Destrutores e Agregações

Paulo Ricardo Lisboa de Almeida

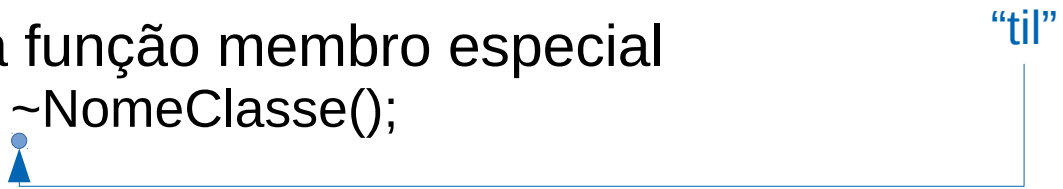
UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Destrutor

- Em C++, classes possuem um ou mais construtores e **um destrutor**
  - Toda classe possui **um, e apenas um, destrutor**
  - Se você não definiu um destrutor para sua classe, um destrutor *default* será injetado automaticamente pelo compilador
    - Um destrutor *default* não realiza tarefa alguma

Destrutor

- O destrutor é uma função membro especial
  - Seu protótipo é ~NomeClasse();
  - Obs.: o ~ representa o operador binário de negação em C/C++
    - O destrutor é a “negação” de um construtor
- Um destrutor **não possui** tipo de retorno
- Um destrutor **não possui** parâmetros
- Um destrutor **não deve lançar exceções**
  - Diferente de um construtor, que pode lançar exceções
  - Veremos exceções no futuro



Destrutor

- O destrutor é chamado logo antes do objeto ser removido da memória
  - Exemplos
    - Quando o objeto sai do escopo em que foi criado
      - Quando o objeto não foi alocado dinamicamente e chega no final do bloco { ... } em que ele foi definido
    - Quando o ***delete*** é chamado para o objeto
- Invocar as funções *exit* ou *abort* forçam o programa a terminar imediatamente
  - Nesse caso, nenhum destrutor é invocado

Destrutor

- Um destrutor **não remove** o objeto em si da memória
  - A função do destrutor é
    - Realizar limpezas internas do objeto
      - Exemplo: remover os dados alocados dinamicamente pelo objeto
    - Executar quaisquer lógicas referentes ao fim da vida do objeto
      - Exemplo: no fim da vida de um objeto que representa um arquivo, devemos forçar a gravação do *buffer* de dados no arquivo (*flush*) e fechar o ponteiro de arquivo

Exemplo

- Vamos criar um destrutor para a classe Disciplina

Disciplina.hpp

```
//...
class Disciplina{
        public:
                Disciplina(std::string nome);
                ~Disciplina();

                //...
        private:
                std::string nome;
                unsigned short int cargaHoraria;
                SalaAula* salaAula;

                Pessoa* professor;
                std::list<Pessoa*> alunos;
                std::list<ConteudoMinistrado*> conteudos;
};
#endif
```

UDESC

Disciplina.cpp

```cpp
#include "Disciplina.hpp"

#include<iostream>

#include "SalaAula.hpp"

Disciplina::Disciplina(std::string nome)
    :nome{nome}, cargaHoraria{0}, salaAula{nullptr}, professor{nullptr} {
}

Disciplina::~Disciplina(){
    std::cout << "Destruindo disciplina " << this->nome << std::endl;
}

//...
```

Teste você mesmo

```cpp
#include <iostream>

#include "Pessoa.hpp"
#include "Disciplina.hpp"

int main(){
	Pessoa prof1{"João", 40};
	Disciplina dis1{"C++"};
	dis1.setProfessor(&prof1);

	dis1.adicionarConteudoMinistrado("Ponteiros", 4);
	dis1.adicionarConteudoMinistrado("Referencias", 2);

	Disciplina dis2{"Java"};
	Disciplina* dis3{new Disciplina{"C#"}};
	delete dis3;

	std::cout << "Fim do Programa" << std::endl;

	return 0;
}
```

Tarefas do destrutor

- Quais tarefas deveriam ser executadas pelo destrutor de Disciplina?

