Exceptions

Ben Langmead
ben.langmead@gmail.com
www.langmead-lab.org

Source markdown available at github.com/BenLangmead/c-cpp-notes

1
We use exceptions to indicate a *fatal* error has occurred, where there is *no reasonable way to continue from the point of the error* (the *throw point*)

It might be possible to continue from *somewhere else*, but not from the throw point
Behold, a bad program:

```cpp
#include <iostream>

using std::cout; using std::endl;

int main() {
    size_t mem = 1;
    while(true) {
        char *lots_of_mem = new char[mem];
        delete[] lots_of_mem;
        mem *= 2;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```
Exceptions

$ g++ -c exceptions1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exceptions1 exceptions1.o
$ ./exceptions1
terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
Exceptions

Keeps allocating bigger arrays until an allocation fails

The exception makes sense:

- Any pointer returned by `new[]` would be unusable; program doesn’t necessarily expect that
- Program can signal that it *does* expect that by *catching* the appropriate exception
  - Since we *don’t* do so here, the exception crashes the program
char *lots_of_mem = new char[mem];

Why not have new[] return NULL on failure, like malloc?

- When call stack is deep: f1() -> f2() -> f3() -> ... propagating errors backward requires much coordination
- If any function fails to propagate error back, chain is broken
- Error encoding must be managed (e.g. 1 = success, 2 = out of memory, ...); no standard

Exceptions are more concise, more flexible, less error prone than manually propagating errors through the chain of callers
When an exception is thrown, a `std::exception` object is created. Exception types ultimately derive from `std::exception` base class. Exception’s type and contents (accessed via `.what()`) describe what went wrong.
Exceptions

Looking in documentation for new/new T[n], you can see the exception thrown is of type bad_alloc

Allocate storage space for array
Default allocation functions (array form).

(1) throwing allocation
Allocates size bytes of storage, suitably aligned to represent any object of that size, and returns a non-null pointer to the first byte of this block.
On failure, it throws a bad_alloc exception.
The default definition allocates memory by calling operator new::operator new(size).
If replaced, both operator new and operator new[] shall return pointers with identical properties.

(2) nothrow allocation
Same as above (1), except that on failure it returns a null pointer instead of throwing an exception.

www.cplusplus.com/reference/new/operator%20new/
Exceptions

Standard exceptions:

```cpp
#include <stdexcept>

exception
  bad_alloc
  logic_error
  runtime_error
    range_error
    overflow_error
    underflow_error
  bad_cast
  length_error
  domain_error
  out_of_range
  invalid_argument
```
```cpp
#include <iostream>
#include <new> // bad_alloc defined here

using std::cout; using std::endl;

int main() {
    size_t mem = 1;
    char *lots_of_mem;
    try {
        while(true) {
            lots_of_mem = new char[mem];
            delete[] lots_of_mem;
            mem *= 2;
        }
    } catch(const std::bad_alloc& ex) {
        cout << "Got a bad_alloc!" << endl
            << ex.what() << endl;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```
$ g++ -c exceptions2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exceptions2 exceptions2.o
$ ./exceptions2
Got a bad_alloc!
std::bad_alloc
Forever is a long time
Another example:

```cpp
#include <iostream>
#include <vector>
#include <stdexcept> // standard exception classes defined

using std::cout; using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    try {
        cout << vec.at(3) << endl;
    } catch(const std::out_of_range& e) {
        cout << "Exception: " << endl << e.what() << endl;
    }
    return 0;
}
```
$ g++ -c exceptions3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exceptions3 exceptions3.o
$ ./exceptions3
Exception:
vector::_M_range_check: __n (which is 3) >= this->size() (which is 3)
try marks block of code where an exception might be thrown

try {
    while(true) {
        lots_of_mem = new char[mem]; // !
        delete[] lots_of_mem;
        mem *= 2;
    }
}

Tells C++ “exceptions might be thrown, and I’m ready to handle some or all of them”
Exceptions

catch block, immediately after try block, says what to do in the event of a particular exception

```cpp
catch(const bad_alloc& ex) {
    cout << "Yep, got a bad_alloc" << endl;
}
```

