"Todos os problemas da humanidade decorrem da incapacidade das pessoas se sentarem quietas em uma sala" (Blaise Pascal).

Smart Pointers

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Pense...

O que pode dar errado no trecho a seguir?

```
#include <iostream>

#include "Professor.hpp"

void funcao(){
   ufpr::Professor* prof{new ufpr::Professor{"Joao",
      11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  delete prof;
}

int main() {
   try{
       funcao();
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

O Problema...

Se uma exceção for lançada depois da criação de `prof`, mas antes do `delete`, temos um **memory leak**!

O delete nunca será chamado.

```
#include <iostream>

#include "Professor.hpp"

void funcao(){
   ufpr::Professor* prof{new ufpr::Professor{"Joao",
      11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  delete prof;
}

int main() {
   try{
       funcao();
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Resolvendo

Uma solução é envolver tudo que está na função em um bloco `try-catch`.

No catch verificar se o ponteiro não foi desalocado.

Desalocar o ponteiro caso ele ainda exista, e relançar a exceção para cima.

```
#include <iostream>

#include "Professor.hpp"

void funcao(){
   ufpr::Professor* prof{new ufpr::Professor{"Joao",
      11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  delete prof;
}

int main() {
   try{
       funcao();
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Smart Pointers

No C++17 temos 3 tipos de ponteiros inteligentes definidos no header `<memory>`.

```
unique_ptr
shared_ptr
weak_ptr
```

Os ponteiros inteligentes são **classes de template**, que encapsulam (são wrappers) ponteiros reais.

Os operadores *, -› são sobrecarregados, então podemos usar como ponteiros convencionais.

Cuidam da desalocação dos objetos automaGicamente.

unique_ptr

Um `unique_ptr` garante que existe somente um ponteiro apontando para a memória alocada.

Quando esse ponteiro perde seu escopo, `delete` é chamado automaticamente.

Exemplo

Quando `prof` sair de escopo (independentemente de como), o delete será chamado automaticamente.

`prof` ainda é um ponteiro internamente, e pode ser usado como tal sem problemas.

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

void funcao(){
   std::unique_ptr<ufpr::Professor> prof{
           new ufpr::Professor{"Joao", 11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";
}

int main() {
   try{
       funcao();
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

The C++ Programming Language

"... veja blocos `try-catch` explícitos em funções com suspeita; muitos deles poderiam ser substituídos por alguma variante da técnica RAII [provida pelos ponteiros inteligentes] ..."

en.cppreference.com/w/cpp/language/raii

learn.microsoft.com/pt-br/cpp/cpp/object-lifetime-and-resource-management-modern-cpp?view=msvc-170

"Um `unique_ptr` tem a propriedade interessante de não possuir overhead quando comparado a um ponteiro convencional..."

Bjarne Stroustrup, 2013.

Somente um ponteiro

A especificação garante que apenas um unique_ptr pode apontar para o objeto em determinado momento.

```
std::unique_ptr<ufpr::Professor> funcao(){
   std::unique_ptr<ufpr::Professor> prof{new ufpr::Professor{"Joao", 11111111111, 100, 40}};
   std::unique_ptr<ufpr::Professor> prof2{prof};//erro de compilação
   std::unique_ptr<ufpr::Professor> prof3;
   prof3 = prof;//erro de compilação

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  return prof;
}
```

Responda ...

O trecho a seguir:

a) Compila e funciona normalmente.
b) Não compila.
c) Compila e dá erro de execução.
d) Todas anteriores.
e) Abacate.

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

std::unique_ptr<ufpr::Professor> funcao(){
   std::unique_ptr<ufpr::Professor> prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  return prof;
}

int main() {
   try{
       std::cout << "Executando o try ... \n";
       std::unique_ptr<ufpr::Professor> profM{funcao()};
       std::cout << "Terminando o try ... \n";
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Responda ...

O trecho a seguir:

a) Compila e funciona normalmente.

O objeto é destruído apenas quando `profM` sai de escopo. É garantido que apenas um `unique_ptr` mantém o professor em determinado momento.

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

std::unique_ptr<ufpr::Professor> funcao(){
   std::unique_ptr<ufpr::Professor> prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  return prof;
}

int main() {
   try{
       std::cout << "Executando o try ... \n";
       std::unique_ptr<ufpr::Professor> profM{funcao()};
       std::cout << "Terminando o try ... \n";
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Responda ...

O compilador não aceita algo como

```
std::unique_ptr<ufpr::Professor> prof{new …};
std::unique_ptr<ufpr::Professor> prof3;
prof3 = prof;//erro de compilação
```

Mas aceita atribuir o `unique_ptr` retornado a um novo `unique_ptr`. Como? Quais mecanismos são usados internamente? Como você criaria a sua própria classe de `unique_ptr`?

