Recursos Avançados de C++

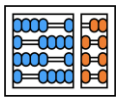
Módulo 3

Prof. Dr. Bruno B. P. Cafeo

Instituto de Computação
Universidade Estadual de Campinas

Agenda

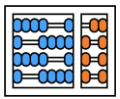
- Tipos de erros
- Gerenciamento classico de erros
- Exceções
  - Tratando exceções
  - Lançando exceções
  - Criando exceções
- Assertions
- Especificações de exceções e noexcept



Tipos de erros

- Erros de Compilação
  - Ocorrem durante a compilação.
  - Resultam de problemas de sintaxe.
  - Impedem que o código seja compilado.
  - Exemplos: falta de ponto e vírgula, parênteses não fechados.

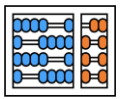
- Erros de Ligação
  - Ocorrem durante a ligação.
  - Resultam de referências a funções ou variáveis não definidas.
  - O programa é compilado, mas não pode ser vinculado.
  - Exemplos: funções não implementadas, conflitos de nome.



Tipos de erros

- Erros de Lógica
  - Ocorrem durante a execução.
  - Não causam falhas imediatas, mas levam a resultados incorretos.
  - Resultam de algoritmos ou lógica de programação incorretos.
  - Exemplos: cálculos errados, condições mal definidas.

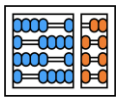
- Erros de Execução
  - Ocorrem durante a execução.
  - Podem causar falhas, travamento ou término inesperado.
  - Resultam de condições inesperadas ou excepcionais.
  - Exemplos: divisão por zero, acesso a memória não alocada.



Alternativas para o gerenciamento/tratamento de erros

- Terminar o programa
- Retornar um valor de erro
- Deixar o programa em um estado de erro
- Chamar uma função que trata o erro

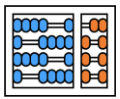
Comumente em um sistema essas funções coexistem de maneira não sistematizada



O que são exceções?

- Em programação, exceções são eventos anormais ou erros que ocorrem durante a execução de um programa.

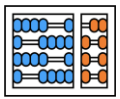
- Exemplos incluem divisão por zero, acesso a um índice fora dos limites de um array, tentativa de abertura de um arquivo inexistente ou um produto inexistente em estoque.



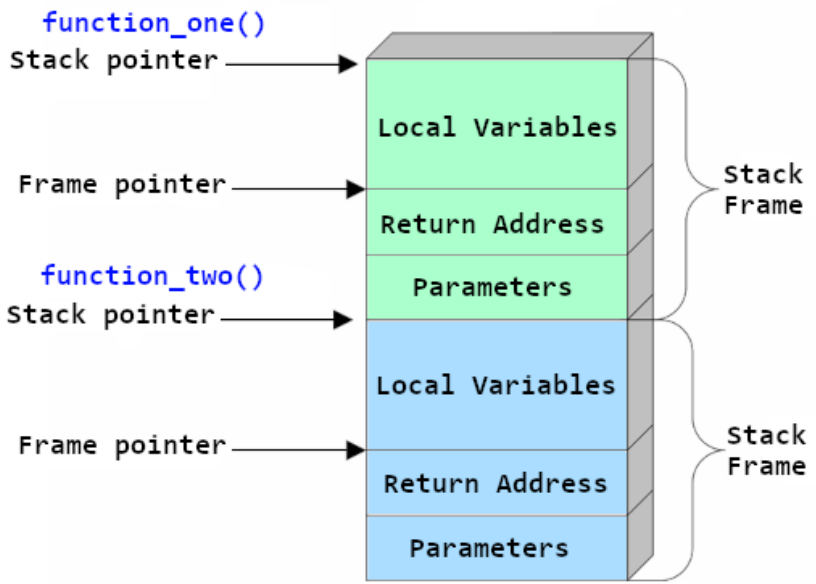
Por que precisamos tratar exceções?

- O tratamento de exceções é essencial para manter a estabilidade de um programa, lidando com erros e fluxos anormais de forma controlada em vez de interromper abruptamente a execução.

- Ele permite que os desenvolvedores tomem medidas adequadas para lidar com erros e fornecer mensagens de erro significativas aos usuários.

