

COMP6771

Advanced C++ Programming

Week 4.2

Exceptions

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

Let's start with an example

What does this produce?

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

demo455-exception1.cpp

Let's start with an example

What does this produce?

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

Exceptions: What & Why?

- **What:**

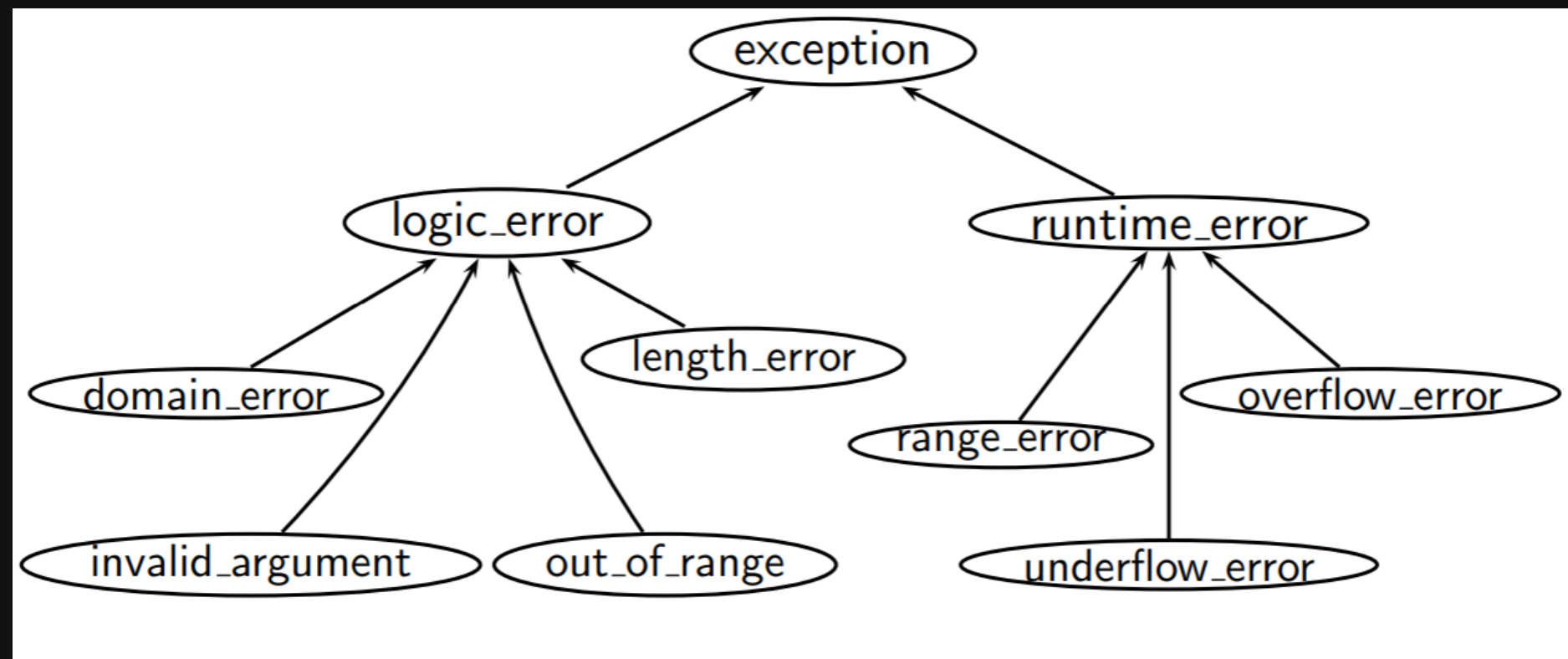
- **Exceptions:** Are for exceptional circumstances
 - Happen during run-time anomalies (things not going to plan A!)
- **Exception handling:**
 - Run-time mechanism
 - C++ detects a run-time error and raises an appropriate exception
 - Another unrelated part of code catches the exception, handles it, 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 }
```

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


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)

(Extra reading for those interested)

- <https://blog.knatten.org/2010/04/02/always-catch-exceptions-by-reference/>

Catch by value is inefficient

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

Catch by value inefficiency

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

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