```
#include <iostream>
#include <memory>

#include "Professor.hpp"

std::unique_ptr<ufpr::Professor> funcao(){
   std::unique_ptr<ufpr::Professor> prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  return prof;
}

int main() {
   try{
       std::cout << "Executando o try ... \n";
       std::unique_ptr<ufpr::Professor> profM{funcao()};
       std::cout << "Terminando o try ... \n";
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Responda ...

Construtores de cópia e operadores de atribuição são `deletados`.

O operador de `move assignment` é sobrecarregado.

O compilador verifica que o ponteiro inteligente retornado é um `rvalue`, que vai sumir da memória.

Usa o operador de `move assignment` sobrecarregado para transferir os dados de um ponteiro para outro, antes de remover da memória.

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

std::unique_ptr<ufpr::Professor> funcao(){
   std::unique_ptr<ufpr::Professor> prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  return prof;
}

int main() {
   try{
       std::cout << "Executando o try ... \n";
       std::unique_ptr<ufpr::Professor> profM{funcao()};
       std::cout << "Terminando o try ... \n";
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Boa prática ...

Prefira retornar um `unique_ptr` de uma função quando a memória é alocada dinamicamente, e precisa ser desalocada pelo cliente.

Evita que o cliente esqueça de realizar um `delete`.

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

std::unique_ptr<ufpr::Professor> funcao(){
   std::unique_ptr<ufpr::Professor> prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

  prof->setCpf(2222222222);
  std::cout << "CPF:" << prof->getCpf() << "\n";

  return prof;
}

int main() {
   try{
       std::cout << "Executando o try ... \n";
       std::unique_ptr<ufpr::Professor> profM{funcao()};
       std::cout << "Terminando o try ... \n";
   }catch(std::exception& e){
       std::cout << e.what() << '\n';
   }

  return 0;
}
```

Funções de unique_ptr

Existem diversas funções úteis para a classe unique_ptr.

Verifique em en.cppreference.com/w/cpp/memory/unique_ptr

Por exemplo, a função `get()` retorna o raw pointer (o ponteiro tradicional armazenado internamente no objeto).

```
void funcaoLegada(ufpr::Professor* prof){
  std::cout << "CPF:" << prof->getCpf() << "\n";
}

int main() {
   std::unique_ptr<ufpr::Professor> prof{new ufpr::Professor{"Joao", 11111111111, 100, 40}};
   funcaoLegada(prof.get());

  return 0;
}
```

shared_ptr

Um `shared_ptr` é um ponteiro inteligente que pode ser **compartilhado**.

Vários ponteiros `shared_ptr` podem apontar para um mesmo objeto.

Internamente, o `shared_ptr` mantém um contador, indicando quantos ponteiros apontam para o objeto atual.

Quando o contador chega em zero, o objeto é deletado automaticamente da memória.

shared_ptr

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

int main() {
   std::shared_ptr<ufpr::Professor> prof{
       new ufpr::Professor{"Joao", 11111111111, 100, 40}};

   {  // novo bloco
       std::shared_ptr<ufpr::Professor> prof2{prof};
       std::cout << prof2->getNome() << '\n';
       // objeto não é deletado aqui, ainda existe um ponteiro vivo apontando para prof
   }
   std::cout << "Fim!\n";
   return 0;
}
```

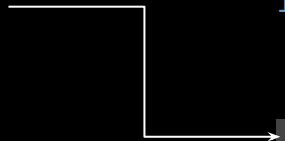
Dando um tiro no próprio pé

Onde está o erro? O que acontece?

