Toda regra tem exceção, exceto a regra que diz que toda regra tem exceção !?

Exceções

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Em caso de erro...

Em programas estruturados (ex.: em C) é comum as funções retornarem algum código de erro em caso de problemas.

Exemplo: `int dividir(int divisor, int dividendo, int* resultado);`

A função pode retornar 0 caso tudo ocorra bem, ou -1 em caso de divisão por zero.

Quais erros o programador pode cometer ao usar uma função assim?

O que se torna estranho nesse tipo de função?

Em caso de erro...

Em programas estruturados (ex.: em C) é comum as funções retornarem algum código de erro em caso de problemas.

Exemplo: `int dividir(int divisor, int dividendo, int* resultado);`

A função pode retornar 0 caso tudo ocorra bem, ou -1 em caso de divisão por zero.

Quais erros o programador pode cometer ao usar uma função assim?

É comum o programador esquecer de verificar o retorno.

O que se torna estranho nesse tipo de função?

No exemplo, o retorno é apenas um flag indicando se tudo correu bem.

O resultado está sendo armazenado em um ponteiro (um tanto confuso).

Exceções

Na orientação a objetos temos o conceito de exceção.

Indicar que algo deu errado.

- **Mecanismo padrão para tratamento de erros.**
  - Criar programas tolerantes a falhas;
  - Detectar uma falha específica e se recuperar;
  - Encontrar a fonte de um problema se torna mais fácil.

Exceções

Uma **exceção** deve ser **lançada** quando um **problema** acontece.

Algo inesperado, ou que não deveria acontecer no fluxo normal do programa.

Em C++, podemos lançar qualquer coisa:

Um inteiro, um char, um objeto, um ponteiro, ...

Veremos que **lançar qualquer coisa é uma má prática.**

Mas por enquanto vamos lançar inteiros como exemplo.

CPF

Abra a classe Pessoa.

O que pode dar errado no setCpf?

CPF

Vamos melhorar a O.O. lançando uma exceção caso um cpf inválido seja passado.

Para **lançar uma exceção**, utilize a palavra-chave `throw`.

Pessoa

Pessoa.cpp

```
void Pessoa::setCpf(const unsigned long cpf){
   if(!validarCPF(cpf))
       throw (int)1;
   this->cpf = cpf;
}
```

Caso cpf inválido, **lance uma exceção** com o número 1.

throw

Quando um `throw` é executado.

O objeto/variável/ponteiro é **imediatamente lançado** para que alguém o pegue.

A função é abortada imediatamente.

- Não termina sua execução;
- Não retorna resultados.

Pegando uma exceção

Para executar um **trecho que pode lançar uma exceção**.

O trecho deve ficar dentro de um bloco `try`.

Após o `try`, adicionamos um bloco `catch`(tipo nomeExceção).

O `catch` "pega" a exceção lançada.

Exemplo

```cpp
#include <iostream>

#include "Pessoa.hpp"

int main(){
   Pessoa *p{nullptr};
   std::string nome;
   unsigned long cpf;

   std::cout << "Digite o nome: ";
   std::cin >> nome;
   std::cout << "Digite o cpf: ";
   std::cin >> cpf;

   try{
       p = new Pessoa{nome};
       p->setCpf(cpf);
       std::cout << p->getNome() << " " << p->getCpf() << "\n";
   }catch(int& ex){
       if(ex == 1){
           std::cout << "CPF inválido\n";
       }else{
           std::cout << "Erro desconhecido\n";
       }
   }
   delete p;
   return 0;
}
```

try-catch

1. O `try` tenta executar o seu bloco;
2. Se nenhuma exceção for lançada, o `try` é executado completamente;
    a. O `catch` não é executado.
3. Se uma exceção é lançada.
    a. As instruções do `try` param de ser executadas imediatamente;
    b. Se o `catch` espera essa exceção.
        i. O `catch` relacionado a exceção é executado.

Construtor de Pessoa

Note o construtor de pessoa.

O que acontece com o `setCpf()` em caso de exceção?

Pessoa.cpp

```
Pessoa::Pessoa(const std::string& nome,
           const unsigned long cpf):Pessoa{nome}{
   this->setCpf(cpf);
}
```

Construtor de Pessoa

Mas e agora, se `setCpf` lançar uma exceção, quem deveria tratá-la?

