

DCC004 - Algoritmos e Estruturas de Dados II

Tratamento de exceções

Renato Martins

Email: renato.martins@dcc.ufmg.br

<https://www.dcc.ufmg.br/~renato.martins/courses/DCC004>

Material adaptado de PDS2 - Douglas Macharet e Flávio Figueiredo

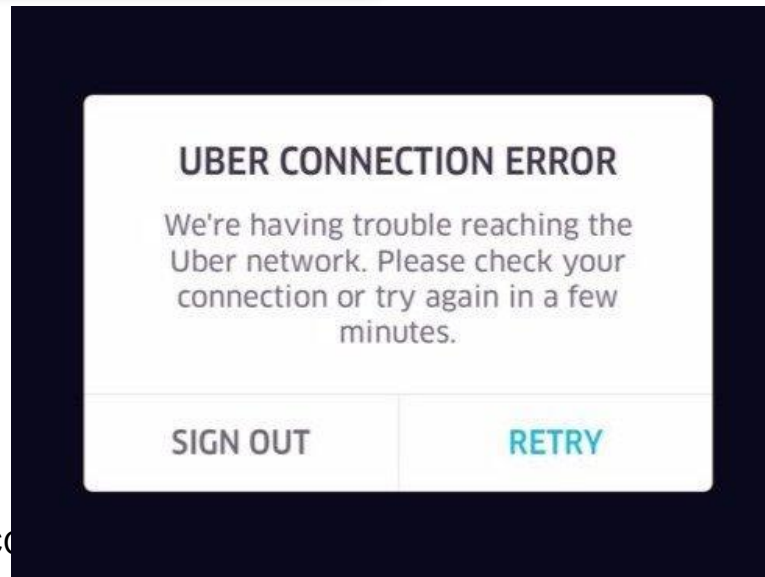
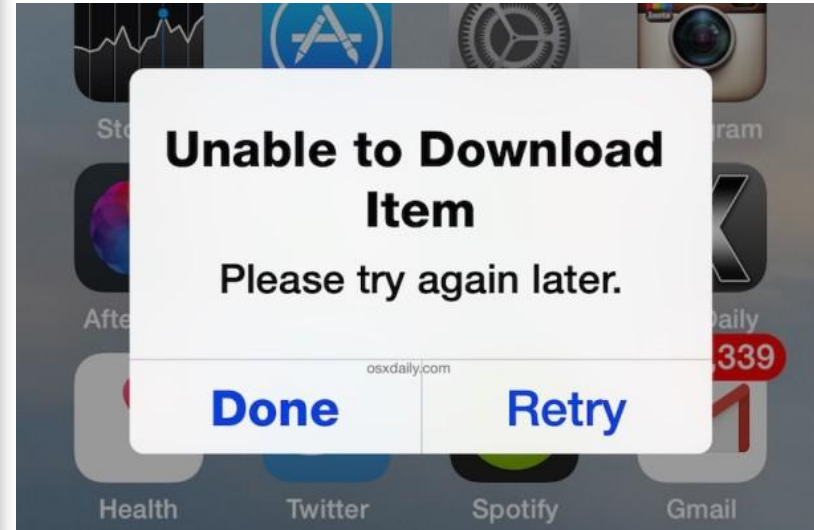
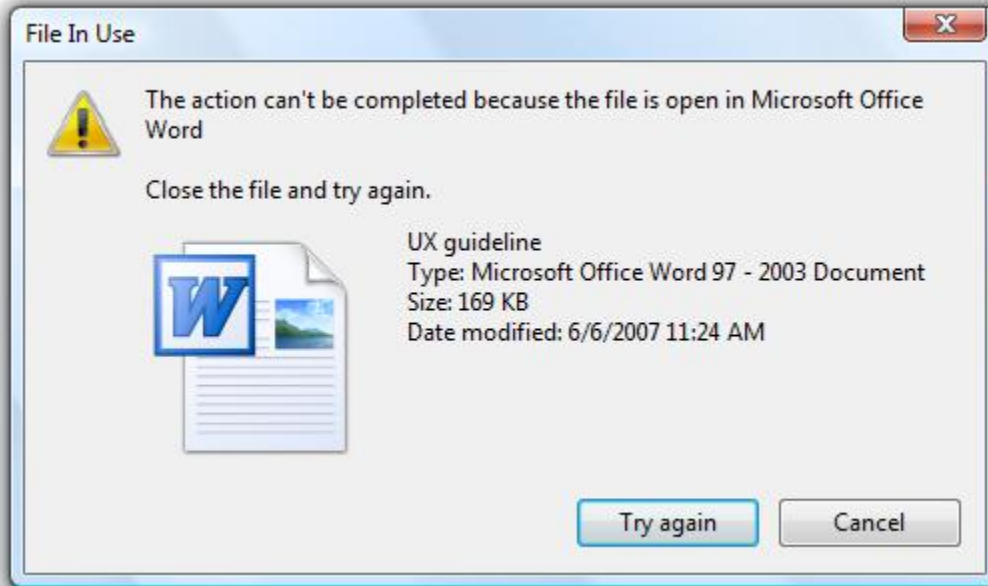
Exceções

