a) {
   if (a % 2 != 0)
      throw std::exception("Odd number given, even number required");
}
int apply_subtraction(unsigned int operand, unsigned int subtractor) {
   if (subtractor > operand)
      throw std::exception("subtractor too large for operand");
   return operand - subtractor;
}
double apply_division(double operand, double divisor) {
   if (divisor == 0)
      throw std::exception("zero divisor operation not allowed");
   return operand / divisor;
}
bool save_result(double operand) {
   if (operand < 0)
      throw std::exception("illegal negative value");
   return true;
}
```

Single function
Single type

```
error: no matching function for call to
'std::exception::exception(const char [39])'
```

Exceptional Hierarchies

```
void check_even(int a) {
  if (a % 2 != 0)
    throw std::runtime_error("Odd number given, even number required");
}
int apply_subtraction(unsigned int operand, unsigned int subtractor) {
  if (subtractor > operand)
    throw std::runtime_error("subtractor too large for operand");
  return operand - subtractor;
}
double apply_division(double operand, double divisor) {
  if (divisor == 0)
    throw std::runtime_error("zero divisor operation not allowed");
  return operand / divisor;
}
bool save_result(double operand) {
  if (operand == 0)
    throw std::runtime_error("illegal negative value");
  return true;
}
```

Single function
Single type

Exceptional Hierarchies

Some questions

If the catch handler is by parent class, why is the thrower not throwing the parent class ?

Why are there derived classes at all, assuming no control flow is being used ?

If the catch handler is performing the same operations for different exceptions, why are these not represented by a single exception class ?

Exceptional Hierarchies

Some questions (contd)

std::exception has no string constructor , why ?

std::exception

    std::runtime_error : public std::**exception**

    std::logic_error : public std::**exception**

The class **std::runtime_error** defines the type of objects thrown as exceptions to report errors *presumably* detectable only when the program executes

The class **std::logic_error** defines the type of objects thrown as exceptions to report errors *presumably* detectable before the program executes, such as violations of logical preconditions or class invariants

These classes are *identical* in every respect except for the name

Exceptional Class Design