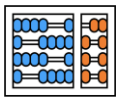


Depuração de erros

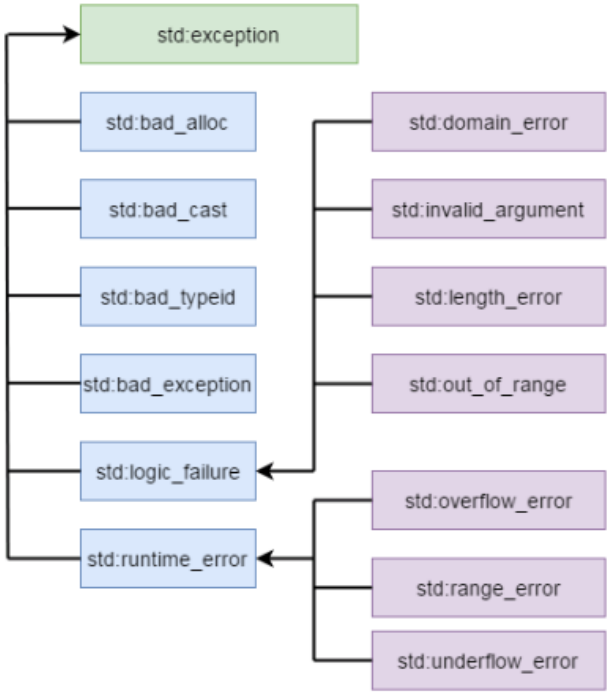


Disclaimer: Eventos síncronos vs. assíncronos

- O mecanismo de tratamento de exceção do C++ não trata eventos assíncronos!!
  - Eventos síncronos: erros de I/O, checagem de intervalo de array, ...
  - Eventos assíncronos: Interrupção de teclado, falha de energia, ...

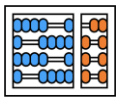


Hierarquia de exceções std::exception



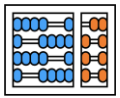
Tratando exceções

- **Sintaxe básica do bloco try-catch**
  - O bloco try é usado para envolver código que pode gerar exceções.
  - O bloco catch captura e trata exceções específicas que podem ocorrer no bloco try.
- **Capturando e tratando exceções**
  - Quando uma exceção é lançada no bloco try, o programa verifica os blocos catch correspondentes para tratar essa exceção.
  - O código dentro do bloco catch é executado se uma exceção compatível for lançada.
- **A necessidade de um bloco catch correspondente para cada bloco try**
  - Cada bloco try deve ter pelo menos um bloco catch correspondente ou um bloco finally (ou ambos).
  - O código de tratamento de exceção deve ser apropriado para o tipo de exceção capturada.



Exemplo

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

int divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw std::runtime_error("Divisão por zero não é permitida.");
    }
    return numerator / denominator;
}

int main() {
    int numerator, denominator;

    try {
        std::cout << "Digite o numerador: ";
        std::cin >> numerator;
        std::cout << "Digite o denominador: ";
        std::cin >> denominator;

        int result = divide(numerator, denominator);
        std::cout << "Resultado da divisão: " << result << std::endl;
    } catch (const std::runtime_error &e) {
        std::cerr << "Erro: " << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Erro desconhecido." << std::endl;
    }

    return 0;
}
```

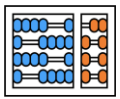


Lançando exceções

- Você pode lançar exceções explicitamente usando a palavra-chave "throw", seguida da exceção a ser lançada.

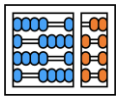
- Isso permite que você crie cenários personalizados de exceção.

```
3
4   int divide(int numerator, int denominator) {
5       if (denominator == 0) {
6           throw std::runtime_error("Divisão por zero não é permitida.");
7       }
8       return numerator / denominator;
9   }
10
11  int main() {
```



Relançando exceções

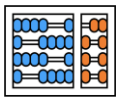
```cpp
#include <iostream>
#include <stdexcept>

int divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw std::runtime_error("Divisão por zero não é permitida.");
    }
    return numerator / denominator;
}

int main() {
    int numerator, denominator;

    try {
        std::cout << "Digite o numerador: ";
        std::cin >> numerator;
        std::cout << "Digite o denominador: ";
        std::cin >> denominator;