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

int main() {
   ufpr::Professor* prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

   std::shared_ptr<ufpr::Professor> profSP{prof};
   std::shared_ptr<ufpr::Professor> profSP2{prof};


   std::cout << "Fim!\n";
   return 0;
}
```

Dando um tiro no próprio pé

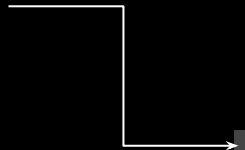
É criado um shared_ptr inicializado com o raw pointer alocado anteriormente.
Um contador é inicializado.
Até aqui tudo bem!



```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

int main() {
   ufpr::Professor* prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

   std::shared_ptr<ufpr::Professor> profSP{prof};
   std::shared_ptr<ufpr::Professor> profSP2{prof};


   std::cout << "Fim!\n";
   return 0;
}
```

Dando um tiro no próprio pé

Um segundo shared_pointer é inicializado com um raw pointer. O shared_pointer não sabe da existência do outro shared_pointer, logo, outro contador é inicializado. Temos agora dois contadores para um mesmo ponteiro!



```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

int main() {
   ufpr::Professor* prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

   std::shared_ptr<ufpr::Professor> profSP{prof};
   std::shared_ptr<ufpr::Professor> profSP2{prof};


   std::cout << "Fim!\n";
   return 0;
}
```

Dando um tiro no próprio pé

Agora você dará um passeio na montanha russa de **comportamentos indefinidos**!

Torça por um segfault assim que profSP2 sair de escopo.

```cpp
#include <iostream>
#include <memory>

#include "Professor.hpp"

int main() {
   ufpr::Professor* prof{
      new ufpr::Professor{"Joao", 11111111111, 100, 40}};

   std::shared_ptr<ufpr::Professor> profSP{prof};
   std::shared_ptr<ufpr::Professor> profSP2{prof};


   std::cout << "Fim!\n";
   return 0;
}
```

Overhead

Diferente de um `unique_ptr`, um `shared_ptr` possui overhead.

Quais são os overheads envolvidos?

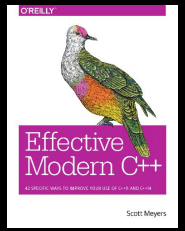
Overhead

Diferente de um `unique_ptr`, um `shared_ptr` possui overhead.

Alguns dos overheads:

- O contador de referências para o objeto.
- As atualizações do contador de referência (no construtor, no destrutor, no operador de atribuição, ...).
  A primeira vista, atualizar o contador é simples, mas não é...
  - O contador precisa ser atualizado de maneira atômica (caso múltiplas threads estejam tentando atualizar simultaneamente).
  - São necessárias estruturas extras para manter ponteiros especiais, como ponteiros para destrutores customizados.

Leia detalhes no capítulo 19 de Meyers, Meyers, S. (2014).



Mais problemas!!!

O que acontece de errado?

```
class Teste{
   void processar(){
       std::vector<std::shared_ptr<Teste>> jaProcessados;

       //...
       loop{
           if(jaProcessados nao contem atual){
               //processar
               if (atual == this)
                   jaProcessados.emplace_back(this);
           }

       }

   }
};
```

Mais problemas!!!

O this é o ponteiro da classe atual.

Construir um `shared_ptr` com `this` pode ser desastroso.

Quando a função acabar, o ponteiro vai tentar destruir o `this`, o que geralmente não faz sentido!

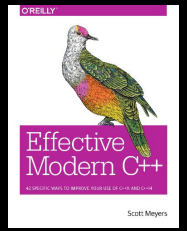
```
class Teste{
   void processar(){
       std::vector<std::shared_ptr<Teste>> jaProcessados;

       //...
       loop{
           if(jaProcessados nao contem atual){
               //processar
               if (atual == this)
                   jaProcessados.emplace_back(this);
           }

       }

   }
};
```

Mais problemas!!!

Quando esse tipo de construção for necessária, sua classe deverá herdar de `std::enable_shared_from_this<SUA_CLASSE>`, e usar a função `shared_from_this()` para criar uma referência segura ao this.

