Curso de Ciência da Computação

Algoritmos e Programação de Computadores 2per

Programação Orientada a Objetos POO

Profa. Fernanda dos Santos Cunha

Objetos *const*

- O qualificador **const** na declaração de um objeto indica que o mesmo é uma constante, e nenhum de seus membros pode ser alterado:

  const Data natal(25,12,2016).

  – Problema: isto faz com que o compilador proíba o acesso a qualquer função membro porque não sabe quais funções alteram os seu dados.

Objetos *const*

- Colocar o **const** após os parênteses dos parâmetros da função faz com que tal função não possa modificar o objeto que a invoca:

```
class Data {
   int d, m, a;
  public:
   Data(int dd, int mm, int aa);
   void initData(int dd, int mm, int aa); // obj const não a acessam
   int bissexto() const {
        return (a%4==0 && a%100 || a%400==0); }
   void printData() const;
   ...
};
void printData() const { // necessário colocar const na definição tb
   cout << endl << d << "/" << m <<"/" << a << endl;
}
```

Objetos como argumentos de funções-membro

```
class Venda {
    int npecas; float preco;
  public:
    Venda(int n, float p);
    void getVenda();
    void printVenda() const;
    void addVenda(Venda v1, Venda v2);
};
void addVenda(Venda v1, Venda v2) {
    npecas = v1.npecas + v2.npecas; preco = v1.preco + v2.preco; }
void printVenda() const {
   cout << setiosflags(ios::fixed) // não notação cientifica
        << setiosflags(ios::showpoint) // ponto decimal
        << setprecision(2) // duas casas
        << setw(10) << npecas; // tamanho 10 espaços na tela
   cout << setw(10) << preco << endl;
   }              // fazer include do <iomanip> para usar estas funcoes
```

Objetos como argumentos de funções-membro

```
class Venda {
    int npecas; float preco;
  public:
    Venda(int n, float p);
    void getVenda();
    void printVenda() const;
    void addVenda(Venda v1, Venda v2);
};
void addVenda(Venda v1, Venda v2) {
    npecas = v1.npecas + v2.npecas; preco = v1.preco + v2.preco;}
int main() {
  Venda a(5,120.50), b, total;
  b.getVenda();
  total.addVenda(a,b);     // objeto total invoca função
  cout<< "Venda A" <<a.printVenda() << endl;
  cout<< "Venda B" <<b.printVenda() << endl;
  cout<< "Total" <<total.printVenda() << endl;     return 1; }
```

Objetos como retorno de funções

```
class Venda {
    int npecas; float preco;
  public:
    Venda(int n, float p);
    void getVenda();
    void printVenda() const;
    Venda addVenda(Venda v1) const; // não modifica o objeto !!!!
};
Venda addVenda(Venda v1) const {
    Venda aux;          aux.npecas =npecas + v1.npecas;
    aux.preco = preco + v1.preco;     return aux; }
int main(){
  Venda a(5,120.50), b, total;
  b.getVenda();
  total = a.addVenda(b);  // objeto a invoca função
  ...
  return 1; }
```

Sobrecarga de Operadores

- Transformar expressões obscuras e complexas por outras mais óbvias e intuitivas
  - Nos programas anteriores :

    total.addVenda(a,b);

    total = a.addVenda(b);
  - Não ficaria mais claro e direto: **t = a + b;**
- Sobrecarregar um operador: redefinir seu símbolo, de modo que ele se aplique também aos tipos definidos pelo programador (classes e estruturas).

Sobrecarga de Operadores

- Sobrecarga é definida por meio de funções chamadas **operadoras**, que podem ser criadas como membros de classes ou independentes, acessíveis a todo o programa.
- Deve-se respeitar a definição original do operador: não se muda um operador binário (2 operandos) para criar um operador unário (1 operando).
- Não se pode estender a linguagem inventando novos operadores, com novos símbolos.

Sobrecarga de Operadores

- Deve-se continuar obedecendo a precedência original do operador.
- Não podem ser sobrecarregados os operadores:
  - Ponto de acesso a membros (.)
  - Resolução de escopo (::)
  - Condicional Ternário (?:)
- Regra básica para sobrecarga:
  - Operador **unário** definido como função-membro de classe não recebe nenhum argumento, enquanto que operador **binário** definido como função-membro de classe recebe um único argumento.