Em outras palavras, onde deve estar o `try-catch`?

Pessoa.cpp

```
Pessoa::Pessoa(const std::string& nome,
           const unsigned long cpf):Pessoa{nome}{
   this->setCpf(cpf);
}
```

Construtor de Pessoa

Mas e agora, se `setCpf` lançar uma exceção, quem deveria tratá-la?

Depende:

Se o construtor de pessoa puder fazer algo útil quanto a exceção, ele deve pegar a exceção.

Caso contrário, não pegue a exceção.

A exceção será lançada para quem chamou o construtor.

Se ninguém pega a exceção, ela é "lançada para cima", até que alguém na hierarquia de chamadas de funções a pegue.

```
Pessoa::Pessoa(const std::string& nome,
               const unsigned long cpf):Pessoa{nome}{
   this->setCpf(cpf);
}
```

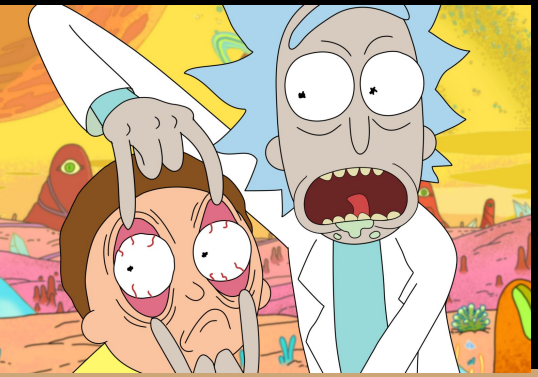
Erro comum

O que está errado?

```
Pessoa::Pessoa(const std::string& nome, const unsigned long cpf)
        : Pessoa(nome) {
    try{
        this->setCpf(cpf);
    }catch(int& ex){
        throw ex;
    }
}
```

Erro comum

- **Não faça isso!**
  - É comum encontrarmos essa construção (errada!!!) em muitos programas.
  - Temos um catch, mas ele não faz nada útil!
    - Pega a exceção, e a relança.
  - O mesmo que se não tivéssemos o `catch`.
- Lançar uma exceção e pegá-la no `catch` custa caro!
- Se o `catch` não faz algo útil, ele não deveria estar ali.



```
Pessoa::Pessoa(const std::string& nome, const unsigned long cpf)
        : Pessoa(nome) {
    try{
        this->setCpf(cpf);
    }catch(int& ex){
        throw ex;
    }
}
```

Não faça.

Exceções e construtores

Quando um construtor lança uma exceção.

Acontece o mesmo que em uma função qualquer.

O construtor é abortado imediatamente.

O objeto não é construído.

Veja essa explicação: https://isocpp.org/wiki/faq/exceptions#ctors-can-throw

Faça você mesmo

```
int main(){
    Pessoa *p{nullptr};
    std::string nome;
    unsigned long cpf;

    std::cout << "Digite o nome: ";
    std::cin >> nome;
    std::cout << "Digite o cpf: ";
    std::cin >> cpf;

    while(p == nullptr){
        try{
            p = new Pessoa{nome, cpf, 18};
            std::cout << p->getNome() << " " << p->getCpf() << "\n";
        }catch(int& ex){
            if(ex == 1){
                std::cout << "CPF inválido, digite outro: ";
                std::cin >> cpf;
            }else{
                std::cout << "Erro desconhecido\n";
            }
        }
    }
    delete p;
    return 0;
}
```

stdexcept

O header `stdexcept` define diversas exceções padrão que podem ser lançadas.

Classes específicas que representam exceções específicas:

www.cplusplus.com/reference/stdexcept

www.cplusplus.com/reference/exception/exception

stdexcept

Boa prática.

