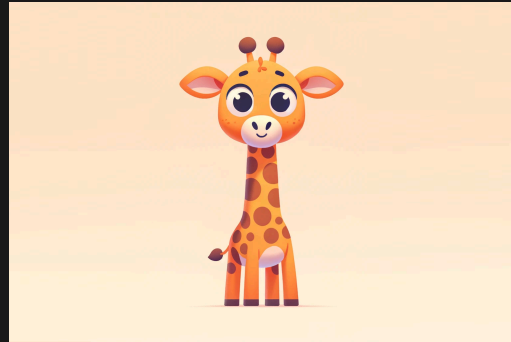


# COMP6771

## Exceptions

### Lecture 5.1

Author(s): Hayden Smith



[\(Download as PDF\)](#)

# In This Lecture

- **Why?** 🤔
  - Sometimes our programs need to deal with unexpected runtime errors and handle them gracefully
- **What?** 📄
  - Exception object
  - Throwing and catching exceptions
  - Rethrowing
  - `noexcept`



# Explore An Example

What does this produce?

```
1 #include <iostream>
2 #include <vector>
3
4 auto main() -> int
5 {
6     std::cout << "Enter -1 to quit\n";
7     std::vector<int> items { 97, 84, 72, 65 };
8     std::cout << "Enter an index: ";
9     for (int print_index; std::cin >> print_index;) {
10         if (print_index == -1)
11             break;
12         std::cout << items.at(static_cast<unsigned int>(print_index)) << '\n';
13         std::cout << "Enter an index: ";
14     }
15 }
```

exception1.cpp



# Explore An Example

Now we use exceptions instead

```
1 #include <iostream>
2 #include <vector>
3
4 auto main() -> int
5 {
6     std::cout << "Enter -1 to quit\n";
7     std::vector<int> items { 97, 84, 72, 65 };
8     std::cout << "Enter an index: ";
9     for (int print_index; std::cin >> print_index;) {
10         if (print_index == -1)
11             break;
12         try {
13             std::cout << items.at(static_cast<unsigned int>(print_index)) << '\n';
14             items.resize(items.size() + 10);
15         } catch (const std::out_of_range& e) {
16             std::cout << "Index out of bounds\n";
17         } catch (...) {
18             std::cout << "Something else happened";
19         }
20         std::cout << "Enter an index: ";
21     }
22 }
```

exception2.cpp



# Explore An Example

Let's take a step back and unpack what we just saw...

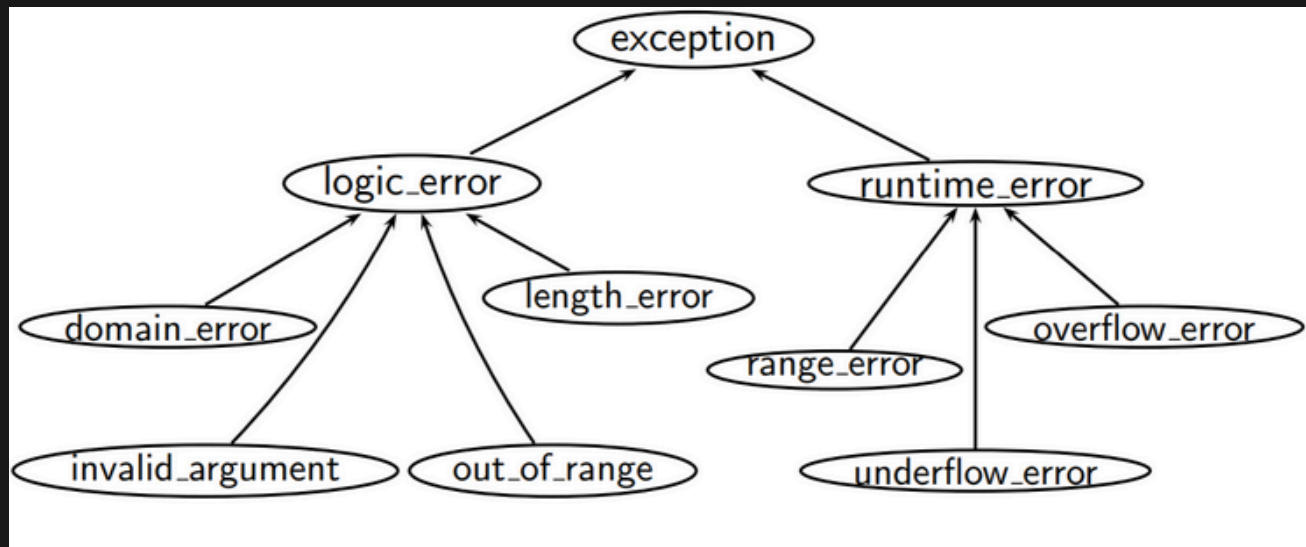
# Exceptions: What & Why?

