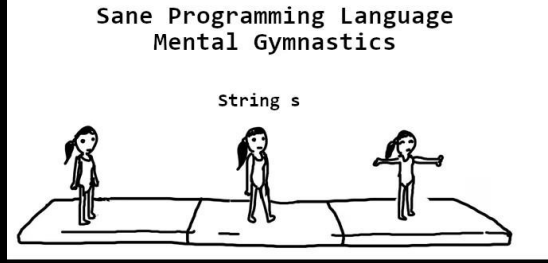
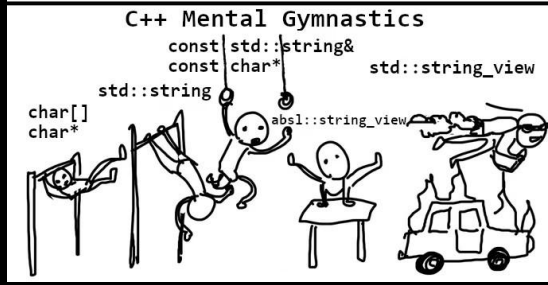




Sobrecarga de operadores

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Considere o exemplo

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

int main(){
   std::string s{"Disciplina de"};
   s+= " Orientação a Objetos";

   std::cout << s << '\n';

   return 0;
}
```

Considere o exemplo

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

int main(){
   std::string s{"Disciplina de"};
   s+= " Orientação a Objetos";

   std::cout << s << '\n';

   return 0;
}
```

O que significa “somar” uma string a outra?

Como esse tipo de comportamento é definido na linguagem.

Sobrecarga de operadores

A maioria dos operadores de C++ podem ser sobrecarregados:

```
++, --, ==, >, <, >=, etc.
```

Sobrecarga de operadores

A maioria dos operadores de C++ podem ser sobrecarregados:

```
++, --, ==, >, <, >=, etc.
```

Com algumas poucas exceções de operadores que não podem ser sobrecarregados:

```
.
.*
::
?:
```

Algumas regras

- Coisas que não são possíveis modificar via sobrecarga de operadores:
  - Precedência dos operadores;
  - Associatividade;
  - Aridade (número de operandos).
- Apenas operadores existentes podem ser sobrecarregados.
  - Não é possível criar novos operadores.
- Não é possível sobrecarregar operadores para tipos primitivos.
- Operadores relacionados, como **+** e **+=** precisam ser sobrecarregados separadamente.

Exemplo

O que estamos comparando?

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

#include "Pessoa.hpp"
#include "CPF.hpp"

int main(){
    ufpr::CPF cpf1{11111111111};
    ufpr::CPF cpf2{11111111111};

    if(cpf1 == cpf2){
        std::cout << "Iguais\n";
    }else{
        std::cout << "Diferentes\n";
    }

    return 0;
}
```

Exemplo

Impossível comparar:

```
no match for ‘operator==’
(operand types are ‘ufpr::CPF’ and ‘ufpr::CPF’)
```

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

#include "Pessoa.hpp"
#include "CPF.hpp"

int main(){
    ufpr::CPF cpf1{11111111111};
    ufpr::CPF cpf2{11111111111};

    if(cpf1 == cpf2){
        std::cout << "Iguais\n";
    }else{
        std::cout << "Diferentes\n";
    }

    return 0;
}
```

Sobrecargas como função membro

É possível sobrecarregar o operador como uma **função membro**, ou como uma **função não membro**.

Vamos começar com funções membro (caso mais comum e simples).

Sobrecarregando o operador ==

CPF.hpp

```
#ifndef CPF_HPP
#define CPF_HPP

namespace ufpr{
class CPF{
   public:
       CPF(const unsigned long numero);
       virtual ~CPF() = default;
       unsigned long getNumero() const;
       void setNumero(const unsigned long numero);

       bool operator==(const CPF& outro) const;

   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

CPF.cpp

```
#include "CPF.hpp"

#include "CPFInvalidoException.hpp"


//...

bool CPF::operator==(const CPF& outro) const{
   return this->numero == outro.numero;
}
```

Exemplo

O compilador vai transformar em:
`cpf1.operator==(cpf2)`

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

#include "CPF.hpp"

int main(){
   ufpr::CPF cpf1{11111111111};
   ufpr::CPF cpf2{11111111111};

   if(cpf1 == cpf2){
       std::cout << "Iguais\n";
   }else{
       std::cout << "Diferentes\n";
   }

   return 0;
}
```

Faça você mesmo

Sobrecarregue agora os operadores `!=`, `<`, `>`, `<=`, e `>=`.