- Sempre lance:
  - Uma exceção de stdexcept.
- Ou suas próprias exceções (derivadas de std::exception).
  - **Toda exceção deve derivar (herdar) de std::exception.**

Exemplo

Pessoa.cpp

```cpp
#include<stdexcept>

//...

void Pessoa::setIdade(const unsigned short int novaIdade){
   if(novaIdade > 120)
        throw std::invalid_argument{"Idade Invalida."};
   idade = (unsigned char)novaIdade;
}
```

Exemplo

```cpp
#include<iostream>

#include"Pessoa.hpp"
#include<stdexcept>

int main(){
    Pessoa *p{nullptr};
    std::string nome;
    unsigned long cpf;
    unsigned short int idade;

    std::cout << "Digite o nome: ";
    std::cin >> nome;
    std::cout << "Digite o cpf: ";
    std::cin >> cpf;
    std::cout << "Digite a idade: ";
    std::cin >> idade;
    try{
        p = new Pessoa{nome, cpf, 18};
        p->setIdade(idade);
        std::cout << p->getNome()
            << " " << p->getCpf()
            << " " << p->getIdade() << std::endl;
    }catch(std::invalid_argument& iv){
        std::cout << "Argumento invalido: " << iv.what() << '\n';
    }
    delete p;
    return 0;
}
```

A função ***what*** retorna a string armazenada para a exceção.

Exceções Personalizadas

Vamos criar nossa própria exceção para deixar claros os erros de cpf inválido.

Seguindo as **boas práticas**, a exceção deve derivar de std::exception, ou de alguma classe que derive de std::exception em algum momento na hierarquia.

Pesquise sobre a exceção `runtime_error`.

CPFInvalidoException

CPFInvalidoException.hpp

```hpp
#ifndef CPF_INVALIDO_EXCEPTION
#define CPF_INVALIDO_EXCEPTION

#include <stdexcept>

class CPFInvalidoException : public
std::runtime_error{
   public:
       const unsigned long cpf;

       CPFInvalidoException(const unsigned long cpf);
       virtual ~CPFInvalidoException() = default;
};
#endif
```

CPFInvalidoException.cpp

```cpp
#include "CPFInvalidoException.hpp"


CPFInvalidoException::CPFInvalidoException(
      const unsigned long cpf)
   :std::runtime_error{"CPF Invalido."}, cpf{cpf}{
}
```

Pessoa.cpp

```
void Pessoa::setCpf(unsigned long cpf){
   if(!validarCPF(cpf))
       throw CPFInvalidoException{cpf};
   this->cpf = cpf;
}
```

Múltiplos catches

Podemos ter tantos blocos `catch` quanto necessários!

É permitido ter um bloco `catch` para tratar cada exceção diferente.

```
int main() {

 //...

 try {
   p = new Pessoa{nome, cpf, 18};
   p->setIdade(idade);
   std::cout << p->getNome() << " " << p->getCpf() << " " << p->getIdade() << "\n";
 } catch (std::invalid_argument& iv) {
   std::cout << "Argumento invalido: " << iv.what() << "\n";
 } catch (CPFInvalidoException& cpfe) {
   std::cout << "Erro de CPF: " << cpfe.what() << cpfe.cpf << "\n";
 }

 delete p;
 return 0;
}
```

Múltiplos catches

Podemos ter tantos blocos `catch` quanto necessários!

É permitido ter um bloco `catch` para tratar cada exceção diferente.

Com a Exceção customizada podemos incluir dados relevantes sobre o erro.

```
int main() {

 //...

 try {
   p = new Pessoa{nome, cpf, 18};
   p->setIdade(idade);
   std::cout << p->getNome() << " " << p->getCpf() << " " << p->getIdade() << "\n";
 } catch (std::invalid_argument& iv) {
   std::cout << "Argumento invalido: " << iv.what() << "\n";
 } catch (CPFInvalidoException& cpfe) {
   std::cout << "Erro de CPF: " << cpfe.what() << cpfe.cpf << "\n";
 }

 delete p;
 return 0;
}
```

Lance por valor

Para reduzir o custo computacional e evitar memory leaks, **lance exceções por valor, e pegue por referência**.

```
void Pessoa::setCpf(unsigned long cpf){
   if(!validarCPF(cpf))
       throw CPFInvalidoException{cpf};
   this->cpf = cpf;
}
```

```
int main() {

 //...

 try {
   p = new Pessoa{nome, cpf, 18};
   p->setIdade(idade);
   std::cout << p->getNome() << " " << p->getCpf()
             << " " << p->getIdade() << "\n";
 } catch (std::invalid_argument& iv) {
   std::cout << "Argumento invalido: "
             << iv.what() << "\n";
 } catch (CPFInvalidoException& cpfe) {
   std::cout << "Erro de CPF: " << cpfe.what()
             << cpfe.cpf << "\n";
 }

 delete p;
 return 0;
}
```

Lance por valor

Para reduzir o custo computacional e evitar possíveis memory leaks, **lance exceções por valor, e pegue por referência**.

