ORIENTAÇÃO A OBJETOS

SISTEMAS DE INFORMAÇÃO

DR. EDNALDO B. PIZZOLATO

SOBRECARGA DE OPERADORES (C++)

A sobrecarga de operadores ocorre quando desejamos utilizar operadores já conhecidos (+,-,*,/,=,+=....) em nossas novas classes.

Assim, podemos criar uma classe fração (x/y) e os operadores relacionados a ela.

Eles permitem que o código fique mais legível (clareza).

É claro que existem muitos outros exemplos além da fração (vetores, matrizes, números complexos...).

Tipos

- Próprios (**`int`**, **`char`**) ou definidos pelo usuário;
- É permitido usar operadores existentes com tipos definidos pelo usuário
  - Não é permitido criar novos operadores

Regras

- Crie uma função para a classe
- Nomeie a função com o nome **`operator`** seguida pelo símbolo.
  - **`Operator+`** para o operador de adição

Ou seja, o nome soma será substituido pelo nome operator+ !!

Regras

Isso significa que antes, quando fazíamos a chamada do método, passávamos o outro elemento a ser adicionado para o método soma. E agora?

```
resultado.atribui(x.soma(y));
resultado = x + y;
```

Regras

A chamada do método atribui é idêntica à chamada do método operator=.

A chamada do método soma é idêntica à chamada do método operator+

```
resultado.atribui(x.soma(y));
resultado.operator=(x.operator+(y));
```

A sintaxe final utilizada pelo programador será: resultado = x + y;

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)
- Numero de operandos
  - **&** é unário, ou seja, tem somente um operando

RESTRIÇÕES

Não se pode criar novos operadores

Operadores devem ser sobrecarregados explicitamente

- Sobrecarga de **+** não sobrecarrega **+=**

Aliás sobrecarga de + é diferente de +=

Programação Orientada a Objetos

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

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

Funções Operadores

- Funções Membro
  - 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 Operadores

- Funções não membro
  - 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`**

O que é friend?

Friends

Se uma classe ou função for "amiga" de outra classe, isso significa que ela poderá acessar os dados da classe.

É uma propriedade que pode "quebrar" a integridade da classe se a "amiga" abusar da amizade e fizer algo errado...

Friends (propriedades)

- "amizade" é oferecida, não pega
  - Classe **`B friend`** da **`A`**
    - Classe **`A`** deve explicitamente declarar **`B`** como **`friend`**
- Não simétrica
  - Classe **`B friend`** da **`A`**
  - Classe **`A`** não necessariamente **`friend`** da **`B`**
- Não transitiva
  - Classe **`A friend`** da **`B`**
  - Classe **`B friend`** da **`C`**
  - Classe **`A`** não necessariamente **`friend`** da **`C`**

Exemplo:

```
class Contador {
     friend void setX( Contador &, int ); // friend
private:
     int x;
public:
      // construtor
        Contador() : x( 0 )  // inicializa x c/ 0
        {  // bla bla }
}; // fim da classe
```

Exemplo:

```
void setX( Contador &c, int val )
{
    c.x = val;  // OK: setX é friend de Contador
} // fim de setx
```

Observe que o objeto c da classe contador é passado para a função setX e que a atribuição ao x é feita sem a verificação do valor val.

Retornando à sobrecarga...

Assim, para as funções não membros, o acesso aos dados só será permitido se forem "amigas"!

Por exemplo, se desejarmos sobrecarregar os operadores de entrada >> e saída << devemos sinalizar que serão friends da classe que estamos construindo...

<< e >>

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

Exemplo

- Classe **`NumeroTelefone`**
  - Armazena um número de telefone
- Imprime no formato
  - **`(123) 456-7890`**

Na definição da classe...

```
class NumeroTelefone
{
   friend ostream &operator<<( ostream&,
   const NumeroTelefone & );

   friend istream &operator>>( istream&,
   NumeroTelefone & );
   ...
```

Fora da classe...

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

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

Fora da classe...

```
istream &operator>>( istream &input, NumeroTelefone &num )
{
     input.ignore();                    // pula o símbolo (
     input >> setw( 4 ) >> num.ddd;     // entrar com o ddd
     input.ignore( 2 );                 // pula o símbolo ) e o espaço
     input >> setw( 4 ) >> num.prefixo; // entrar com prefixo
     input.ignore();                    // pula (-)
     input >> setw( 5 ) >> num.numero;  // entrar com número

     return input;     // permite cin >> a >> b >> c;
}
```

Programação Orientada a Objetos

E no programa principal...

```
int main()
 {
     NumeroTelefone  x; // cria o objeto x
     cout << "Entre c/ o nro do telefone (xxx) xxx-xxxx:\n";
     // cin >>  x invoca o operador >>
     // chamada equivale a operator>>( cin, x )
     cin >> x ;
     cout << "O número do telefone é : " ;
     // cout << x invoca operador<<
     // chamada equivale a operator<<(cout, x )
     cout << x << endl;
     return 0;
 }
```

OPERADOR UNÁRIO

Sobrecarga de **!** para testar strings vazias

- s.operator!()

```
class String {
 public:
    bool operator!() const;
    ...
};
```

- operator!(s)

```
class String {
   friend bool operator!( const String & )
   ...
}
```

OPERADOR BINÁRIO

```
class String
{
public:
    const String &operator+=(
  const String & );
    ...
};
```

`y += z` equivale a `y.operator+=( z )`

OPERADOR BINÁRIO

```
class String
{
    friend const String &operator+=(
                String &, const String & );
    ...
};
```

y += z equivale a operator+=(y, z)

ESTUDO DE CASO
ARRAYS

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 **VETOR** com as seguintes características

- Verificação de limites
- Atribuição
- Arrays que conheçam o próprio tamanho
- Entrada e saída com **<<** e **>>**
- Comparações com **==** e **!=**

Pronto para começar?

