Exceções

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP
Baseado nos slides do Prof. Marco Antônio Carvalho

Introdução

- Uma **exceção** é uma indicação de um **problema que ocorre** durante a **execução** de um programa
- O próprio nome indica que o problema é infrequente;
- A regra é que o programa execute corretamente.

Introdução (cont.)

Introdução (cont.)

- O **tratamento de exceções permite** que os **programas** sejam **mais robustos** e também **tolerantes a falhas**
- Os **erros** de execução **são processados**;
- O programa **trata o problema** e **continua executando** como se nada tivesse acontecido;
- Ou pelo menos, termina sua execução elegantemente.

Introdução (cont.)

- Considere o pseudocódigo:

```
Realize uma tarefa
Se a tarefa precedente não executou corretamente
    Realize processamento de erro
Realize a próxima tarefa
Se a tarefa precedente não executou corretamente
    Realize processamento de erro
```

Introdução (cont.)

- Considere o pseudocódigo:

```
Realize uma tarefa
Se a tarefa precedente não executou corretamente
    Realize processamento de erro
Realize a próxima tarefa
Se a tarefa precedente não executou corretamente
    Realize processamento de erro
```

Alerta

Mistura de lógica e tratamento de erro pode tornar o programa difícil de ler/depurar

Introdução (cont.)

- Tratamento de exceção **remove correção de erro da "linha principal"** do programa
  - Torna o **programa mais claro** e melhora a manutenção
  - Programadores podem decidir se **tratam todas** as excepções **ou algumas** de um tipo específico
  - Objetos de classes específicas tratam os erros: possibilidade do uso de **herança e polimorfismo**

Introdução (cont.)

- Só **pode tratar erros síncronos**:
  - Aqueles que seguem a "linha de execução" do programa
  - Exs.: divisão por zero, ponteiro nulo

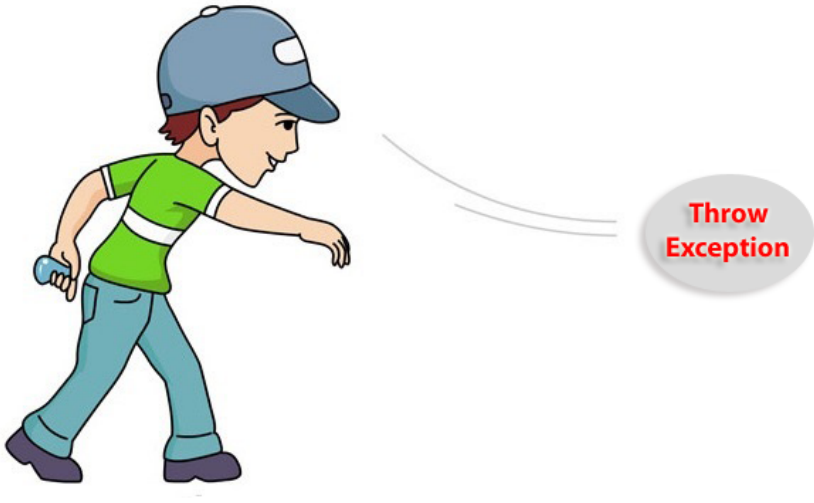
Introdução (cont.)

- **Não pode** tratar **erros assíncronos** (independente do programa)
  - Ex.: I/O de disco, mouse, teclado, mensagens de rede que ocorrem em paralelo e de maneira independente do fluxo de controle do programa em execução
- Erros mais fáceis de tratar

Tratamento de Exceção

- Terminologia
  - Função que tem erros dispara uma exceção (*throws an exception*)
  - Tratamento de exceção (se existir) pode lidar com problema
    - Pega (*catches*) e trata (*handles*) a exceção
  - Se não houver tratamento de exceção, exceção não é pega
    - Pode terminar o programa (*uncaught*)

Tratamento de Exceção (cont.)



Tratamento de Exceção (cont.)

- Envolve três conceitos:
  - Utilização de um bloco tentar (*try*)
  - Captura da uma exceção por um manipulador de exceções (*catch*)
  - Disparo de uma exceção (*throw*)

Tratamento de Exceção (cont.)

