

SOBRECARGA

Sobrecarga de métodos

Sobrecarga de operadores
Fundamentos

Restrições

Operador vs. friends

Estudo de casos

SOBRECARGA DE MÉTODOS / FUNÇÕES

- Métodos com mesmo nome mas **assinaturas diferentes**: Tipos, ordem ou número de parâmetros diferentes

- Métodos com a mesma assinatura não podem ter tipos de retorno diferentes

Tipo_retorno **metodoX**(*tipo1* y);

Tipo_retorno **metodoX**(*tipo1* y, *tipo2* z);

Tipo_retorno **metodoX**(*tipo1* y, *tipo3* w);

Tipo_retorno **metodoX**(*tipo2* y, *tipo1* z);

- Compilador chama o método adequado examinando a assinatura

Exemplo: fração

JÁ USAMOS SOBRECARGA DE MÉTODOS...

- Sobrecarga de construtores
 - Construtor sem parâmetros/ construtor com parâmetros
- Data casamento;
- Data nascimento(10,5,2010);

CONSTRUTOR DE CÓPIA

- Usado sempre que uma cópia de um objeto for necessária
 - Passagem por valor ou retorno por valor
 - Inicializar um objeto com uma cópia de outro
 - `ClasseX novoObj(velhoObj);`
 - `novoObj` é cópia de `velhoObj`
- Protótipo para classe `ClasseX`
 - `ClasseX(const ClasseX &);`
 - Deve obter referência
 - Do contrário seria passagem por valor
 - Tenta fazer cópia chamando o construtor de cópia
 - Loop infinito

Muito importante quando alocação dinâmica é utilizada

Exemplo: vetor

OPERADORES COM OBJETOS

- Uso de operadores c/ objetos
 - Ex.: Como somar 2 frações?

SOBRECARGA DE OPERADORES

- Uso de operadores c/ objetos
 - Utilização de operadores da linguagem para manipular objetos
 - Clareza
 - Operadores são sensíveis ao contexto
- Exemplo
 - +
 - Operações aritméticas (integers, floats, etc.)
 - Classe matriz:
 - Matriz A, B, C;
 - `C = A+B;`

FUNDAMENTOS DE SOBRECARGA DE OPERADORES

- Sobrecarga apresenta uma notação mais clara e objetiva:

Considere a criação de um método soma:

```

• objeto3 = objeto1.soma(objeto2);
  ○ Passando o objeto 2 para o método soma do objeto1

• objeto3 = objeto1 + objeto2;
  ○ = objeto1.operator+(objeto2)
    
```

FUNDAMENTOS DE SOBRECARGA DE OPERADORES

- Sobrecarga de operadores: Como fazer?
 - Crie uma função para a classe/tipo de dados
 - Nomeie a função com o nome **operator** seguida pelo símbolo.
 - **Operator+** para o operador de adição

```

• objeto3 = objeto1 + objeto2;

• = objeto1.operator+(objeto2)
    
```

RESTRIÇÕES

- Não se pode mudar
 - Como os operadores trabalham com os tipos **pré-definidos**
 - Soma de inteiros deve ser sempre soma de inteiros
 - Precedência de operadores (ordem de avaliação da expressão)
 - Use parênteses p/ forçar mudança na ordem
 - Associatividade (esq-p/-direita ou direita-p/-esq)
 - Número de operandos
 - & é unário, ou seja, tem somente um operando
- Não se pode criar novos operadores
- Operadores devem ser sobrecarregados explicitamente
 - Sobrecarga de + não sobrecarrega +=

RESTRIÇÕES

Operadores que podem ser sobrecarregados							
+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<=	>>=	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operadores que não podem ser sobrecarregados				
.	.*	::	?:	sizeof

MEMBROS X FRIENDS

- Funções Operadores
 - Funções Membro/ métodos (Preferencialmente)
 - Use **this** p/ obter argumentos implicitamente
 - Obtem operando **à esquerda** para operador binário (como *)
 - Objeto mais à esquerda deve ser da mesma classe que o operador
 - Funções não membro/globais
 - Necessitam parâmetros para ambos os operandos
 - Podem ter objeto de classe diferente da do operador
 - Tem que ser **friend** para acessar dados **private** ou **protected**

