

Programação e Desenvolvimento de Software 2

Tratamento de exceções

Prof. Héctor Azpúrua
(slides adaptados do Prof. Douglas Macharet)

Programação Defensiva

Abordagem de desenvolvimento de software que visa escrever um código que seja mais robusto seguro e capaz de lidar com situações inesperadas ou entradas inválidas. O objetivo é minimizar os erros e falhas durante a execução do programa, melhorando sua confiabilidade e manutenção.

“Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.”

— Steve McConnell, Code Complete

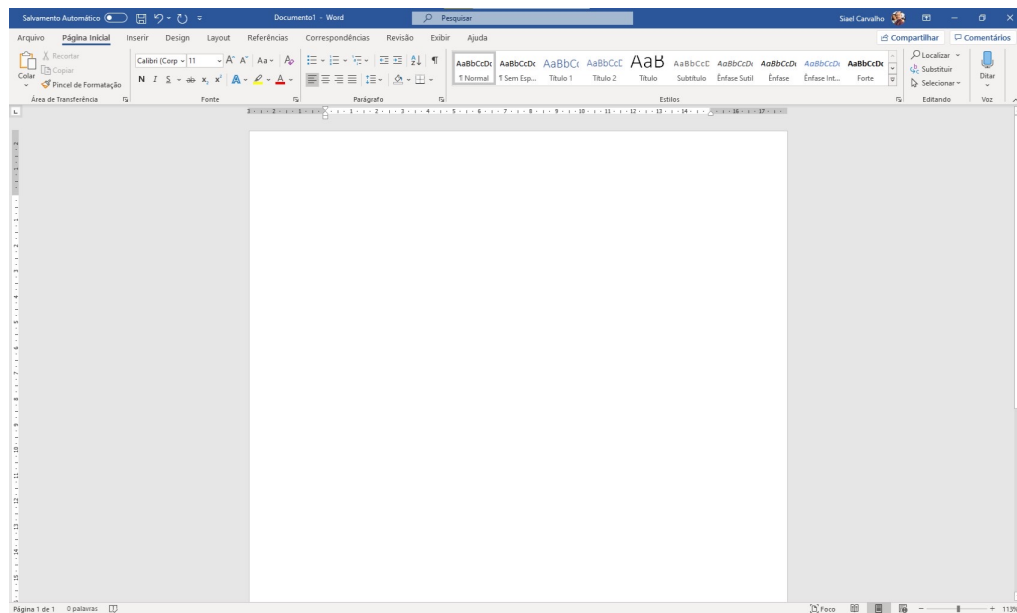
Programação defensiva

Robustez vs. Corretude

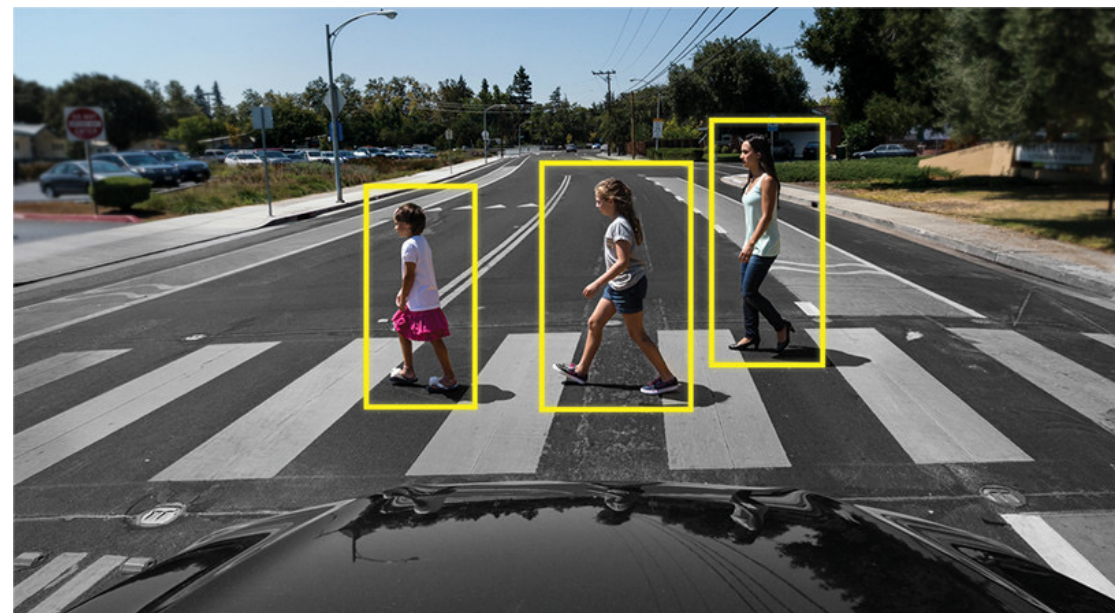
- Robustez
 - Sempre tentar fazer algo que permita que o software continue operando, mesmo que isso às vezes leve a resultados imprecisos
- Corretude (exatidão)
 - Significa nunca retornar um resultado impreciso
 - Não retornar nenhum resultado é melhor do que um incorreto
- Qual característica deve ser priorizada? ➔ Depende!