- What:
  - Exceptions: Are for exceptional circumstances
    - Happen during run-time anomalies (things not going to plan A!)
  - Exception handling:
    - Run-time mechanism
    - C++ detects a run-time error and raises an appropriate exception
    - Another unrelated part of code catches the exception, handles it, and potentially rethrows it
  - Why:
    - Allows us to gracefully and programmatically deal with anomalies, as opposed to our program crashing.



# What Are "Exception Objects"

- Any type we derive from `std::exception`
  - `throw std::out_of_range("Exception!");`
  - `throw std::bad_alloc("Exception!");`
- Why `std::exception`? Why classes?
- `#include <exception>` for `std::exception` object
- `#include <stdexcept>` for objects that inherit `std::exception`



- <https://en.cppreference.com/w/cpp/error/exception>
- <https://stackoverflow.com/questions/25163105/stdexcept-vs-exception-headers-in-c>



# Conceptual Structure

- Exceptions are treated like lvalues
- Limited type conversions exist (pay attention to them):
  - nonconst to const
  - other conversions we will not cover in the course

```
1 try {
2     // Code that may throw an exception
3 } catch (/* exception type */) {
4     // Do something with the exception
5 } catch (...) { // any exception
6     // Do something with the exception
7 }
```

- [https://en.cppreference.com/w/cpp/language/try\\_catch](https://en.cppreference.com/w/cpp/language/try_catch)

# Multiple Catch Options

This does not mean multiple catches will happen, but rather that multiple options are possible for a single catch

```
1 #include <iostream>
2 #include <vector>
3
4 auto main() -> int
5 {
6     auto items = std::vector<int> {};
7     try {
8         items.resize(items.max_size() + 1);
9     } catch (std::bad_alloc& e) {
10         std::cout << "Out of bounds.\n";
11     } catch (std::exception&) {
12         std::cout << "General exception.\n";
13     }
14 }
```

multiple.cpp



# Rethrow

- When an exception is caught, by default the catch will be the only part of the code to use/action the exception
- What if other catches (lower in the precedence order) want to do something with the thrown exception?

```
1 try {
2     try {
3         try {
4             throw T{};
5         } catch (T& e1) {
6             std::cout << "Caught\n";
7             throw;
8         }
9     } catch (T& e2) {
10        std::cout << "Caught too!\n";
11        throw;
12    }
13 } catch (...) {
14    std::cout << "Caught too!!\n";
15 }
```



# Catching The Right Way

- Throw by value, catch by const reference
- Ways to catch exceptions:
  - By value (no!)
  - By pointer (no!)
  - By reference (yes)
- References are preferred because:
  - more efficient, less copying (exploring today)
  - no slicing problem (related to polymorphism, exploring later)

# Exploring Catch By Value

```
1 #include <iostream>
2
3 class Giraffe {
4 public:
5     Giraffe() { std::cout << "Giraffe constructed" << '\n'; }
6     Giraffe(const Giraffe& g) { (void) g; std::cout << "Giraffe copy-constructed" << '\n'; }
7     ~Giraffe() { std::cout << "Giraffe destructed" << '\n'; }
8 };
9
10 void zebra()
11 {
12     throw Giraffe {};
13 }
14
15 void llama()
16 {
17     try {
18         zebra();
19     } catch (Giraffe g) {
20         (void) g;
21         std::cout << "caught in llama; rethrow" << '\n';
22         throw;
23     }
24 }
25
26 int main()
27 {
28     try {
29         llama();
30     } catch (Giraffe g) {
31         (void) g;
32         std::cout << "caught in main" << '\n';
33     }
34 }
```