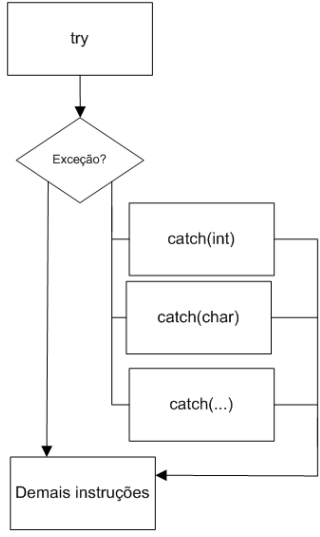
Tratamento de Exceção (cont.)

- Código C++

```
try {
    código que pode provocar uma exceção
}
catch (exceptionType){
    código para tratar a exceção
}
```

- Bloco *try* possui código que pode provocar exceção
- **Um ou mais blocos** *catch* devem ser escritos imediatamente após o bloco *try* correspondente

Tratamento de Exceção (cont.)



Tratamento de Exceção

Realizar a divisão de dois números digitados.

Tratamento de Exceção (cont.)

```
int main()
{
    int a, b;
    double c;

    cin >> a >> b;

    try{
        if (b == 0)
            throw "Divisao por zero \n";
        c = static_cast<double>(a) / b;
        cout << "Resposta: " << c << endl;
    }
    catch (const char* e){
        cerr << "Erro: " << e;
    }
    return 0;
}
```

Tratamento de Exceção (cont.)

Criar uma função que encontre o fatorial de um número. Gerar uma exceção quando o numero fornecido como parâmetro, é negativo

Tratamento de Exceção (cont.)

```
int fatorial(int n)
{
    if (n < 0)
        throw "Numero negativo! \n";
    int f = 1;
    for (int i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

Tratamento de Exceção (cont.)

```
int main()
{
   try{
        cout << "5! = " << fatorial(5) << endl;
        cout << "-5! = " << fatorial(-5) << endl;
    }
    catch (const char* e){
        cerr << "Erro: " << e;
    }
    return 0;
}
```

Tratamento de Exceção (cont.)

```
5! = 120
Erro: Numero negativo!
```

Exceção usando classes

Criar a classe que trate a exceção da divisão por zero, identificando a linha onde acontece o erro.

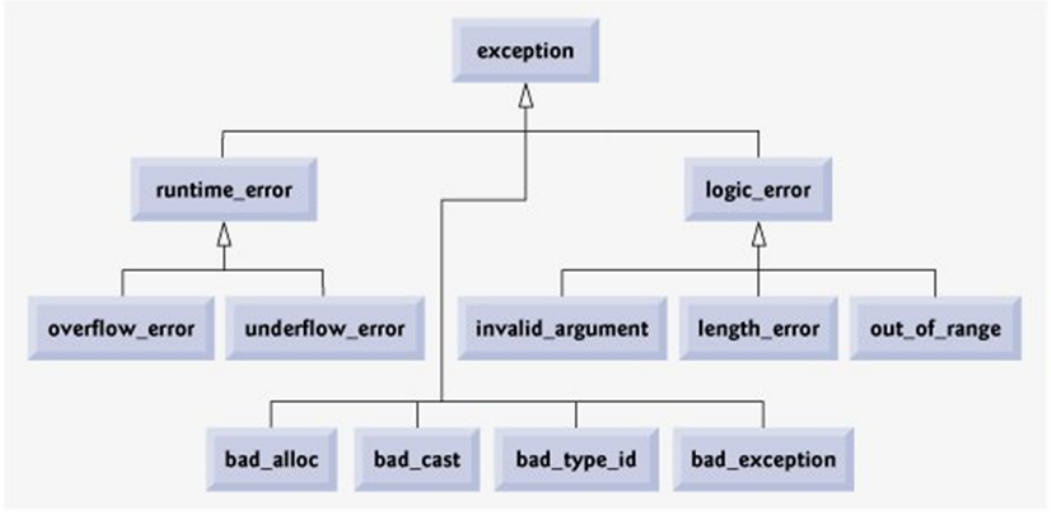
Exceção usando classes (cont.)

```
class DivisionByZero{
    string msg;
    int line;
public:
    DivisionByZero(const string& msg, int line) :
        msg(msg), line(line) {}
    string what() const {
        return msg + " na linha " + to_string(line);
    }
};
```

Exceção usando classes (cont.)

