INF 112: Programação II

Aula 15

➔ **Programação Orientada a Objetos em C++ (parte 4)**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

POO em C++ (parte 4)

- Vejamos outra questão importante sobre sobrecarga de operadores.

- Quando uma função qualquer retorna um determinado valor, ela está na verdade retornando uma cópia (que é mantida na pilha) do valor indicado no *return*.

- É fácil ver o porque disto. Imagine que uma função possua uma certa variável local e, ao final, retorne seu valor para a função chamadora. Ora, a variável local "morre" ao final da função sendo executada, ou seja, a função precisa copiar o valor daquela variável, antes que a mesma saia de escopo, para então poder retorná-lo.

- Para retornar objetos de certas classes, a cópia, no momento da saída da função, pode ser custosa ou mesmo causar problemas se os atributos forem ponteiros, como em alguns casos que já citamos (para justificar o uso do construtor de cópia, por exemplo).

- Portanto, quando retornarmos um objeto *TresD* no nosso exemplo passado, na sobrecarga do =, há a criação de um objeto *TresD* temporário que será uma cópia do objeto colocado à frente do *return*.

- Isto então é uma instanciação de um novo objeto e implica em chamada do construtor e do destrutor (o destrutor é chamado logo em seguida, após a volta da execução para função chamadora).

- Assim, se não houver construtor de cópia, isto pode causar problemas, o que não é o caso da classe *TresD*, pois os atributos são *int* e a cópia *bit* a *bit* não causaria problemas.

- Mas já no caso da classe *String*, onde se aloca o arranjo de *char* dinamicamente, poderia sim haver problemas. Também poderia causar efeitos indesejados na classe *Lista* (que vimos nas aulas práticas) que utiliza um arranjo de *int* dinâmico.

- Independente se há problemas ou não na cópia criada ao sair da função, a criação de um objeto, toda vez que o método realiza o *return*, pode causar um impacto considerável no desempenho do programa.

- Assim, em alguns casos que vimos de sobrecarga de operadores, sugere-se que ao retornar o objeto, retorne-se não o valor mas sim a referência para o objeto. Mas atenção!!! Em algumas sobrecargas de operador que mostramos o objeto não era local. Retornar uma referência para um objeto local está ERRADO, já que o objeto sairá de escopo.

- A classe *TresD* então ficaria como se segue. Note que a mudança é mínima. Somente o acréscimo de um `&` à frente do tipo retornado pelo método. Não se muda mais nada.

POO em C++ (parte 4)

```
class TresD {
  private:
    int x, y, z; // coordenadas 3D
  public:
    TresD operator+(const TresD &t);
    TresD& operator=(const TresD &t);
    TresD operator++(int); // Incremento pós-fixado
    TresD& operator++();  // Incremento pré-fixado
    void mostra();
    void atribui(int mx, int my, int mz);
};
...
```

POO em C++ (parte 4)

```
...
// Sobrecarrega o +
// Note que aqui NAO mudou, pois o objeto retornado é
// criado localmente e sairá de escopo ao final do método.
TresD TresD::operator+(const TresD &t) {
  TresD temp;
  temp.x = x + t.x;
  temp.y = y + t.y;
  temp.z = z + t.z;
  return temp;
}

// Sobrecarrega o =
TresD& TresD::operator=(const TresD &t) {
  x = t.x;
  y = t.y;
  z = t.z;
  return *this;
}
...
```

POO em C++ (parte 4)

```
...
// Sobrecarrega o operador unario ++ pós-fixado
TresD TresD::operator++(int) {
  TresD temp;
  x++;
  y++;
  z++;
  temp.atribui(x-1,y-1,z-1);
  return temp; // *** Aqui também não se pode retornar
}              // a referência para temp!!!

// sobrecarrega o operador unario ++ pré-fixado.
TresD& TresD::operator++() {
  x++;
  y++;
  z++;
  return *this;
}
...
```

```
...
// mostra as coordenadas X, Y, Z
void TresD::mostra() {
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

// atribui coordenadas
void TresD::atribui(int mx, int my, int mz) {
  x = mx;
  y = my;
  z = mz;
}
...
```

POO em C++ (parte 4)

- Vamos mostrar agora outro exemplo de sobrecarga para apontar uma situação que pode causar dúvidas.