- Indicam condições anormais/especiais
- Erros de leitura de arquivos
- Erros de conexão de rede
- Parâmetros inválidos
- Tentativas de acessos inválidos
- . . .

Exceções

- Como sabemos, a maioria dos erros não podem ser detectados em tempo de compilação
- Alguns erros são bugs no programa
- Exceções:
 - Nem um nem outro
 - Condição anormal que precisa ser tratada

Exceções

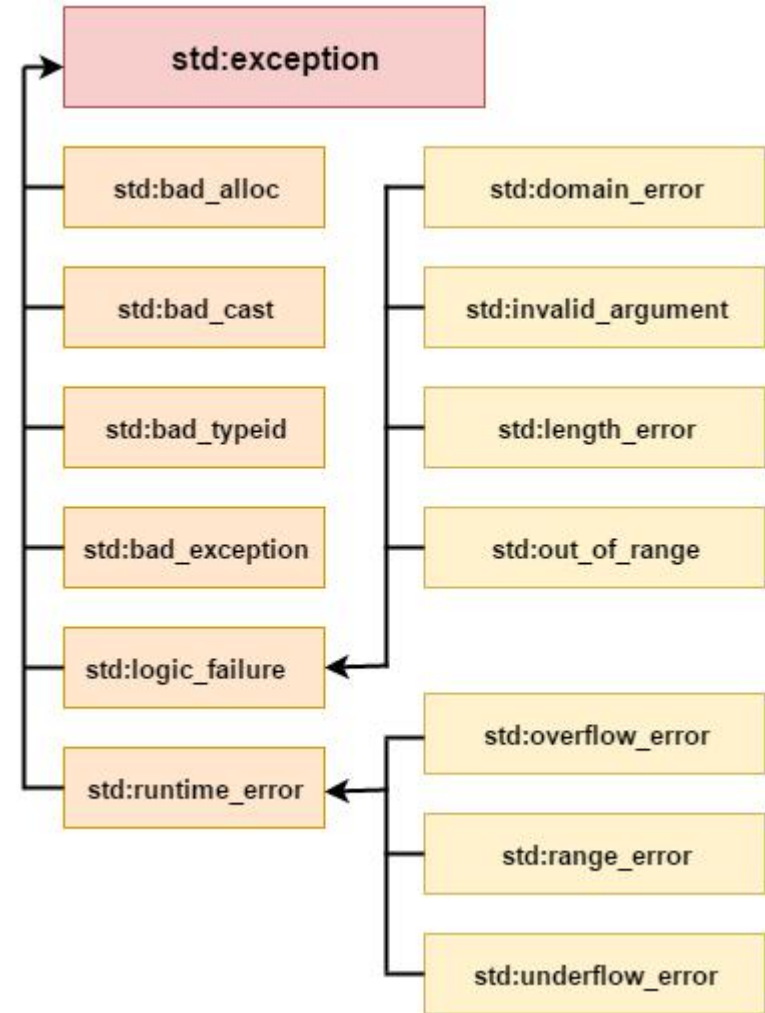


Motivação

- Gerar programas mais robustos
- Permitem ao código/usuário agir
 - Re-conectar
 - Escolher outro arquivo
 - Outro parâmetro
- Simples de usar
- Alguém tem que **tratar** a exceção

