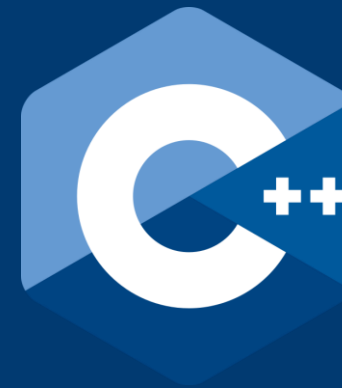


Aula 09:



# Tratamento de Exceções

(try - catch - throw)

ECOP13A - Programação Orientada a Objetos

Prof. André Bernardi  
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá



# Tratamento de Exceções



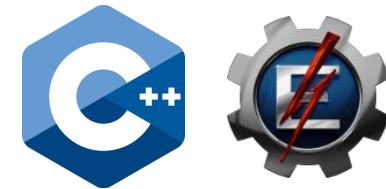
Exceção

ex.ce.ção

***sf (lat. *exceptione*)*** **1** Ato ou efeito de excetuar.  
**2** Desvio de regra, de lei, de princípio ou de ordem. **3** A coisa excetuada; aquilo que se desvia da regra.

# Exceção == Erro?

Quase isso.



No universo de linguagens de programação, uma **exceção** é uma **indicação de um problema** que ocorre durante a **execução** de um programa. O nome “**exceção**” indica que o problema ocorre com **pouca frequência**: a regra é que uma instrução seja corretamente executada, e a exceção à regra é um problema.

Aprenderemos nesta aula a tratar estas exceções. Um problema mais grave poderia impedir que um programa prosseguisse sua execução.

Conhecemos técnicas que fazem com que, ao se deparar com um problema, um programa o resolva (*handle*) da melhor maneira e tente prosseguir sua execução.

Programas serão mais **robustos** e **tolerantes a falhas**.



A lógica do programa testa, constantemente, as condições que determinam como a execução prossegue. Normalmente, até agora, fazemos o seguinte:

- *Execute a primeira tarefa;*
- *Se a tarefa anterior não foi executada corretamente*
  - *Realizar processamento de erro*
- *Executa próxima tarefa*
- *Se a tarefa anterior não foi executada corretamente*
  - *Realizar processamento de erro*
- *E assim por diante...*

Embora este tipo de abordagem funcione, misturar lógica do programa (**azul**) com lógica de tratamento (vermelho) de erro pode resultar em um código confuso, difícil de ser lido e modificado – especialmente para grandes aplicações.



O **tratamento de exceções** (*exception handling*) permite que você remova o código de tratamento de erro da “linha principal” de execução do seu programa, aumentando a inteligibilidade do código e facilitando sua modificação.

Além disso, tenha em mente o seguinte:

Se problemas potenciais ocorrem com pouca frequência (ocasiões incomuns, erros de usuário, etc.) misturar a lógica do programa com o tratamento de erros atrapalha a eficiência do código, uma vez que uma série de testes será realizada TODAS as vezes, mesmo quando nada de ruim acontece.

Você pode decidir tratar quaisquer exceções que escolher: todas as exceções, todas de um mesmo tipo, apenas um tipo. Essa flexibilidade reduz a probabilidade de se negligenciar algum erro.

O tratamento de exceções em C++ deve ser realizado desde o início do projeto. Vejamos um exemplo: **como trataríamos uma tentativa de divisão por zero.**



Em C++, a **divisão por zero** utilizando aritmética de inteiros normalmente faz com que um programa termine abruptamente. Neste exemplo, tentaremos evitar este problema. Para tal, iremos:

- Criar um tipo de exceção

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

// detecta tentativas de divisão por zero durante a execução
class DivideByZeroException : public runtime_error
{
    public:
        // construtor especifica mensagem de erro padrão
        DivideByZeroException()
            : runtime_error("Tentativa de divisao por zero.") {}
};
```



Mas como **utilizamos** a exceção? Como **sabemos** se ela ocorreu? Vamos utilizá-la em um programa de teste.



Criaremos uma função chamada **quociente**, que recebe um numerador e um denominador e retorna um valor inteiro com o valor inteiro da divisão dos dois. Nela, indicaremos a existência de um problema caso o denominador seja igual a 0. Veja:

```
// retorna divisão de num por den (inteiro)
int quociente(int num, int den)
{
    return num/den;
}
```

sem exceção: **vulnerável**.

```
// retorna divisão de num por den (inteiro)
int quociente(int num, int den)
{
    if(den == 0) // dispare uma exceção (ela será tratada em algum lugar)
        throw DivideByZeroException(); // termina a função por aqui.
    return num/den;
}
```

com exceção: **preparado para tratamento**.