```
struct ExceptionBadValue : std::runtime_error {
  using std::runtime_error::runtime_error;
};
struct ExceptionBadId : std::runtime_error {
  using std::runtime_error::runtime_error;
};
struct ExceptionEmptyClientString : std::runtime_error {
  using std::runtime_error::runtime_error;
};
struct ExceptionTransactionFormat : std::runtime_error {
  using std::runtime_error::runtime_error;
};
```

Exceptional Hierarchies

Some questions (contd)

std::exception has no string constructor , why ?

std::exception

    std::runtime_error : public std::**exception**

    std::logic_error : public std::**exception**

The class **std::runtime_error** defines the type of objects thrown as exceptions to report errors *presumably* detectable only when the program executes

The class **std::logic_error** defines the type of objects thrown as exceptions to report errors *presumably* detectable before the program executes, such as violations of logical preconditions or class invariants

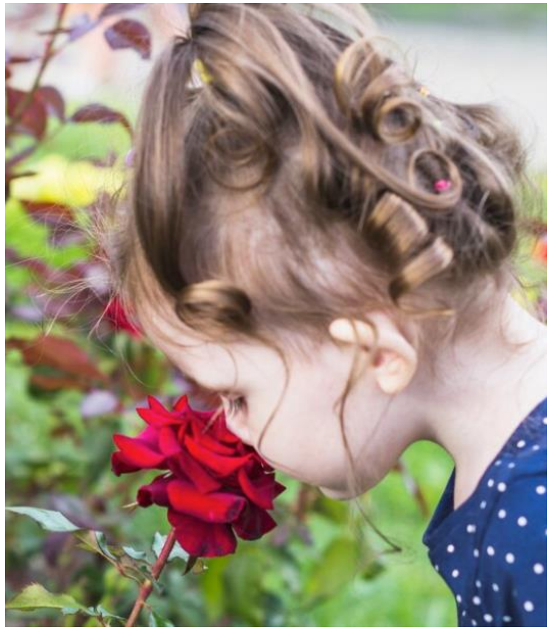
These classes are *identical* in every respect except for the name

Exceptional Hierarchies

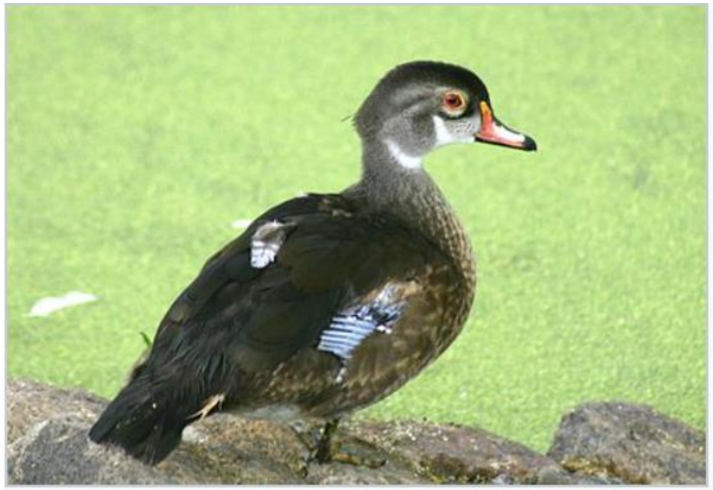
Back to basics :

"A rose by any other name would smell as sweet" is a popular adage from William Shakespeare's play Romeo and Juliet. The reference is used to state that the names of things do not affect what they really are.

The duck test—"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"—suggests that something can be identified by its habitual characteristics.



Credit : freepik.com/free-photo/close-up-girl-smelling-beautiful-red-rose_3075318



"Well, it *could* be a rabbit in disguise..." (but it isn't)

Credit : https://en.wikipedia.org/wiki/Wikipedia:The_duck_test

Exceptional Hierarchies

Situation : I have lots of different error modes and want to use exception types to represent each

Exceptions can be many and varied, with the type indicating some categorization of the problem however the real information is (usually) in the message/data contained within.

Pattern : For a single try block, many exception catch handlers **OR** a single catch handler for the parent class

Conclusion : Prefer to throw a base class if the catch handler is for a base class

Additional - Prefer a single exception type / catch handler if the operations are the same in multiple catch handlers

Exceptional Definition

What is an exception ?

- Behavior-based view
  - Something that's thrown
  - That gets caught (hopefully)
- Situation/Philosophy-based view
  - Rare unhappy path events
  - Non trivial errors
- Usage-based view
  - What are people doing with exceptions in real code ?
  - What should people be doing with exceptions in real code ?

Exceptionally Unwindy

Categories of stack unwinding :

- Terminally Fatal
  - Exception terminates process
- Transactionally Fatal
  - Exception resets process loop on a unit of work
- Local error handling
  - Used for erase/mitigate/recover from the exception
  - Probably a misuse of exceptions
    - However the thrower does not know return depth as it's a hidden relationship
    - Imposed by 3rd parties
  - Handle without exceptions when stack return is shallow

Truly Exceptional

Question :

Exceptions, What should they be used for ?

- Error tracing / logging, what makes a good logger ?
  - What is the error ?
  - Where in the code ?
    - std::source_location
  - Where in the stack ?
    - std::stacktrace
- Stack unwinding
  - Deep stack returns
  - Process termination, transactional reset
- Control flow
  - Encapsulate control data

** Without long repetitive class hierarchies*

Exceptional Exception Class

Exceptional Exception Class

```
template<typename DATA_T>
class OmegaException {
public:
  OmegaException(std::string str, DATA_T data, const std::source_location& loc =
     std::source_location::current(), std::stacktrace trace = std::stacktrace::current())
   :err_str_{std::move(str)}, user_data_{std::move(data)}, location_{loc}, backtrace_{trace}{}

  std::string& what() { return err_str_; }
  const std::string& what() const noexcept { return err_str_; }
  const std::source_location& where() const noexcept { return location_; }
  const std::stacktrace& stack() const noexcept { return backtrace_; }
  DATA_T& data(){ return user_data_;}
  const DATA_T& data() const noexcept { return user_data_; }

private:
  std::string err_str_;
  DATA_T user_data_;
  const std::source_location location_;
  const std::stacktrace backtrace_;
};
```

Exceptional Utility Functions