Tarefas do destrutor

- Quais tarefas deveriam ser executadas pelo destrutor de Disciplina?
  - Apagar os conteúdos ministrados da memória
    - Os conteúdos foram alocados internamente nos objetos de Disciplina
    - Não faz sentido um conteúdo continuar existindo quando a disciplina deixa de existir

Tarefas do destrutor

- Quais tarefas deveriam ser executadas pelo destrutor de Disciplina?
  - Apagar os conteúdos ministrados da memória
    - Os conteúdos foram alocados internamente nos objetos de Disciplina
    - Não faz sentido um conteúdo continuar existindo quando a disciplina deixa de existir
  - Remover a disciplina sendo destruída da sala de aula
    - Lembre-se que o relacionamento Disciplina – SalaAula é uma associação bidirecional, onde ambos se conhecem
    - É prudente solicitar a remoção da disciplina sendo destruída da sala de aula

Tarefas do destrutor

- Apesar de ser uma boa prática e blá blá blá, sair avisando a todos que o objeto foi destruído
  - Não é viável → não sabemos todos os objetos que usam os objetos da nossa classe (nem todo relacionamento é bidirecional)
  - Pode custar caro em alguns cenários!
    - Nesses casos talvez seja mais eficiente deixar para o programador gerenciar as relações
- Dessa forma, criar um destrutor **não é trivial**
  - Na verdade, é uma das tarefas mais difíceis de serem realizas
    - Fizemos a limpeza de tudo que precisávamos?
    - A ordem da limpeza está correta?
    - Avisamos a todos que deveríamos sobre a destruição do objeto?
    - Fechamos os recursos abertos (ex.: arquivos utilizados pela classe)?
    - …
  - Grande parte dos problemas de *memory leak* em programas C++ são causados por destrutores criados incorretamente

Disciplina.cpp

```
//...
Disciplina::~Disciplina(){
	//o setSalaAula vai remover a disciplina da sala de aula antiga, caso ela exista
	if(this->salaAula != nullptr)//se já existia uma sala, remover a disciplina dessa sala
		this->salaAula->removerDisciplina(this);
	std::list<ConteudoMinistrado*>::iterator it;
	for(it=conteudos.begin(); it!=conteudos.end(); it++)
		delete *it;//liberando a memória de cada conteúdo
}
//...
```

Teste no main

Faça um main **exatamente** como o ao lado.

make clean
make

Execute
O que acontece?

UDESC

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


#include "Disciplina.hpp"
#include "SalaAula.hpp"
#include "ConteudoMinistrado.hpp"


int main(){
        Disciplina dis1{"C++"};
        Disciplina* dis2{new Disciplina{"Java"}};
        SalaAula sala{"F203", 40};

        dis1.setSalaAula(&sala);
        dis2->setSalaAula(&sala);


        std::list<Disciplina*> disSala = sala.getDisciplinas();
        std::list<Disciplina*>::iterator it;
        for(it = disSala.begin(); it != disSala.end(); it++)
                std::cout << (*it)->getNome() << std::endl;


        delete dis2;


        std::cout << "Fim do Programa" << std::endl;