```
int main()
{
    int n = 0, d = 0;
    cout << "Entre com um numerador e um denominador: ";
    cin >> n >> d;
    cout << "Resultado: " << quociente(n,d) << "\n";
}
```

Sem tratamento: programa fecha abruptamente.

```
int main()
{
    int n = 0, d = 0;
    cout << "Entre com um numerador e um denominador: ";
    cin >> n >> d;

    try
    {
        cout << "Resultado: " << quociente(n,d) << "\n";
    }
    catch(DivideByZeroException &ex)
    {
        cout << "Excecao capturada: " << ex.what() << "\n";
    }
}
```

Com tratamento: programa continua sua execução – sob controle.

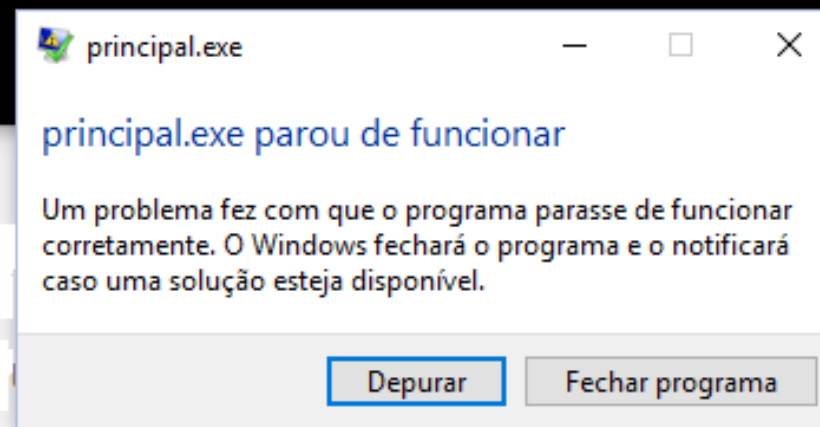


Veja o resultado de uma execução para o programa cujo tratamento da exceção `DivideByZeroException` não é realizado, quando inserimos um denominador nulo:



```
Entre com um numerador e um denominador: 10 0
terminate called after throwing an instance of 'DivideByZeroException'
  what(): Tentativa de divisao por zero.

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```



Agora, o mesmo programa com captura e tratamento de exceção:

```
Entre com um numerador e um denominador: 10 0
Excecao capturada: Tentativa de divisao por zero.
```

way better.



- A classe **DivideByZeroException** é uma classe derivada da classe **runtime\_error**, definida em `<stdexcept>`. A classe **runtime\_error**, por sua vez, herda da classe **exception**, da biblioteca padrão, definida em `<exception>`.
- A classe **exception** é a classe base padrão de C++ para TODAS as exceções. Há uma hierarquia de classes, que veremos adiante.
- Uma classe de exceção típica derivada de **runtime\_error** define apenas um construtor que passa uma string contendo uma mensagem de erro para o construtor de **runtime\_error**.
- Todas as classes que derivam (direta ou indiretamente) de **exception** contém a função virtual **what()**, que retorna a mensagem de erro de um objeto de exceção (string).
  - Não é necessário derivar uma classe de exceção personalizada como a nossa de uma classe padrão, mas isso é esperado e vantajoso.



Uma função que não “sabe lidar” com um problema, pode **disparar uma exceção** (***throw***), na esperança de que seu chamador saiba lidar com o mesmo.

A função **quociente()** apenas identifica o problema, testando se o denominador é igual a zero. Quem toma providências é seu chamador, ou seja, a função `main()`.

Veja novamente:

```
// retorna divisão de num por den (inteiro)
int quociente(int num, int den)
{
    if(den == 0) // dispare uma exceção (ela será tratada em algum lugar)
        throw DivideByZeroException(); // termina a função por aqui.
    return num/den;
}
```

Um componente chamador indica os tipos de exceções que está disposto a tratar, especificando seus tipos em cláusulas **catch** de blocos **try**. Um bloco **try** consiste na palavra reservada **try**, seguida de chaves (obrigatório) que envolvem um trecho de código (de qualquer tamanho) em que as exceções podem ocorrer. Ele delimita instruções que poderiam causar exceções e TAMBÉM, instruções que precisam ser puladas caso haja uma exceção. Veja no exemplo:

```
try
{
    cout << "Resultado: " << quociente(n,d) << "\n";
    // instruções daqui pra baixo somente serão executadas caso
    // uma exceção DivideByZeroException não seja capturada antes.
    cout << "Seu calculo ocorreu normalmente - sem excecao.\n";
}
catch(DivideByZeroException &ex)
{
    cout << "Excecao capturada: " << ex.what() << "\n";
}
```

Uma exceção no bloco **try**, imediatamente redireciona o fluxo de execução para dentro da cláusula **catch** correspondente à exceção capturada.