CLASSE PARA EXEMPLO

```

class Ponto
{
    public:
        Ponto(int x1=0, int y1=0) {
            x = x1;
            y = y1;
        }
        void print();

    private:
        int x,y;
};
    
```

SOBRECARGA ++ E --

- Retorna valores
 - Pré-incremento
 - Retorno por referência (Ponto &)
 - Constante pode ser atribuída
 - Pós-incremento
 - Retorno por valor
 - Retorna objeto temporário com valor antigo

SOBRECARGA DE OPERADORES UNÁRIOS

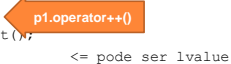
Sobrecarga de incremento pré-fixado

- Ponto &operator++();

```
Ponto & operator++() {  
    ++x;  
    ++y;  
    return *this;  
}
```

Exemplo de uso:

```
p2 = ++p1;  
(++p2).print();  
++p1 = ...
```



SOBRECARGA DE OPERADORES UNÁRIOS

Opção:

Sobrecarga de incremento pré-fixado usando função global

```
friend Ponto &operator++(Ponto &);//dentro da classe
```

```
Ponto &operator++(Ponto &p1){//Fora da classe  
    ++p1.x;  
    ++p1.y;  
    return p1;  
}
```

```
cout << ++p1;
```



SOBRECARGA DE OPERADORES UNÁRIOS

Sobrecarga de incremento pós-fixado

Pós-incremento tem um parâmetro "fantasma" int
- diferenciar de pré-incremento

```
Ponto operator++(int);
```

```
Ponto operator++(int) {  
    Ponto pAntesInc(x, y);  
    ++x;  
    ++y;  
    return pAntesInc;  
}
```

p1++ -> rvalue (não usa referência)

SOBRECARGA DE OPERADORES BINÁRIOS

Sobrecarga de soma entre 2 objetos da classe ponto

```
Ponto operator+(Ponto p2) const {  
    return Ponto(x+p2.x , y+p2.y);  
}
```


`return Ponto(((*this).x+p2.x , (*this).y+p2.y);`

MEMBROS X FRIENDS

Operadores Comutativos

- Vc pode querer que o operador + seja comutativo
 - Assim "a + b" e "b + a" tem que produzir mesmo resultado
- Suponha que tenhamos duas classes diferentes
- Sobrecarga do operador pode ser função membro somente quando sua classe está à esquerda
 - ClassePonto + int
- Quando for ao contrário precisa ser não membro
 - int + ClassePonto

SOBRECARGA DE OPERADORES BINÁRIOS

Sobrecarga de soma entre ponto e inteiro:

Ponto + int : p3 = p1 + 5

```
Ponto operator+(int z) const {  
    return Ponto(x+z , y+z);  
}
```

p3 = p1.operator+(5)

Obtém operando à esquerda para operador binário

SOBRECARGA DE OPERADORES BINÁRIOS

Sobrecarga de soma entre inteiro e ponto:

int + Ponto: p3 = 5 + p1 ???

p3 = 5.operator+(p1)

Obtém operando à esquerda para operador binário

SOBRECARGA DE OPERADORES BINÁRIOS

Sobrecarga de soma entre inteiro e ponto:

p3 = 5 + p1

```
friend Ponto operator+(int z, Ponto p);  
.  
.  
.  
Ponto operator+(int z, Ponto p) {  
    return Ponto(z + p.x, z + p.y );  
}
```

EXERCÍCIO: * PARA A CLASSE PONTO

Multiplicação para classe Ponto:

(Ponto * Ponto), (Ponto * Inteiro) e (Inteiro * Ponto)

```
Ponto Ponto::operator*(Ponto p2) const {  
    Ponto m;  
    m.x = x * p2.x;  
    m.y = y * p2.y;  
    return m;  
}  
  
Ponto Ponto::operator*(int v) const {  
    Ponto m;  
    m.x = x * v;  
    m.y = y * v;  
    return m;  
}  
  
friend Ponto operator*(int v, Ponto p2); //(dentro da classe)  
Ponto operator*(int v , Ponto p2); //(fora da classe)  
{  
    Ponto m;  
    m = p2 * v;  
    return m;  
}
```