Programação defensiva

Robustez vs. Corretude



Robustez



Corretude

Programação defensiva

Estratégias

- Validação das entradas
- Asserções
- Programação por contrato
- Barricadas
- **Tratamento de exceções**

Introdução

- O que é uma exceção?
 - “Exceptional event”
 - Evento/acontecimento inesperado que ocorre no contexto da execução do programa (não necessariamente erro de lógica)
- Importante tratar e gerenciar esses eventos
- Demandam alteração no fluxo de execução

Introdução

- O que pode gerar uma exceção?
 - Erros de Código, entradas inválidas, falhas de hardware, ...
 - Exemplos
 - Timeout ao enviar dados pela rede
 - Abrir um arquivo inexistente
 - Acessar uma posição inválida em um vetor
 - Divisão por zero
- Como o sistema deve se comportar nesses casos?

A problem has been detected and windows has been shut down to prevent damage to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x00000000,0xF86B5A89)

*** gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory

Physical memory dump complete.

Contact your system administrator or technical support group for further assistance.


Introdução

Tratamento

- Ignorar
 - É um alarme falso, continuar a execução
- Reportar
 - Escrever uma mensagem na tela (arquivo)
- Terminar
 - Interromper completamente a execução
- Reparar
 - Corrigir e tentar se recuperar (prosseguir)

Introdução

Tratamento

- Definir valor de uma variável global
- Convenção de códigos de retorno
 - C/C++: 0 e $\neq 0$
 - Java: boolean
- Chamar rotina de processamento de erros
 - Retornar a última resposta válida
 - Retornar o valor válido mais próximo
- Lançar uma exceção e tratá-la! 

Exceções

- Maneira estruturada de informar/tratar que o programa (rotina) não deve (pode) continuar a execução normal
- **Sinalização** da existência de um problema
 - É criada uma **variável** que representa a exceção
 - A exceção (variável) deve então ser “**lançada**”
 - O código é desviado do fluxo normal de execução
- **Tratamento**
 - A “**captura**” da exceção é feita e um comportamento escolhido

Exceções



Exceções

- C++
 - Tratamento estruturado (parte da linguagem)
 - Mais poderoso e flexível que códigos de retorno
 - Instruções: `try-throw-catch`
- Exceções definidas como classes (ou outro tipo)
 - Classes → Vantagens do paradigma OO
 - Contém informações sobre o erro (contexto)
 - Pré-definidas ou Criadas pelo programador

Exceções

- Observada pela instrução `try`
 - Região protegida (observável)
 - Bloco de código onde pode ocorrer a exceção
- Capturada pela instrução `catch`
 - Bloco específico para cada tipo de exceção
 - Responsável pelo tratamento (manipulação)

Exceções

Exemplo 1

Região protegida

Região captura/
tratamento

```
int main() {  
    try {  
        std::string("abc").substr(10);  
    }  
    catch(std::out_of_range & e) {  
        std::cout << e.what() << std::endl;  
    }  
}
```

Código que resultará
em uma exceção
durante a execução.

Tipo da exceção que
deverá
ser capturada e
tratada.

Mensagem de
erro associada à
exceção.

[Wandbox](https://en.cppreference.com/w/cpp/string/basic_string/substr)

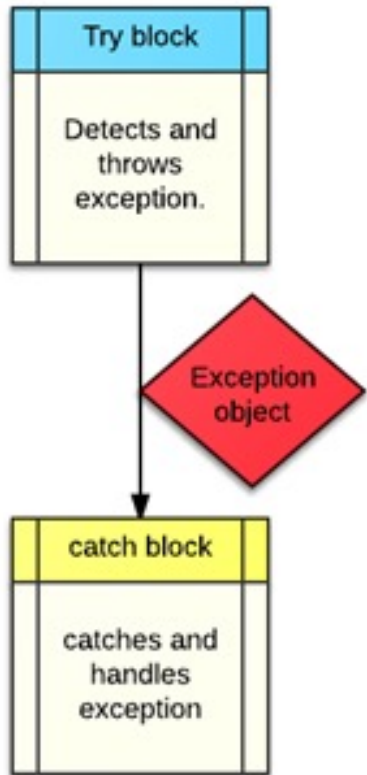
https://en.cppreference.com/w/cpp/string/basic_string/substr

