CC BY

"A única constante na vida é a mudança" (Heráclito).

Constantes

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

const

Ao declarar um objeto ou variável como `const`, estamos indicando que o objeto (ou variável) não pode ser alterado depois de inicializado.

Exemplo:

```
int main(){
   const Pessoa p{"Joao", 11111111111, 20};
   const int valor{1};


   //...
   return 0;
}
```

const

Ao declarar um objeto ou variável como `const`, estamos indicando que o objeto (ou variável) não pode ser alterado depois de inicializado.

> **Tentar alterar um objeto const resulta em um erro de compilação.**

Exemplo:

```
int main(){
   const Pessoa p{"Joao", 11111111111, 20};
   const int valor{1};

   valor++;

   //...
   return 0;
}
```

const

Você não pode acessar membros não const de um objeto const.

Somente membros const podem ser acessados.

Caso contrário, um set por exemplo poderia alterar o objeto (que deveria ser constante).

```
#include "Pessoa.hpp"

#include<iostream>

int main(){
   const Pessoa p{"Joao", 11111111111, 20};
   const int valor{1};

   std::cout << p.getNome() << "\n";//ERRO
   std::cout << valor << "\n";

   return 0;
}
```

Funções const

Uma **função membro `const`** garante ao compilador de que a função não modifica o estado objeto.

Não modifica os dados membro do objeto.

Adicione no final da Função.

```
retorno nome(parametros) const;
```

Que tipos de função tipicamente devem ser `const`?

Funções const

Uma **função membro `const`** garante ao compilador de que a função não modifica o estado objeto.

Não modifica os dados membro do objeto.

Adicione no final da Função.

```
retorno nome(parametros) const;
```

Que tipos de função tipicamente devem ser `const`?

Os gets geralmente são const, já que não alteram o objeto.

Comumente retornam uma cópia de algum dado membro.

Exemplo

```hpp
#ifndef PESSOA_H
#define PESSOA_H

#include<string>

class Pessoa{
   public:

       //...

       std::string getNome() const;

   private:
       bool validarCPF(unsigned long cpfTeste);

       std::string nome;
       unsigned long cpf;
       unsigned char idade;
};
#endif
```

```cpp
#include "Pessoa.hpp"

std::string Pessoa::getNome() const{
    return this->nome;
}

//...
```

const

Agora isso é válido!

```cpp
#include "Pessoa.hpp"

#include<iostream>

int main(){
   const Pessoa p{"Joao", 11111111111, 20};
   const int valor{1};

   std::cout << p.getNome() << "\n";//Agora é válido
   std::cout << valor << "\n";

   return 0;
}
```

Faça você mesmo

Faça com que a função setNome seja const.

Modifique no .hpp e no .cpp.

```
make clean
```

```
make
```

O que acontece? Por que?

Faça você mesmo

Faça com que a função setNome seja const.

Modifique no .hpp e no .cpp.

```
make clean
```

```
make
```

O que acontece? Por que?

Erro de compilação.

O compilador detectou que você fez uma promessa e não cumpriu.

A função setNome modifica o estado do objeto, logo não pode ser const.

Construtores e Destrutores

Construtores e destrutores não podem ser const.

Construtores e destrutores **podem modificar os itens const da classe.**

Os itens const da classe são constantes apenas:

- Depois do construtor terminar de construir o objeto;
- E até a chamada do destrutor.

Parâmetros Const

Podemos indicar que os **parâmetros de funções são const.**

Indica que a função vai **apenas ler** os dados desse parâmetro, e não vai alterá-lo de forma alguma.

Parâmetros Const

Podemos indicar que os **parâmetros de funções são const.**

Indica que a função vai **apenas ler** os dados desse parâmetro, e não vai alterá-lo de forma alguma.

Especialmente útil quando passamos parâmetros via referência.

Por quê?

Parâmetros Const

Podemos indicar que os **parâmetros de funções são const.**

Indica que a função vai **apenas ler** os dados desse parâmetro, e não vai alterá-lo de forma alguma.

Especialmente útil quando passamos parâmetros via referência.