        int result = divide(numerator, denominator);
        std::cout << "Resultado da divisão: " << result << std::endl;
    } catch (const std::runtime_error &e) {
        throw;
    } catch (...) {
        std::cerr << "Erro desconhecido." << std::endl;
    }

    return 0;
}
```



Exceções personalizadas

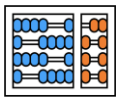
- Em algumas situações, as exceções padrão não representam adequadamente os erros específicos do seu aplicativo.

- Criar exceções personalizadas permite modelar erros relacionados ao domínio do seu software.



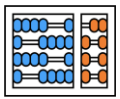
Como criar uma exceção personalizada

- Crie uma classe que herde de std::exception ou de uma de suas subclasses, como por exemplo std::runtime_error.
- Adicione construtores e métodos para personalizar a exceção.
- Lance a exceção personalizada quando apropriado e trate-a conforme necessário.



Exemplo

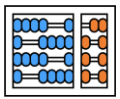
```
#include <iostream>
#include <stdexcept>

class DivisaoPorZeroException : public std::runtime_error {
public:
    DivisaoPorZeroException() : std::runtime_error("Divisão por zero não é permitida.") {}
};

int divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw DivisaoPorZeroException();
    }
    return numerator / denominator;
}

```



Lançando e tratando uma exceção personalizada

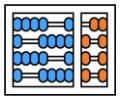
- As exceções personalizadas podem ser lançadas em resposta a erros específicos ao domínio, como "Produto não encontrado" em um sistema de comércio eletrônico.

- Trate exceções personalizadas de forma adequada para fornecer feedback significativo ao usuário e solucionar problemas.



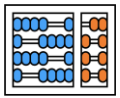
Exemplo

```
16  int main() {
17      int numerator, denominator;
18
19      try {
20          std::cout << "Digite o numerador: ";
21          std::cin >> numerator;
22          std::cout << "Digite o denominador: ";
23          std::cin >> denominator;
24
25          int result = divide(numerator, denominator);
26          std::cout << "Resultado da divisão: " << result << std::endl;
27      } catch (const DivisaoPorZeroException &e) {
28          std::cerr << "Erro: " << e.what() << std::endl;
29      } catch (...) {
30          std::cerr << "Erro desconhecido." << std::endl;
31      }
32
33      return 0;
34  }
35
```



Assertions

- São usadas para verificar condições pré e pós-condições em seu código.
- São usadas principalmente para depuração e verificação de suposições durante o desenvolvimento.
- Normalmente desabilitadas em versões de produção para melhor desempenho.
- Encerram o programa imediatamente se a condição não for atendida.

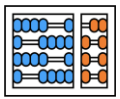


Assertions

- A opção NDEBUG deve estar desabilitada para que o programa informe que a condição falhou

```cpp
#include <iostream>
#include <cassert>

int main() {
    int valor = 10;

    // Verifica se o valor é maior que 5 usando uma afirmação.
    assert(valor > 5);

    std::cout << "O valor é: " << valor << std::endl;

    return 0;
}
```

```cpp
#include <iostream>

template <typename T>
void process_data(T data) {
    static_assert(std::is_integral<T>::value, "T deve ser um tipo inteiro.");
    // O resto do código aqui...
}

int main() {
    int value = 42;
    process_data(value);

    double notAnInt = 3.14;
    // Isso gerará um erro de compilação devido à assertiva.
    // process_data(notAnInt);

    std::cout << "Programa continuando após processar os dados." << std::endl;
    return 0;
}
```



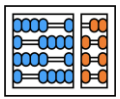
UNICAMP

Especificação de exceções vs. noexcept

- Especificações de exceções (Exception Specification) são uma parte da linguagem C++ que permite especificar quais exceções uma função pode lançar ou se ela não lançará exceções.

- As especificações de exceções foram depreciadas no C++11