```
int main()
{
    int a, b;
    try {
        cin >> a >> b;
        if (b == 0)
            throw DivisionByZero("Divisao por zero", __LINE__);
        cout << static_cast<double>(a) / b;
    }
    catch (DivisionByZero& e){
        cerr << "Erro: " << e.what();
    }
    return 0;
}
```

Classes de Exceções da Biblioteca padrão



Classes de Exceções da Biblioteca padrão (cont.)

- Exceções
  - *bad_alloc:* falha de alocação
  - *bad_cast:* falha de conversão
  - *bad_typeid:* falha de verificação de tipo
  - *bad_exception:* falha de exceção

Classes de Exceções da Biblioteca padrão (cont.)

- Erros lógicos
  - *invalid_argument:* argumento inválido
  - *length_error:* dimensão errada
  - *out_of_range:* fora do intervalo

Classes de Exceções da Biblioteca padrão (cont.)

- Erros de *runtime*
  - *overflow_error:* número muito grande
  - *underflow_error:* número muito pequeno

Exemplo

Modificar o código de divisão por zero, usando uma excecão do sistema.

Exemplo (cont.)

```
#include <iostream>
#include <stdexcept>
class DivideByZeroException : public runtime_error{
public:
    DivideByZeroException(const string& msg) :
            runtime_error(msg){}
};

double divisao(int num, int den){
    if (den == 0)
        throw DivideByZeroException("Divisao por zero! \n");
    return static_cast<double>(num) / den;
}
```

Exemplo (cont.)

```
int main(){
    int num, den;
    double result;
    cout << "Digite dois numeros inteiros (ctrl+z para finalizar)\n";
    while (cin >> num >> den){
        try{
            result = divisao(num, den);
            cout << "A resposta: " << result << endl;
        }
        catch (DivideByZeroException& e){
            cout << "Error: " << e.what() << endl;
        }
        cout << "Digite dois numeros inteiros (ctrl+z para finalizar)\n
            ";
        }
        return 0;
}
```

Exemplo 2

Criar um vetor usando a classe *vector* e evitar o acesso a uma posição inválida

Exemplo 2 (cont.)

public member function

std::vector::at

<vector>

```
      reference at (size_type n);
const_reference at (size_type n) const;
```

Access element

Returns a reference to the element at position ***n*** in the vector.

The function automatically checks whether ***n*** is within the bounds of valid elements in the vector, throwing an out_of_range exception if it is not (i.e., if ***n*** is greater than, or equal to, its size). This is in contrast with member operator[], that does not check against bounds.

Exemplo 2 (cont.)

```
#include <iostream>
#include <stdexcept>
#include <vector>
int main(){
    vector<int> v(4);
    int i, valor;
    try{
        cin >> i >> valor;
        v.at(i) = valor;
    }
    catch (out_of_range &e){
        cout << "Error: " << e.what();
    }
    catch (...){
        cout << "Erro inesperado";
    }
    return 0;
}
```

Processando falhas *new*

Criar um vetor de ponteiros de dimensão 50 e alocar dinâmicamente para cada ponteiro um vetor de 50 milhões de posições

Processando falhas *new* (cont.)

operator new

<new>

C++98 C++11

```
throwing (1)   void* operator new (std::size_t size);
nothrow (2)    void* operator new (std::size_t size, const std::nothrow_t& nothrow_value) noexcept;
placement (3)  void* operator new (std::size_t size, void* ptr) noexcept;
```

Allocate storage space

Default ***allocation functions*** (single-object 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.

(2) nothrow allocation

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

C++98 C++11

The default definition allocates memory by calling the the first version: `::operator new (size)`.

If replaced, both the first and second versions shall return pointers with identical properties.

Processando falhas *new* (cont.)

```
int main(){
    double *ptr[50];
    int i;

    for (i = 0; i < 50; i++){
        ptr[i] = new (nothrow) double[50000000];
        if (ptr[i] == nullptr){
            cerr << "Memory allocation fail! \n";
            break;
        }
        else
            cout << "Allocated 50 millon doubles\n";
    }
```

Processando falhas *new* (cont.)

```
    cout << "\ndeleting...";
    for (int j = 0; j < i; j++){
        if (ptr[i] != nullptr)
            delete[] ptr[j];
    }
    cout << "\ndeleted";
    return 0;
}
```

Processando falhas *new* (cont.)

```
Allocated 50 millon doubles
Allocated 50 millon doubles
Allocated 50 millon doubles
Memory allocation fail!

deleting...
deleted
```

Processando falhas *new* (cont.)