- Uma passagem por referência é mais leve para objetos complexos.
- Mas como garantir que a função chamada não vai alterar o nosso dado original?
  - Se o parâmetro é const, essa garantia é dada.

Exemplo

Modifique a função `setNome` de Pessoa para receber uma referência para a string do nome.

Evitamos que a string toda seja copiada ao ser passada para a função.

- rápido.

Garanta que a string original não é alterada, anotando-a como `const`.

```
class Pessoa{
   public:
       //...
       void setNome(const std::string& nome);
   private:
       bool validarCPF(unsigned long cpfTeste);

       std::string nome;
       unsigned long cpf;
       unsigned char idade;
};
```

```
void Pessoa::setNome(const std::string& nome){
   this->nome = nome;
}
```

Indo mais rápido

Variáveis e parâmetros `const` abrem espaço para muitas **otimizações.**

O compilador pode realizar otimizações que não são possíveis em variáveis não `const`.

O compilador pode, por exemplo, salvar os dados no segmento de dados ou segmento de texto do programa.

Cálculos de endereço são mais simples nesses segmentos.

Exemplo

```cpp
#include<iostream>

int main(){
   int valor{1330};

   std::cout << valor << std::endl;

   return 0;
}
```

Assembly x86 gerado no g++ 9.3.0

g++ -S teste.cpp

```
 ...
 main:
 .LFB1522:
     .cfi_startproc
     endbr64
     pushq    %rbp
     .cfi_def_cfa_offset 16
     .cfi_offset 6, -16
     movq     %rsp, %rbp
     .cfi_def_cfa_register 6
     subq     $16, %rsp
     movl     $1330, -4(%rbp)
     movl     -4(%rbp), %eax
     movl     %eax, %esi
     leaq     _ZSt4cout(%rip), %rdi
     call     _ZNSolsEi@PLT
     movq     %rax, %rdx
     movq
     ...
```

Exemplo

```cpp
#include<iostream>

int main(){
   const int valor{1330};

   std::cout << valor << std::endl;

   return 0;
}
```

Assembly x86 gerado no g++ 9.3.0

g++ -S teste.cpp

```
...
main:
.LFB1522:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $1330, -4(%rbp)
    movl    $1330, %esi
    leaq    _ZSt4cout(%rip), %rdi
    call    _ZNSolsEi@PLT
    movq    %rax, %rdx
    movq
    ...
```

Indo mais rápido

Com const.

```
...
main:
.LFB1522:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $1330, -4(%rbp)
    movl    $1330, %esi
    leaq    _ZSt4cout(%rip), %rdi
    call    _ZNSolsEi@PLT
    movq    %rax, %rdx
    movq
    ...
```

O compilador não precisou carregar o valor da memória, pois assumiu que sempre vai ser 1330, e carregou como um imediato para o registrador esi.

Sem const.

```
...
main:
.LFB1522:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $1330, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    leaq    _ZSt4cout(%rip), %rdi
    call    _ZNSolsEi@PLT
    movq    %rax, %rdx
    movq
    ...
```

Const ou não const, eis a questão

O parâmetro idade deveria ser const? Há algum ganho nisso?

```
void Pessoa::setIdade(unsigned short int idade){
      this->idade = (unsigned char)idade;
}
```

Const ou não const, eis a questão

- A passagem é feita por cópia, e não ponteiro ou referência.
  - O valor original da variável passada como parâmetro não é alterado.
    - Como const garante que o valor original não vai ser alterado, isso se torna redundante.
- **Mas ainda podemos ter ganhos.**
  - Do ponto de vista da engenharia de software.
    - Deixamos claro que esse parâmetro é algo que vai ser usado somente para leitura.
  - Do ponto de vista da performance.
    - O compilador pode otimizar as coisas.

```
void Pessoa::setIdade(unsigned short int idade){
      this->idade = (unsigned char)idade;
}
```

Principle of Least Privilege

Princípio do menor privilégio.

Suas funções devem ter apenas acesso o suficiente aos dados para poder cumprir sua tarefa, e não mais do que isso (DEITEL; DEITEL, 2017).

Continuando ...

