INF 112: Programação II

Aula 14

➔ **Programação Orientada a Objetos em C++ (parte 3)**

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

POO em C++ (parte 3)

- Vejamos mais alguns aspectos sobre construtores e destrutores.

- Com relação à classe *String* mostrada anteriormente, este é um exemplo onde seria interessante ter um construtor de cópia, uma vez que há um atributo ponteiro e a cópia *default* de atributos poderia causar problemas. Vejamos então como ficaria.

- Aproveitemos o exemplo para apresentar o uso do *const* no parâmetro de uma função.

- Também usaremos o exemplo para introdução ao ponteiro *this*.

POO em C++ (parte 3)

```
class String {
  private:
    Char * str;
  public:
    String(char * s);
    String();
    String(const String &os); // Construtor de copia
    ~String();
    char* leStr();
    void escStr(char * s);
};
...
```

POO em C++ (parte 3)

```
...
String::String() {
  this->str = new char[1];
  this->str[0] = '\0';
}

String::String(char * s) {
  int tam = strlen(s);
  this->str = new char[tam+1];
  strcpy(this->str,s);
}

String::String(const String &os) { // C. de copia
  int tam = strlen(os.str);
  this->str = new char[tam+1];
  strcpy(this->str,os.str);
}
...
```

POO em C++ (parte 3)

```
...
String::~String() {
  delete [] this->str;
}

char* String::leStr() {
  return this->str;
}

void String::escStr(char * s) {
  int tam = strlen(s);
  delete [] this->str;
  this->str = new char[tam+1];
  strcpy(this->str,s);
}
...
```

POO em C++ (parte 3)

```
...
main() {
  String s1("Professor");
  String s2("Fabio");
  String s3(s1); // s3 serah uma copia de s1 (copia
                 // arranjo de s1 em s3). Ou: s3=s1
  String s4("Andre");

  // O que vai ser impresso na tela???
  cout << s1.leStr() << " " << s2.leStr() << endl;
  cout << s3.leStr() << " " << s4.leStr() << endl;
}
```

- Note a palavra chave *const* na declaração do parâmetro do construtor de cópia. Isto garante que o objeto passado por referência não seja em hipótese alguma alterado.

- Outra observação é com relação ao ponteiro *this*. Cada vez que se invoca uma função membro, automaticamente é passado um ponteiro para o objeto que a invoca. Podemos utilizar este ponteiro usando a palavra chave *this*. Este ponteiro é um parâmetro implícito para todas as funções membro.

- No caso mostrado, se não usássemos o *this*, o efeito seria o mesmo. Na verdade, a sua omissão implica no seu uso implícito. Mas haverá casos em que o uso deste ponteiro deverá ser de forma explícita, como será visto mais à frente.

POO em C++ (parte 3)

- Um outro ponto sobre construtores e destrutores é a ordem em que estes são chamados quando o objeto em questão possui outros objetos como membro.

- Vimos um caso assim quando mostramos a classe *Reta* que possui dois objetos *Ponto* como membro.

- Vamos rever este exemplo, agora com construtores e destrutores e códigos dentro destes para imprimir uma mensagem na tela. Com as mensagens, ficará clara a ordem em que estes métodos especiais são executados em casos assim.

POO em C++ (parte 3)

```
class Ponto {
  private:
    float x;
    float y;

  public:
    Ponto();
    Ponto(float a,float b);
    Ponto(const Ponto &p);
    ~Ponto();
    void mostra();
    void move(float dx,float dy);
};
...
```

POO em C++ (parte 3)

```
...
class Reta {
  private:
    Ponto p1;  // objetos de
    Ponto p2;  // outra classe

  public:
    Reta(float x1,float y1,float x2,float y2);
    ~Reta();
    void mostra();
};
...
```

POO em C++ (parte 3)

```
...
Ponto::Ponto() {
  x=y=0;
}
Ponto::Ponto(float a, float b) {
  cout << "Construtor da classe Ponto." << endl;
  x=a;
  y=b;
}
Ponto::Ponto(const Ponto &p) { // Construtor de copia
  x = p.x;
  y = p.y;
}
...
```