- Imagine que acrescentemos mais um método para sobrecarregar o +, mas desta vez recebendo um *int* de modo a somá-lo às três coordenadas de um objeto *TresD*.

- Assim, o seguinte programa seria válido (veja o método no próximo *slide*):

```
main() {
  TresD a, b;
  a.atribui(4, 5, 6);
  b=a+5; // O + sobrecarregado.
  b.mostra(); // b ficaria com 9, 10 e 11, respec.
}
```

POO em C++ (parte 4)

```
class TresD {
  private:
    int x, y, z; // coordenadas 3D
  public:
    ... // igual a antes
    TresD operator+(int s);
    ... // igual a antes
};
... // igual a antes
TresD TresD::operator+(int s) {
  TresD temp;
  temp.x = x + s;
  temp.y = y + s;
  temp.z = z + s;
  return temp;
}
... // igual a antes
```

- Tudo bem até aqui. Porém, se ao invés de `b=a+5`, quiséssemos fazer `b=5+a`, não seria possível. Veja que fazer `a+5`, é o mesmo que fazer `a.operator+(5)`. Então a segunda forma não poderia ser utilizada, uma vez que o primeiro operando é do tipo *int*, para o qual não há a definição do + para tais situações.

- A solução para isto é o uso de funções *friend*. Quando a classe declara que é *friend* de uma função, significa que esta poderá acessar os membros privados de objetos desta classe.

- Note que não há o ponteiro *this* na execução de funções *friend*, pois as mesmas não pertencem à classe, só têm acesso aos membros dos objetos desta. Vejamos como ficará a classe *TresD* para que este conceito se torne mais claro.

POO em C++ (parte 4)

```
class TresD {
  private:
    int x, y, z; // coordenadas 3D
  public:
    TresD operator+(const TresD &t);
    TresD operator+(int s);
    friend TresD operator+(int s, const TresD &t);
    TresD operator++(int); // Incremento pós-fixado
    TresD& operator++();  // Incremento pré-fixado
    TresD& operator=(const TresD &t);
    void mostra();
    void atribui(int mx, int my, int mz);
};
...
```

POO em C++ (parte 4)

```
...
// Note que este *NAO* é um método da classe TresD
TresD operator+(int s, const TresD &t) {
  TresD temp;
  temp.x = t.x + s; // Consegue acessar x, y e z de t
  temp.y = t.y + s; // e temp porque a classe
  temp.z = t.z + s; // TresD é amiga desta função
  return temp;
}

TresD TresD::operator+(int s) {
  TresD temp;
  temp.x = x + s;
  temp.y = y + s;
  temp.z = z + s;
  return temp;
}
...
```

```
...
TresD TresD::operator+(const TresD &t) {
  TresD temp;
  temp.x = x + t.x;
  temp.y = y + t.y;
  temp.z = z + t.z;
  return temp;
}

TresD& TresD::operator=(const TresD &t)
{// Como antes}
TresD TresD::operator++(int) {// Como antes}
TresD& TresD::operator++() {// Como antes}
void TresD::mostra() {// Como antes}
void TresD::atribui(int mx, int my, int mz)
{// Como antes}
```

Exercícios

- Redefina o operador = na classe *Lista* para que comandos de atribuição da forma `l1=l2` (onde `l1` e `l2` são objetos `Lista`) sejam possíveis.

- Redefina o operador + na classe *Lista* de forma que o mesmo signifique inserção no fim da lista. Por exemplo, se `li` é um objeto `Lista`, então `li+5` seria válido e significaria inserção do 5 no final da lista. Depois, estenda sua classe acrescentando outra redefinição do operador + de forma que o mesmo signifique inserção no início da lista, ou seja, que comandos como `5+li` (inserção do 5 no início) sejam válidos.

- Estenda mais ainda sua classe *Lista*, sobrecarregando os operadores ++ pré-fixado e ++ pós-fixado, de modo que signifiquem o incremento de uma unidade a todos os elementos da lista. Teste sua implementação fazendo ora `l1=l2++`, ora `l1=++l2` e veja como ficaram `l1` e `l2` (onde `l1` e `l2` são objetos `Lista`).

Exercícios

➔ Para a classe `String` vista, estenda a mesma de modo que comandos da forma abaixo sejam aceitos, todos significando concatenação, onde `s1` e `s2` são objetos `String`:

- `s1 + s2`
- `s1 + "ola"`
- `"ola" + s1`