Você consegue identificar o problema criado com as funções a seguir na classe Disciplina?

```
class Disciplina{
   public:
       //...

       void adicionarAluno(Pessoa* aluno);
       void removerAluno(Pessoa* aluno);
       void removerAluno(unsigned long cpf);
       std::list<Pessoa*>& getAlunos();
   private:
       // ...
       std::list<Pessoa*> alunos;
       std::list<ConteudoMinistrado*> conteudos;
};
```

Problema

Quebra de encapsulamento.

Para economizar memória e processamento, é retornada uma referência para um dado membro.

Através dessa referência, a classe cliente pode fazer o que quiser com esse dado membro que é **privado!**

```
class Disciplina{
   public:
       //...

       void adicionarAluno(Pessoa* aluno);
       void removerAluno(Pessoa* aluno);
       void removerAluno(unsigned long cpf);
       std::list<Pessoa*>& getAlunos();
   private:
       // ...
       std::list<Pessoa*> alunos;
       std::list<ConteudoMinistrado*> conteudos;
};
```

Faça você mesmo

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

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

int main(){
   Disciplina d{"C++", nullptr};
   Pessoa p1{"Joao"};

   d.adicionarAluno(&p1);
   std::list<Pessoa*>& alunos{d.getAlunos()};//acessamos a lista original
   Pessoa p2{"Maria"};
   alunos.push_back(&p2);//modificamos o objeto interno de Disciplina

   std::list<Pessoa*>::iterator it = d.getAlunos().begin();
   for( ;it != d.getAlunos().end(); it++){
       std::cout << (*it)->getNome() << std::endl;
   }

   return 0;
}
```

Retornos const

Ao inserir const **na frente da declaração da função**, indicamos que o retorno da função é `const`.

Somente leitura.

Caso o retorno seja um objeto, **funções membro não `const` do objeto retornado não podem ser invocados.**

Exemplo

Disciplina.cpp

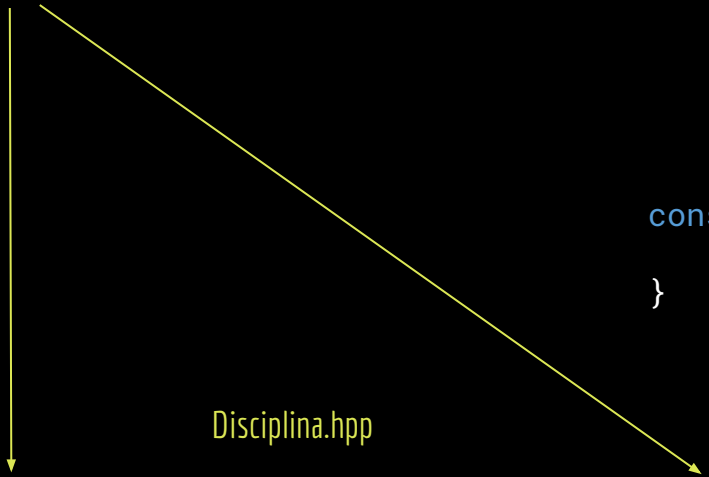
```
const std::list<Pessoa*>& Disciplina::getAlunos() const{
   return this->alunos;
}
```

Disciplina.hpp

```
const std::list<Pessoa*>& getAlunos() const;
```

Exemplo

Dois consts?



Disciplina.hpp

```
const std::list<Pessoa*>& getAlunos() const;
```

Disciplina.cpp

```
const std::list<Pessoa*>& Disciplina::getAlunos() const{
    return this->alunos;
}
```

Exemplo

Primeiro `const`: O retorno vai ser `const` (somente leitura).

Segundo `const`: A função é `const` (não altera o estado do objeto).

Disciplina.cpp

```cpp
const std::list<Pessoa*>& Disciplina::getAlunos() const{
   return this->alunos;
}
```

Disciplina.hpp

```cpp
const std::list<Pessoa*>& getAlunos() const;
```

Faça você mesmo

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

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

int main(){
   Disciplina d{"C++", nullptr};
   Pessoa p1{"Joao"};

   d.adicionarAluno(&p1);
   const std::list<Pessoa*>& alunos{d.getAlunos()};//acessamos a lista original
   //Pessoa p2{"Maria"};
   //alunos.push_back(&p2);//não podemos fazer isso em uma referência const

   std::list<Pessoa*>::const_iterator it = alunos.begin();
   for( ;it != d.getAlunos().end(); it++){
       std::cout << (*it)->getNome() << std::endl;
   }

   return 0;
}
```

Valores retornados por cópia vs. referência

Geralmente valores retornados por cópia não são const.

