INF 112: Programação II

Aula 13

➔ **Programação Orientada a Objetos em C++ (parte 2)**

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

POO em C++ (parte 2) - Construtores

➔ ***Construtores*** **são métodos especiais chamados pelo sistema no momento da criação de um objeto.**

➔ **Construtores têm *sempre o mesmo nome da classe* e não possuem valor de retorno, porque não se pode chamar um construtor para um objeto.**

➔ **Construtores representam uma oportunidade de inicializar de forma organizada os objetos.**

- Vejamos um exemplo:

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

  public:
    Ponto(float a,float b); // Não há retorno
    void mostra();
    void move(float dx,float dy);
};
...
```

POO em C++ (parte 2) - Construtores

```
...
Ponto::Ponto(float a,float b) {
  x=a; // inicializando atributos da classe
  y=b;
}

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

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

POO em C++ (parte 2) - Construtores

```
...
main()
{
  // O construtor será chamado abaixo, pois
  // a declaracao é o momento da instanciacao (neste
  // caso estatica) do objeto.
  Ponto p(0.0,0.0); // Inicializa x e y com 0.0

  p.mostra();
  p.move(1.0,1.0);
  p.mostra();
}
```

POO em C++ (parte 2) – Agregação de Construtores

➔ **Imagine agora uma situação em que uma classe possua como atributos objetos de uma outra classe. Neste caso, ao se instanciar um objeto da primeira, os objetos contidos neste também serão instanciados e os construtores desses objetos precisarao, da mesma forma, ser chamados. Vejamos o exemplo a seguir:**

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

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

POO em C++ (parte 2) – Agregação de Construtores

```
...
Reta::Reta(float x1,float y1,float x2,float
y2):p1(x1,y1),p2(x2,y2) {
    // nada mais a fazer. Os construtores de p1
    // e p2 jah foram chamados. Sem esta extensao
    // do cabecalho do metodo daria erro de
    // compilacao pois os objetos p1 e p2 teriam
    // seus construtores chamados, mas não haveria
    // argumentos para os mesmos.
}

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

POO em C++ (parte 2) – Agregação de Construtores

```
...
main() {
  // Instanciacao de um objeto Reta.
  // Construtor de Reta eh chamado e em seguida
  // o de Ponto é chamado duas vezes.
  Reta r1(1.0,1.0,10.0,10.0);
  r1.mostra();
}
```

POO em C++ (parte 2) – Sobrecarga de Construtores

➔ **Uma classe pode conter mais de um construtor, desde que os construtores tenham parâmetros diferentes (sobrecarga de métodos, como veremos mais adiante). Vejamos o mesmo exemplo anterior, da classe *Ponto*, mas com dois construtores.**

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

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

POO em C++ (parte 2) – Sobrecarga de Construtores

```
...
Ponto::Ponto() {
  x=y=0;
}

Ponto::Ponto(float a,float b) {
  x=a;
  y=b;
}

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

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

POO em C++ (parte 2) – Sobrecarga de Construtores

```
...
main() {
   Ponto p; // Inicializa x e y de p com 0.0
   Ponto p2(3.0,5.0); // Inicializa x e y de
                      // p2 com 3.0 e 5.0

   p.mostra();
   p2.mostra();
   p.move(2.0,5.0);
   p2.move(3.0,1.0);
   p.mostra();
   p2.mostra();
}
```

➔ Lembra-se de que o construtor é chamado sempre na instanciação do objeto. Portanto, quando se usa alocação dinâmica, o construtor é chamado ao se utilizar o comando new.

```
...
main() {
  Ponto * pp; // Aqui só uma variável ponteiro simples
              // é criada de forma estática.

  // O construtor Ponto é chamado aqui
  pp = new Ponto(2.0,3.5); // pp então não é um
                           // objeto Ponto. É só um
                           // ponteiro que aponta
                           // para um objeto Ponto.
  pp->move(2.5,5.0); // Ou (*pp).move(2.5,5.0);
  pp->mostra();
  delete pp; // Objeto Ponto é desalocado da mem.
}
```

POO em C++ (parte 2) – Construtor de cópia

➔ **Quando um objeto é usado para inicializar outro, uma cópia *bit* a *bit* é feita. Em muitos casos, este mecanismo de cópia é perfeitamente adequado. Porém, há casos onde esta cópia simples pode ser perigosa; por exemplo, quando um objeto contém apontadores para áreas alocadas dinamicamente. Em tais casos, a cópia implicaria em um compartilhamento de dados que poderia não ter exatamente o efeito desejado.**

➔ **Para tais situações, C ++ permite a definição de um *construtor de cópia*. Quando tal construtor existe, a cópia é feita de modo controlado. A forma geral de declarar um construtor de cópia é:**

```
nome-classe (nome-classe &obj) { ... };
```

POO em C++ (parte 2) – Construtor de cópia

- Vejamos um exemplo:

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

  public:
    Ponto();
    Ponto(float a,float b);
    Ponto(Ponto &p); // construtor de cópia
    void mostra();
    void move(float dx,float dy);
};
...
```

POO em C++ (parte 2) – Construtor de cópia

```
...
Ponto::Ponto() {
  x=y=0;
}
Ponto::Ponto(float a,float b) {
  x=a;
  y=b;
}
Ponto::Ponto(Ponto &p) { // Construtor de copia
  x = p.x;
  y = p.y;
}
void Ponto::mostra() { ... // como antes }
void Ponto::move(float dx,float dy) { ... // como
antes }
...
```

POO em C++ (parte 2) – Construtor de cópia