by-value.cpp

# Exploring Catch By Value

```
1 #include <iostream>
2
3 class Giraffe {
4 public:
5     Giraffe() { std::cout << "Giraffe constructed" << '\n'; }
6     Giraffe(const Giraffe& g) { (void) g; std::cout << "Giraffe copy-constructed" << '\n'; }
7     ~Giraffe() { std::cout << "Giraffe destructed" << '\n'; }
8 };
9
10 void zebra()
11 {
12     throw Giraffe {};
13 }
14
15 void llama()
16 {
17     try {
18         zebra();
19     } catch (Giraffe g) {
20         (void) g;
21         std::cout << "caught in llama; rethrow" << '\n';
22         throw;
23     }
24 }
25
26 int main()
27 {
28     try {
29         llama();
30     } catch (Giraffe g) {
31         (void) g;
32         std::cout << "caught in main" << '\n';
33     }
34 }
```

by-value.cpp



# Exploring Catch By Reference

```
1 #include <iostream>
2
3 class Giraffe {
4 public:
5     Giraffe() { std::cout << "Giraffe constructed" << '\n'; }
6     Giraffe(const Giraffe& g) { (void) g; std::cout << "Giraffe copy-constructed" << '\n'; }
7     ~Giraffe() { std::cout << "Giraffe destructed" << '\n'; }
8 };
9
10 void zebra()
11 {
12     throw Giraffe {};
13 }
14
15 void llama()
16 {
17     try {
18         zebra();
19     } catch (const Giraffe& g) {
20         (void) g;
21         std::cout << "caught in llama; rethrow" << '\n';
22         throw;
23     }
24 }
25
26 int main()
27 {
28     try {
29         llama();
30     } catch (const Giraffe& g) {
31         (void) g;
32         std::cout << "caught in main" << '\n';
33     }
34 }
```

by-ref.cpp



# Exploring Catch By Reference

```
1 #include <iostream>
2
3 class Giraffe {
4 public:
5     Giraffe() { std::cout << "Giraffe constructed" << '\n'; }
6     Giraffe(const Giraffe& g) { (void) g; std::cout << "Giraffe copy-constructed" << '\n'; }
7     ~Giraffe() { std::cout << "Giraffe destructed" << '\n'; }
8 };
9
10 void zebra()
11 {
12     throw Giraffe {};
13 }
14
15 void llama()
16 {
17     try {
18         zebra();
19     } catch (const Giraffe& g) {
20         (void) g;
21         std::cout << "caught in llama; rethrow" << '\n';
22         throw;
23     }
24 }
25
26 int main()
27 {
28     try {
29         llama();
30     } catch (const Giraffe& g) {
31         (void) g;
32         std::cout << "caught in main" << '\n';
33     }
34 }
```

by-ref.cpp



# Exception Safety Levels

- This part is not specific to C++
- Operations performed have various levels of safety
  - No-throw (failure transparency)
  - Strong exception safety (commit-or-rollback)
  - Weak exception safety (no-leak)
  - No exception safety



# No-Throw Guarantee

- Also known as failure transparency
- Operations are guaranteed to succeed, even in exceptional circumstances
  - Exceptions may occur, but are handled internally
- No exceptions are visible to the client
- This is the same, for all intents and purposes, as `noexcept` in C++
- Examples:
  - Closing a file
  - Freeing memory
  - Anything done in constructors or moves (usually)
  - Creating a trivial object on the stack (made up of only ints)



# No-Throw Guarantee

## The noexcept specifier

- Specifies whether a function could potentially throw
- **It doesn't not actually prevent a function from throwing an exception**
- [https://en.cppreference.com/w/cpp/language/noexcept\\_spec](https://en.cppreference.com/w/cpp/language/noexcept_spec)
- STL functions can operate more efficiently on noexcept functions

```
1 class S {
2     public:
3     int foo() const; // may throw
4 }
5
6 class S {
7     public:
8     int foo() const noexcept; // does not throw
9 }
```