Exceções

- Lançamento através da instrução **throw**
 - Sempre dentro de um bloco **try** (local ou não)
 - Se nada tratar (**catch**), o programa terminará
- A exceção lançada é um objeto
 - Previamente instanciado (raro)
 - Instanciado no momento do lançamento
 - Tipo deve ser parâmetro de um bloco **catch**

Exceções

Exemplo 2



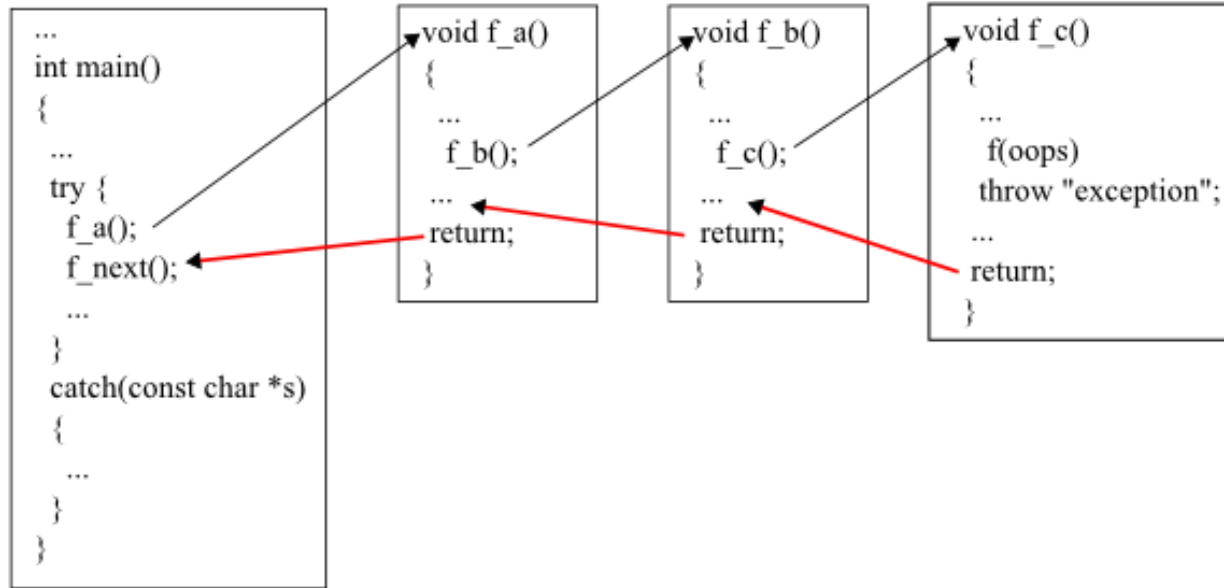
```
double metodo(double d) {  
    if (d > 1e7) {  
        throw std::overflow_error("Valor acima do esperado!");  
    }  
  
    return d*d;  
}  
  
int main() {  
    try {  
        double x = metodo(1e10);  
        std::cout << x << std::endl;  
    } catch (std::overflow_error& e) {  
        std::cout << "Erro. " << e.what() << std::endl;  
    }  
}
```

Atenção ao **tipo** da
exceção sendo
lançada/capturada!

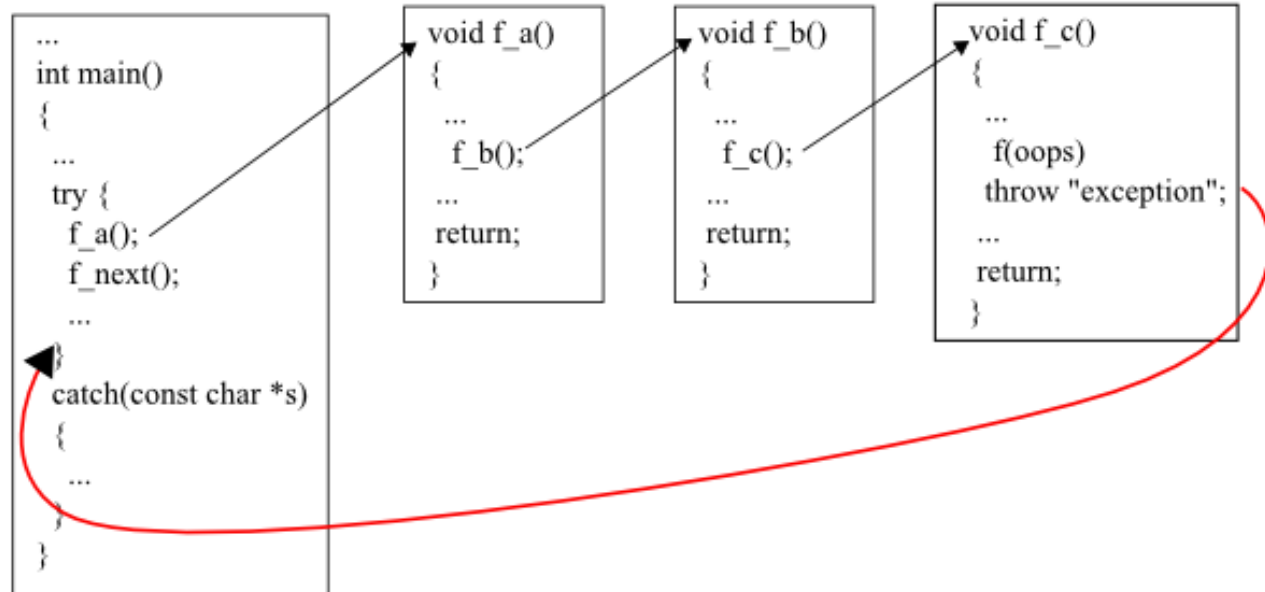
[Wandbox](#)

