Foo Bar é possivelmente uma alusão jocosa à gíria da segunda guerra mundial FUBAR (Fucked Up Beyond All Recognition).
https://en.wikipedia.org/wiki/Foobar

Sobrecargas Avançadas

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Entenda

Faça o download da versão corrente do projeto.

Verifique **atentamente** o que foi feito e como as coisas funcionam.

- Foi criada uma classe Livro;
- Foi criada uma classe Ementa;
- A classe disciplina pode agora ter uma Ementa;

O que está acontecendo?

O que está acontecendo aqui?

```
int main(){
   ufpr::Livro l1{"C++ How To Program", 2017};
   l1.addAutor("Paul Deitel");
   l1.addAutor("Harvey Deitel");

   ufpr::Livro l2{"The C++ Programming Language", 2013};
   l1.addAutor("Bjarne Stroustrup ");

   ufpr::Ementa* ementa1{new ufpr::Ementa{
            "Curso de C++. Aulas de Orientação a Objetos."}};
   ementa1->addLivro(l1);
   ementa1->addLivro(l2);

   ufpr::Ementa ementaCopia{*ementa1};

   std::cout << "Dados da Ementa Copia "
            << ementaCopia.getDescricao() << '\n';
   std::list<ufpr::Livro>::const_iterator
            it{ementaCopia.getLivros()->begin()};
   for( ; it != ementaCopia.getLivros()->end(); ++it){
       std::cout << it->getTitulo() << '\n';
   }

  return 0;
}
```

O que está acontecendo?

O **construtor de cópia** está sendo invocado.

Como não definimos esse construtor, o compilador gerou automaticamente.

É criado um novo objeto Ementa na memória, e cada um dos membros de ementa1 é copiado para os membros de ementaCopia automaGicamente.

```
int main(){
   ufpr::Livro l1{"C++ How To Program", 2017};
   l1.addAutor("Paul Deitel");
   l1.addAutor("Harvey Deitel");

   ufpr::Livro l2{"The C++ Programming Language", 2013};
   l1.addAutor("Bjarne Stroustrup ");

   ufpr::Ementa* ementa1{new ufpr::Ementa{
            "Curso de C++. Aulas de Orientação a Objetos."}};
   ementa1->addLivro(l1);
   ementa1->addLivro(l2);

   ufpr::Ementa ementaCopia{*ementa1};

   std::cout << "Dados da Ementa Copia "
            << ementaCopia.getDescricao() << '\n';
   std::list<ufpr::Livro>::const_iterator
            it{ementaCopia.getLivros()->begin()};
   for( ; it != ementaCopia.getLivros()->end(); ++it){
       std::cout << it->getTitulo() << '\n';
   }

  return 0;
}
```

O que está acontecendo?

Parece tudo correto, mas ...

Existe um **erro grave** na forma com que as classes foram projetadas.

Você consegue identificar?

```
int main(){
   ufpr::Livro l1{"C++ How To Program", 2017};
   l1.addAutor("Paul Deitel");
   l1.addAutor("Harvey Deitel");

   ufpr::Livro l2{"The C++ Programming Language", 2013};
   l1.addAutor("Bjarne Stroustrup ");

   ufpr::Ementa* ementa1{new ufpr::Ementa{
            "Curso de C++. Aulas de Orientação a Objetos."}};
   ementa1->addLivro(l1);
   ementa1->addLivro(l2);

   ufpr::Ementa ementaCopia{*ementa1};

   std::cout << "Dados da Ementa Copia "
            << ementaCopia.getDescricao() << '\n';
   std::list<ufpr::Livro>::const_iterator
            it{ementaCopia.getLivros()->begin()};
   for( ; it != ementaCopia.getLivros()->end(); ++it){
       std::cout << it->getTitulo() << '\n';
   }

  return 0;
}
```

O que está acontecendo?