OPERADOR X FRIENDS

○ Sobrecarga de <<

○ Ex.: cout << objPonto;

- Operando à esquerda de << é do tipo ostream &
 - Tal como cout
- Similarmente, sobrecarga >> necessita istream &
- Assim, ambos devem ser não membros

EXEMPLO COM ENTRADA E SAÍDA

○ << e >>

- Já sobrecarregados para cada tipo pré-definido
- Pode também processar uma classe definida pelo usuário

○ Exemplo

- Classe **PhoneNumber**
 - Armazena um nro telefonico
- Imprime no formato
 - (123) 456-7890

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

// Definição da classe
class PhoneNumber {
    friend ostream &operator<<( ostream&, const PhoneNumber & );
    friend istream &operator>>( istream&, PhoneNumber & );
private:
    char ddd [ 4 ];
    char prefixo[ 4 ];
    char numero [ 5 ];
};
```

```
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
    output << "(" << num.ddd << " " <<
        << num.prefixo << "-" << num.numero;

    return output; // permite cout << a << b << c;
}

istream &operator>>( istream &input, PhoneNumber &phone )
{
    input.ignore();
    input >> setw( 4 ) >> num.ddd;
    input.ignore( 2 );
    input >> setw( 4 ) >> num.prefixo;
    input.ignore();
    input >> setw( 5 ) >> num.numero;

    return input; // permite cin
}
```

A expressão:
`cout << phone;`
é interpretada como uma chamada de função:
`operator<<(cout, phone);`

Permite "cascata"
`cout << phone1 << phone2;`
primeiro chama
`operator<<(cout, phone1);`, e
retorna `cout`.
Depois faz `cout << phone2...`

O manipulador `setw`
restringe o número de
caracteres lidos. `setw(4)`
permite 3 caracteres a serem
lidos, permitindo a existência
do caractere null.

```
int main()
{
    PhoneNumber phone; // cria o objeto

    cout << "Entre c/ o nro do telefone (xxx) xxx-xxxx:\n";

    // cin >> phone invoca o operador >>
    // operator>>( cin, phone )
    cin >> phone;

    cout << "O número do telefone é : " ;

    // cout << phone invoca operador<<
    // operator<<( cout, phone )
    cout << phone << endl;

    return 0;
}
```

Entre c/ o nro do telefone (xxx) xxx-xxxx:
(800) 555-1212
O número do telefone é : (800) 555-1212

EXERCÍCIO: << PARA A CLASSE PONTO

```
friend ostream &operator<<( ostream&, const Ponto & );

ostream &operator<<( ostream &output, const Ponto &p )
{
    output << "(" << p.x << " , " << p.y << " ) ";
    return output;
}
```

EXERCÍCIO: == PARA A CLASSE PONTO

```
bool operator==(Ponto p2) const{
    if ( (p2.x==x) and (p2.y == y) )
        return true;
    else
        return false;
};
```

ESTUDO DE CASO

- Arrays em C++
 - Não há verificação de limites
 - Não se compara com ==
 - Não há atribuição
 - Não se pode ler/escrever arrays inteiros de uma única vez
 - Um elemento por vez
- Exemplo: Implemente uma Classe **Array** com as seguintes características
 - Verificação de limites
 - Atribuição
 - Arrays que conheçam o próprio tamanho
 - Entrada e saída com << e >>
 - Comparações com == e !=

ESTUDO DE CASO

- Construtor de cópia
 - Usado sempre que uma cópia de um objeto for necessária
 - Passagem por valor (retorna valor ou parametro)
 - Inicializar um objeto com uma cópia de outro
 - `Array newArray(oldArray);`
 - `newArray` é cópia de `oldArray`
 - Protótipo para classe `Array`
 - `Array(const Array &);`
 - Deve obter referência
 - Do contrário seria passagem por valor
 - Tenta fazer cópia chamando o construtor de cópia
 - Loop infinito

CLASSES COM ALOCAÇÃO DINÂMICA DE MEMÓRIA

- Construtor de cópia, destrutor e operador de atribuição

```
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>

using std::ostream;
using std::istream;

class Array {
    friend ostream &operator<<( ostream &, const Array & );
    friend istream &operator>>( istream &, Array & );
public:
    Array( int = 10 );           // construtor
    Array( const Array & );     // construtor de cópia
    ~Array();                   // destrutor
    int getSize() const;        // tamanho

    // operador de atribuição
    const Array &operator=( const Array & );

    // operador de igualdade
    bool operator==( const Array & ) const;
```

A maioria dos operadores sobrecarregados é função membro (exceto <<e>>).

Protótipo para o construtor de cópia.