Exceções

Sem exceção



Com exceção



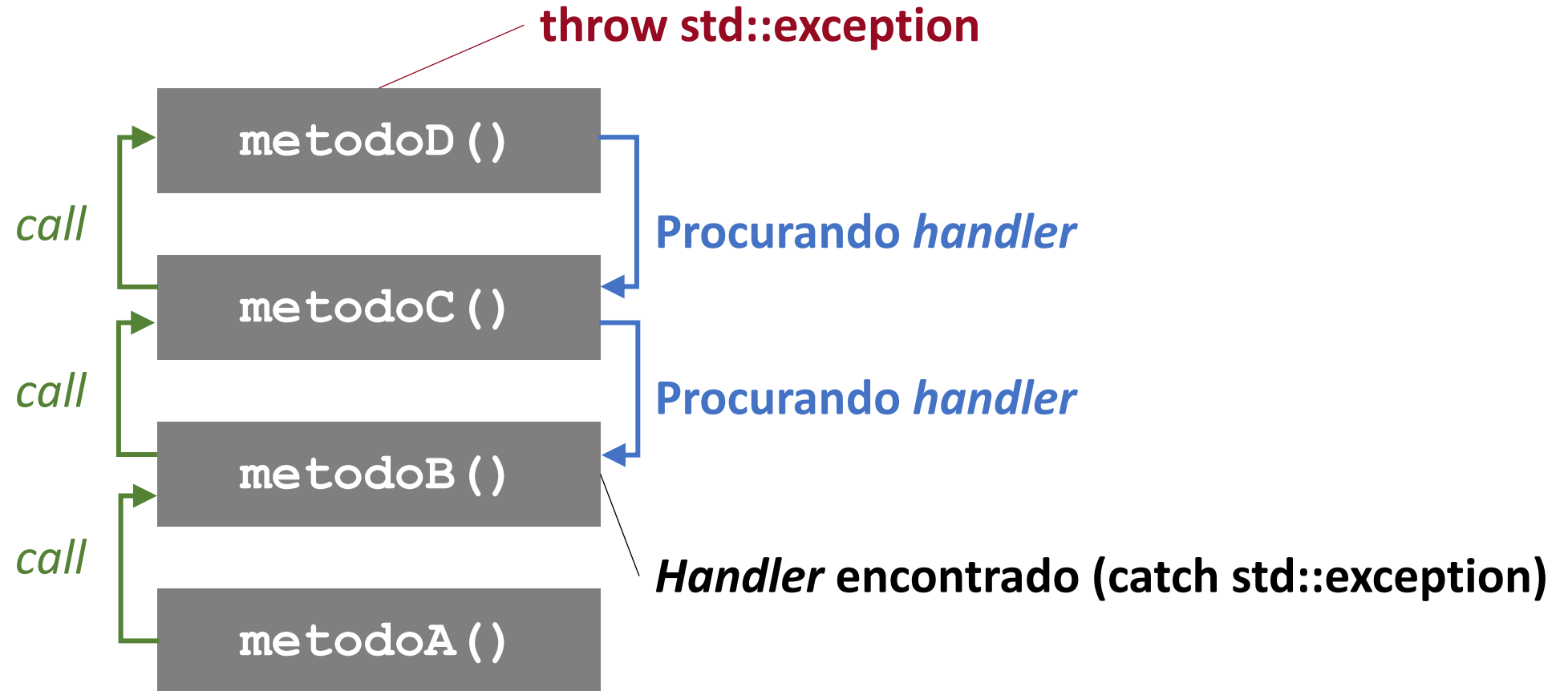
Exceções

- Pilha de execução/chamada
 - Toda invocação é empilhada em uma estrutura que isola a área de memória e armazena informações sobre as sub-rotinas ativas
 - Um dos principais usos é registrar o ponto em que cada sub-rotina deve retornar o controle de execução quando terminar
- Stack Unwinding (desenrolamento da pilha)
 - Procurar primeiro `catch` apropriado para tratar a exceção
 - Variáveis no escopo entre o `throw` e o `catch` são destruídas