Tipos Comuns de Exceções

- C++ já tem exceções comuns na biblioteca padrão
- Podemos definir novos tipos para erros específicos do programa



Exemplo

Qual o problema com o código abaixo?

```
#include <string>
#include <iostream>

int main() {
    std::string texto;
    std::cin >> texto;
    texto.substr(10);
    return 0;
}
```

Exemplo

Ao executar o código com uma entrada com menos do que 10 caracteres:

```
libc++abi.dylib: terminating with  
uncaught exception of type  
std::out_of_range: basic_string  
Abort trap: 6
```


Tratando Exceções

- Exceções podem ser tratadas
- Ou lançadas para frente
- Para tratar: fazemos uso de **try/catch**
- Para lançar: fazemos uso de **throw**
- Existem casos onde uma função/método não sabe tratar um erro.
Repassa o mesmo
- Em algum momento chegamos no main

Exemplo com Métodos

```
#include <string>
#include <iostream>

std::string pega_sub_string(std::string str, int k) {
    return str.substr(k);
}

std::string le_entrada() {
    std::string texto;
    std::cin >> texto;
    return pega_sub_string(texto, 10);
}

int main() {
    std::cout << le_entrada();
    return 0;
}
```

Tratando Exceções


Usamos o **try/catch**

```
std::string le_entrada() {  
    std::string texto;  
    try {  
        std::cin >> texto;  
        return pega_sub_string(texto, 10);  
    } catch (std::out_of_range &e) {  
        std::cerr << "Entrada invalida!" << std::endl;  
        return "";  
    }  
}
```

Tratando Exceções

Usamos o **try/catch**

```
std::string le_entrada() {  
    std::string texto;  
    try {  
        std::cin >> texto;  
        return pega_sub_string(texto, 10);  
    } catch (std::out_of_range &e) {  
        std::cerr << "Entrada invalida!" << std::endl;  
        return "";  
    }  
}
```



Método que causa a exception

Tratando Exceções

Neste caso, é um bom tratamento?

```
std::string le_entrada() {  
    std::string texto;  
    try {  
        std::cin >> texto;  
        return pega_sub_string(texto, 10);  
    } catch (std::out_of_range &e) {  
        std::cerr << "Entrada invalida!" << std::endl;  
        return "";  
    }  
}
```