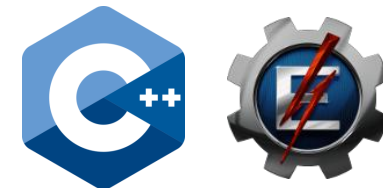
As exceções são, portanto, processadas nos blocos **catch**, que capturam e tratam exceções. Ao menos **um** bloco **catch** deve vir após um bloco **try**, mas podem existir **vários**, um para cada tipo de exceção que poderia ocorrer dentro do bloco **try**.

Cada bloco **catch** começa com a palavra reservada **catch** e especifica entre parênteses o tipo da exceção que será tratada: **um parâmetro de exceção**. Se um parâmetro de exceção incluir um nome (não é obrigatório), este nome (**ex**, no exemplo), o tratador do **catch** pode utilizar este nome para interagir com o objeto da exceção.

No exemplo, chamamos a função **what()** de **ex** (que é uma referência) para escrever na tela a mensagem de erro. Veja:

```
catch(DivideByZeroException &ex)
{
    cout << "Excecao capturada: " << ex.what() << "\n";
}
```

# Resumindo...



Se houver uma exceção como resultado de uma instrução dentro de um bloco **try**, este termina imediatamente. Em seguida, o programa procura o primeiro tratador de **catch** que possa processar o tipo de exceção que ocorreu, comparando (em ordem) o tipo da exceção disparada com o tipo do parâmetro de exceção de cada **catch** até encontrar uma correspondência.

Uma correspondência ocorre quando os **tipos forem idênticos** ou se o tipo da exceção disparada for de uma **classe derivada** do tipo do parâmetro de exceção. Classes mais genéricas devem ser tratadas por último.

Após tratada a exceção, o programa não retoma a execução do ponto onde parou, mas segue sua execução a partir da primeira instrução após o último **catch** do bloco **try**.



## Veja um exemplo:



```
try
{
    cout << "Resultado: " << quociente(n,d) << "\n"; // aqui pode ocorrer exceção
    cout << "Seu calculo ocorreu normalmente - sem excecao.\n";
}
catch(runtime_error &ex)
{
    cout << "runtime_error: " << ex.what() << "\n";
}
catch(DivideByZeroException &ex)
{
    // nunca será executada, uma vez que DivideByZeroException
    // é uma classe derivada de runtime_error, e as comparações são feitas
    // na ordem. runtime_error será sempre a primeira correspondência.
    cout << "DivideByZeroException: " << ex.what() << "\n";
}

// execução retomada a partir daqui
cout << "Primeira instrucao apos ultimo catch!\n";
```

O próprio compilador gera um **aviso** durante a compilação:  
“DivideByZeroException *will be caught by earlier handler* for “std::runtime\_error”.

```
Entre com um numerador e um denominador: 10 0
runtime_error: Tentativa de divisao por zero.
Primeira instrucao apos ultimo catch!
```

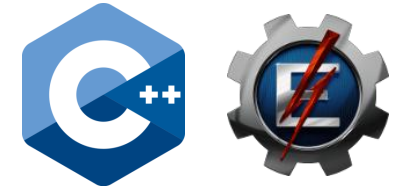
# Quando devemos utilizar o tratamento de exceção?



Esta ferramenta foi projetada para processar erros síncronos, que ocorrem quando a instrução é executada. Exemplos:

- Subscritos de array fora do intervalo;
- Overflow aritmético (número fora de intervalo, estouro);
- Divisão por zero;
- Parâmetro de função inválidos;
- Alocação de memória sem sucesso (devido á falta de memória), etc.

O tratamento de exceção não é projetado para processar **erros assíncronos**, ou seja, erros que ocorram em paralelo e sejam independentes do fluxo do programa como: chegada de mensagem na rede, clique de mouse, toques de tecla, etc.).