Exemplo

```
class CPF{
   public:
       CPF(const unsigned long numero);
       virtual ~CPF() = default;
       unsigned long getNumero() const;
       void setNumero(const unsigned long numero);

       bool operator==(const CPF& outro) const;
       bool operator!=(const CPF& outro) const;

       bool operator<(const CPF& outro) const;
       bool operator>(const CPF& outro) const;
       bool operator<=(const CPF& outro) const;
       bool operator>=(const CPF& outro) const;

   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
```

```
#include "CPF.hpp"

//...

bool CPF::operator==(const CPF& outro) const{
   return this->numero == outro.numero;
}

bool CPF::operator!=(const CPF& outro) const{
   return !(*this==outro);
}

bool CPF::operator<(const CPF& outro) const{
   return this->numero < outro.numero;
}

bool CPF::operator>(const CPF& outro) const{
   return (outro < *this);
}

bool CPF::operator<=(const CPF& outro) const{
   return !(*this > outro);
}

bool CPF::operator>=(const CPF& outro) const{
   return !(*this < outro);
}
```

STL e Sobrecarga

Com as funções sobrecarregadas, você agora pode desfrutar de estruturas mais poderosas da STL.

Obs.: para usar a maioria das estruturas, basta sobrecarregar ‹.

Mas é uma **boa prática** sobrecarregar os demais comparadores.

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

#include "CPF.hpp"

int main(){
   ufpr::CPF cpf1{44444444444};
   ufpr::CPF cpf2{22222222222};
   ufpr::CPF cpf3{88888888888};
   ufpr::CPF cpf4{11111111111};

   std::set<ufpr::CPF> meuSet;
   meuSet.insert(cpf1);
   meuSet.insert(cpf2);
   meuSet.insert(cpf3);
   meuSet.insert(cpf4);

   std::set<ufpr::CPF>::const_iterator
            it{meuSet.begin()};
   for( ; it != meuSet.end(); ++it)
       std::cout << it->getNumero() << " ";
   std::cout << '\n';

   return 0;
}
```

Atribuição

A sobrecarga do operador de atribuição pode ser feita na forma:

```
#ifndef CPF_HPP
#define CPF_HPP

namespace ufpr{
class CPF{
   public:

       //...

       const CPF& operator=(const CPF& outro);
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

```
#include "CPF.hpp"

using namespace ufpr;

//...

const CPF& CPF::operator=(const CPF& outro) {
   //cuidado com auto atribuição
   if (&outro != this) {
       this->numero = outro.numero;
   }
   return *this;  // para permitir x = y = z
}
```

Atribuição

```cpp
#include "CPF.hpp"

using namespace ufpr;

//...

const CPF& CPF::operator=(const CPF& outro) {
    //cuidado com auto atribuição
    if (&outro != this) {
        this->numero = outro.numero;
    }
    return *this;  // para permitir x = y = z
}
```

Teste de auto-atribuição. Nesse caso seria mais eficiente não fazer o teste (mas fica como exemplo). Mas em classes complexas, evita a cópia desnecessária de elementos quando um objeto é atribuído a ele mesmo.

Atribuição

```cpp
#include "CPF.hpp"

using namespace ufpr;

//...

const CPF& CPF::operator=(const CPF& outro) {
   //cuidado com auto atribuição
   if (&outro != this) {
       this->numero = outro.numero;
   }
   return *this;  // para permitir x = y = z
}
```

Sempre retorne uma referência constante para permitir construções do tipo x=y=z=...

Aceitando inteiros

Para essa versão precisamos fazer verificações extras, quais?

```
#ifndef CPF_HPP
#define CPF_HPP

namespace ufpr{
class CPF{
   public:

       //...

       const CPF& operator=(const CPF& outro);
       const CPF& operator=(const unsigned long numero);
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

Aceitando inteiros

```
#ifndef CPF_HPP
#define CPF_HPP

namespace ufpr{
class CPF{
   public:

       //...

       const CPF& operator=(const CPF& outro);
       const CPF& operator=(const unsigned long numero);
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

```
#include "CPF.hpp"

using namespace ufpr;

//...

const CPF& CPF::operator=(const unsigned long numero){
     if (!validarCPF(numero))
       throw CPFInvalidoException{numero};
   this->numero = numero;
   return *this;
}
```

Faça você mesmo

Coloque `couts` nos operadores de atribuição sobrecarregados e verifique quando qual deles é chamado no trecho a seguir.