```
// oposto do operador ==
bool operator!=( const Array &right ) const
{
    return ! ( *this== right ); // invoca Array::operator==
}

// operador de subscripto p/ o
int &operator[] ( int );

//operador de subscripto para objetos constantes
const int operator[] ( int ) const;

private:
    int size; // tamanho
    int *ptr; // ponteiro para o primeiro elemento do array
};

#endif
```

!= operador simplesmente retorna o oposto do operador ==. Assim, basta definir o operador ==.

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <new>

#include <cstdlib>

#include "array1.h" // Definição da Classe

// construtor padrão, com tamanho default de 10
Array::Array( int arraySize )
{
    // valida o array
    size = ( arraySize > 0 ? arraySize : 10 );

    ptr = new int[ size ]; // aloca o array
```

```
for ( int i = 0; i < size; i++ )
    ptr[ i ] = 0; // inicializa o array

} // fim do construtor

// construtor de cópia
// DEVE receber uma referencia para evitar loop infinito
Array::Array( const Array &arrayToCopy )
: size( arrayToCopy.size )
{
    ptr = new int[ size ]; // aloca array

    for ( int i = 0; i < size; i++ )
        ptr[ i ] = arrayToCopy.ptr[ i ]; // copia p/ objeto

} // fim do construtor de cópia

// destrutor
Array::~Array()
{
    delete [] ptr; // recupera espaço alocado anteriormente
}
```

Você deve declarar um novo array de inteiros para que os objetos não apontem para a mesma posição de memória.

```

int Array::getSize() const
{
    return size;
}

// sobrecarga do operador de atribuição
// retornando const não permite: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
    if ( &right != this )
    { // verifica auto atribuição

        // p/ arrays de tamanhos diferentes, eliminar original
        // e alocar novo array
        if ( size != right.size ) {
            delete [] ptr; // recupera espaço
            size = right.size; // redimensiona o objeto
            ptr = new int[ size ]; // aloca novo espaço
        }

        for ( int i = 0; i < size; i++ )
            ptr[ i ] = right.ptr[ i ]; // copia o array
    }
}

```

```

return *this; // permite x = y = z, por exemplo
}

// determina se dois arrays são iguais
// retornando verdadeiro ou falso
bool Array::operator==( const Array &right ) const
{
    if ( size != right.size )
        return false; // tamanhos diferentes

    for ( int i = 0; i < size; i++ )

        if ( ptr[ i ] != right.ptr[ i ] )
            return false; // elemento diferente-> array dif.

    return true; // iguais
}

```

```

98 // sobrecarga de subscripto
99
100 int &Array::operator[]( int subscript )
101 {
102     // verifica se índice está fora do range
103     if ( subscript < 0 || subscript >= size ) {
104         cout << "\nErro: índice " << subscript
105             << " fora do range" << endl;
106
107         exit( 1 );
108     } // end if
109 } // end if
110
111 return ptr[ subscript ];
112
113 }
114

```

inteiro1[5] chama
inteiro1.operator[](5)

```

116 // sobrecarga para arrays constantes
117 const int &Array::operator[]( int subscript ) const
118 {
119     // verifica limites
120     if ( subscript < 0 || subscript >= size ) {
121         cout << "\nErro: Índice " << subscript
122             << " fora dos limites" << endl;
123
124         exit( 1 ); //
125     }
126 }
127
128 return ptr[ subscript ];
129
130 }
131 // sobrecarga de leitura
132 istream &operator>>( istream &input, Array &a )
133 {
134     for ( int i = 0; i < a.size; i++ )
135         input >> a.ptr[ i ];
136
137     return input; // permite cin >> x >> y;
138
139 }