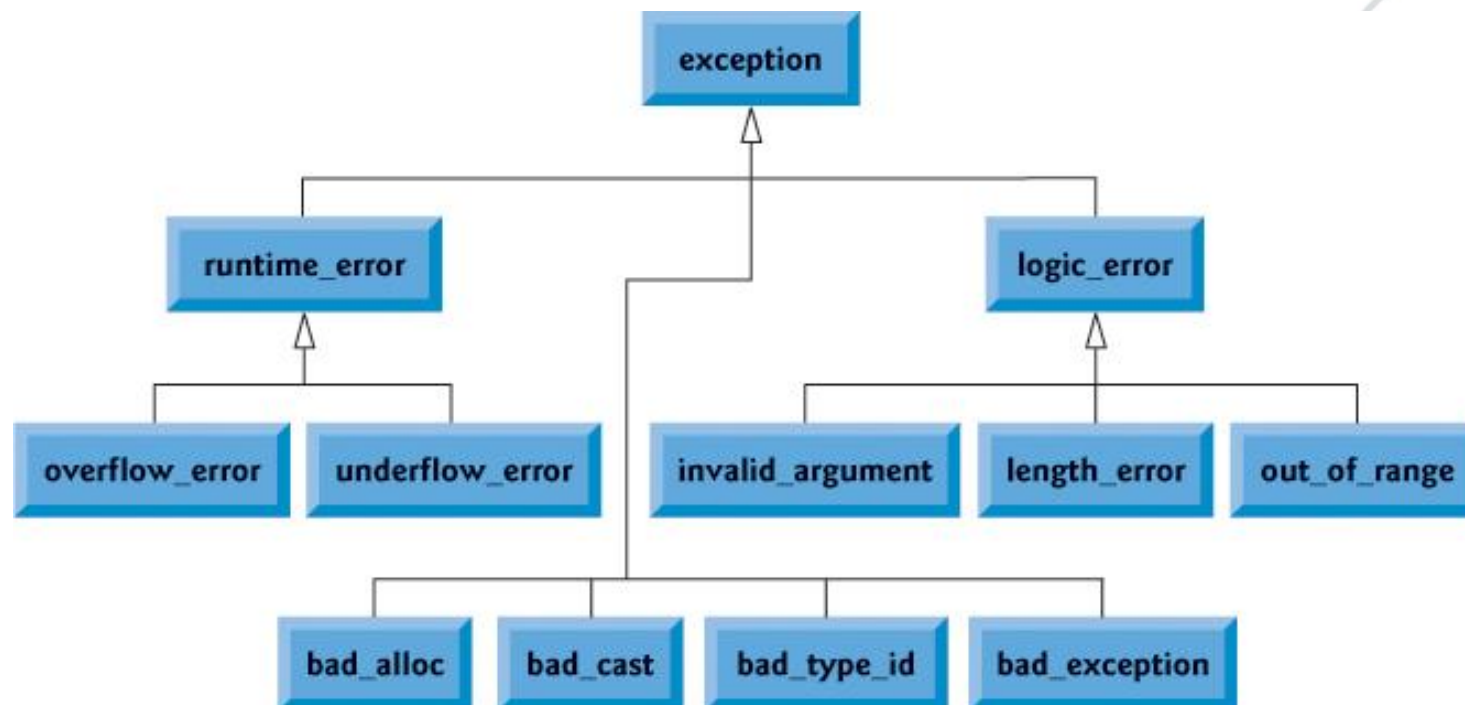
## Dicas:

1. Incorpore a estratégia de tratamento de exceção **desde o princípio** de seu projeto. Pode ser complicado incluir um tratamento eficaz depois que o sistema estiver pronto.
2. Evite utilizar o tratamento de exceção como uma **forma alternativa de fluxo de controle**. Exceções “adicionais” podem atrapalhar e ser confundidas com tratamento de exceções genuínas.
3. Quando não ocorrem exceções, o código de tratamento fica pouco ou nada sujeito a penalidade no desempenho. Assim, seu **programa fica mais eficiente**.
4. Funções com condições de erro comuns devem retornar valores de erro (como *0*, *nullptr*, etc.) ao invés de disparar exceções. Assim, a função chamadora poderá fazer um teste simples para determinar erro.

# Hierarquia de Exceções da Biblioteca-Padrão



A própria biblioteca padrão de C++ define uma série de tipos de exceções, organizadas em uma hierarquia, que podem ser utilizadas em nossos programas. Essa hierarquia é encabeçada pela classe-base **exception** (`<exception>`), que contém a função virtual **what()** cujas classes derivadas podem sobrepor para emitir mensagens de erro apropriadas. Veja algumas classes dessa hierarquia:





## Algumas classes derivadas de *exception* são:

- **bad\_alloc**: disparada pelo comando *new*, quando não é possível alocar memória. `<new>`
- **runtime\_error**: Indica erros em tempo de execução. `<stdexcept>`

Dela são derivadas:

- **overflow\_error**: que descreve um *overflow* aritmético, ou seja, quando o resultado de uma operação é maior que o número que pode ser armazenado no computador.
- **underflow\_error**: que é um erro de *underflow* aritmético, quando o número é menor do que o computador consegue armazenar.