 Doesn’t strictly have to be a const reference; could be normal reference. But we rarely modify the exception object.
Exceptions

When exception is thrown, we don’t proceed to the next statement
Instead we follow a process of “unwinding”
Unwinding: keep moving “up” to wider enclosing scopes; stop at try block with relevant catch clause

```cpp
if(a == b) {
  try {
    while(c < 10) {
      try {
        if(d % 3 == 1) {
          throw std::runtime_error("!");
        }
      }
      catch(const bad_alloc &e) {
        ...
      }
    }
    catch(const bad_alloc &e) {
      ...
    }
    catch(const runtime_error &e) {
      // after throw, control moves here
      ...
    }
  }
}
```
If we unwind all the way to the point where our scope is an entire function, we jump back to the caller and continue the unwinding
void fun2() { // (called by fun1)
    while(...) {
        try {
            // unwinding from here...
            throw std::runtime_error("whoa");
        } catch(const bad_alloc& e) {
            // only catches bad_alloc, not runtime_error
            ...
        }
    }
}

void fun1() {
    try {
        fun2();
        fun2();
    } catch(const runtime_error& e) {
        // ends up here...
        ...
    }
}
If exception is never caught – i.e. we unwind all the way through main – exception info is printed to console & program exits.

That’s what happened in the original `bad_alloc` example.
Exceptions

```cpp
#include <iostream>

using std::cout; using std::endl;

int main() {
    size_t mem = 1;
    while(true) {
        char *lots_of_mem = new char[mem];
        delete[] lots_of_mem;
        mem *= 2;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}

$ g++ -o exceptions1 exceptions1.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./exceptions1
terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
```
#include <iostream>
#include <stdexcept>

using std::cout; using std::endl;

void fun2() {
    cout << "fun2: top" << endl;
    throw std::runtime_error("runtime_error in fun2");
    cout << "fun2: bottom" << endl;
}

void fun1() {
    cout << "fun1: top" << endl;
    fun2();
    cout << "fun1: bottom" << endl;
}

int main() {
    try {
        cout << "main: try top" << endl;
        fun1();
        cout << "main: try bottom" << endl;
    } catch(const std::runtime_error &error) {
        cout << "Exception handled in main: " << error.what() << endl;
    }
    cout << "main: bottom" << endl;
    return 0;
}
$ g++ -c except_unwind.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o except_unwind except_unwind.o
$ ./except_unwind

main: try top
fun1: top
fun2: top
Exception handled in main: runtime_error in fun2
main: bottom
Unwinding causes local variables to go out of scope

Destructor is called when object goes out of scope, regardless of whether exit is due to return, break, continue, exception, ...
// Prints messages upon construction and destruction
// For investigating when constructors/destructors are called
class HelloGoodbye {
public:
    HelloGoodbye(const std::string& nm) : name(nm) {
        std::cout << name << " : hello" << std::endl;
    }

    ~HelloGoodbye() {
        std::cout << name << " : goodbye" << std::endl;
    }

private:
    std::string name;
};
```
#include <iostream>
#include <stdexcept>
#include "hello_goodbye.h"

using std::cout; using std::endl;

void fun2() {
    HelloGoodbye fun2_top("fun2_top");
    throw std::runtime_error("runtime_error in fun2");
    HelloGoodbye fun2_bottom("fun2_bottom");
}

void fun1() {
    HelloGoodbye fun1_top("fun1_top");
    fun2();
    HelloGoodbye fun1_bottom("fun1_bottom");
}

int main() {
    try {
        HelloGoodbye main_top("main_top");
        fun1();
        HelloGoodbye main_bottom("main_bottom");
    } catch(const std::runtime_error &error) {
        cout << "Exception handled in main: " << error.what() << endl;
    }
    return 0;
}
```
$ g++ -c except_unwind2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o except_unwind2 except_unwind2.o
$ ./except_unwind2
main_top: hello
fun1_top: hello
fun2_top: hello
fun2_top: goodbye
fun1_top: goodbye
main_top: goodbye
Exception handled in main: runtime_error in fun2
C++ passes control to first catch block whose type equals or is a base class of the thrown exception.