Deveria ser possível remover ementa1 da memória sem afetar ementaCopia, mas não é isso que acontece!

```
int main(){
    ufpr::Livro l1{"C++ How To Program", 2017};
    l1.addAutor("Paul Deitel");
    l1.addAutor("Harvey Deitel");

    ufpr::Livro l2{"The C++ Programming Language", 2013};
    l1.addAutor("Bjarne Stroustrup ");

    ufpr::Ementa* ementa1{new ufpr::Ementa{
            "Curso de C++. Aulas de Orientação a Objetos."}};
    ementa1->addLivro(l1);
    ementa1->addLivro(l2);

    ufpr::Ementa ementaCopia{*ementa1};

    delete ementa1;

    std::cout << "Dados da Ementa Copia "
            << ementaCopia.getDescricao() << '\n';
    std::list<ufpr::Livro>::const_iterator
            it{ementaCopia.getLivros()->begin()};
    for( ; it != ementaCopia.getLivros()->end(); ++it){
        std::cout << it->getTitulo() << '\n';
    }

  return 0;
}
```

```
Dados da Ementa Copia Curso de C++. Aulas de Orientação a Objetos.
Falha de segmentação (imagem do núcleo gravada)
```

O que está acontecendo

O construtor de cópia padrão copia os itens dos dados membro para o novo objeto.

Copia o ponteiro da lista de livros para o novo objeto.

Agora os dois objetos apontam para a **mesma lista.**

```
class Ementa{
    public:
        Ementa();
        Ementa(const std::string descricao);
        virtual ~Ementa();

        void setDescricao(const std::string& descricao);
        const std::string& getDescricao() const;

        void addLivro(const Livro& livro);
        const std::list<Livro>* getLivros() const;

    private:
        std::string descricao;
        std::list<Livro>* livros;
};
}
```

Shallow Copy

O construtor de cópia padrão faz uma **cópia rasa.**

Shallow Copy.

Copia os dados membro de um objeto para o outro, sem entrar nos objetos internos recursivamente.

Prós e contras?

Shallow Copy

Prós:

+ A cópia é simples;
+ Baixo overhead;
+ As vezes é isso o que desejamos

Estude o Padrão de Projeto Flyweight.

Shallow Copy

Prós:

+ A cópia é simples;
+ Baixo overhead;
+ As vezes é isso o que desejamos

Estude o Padrão de Projeto Flyweight.

Contras:

- Os objetos ficam atrelados na memória.

Alterar um objeto pode resultar em modificações em outros objetos.

- Pode levar a erros difíceis de encontrar.

Sobrecarregando o construtor de cópia

Podemos sobrescrever o construtor de cópia.

Fazer um construtor de cópia que cria objetos consistentes na memória.

Vamos criar um construtor que faz uma **cópia profunda.**

Deep Copy.

Criar cópias recursivamente dos dados internos para que nada seja compartilhado entre as cópias.

Deep copy

O construtor de cópia deve receber uma **referência const** para um objeto da mesma classe.

```
#ifndef EMENTA_HPP
#define EMENTA_HPP

#include <string>
#include <list>

#include "Livro.hpp"

namespace ufpr{
class Ementa{
   public:
       Ementa();
       Ementa(const std::string descricao);
       Ementa(const Ementa& ementa);//construtor de cópia
       virtual ~Ementa();

       void setDescricao(const std::string& descricao);
       const std::string& getDescricao() const;

       void addLivro(const Livro& livro);
       const std::list<Livro>* getLivros() const;

   private:
       std::string descricao;
       std::list<Livro>* livros;
};
}
#endif
```

Deep copy

Criar um nova lista de livros.

Invocando o construtor de cópia da classe list, para copiar os elementos da lista original.

```
#include "Ementa.hpp"


Ementa::Ementa(const Ementa& ementa)
        :descricao{ementa.descricao}, livros{new std::list<Livro>{*ementa.livros}}{
}

//...
```

Em outras linguagens

Linguagens como Java e C# utilizam outras estratégias para realizar as cópias.

Em Java, existe um método (função membro) `clone()` dentro da classe `Object`. O programador precisa sobrescrever a função.

- Precisa chamar a função explicitamente quando deseja criar uma cópia.
- https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Em .Net existe uma classe que define a função `clone()`.

- Precisa herdar dessa classe, e chamar a função explicitamente.
  - E na falta de herança múltipla, o problema é resolvido via um interface.
- https://learn.microsoft.com/pt-br/dotnet/api/system.icloneable.clone?view=net-6.0

O que acontece?

Quantos objetos do tipo `Ementa` são construídos na memória?