```
...
main() {
 Ponto p1(2.0,5.0);
 // As duas formas abaixo são equivalentes para
 // inicializacao de um objeto a partir de outro:
 Ponto p2=p1;//x e y de p2 serao os mesmos x e y de p1
 Ponto p3(p2);//x e y de p3 serao os mesmos x e y de p2

 p1.mostra();
 p2.mostra();
 p3.mostra();
}
```

- **Ambas as formas:**

```
Ponto p2=p1;
```

e:

```
Ponto p3(p2);
```

acionam o construtor de cópia.

- **Mas veja que o construtor de cópia só é chamado automaticamente para inicializações. Um comando de atribuição no meio de uma função não ativa o construtor de cópia. Se este for o efeito desejado, o operador = deve ser sobrecarregado. Veremos sobrecarga de operadores mais adiante.**

- **Construtor de cópia também se faz necessário para passagem de um objeto por valor para uma função.**

POO em C++ (parte 2) - Destrutores

➔ **Análogos aos construtores, os *destrutores* também são funções membro chamadas pelo sistema, mas são chamadas quando o objeto sai de escopo ou, em alocação dinâmica, tem sua área desalocado.**

➔ **Não se pode chamar o destrutor. O que se faz é descrever o código a ser executado quando o objeto é destruído.**

➔ **Os destrutores são muito úteis para "limpar a casa" quando um objeto é destruído. Usados em conjunto com alocação dinâmica, destrutores fornecem uma maneira muito prática e segura de organizar o uso da *heap*.**

➔ **A sintaxe do destrutor é simples. Ele também tem o mesmo nome da classe só que precedido por ~ , não possui valor de retorno e nunca possui parâmetros:**

```
~nomedaclasse() { ... }
```

- **Para reforçar nossa visão sobre vantagens e desvantagens de construtores e destrutores, vejamos um exemplo para manipular *strings*. Primeiro, veremos uma abordagem não orientada a objetos, para em seguida utilizar uma classe para facilitar o uso de *strings*.**

POO em C++ (parte 2)

```
main(){
   char * s1;
   char * s2;
   char * s3;

   s1 = new char[6];
   s2 = new char[8];
   s3 = new char[10];
   strcpy(s1,"Fabio");
   strcpy(s2,"Ribeiro");
   strcpy(s3,"Cerqueira");
   cout << s1 << " " << s2 << " " << s3;
   delete [] s1;
   delete [] s2;
   delete [] s3;
}
```

➔ **Veja que para cada *string* houve a preocupação de alocar e desalocar. Se imaginarmos um programa grande, com várias *strings*, isto se torna um incômodo para o programador. Vejamos a versão orientada a objetos.**

```
class String {
  private:
    char * str;
 public:
   String(char * s);
   String(); // mais uma opcao para o usuario
   ~String();
   char* leStr();
   void escStr(char * s);
};
...
```

```
...
String::String() {
  // para o construtor com nenhum parametro
  // vamos considerar a string vazia.
  str = new char[1];
  str[0] = '\0';
}

String::String(char * s) {
  int tam = strlen(s);
  str = new char[tam+1]; //lembre-se, o +1 é para o \0
  strcpy(str,s);
}
...
```

POO em C++ (parte 2)

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

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

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

POO em C++ (parte 2)

➔ **Veja que o construtor nos poupa do trabalho de nos preocuparmos com alocação de memória. Da mesma forma, o destrutor desaloca a memória de modo que o programador não se ocupe com isto. Com o programa exemplo abaixo, que equivale ao programa mostrado anteriormente, fica claro o uso da classe *String* e como ela reduz o código.**

```
...
main(){
  String s1("Fabio");
  String s2("Ribeiro");
  String s3("Cerqueira");

  cout << s1.leStr() << " " << s2.leStr() << " " <<
  s3.leStr();
} // O destrutor da classe String é chamado aqui, no
  // fecha-chave (fim do escopo dos objetos), 3 vezes
  // (para s1, s2 e s3).
```

- No programa anterior, o destrutor para `s1`, `s2` e `s3` será chamado no fim da vida destes objetos, ou seja, no fecha-chave que finaliza o bloco onde estes objetos foram instanciados.

- Se os objetos fossem criados com alocação dinâmica, utilizando o `new`, o destrutor seria chamado no `delete` correspondente, que é quando o objeto criado "morre" .

- Outra observação importante é que o caso da classe *String* mostrada é um exemplo onde seria interessante ter um construtor de cópia uma vez que há um atributo ponteiro e a cópia *default* neste caso poderia causar problemas. Vejamos então como ficaria.

POO em C++ (parte 2)

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

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

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

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

POO em C++ (parte 2)

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

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

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

POO em C++ (parte 2)

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

  // 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.**

- **Seu uso é opcional, mas é interessante para garantir que o objeto passado por referência não seja em hipótese alguma alterado.**

Exercício

➔ **Mostre como seria aquela implementação do TAD lista que vimos, agora com a utilização de POO. Faça a alocação do arranjo que representa a lista de maneira dinâmica, colocando isto em um construtor. A desalocação deve estar no destrutor. Mostre também um construtor de cópia.**