"Run to the Hills" (Maiden, I.).

Destrutores e Agregações

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Destrutor

Em C++, classes possuem um ou mais construtores e **um destrutor**.

Toda classe possui um, e apenas um, destrutor.

Se você não definir um destrutor, um destrutor padrão (default) será injetado automaticamente pelo compilador.

Um destrutor padrão não realiza tarefa alguma.

Destrutor

O destrutor é uma função membro especial.

Seu protótipo é `~NomeClasse();`

Obs.: o ~ (til) 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.

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.

- Nenhum destrutor é invocado.

Destrutor

O destrutor **não remove o objeto em si da memória.**

Algumas funções do destrutor:

- 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

```
class Disciplina{
   public:
   Disciplina(std::string nome);
       Disciplina(std::string nome, SalaAula* sala);

       ~Disciplina();


    //...

   private:
       std::string nome;
       unsigned short int cargaHoraria;
       Pessoa* professor;
       SalaAula* sala;

       std::list<ConteudoMinistrado*> conteudos;
};
```

```
#include "Disciplina.hpp"

#include <iostream>

#include "SalaAula.hpp"

Disciplina::Disciplina(std::string nome)
   :nome{nome},sala{nullptr} {
}

Disciplina::Disciplina(std::string nome,
           SalaAula* sala)
           :Disciplina{nome} {
   this->setSalaAula(sala);
}

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

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.
- Remover a disciplina sendo destruída da sala de aula.
  - Lembre-se que o relacionamento Disciplina – SalaAula é uma associação bidirecional.
  - A sala de aula vai apontar para uma disciplina que não existe se não fizermos isso.

Tarefas do Destrutor

Apesar de ser uma boa prática e blá blá blá, sair avisando a todos que o objeto foi destruído muitas vezes.

- 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.

Tarefas do Destrutor

Criar um destrutor não é trivial.

- Na verdade, é uma das tarefas mais difíceis de serem realizadas.

Tarefas do Destrutor

Criar um destrutor não é trivial.

- Na verdade, é uma das tarefas mais difíceis de serem realizadas.
  - Fizemos a limpeza de tudo que precisávamos?

Tarefas do Destrutor

Criar um destrutor não é trivial.

- Na verdade, é uma das tarefas mais difíceis de serem realizadas.
  - Fizemos a limpeza de tudo que precisávamos?
  - A ordem da limpeza está correta?

Tarefas do Destrutor

Criar um destrutor não é trivial.

- Na verdade, é uma das tarefas mais difíceis de serem realizadas.
  - Fizemos a limpeza de tudo que precisávamos?
  - A ordem da limpeza está correta?
  - Avisamos a todos que deveríamos sobre a destruição do objeto?

Tarefas do Destrutor

Criar um destrutor não é trivial.

- Na verdade, é uma das tarefas mais difíceis de serem realizadas.
  - 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)?

Tarefas do Destrutor

Criar um destrutor não é trivial.

- Na verdade, é uma das tarefas mais difíceis de serem realizadas.
  - 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 incorretos.

Disciplina.cpp

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

void Disciplina::setSalaAula(SalaAula* sala){
    if(this->sala != nullptr)//se já existia uma sala, remover a disciplina dessa sala
        this->sala->disciplinasMinistradas.remove(this);
    this->sala = sala;
    if(this->sala != nullptr)
        this->sala->disciplinasMinistradas.push_back(this);//adicionar a disciplina na nova sala
}
```

Faça você mesmo

Faça um main exatamente como o ao lado.

`make clean`

`make`

Execute.

O que acontece?

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

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

int main(){
   Disciplina dis1{"C++", nullptr};
   Disciplina* dis2{new Disciplina{"Java", nullptr}};

   SalaAula sala{"Lab Info 1", 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;
}
```

Faça você mesmo

Ocorreu uma falha de segmentação.

Mas como???

Você consegue identificar o problema?

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

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

int main(){
   Disciplina dis1{"C++", nullptr};
   Disciplina* dis2{new Disciplina{"Java", nullptr}};

   SalaAula sala{"Lab Info 1", 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;
}
```

cout vs. cerr

- cout escreve na saída padrão do sistema.
  - A saída padrão geralmente possui um buffer;
  - O Sistema Operacional espera acumular dados o suficiente no buffer antes de realmente escrever os dados.
    + Economiza Processamento;
    - As saídas podem não estar sincronizadas com a execução do programa.
- 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;
  - E nada mais.

cout vs. cerr

Dica: std::endl injeta um \n e automaticamente faz um flush da stream armazenada no buffer.

- cout escreve na saída padrão do sistema.
  - A saída padrão geralmente possui um buffer;
  - O Sistema Operacional espera acumular dados o suficiente no buffer antes de realmente escrever os dados.
    - \+ Economiza Processamento;
    - \- As saídas podem não estar sincronizadas com a execução do programa.
- 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;
  - E nada mais.

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”.

Faça você mesmo

No console é exibido:

```
C++
Java
Destruindo disciplina Java
Fim do Programa
Destruindo Sala Lab Info 1
Destruindo disciplina C++
Abortado (imagem do núcleo gravada)
```

E agora, você consegue identificar o problema?

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

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

int main(){
   Disciplina dis1{"C++", nullptr};
   Disciplina* dis2{new Disciplina{"Java", nullptr}};

   SalaAula sala{"Lab Info 1", 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;
}
```

Faça você mesmo

Primeiro a disciplina Java é destruída (via delete).

A disciplina se remove automaticamente da Sala “Lab 1”.

