Exceções – Parte 3

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Lance por valor

- Para reduzir a carga computacional, e evitar possíveis problemas de memory leak, **lance exceções por valor**, e **pegue as exceções por referência**

Lance por valor

- Para reduzir a carga computacional, e evitar possíveis problemas de memory leak, **lance exceções por valor**, e **pegue as exceções por referência**
  - Exemplo:

Lançamento por valor

```
void Pessoa::setIdade(const unsigned short int idade){
        if(idade > 150)
                throw std::invalid_argument{"Idade Invalida."};
        this->idade =  idade;
}
```

catch por referência para evitar a cópia da exceção na memória

```
try{
        p = new Pessoa{nome, cpf};
        p->setIdade(idade);
        std::cout << p->getNome()
                << " " << p->getCpf()
                << " " << p->getIdade() << std::endl;
}catch(std::invalid_argument& iv){
        std::cout << "Argumento inválido: " <<  iv.what() << std::endl;
}
```

UDESC

Lance por valor

- Para reduzir a carga computacional, e evitar possíveis problemas de memory leak, **lance exceções por valor**, e **pegue as exceções por referência**
  - Poderíamos ter uma economia de recursos similar se alocássemos a exceção dinamicamente, e lançássemos o ponteiro da exceção
    - O *catch* pega um ponteiro para a exceção
    - Problemas com essa abordagem?

Lance por valor

- Para reduzir a carga computacional, e evitar possíveis problemas de memory leak, **lance exceções por valor**, e **pegue as exceções por referência**
  - Poderíamos ter uma economia de recursos similar se alocássemos a exceção dinamicamente, e lançássemos 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

- Aprendemos que podemos colocar um *catch* para cada exceção específica
  - Realizamos 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
    - Essa exceção ainda deve derivar de std::except
    - Podemos tratar como um erro genérico
    - Serve como uma “rede de segurança” final

Exceções e herança

Se dentro do *try* uma exceção “desconhecida” for lançada, ela vai cair nesse último *catch*. Essa é a importância de se seguir as boas práticas – o que acontece se sua exceção não derivar de std::exception?

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

UDESC

O que vai acontecer?

- Caso o CPF digitado 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() << std::endl;
}catch(std::exception& ex){
        std::cout << "Ocorreu um erro generico " << ex.what() << std::endl;
}catch(std::invalid_argument& iv){
        std::cout << "Argumento inválido: " <<  iv.what() << std::endl;
}catch(CPFInvalidoException& ci){
        std::cout << "Erro de CPF: " <<  ci.what() << "CPF incorreto: " << ci.cpf << std::endl;
}
```

UDESC

O que vai acontecer?

- Caso o CPF digitado seja inválido, qual(is) catch(s) será(ão) acionado(s)?
  - A sua exceção sempre vai ser pega **pelo primeiro *catch* compatível**
  - Dessa forma 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() << std::endl;
}catch(std::exception& ex){
        std::cout << "Ocorreu um erro generico " << ex.what() << std::endl;
}catch(std::invalid_argument& iv){
        std::cout << "Argumento inválido: " <<  iv.what() << std::endl;
}catch(CPFInvalidoException& ci){
        std::cout << "Erro de CPF: " <<  ci.what() << "CPF incorreto: " << ci.cpf << std::endl;
}
```

UDESC

O que vai acontecer?

O próprio g++ identifica a besteira que estamos fazendo nesse caso. Veja a saída:

```
...
main.cpp: In function ‘int main()’:
main.cpp:29:3: warning: exception of type ‘std::invalid_argument’ will be caught
  }catch(std::invalid_argument& iv){
   ^~~~~
main.cpp:27:3: warning:    by earlier handler for ‘std::exception’
  }catch(std::exception& ex){
   ^~~~~
...
```

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

UDESC

Ordem

- Os catchs precisam **estar em ordem**
  - Sempre coloque os catch 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 (*static*) ou um objeto global
  - Por quê?

Prevenindo erros

- Não lance exceções em um construtor que está sendo usado para criar um objeto estático (*static*) ou um objeto global
  - Esses objetos são construídos antes do main executar
    - Não podemos “pegar” essas exceções
  - Dica: quando necessário, crie um construtor específico que será invocado apenas para criar objetos estáticos ou globais da sua classe
    - Esse construtor em específico não deve lançar exceções

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***
  - Uma forma de prevenir isso é usar um **unique_ptr**
    - Disponível a partir do C++11

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 declarada como noexcept que por algum motivo lança uma exceção faz com que o programa termine
- Note que esse é um comportamento justamente contrário ao do Java, que obriga ao programador informar que a função **lança** uma exceção

O operador new

- Como sabemos se um malloc em C executou corretamente ou não?

O operador new

- Como sabemos se um malloc em C executou corretamente ou não?
  - Em caso de erro, NULL é retornado
- O comportamento padrão do operador ***new*** em C++ é lançar uma exceção do tipo ***bad_alloc*** em caso de problemas

Exceções e Destrutores

- **Destrutores nunca devem lançar exceções**
  - Por quê?

Exceções e 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? Isso pode corromper a memória!!!
    - Lançar uma exceção no destrutor pode levar ao término do seu programa, ou a comportamentos indefinidos

Referências

- DEITEL, P.; DEITEL, H. **C++ how to Program**. [S.l.]: Pearson, 2017. ISBN 9780134448237
- STROUSTRUP, B. **The C++ Programming Language**. Pearson Education, 2013. ISBN 9780133522853.
- https://isocpp.org