```
class Teste: : public std::enable_shared_from_this<Teste>{
   void processar(){
       std::vector<std::shared_ptr<Teste>> jaProcessados;

       //...
       loop{
           if(jaProcessados nao contem atual){
               //processar
               if (atual == this)
                   jaProcessados.emplace_back(shared_from_this());
           }

       }

   }
};
```

Leia detalhes no capítulo 19 de Meyers, Meyers, S. (2014).



Cuidado!

Cuidado com referências cíclicas. Nesse caso, os contadores nunca chegarão em zero!

Veja uma explicação interessante aqui: www.codementor.io/@sandesh87/smart-pointers-in-c-part-5-1jdn6o4bcb

Veremos como resolver através de `weak_ptr`.

Não use com arrays

Não é possível criar um shared_ptr para arrays crus!

Exemplo: `std::shared_ptr<T[]>` **é proibido!**

Se você precisar de um array, use alguma classe que armazena uma coleção de objetos.

std::array, std::vector, std::list, ...

Modificando o projeto

Vamos alterar o projeto para que os livros de uma ementa sejam representados por shared_ptrs.

Faça você mesmo

Verifique no projeto disponibilizado (“Programas antes da aula”) como `shared_ptrs` foram usados para representar os livros em uma ementa.

Agora os livros podem ser compartilhados entre ementas, sem criar complicações quanto a quem será o responsável por remover os livros da memória.

Criando uma função de fábrica

Todo livro tem um identificador único.

Vamos criar uma função de fábrica na classe Livro, que carrega o livro de um SGBD, ou arquivo, por exemplo.

Criando uma função de fábrica

Função estática é responsável por fabricar (carregar) um novo livro, baseado no ISBN.

Construtor é privado!

```
namespace ufpr{
class Livro{
   public:
       virtual ~Livro();

       static std::shared_ptr<const Livro>
            carregarLivro(const unsigned long ISBN);

       //…

   private:
       Livro(const unsigned long ISBN,
            const std::string& titulo, const short int ano);

       const unsigned long ISBN;//número de identificação único
       const std::string titulo;
       short int ano;
       std::string abstract;
       std::list<std::string> autores;
};
}
#endif
```

Criando uma função de fábrica

```cpp
#include "Livro.hpp"

#include <iostream>
#include <string>

namespace ufpr{

std::shared_ptr<const Livro> Livro::carregarLivro(const unsigned long ISBN){
   //abaixo a carga seria feita de um banco de dados.
   //vamos apenas simular colocando alguns dados no livro

   std::string nomeLivro{"Nome do Livro " +  std::to_string(ISBN)};

   return std::shared_ptr<const Livro>{new Livro{ISBN, nomeLivro, 1999}};
}
//…
}
```

Criando uma cache

Carregar um livro (ex.: de um banco de dados) custa tempo.

Se o livro já estiver carregado, podemos reutilizar.

Vamos criar uma cache para isso.

Cache

```
namespace ufpr{
class Livro{
   public:
       virtual ~Livro();

       static std::shared_ptr<const Livro> carregarLivro(const unsigned long ISBN);
      //…



   private:
       static std::unordered_map<unsigned long, std::shared_ptr<const Livro>> cache;

       Livro(const unsigned long ISBN, const std::string& titulo, const short int ano);

       const unsigned long ISBN;//número de identificação único
       const std::string titulo;
       short int ano;
       std::string abstract;
       std::list<std::string> autores;
};
}
```

Cache

```
namespace ufpr{

std::unordered_map<unsigned long, std::weak_ptr<const Livro>> Livro::cache;

std::shared_ptr<const Livro> Livro::carregarLivro(const unsigned long ISBN){
   //abaixo a carga seria feita de um banco de dados.
   //vamos apenas simular colocando alguns dados no livro

   std::unordered_map<unsigned long, std::shared_ptr<const Livro>>::const_iterator
     it{Livro::cache.find(ISBN)};

    if (it != Livro::cache.end()){
       std::cout << "Livro de ISBN " << ISBN << " na cache\n";
       return it->second;
    }

    std::cout << "Carregando livro de ISBN " << ISBN << '\n';
    std::string nomeLivro{"Nome do Livro " +  std::to_string(ISBN)};

    std::shared_ptr<const Livro> livro{new Livro{ISBN, nomeLivro, 1999}};
    Livro::cache[ISBN] = livro;

    return livro;
}

//...
}
```

Cache - Teste

```cpp
#include <iostream>
#include <memory>

#include "Livro.hpp"
#include "Ementa.hpp"

int main() {
   ufpr::Ementa* ementa1{new ufpr::Ementa{"Paradigmas de programação"}};
   ufpr::Ementa* ementa2{new ufpr::Ementa{"Orientação a Objetos"}};

   ementa1->addLivro(ufpr::Livro::carregarLivro(1234));
   ementa1->addLivro(ufpr::Livro::carregarLivro(5678));

   ementa2->addLivro(ufpr::Livro::carregarLivro(1234));


   delete ementa1;
   std::cout << "Ementa 1 deletada, deletando ementa 2\n";
   delete ementa2;

   return 0;
}
```

Cache - Teste

Qual o problema?