Sobrecarga de Operador ++ Pré

```
class Ponto{
   int x,y;
  public:
   Ponto(int x=0, int y=0) { this->x = x; this->y = y; };
   void operator ++ () { ++x; ++y; }; // operador incremento pré
   void printPonto () const;
};
void printPonto () const {
  cout<< "(" << x << "," << y<< << ")"<< endl;
  }
int main(){
  Ponto p1, p2(2,3);
  cout<< "p1 = " << p1.printPonto() << endl;
  cout<< "p2 = " << p2.printPonto() << endl;
  ++p1; ++ p2; // incrementa pontos
  cout<< "++p1 = " << p1.printPonto() << endl;
  cout<< "++p2 = " << p2.printPonto() << endl;
  return 1; }
```

Saída:
p1=(0,0)
p2=(2,3)
++p1=(1,1)
++p2=(3,4)

Sobrecarga de Operador ++ Pré

- Valor de retorno da função operadora:
- A função operator++() da classe ponto tem um defeito sutil se fizer a instrução:

  p2 = ++p1;
- Compilador não irá compilar esta instrução, pois a função operadora não tem retorno definido !!

```
class Ponto{
   ...
   Ponto operator ++ () {
     ++x; ++y;
     Ponto aux; aux.x = x; aux.y = y; return aux;
   };
};
```

Sobrecarga de Operador ++ Pré

```
int main(){
  Ponto p1, p2(2,3), p3;
  cout<< "p1 = " << p1.printPonto() << endl;
  cout<< "p2 = " << p2.printPonto() << endl;
  cout<< "++p1 = ";
  (++p1).printPonto(); // incrementa e imprime, antes não dava
  cout<< "++p2 = ";
  (++p2).printPonto(); // incrementa e imprime
  p3 = ++p1; // incrementa e atribui
  cout<< "p3 = ";
  p3.printPonto(); // incrementa e imprime
  return 1;
  }
```

Saída:
p1=(0,0)
p2=(2,3)
++p1=(1,1)
++p2=(3,4)
p3=(2,2)

Objetos temporários sem nome

- Ao invés de criar o objeto temporário aux

```
class Ponto{
   ...
   Ponto operator ++ () {
     ++x; ++y; Ponto aux; aux.x = x; aux.y = y; return aux; };
};
```

- O meio mais conveniente é

```
class Ponto{
   ...
   Ponto operator ++ () {
     ++x; ++y; return Ponto(x,y); // objeto sem nome
   };
};
```

Sobrecarga de Operador ++ Pós

- Como o operador ++ tem 2 maneiras de uso (pré e pós fixada), a sobrecarga vai ser diferenciada pelo compilador pelo número de argumentos da função:
  - Sem nenhum parâmetro => operação pré fixada
- Assim, para definir a operação pós fixada usa-se a função **operator ++(int)**
  - O parâmetro servirá apenas para o compilador diferenciar as duas formas de uso, a função também irá ignorar este argumento.

Sobrecarga de Operador ++ Pós

```
class Ponto{
   int x,y;
 public:
   Ponto(int x=0, int y=0) { this->x = x; this->y = y; };
   Ponto operator ++ () { // operador incremento pré
      ++x; ++y; return Ponto(x,y); };
   Ponto operator ++ (int) { // operador incremento pós
      ++x; ++y; return Ponto(x-1,y-1); };
      // incrementou o objeto mas retorna o ponto anterior
   void printPonto () const;
};
int main(){
 Ponto p1, p2(2,3);
 cout<<"p1= "<< p1.printPonto() <<endl;
 cout<<"p2= "<< p2.printPonto() <<endl;
 cout<<"++p1= "<< (++p1).printPonto() <<endl;
 cout<<"p2++= "<< (p2++).printPonto() <<endl;
 cout<<"p2= "<< p2.printPonto() <<endl; return 1; }
```

Saída:
p1=(0,0)
p2=(2,3)
++p1=(1,1)
p2++=(2,3)
p2=(3,4)

Sobrecarga de Operadores Binários

- Relembrando a regra básica para sobrecarga:
  - Operador **unário** definido como função-membro de classe não recebe nenhum argumento, enquanto que operador **binário** definido como função-membro de classe recebe um único argumento.
- Exemplo: sobrecarga de operador aritmético +

Sobrecarga de Operadores Binários


```
class Venda {
    int npecas; float preco;
  public:
    Venda(int n, float p);
    void getVenda();
    void printVenda() const;
    Venda operator + (Venda v) const; // funcao operadora !!!!
};
Venda operator + (Venda v) const {
    int pec = npecas + v.npecas;       float pre = preco + v.preco;
    return Venda(pec, pre); }
int main(){
  Venda a(5,120.50), b, c(30,6000.3), t, total;         b.getVenda();
  t = a + b; // usando sobrecarga
  total = a + b + c; // somas múltiplas
  ...
  return 1; }
```