```
int main(){
    Disciplina dis1{"C++", nullptr};
    Disciplina* dis2{new Disciplina{"Java", nullptr}};

    SalaAula sala{"Lab Info 1", 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;
}
```

```
...
Destruindo disciplina Java
Fim do Programa
Destruindo Sala Lab Info 1
Destruindo disciplina C++
Abortado (imagem do núcleo gravada)
```

Faça você mesmo

Primeiro a disciplina Java é destruída (via delete).

A disciplina se remove automaticamente da Sala “Lab 1”.

O main continua até o final do seu escopo. Quando o escopo do main termina, suas variáveis e objetos locais são destruídos.

```
...
Destruindo disciplina Java
Fim do Programa
Destruindo Sala Lab Info 1
Destruindo disciplina C++
Abortado (imagem do núcleo gravada)
```

```cpp
int main(){
    Disciplina dis1{"C++", nullptr};
    Disciplina* dis2{new Disciplina{"Java", nullptr}};

    SalaAula sala{"Lab Info 1", 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;
}
```

Faça você mesmo

Primeiro a disciplina Java é destruída (via delete).

A disciplina se remove automaticamente da Sala “Lab 1”.

O main continua até o final do seu escopo. Quando o escopo do main termina, suas variáveis e objetos locais são destruídos.

A sala “Lab 1” é destruída primeiro.

O destrutor da sala não faz nada útil.

```
int main(){
    Disciplina dis1{"C++", nullptr};
    Disciplina* dis2{new Disciplina{"Java", nullptr}};

    SalaAula sala{"Lab Info 1", 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;
}
```

```
...
Destruindo disciplina Java
Fim do Programa
Destruindo Sala Lab Info 1
Destruindo disciplina C++
Abortado (imagem do núcleo gravada)
```

Faça você mesmo

Primeiro a disciplina Java é destruída (via delete).

A disciplina se remove automaticamente da Sala “Lab 1”.

O main continua até o final do seu escopo. Quando o escopo do main termina, suas variáveis e objetos locais são destruídos.

A sala “Lab 1” é destruída primeiro.

O destrutor da sala não faz nada útil.

Depois, a disciplina C++ é destruída.

O destrutor remove a disciplina da Sala “Lab 1”.

Mas a sala “Lab 1” já não existe na memória!!!

```
int main(){
    Disciplina dis1{"C++", nullptr};
    Disciplina* dis2{new Disciplina{"Java", nullptr}};

    SalaAula sala{"Lab Info 1", 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;
}
```

```
...
Destruindo disciplina Java
Fim do Programa
Destruindo Sala Lab Info 1
Destruindo disciplina C++
Abortado (imagem do núcleo gravada)
```

Corrigindo

Como corrigir o problema?

Corrigindo

O destrutor de SalaAula deveria informar suas disciplinas que a sala não existe mais.

Fica como exercício a correção.

Agregação Forte

O que criamos entre `Disciplina` e `ConteudoMinistrado` é uma **agregação forte** ou **composição**.

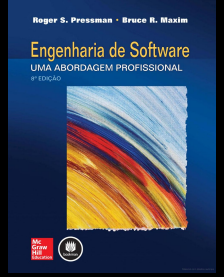
- Em uma agregação forte (composição):
  - O membro parte ajuda a compor o objeto.
    - A disciplina é composta de diversos conteúdos ministrados.
- O membro parte pode pertencer a apenas um objeto.
  - O conteúdo é ministrado em apenas uma disciplina.
    - A existência do membro é gerenciada pelo 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

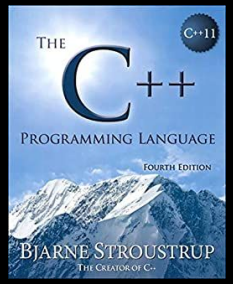
1. Estude em detalhes sobre agregações fortes e fracas.
2. Corrija a falha de segmentação causada pela implementação incorreta do destrutor de `SalaAula`.
3. Crie destrutores para todas as classes criadas até agora.
    a. Mesmo que a maioria dos destrutores não façam trabalho algum.
4. Atualize o diagrama de classes UML.
    a. Represente a agregação forte entre `Disciplina` e `ConteudoMinistrado`.
    b. Não se esqueça de representar as demais Classes e Relações.
    c. Dica: Geralmente não adicionamos destrutores no diagrama de classes.
        i. O destrutor é sempre padrão, sem parâmetros, e não adiciona informação útil ao modelo, além de ser algo que algumas linguagens O.O. não possuem.
        ii. Você pode adicionar, mas precisa de um bom motivo para isso

Referências

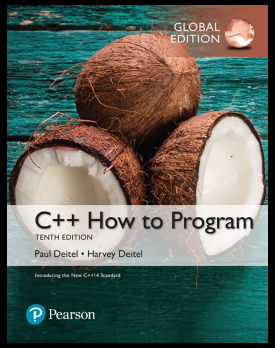
Engenharia de Software - 8ª Edição. McGraw Hill Brasil. 2016.



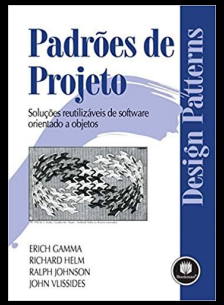
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



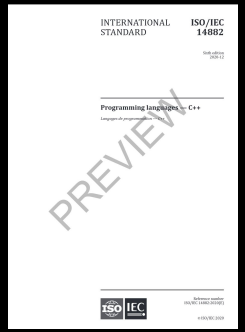
Deitel, H. M., Deitel, P. J. C++: como programar. 10a ed. Pearson Prentice Hall. 2017.



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



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



Licença