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

#include "CPF.hpp"

int main(){
   ufpr::CPF cpf1{44444444444};

   std::cout << cpf1.getNumero() << '\n';

   ufpr::CPF cpf2{22222222222};
   cpf1 = cpf2;
   std::cout << cpf1.getNumero() << '\n';

   cpf1 = 33333333333;
   std::cout << cpf1.getNumero() << '\n';

   cpf1 = cpf2 = 44444444444;
   std::cout << cpf1.getNumero()
      << ' ' << cpf2.getNumero() << '\n';

   return 0;
}
```

Sobrecarga como funções não membro

É possível sobrecarregar operações como funções não-membro.

Sobrecarga do operador «

```
#ifndef CPF_HPP
#define CPF_HPP

#include <iostream>

namespace ufpr{
class CPF{
   friend std::ostream& operator<<(std::ostream& stream, const CPF& cpf);

   public:
      //...
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

Sobrecarga do operador «

Pelo bem da eficiência, declaramos a função como amiga, para acessar diretamente os membros da classe sem precisar passar pelos gets.

```
#ifndef CPF_HPP
#define CPF_HPP

#include <iostream>

namespace ufpr{
class CPF{
   friend std::ostream& operator<<(std::ostream& stream, const CPF& cpf);

   public:
      //...
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

Sobrecarga do operador «

`std::cout << cpf` é montado pelo compilador na forma `operator<<(cout, cpf)`. A única forma de definir uma função que aceita esse protótipo é através de uma função não membro.

Operadores binários podem ser funções membro apenas se o operando esquerdo é da mesma classe da qual a função é membro (o que não é o caso).

```
#ifndef CPF_HPP
#define CPF_HPP

#include <iostream>

namespace ufpr{
class CPF{
   friend std::ostream& operator<<(std::ostream& stream, const CPF& cpf);

   public:
      //...
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

Sobrecarga do operador «

A função precisará ser definida dentro do namespace ufpr!

```
#ifndef CPF_HPP
#define CPF_HPP

#include <iostream>

namespace ufpr{
class CPF{
   friend std::ostream& operator<<(std::ostream& stream, const CPF& cpf);

   public:
      //...
   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

Sobrecarga do operador «

```cpp
#include <iostream>

#include "CPF.hpp"

int main(){
   //abra um gerador de cpf e coloque
   //um cpf que começa com 0
   ufpr::CPF cpf1{11111111111};
   std::cout << cpf1 << '\n';

   return 0;
}
```

```cpp
#include "CPF.hpp"

#include <iomanip>

#include "CPFInvalidoException.hpp"

namespace ufpr {

//...

std::ostream& operator<<(std::ostream& stream, const ufpr::CPF& cpf) {
   unsigned int verificador{(unsigned int)(cpf.numero % 100)};
   unsigned long prim{cpf.numero / 100};
   unsigned int ter{(unsigned int)(prim % 1000)};
   prim /= 1000;
   unsigned int seg{(unsigned int)(prim % 1000)};
   prim /= 1000;

   stream << std::setw(3) << std::setfill('0');
   stream << prim << '.' << seg << '.' << ter << '-' << verificador;
   return stream;  // permitir cout << a << b << c;
}
}  // namespace ufpr
```

Sobrecarga do operador «

Fazer com using namespace ... agora resultaria em um erro. Mais uma vez **cuidado com os using!**

```cpp
#include <iostream>

#include "CPF.hpp"

int main(){
   //abra um gerador de cpf e coloque
   //um cpf que começa com 0
   ufpr::CPF cpf1{11111111111};
   std::cout << cpf1 << '\n';

   return 0;
}
```

```cpp
#include "CPF.hpp"

#include <iomanip>

#include "CPFInvalidoException.hpp"

namespace ufpr {

//...

std::ostream& operator<<(std::ostream& stream, const ufpr::CPF& cpf) {
   unsigned int verificador{(unsigned int)(cpf.numero % 100)};
   unsigned long prim{cpf.numero / 100};
   unsigned int ter{(unsigned int)(prim % 1000)};
   prim /= 1000;
   unsigned int seg{(unsigned int)(prim % 1000)};
   prim /= 1000;

   stream << std::setw(3) << std::setfill('0');
   stream << prim << '.' << seg << '.' << ter << '-' << verificador;
   return stream;  // permitir cout << a << b << c;
}
}  // namespace ufpr
```

Ideia de Jerico

O C++ vai aceitar a construção a seguir.

O que estamos tentando fazer?

```
#ifndef CPF_HPP
#define CPF_HPP

namespace ufpr{
class CPF{
   public:
       CPF(const unsigned long numero);
       virtual ~CPF() = default;
       unsigned long getNumero() const;
       void setNumero(const unsigned long numero);

       bool operator==(const CPF& outro) const;
       bool operator==(const unsigned long& outro) const;

   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

Ideia de Jerico

O C++ vai aceitar a construção a seguir.