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

#include "Ementa.hpp"

int main(){
   std::list<ufpr::Ementa> ementas;
   ementas.push_back(ufpr::Ementa{"Ementa 1"});
   ementas.push_back(ufpr::Ementa{"Ementa 2"});

   std::list<ufpr::Ementa>::const_iterator
            it{ementas.begin()};
   for( ; it != ementas.end(); ++it)
       std::cout << it->getDescricao() << '\n';

  return 0;
}
```

O que acontece?

2 objetos?


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

#include "Ementa.hpp"

int main(){
    std::list<ufpr::Ementa> ementas;
    ementas.push_back(ufpr::Ementa{"Ementa 1"});
    ementas.push_back(ufpr::Ementa{"Ementa 2"});

    std::list<ufpr::Ementa>::const_iterator
            it{ementas.begin()};
    for( ; it != ementas.end(); ++it)
        std::cout << it->getDescricao() << '\n';

  return 0;
}
```

O que acontece?

São criados **4 objetos.**

Coloque `couts` em cada um dos construtores para comprovar.

Não esqueça do construtor de cópia.

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

#include "Ementa.hpp"

int main(){
   std::list<ufpr::Ementa> ementas;
   ementas.push_back(ufpr::Ementa{"Ementa 1"});
   ementas.push_back(ufpr::Ementa{"Ementa 2"});

   std::list<ufpr::Ementa>::const_iterator
            it{ementas.begin()};
   for( ; it != ementas.end(); ++it)
       std::cout << it->getDescricao() << '\n';

  return 0;
}
```

Uma questão de desempenho

As coisas parecem um tanto complicadas (e realmente são).

A vasta maioria das linguagens O.O. não suportam os mecanismos discutidos.

- As coisas são mais simples, mas cópias completas ou braçais são comumente mandatórias.
- O overhead é muitas vezes gigantesco.
  - O programador só não está ciente disso (a ignorância é a mãe da felicidade).

Indo mais rápido

A partir do C++11, podemos criar **Move Constructors** e operadores de **Move Assignment**.

Criar novos objetos a partir de objetos temporários.

Construção mais leve.

Move constructor

O && indica uma **referência RValue.**

Uma referência RValue refere a um objeto temporário. O compilador garante que o objeto nunca mais será usado.

Veja detalhes na Seção 7.7.2 de Stroustrup (2013).

```
class Ementa{
   public:
       Ementa();
       Ementa(const std::string descricao);
       Ementa(const Ementa& ementa);//construtor de cópia
       Ementa(Ementa&& ementa);//move constructor
       virtual ~Ementa();

       void setDescricao(const std::string& descricao);
       const std::string& getDescricao() const;

       void addLivro(const Livro& livro);
       const std::list<Livro>* getLivros() const;

   private:
       std::string descricao;
       std::list<Livro>* livros;
};
```

Move constructor

No move constructor podemos “roubar” os recursos alocados na ementa RValue.

- Economizar tempo com as cópias.
- O RValue vai ser removido da memória depois da chamada do construtor.

Não podemos esquecer de realizar ajustes no destrutor.

```
Ementa::Ementa(Ementa&& ementa)
        :descricao{ementa.descricao}, livros{ementa.livros}{
    ementa.livros = nullptr;
}

Ementa::~Ementa(){
    if(livros!=nullptr)
        delete livros;
}
```

Move Assignment

O operador de move assignment é similar ao move constructor.

A diferença é que ele serve para operações de atribuição.

Atenção: ao declarar um move constructor, ou move assignment, você automaticamente precisará declarar:

move constructor;

move assignment;

operador de atribuição.

Esses operadores deixam de ser declarados implicitamente.

Move Assignment

```
namespace ufpr{
class Ementa{
   public:
       Ementa();
       Ementa(const std::string descricao);
       Ementa(const Ementa& ementa);//construtor de
cópia
       Ementa(Ementa&& ementa);//move constructor
       virtual ~Ementa();

       void setDescricao(const std::string& descricao);
       const std::string& getDescricao() const;

       void addLivro(const Livro& livro);
       const std::list<Livro>* getLivros() const;

       Ementa& operator=(Ementa&& ementa);
       const Ementa& operator=(const Ementa& ementa);

   private:
       std::string descricao;
       std::list<Livro>* livros;
};
}
```

Move Assignment