```
std::ostream& operator << (std::ostream& os, const std::source_location& location) {
  os << location.file_name() << "("
     << location.line() << ":"
     << location.column() << "), function `"
     << location.function_name() << "`";
  return os;
}

std::ostream& operator << (std::ostream& os, const std::stacktrace& backtrace) {
  for(auto iter = backtrace.begin(); iter != (backtrace.end()-3); ++iter) {
    os << iter->source_file() << "(" << iter->source_line()
    << ") : " << iter->description() << "\n";
  }
  return os;
}
```

Exception Class in Motion

Exception Class in Motion

```
enum Errs1 { bad = 1, real_bad };
enum class Errs2 { not_bad = 10, not_good };
```

```
using MyExceptionErrs1 = OmegaException<Errs1>;
using MyExceptionErrs2 = OmegaException<Errs2>;
```

```
throw MyExceptionErrs1("Bad Order id", real_bad);
```

```
catch(const MyExceptionErrs1& e) {
    std::cout << "Failed to process with code (" << e.data() << ") : "
      << e.what() << "\n" << e.where() << std::endl;
}
```

```
Failed to process with code (2) : Bad Order id
/app/example.cpp(76:69), function `Order findOrder(unsigned int)`
```

Exception Class in Motion

```
struct bucket {
  int id_;
  Msg msg_;
};
using MyExceptionBucket = OmegaException< bucket >;

throw MyExceptionBucket ("bad error", bucket{cliendId, amsg});

catch(MyExceptionBucket& eb) {
   std::cout << "Failed to process id (" << e.data().id_ << ") : "
     << e.what() << "\n" << e.stack() << std::endl;
   send_error(eb.data().msg_);
}
```

```
Failed to process id (222) : Bad Order id
example.cpp(76) : findOrder(unsigned int)
example.cpp(82) : processOrder(unsigned int)
example.cpp(97) : main
```

Conclusion

Exceptions are defined by their (Catch handler) usage :

What should they used for ?

- Error tracing (logging)
- Stack unwinding (reset / termination)
- Data passing / Control flow (when necessary)

When should they used ?

- As “Rarely” as possible
- Serious/infrequent/unexpected errors
- With as few exception types as possible (as determined by catch functionality)
  - Preferably using only the **OmegaException** class

Other Engineering talks :

Retiring The Singleton Pattern : Concrete Suggestions on What to Use Instead
Redesigning Legacy Systems : Keys to success
Managing External APIs in Enterprise Systems

Godbolt Examples

Using return codes : https://godbolt.org/z/9bs7P8WW8

Using exceptions : https://godbolt.org/z/b7qz36WaM

Bad Resource mgmt : https://godbolt.org/z/jorv9Go77

RAII Resource mgmt : https://godbolt.org/z/b7zf1ajeE

Adding context with cout : https://godbolt.org/z/vs79eYWff

Adding context with repackage : https://godbolt.org/z/4WY97bWa3

Adding context with mutable what() : https://godbolt.org/z/7q9aY7MqY

Checking with exceptions : https://godbolt.org/z/KefGoanWe

Checking with return enum : https://godbolt.org/z/f93b4cebd

Retrying with exceptions : https://godbolt.org/z/va3EzKfKT

Retrying without exceptions : https://godbolt.org/z/bfnhTjabj

Search failure with exceptions : https://godbolt.org/z/7chjP3K1f

Search failure with Optional monadics : https://godbolt.org/z/rdfY1dEca

Exception data passing : https://godbolt.org/z/7bbxqqzMG

Expected data passing : https://godbolt.org/z/GTzKx34PG

Expected templated data passing : https://godbolt.org/z/8rcYhG6Tn

Exceptional control flow, same operation : https://godbolt.org/z/c63zqsj3x

Expected Control flow, generic lambda : https://godbolt.org/z/c63zqsj3x

Exceptional control flow, different operations : https://godbolt.org/z/Pr7cEcao9

Expected Control flow, different operations : https://godbolt.org/z/veE64f5vf

OmegaException class : https://godbolt.org/z/YsWEoczsW

Questions ?

Contact : pmuldoon1@Bloomberg.net

We are Hiring

https://careers.bloomberg.com/job/search