POO em C++ (parte 3)

```
...
void Ponto::mostra() {
  cout << "X: " << x << ", Y: " << y << endl;
}

void Ponto::move(float dx,float dy) {
  x+=dx;
  y+=dy;
}

Ponto::~Ponto() {
  cout << "Destrutor da classe Ponto." << endl;
}
...
```

POO em C++ (parte 3)

```
...
Reta::Reta(float x1,float y1,float x2,float
y2):p1(x1,y1),p2(x2,y2) {
    cout << "Construtor da classe Reta." << endl;
}

Reta::~Reta() {
  cout << "Destrutor da classe Reta." << endl;
}

void Reta::mostra() {
  p1.mostra();
  p2.mostra();
}
...
```

```
...
main() {
  Reta r1(1.0,1.0,10.0,10.0);
  r1.mostra();
}
```

- Ao executar o programa acima, a seguinte saída será exibida:

```
Construtor da classe Ponto.
Construtor da classe Ponto.
Construtor da classe Reta.
x: 1, y: 1
x: 10, y: 10
Destrutor da classe Reta.
Destrutor da classe Ponto.
Destrutor da classe Ponto.
```

POO em C++ (parte 3) – Sobrecarga de métodos

➔ ***Sobrecarga (overload) de métodos***, **em uma classe, é a capacidade de ter métodos exatamente com o mesmo nome mas que desempenham papéis não necessariamente iguais. Isto é útil quando precisamos de métodos que realizam tarefas afins e, neste caso, não queremos dar nomes diferentes para cada método.**

➔ **Como o compilador diferencia esses métodos se eles têm o mesmo nome? A resposta é: pelos parâmetros, ou seja, o tipo e o número de parâmetros. Diferenciar pelo tipo do retorno não é válido, o que se analisa são sempre os parâmetros. A seguir, podemos ver o método *escAtrib* sobrecarregado na classe *Aluno*.**

POO em C++ (parte 3) – Sobrecarga de métodos

```
class Aluno {
  private:
    char nome[50];
    int matricula;
    float mediaGrad;
  public:
    Aluno();
    Aluno(char * n, int mat, float med);
    char* leNome();
    int leMatricula();
    float leMedia();
    void escAtrib(char * n);
    void escAtrib(int mat);
    void escAtrib(float med);
};
...
```

POO em C++ (parte 3) – Sobrecarga de métodos

```
...
Aluno::Aluno(){}

Aluno::Aluno(char * n, int mat, float med) {
  strcpy(nome, n);
  matricula = mat;
  mediaGrad = med;
}

char* Aluno::leNome() {return nome;}

int Aluno::leMatricula() {return matricula;}

float Aluno::leMedia() {return mediaGrad;}
...
```

POO em C++ (parte 3) – Sobrecarga de métodos

```
...
// Abaixo o exemplo de sobrecarga:

void Aluno::escAtrib(char * n) {
  strcpy(nome, n);
}

void Aluno::escAtrib(int mat) {
  matricula = mat;
}

void Aluno::escAtrib(float med) {
  mediaGrad = med;
}
...
```

POO em C++ (parte 3) – Sobrecarga de métodos

```
...
main() {
  Aluno a1;
  char nome[50];
  int matricula;
  float media;
  gets(nome);
  cin >> matricula;
  cin >> media;
  a1.escAtrib(nome);
  a1.escAtrib(matricula);
  a1.escAtrib(media);
  cout << "Nome do aluno: " << a1.leNome() << endl;
  cout << "Matricula: " << a1.leMatricula() << endl;
  cout << "Media: " << a1.leMedia() << endl;
}
```

POO em C++ (parte 3) – Sobrecarga de operadores

- **Outra característica do C++ relacionada a sobrecarga de funções é a *sobrecarga de operadores*.**

- **Muitos operadores do C++ podem receber significados especiais relativos a classes específicas. Por exemplo, uma classe para implementar o TAD lista pode usar o operador + para adicionar um elemento à lista. Outra classe pode usar o operador + de uma maneira inteiramente diferente.**

- **Quando um operador é sobrecarregado, nada do seu significado original é perdido. Para sobrecarregar um operador devemos definir o que a dada operação significa em relação à classe em que ela é aplicada. Para isso criamos um método *operator* que define a sua ação.**

POO em C++ (parte 3) – Sobrecarga de operadores

- A forma geral do método *operator* é a que se segue. O símbolo # representa o operador sendo sobrecarregado.

```
tipo nome_da_class::operator#(lista de argumentos)
{
// Definicao da operacao no contexto da classe.
}
```

- Aqui o tipo é o tipo do valor retornado pela operação específica. Um operador sobrecarregado tem frequentemente um retorno do mesmo tipo da *classe* onde o operador está sendo definido.