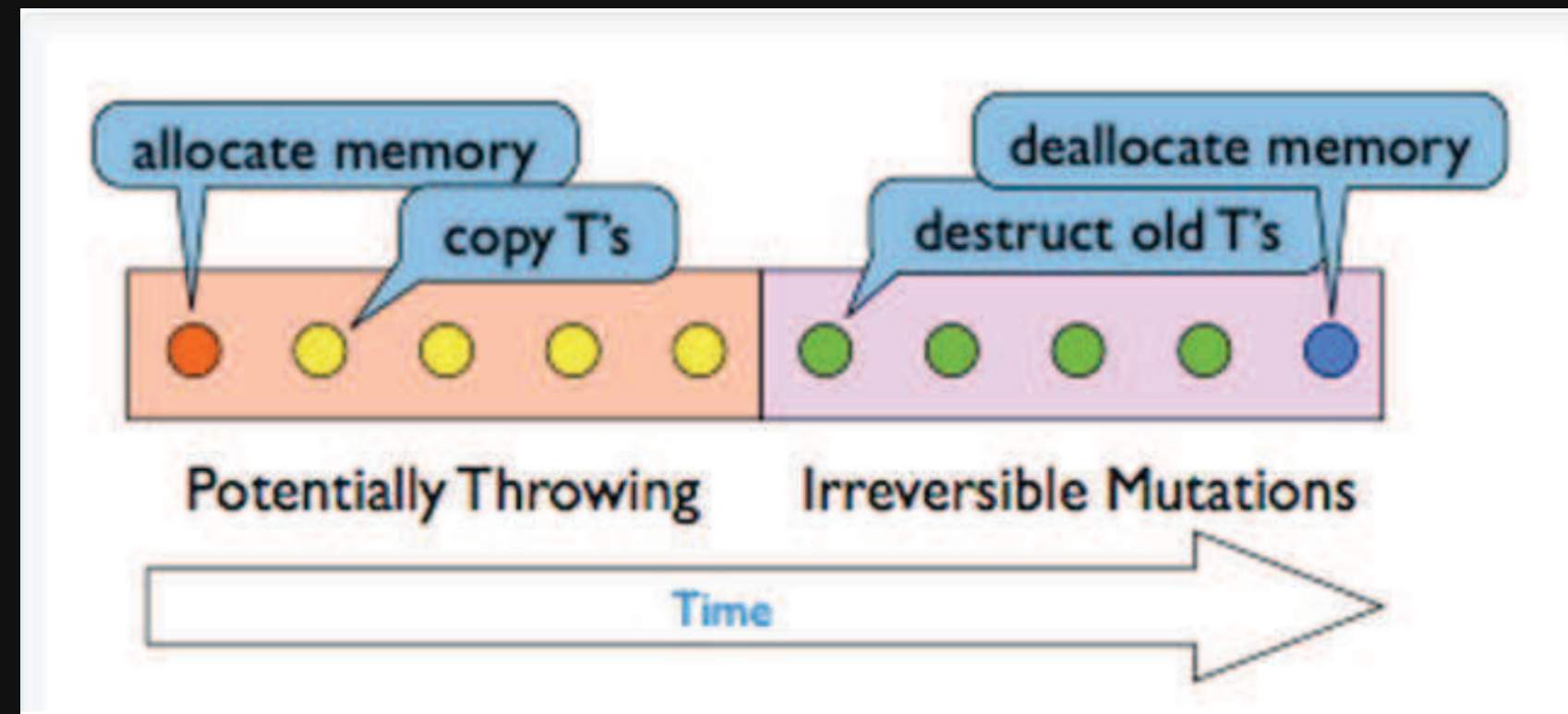
- Also known as failure transparency
- Operations are guaranteed to succeed, even in exceptional circumstances
 - Exceptions may occur, but are handled internally
- No exceptions are visible to the client
- This is the same, for all intents and purposes, as noexcept in C++
- Examples:
 - Closing a file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

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

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
- 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 }
```

Testing exceptions

`CHECK_NOTHROW(expr);`

Checks *expr* doesn't throw an exception.

`CHECK_THROWS(expr);`

Checks *expr* throws an exception.

`CHECK_THROWS_AS(expr, type);`

Checks *expr* throws *type* (or something derived from *type*).

`REQUIRES_THROWS*` also available.

Testing exceptions

```
namespace Matchers = Catch::Matchers;  
CHECK_THROWS_WITH(  
    expr,  
    Matchers::Message( "message" ) );
```

Checks *expr* throws an exception
with a message.

```
CHECK_THROWS_MATCHES(  
    expr,  
    type,  
    Matchers::Message( "message" ) );
```

CHECK_THROWS_AS and
CHECK_THROWS_WITH
in a single check.

REQUIRES_THROWS* also available.

Feedback

