EXCEPTIONS

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school

AGENDA

1. Error handling methods
2. Exceptions
3. Efficiency and cost

ERROR HANDLING METHODS

goto

```cpp
#include <iostream>

bool isValid() { /* ... */ }

int main() {
    // ...
    if(!isValid()) {
        goto error;
    }

error:
    std::cerr << "Error occured" << '\n';
    return 1;
}
```

ERROR CODES

```cpp
#include <iostream>

enum class ErrorCode { Ok, FailCause1, FailCause2 };
bool isValid() { /* ... */ }

ErrorCode foo() {
    if(!isValid()) {
        return ErrorCode::FailCause1;
    }
    // ...
    return ErrorCode::Ok;
}

int main() {
    if(foo() == ErrorCode::FailCause1) {
        std::cerr << "Error occured" << '\n';
        return 1;
    }
    return 0;
}
```

ERROR HANDLING IN CONSTRUCTORS AND OPERATORS

Constructors and operators have strictly defined return types (or no return type). It is impossible to return a custom error code from them.

```
struct FileWrapper {
    FileWrapper(std::string const& filePath)
            : m_file(fopen(filePath.c_str(), "rw")) {
        /* What if the file did not open? */
    }

    ~FileWrapper() {
        fclose(m_file);
    }

    FileWrapper & operator<<(std::string const& text) {
        /* What if the file did not open? */
        fputs(text.c_str(), m_file);
        return *this;
    }

private:
    FILE* m_file;
};
```

throw

Instead of returning a special value from a function or setting an error code we just `throw` an exception. It indicates that something went wrong and we can handle this case in another place.

```
struct FileWrapper {
    FileWrapper(std::string const& filePath)
            : m_file(fopen(filePath.c_str(), "rw")) {
        if(!m_file) {
            throw std::runtime_error("File not opened");
        }
    }

    ~FileWrapper() {
        fclose(m_file);
    }

    FileWrapper & operator<<(std::string const& text) {
        /* Not validation needed, invalid object cannot be created */
        fputs(text.c_str(), m_file);
        return *this;
    }

private:
    FILE* m_file;
};
```

try/catch

`try` block is a place where we can expect an exception. `catch` blocks tries to match the exception type.

```cpp
#include <iostream>
#include <stdexcept>

void foo() { throw std::runtime_error("Error"); }

int main() {
    try {
        foo();
    } catch(std::runtime_error const&) {
        std::cout << "std::runtime_error" << std::endl;
    } catch(std::exception const& ex) {
        std::cout << "std::exception: " << ex.what() << std::endl;
    } catch(...) {
        std::cerr << "unknown exception" << std::endl;
    }
}
```

RESULT

```
std::runtime_error
```

WHAT IS AN EXCEPTION?

Every object can work as an exception.

```
throw 42;
```

However, it's not recommended to use build-in types or any user created classes as an exception.

```
throw std::runtime_error{"Huston, we have a problem"};
```

It is recommended to use exceptions from the standard library (like `std::runtime_error`) of create own exception classes that inherits from `std::exception`.

HOW DOES IT WORK?

MATCHING EXCEPTIONS

```
struct TalkingObject {
    TalkingObject()  { cout << "Constructor" << '\n'; }
    ~TalkingObject() { cout << "Destructor" << '\n'; }
};

void foo() { throw std::runtime_error("Error"); }

int main() {
    TalkingObject outside;
    try {
        TalkingObject inside;
        foo();
    } catch(runtime_error const& ex) {
        cout << "runtime_error: " << ex.what() << '\n';
    } catch(exception const&) {
        cout << "exception" << '\n';
    }
}
```

RESULT

```
Constructor
Constructor
Destructor
runtime_error: Error
Destructor
```

STACK UNWINDING MECHANISM

- Thrown exceptions starts a stack unwinding mechanism
- The exception type is being matched with consecutive `catch` clauses
- If the exception type is matched:
  - Everything allocated on stack is destroyed in a reversed order until reaching `try` block
  - The code from matching `catch` clause is executed
  - The exception object is destroyed
- If the exception type is not matched with any `catch` clause, the stack unwinding continues to the next `try` block