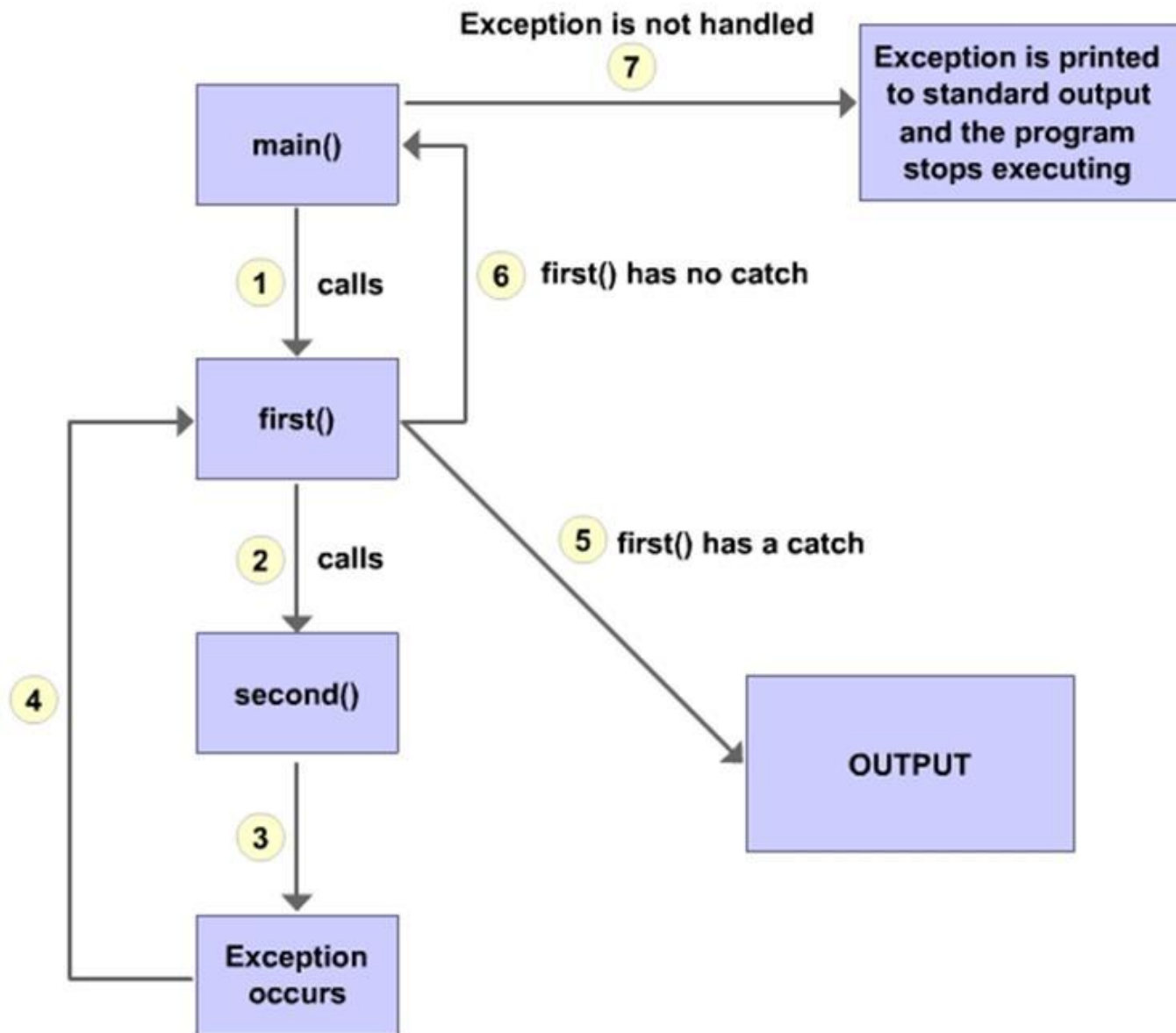
Método que causa a exception

Tratamento

Tratando Exceções

- Idealmente teremos uma ação a ser seguida. Caso contrário, é melhor repassar o erro para frente
- Ao não realizar o catch, a exceção continua sendo lançada na pilha de chamadas

Stack Unwind



Tratamento Melhor

- Temos uma ação
- Continuar no laço até a entrada ser ok!

```
std::string le_entrada() {
    std::string texto;
    while (1) {
        try {
            std::cin >> texto;
            return pega_sub_string(texto, 10);
        } catch (std::out_of_range &e) {
            std::cerr << "Entrada invalida! Digite novamente.\n";
        }
    }
}
```


Lançando

Existem situações que nosso código deve lançar uma exceção. Usamos **throw**

```
#include <stdexcept>

int fatorial(int n) {
    if (n < 0) {
        throw std::invalid_argument("Não existe fatorial de n < 0");
    }
    if (n <= 1) {
        return 1;
    }
    return n * fatorial(n-1);
}
```

Lançando

- Escolha uma exceção de acordo com o erro. Podemos lançar mais de uma
- Por exemplo, a maioria dos computadores não vai computar o fatorial de $n \geq 20$ corretamente.
- Overflow: -2102132736
- Como sinalizar para o usuário?