- Poderíamos ter uma economia de recursos similar alocando a exceção dinamicamente (via new), e lançando o ponteiro da exceção.
  - O catch pega um ponteiro para a exceção.
  - Problemas com essa abordagem?

Lance por valor

Para reduzir o custo computacional e evitar possíveis memory leaks, **lance exceções por valor, e pegue por referência**.

- Poderíamos ter uma economia de recursos similar alocando a exceção dinamicamente (via new), e lançando o ponteiro da exceção.
  - O catch pega um ponteiro para a exceção.
  - Problemas com essa abordagem?
    - Temos agora um objeto alocado dinamicamente
      - De quem é a responsabilidade de remover esse objeto da memória?
    - O que acontece se não cair em nenhum bloco catch?
      - Temos um memory leak?

Exceções e herança

- Podemos colocar um catch para cada exceção específica;
  - Tratamentos diferentes para exceções diferentes
- **Se usarmos as boas práticas.**
  - **Toda exceção deve derivar de `std::except`;**
    - Podemos criar um último catch que pega exceções do tipo `std::except`.
  - Caso seja lançada uma exceção que não esperávamos;
    - Podemos tratar como um erro genérico.
  - Serve como uma **"rede de segurança"** final.

Exceções e Herança

Mesmo se uma exceção inesperada for lançada, ainda somos capazes de identificar que um "erro genérico" aconteceu. **O que aconteceria ao lançar uma exceção que não deriva de std::exception?**

```
int main() {

 //...

 try {
   p = new Pessoa{nome, cpf, idade};
   std::cout << p->getNome() << " " << p->getCpf() << " " << p->getIdade() << "\n";
 } catch (std::invalid_argument &iv) {
   std::cout << "Argumento invalido: " << iv.what() << "\n";
 } catch (CPFInvalidoException &cpfe) {
   std::cout << "Erro de CPF: " << cpfe.what() << cpfe.cpf << "\n";
 } catch (std::exception &ex) {
   std::cout << "Erro Generico: " << ex.what() << "\n";
 }

 delete p;
 return 0;
}
```

O que vai acontecer?

Caso o CPF seja inválido, qual(is) catch(s) será(ão) acionado(s)?

```
try {
    p = new Pessoa{nome, cpf};
    p->setIdade(idade);
    std::cout << p->getNome() << " " << p->getCpf() << " " << p->getIdade() << "\n";
} catch (std::exception& ex) {
    std::cout << "Ocorreu um erro generico " << ex.what() << "\n";
} catch (std::invalid_argument& iv) {
    std::cout << "Argumento inválido: " << iv.what() << "\n";
} catch (CPFInvalidoException& ci) {
    std::cout << "Erro de CPF: " << ci.what() << "CPF incorreto: " << ci.cpf << "\n";
}
```

O que vai acontecer?

Caso o CPF seja inválido, qual(is) catch(s) será(ão) acionado(s)?

A sua exceção sempre vai ser pega pelo **primeiro catch compatível.**

Os dois últimos catchs do exemplo nunca serão acionados.

```
try {
    p = new Pessoa{nome, cpf};
    p->setIdade(idade);
    std::cout << p->getNome() << " " << p->getCpf() << " " << p->getIdade() << "\n";
} catch (std::exception& ex) {
    std::cout << "Ocorreu um erro generico " << ex.what() << "\n";
} catch (std::invalid_argument& iv) {
    std::cout << "Argumento inválido: " << iv.what() << "\n";
} catch (CPFInvalidoException& ci) {
    std::cout << "Erro de CPF: " << ci.what() << "CPF incorreto: " << ci.cpf << "\n";
}
```

Questão de ordem

Os `catches` precisam estar **em ordem**.