Arrange catch blocks from most to least specific type.

- E.g. `catch(const std::runtime_error& e)` before `catch(const std::exception& e)`
```cpp
#include <iostream>
#include <stdexcept>

using std::cout; using std::endl;

int main() {
    try {
        throw std::out_of_range("not a runtime_error");
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        cout << "runtime_error: " << e.what() << endl;
    } catch(const std::exception& e) {
        // out_of_range is derived from exception
        // but *not* from runtime_error
        cout << "exception: " << e.what() << endl;
    }
    return 0;
}
```
Exceptions

```
$ g++ -c exc_spec.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exc_spec exc_spec.o
$ ./exc_spec
exception: not a runtime_error

Less specific catch(const std::exception& e) block is used
```
You can define your own exception class, derived from exception.

Since exceptions are related through inheritance, you can choose whether to catch a base class (thereby catching more different things) or a derived class.

Following card-game example demonstrates both points.
#ifndef CARD_GAME_H
#define CARD_GAME_H

#include <iostream>
#include <sstream>
#include <stdexcept>
#include <vector>
#include <utility>
#include <string>
#include <algorithm>

enum class Suit { HEART, DIAMOND, SPADE, CLUB };

enum class Rank { ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING };

// Card = suit + rank
typedef std::pair<Suit, Rank> Card;
class BadCardError : public std::runtime_error {
public:
    BadCardError(Card c) :
        std::runtime_error("bad card"), card(c) { }
private:
    Card card;
};
```cpp
class CardGame {
public:
    CardGame() : deck(), discard_pile() {
        for (int s = (int)Suit::HEART; s <= (int)Suit::CLUB; s++) {
            for (int r = (int)Rank::ACE; r <= (int)Rank::KING; r++) {
                deck.push_back(std::make_pair((Suit)s, (Rank)r));
            }
        }
        std::random_shuffle(deck.begin(), deck.end());
    }
    Card draw(); // user takes card from deck
    void discard(Card c); // user puts card on discard pile

    size_t deck_size() const { return deck.size(); }
private:
    std::vector<Card> deck, discard_pile;
};
```

#include "card_game.h"
```
#include "card_game.h"

Card CardGame::draw() {
    Card c = deck.back();
    deck.pop_back();
    return c;
}

void CardGame::discard(Card c) {
    // sanity check the card first
    if (c.first < Suit::HEART || c.first > Suit::CLUB ||
        c.second < Rank::ACE || c.second > Rank::KING)
    {
        throw BadCardError(c);
    }
    discard_pile.push_back(c);
}

#include "card_game.h"

using std::cout; using std::endl;

int main() {
    CardGame cg;
    Card c = cg.draw();
    try {
        cg.discard(c);
        cout << "no exception" << endl;
    }
    catch(const std::runtime_error& e) {
        cout << "runtime_error: " << e.what() << endl;
    }
    return 0;
}
$ g++ -o card_game_main1 card_game_main1.cpp card_game.cpp
$ ./card_game_main1
no exception
#include "card_game.h"

using std::cout; using std::endl;

int main() {
    CardGame cg;
    Card c = cg.draw();
    try {
        c.first = (Suit)5; // Card is now malformed!
        cg.discard(c);
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        cout << "runtime_error: " << e.what() << endl;
    }
    return 0;
}
$ g++ -o card_game_main2 card_game_main2.cpp card_game.cpp
$ ./card_game_main2
runtime_error: bad card

Our catch block caught the exception
```cpp
#include "card_game.h"

using std::cout; using std::endl;

int main() {
    CardGame cg;
    Card c = cg.draw();
    try {
        c.first = (Suit)5;
        cg.discard(c);
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        // first catch block that either equals or is a base class of the thrown exception is the one used
        cout << "runtime_error: " << e.what() << endl;
    } catch(const std::exception& e) {
        cout << "exception: " << e.what() << endl;
    }
    return 0;
}
```
$ g++ -o card_game_main3 card_game_main3.cpp card_game.cpp
$ ./card_game_main3
runtime_error: bad card