# Tratamento de Exceções em C++

## Sumário

1. Introdução
2. Mecanismos Básicos
3. Hierarquia de Exceções
4. Mecanismos Avançados
5. Boas Práticas
6. Novidades em C++11/14/17/20/23
7. Tratamento de Erros Alternativos
8. Performance e Otimização

---

## 1. Introdução

### 1.1 O que são Exceções?

Exceções são eventos anormais ou erros que ocorrem durante a execução de um programa, interrompendo o fluxo normal de instruções. Em C++, o tratamento de exceções oferece um mecanismo estruturado para detectar e responder a estes eventos, permitindo separar o código de tratamento de erros da lógica principal do programa.

### 1.2 Benefícios do Tratamento de Exceções

O tratamento de exceções em C++ oferece vários benefícios significativos:

1. **Separação de Código**: Separa a lógica principal do tratamento de erros, tornando o código mais limpo e legível
2. **Propagação de Erros**: Permite que erros sejam propagados pela pilha de chamadas sem código adicional
3. **Agrupamento de Erros**: Permite tratar diferentes tipos de erros em locais centralizados
4. **Desacoplamento**: Componentes que detectam erros não precisam conhecer como eles serão tratados
5. **Tipos de Erros**: Informações específicas sobre o erro podem ser encapsuladas no objeto de exceção
6. **Garantias de Limpeza**: O mecanismo de RAII combinado com exceções garante limpeza adequada de recursos

### 1.3 Quando Usar Exceções

Exceções são apropriadas para:

1. **Erros de Construção**: Quando um objeto não pode ser inicializado corretamente
2. **Condições Inesperadas**: Situações que impelem a lógica normal do programa
3. **Erros de Recursos**: Falhas na alocação de memória, acesso a arquivos, conexões de rede
4. **Erros de Validação**: Dados de entrada inválidos que não podem ser processados
5. **Estados Inválidos**: Quando o programa atinge um estado que não deveria ser possível

O tratamento de exceções não é recomendado para:

1. **Controle de Fluxo Regular**: Não use exceções para decisões lógicas normais
2. **Erros Esperados**: Condições que fazem parte do funcionamento normal
3. **Erros que Podem Ser Verificados Previamente**: Quando validações simples evitariam o erro

### 1.4 Anatomia do Mecanismo de Exceções

O mecanismo de exceções em C++ é composto por três componentes principais:

1. **throw**: Lança uma exceção quando um problema é detectado
2. **try**: Define um bloco de código onde exceções podem ser lançadas
3. **catch**: Define um bloco que manipula as exceções lançadas no bloco try

Exemplo básico:

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

double divide(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Divisão por zero");
    }
    return a / b;
}

int main() {
    try {
        std::cout << divide(10, 2) << std::endl; // OK
        std::cout << divide(10, 0) << std::endl; // Lança exceção
    }
    catch (const std::exception& e) {
        std::cerr << "Erro capturado: " << e.what() << std::endl;
    }

    return 0;
}
```

## 2. Mecanismos Básicos

### 2.1 Lançando Exceções (throw)

O comando `throw` é usado para lançar uma exceção quando um problema é detectado. Pode lançar qualquer tipo de objeto:

```
// Lançando um tipo padrão
throw std::runtime_error("Mensagem de erro");

// Lançando um inteiro
throw 42;
```

```
// Lançando uma string
throw std::string("Erro crítico");

// Lançando um objeto customizado
throw MinhaExcecao("Detalhes do erro");
```

**Funcionamento**: Quando uma exceção é lançada, o fluxo de execução normal é interrompido. O C++ começa a procurar na pilha de chamadas pelo bloco `catch` mais próximo que possa tratar a exceção. Se nenhum manipulador adequado for encontrado, a função `std::terminate()` é chamada, encerrando o programa.

### 2.2 Blocos try-catch

Os blocos `try-catch` são usados para capturar e tratar exceções lançadas:

```
try {
    // Código que pode lançar exceções
    funcaoPerigosa();
}
catch (const std::runtime_error& e) {
    // Trata exceções do tipo std::runtime_error
    std::cerr << "Erro de tempo de execução: " << e.what() << std::endl;
}
catch (const std::exception& e) {
    // Trata outras exceções derivadas de std::exception
    std::cerr << "Exceção padrão: " << e.what() << std::endl;
}
catch (...) {
    // Captura qualquer outro tipo de exceção
    std::cerr << "Exceção desconhecida capturada" << std::endl;
}
```

**Aspectos importantes**:

- Os blocos `catch` são verificados na ordem em que aparecem
- Use `catch(...)` para capturar qualquer tipo de exceção não tratada anteriormente
- Apenas o primeiro bloco `catch` correspondente será executado
- Exceções derivadas devem ser capturadas antes das classes base

### 2.3 Cláusula de Exceção (throw)

A cláusula de exceção especifica quais exceções uma função pode lançar:

```
// C++98/03 (agora deprecated)
void funcao() throw(std::exception, std::runtime_error) {
    // Essa função pode lançar std::exception ou std::runtime_error
}

// C++11 e posterior - noexcept
void funcaoSegura() noexcept {
```

```
    // Esta função não deve lançar exceções
}

// C++11 e posterior - noexcept condicional
template <typename T>
void funcao() noexcept(noexcept(T())) {
    // Não lança exceções se o construtor de T não lançar
    T obj;
}
```

**Evolução das especificações de exceção**:

- As especificações de exceção dinâmicas (`throw(...)`) foram descontinuadas em C++11 e removidas em C++17
- `noexcept` foi introduzido em C++11 como um substituto melhor e mais eficiente
- `noexcept` tem impacto na otimização do código e na propagação de exceções

### 2.4 Relançando Exceções

É possível capturar uma exceção e relançá-la, permitindo tratamento parcial ou modificação:

```
try {
    // Código que pode lançar exceções
    funcaoPerigosa();
}
catch (std::exception& e) {
    // Faz algum tratamento
    registraErro(e);

    // Relança a mesma exceção
    throw;

    // Alternativa: lança uma nova exceção
    throw std::runtime_error("Erro encapsulado: " + std::string(e.what()));
}
```

**Opções de relançamento**:

- `throw;` relança a exceção atual sem modificação (preservando o tipo dinâmico)
- `throw e;` cria uma nova exceção, possivelmente perdendo informações da exceção original devido ao objeto de exceção ser copiado (slicing)
- Pode-se lançar uma nova exceção que encapsula ou adiciona informações à original

## 3. Hierarquia de Exceções

### 3.1 Hierarquia Padrão do C++

A biblioteca padrão C++ fornece uma hierarquia de exceções baseada na classe `std::exception`:

```
std::exception
 ├── std::logic_error
 │    ├── std::invalid_argument
 │    ├── std::domain_error
 │    ├── std::length_error
 │    ├── std::out_of_range
 │    └── std::future_error (C++11)
 ├── std::runtime_error
 │    ├── std::range_error
 │    ├── std::overflow_error
 │    ├── std::underflow_error
 │    ├── std::regex_error (C++11)
 │    ├── std::system_error (C++11)
 │    │    └── std::ios_base::failure (C++11)
 │    └── std::filesystem::filesystem_error (C++17)
 ├── std::bad_alloc
 ├── std::bad_cast
 ├── std::bad_typeid
 ├── std::bad_exception
 ├── std::bad_function_call (C++11)
 ├── std::bad_weak_ptr (C++11)
 ├── std::bad_optional_access (C++17)
 ├── std::bad_variant_access (C++17)
 └── std::bad_any_cast (C++17)
```

**Categorias principais**:

- `std::logic_error`: Erros detectáveis por análise do código (erro de programação)
- `std::runtime_error`: Erros detectáveis apenas durante a execução (erro externo)
- Outras classes para erros específicos (falha de alocação, cast inválido, etc.)

## 3.2 Criando Exceções Personalizadas

É possível criar hierarquias próprias de exceções derivando das classes padrão:

```cpp
class MinhaBaseException : public std::exception {
private:
    std::string mensagem;

public:
    explicit MinhaBaseException(const std::string& msg)
        : mensagem(msg) {}

    const char* what() const noexcept override {
        return mensagem.c_str();
    }

    virtual ~MinhaBaseException() noexcept = default;
};
```

```cpp
class DatabaseException : public MinhaBaseException {
private:
    int codigo_erro;

public:
    DatabaseException(const std::string& msg, int code)
        : MinhaBaseException(msg), codigo_erro(code) {}

    int getCodigoErro() const noexcept {
        return codigo_erro;
    }
};
```

**Práticas recomendadas**:

- Derive de `std::exception` ou suas classes derivadas
- Sobrescreva o método `what()` para fornecer mensagens descritivas
- Torne sua classe base de exceção abstrata se apropriado
- Adicione informações específicas relevantes para o diagnóstico
- Garanta que todos os métodos em classes de exceção sejam `noexcept`

### 3.3 Captura Polimórfica

A hierarquia de exceções permite captura polimórfica, onde uma classe base pode capturar exceções derivadas:

```cpp
try {
    // Pode lançar diferentes tipos de exceções
    conexaoBanco.executar(query);
}
catch (const DatabaseException& e) {
    // Tratamento específico para erros de banco de dados
    std::cerr << "Erro de BD: " << e.what()
              << " (Código: " << e.getCodigoErro() << ")" << std::endl;
}
catch (const MinhaBaseException& e) {
    // Tratamento para outros erros personalizados
    std::cerr << "Erro personalizado: " << e.what() << std::endl;
}
catch (const std::exception& e) {
    // Tratamento para exceções padrão
    std::cerr << "Exceção padrão: " << e.what() << std::endl;
}
```

**Dicas importantes**:

- Capture por referência constante para evitar object slicing
- Ordene os blocos catch do mais específico para o mais geral
- Considere adicionar informações de contexto em cada nível

# 4. Mecanismos Avançados

## 4.1 Stack Unwinding

Quando uma exceção é lançada, o C++ executa um processo chamado "stack unwinding" (desenrolamento da pilha):

1. Procura um manipulador adequado subindo pela pilha de chamadas
2. Destrói automaticamente todos os objetos com escopo automático durante o processo
3. Executa os destrutores desses objetos, garantindo liberação de recursos
4. Se nenhum manipulador for encontrado, chama `std::terminate()`

Exemplo demonstrando o processo:

```
class Recurso {
private:
    std::string nome;

public:
    explicit Recurso(const std::string& n) : nome(n) {
        std::cout << "Adquirindo " << nome << std::endl;
    }

    ~Recurso() {
        std::cout << "Liberando " << nome << std::endl;
    }
};

void funcao3() {
    Recurso r3("Recurso 3");
    std::cout << "Lançando exceção de funcao3()" << std::endl;
    throw std::runtime_error("Erro em funcao3");
}

void funcao2() {
    Recurso r2("Recurso 2");
    funcao3();
    std::cout << "Este código nunca será executado" << std::endl;
}

void funcao1() {
    Recurso r1("Recurso 1");
    try {
        funcao2();
    }
    catch (const std::exception& e) {
        std::cout << "Exceção capturada: " << e.what() << std::endl;
    }
    std::cout << "Continuando após o tratamento da exceção" << std::endl;
}
```

Neste exemplo, quando `funcao3()` lança uma exceção:

1. O destrutor de `r3` é chamado automaticamente
2. O controle volta para `funcao2()`, chamando o destrutor de `r2`
3. O controle volta para `funcao1()`, onde a exceção é tratada
4. O destrutor de `r1` é chamado apenas quando `funcao1()` termina

### 4.2 RAII (Resource Acquisition Is Initialization)

RAII é um idioma de programação C++ onde a aquisição de recursos ocorre durante a inicialização do objeto, e a liberação ocorre no destrutor. Combinado com exceções, RAII garante que recursos sejam liberados adequadamente mesmo quando exceções são lançadas.