```
Ementa& Ementa::operator=(Ementa&& ementa){
    if(this == &ementa)
        return *this;
    this->descricao = ementa.descricao;
    this->livros = ementa.livros;
    ementa.livros = nullptr;

    return *this;
}

const Ementa& Ementa::operator=(const Ementa& ementa){
    if(this == &ementa)
        return *this;
    this->descricao = ementa.descricao;
    delete this->livros;
    this->livros = new std::list{*ementa.livros};

    return *this;
}
```

Faça você mesmo

Sem executar o programa, identifique no main a seguir qual construtor/atribuição será invocado (construtor de cópia, move constructor, move assigment, ...).

Depois, coloque `couts` em cada componente criado na aula (construtor de cópia, move constructor, move assigment, ...) e execute o programa. Você estava certo?

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

#include "Ementa.hpp"
#include "Disciplina.hpp"

int main(){
   std::list<ufpr::Ementa> ementas;
   ementas.push_back(ufpr::Ementa{"Ementa 1"});
   ementas.push_back(ufpr::Ementa{"Ementa 2"});

   ufpr::Ementa ementaDis{"Ementa Dis"};
   ufpr::Disciplina disciplina{"C++"};
   disciplina.setEmenta(ementaDis);

   ementaDis = ufpr::Ementa{"Outra Ementa"};

   std::list<ufpr::Ementa>::const_iterator
          it{ementas.begin()};
   for( ; it != ementas.end(); ++it)
       std::cout << it->getDescricao() << '\n';

  return 0;
}
```

Move Assignment

Os move assignment são especialmente eficientes em cenários do tipo var = valorRetornadoFuncao() ...

Exemplo:

```
class Foo{
    public:
        …
        static Bar criarBar();
}

main{
    Bar nova{...};
    nova = Foo::criarBar();
    //…
    return 0;
}
```

É retornado um item por cópia. O item será descartado da memória após a cópia. Então um move assignment será chamado (se você o criar), o que será mais eficiente.

Deletando

Antes do C++ 11, para se remover uma função implicitamente declarada, as funções eram declaradas como privadas.

Por exemplo: remover o construtor de cópia padrão, atribuição padrão, ... para evitar a cópia de objetos.

A partir do C++11, é possível explicitamente deletar funções.

Veja detalhes em no capítulo 10 de Deitel e Deitel (2017).

Exemplo

A classe console possui apenas funções estáticas.

Não faz sentido instanciar objetos do tipo Console.

O construtor padrão foi deletado para evitar instanciação.

O mesmo efeito se daria ao declarar o construtor como privado.

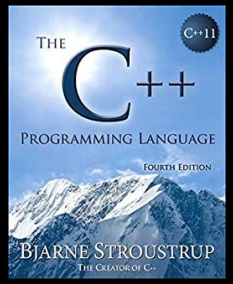
```
#ifndef CONSOLE_HPP
#define CONSOLE_HPP

#include "Disciplina.hpp"

namespace ufpr{
class Console{
   public:
       Console() = delete;
       virtual ~Console() = default;

       static void imprimirDadosDisciplina(
          const Disciplina& disciplina);
};
}
#endif
```

Exercícios

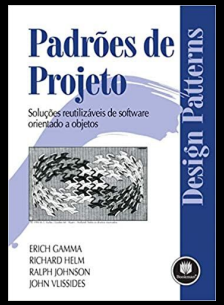
1. Leia sobre como os operadores de incremento e decremento prefixos e pós-fixos devem ser sobrecarregados, e como isso afeta, por exemplo, a classe iterator. Por que é mais eficiente fazer `++it` do que `it++`?
2. Sobrecarregue **todos** os operadores, construtores e mecanismos de atribuição que fizerem sentido para a classe Disciplina.

Referências

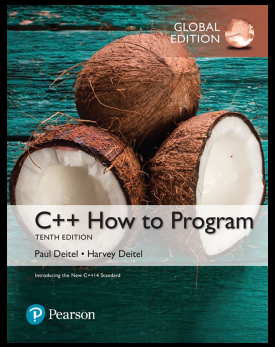
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



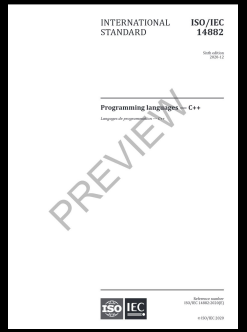
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



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