- No exemplo que se segue, implementamos uma classe chamada *TresD* para manter as coordenadas de um objeto no espaço tridimensional. Os operadores + e = são sobrecarregados.

- Utilizaremos passagem por referência e *const* nos sobrecarregamentos pelos mesmos motivos já expostos (quando falamos de construtor de cópia).

POO em C++ (parte 3) – Sobrecarga de operadores

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

POO em C++ (parte 3) – Sobrecarga de operadores

```
...
// sobrecarrega o +
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 3) – Sobrecarga de operadores

```
...
// 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 3) – Sobrecarga de operadores

```
...
main() {
  TresD a, b, c;
  a.atribui(1, 2, 3);
  b.atribui(10, 10, 10);
  a.mostra();
  b.mostra();
  c = a + b; // Uso do operator + e, em seguida, do =
  c.mostra();
  c = a + b + c; // Uso multiplo do + e depois do =
  c.mostra();
  c = b = a; // Atribuicao multipla
  c.mostra();
  b.mostra();
}
```

- **O resultado do programa será:**

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

- Quando examinamos o exemplo anterior, vemos que as funções *operator* tem somente um parâmetro, embora sobrecarreguem operações binárias. A razão é que só precisamos do segundo argumento da operação binária pois o primeiro está implícito usando o ponteiro *this*.

- Assim na linha:

```
temp.x = x + t.x;
```

o `x` refere-se a `this->x` que é o `x` associado ao objeto que solicitou a chamada do método.

➔ No exemplo, quando o operador + é sobrecarregado, ele retorna um objeto do tipo *TresD*. Assim dois objetos são somados, retornando um terceiro objeto.

➔ Já o operador de atribuição faz com que o objeto à esquerda da atribuição seja modificado. A operação é realizada no objeto que gera a chamada do método pela passagem implícita do ponteiro *this*. Assim o conteúdo do objeto alterado é retornado com `*this`.

➔ Os únicos operadores que não podem ser sobrecarregados são:

```
.   ::   .*   ?
```

POO em C++ (parte 3) – Sobrecarga de operadores

- Veja agora um exemplo envolvendo sobrecarga do operador unário ++ para incremento pré- e pós-fixado.

- Note que em ambos os casos o nome do método seria:

```
operator++
```

- Para que haja a desambiguação entre os dois casos, convencionou-se que o método para sobrecarregar o ++ pós-fixado deveria receber um parâmetro *int*, enquanto que o método para o ++ pré-fixado deveria ter a lista de parâmetros vazia.

- Assim, pode-se diferenciar ambos pelos parâmetros.

POO em C++ (parte 3) – Sobrecarga de operadores

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

POO em C++ (parte 3) – Sobrecarga de operadores

```
...
TresD TresD::operator+(const TresD &t) {
// Como antes...
}

TresD TresD::operator=(const TresD &t) {
// Como antes...
}
...
```

POO em C++ (parte 3) – Sobrecarga de operadores

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

POO em C++ (parte 3) – Sobrecarga de operadores

```
...
// sobrecarrega do operador unario ++ pós-fixado.
TresD TresD::operator++(int) {
  TresD temp;
  x++;
  y++;
  z++;
  temp.atribui(x-1,y-1,z-1);
  return temp;
}
...
```

POO em C++ (parte 3) – Sobrecarga de operadores

```
...
void TresD::mostra() { // Como antes...}

void TresD::atribui(int mx, int my, int mz) {
  // Como antes...
}

main {
  TresD a, b;
  a.atribui(4, 5, 6);

  b=++a; // Experimente assim e depois assim: b=a++;
  b.mostra();
  a.mostra();
}
```

Exercício

➔ **Estenda a classe *String* mostrada anteriormente, acrescentando os operadores + e = para concatenar duas *strings* e atribuir uma *string* à outra, respectivamente.**