Desejamos ser capazes de testar se um cpr é igual a outro, e também se um cpf é igual a um long.

A ideia parece muito boa, mas ...

Você consegue enxergar os problemas?

```
#ifndef CPF_HPP
#define CPF_HPP

namespace ufpr{
class CPF{
   public:
       CPF(const unsigned long numero);
       virtual ~CPF() = default;
       unsigned long getNumero() const;
       void setNumero(const unsigned long numero);

       bool operator==(const CPF& outro) const;
       bool operator==(const unsigned long& outro) const;

   private:
       bool validarCPF(unsigned long cpfTeste) const;
       unsigned long numero;
};
}
#endif
```

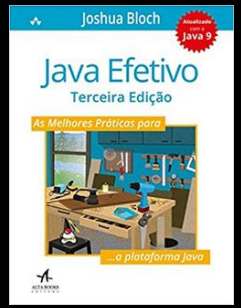
Regras

Uma comparação de igualdade (regras similares são aplicadas a comparadores ‹, ›, !=, ...) devem seguir algumas regras.

Considerando três objetos, X, Y e Z, uma igualdade deve ser:

- Reflexiva: X == X deve ser verdadeiro.
- Simétrica: Se X == Y é verdadeiro, então Y == X é verdadeiro.
- Transitiva: Se X == Y é verdadeiro, e Y == Z é verdadeiro, então X ==Z é verdadeiro.
- Consistente: Múltiplas chamadas de X == Y deve retornar sempre o mesmo resultado se os objetos não foram alterados.

Leia mais sobre isso em Bloch, (2018). Effective Java.



Exercícios

1. Na classe CPF, sobrecarregue o operador **>>** usado, por exemplo, pelo cin.
2. Na classe CPF, sobrecarregue o operador **[ ]**, de forma que seja possível **acessar** um dígito do cpf via índice (como se fosse um vetor). Caso seja acessado um índice inválido, lance uma exceção do tipo out_of_range.
    a. O protótipo é `unsigned short operator[](const int idx) const;`
        i. Note que é const. Logo o valor retornado é uma cópia.

    Para possibilitar uma construção do tipo cpf[3] = 5, seria necessário também sobrecarregar o operador:

    ```
    unsigned short& operator[](const int idx);
    ```

    Note que uma referência é retornada. **Não** sobrecarregue esse operador para cpf porque:

    Seria especialmente complicado retornar uma referência modificável da posição do cpf em que não fosse possível fazer besteiras do tipo cpf[1] = 123;

    É estranho modificar um dígito único de um cpf já pronto na memória (ex.: como validar o novo número formado?)

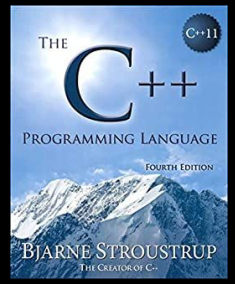
3. Sobrecarregue os operadores de comparação para a classe Pessoa. Como definir se dois objetos são a mesma pessoa?
4. Na classe CPF, Sobrecarregue o operador **( )** para que seja possível uma construção do tipo:

```
Cpf c{12345678912}; //cpf 123.456.789-12
unsigned long trecho{cpf(3,3)};//pegue 3 dígitos do cpf a partir da posição 3
std::cout << trecho << ' ' <<  cpf(7,2) << '\n'; \\ vai imprimir 456 89
```

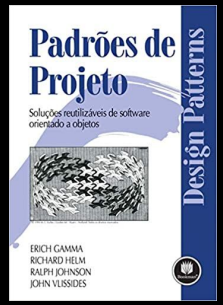
Pesquise sobre como sobrecarregar esse operador. Lance exceções corretamente em caso de argumentos inválidos.

Referências

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

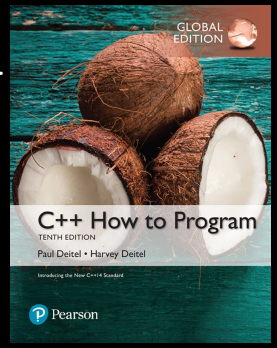


Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.

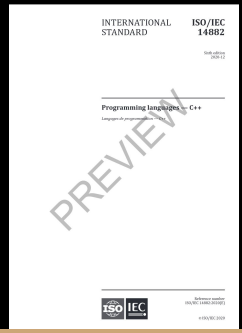


Bloch, J. Effective Java. Addison-Wesley. 2018.

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