Overloading de Operadores

João Pascoal Faria (versão original)
Ana Paula Rocha (versão 2004/2005)
Luis Paulo Reis (versão 2005/2006)

Overloading de Operadores

- C++ permite utilizar operadores standard para realizar operações com classes em adição aos tipos fundamentais. Por exemplo:

```
int a, b, c;
a = b + c;
```

- É obviamente válido em C++. No entanto não é tão obvio se isto é válido:

```
struct {
   string produto;
   float preco;
} a, b, c;
a = b + c;
```

Na Realidade conduz a um Erro de Compilação

Funções-operador

- Função com nome **`operator`** seguido do símbolo de um operador de C++ define o significado desse operador para objectos de uma classe (ou combinação de classes)
- Exemplo (para colmatar o facto de, por omissão, não se poderem comparar objectos de uma classe com operadores habituais de comparação):

```
class Data {
  public:
    bool operator==(Data b)
      { return dia == b.dia && mes==b.mes && ano==b.ano; }

  private:
    int dia, mes, ano;
};

... Data d1, d2; ... if (d1 == d2) ...
```


Exemplo de Overload de Operadores

```
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a, int b);
    CVector operator + (CVector param);
};

CVector::CVector (int a, int b) { x = a;  y = b; }

CVector CVector::operator+ (CVector param) {
   return CVector(param.x+x,param.y+y);
}

int main () {
  CVector a (3,1), b (1,2), c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

CVector (int, int); // nome da função CVector (constructor)
CVector operator+ (CVector); // função retorna um CVector

Chamada do Operador:
c = a + b; //ou
c = a.operator+ (b);

Redefinição de Operadores

- Quase todos os operadores podem ser redefinidos:
  - aritméticos: `+ -` *(unário ou binário)* `* / %`
  - bit-a-bit: `^ & | ~ << >>`
  - lógicos: `! && ||`
  - de comparação: `== != > < <= >=`
  - de incremento e decremento: `++ --` *(pósfixos ou préfixos)*
  - de atribuição: `= += -= *= /= %= |= &= ^= ~= <<= >>=`
  - de alocação e libertação de memória: `new new[] delete delete[]`
  - de sequenciação: `,`
  - de acesso a elemento de array: `[]`
  - de acesso a membro de objecto apontado: `-> ->*`
  - de chamada de função: `()`
- Operadores que não podem ser redefinidos:
  - de resolução de âmbito: `::`
  - de acesso a membro de objecto: `. .*`

Overloading de operadores unários

- Um operador unário prefixo (`- ! ++ --`) pode ser definido por:
  1) um membro-função não estático sem argumentos, ou
  2) uma função não membro com um argumento
- Para qualquer operador unário préfixo `@`, `@x` pode ser interpretado como:
  1) `x.operator@()`, ou
  2) `operator@(x)`
- Os operadores unários posfixos (`++ --`) são definidos com um argumento adicional do tipo `int` que nunca é usado (serve apenas para distinguir do caso prefixo):

  Para qualquer operador unário posfixo `@`, `x@` pode ser interpretado como:
  1) `x.operator@(int)`, ou
  2) `operator@(x,int)`

Overloading de operadores binários

- Um operador binário pode ser definido por:
  1) um membro-função não estático com um argumento, ou
  2) uma função não membro com dois argumentos
- Para qualquer operador binário `@`, `x@y` pode ser interpretado como:
  1) `x.operator@(y)`, ou
  2) `operator@(x,y)`
- Os operadores `=` `[]` `()` e `->` só podem ser definidos da 1ª forma (por membros-função não estáticos), para garantir que do lado esquerdo está um *lvalue*

Entrada e saída de dados com << e >>

```
class Data {
  public:
    friend ostream & operator<<(ostream & o, const Data & d);
    friend istream & operator>>(istream & i, Data & d);
    // ...
  private:
    int dia, mes, ano;
};

ostream & operator<<(ostream & o, const Data & d)
{
  o << d.dia << '/' << d.mes << '/' << d.ano;
  return o;
}
istream & operator>>(istream & i, Data & d)
{
  char b1, b2;
  i >> d.dia >> b1 >> d.mes >> b2 >> d.ano;
  return i;
}
```

Exemplo com operador unário

```
class Data {
  public:
     Data & operator++();     // prefixo
     Data & operator++(int); // posfixo
     // ...
   private:
     int dia, mes, ano;
};

Data & Data::operator++()
{
   if (dia ==  numDiasMes(ano,mes)) {
       dia = 1;
       mes = mes == 12? 1 : mes+1;
   }
   else
       dia++;
   return *this;
}

inline Data & Data::operator++(int)
{ return operator++(); }
```

```
main()
{
   Data d1 (30,12,2000);
   cout << d1 << '\n';
   d1++;
   cout << d1 << '\n';
   ++d1;
   cout << d1 << '\n';
   return 0;
}
```

Nota:

```
d2 = ++d1;
```

atribui a `d2` o valor de `d1` já incrementado!

Overloading do operador de atribuição

```
class Pessoa {   // (ver construtor de cópia)
    char *nome;  // alocado dinamicamente no construtor
  public:
    void setNome(const char *nm) { /* liberta, aloca e copia */ }

    Pessoa & operator=(const Pessoa & p)
      { setNome(p.nome); return *this; }

    Pessoa & operator=(const char *nome)
      { setNome(nome); return *this; }

  // ...
};

void teste()
{  Pessoa p1 ("Joao");
   Pessoa p2 ("Maria");
   p2 = p1;             // Agora é seguro!
   p2 = "Jose";         // Agora é possível!
}
```

Overloading de operadores de conversão

```
class Pessoa {   // (ver construtor de cópia)
    char *nome;  // alocado dinamicamente no construtor
  public:
    void setNome(const char *nm) { /* liberta, aloca e copia */ }

    operator const char *() const { return nome; }
  // ...
};

void teste()
{
   Pessoa p1 ("Joao");
   Pessoa p2 ("Maria");
   const char *s = p2; // Agora é possível
}
```


Overloading do operador de função

```
class Polinomio {
    double *coefs;
    int grau;
  public:
    double operator() (double x) const;
    // ...
};

// Calcula valor de polinómio num ponto x
double Polinomio::operator() (double x) const
{ double res = coefs[grau];
  for (int i = grau-1; i >= 0; i--) res = res * x + coefs[i];
  return res;
}

void teste()
{
  Polinomio pol; cout << "pol? "; cin >> pol;
  double x; cout << "x? "; cin >> x;
  cout << "pol(x)=" << pol(x) << '\n';
}
```

Membro-função com o nome **`operator()`** permite usar um objecto como se fosse uma função (neste caso função constante com um argumento e retorno do tipo `double`)!

Classe para números complexos

```
class Complex {
    double re, im;
 public:
    // construtores
    Complex(double r=0) : re(r), im(i) {}
    Complex(double r, double i) : re(r), im(i) {}
    Complex(const Complex &c) : re(c.re), im(c.im) {}

    ~Complex() {}    // destrutor

    // operadores aritméticos binários
    Complex operator+ (const Complex & c) const;
    Complex operator- (const Complex & c) const;
    Complex operator* (const Complex & c) const;
    Complex operator/ (const Complex & c) const;

    // operadores relacionais
    bool operator== (const Complex & c) const;
    bool operator!= (const Complex & c) const;
```


Classe para números complexos

```
    // operadores de atribuição
    const Complex & operator= (const Complex & c);
    const Complex & operator+= (const Complex & c);
    const Complex & operator-= (const Complex & c);
    const Complex & operator*= (const Complex & c);
    const Complex & operator/= (const Complex & c);


    // operadores unários
    Complex operator-() const;

    // membro-função
    double size() const
        { return sqrt(re*re + im*im); }

    // I/O
    friend  ostream & operator<< (ostream & o, const Complex & c);
    friend  istream & operator>> (istream & i, const Complex & c);

};
```

Nºs complexos: operadores aritméticos e relacionais

```
Complex Complex::operator+ (const Complex & c) const
{
  return Complex(re+c.re, im+c.im);
}

Complex Complex::operator* (const Complex & c) const
{
  return Complex(re*c.re-im*c.im, im*c.re+re*c.im);
}


bool Complex::operator== (const Complex & c) const
{
  return ( re==c.re && im==c.im );
}

bool Complex::operator!= (const Complex & c) const
{
  return ( re!=c.re || im!=c.im );
}
```


Nºs complexos: operadores atribuição e operador unário

```
const Complex & Complex::operator= (const Complex & c)
{
  if (this != &c)
  {
    re=c.re;
    im=c.im;
  }
  return *this;
}

const Complex & Complex::operator+= (const Complex & c)
{
  re += c.re;
  im += c.im;
  return *this;
}

Complex Complex::operator-() const
{
  return Complex(-re, -im);
}
```

Nºs complexos: entrada/saída

```
istream & operator>> (istream & i, const Complex & c)
{
  return i >> c.re >> c.im;
}

ostream & operator<< (ostream & o, const Complex & c)
{
  o << c.re;
  if (c.im > 0)
    o << "+" << c.im << "j";
  else
    o << c.im << "j";
  return o;
}
```


Nºs complexos: teste

```
int main()
{
  Complex c1;
  Complex soma=0;
  cout << "Escreva um n° complexo";
  while (cin >> c1)
  {
    soma += c1;
    cout << "Escreva outro n° complexo";
  }
  cout << "Soma é "<< soma;

  return 0;
}
```