```
class FileHandler {
private:
    FILE* file;

public:
    FileHandler(const char* filename, const char* mode) {
        file = fopen(filename, mode);
        if (!file) {
            throw std::runtime_error("Não foi possível abrir o arquivo");
        }
    }

    ~FileHandler() {
        if (file) {
            fclose(file);
        }
    }

    void write(const std::string& data) {
        if (fputs(data.c_str(), file) == EOF) {
            throw std::runtime_error("Erro ao escrever no arquivo");
        }
    }
};

void processarArquivo() {
    FileHandler file("dados.txt", "w");  // Abre o arquivo

    // Se uma exceção for lançada aqui, o destrutor de FileHandler
    // será chamado automaticamente, fechando o arquivo
    file.write("Linha 1\n");
    file.write("Linha 2\n");

    // Quando a função terminar, o destrutor também será chamado
}
```

**Benefícios do RAII com exceções**:

- Garante liberação de recursos mesmo quando exceções ocorrem
- Elimina vazamentos de recursos, tornando o código mais robusto
- Simplifica o código eliminando blocos try-catch apenas para limpeza

### 4.3 Function try Blocks

Function try blocks permitem capturar exceções lançadas durante a inicialização de membros e na lista de inicialização de construtores:

```
class DatabaseConnection {
private:
    Connection conn;
    Logger log;

public:
    // Function try block para o construtor
    DatabaseConnection(const std::string& connectionString)
    try : conn(connectionString),
          log("database.log") {
        // Corpo do construtor
        std::cout << "Conexão estabelecida" << std::endl;
    }
    catch (const ConnectionException& e) {
        std::cerr << "Falha na conexão: " << e.what() << std::endl;
        throw; // Relança a exceção
    }
    catch (const std::exception& e) {
        std::cerr << "Erro durante inicialização: " << e.what() << std::endl;
        throw;
    }
};
```

**Características importantes**:

- Function try blocks podem ser usados com qualquer função, não apenas construtores
- São especialmente úteis para construtores, onde exceções na lista de inicialização seriam inacessíveis de outra forma
- Se uma exceção for capturada mas não relançada, a função ainda terminará lançando implicitamente a exceção original
- Para construtores, relançar a exceção é geralmente necessário, pois o objeto não pode ser parcialmente construído

### 4.4 Nested Exceptions

C++11 introduziu o conceito de exceções aninhadas, permitindo encapsular uma exceção dentro de outra:

```
#include <exception>
#include <iostream>
#include <nested_exception>
```

```cpp
void processar() {
    try {
        // Código que pode lançar exceções
        throw std::runtime_error("Erro na camada de processamento");
    }
    catch (const std::exception& e) {
        // Adiciona contexto e relança
        throw std::nested_exception();
    }
}

void executar() {
    try {
        processar();
    }
    catch (const std::nested_exception& nested) {
        try {
            std::cout << "Exceção externa capturada" << std::endl;
            nested.rethrow_nested();
        }
        catch (const std::exception& e) {
            std::cout << "Exceção interna: " << e.what() << std::endl;
        }
    }
}
```

**Funções úteis para exceções aninhadas**:

- `std::throw_with_nested`: Lança uma nova exceção contendo a atual como aninhada
- `std::rethrow_nested`: Relança a exceção aninhada contida na exceção atual
- `std::nested_exception`: Classe base para exceções que podem conter outras exceções
- `std::is_nothrow_copy_constructible`: Trait para verificar se um tipo pode ser copiado sem lançar exceções

# 5. Boas Práticas

## 5.1 Quando Lançar e Capturar Exceções

**Lançar exceções quando**:

- Um objeto não pode ser construído corretamente
- Uma operação não pode ser concluída com sucesso
- Precondições críticas não são atendidas
- Um recurso necessário não está disponível

**Capturar exceções quando**:

- Você pode tratar o erro de maneira significativa
- Precisa converter um tipo de exceção em outro
- Em limites de componentes ou threads

- Em pontos de recuperação onde o programa pode continuar

## 5.2 Design Robusto com Exceções

Diretrizes para criar código robusto com exceções:

1. **Garantia Básica**: Se uma exceção for lançada, nenhum recurso vaza e todos os objetos permanecem em um estado válido (mas possivelmente diferente)
2. **Garantia Forte**: Se uma exceção for lançada, o estado do programa é como se a operação nunca tivesse sido tentada (operação atômica)
3. **Garantia de Não-lançamento**: A operação nunca lança exceções (marcada como `noexcept`)

Exemplo de implementação com garantia forte:

```
class BancoDados {
private:
    Conexao conn;

public:
    // Implementação com garantia forte
    void transferir(Conta& origem, Conta& destino, double valor) {
        // Abordagem "copy-and-swap"
        double saldo_origem_original = origem.getSaldo();
        double saldo_destino_original = destino.getSaldo();

        try {
            // Tenta fazer a transferência
            origem.debitar(valor);
            destino.creditar(valor);

            // Confirma a transação
            conn.commit();
        }
        catch (...) {
            // Restaura o estado original
            origem.setSaldo(saldo_origem_original);
            destino.setSaldo(saldo_destino_original);

            // Desfaz a transação
            conn.rollback();

            // Relança para tratamento em nível superior
            throw;
        }
    }
};
```

## 5.3 Exceções vs. Códigos de Erro

**Quando usar exceções**:

- Para condições excepcionais reais (não para controle de fluxo regular)
- Quando o erro precisa ser propagado por várias camadas
- Para erros que não podem ser tratados localmente
- Quando precisar de detalhes específicos sobre o erro

**Quando usar códigos de erro**:

- Para condições que ocorrem regularmente (não excepcionais)
- Quando o desempenho é crítico
- Em interfaces C
- Em ambientes com restrições de tempo real
- Em código com requisitos de exceção zero

### 5.4 Diretrizes Gerais

1. **Sempre capture exceções por referência** para evitar slice de objetos
2. **Não capture exceções muito genericamente** quando precisar de tratamento específico
3. **Use mensagens descritivas** para facilitar o diagnóstico
4. **Documente as exceções** que sua API pode lançar
5. **Limite onde as exceções podem ser lançadas** usando `noexcept`
6. **Crie hierarquias de exceções significativas** derivando de `std::exception`
7. **Siga o RAII** para garantir limpeza adequada de recursos
8. **Teste o caminho de exceção** tão cuidadosamente quanto o caminho normal

## 6. Novidades em C++11/14/17/20/23

### 6.1 Novidades do C++11

`noexcept`: Substituto para especificações de exceção dinâmicas com melhor desempenho.

```
void funcao() noexcept; // Não lança exceções
void funcao() noexcept(expressão); // Não lança se expressão for true
```

**Exceções aninhadas**: Permite encapsular uma exceção dentro de outra.

```
try {
    funcaoQuePoderiaLancarExcecao();
}
catch (const std::exception& e) {
    std::throw_with_nested(MinhaExcecao("Erro durante processamento"));
}
```

**Smart Pointers**: Facilitam o gerenciamento de memória com exceções.

```
std::unique_ptr<Recurso> ptr(new Recurso());
// ptr será automaticamente liberado se uma exceção for lançada
```

## 6.2 Novidades do C++14

**Predicado noexcept melhorado**: Verificação de `noexcept` em tempo de compilação.

```
template <typename T>
void funcao(T&& t) noexcept(noexcept(t.processo()))
{
    t.processo();
}
```

## 6.3 Novidades do C++17

**Remoção de especificações de exceção dinâmicas**: `throw(...)` foi removido completamente.

**`std::uncaught_exceptions()`**: Substituto para `std::uncaught_exception()`, permite saber quantas exceções estão atualmente ativas.

```
class Guard {
public:
    Guard() : exceptions_on_entry(std::uncaught_exceptions()) {}

    ~Guard() {
        // Verifica se uma nova exceção foi lançada durante o escopo deste objeto
        if (std::uncaught_exceptions() > exceptions_on_entry) {
            // Estamos saindo devido a uma exceção
            std::cout << "Destrutor chamado durante unwinding" << std::endl;
        } else {
            // Saída normal
            std::cout << "Destrutor chamado normalmente" << std::endl;
        }
    }

private:
    int exceptions_on_entry;
};
```

**Novas classes de exceção**:

- `std::bad_any_cast`
- `std::bad_optional_access`
- `std::bad_variant_access`
- `std::filesystem::filesystem_error`

## 6.4 Novidades do C++20

`std::source_location`: Fornece informações sobre a localização no código fonte.