UNHANDLED EXCEPTION

```
struct TalkingObject { /*...*/ };

void foo() { throw std::runtime_error("Error"); }

void bar() {
    try {
        TalkingObject inside;
        foo();
    } catch(std::logic_error const&) {
        std::cout << "std::logic_error" << '\n';
    }
}

int main() {
    TalkingObject outside;
    bar();
}
```

RESULT

Constructor

Constructor

>> abort() <<

WHY DESTRUCTORS HAVE NOT BEEN CALLED?

- The stack unwinding mechanism first check for a matching `catch` clause in a current `try` block before destroying objects
- An exception which was not caught and falls out of the main function scope calls `std::terminate()`. It kills the program.

EXCEPTION RETHROWING

```
struct TalkingObject { /*...*/ };

void foo() { throw std::runtime_error("Error"); }

void bar() try {
    TalkingObject inside;
    foo();
} catch(std::exception const&) {
    std::cout << "exception" << '\n';
    throw;
}

int main() {
    TalkingObject outside;
    try {
        bar();
    } catch(std::runtime_error const& ex) {
        std::cout << "runtime_error: " << ex.what() << '\n';
    }
}
```

Bare `throw` in a `catch` clause rethrows a current exception.

RESULT

```
Constructor

Constructor

Destructor

exception

runtime_error: Error

Destructor
```

EXCEPTION RETHROWING

- Rethrown exception starts a stack unwinding once again
- Stack unwinding continues until another `try` block is reached
- `catch` clause for a base type can catch an exception of a derived type
- It does not change the original exception type, when it is rethrown

THROWING AN EXCEPTION DURING STACK UNWINDING

```
struct TalkingObject { /*...*/ };
struct ThrowingObject {
    ThrowingObject() { std::cout << "Throwing c-tor\n"; }
    ~ThrowingObject() {
        throw std::runtime_error("error in destructor");
    }
};

void foo() { throw std::runtime_error("Error"); }

int main() {
    TalkingObject outside;
    try {
        ThrowingObject inside;
        foo();
    } catch(std::exception const&) {
        std::cout << "std::exception" << '\n';
        throw;
    }
}
```

RESULT

```
Constructor

Throwing c-tor

>> abort() <<
```

CONCLUSIONS

- Only one exception can be handled at a time
- The exception thrown during stack unwinding causes termination of the program - `std::terminate()` is called
- You should never throw an exception in a destructor

ARE EXCEPTIONS EXPENSIVE?

- My YT video with explanation (in Polish)
- Casual program flow
- Exceptional flow

EXCEPTIONS

ADVANTAGES

- Error signalling and handling can be done separately
- Code readability - functions have only required logic without handling special cases
- Errors can be handled in constructors and operators
- No extra checks on casual flow = no extra if = no cost

DISADVANTAGES

- Binary size is increased (extra error handling code is added in the end of all `noexcept` functions)
- Time of exception handling is not defined
- Usually requires real-time information to track the program flow (core dump, debugger)

CONCLUSIONS

- Time of exception handling is not defined
  - It depends on the number and types of objects allocated on stack between the place where the exception was thrown and when it was actually caught
- Do not use exceptions in real-time devices with strictly defined time of execution (eg. in healthcare systems, automotive)
- If you want to use exceptions check the usage of the program. If exceptional program flow is really rare - measure and benchmark which version is faster

RECOMMENDATIONS

- Use STL exceptions check cppreference.com
- Inherit own exceptions from STL exceptions
  - `catch(const std::exception & e)` will catch all of them
- Avoid `catch(...)` - it catches absolutely everything and usually is not a good practice
- Catch exceptions by `const &` - it prevents redundant exception copies
- Use exceptions only in unusual situations and do not build a casual program flow on exceptions
- Use `noexcept` keyword to indicate functions from which the exception will not be thrown. It helps a compiler to optimize the code and reduce a binary size.

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school