Mas é comum termos retornos de ponteiro ou referência const.

Ponteiros const

Temos **quatro cenários** possíveis para ponteiros const.

Ponteiro não const para dados não const

Um **ponteiro não const para dados não const** é o que estamos fazendo até o momento.

O ponteiro pode apontar para qualquer objeto.

Podemos alterar os dados do objeto apontado.

```
int main(){
    Pessoa p1{"João"};
    Pessoa p2{"Maria"};

    Pessoa* ptr1{&p1};
    ptr1->setNome("Pedro");

    std::cout << ptr1->getNome() << std::endl;
    ptr1 = &p2;
    std::cout << ptr1->getNome() << std::endl;

    return 0;
}
```

Ponteiro não const para dados const

Ponteiro **não const para dados const.**

O ponteiro pode apontar para qualquer objeto.

Não podemos alterar os dados do objeto apontado.

Ex.: Não podemos chamar funções membro não const.

Declarado **adicionando const na frente** da declaração do ponteiro.

Não é possível chamar funções que alteram o estado do objeto (funções não const).

Pode trocar de objeto apontado.

```
int main(){
    Pessoa p1{"João"};
    Pessoa p2{"Maria"};

    const Pessoa* ptr1{&p1};
    //ptr1->setNome("Pedro");

    std::cout << ptr1->getNome() << std::endl;
    ptr1 = &p2;
    std::cout << ptr1->getNome() << std::endl;

    return 0;
}
```

Ponteiro const para dados não const

Ponteiro **const para dados não const.**

O ponteiro não pode trocar de objeto apontado.

Depois de inicializado, sempre aponta para a mesma região da memória.

Deve ser inicializado assim que criado.

Pode alterar os dados do objeto apontado.

Declarado adicionando **const após o tipo da variável.**

Pode alterar o estado do objeto apontado →

Não pode trocar de objeto apontado →

```
int main(){
    Pessoa p1{"João"};
    Pessoa p2{"Maria"};

    Pessoa* const ptr1{&p1};
    ptr1->setNome("Pedro");

    std::cout << ptr1->getNome() << std::endl;
    //ptr1 = &p2;

    return 0;
}
```

Ponteiro const para dados não const

Ponteiro **const para dados não const.**

O ponteiro não pode trocar de objeto apontado.

Depois de inicializado, sempre aponta para a mesma região da memória.

Deve ser inicializado assim que criado.

Pode alterar os dados do objeto apontado.

Declarado adicionando **const após o tipo da variável.**

Qual a diferença entre um *ponteiro const para dados não const* e uma *referência?*

```
int main(){
    Pessoa p1{"João"};
    Pessoa p2{"Maria"};

    Pessoa* const ptr1{&p1};
    ptr1->setNome("Pedro");

    std::cout << ptr1->getNome() << std::endl;
    //ptr1 = &p2;

    return 0;
}
```

Ponteiro const para dados não const

Qual a diferença entre um ponteiro const para dados não const e uma referência?

- Um ponteiro pode ser nulo.
  - Uma referência pode não existir na memória.
    - O compilador pode entender que uma referência só é um alias para o objeto original.
  - Um ponteiro sempre existe na memória.
    - Ocupa espaço e tem seu próprio endereço na memória.
    - Internamente armazena o endereço do objeto apontado.
  - A sintaxe muda.

```
int main(){
    Pessoa p1{"João"};
    Pessoa p2{"Maria"};

    Pessoa* const ptr1{&p1};
    ptr1->setNome("Pedro");

    std::cout << ptr1->getNome() << std::endl;
    //ptr1 = &p2;

    return 0;
}
```

Finalmente

O que isso significa?

```
const Pessoa* const ptr1{&p1};
```

Ponteiro const para dados const

Ponteiro const para dados const.