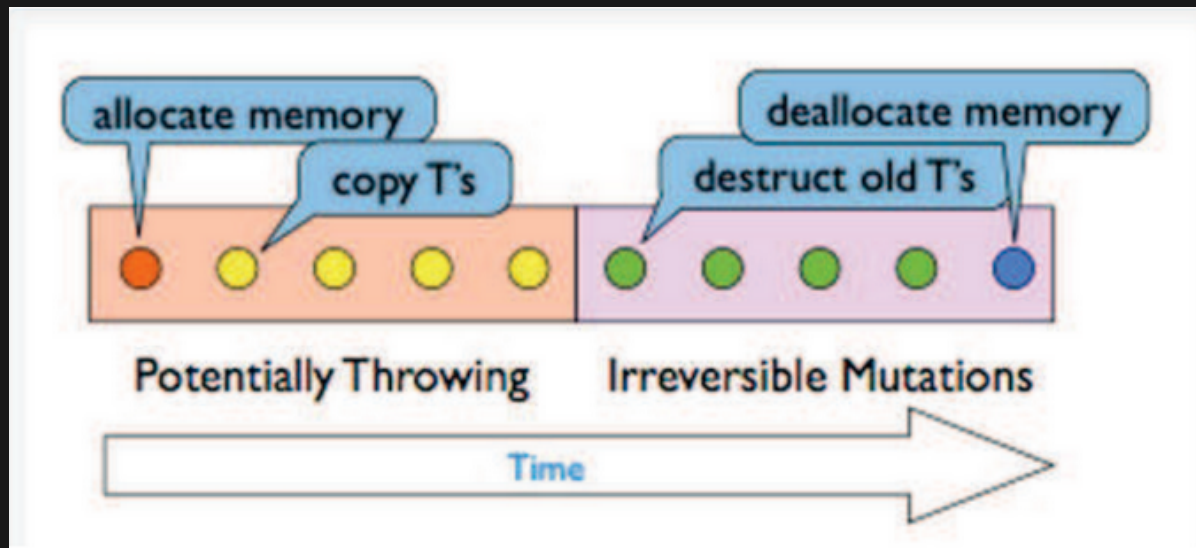
# Strong Exception Safety

- Also known as "commit or rollback" semantics
- Operations can fail, but failed operations are guaranteed to have no visible effects
- Probably the most common level of exception safety for types in C++
- All your copy-constructors should generally follow these semantics
- Similar for copy-assignment
  - Copy-and-swap idiom (usually) follows these semantics (why?)
  - Can be difficult when manually writing copy-assignment



# Strong Exception Safety

- To achieve strong exception safety, you need to:
  - First perform any operations that may throw, but don't do anything irreversible
  - Then perform any operations that are irreversible, but don't throw





# Basic Exception Safety

- This is known as the no-leak guarantee
- Partial execution of failed operations can cause side effects, but:
  - All invariants must be preserved
  - No resources are leaked
- Any stored data will contain valid values, even if it was different now from before the exception
  - Does this sound familiar? A "valid, but unspecified state"
  - Move constructors that are not noexcept follow these semantics



# No Exception Safety

- No guarantees
- Don't write C++ with no exception safety
  - Very hard to debug when things go wrong
  - Very easy to fix - wrap your resources and attach lifetimes
    - This gives you basic exception safety for free



# Exceptions And Catch2

```
1 CHECK_NO_THROW(expr);
```

Checks expr doesn't throw an exception.

```
1 CHECK_THROWS(expr);
```

Checks expr throws an exception.

```
1 CHECK_THROWS_AS(expr, type);
```

Checks expr throws an exception.

```
1 namespace Matchers = Catch::Matchers;  
2 CHECK_THROWS_WITH(  
3   expr,  
4   Matchers::Message("message"));
```

Checks expr throws an exception with a message.

```
1 CHECK_THROWS_MATCHES(  
2   expr,  
3   type,  
4   Matchers::Message("message"));
```

CHECK\_THROWS\_AS and  
CHECK\_THROWS\_WITH in a single check.

# Feedback



Or go to the [form here](#).