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
    - `VETOR NOVOVETOR(VELHO); //NOVOVETOR` é cópia de `VELHO`

Construtor de cópia

- Protótipo para classe VETOR
  - **`VETOR( const VETOR & );`**
  - *Deve* obter referência
    - Do contrario seria passagem por valor
    - Tenta fazer copia chamando o construtor de copia
    - Loop infinito

```
class VETOR {
    friend ostream &operator<<( ostream &, const VETOR & );
    friend istream &operator>>( istream &, VETOR & );

  public:
    VETOR( int = 10 );                    // constructor padrão
    VETOR( const Array & );               // construtor de cópia
    ~VETOR();                             // destrutor
    int getTam() const;                   // tamanho

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

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

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

    // operador de subscrito p/ objetos não constantes
    int &operator[]( int );

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

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

```
VETOR::VETOR( int T )
{
      // valida o array
      tam = ( T > 0 ? T : 10 );
      ptr = new int[ tam ]; // aloca o array
      for ( int i = 0; i < tam; i++ )
         ptr[ i ] = 0;   // inicializa o array
}
```

Construtor padrão

```
VETOR::VETOR( const VETOR &v ):tam(v.tam)
{
      ptr = new int[ tam ];
     for ( int i = 0; i < tam; i++ )
        ptr[ i ] = v[i];   // copia o array
}
```

Construtor de cópia

```
VETOR::~VETOR( )
{
      delete [] ptr;
}
```

Destrutor

```
VETOR::getTam( )
{
    return tam;
}
```

Programação Orientada a Objetos

```
// sobrecarga do operador de atribuição
// retornando const não permite: ( a1 = a2 ) = a3
const VETOR &VETOR::operator=( const VETOR &direita )
{
   if ( &direita != this ) {  // verifica auto atribuição
         // p/ arrays de tamanhos diferentes, eliminar original
        // e alocar novo array
        if ( tam != direita.tam ) {
              delete [] ptr;                    // recupera espaço
              tam = direita.tam;                // redimensiona o objeto
              ptr = new int[ tam ];             // aloca novo espaço
         }
         for ( int i = 0; i < tam; i++ )
              ptr[ i ] = direita.ptr[ i ];      // copia o array
    }
   return *this;
}
```

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

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

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

   return true;        // iguais
}
```

Pode-se criar a sobrecarga do operador != utilizando a chamada do operador ==

```
// sobrecarga de leitura
istream &operator>>( istream &in, VETOR &a )
{
   for ( int i = 0; i < a.tam; i++ )
      in >> a.ptr[ i ];
    return in;   // permite cin >> x >> y;
 }
```

Programação Orientada a Objetos

```
int main()
 {
      VETOR X1( 17 );
      VETOR X2;

      // imprime X1 (tamanho e conteudo)
          cout << "Tamanho do VETOR X1 é  "
          << X1.getTam()
          << "\n Vetor após inicialização:\n" << X1;
       ...
     if ( X1 != X2 )
          cout << "X1 e X2 sào diferentes\n";
      ...

     VETOR L3( X1 );  // construtor de cópia
     ...
}
```

INCREMENTO E DECREMENTO

- Adicione 1 ao objeto **`XX`**
- Protótipo
  - **`XX &operator++();`**
  - **`++d1`** significa o mesmo que **`d1.operator++()`**
- Protótipo
  - **`Friend XX &operator++(XX &);`**
  - **`++d1`** significa o mesmo que **`operator++(d1)`**

INCREMENTO E DECREMENTO

Distinguindo pre/pós incremento

- Pós-incremento tem um parametro "fantasma"
- Protótipo
  - `XX operator++(int);`
  - `d1++` significa `d1.operator++(0)`
- Protótipo
  - `friend XX operator++(XX &, int);`
  - `d1++` significa o mesmo que `operator++(d1,0)`
- Parametro **inteiro** não tem nome
  - Nem mesmo na definição

INCREMENTO E DECREMENTO

Retorna valores

- Pre-incremento
  - Retorno por referência (**XX &**)
  - Constante pode ser atribuida
- Pos-incremento
  - Retorno por valor
  - Retorna objeto temporário com valor antigo

INCREMENTO E DECREMENTO

Exemplo classe DATA

- Sobrecarga do operador ++

```
DATA &operator++();        // pre-incremento
DATA operator++( int );    // pós-incremento
```

```
// pre-incremento
DATA &DATA::operator++()
{
   Incrementa();
   return *this;
}
```

```
//pós-incremento
DATA DATA::operator++(int)
{
   DATA temp = *this;
   Incrementa();
   return temp;
}
```

Conclusões:

Sobrecarga de operadores é um recurso bastante interessante de C++. Permite que o programador escreva expressões similares às utilizadas com tipos primitivos...

Observe que sobrecarga de operadores não ocorre em Java.

FIM