```
#include <source_location>
#include <iostream>

void log_error(const std::string& message,
               const std::source_location& location =
                   std::source_location::current()) {
    std::cout << "Erro em "
              << location.file_name() << ":"
              << location.line() << ":"
              << location.column() << " - "
              << location.function_name() << ": "
              << message << std::endl;
}

void funcao() {
    log_error("Algo deu errado");
}
```

`std::unexpected`: Nova função para lidar com exceções inesperadas.

### 6.5 Novidades Propostas para C++23 e Futuras

`std::expected<T, E>`: Um tipo que representa ou um valor esperado ou um erro (alternativa a exceções).

```
#include <expected>

std::expected<int, std::error_code> dividir(int a, int b) {
    if (b == 0) {
        return std::unexpected(std::make_error_code(std::errc::invalid_argument));
    }
    return a / b;
}

void usar() {
    auto resultado = dividir(10, 2);
    if (resultado) {
        std::cout << "Resultado: " << *resultado << std::endl;
    } else {
        std::cout << "Erro: " << resultado.error().message() << std::endl;
    }
}
```

`try-expressions` (proposta): Permite usar try-catch em expressões.

```
// Proposta, não implementada ainda
auto resultado = try {
```

```
    return funcaoQuePoderiaLancarExcecao();
} catch (const std::exception& e) {
    return valor_default;
};
```

# 7. Tratamento de Erros Alternativos

## 7.1 Valores de Retorno Opcionais

C++17 introduziu `std::optional` para representar valores que podem ou não existir:

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

std::optional<double> dividir(double a, double b) {
    if (b == 0.0) {
        return std::nullopt; // Sem valor
    }
    return a / b; // Retorna valor encapsulado em optional
}

void usar() {
    auto resultado = dividir(10.0, 2.0);
    if (resultado) {
        std::cout << "Resultado: " << *resultado << std::endl;
    } else {
        std::cout << "Divisão por zero" << std::endl;
    }
}
```

## 7.2 Códigos de Erro e Status

Abordagem tradicional usando códigos de erro:

```
enum class StatusCode {
    Success,
    InvalidInput,
    ResourceNotFound,
    NetworkError
};

struct Result {
    StatusCode status;
    std::string mensagem;
    int dados;
};

Result processarDados(const std::string& entrada) {
```

```
    if (entrada.empty()) {
        return {StatusCode::InvalidInput, "A entrada não pode ser vazia", 0};
    }

    // Processamento
    return {StatusCode::Success, "Processado com sucesso", 42};
}
```

### 7.3 std::expected (C++23)

`std::expected` combina valores de retorno com informações de erro:

```
#include <expected>
#include <string>
#include <system_error>

std::expected<int, std::error_code> parsearInteiro(const std::string& s) {
    try {
        int valor = std::stoi(s);
        return valor;
    } catch (const std::invalid_argument&) {
        return std::unexpected(std::make_error_code(std::errc::invalid_argument));
    } catch (const std::out_of_range&) {
        return
std::unexpected(std::make_error_code(std::errc::result_out_of_range));
    }
}

void usar() {
    auto resultado = parsearInteiro("42");
    if (resultado) {
        std::cout << "Valor: " << *resultado << std::endl;
    } else {
        std::cout << "Erro: " << resultado.error().message() << std::endl;
    }
}
```

### 7.4 std::variant e std::visit

Usando `std::variant` para representar sucesso ou erro:

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

struct Erro {
    std::string mensagem;
};
```

```cpp
template <typename T>
using Resultado = std::variant<T, Erro>;

Resultado<int> dividir(int a, int b) {
    if (b == 0) {
        return Erro{"Divisão por zero"};
    }
    return a / b;
}

void processar(int a, int b) {
    auto resultado = dividir(a, b);

    std::visit([](auto&& arg) {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, int>)
```