<https://docs.microsoft.com/en-us/cpp/cpp/exceptions-and-stack-unwinding-in-cpp?view=vs-2019>

Exceções

Exemplo 3



Exceções

Exemplo 4.1

```
class ClasseB {
public:
    ClasseB() { cout << "Construtor::B" << endl; }
    ~ClasseB() { cout << "Destructor::B" << endl; }
    void metodoB() {
        throw exception();
    }
};

class ClasseA {
public:
    ClasseA() { cout << "Construtor::A" << endl; }
    ~ClasseA() { cout << "Destructor::A" << endl; }
    void metodoA() {
        ClasseB b;
        b.metodoB();
    }
};
```

```
int main() {
    try {
        ClasseA a;
        a.metodoA();
        cout << "rodei?" << endl;
    } catch(exception& e) {
        cout << e.what() << endl;
    }
}
```

Construtor::A
Construtor::B
Destructor::B
Destructor::A
std::exception

Exceções

■ Porque o `Destructor::B` foi chamado?

- <https://timsong-cpp.github.io/cppwp/n3337/except.ctor>
- Quando o controle do fluxo vai do `throw` para o `catch`:
 1. Destrutores dos objetos criados dentro do `try` são chamados
 2. Os destrutores de sub-objetos corretamente instanciados também serão chamados
- Se exceções são lançadas dentro do destrutor:
 - Os outros destrutores das sub-classes não serão chamados!
- Se exceções são lançadas dentro do construtor:
 - Nem o próprio destrutor nem outros destrutores serão chamados!

■ Porque o "rodei?" não foi chamado?

- O fluxo do programa é retornado para o final do `try`, e é capturado pelo `catch`!
 - Todo o que estiver após a chamada `a.metodoA()` não vai ser executado pois foi nessa chamada que a exceção foi lançada

```
int main() {  
    try {  
        ClasseA a;  
        a.metodoA();  
        cout << "rodei?" << endl;  
    } catch(exception& e) {  
        cout << e.what() << endl;  
    }  
}
```

Construtor::A
Construtor::B
Destructor::B
Destructor::A
std::exception

Exceções

Exemplo 4.2 (Exceção no Construtor)

Exceção no construtor!

```
class ClasseB {  
    public:  
        ClasseB() {  
            cout << "Construtor::B" << endl;  
            throw exception();  
        }  
        ~ClasseB() { cout << "Destrutor::B" << endl; }  
        void metodoB() {}  
};
```

```
class ClasseA {  
    public:  
        ClasseA() { cout << "Construtor::A" << endl; }  
        ~ClasseA() { cout << "Destrutor::A" << endl; }  
        void metodoA() {  
            ClasseB b;  
            b.metodoB();  
        }  
};
```

```
int main() {  
    try {  
        ClasseA a;  
        a.metodoA();  
        cout << "rodei?" << endl;  
    } catch(exception& e) {  
        cout << e.what() << endl;  
    }  
}
```

Construtor::A
Construtor::B
Destrutor::A
std::exception

Exceções

- Porque o `Destructor::B` **NAO** foi chamado?
 - O destrutor vai ser chamado só para objetos completamente construídos

```
int main() {  
    try {  
        ClasseA a;  
        a.metodoA();  
        cout << "rodei?" << endl;  
    } catch(exception& e) {  
        cout << e.what() << endl;  
    }  
}
```

Construtor::A
Construtor::B
Destructor::A
std::exception

Exceções

Exemplo 4.3 (Exceção no Construtor do Filho)

```
class Base {
public:
    Base() { cout << "Base()" << endl; }
    ~Base() { cout << "~Base()" << endl; }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived()" << endl;
        throw exception();
    }
    ~Derived() { cout << "~Derived" << endl; }
};

int main() {
    try {
        Derived d1;
        cout << "rodei?" << endl;
    } catch(exception& e) {
        cout << e.what() << endl;
    }
}
```

Exceção no construtor do filho!

```
int main() {
    try {
        Derived d1;
        cout << "rodei?" << endl;
    } catch(exception& e) {
        cout << e.what() << endl;
    }
}
```