```

```

142
143 // sobrecarga de saída
144 ostream &operator<<( ostream &output, const Array &a )
145 {
146     int i;
147
148     for ( i = 0; i < a.size; i++ ) {
149         output << setw( 12 ) << a.ptr[ i ];
150
151         if ( ( i + 1 ) % 4 == 0 )
152             output << endl;
153     } // end for
154
155     if ( i % 4 != 0 )
156         output << endl;
157
158     return output; // permite cout << x << y;
159
160 }
161
162 }

```

```

3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "array1.h"
10
11 int main()
12 {
13     Array inteiro1( 7 );
14     Array inteiro2;
15
16     // imprime inteiro1 (tamanho e conteúdo)
17     cout << "Tamanho do array 1 é "
18         << inteiro1.getSize()
19         << "\n Array após inicialização:\n" << inteiro1;
20
21     // imprime inteiro2 (tamanho e conteúdo)
22     cout << "\nTamanho do inteiro 2 é "
23         << inteiro2.getSize()
24         << "\nArray após a inicialização:\n" << inteiro2;
25

```

```

26 // le e imprime inteiro 1 e inteiro 2
27 cout << "\nEntre com 17 inteiros:\n";
28 cin >> inteiro1 >> inteiro2;
29
30 cout << "\nAgora os arrays contém:\n"
31 << "inteiro1:\n" << inteiro1
32 << "inteiro2:\n" << inteiro2;
33
34 // usando (!=)
35 cout << "\n usando != \n";
36
37 if ( inteiro1 != inteiro2 )
38     cout << "inteiro1 e inteiro2 são diferentes\n";
39
40 // criando array inteiros3 usando inteiro1 p/ inicializar
41
42 Array inteiro3( inteiro1 ); // construtor de cópia
43
44 cout << "\nTamanho do array inteiros3 é "
45 << inteiro3.getSize()
46 << "\nApós inicializar:\n" << inteiro3;
47

```

```

49 cout << "\n Atribuição:\n";
50 inteiro1 = inteiro2; // observar diferença de tamanho
51
52 cout << "inteiro1:\n" << inteiro1
53 << "inteiro2:\n" << inteiro2;
54
55 // usando (==)
56 cout << "\n usando == \n";
57
58 if ( inteiro1 == inteiro2 )
59     cout << "inteiro1 e inteiro2 são iguais\n";
60
61 // usando índice
62 cout << "\ninteiro[5] é " << inteiro[ 5 ];
63
64
65 cout << "\n\nAtribuição a inteiro[5]\n";
66 inteiro[ 5 ] = 1000;
67 cout << "inteiro1:\n" << inteiro1;
68
69 // tentando acessar posição fora do array
70 cout << "\ninteiro[15]" << endl;
71 inteiro[ 15 ] = 1000; // ERRO
72
73 return 0;
74 }

```

EXERCÍCIO: NÚMEROS COMPLEXOS

Um número complexo pode ser escrito da forma $z = a + bi$, onde a e b são números reais e i é a unidade imaginária. Crie uma classe para números complexos com todos os métodos necessários para seu bom funcionamento.

Faça a sobrecarga dos seguintes operadores: $+$, $-$, $*$, $++$ (pré e pós fixado), $--$ (pré e pós fixado), $<<$ (saída) e $>>$ (entrada). As operações de adição/subtração/multiplicação devem ser comutativas (pode ser realizada com outro objeto do mesmo tipo (número complexo) e com números reais, em qualquer ordem). Mostre a utilização da classe e dos operadores sobrecarregados em um programa principal.

SOBRECARGA $++$ E $--$

- Distinguindo pre/pos incremento
 - Pos-incremento tem um parametro "fantasma"
 - Protótipo (membro)
 - `Data operator++(int);`
 - `d1++` significa `d1.operator++(0)`
 - Protótipo (não-membro)
 - `friend Data operator++(Data &, int);`
 - `d1++` significa o mesmo que `operator++(d1, 0)`
 - Parametro inteiro não tem nome
 - Nem mesmo na definição