Lançando duas Exceções

- Escolher a exceção correta para o caso

```
int fatorial(int n) {  
    if (n < 0) {  
        throw std::invalid_argument("Não existe fatorial de n < 0");  
    }  
    if (n >= 20) {  
        throw std::overflow_error("Não consigo computar para n>=20");  
    }  
    if (n <= 1) {  
        return 1;  
    }  
    return n * fatorial(n-1);  
}
```

Exemplo Multicatch

- `e.what()` imprime o erro
- Qual o problema do código abaixo?

```
int main() {  
    try {  
        std::cout << fatorial(-2);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    }  
}
```

Exemplo Multicatch

- `e.what()` imprime o erro
- Qual o problema do código abaixo?
- Não tratamos o caso a seguir

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    }  
}
```

Exemplo Multicatch

e.what() imprime o erro

Qual o problema do código abaixo?

Resolvendo

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    } catch (std::overflow_error &e) {  
        std::cout << e.what();  
    }  
}
```

Exemplo Multicatch

e.what() imprime o erro

Qual o problema do código abaixo?

Qual o problema agora?

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::invalid_argument &e) {  
        std::cout << e.what();  
    } catch (std::overflow_error &e) {  
        std::cout << e.what();  
    }  
}
```

Hierarquia de Exceções

- A definição de qual bloco **catch** vai ser executado depende de dois fatores:
 - 1) Tipo
 - 2) Ordem
- Assim:
 - Logo que o tipo casar com um dos blocos **catch** vamos entrar no bloco
 - Podemos explorar herança

Pegando Exceções Genéricas

- Agora o código funciona com a exceção genérica: **exception**

```
int main() {  
    try {  
        std::cout << fatorial(20);  
    } catch (std::exception &e) {  
        std::cout << e.what();  
    }  
}
```

Ajudando o usuário do método

- Podemos usar **noexcept** para
 - Definir que uma função nunca lança
- Ou podemos usar **noexcept(false)**
 - Deixando claro que a função pode lançar
- Ou **throw**
 - Indica o tipo que pode ser lançado

```
void f() noexcept; // the function f() does not throw
void f() noexcept(false); // g may throw
void f() throw(std::invalid_argument); // lança aquele tipo
```

Definindo Exceções

- Em C++ podemos lançar qualquer coisa para frente
- Idealmente, lançaremos uma sub-classe da classe **std::exception**
 - Deixando claro que é um erro
- Porém podemos fazer:
 - **throw "ocorreu um erro";**

Definindo Exceções

- Sugiro usar herança na classe `exception`
- Lembrando, podemos fazer `catch` ou na super-classe ou na sub-classe

```
class ContaSemSaldoException : public std::exception {  
    // . . . codigo aqui  
};
```

```
class ContaSemSaldoException : public std::invalid_argument {  
    // . . . codigo aqui  
};
```

Definindo Exceções

- Podemos sobrescrever os métodos da classe base. São **virtual**
- No exemplo abaixo definimos o nosso **what**, podemos usar qualquer

```
class ContaSemSaldoException : public std::exception {  
public:  
    virtual const char* what() const noexcept override;  
};
```

```
const char* ContaSemSaldoException::what() const noexcept {  
    return "Conta sem saldo!";  
}
```

Uso da Exceção

```
class Conta {  
private:  
    int __agencia;  
    int __numero;  
    double __saldo = 0;  
    bool possui_saldo(double valor) {  
        return (__saldo - valor) > 0;  
    }  
public:  
    void sacar(double valor) throw(ContaSemSaldoException) {  
        if (!possui_saldo(valor)) {  
            throw ContaSemSaldoException();  
        }  
        this->__saldo -= valor;  
    }  
};
```

Indicamos que vai
lançar

Lançamos nossa exceção