Base()
Derived()
~Base()
std::exception

Exceções

- Porque o `~Derived()` **NAO** foi chamado?
 - O objeto `Derived` não foi completamente criado mais o `Base` sim!
 - Lembrando a ordem de chamada dos construtores:
Primeiro são criados os pais, e depois os filhos
 - O construtor e destrutor de `Base` são chamados normalmente

```
int main() {  
    try {  
        Derived d1;  
        cout << "rodei?" << endl;  
    } catch(exception& e) {  
        cout << e.what() << endl;  
    }  
}
```

```
Base()  
Derived()  
~Base()  
std::exception
```

Exceções

Exemplo 4.4 (Exceção no Destrutor)

```
class ClasseB {  
    public:  
        ClasseB() { cout << "Construtor::B" << endl; }  
        ~ClasseB() {  
            cout << "Destrutor::B" << endl;  
            throw exception();  
        }  
        void metodoB() {}  
};
```

```
class ClasseA {  
    public:  
        ClasseA() { cout << "Construtor::A" << endl; }  
        ~ClasseA() { cout << "Destrutor::A" << endl; }  
        void metodoA() {  
            ClasseB b;  
            b.metodoB();  
        }  
};
```

Exceção no destrutor!

```
int main() {  
    try {  
        ClasseA a;  
        a.metodoA();  
        cout << "rodei?" << endl;  
    } catch(exception& e) {  
        cout << e.what() << endl;  
    }  
}
```

```
Construtor::A  
Construtor::B  
Destrutor::B  
libc++abi: terminating ...  
[1] 38889 abort ./main
```

Exceções

- Porque o `Destructor::A()` **NAO** foi chamado?
 - Exceções são lançadas no destrutor quebram o fluxo do *stack unwinding*;
 - O fluxo do programa para, e os destrutores das outras classes não são chamados
 - `std::terminate`
 - A recomendação é que os destrutores capturem e não propagem exceções

```
int main() {  
    try {  
        Derived d1;  
        cout << "rodei?" << endl;  
    } catch(exception& e) {  
        cout << e.what() << endl;  
    }  
}
```

```
Construtor::A  
Construtor::B  
Destructor::B  
libc++abi: terminating ...  
[1] 38889 abort ./main
```

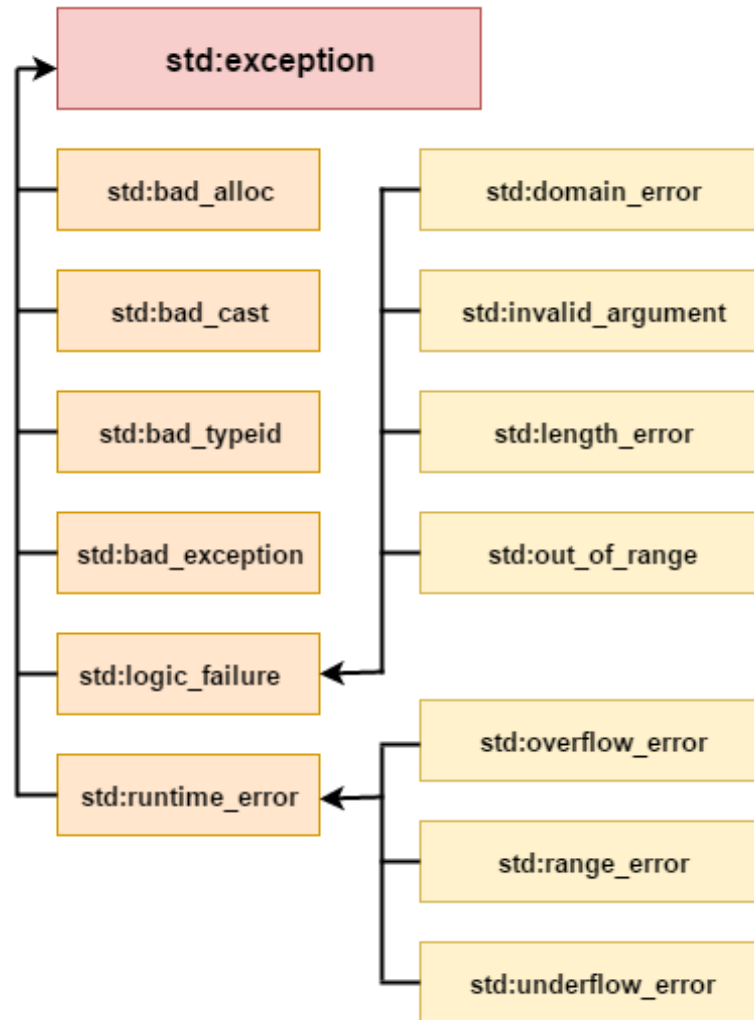
Exceções

- Após a exceção o código é desviado
 - Comandos subsequentes não serão executados
 - Problema em algumas situações
 - Fechamento de uma conexão ou arquivo
- Adicionar região que sempre seria executada
 - Suportado em algumas linguagens (`finally`)
 - C++ não possui essa instrução
 - Responsabilidade do programador e não usuário
 - RAI (STL Vectors, Smart Pointers, etc)