```
#include<stdexcep>
int main(){
    double *ptr[50];
    int i;
    try{
        for (i = 0; i < 50; i++){
            ptr[i] = new double[50000000];
            cout << "Allocated 50000000 doubles\n";
        }
    }
    catch(bad_alloc& e){
        cerr << "Error: " << e.what();
    }
    for (int j = i-1; j >= 0; j--){
        if (ptr[i] != nullptr)
            delete[] ptr[i];
    }
    return 0;
}
```

Processando falhas *new* (cont.)

```
Allocated 50 millons doubles
Allocated 50 millons doubles
Allocated 50 millons doubles
Allocated 50 millons doubles
Error: std::bad_alloc
deleting...
deleted
```

Exemplo

Inserir vários pares de números e dividir os mesmos, lançar uma exceção se o divisor é zero ou os valores digitados são inválidos

Exemplo (cont.)

```
#include <iostream>
#include <stdexcept>
#include <limits>
using namespace std;

int main()
{
    int num, den;
    double res;
    cout << "\nDigite dois numeros: ";
    while (true){
        try{
            if (  !(cin >> num >> den) ){
                throw invalid_argument("\nInserir somente inteiros");
            }
            if ( den == 0 ){
                throw runtime_error("\ndivisao por zero");
            }
```

Exemplo (cont.)

```
            res = static_cast<double>(num) / den;
            cout << "\nresposta: " << res;
        }
        catch(invalid_argument& e){
            cerr << "\nErro: " << e.what();
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
        catch(runtime_error& e){
            cerr << "\nErro: " << e.what();
        }
        cout << "\nDigite dois numeros (ctrl+c para finalizar): " ;
    }
    return 0;
}
```

Exemplo 2

Alterar o tamanho de um vetor definido usando a classe *vector*

```
#include <iostream>          // std::cerr
#include <stdexcept>         // std::length_error
#include <vector>            // std::vector
using namespace std;
int main (void) {
  try {
    // vector throws a length_error if resized above max_size
    vector<int> myvector;
    myvector.resize(myvector.max_size()+1);
  }
  catch (const length_error& le) {
          std::cerr << "Length error: " << le.what() << '\n';
  }
  return 0;
}
```

Exceção *bad type id*

```
// exception example
#include <iostream>        // std::cerr
#include <typeinfo>        // operator typeid
#include <exception>       // std::exception

class Polymorphic {virtual void member(){}};

int main () {
  try{
    Polymorphic * pb = 0;
    typeid(*pb);  // lanca a excecao bad_typeid
  }
  catch (std::exception& e){
    std::cerr << "exception caught: " << e.what() << '\n';
  }
  return 0;
}
```

Exceção *bad cast*

```
#include <stdexcept>
#include <iostream>
class Base {virtual void fun() {}};
class Derivada : public Base {};
int main() {
    try {
        Base a;
        Derivada b = dynamic_cast<Derivada&>(a);
    } catch (std::bad_cast& e) {
        std::cerr << e.what() << '\n';
    }
}
```

Conta

```
class ExcecaoSaldoInsuficiente : public runtime_error{
public:
    ExcecaoSaldoInsuficiente(const string msg): runtime_error(msg){}
};

class Conta
{
public:
    Conta(int=0,int=0,double=0.0);
    void sacar(double);
    double getSaldo() const;
private:
    int agencia;
    int numero;
    double saldo;
    bool possuiSaldoSuficiente(double);
};
```

Conta (cont.)

```
Conta::Conta(int agencia,int numero, double saldo) :agencia(agencia),
    numero(numero), saldo(saldo) {}

bool Conta::possuiSaldoSuficiente(double valor){
    return (saldo - valor) >= 0;
}

void Conta::sacar(double valor){
    if (!possuiSaldoSuficiente(valor))
        throw ExcecaoSaldoInsuficiente("Saldo insuficiente");
    saldo -= valor;
}

double Conta::getSaldo() const{
    return saldo;
}
```

Conta (cont.)

```
int main(){
    try{
        Conta c;
        c.sacar(1000);
    } catch(ExcecaoSaldoInsuficiente& e){
        cerr << "Erro: " << e.what();
    }
    return 0;
}
```

```
Erro: Saldo insuficiente
```

FIM