```cpp
#include <iostream>
#include <memory>

#include "Livro.hpp"
#include "Ementa.hpp"

int main() {
   ufpr::Ementa* ementa1{new ufpr::Ementa{"Paradigmas de programação"}};
   ufpr::Ementa* ementa2{new ufpr::Ementa{"Orientação a Objetos"}};

   ementa1->addLivro(ufpr::Livro::carregarLivro(1234));
   ementa1->addLivro(ufpr::Livro::carregarLivro(5678));

   ementa2->addLivro(ufpr::Livro::carregarLivro(1234));


   delete ementa1;
   std::cout << "Ementa 1 deletada, deletando ementa 2\n";
   delete ementa2;

   return 0;
}
```

Cache - Teste

Qual o problema?

Como a cache é uma lista estática, todo livro carregado vai permanecer na memória até o final do programa.

Mesmo que já não esteja sendo usado por alguma ementa.

```cpp
#include <iostream>
#include <memory>

#include "Livro.hpp"
#include "Ementa.hpp"

int main() {
    ufpr::Ementa* ementa1{new ufpr::Ementa{"Paradigmas de programação"}};
    ufpr::Ementa* ementa2{new ufpr::Ementa{"Orientação a Objetos"}};

    ementa1->addLivro(ufpr::Livro::carregarLivro(1234));
    ementa1->addLivro(ufpr::Livro::carregarLivro(5678));

    ementa2->addLivro(ufpr::Livro::carregarLivro(1234));


    delete ementa1;
    std::cout << "Ementa 1 deletada, deletando ementa 2\n";
    delete ementa2;

    return 0;
}
```

O que precisamos

Precisamos que o `unordered_map` armazene ponteiros que não impeçam o item de ser removido da memória.

Não incrementem o contador de referência.

Os ponteiros devem ser capazes de identificar se o objeto já foi eliminado da memória - um **"dangling pointer"**.

O que precisamos

Precisamos que o `unordered_map` armazene ponteiros que não impeçam o item de ser removido da memória.

Não incrementem o contador de referência.

Os ponteiros devem ser capazes de identificar se o objeto já foi eliminado da memória - um **"dangling pointer"**.

Um `weak_ptr` faz exatamente isso.

weak_ptr

Um weak_ptr não tem muita utilidade por si só.

Geralmente é criado a partir de um shared_ptr.

- Aponta para o mesmo objeto que o shared_ptr.
- Não altera o contador de referências.

weak_ptr

O `unordered_map` armazena agora `weak_ptrs`.

```
std::shared_ptr<const Livro> Livro::carregarLivro(const unsigned long ISBN){
   //abaixo a carga seria feita de um banco de dados.
   //vamos apenas simular colocando alguns dados no livro

   std::unordered_map<unsigned long, std::weak_ptr<const Livro>>::const_iterator
      it{Livro::cache.find(ISBN)};

    if (it != Livro::cache.end()){//um weak_ptr para o livro está na cache
       //resta verificar se o weak_ptr já foi deletado da memória
       std::shared_ptr<const Livro> sptrLivro{it->second.lock()};//null se dangling
       if(sptrLivro != nullptr){
           std::cout << "Livro de ISBN " << ISBN << " na cache\n";
           return sptrLivro;
       }else{
           std::cout << "Livro " << ISBN << " ja foi removido da cache\n";
       }
    }

    std::cout << "Carregando livro de ISBN " << ISBN << '\n';
    std::string nomeLivro{"Nome do Livro " +  std::to_string(ISBN)};

    std::shared_ptr<const Livro> livro{new Livro{ISBN, nomeLivro, 1999}};
    Livro::cache[ISBN] = livro;

    return livro;
}
```

weak_ptr

Procuramos pelo `weak_ptrs` na cache.