```
1  #include <iostream>
2
3  // Exemplo de função com uma exception specification tradicional.
4  void minhaFuncao() throw(std::runtime_error) {
5      // Código da função que pode lançar uma exceção std::runtime_error.
6      throw std::runtime_error("Exemplo de exceção std::runtime_error.");
7  }
8
```

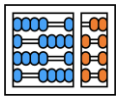


Especificação de exceções vs. noexcept

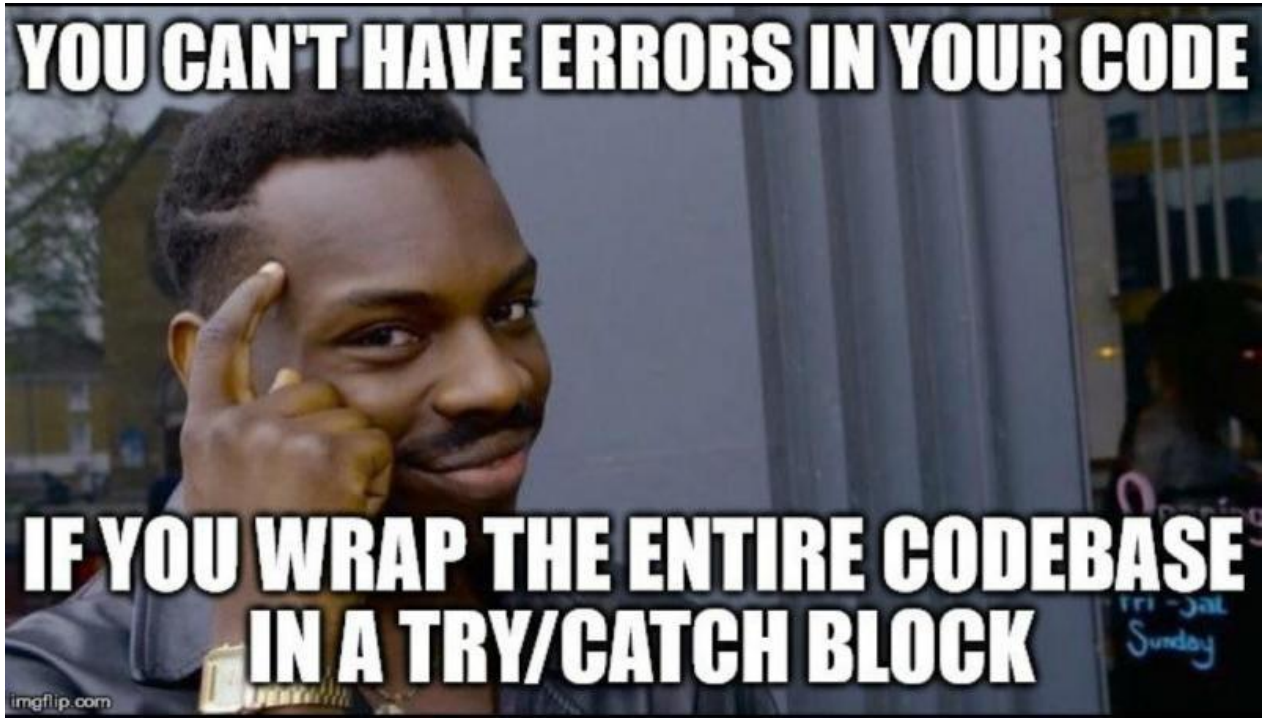
- No C++11 e versões posteriores, a cláusula noexcept é usada para especificar que uma função não lançará exceções.

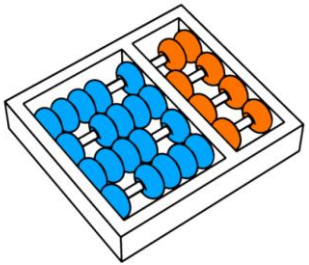
- Isso ajuda a melhorar a segurança e a otimização do código, substituindo as exception specifications tradicionais.

```
1  #include <iostream>
2
3  // Exemplo de função com uma exception specification tradicional.
4  void minhaFuncao() throw(std::runtime_error) {
5      // Código da função que pode lançar uma exceção std::runtime_error.
6      throw std::runtime_error("Exemplo de exceção std::runtime_error.");
7  }
8
```



Use exceções de maneira consciente!





INSTITUTO DE COMPUTAÇÃO

UNICAMP

Prof. Dr. Bruno B. P. Cafeo

Sala 04
Instituto de Computação - Unicamp
Av. Albert Einstein, 1251
Cidade Universitária
Campinas – SP
13083-852

https://ic.unicamp.br/~cafeo/
cafeo@ic.unicamp.br