Exceções

- Um método pode lançar diferentes tipos de exceções, e um bloco `try` poderá ter vários blocos `catch` associados à ele
- Bloco selecionado pelo tipo de exceção
 - Será utilizado o primeiro tratador encontrado cujo parâmetro seja do mesmo tipo da exceção (atenção com uso de herança!)
 - É recomendado usar blocos mais específicos
- C++ já possui uma hierarquia de tipos pré-definidos que nós podemos utilizar, mas também podemos implementar outras

Exceções



Exception	Description
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by dynamic_cast .
std::bad_typeid	This exception is generally be thrown by typeid .
std::bad_alloc	This exception is generally be thrown by new .

Exceções

Exemplo 5

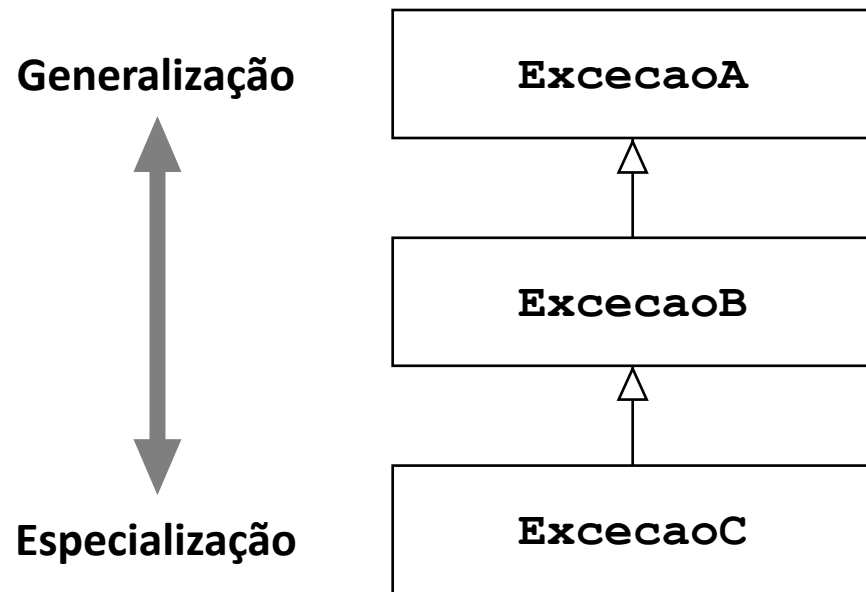
Como fazer o tratamento ao chamar esse código?

```
int factorial(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 * factorial(n-1);  
}
```

[Wandbox](#)

Exceções

- Havendo mais de uma cláusula `catch`, elas devem estar ordenadas do tipo mais específico para o tipo mais genérico (dada hierarquia)



```
{...}  
  
try { ... }  
catch (ExcecaoTipoC& e) { ... }  
catch (ExcecaoTipoB& e) { ... }  
catch (ExcecaoTipoA& e) { ... }  
  
{...}
```

Exceções

- Em C++ as exceções podem ser de qualquer tipo
- É possível criar exceções especializadas usando OO
 - Um novo tipo independente, definido por você
 - Herdar de `std::exception` (ou subclasse)
 - Sempre que possível utilizar as existentes
- Permitem tratamento mais específico
 - Facilitar a identificação dos casos excepcionais
 - Informações necessárias para o tratamento

Exceções

Exemplo 6

```
class ExcecaoSaldoInsuficiente : public std::exception {
public:
    virtual const char* what() const throw() {
        return "Erro: SaldoInsuficiente.";
    }
};

class Conta {
    int _agencia; int _numero; double _saldo = 0;
private:
    bool possuiSaldoSuficiente(double valor) {
        return (_saldo - valor) >= 0;
    }
public:
    void sacar(double valor) {
        if (!possuiSaldoSuficiente(valor)) {
            throw ExcecaoSaldoInsuficiente();
        }
        this->_saldo -= valor;
    }
};
```

Exceções

Exemplo 6

```
int main() {  
  
    try {  
        Conta c;  
        c.sacar(100);  
    } catch (ExcecaoSaldoInsuficiente& e) {  
        std::cout << e.what() << std::endl;  
    }  
  
    return 0;  
}
```

[Wandbox](#)