```
std::shared_ptr<const Livro> Livro::carregarLivro(const unsigned long ISBN){
   //abaixo a carga seria feita de um banco de dados.
   //vamos apenas simular colocando alguns dados no livro

   std::unordered_map<unsigned long, std::weak_ptr<const Livro>>::const_iterator
      it{Livro::cache.find(ISBN)};

   if (it != Livro::cache.end()){//um weak_ptr para o livro está na cache
      //resta verificar se o weak_ptr já foi deletado da memória
      std::shared_ptr<const Livro> sptrLivro{it->second.lock()};//null se dangling
      if(sptrLivro != nullptr){
          std::cout << "Livro de ISBN " << ISBN << " na cache\n";
          return sptrLivro;
      }else{
          std::cout << "Livro " << ISBN << " ja foi removido da cache\n";
      }
   }

   std::cout << "Carregando livro de ISBN " << ISBN << '\n';
   std::string nomeLivro{"Nome do Livro " +  std::to_string(ISBN)};

   std::shared_ptr<const Livro> livro{new Livro{ISBN, nomeLivro, 1999}};
   Livro::cache[ISBN] = livro;

   return livro;
}
```

weak_ptr

Se estava na cache, precisamos ainda verificar se ele aponta para algo que já foi deletado da memória (dangling).

A função `lock` retorna um `shared_ptr` para o item, ou `nullptr` se o item estiver deletado.

O `lock` cuida de condições de corrida automaticamente. Caso contrário, enquanto a função está retornando o ponteiro, outra thread poderia estar removendo o item da memória.

```
std::shared_ptr<const Livro> Livro::carregarLivro(const unsigned long ISBN){
   //abaixo a carga seria feita de um banco de dados.
   //vamos apenas simular colocando alguns dados no livro

   std::unordered_map<unsigned long, std::weak_ptr<const Livro>>::const_iterator
      it{Livro::cache.find(ISBN)};

    if (it != Livro::cache.end()){//um weak_ptr para o livro está na cache
       //resta verificar se o weak_ptr já foi deletado da memória
       std::shared_ptr<const Livro> sptrLivro{it->second.lock()};//null se dangling
       if(sptrLivro != nullptr){
           std::cout << "Livro de ISBN " << ISBN << " na cache\n";
           return sptrLivro;
       }else{
           std::cout << "Livro " << ISBN << " ja foi removido da cache\n";
       }
    }

    std::cout << "Carregando livro de ISBN " << ISBN << '\n';
    std::string nomeLivro{"Nome do Livro " +  std::to_string(ISBN)};

    std::shared_ptr<const Livro> livro{new Livro{ISBN, nomeLivro, 1999}};
    Livro::cache[ISBN] = livro;

    return livro;
}
```

Teste você mesmo

```cpp
#include <iostream>
#include <memory>

#include "Livro.hpp"
#include "Ementa.hpp"

int main() {
   ufpr::Ementa* ementa1{new ufpr::Ementa{"Paradigmas de programação"}};
   ufpr::Ementa* ementa2{new ufpr::Ementa{"Orientação a Objetos"}};
   ufpr::Ementa* ementa3{new ufpr::Ementa{"Prog. Em Java"}};

   ementa1->addLivro(ufpr::Livro::carregarLivro(1234));
   ementa1->addLivro(ufpr::Livro::carregarLivro(5678));

   ementa2->addLivro(ufpr::Livro::carregarLivro(1234));

   delete ementa1;
   std::cout << "Ementa 1 deletada, deletando ementa 2\n";
   delete ementa2;
   ementa3->addLivro(ufpr::Livro::carregarLivro(5678));
   delete ementa3;

   return 0;
}
```

Mais usos

`weak_ptr`s possuem diversos outros usos.

Podem, por exemplo, tratar de maneira mais simples algumas associações bidirecionais.

São usados para tratar referências cíclicas.

Podem ser usados para criar implementações elegantes do pattern observer.

No nosso caso da cache, usamos para implementar algo parecido com um Flyweight.

Mas possuem um custo.

Especialmente quando consideramos que o `lock()` precisa ser feito de maneira atômica.

Pesquise!

Dica

Em muitos casos é preferível usar as funções std::make_unique e std::make_shared ao invés de criar os pronteiros inteligentes diretamente.

Exemplo:

```
std::unique_ptr upw1(std::make_unique<Widget>());  // usando função make

std::unique_ptr<Widget> upw2(new Widget);  // sem função make
```

Pesquise os motivos.

Smart Pointers versus Garbage Collector

Não confunda os mecanismos do C++, como o shared_ptr, com coletores de lixo implementados em linguagens como Java e C#.

Cada método tem suas vantagens e desvantagens.

Algumas vantagens do smart pointer:

- Overhead (muito) menor.
- A memória é liberada assim que o contador chega em zero.

Algumas vantagens dos coletores de lixo (tradicionais):

- Mais fácil para o programador.
- Tratam automaticamente de problemas complicados, como referências cíclicas.

Pesquise!

Para mais vantagens e desvantagens, **pesquise você mesmo.**

Muito cuidado! A proporção de asneiras por m^2 sobre o assunto de overheads e comparação de smart pointers com garbage collectors na internet (especialmente no Stack Overflow) é impressionante!

Artigo interessante sobre overheads de um garbage collector:

Hertz, Matthew and Berger, Emery D. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. SIGPLAN Not. 2005.

https://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf

Curiosidades

A linguagem de programação Rust utiliza contadores de referências (da mesma forma que smart pointers) para gerenciar a memória em alguns casos.

Originalmente o Python (até a versão 2.0) utilizava apenas contadores de referências para gerenciar a memória.

> Como não contava com outros mecanismos, e não deixava o programador gerenciar memória, causava *memory leaks* quando, por exemplo, eram geradas referências cíclicas em objetos.

Boost C++

A Boost C++ oferece uma enorme quantidade de bibliotecas extras para o C++.

Extensão padrão e oficial do C++.

Itens que vão para o C++ ISO são primeiramente implementados na Boost.

Smart Pointers, loops foreach, classes de data/hora, ...

www.boost.org

```
boost::gregorian::date currentDay;
std::map<boost::posix_time::ptime,mtopk::ImageData*>::const_iterator it{imgData->begin()};
```

Mais

Esse foi o básico de orientação a objetos em C++.

Existem vários outros conceitos interessantes, fica para uma próxima:

- Concepts
- Expressões Lambda
- Concorrência
- ...

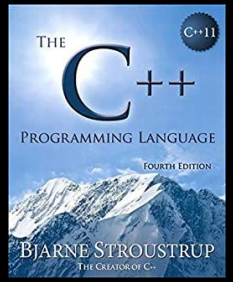

The problem of being faster than light is that you can only live in darkness

Exercícios

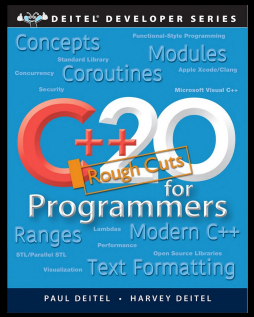
1. Implemente o mecanismo de cache discutido na aula.
2. Em `Ementa`, troque `std::list<std::shared_ptr<const Livro>>* livros` por um `unique_ptr`. Você vai precisar usar algumas funções especiais de `unique_ptr` para que tudo funcione (por exemplo, no `move constructor`).
3. Verifique no projeto outros locais onde ponteiros inteligentes podem ser utilizados, e faça a substituição.

Referências

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

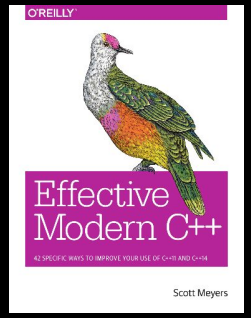


Deitel, P., Deitel, H. C++20 for Programmers: An Objects-Natural Approach. 2022.

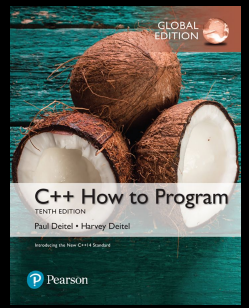


www.boost.org

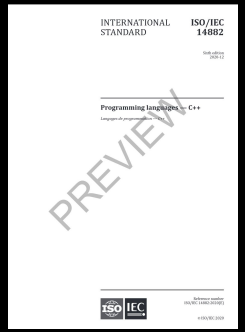
Meyers, S. D., Meyers, S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. 2014.



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