O ponteiro não pode trocar de objeto apontado.

Não podemos alterar os dados do objeto apontado.

Declarado adicionando **const antes e após o tipo da variável.**

Ambas linhas geram erro de compilação.

```
int main(){
    Pessoa p1{"João"};
    Pessoa p2{"Maria"};

    const Pessoa* const ptr1{&p1};
    //ptr1->setNome("Pedro");

    std::cout << ptr1->getNome() << std::endl;
    //ptr1 = &p2;

    return 0;
}
```

Exemplo

A função membro retorna um ponteiro `const` (o objeto não pode ser alterado através do ponteiro) para uma pessoa.

A função em si também não altera o estado do objeto.

```
const Pessoa* getProfessor() const;
```

constexpr

Introduzido no C++11 e melhorado no C++14.

Similar a um const, mas informa o compilador para tentar resolver em **tempo de compilação.**

Exemplo: para uma variável estática `constexpr`, o compilador vai tentar usar o valor da variável diretamente, sem precisar carregar da memória.

Algo similar a um `#define`, onde o valor da macro é substituído no uso.

https://en.cppreference.com/w/cpp/language/constexpr

constexpr

`constexpr` pode avaliar o resultado de uma função em tempo de compilação.

Exemplo:

```
constexpr int length_a = sizeof(a)/sizeof(int);
```

Quando aplicado a uma variável estática, `constexpr` implica em `inline` e `const`.

A partir do C++17.

Exemplo

Veja o assembly das versões do programa a seguir com const static, e constexpr static para a variável TAM.

```
g++ -S main.cpp
```

Com const

```cpp
#ifndef CONSTANTES_HPP
#define CONSTANTES_HPP

#include <cstdlib>

class Constantes{
   public:
       static std::size_t calcularMaxTam (std::size_t val){
           if (val % 4 == 0)
               return val;
           else
               return val + (4-val%4);
       }
};
#endif
```

```cpp
#include <iostream>
#include <cstdlib>

#include "Constantes.hpp"

//uma variável global para uma constante

const std::size_t TAM{Constantes::calcularMaxTam(97)};

int main(){
   std::cout << TAM << "\n";

   return 0;
}
```

Com constexpr

```
#ifndef CONSTANTES_HPP
#define CONSTANTES_HPP

#include <cstdlib>

class Constantes{
   public:
       constexpr static std::size_t calcularMaxTam (std::size_t val){
           if (val % 4 == 0)
               return val;
           else
               return val + (4-val%4);
       }
};
#endif
```

```
#include <iostream>
#include <cstdlib>

#include "Constantes.hpp"

//uma variável global para uma constante

constexpr std::size_t TAM{Constantes::calcularMaxTam(97)};

int main(){
   std::cout << TAM << "\n";

   return 0;
}
```

constexpr

```
        .file   "main.cpp"
        .text
        .local  _ZStL8__ioinit
        .comm   _ZStL8__ioinit,1,1
        .section        .rodata
        .align 8
        .type   _ZL3TAM, @object
        .size   _ZL3TAM, 8
_ZL3TAM:
        .quad   100
.LC0:
        .string "\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB1732:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $100, %esi
        leaq    _ZSt4cout(%rip), %rax
        movq    %rax, %rdi
        call    _ZNSolsEm@PLT
        movq    %rax, %rdx
        leaq    .LC0(%rip), %rax
        movq    %rax, %rsi
        movq    %rdx, %rdi
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PK
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1732:
        .size   main, .-main
        .type   _Z41__static_initialization_and_destruction_0ii, @funct
_Z41__static_initialization_and_destruction_0ii:
.LFB2231:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
```