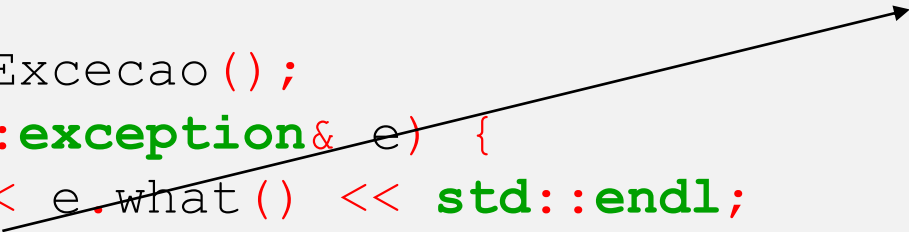
Exceções

Exemplo 7

```
class MinhaExcecao {  
    public:  
        std::string mensagem() {  
            return "Erro: MinhaExcecao.";  
        }  
};
```

```
int main() {  
  
    try {  
        throw MinhaExcecao();  
    } catch (std::exception& e) {  
        std::cout << e.what() << std::endl;  
    } catch (...) {  
        std::cout << "Excecao desconhecida!" << std::endl;  
    }  
  
    return 0;  
}
```

Tratamento para
qualquer tipo de
exceção que
ocorra!



Exceções

- Propagação (rethrow)
 - Não se quer (ou sabe) tratar uma determinada exceção em um escopo (bloco `catch`) ou deseja-se fazer um tratamento parcial
 - “Relançar” a exceção capturada
- Chamar novamente `throw`
 - Não passar parâmetros (mesmo tipo do `catch`)
 - Podem ser passados parâmetros, mas cuidado!
 - Comportamentos inesperados (herança)
 - Novos objetos de exceção gerados (cópia)

Exceções

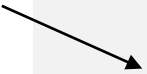
Exemplo 8

```
class MyException : public std::exception {  
    public:  
        virtual const char* what() const throw() {  
            return "Erro: MyException.";  
        }  
};
```

```
void metodoTeste() {  
    try {  
        throw MyException();  
    } catch (MyException& e) {  
        std::cout << "Dentro -> " << e.what() << std::endl;  
        throw;  
    }  
}
```

```
int main() {  
    try {  
        metodoTeste();  
    } catch (MyException& e) {  
        std::cout << "Fora -> " << e.what() << std::endl;  
    }  
    return 0;  
}
```

Propaga a exceção. Pode ser capturada novamente em outro catch.



O que acontece se não for relançada?

Não vai haver exceção a ser tratada no main

Exceções

Especificações de exceções

- Indicam a intenção do programador sobre os tipos das exceções que podem ser lançadas ou não por um método
- Adicionadas às assinaturas dos métodos
 - C++
 - C++11: `throw()` / `throw(type)`
 - C++17: `noexcept(true)` / `noexcept(false)`
 - Java: `throws`
- Em C++ a verificação não é feita em tempo de compilação!
 - Java: Verificadas (compilação) vs. Não Verificadas

Exceções

Especificações de exceções

C++11

```
void f() throw(); // NÃO lança exceções  
void f() throw(std::invalid_argument); // Se lançar, é desse tipo específico
```

>= C++17

```
void f() noexcept(true); // NÃO lança exceções  
void f() noexcept(false); // Pode lançar alguma exceção  
void f() throw(std::invalid_argument); // Se lançar, é desse tipo específico
```

Exceções

Exemplo 8

```
class Conta {  
    int _agencia;  
    int _numero;  
    double _saldo = 0;  
  
    private:  
        bool possuiSaldoSuficiente(double valor) {  
            return (_saldo - valor) > 0;  
        }  
  
    public:  
        void sacar(double valor) throw() {  
            if (!possuiSaldoSuficiente(valor)) {  
                throw ExcecaoSaldoInsuficiente();  
            }  
            this->_saldo -= valor;  
        }  
};
```

Se uma exceção for lançada, o programa encerrará abruptamente com a chamada da função `terminate()`!

Exercício

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

std::string pegar_sub_string(std::string str, int n) {
    return str.substr(n);
}

std::string ler_entrada() {
    std::string texto;
    std::cin >> texto;
    return pegar_sub_string(texto, 10);
}

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

Exercício

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


Exercício

Tarefa: Como fazer para limitar a 3 tentativas incorretas?

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

[Wandbox](#)

Considerações finais

- O tratamento de exceções permite informar, detectar e manipular situações problemáticas de maneira facilitada
- Permitem estruturar o programa pensando no fluxo de execução, evitando o uso e avaliação de condicionais
- Facilita a extensão, manutenção e teste do programa

Considerações finais

- **Problemas** no tratamento de exceções
 - Misturar (ou favorecer) outros tratamentos
 - Utilização de códigos de erros
 - Não entender o processo de stack unwinding
 - Lançar exceções nos destrutores
 - Pode ocasionar a chamada de `terminate()`
 - Não capturar uma exceção por referência
 - *Always throw an exception by value and catch by reference or const reference if possible*