- **logic\_error**: Indica erros de lógica no programa. `<stdexcept>`

Dela são derivadas:

- **invalid\_argument**: que indica que um argumento inválido foi passado para uma função. Repare que isso pode ser evitado na implementação. Ex: *// bitset constructor throws an invalid\_argument if initialized with a string containing characters other than 0 and 1*
- **length\_error**: indica que um componente maior que o tamanho máximo permitido para um objeto sendo manipulado foi usado para esse objeto. Ex: *vector throws a length\_error se redimensionado para um valor acima de max\_size.*
- **out\_of\_range**: indica que houve uma tentativa de se acessar um elemento do array fora de seu intervalo permitido.



# Exemplos: bad\_alloc



```
#include <iostream>
#include <new>

using namespace std;

int main () {

    try
    {
        // tentativa de alocação sem noção, gera exceção bad_alloc
        int *myarray= new int[10000000000];
    }
    catch (bad_alloc &ba)
    {
        cerr << "bad_alloc capturada: " << ba.what() << '\n';
    }

}
```

bad\_alloc capturada: std::bad\_array\_new\_length

# Exemplos: overflow\_error



```
// overflow_error exception, reserved storage is not enough
#include <bitset>
#include <iostream>
#include <stdexcept>

using namespace std;

int main() {
    try
    {
        // template based
        bitset<100> bitset;
        bitset[99] = 1;
        bitset[0] = 1;

        // to_ulong() é a única função de C++ que dispara a exceção overflow_error
        unsigned long Test = bitset.to_ulong();
    }
    catch(overflow_error &err)
    {
        cerr<<"overflow_error capturado: "<<err.what()<<endl;
    }
}
```

overflow\_error capturado: \_Base\_bitset::\_M\_do\_to\_ulong

# Exemplos: out\_of\_range



```
#include <iostream>
#include <stdexcept>
#include <vector>


using namespace std;

int main () {
    vector<int> myvector(10);
    try
    {
        myvector.at(20)=100; // vector::at throws an out-of-range
    }
    catch (out_of_range& oor)
    {
        std::cerr << "Out of Range error: " << oor.what() << '\n';
    }
}
```

```
Out of Range error: vector::_M_range_check: __n (which is 20) >= this->size() (which is 10)
```

O operador `new`, assim como as funções `at()` de `vector` e `to_ulong` de `bitset` dispararam as exceções que capturamos em nossos programas.

Da mesma maneira, podemos disparar exceções da biblioteca padrão em nossos programas, quando for necessário. Veja:



```
// exemplo simples
void set_valor(int *v, int val, int pos, int sz)
{
    if(pos >= sz)
        throw std::out_of_range("set_valor() : indice fora dos limites");
    v[pos] = val;
}

int main()
{
    const int size = 10;
    int vec[size];
    try {
        // vec recebe o valor 123 na posicao 12, tem tamanho igual size
        set_valor(vec, 123, 12, size);
    } catch(out_of_range &ex) {
        cout << "out_of_range capturada: " << ex.what() << "\n";
    }
}
```

```
out_of_range capturada: set_valor() : indice fora dos limites
```

# Exemplos:

um exemplo mais completo...



```
// exemplo simples
void set_valor(int *v, int val, int pos, int sz)
{
    if(pos >= sz)
        throw out_of_range("set_valor() : indice fora dos limites.");
    if(val > 100)
        throw overflow_error("set_valor() : valores precisam ser menores que 100.");
    v[pos] = val;
}

int main()
{
    const int size = 10;
    int vec[size];
    try {
        // vec recebe o valor 123 na posicao 1, tem tamanho igual size
        set_valor(vec, 123, 1, size);
    } catch(out_of_range &ex) {
        cout << "out_of_range capturada: " << ex.what() << "\n";
    } catch(overflow_error &ex) {
        cout << "overflow_error capturada: " << ex.what() << "\n";
    } catch(...) { // exception não dá na mesma? NÃO! Não é obrigado a herdar de exception.
        cout << "Captura qualquer exceção\n"; // default
    }
}
```



“É uma questão de bom senso escolher um método e experimentá-lo. Se ele falhar, admita-o francamente e tente outro. Mas, acima de tudo, tente alguma coisa.”

Franklin D. Roosevelt





# Referências

- <https://cplusplus.com/reference/>
- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.