const

```
        .weak   _ZN10Constantes14calcularMaxTamEm
        .type   _ZN10Constantes14calcularMaxTamEm, @function
_ZN10Constantes14calcularMaxTamEm:
.LFB1731:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        andl    $3, %eax
        testq   %rax, %rax
        jne     .L2
        movq    -8(%rbp), %rax
        jmp     .L3
.L2:
        movq    -8(%rbp), %rax
        andq    $-4, %rax
        addq    $4, %rax
.L3:
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1731:
        .size   _ZN10Constantes14calcularMaxTamEm, .-_ZN10Constantes14
        .local  _ZL3TAM
        .comm   _ZL3TAM,8,8
        .section        .rodata
.LC0:
        .string "\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB1732:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    _ZL3TAM(%rip), %rax
        movq    %rax,.%rsi
        leaq    _ZSt4cout(%rip), %rax
```

Regra de ouro

Em C++ **não use** #define para definir constantes.

- Dificulta a compilação e detecção de erros.
- Macros criadas via define não possuem tipo (logo, não possuem checagem de tipo).
- O que acontece quando existe uma colisão de nomes de macros?
  - Macros não possuem escopo!

Quando precisar definir uma constante, defina como uma variável `constexpr static`.

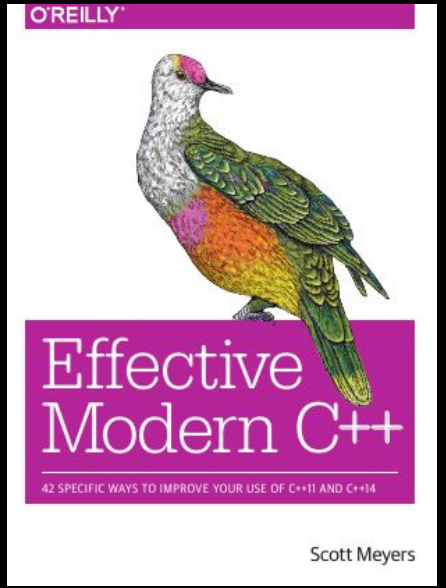
- A variável pode ter escopo (e.g., pertencer a uma classe), ser pública, privada, ...

Leia

Veja as diretivas do Google sobre uso de `const` e `constexpr`:

https://google.github.io/styleguide/cppguide.html#Use_of_const

Leia “Item 15: Use constexpr whenever possible.” de Effective Modern C++: S. D., Meyers.



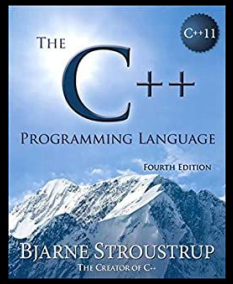
Exercícios

1. Adicione o modificador const em todos os trechos do seu projeto onde isso fizer sentido (retornos const, funções const, parâmetros const, ponteiros const, ...).
2. Considere o protótipo da função membro a seguir e explique o objetivo de cada const na função:

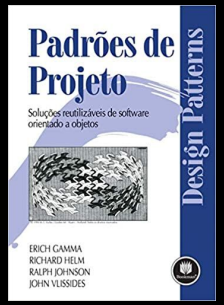
```
const double* calcularImposto(const Investimento* const inv) const;
```

Referências

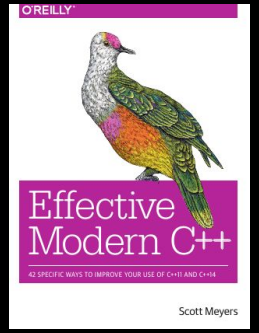
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



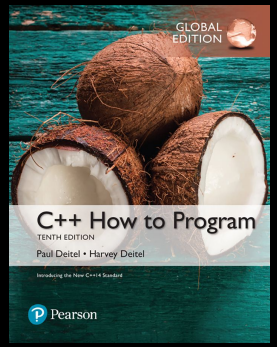
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



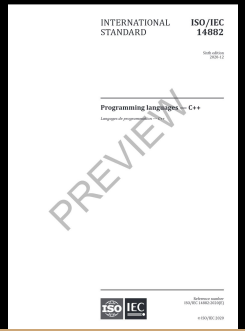
Meyers, S. D. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media. 2014.



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

Esta obra está licenciada com uma Licença Creative Commons Atribuição 4.0 Internacional.