        return 0;
}
```

Falha de segmentação

- Ocorreu uma falha de segmentação
- Mas como???
  - Aparentemente tudo está certo
- Você consegue identificar o problema?

cout vs. cerr

- **cout** escreve na **saída padrão** do sistema
  - A saída padrão geralmente possui um buffer
  - O Sistema Operacional espera juntar dados o suficiente no buffer antes de realmente escrever os dados
    + Economiza Processamento
    - As saídas podem não estar sincronizadas com o código
      - Quando o programa se quebra, as últimas coisa que você escreveu podem não aparecer
- **cerr** escreve na **saída padrão de erros** do sistema
  - Não possui buffers
    + Escreve assim que solicitamos
    - Gasta mais recursos
- **Atenção**
  - Escreva em **cerr**
    - Os erros do seu programa
    - Ou, em nosso caso, as saídas de *debug* para entendermos melhor o que está acontecendo

Entendendo o erro

- No destrutor de *Disciplina*
  - Escreva na tela através de cerr
    - Use como cout
      std::cerr <<
    - “Destruindo a disciplina NOME_DISCIPLINA”
- Crie um destrutor para *SalaAula*
  - A única coisa que o destrutor de *SalaAula* faz é imprimir através de cerr
  - “Destruindo a Sala de Aula NOME_SALA”

Entendendo o erro

No console, é exibido:

C++
Java
Destruindo a disciplina Java
Fim do Programa
Destruindo a sala F203
Destruindo a disciplina C++
Falha de segmentação (imagem do núcleo gravada)

E agora, você consegue identificar o problema?

UDESC

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

#include "Disciplina.hpp"
#include "SalaAula.hpp"
#include "ConteudoMinistrado.hpp"

int main(){
        Disciplina dis1{"C++"};
        Disciplina* dis2{new Disciplina{"Java"}};
        SalaAula sala{"F203", 40};

        dis1.setSalaAula(&sala);
        dis2->setSalaAula(&sala);

        std::list<Disciplina*> disSala = sala.getDisciplinas();
        std::list<Disciplina*>::iterator it;
        for(it = disSala.begin(); it != disSala.end(); it++)
                std::cout << (*it)->getNome() << std::endl;

        delete dis2;

        std::cout << "Fim do Programa" << std::endl;

        return 0;
}
```

Entendendo o erro

...
Destruindo a disciplina Java
Fim do Programa
Destruindo a sala F203
Destruindo a disciplina C++

- Primeiro a disciplina Java é destruída (via delete)
  - A disciplina se remove automaticamente da Sala F203
- O *main* continua até o final do seu escopo
- Quando o escopo do main termina, suas variáveis e objetos locais são destruídos
  - A sala F203 é destruída primeiro
    - O destrutor da sala não faz nada de útil
  - Depois, a disciplina C++ é destruída
    - O destrutor remove a disciplina da Sala F203
      - **Mas a sala F203 não existe mais na memória!!!**

Corrigindo

- Como corrigir o problema?

Corrigindo

- Como corrigir o problema?
  - O destrutor de sala de aula deveria informar suas disciplinas que a sala não existe mais
    - Fazer um *disciplina→setSalaAula(nullptr)* para cada disciplina
      - Indicando que a disciplina ficou sem sala
- **Fica como exercício a correção**
  - Note no exemplo disponibilizado (“ProgramasAula16”) que para fazer isso funcionar, *void Disciplina::setSalaAulaSemAtualizarSala* precisou ser criada
    - Evitar que o iterador do destrutor de SalaAula fosse invalidado
      - Poderíamos criar uma cópia da lista, mas **custaria caro**
    - Uma solução ainda melhor, e ainda computacionalmente eficiente, seria *Disciplina::setSalaAulaSemAtualizarSala* ser **privada**, e **somente SalaAula pudesse acessar essa função** (função amiga)

Agregação forte

- O que criamos entre *Disciplina* e *ConteudoMinistrado* é uma **agregação forte ou composição**
- Em uma agregação forte (composição)
  - O membro parte ajuda a compor o objeto
    - A disciplina é composta de diversos conteúdos ministrados
  - O membro parte pode pertencer a apenas um objeto
    - O conteúdo é ministrado em apenas uma disciplina
  - A existência do membro é gerenciada pelo objeto
    - A disciplina se encarrega de criar e destruir os objetos ConteudoMinistrado
  - Os objetos membro não existem sem o objeto principal
    - Não faz sentido os conteúdos ministrados continuarem existindo quando a disciplina deixa de existir
      - Por isso destruímos via o destrutor de disciplina

Exercícios

1. Estude em detalhes sobre agregações fortes e fracas

2. Corrija a falha de segmentação causada pela implementação incorreta do destrutor de SalaAula.
   - Em caso de problemas, veja o exemplo disponibilizado em "ProgramasAula16"

3. Crie destrutores para todas as classes criadas até agora
   - Mesmo que a maioria dos destrutores não façam trabalho algum

4. Atualize o diagrama de classes UML
   - Represente a **agregação forte** entre *Disciplina* e *ConteudoMinistrado*
     - Não se esqueça de representar as demais Classes e Relações
   - Adicione os destrutores

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.