Sempre coloque os **`catches` mais específicos primeiro.**

O `catch` que pega `std::exception` (se existir) sempre deve **ficar por último.**

Prevenindo erros

Não lance exceções em um construtor que está sendo usado para criar um objeto estático ou global.

Esses objetos são construídos antes do main executar.

Como pegar essas exceções?

Dica: quando necessário, crie um construtor específico que será invocado apenas para criar objetos estáticos/globais.

Memória dinamicamente alocada

Se o seu construtor aloca algo de maneira dinâmica.

Antes de lançar qualquer exceção libere a memória alocada.

Esquecer disso ou fazer de maneira incorreta é **uma das principais causas de resources leaks.**

noexcept

A partir do C++11, **uma função pode informar que ela não lança exceção alguma**, e que ela não chama outras funções que podem lançar exceções.

Para isso, adicione `noexcept` no final da declaração (.hpp) e implementação (.cpp) da função.

Caso a função seja `const`, o `noexcept` deve ficar logo após o `const`.

Uma função `noexcept` que lança uma exceção faz com que o **programa termine.**

Note que esse é um comportamento contrário do Java, que obriga ao programador informar que a função lança uma exceção.

Operador new

O comportamento padrão do operador `new` em C++ é lançar uma exceção `bad_alloc` em caso de problemas.

Destrutores

Destrutores nunca devem lançar exceções.

Por quê?

Destrutores

Destrutores nunca devem lançar exceções.

Lembre-se que ao lançar uma exceção, a execução da função é imediatamente abortada.

Como abortar a execução de um destrutor?

O destrutor executa pela metade? Como fica a memória?

Lançar uma exceção no destrutor pode levar ao término do seu programa, ou a comportamentos indefinidos.

Isso geraria problemas no stack frame.

Veja https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw

Exceção não é If-else

Exceções não são `if-else`.

> Erro grave que encontro em vários programas.
>
> Além de ser uma péssima prática que pode levar a erros difíceis de tratar, é altamente ineficiente.

```
try{
      escrever_no_arquivo
}catch (ArquivoFechado e){
      abrir_arquivo
      escrever_no_arquivo
}
```

Errado. Se faz parte do fluxo normal do programa chegar neste trecho com o arquivo fechado, trate com if else.

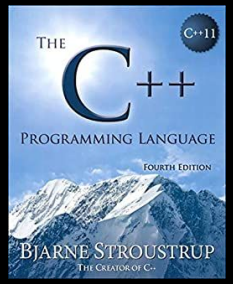
```
try{
      if(arquivo_fechado)
            abrir_arquivo
      escrever_no_arquivo
}catch (Exception e){
      //tratar algo inesperado aqui, como impossibilidade
      //de escrever no arquivo (por exemplo, o disco
      //estava cheio).
}
```

Exercícios

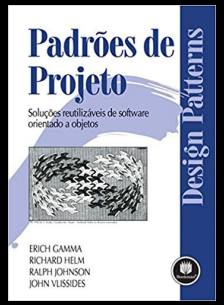
1. Localize os demais trechos do programa onde exceções podem ser necessárias, e as lance.
    a. Você deve criar pelo menos uma exceção customizada;
    b. Você deve adicionar try-catchs no main, ou em outro lugar onde isso fizer sentido;

Referências

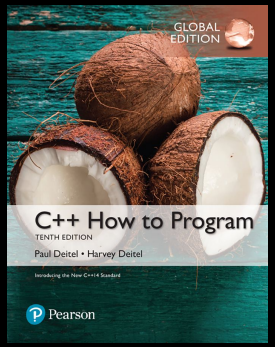
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



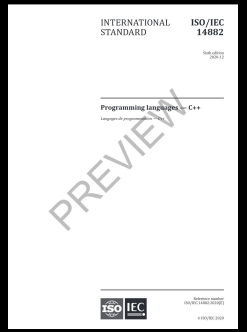
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



Deitel, H. M., Deitel, P. J. C++: como programar. 10a ed. Pearson Prentice Hall. 2017.



ISO/IEC 14882:2020 Programming languages - C++: www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en



Licença