Estruturas de dados II

Revisão de C++

André Tavares da Silva

andre.silva@udesc.br

UDESC

Introdução à linguagem C++

- Criada por Bjarne Stroustrup em 1983:

  Extensão da linguagem C para suportar classes;
  - Padronizada em 1998 com revisões em:
  - 2003, 2011, 2014, 2017 - ISO/IEC 14882:2017 (C++17).
- O C++ é um superconjunto da linguagem C;
- Todo programa em C válido também é um programa C++ válido;
- É uma linguagem híbrida pois permite que programas estruturados sejam também desenvolvidos.

UDESC

Recordando conceitos de POO

- Objetos:
  - Entidades lógicas que contém dados (atributos) e o código (métodos) para manipular estes dados;
  - Encapsulamento: Ligação entre atributos e métodos.
- Polimorfismo:
  - Um nome pode ser usado para muitos propósitos relacionados, mas ligeiramente diferentes;
  - Permite que um nome seja usado para uma classe geral de ações.
- Herança:
  - Processo em que um objeto pode adquirir as propriedades de outro objeto ☾ especialização;
  - Definição das característica únicas de um objeto. As outras podem ser herdadas de classes mais gerais.

UDESC

Entradas e Saídas em C++

- Objetos de E/S:
- cin: objeto associado ao dispositivo padrão de entrada (teclado);
- cout: objeto associado ao dispositivo padrão de saída (monitor).
- Operadores de redirecionamento:

  <<: de saída (da variável para cout);

  >>: de entrada (de cin para a variável).

UDESC

Entradas e Saídas em C++

```
#include <iostream> //biblioteca para operações de E/S em C++
#include <string> //biblioteca para a classe string

using namespace std;

int main (){
  int i;
  string str;
  cout << "C++ e facil" << endl;
  cout << "Informe um numero: "; //leitura de um número
  cin >> i;
  cout << "Seu numero: " << i << endl; //exibição de um número
  cout << "Informe uma string: "; //leitura de uma string
  cin >> str;
  cout << "Sua string: " << str << endl; //exibição de uma string
  return (0);
}
```

Elementos básicos em C++

- Tipos
  - **bool**, char, int, float, double, long, short
  - enum
  - void
  - Ponteiros: int*
  - Vetores: char[]
  - **Referências**: double**&**
- Declarações e definições
  - char ch;
  - int count = 1;
  - enum hemis {norte, sul};
  - float val = 1.6e-19;

UDESC

Referência

- Ponteiros são como em C ANSI
- Referências: são nomes alternativos para objetos:
  - Uma referência é declarada segundo a estrutura:
  - Tipo& VarRef = VariavelReferenciada;
  - Por exemplo:
    - int i = 100;
    - int& ri = i; // ou int &ri = i;
  - São diferentes de ponteiros pois uma referência precisa ser inicializada durante sua declaração:
    - int &r1, &r2; // Erro!!!

Referência

```
#include <iostream> //biblioteca para operações de E/S em C++

using namespace std;

int main (){
  int i;
  int &iAlias = i;
  cin >> i;
  cout << "Valor de iAlias: " << iAlias << endl;
  return (0);
}
```

// Qual a saída desse trecho de código?

UDESC

Alocação dinâmica em C++

- Operadores **new** e **delete**:
  - **new** é equivalente ao malloc em C;
  - **new** cria objetos na memória disponível (independe do escopo);
  - um objeto criado por **new** existe até que seja destruído por **delete**.

```
//Em C
int main()
{
//aloca ponteiro para double
double *ptr;
ptr = (double*)malloc(sizeof(double));
...
}
```

```
//Em C++
int main()
{
//aloca ponteiro para double
double *ptr;
ptr = new double;
...
}
```

UDESC

Sobrecarga de funções

- Um tipo de polimorfismo:
- Duas ou mais funções podem compartilhar o mesmo nome, contanto que as suas declarações de parâmetros sejam diferentes.

- Por exemplo:

```
// função quadrado é sobrecarregada 3x
int quadrado(const int &i);
double quadrado(const double &d);
long quadrado(const long &l);
```

UDESC

Templates

- Forma mais compacta de codificar funções sobrecarregadas que realizam a mesma função sobre conjuntos de dados de diferentes tipos;
- A função é declarada com um tipo de dado "em aberto", o qual é substituído em tempo de compilação pelo tipo de dado adequado à cada situação.

```
template <typename X> void swapargs(X &a, X &b){
  X temp;
  ...
```

UDESC

Templates

```
#include <iostream>
using namespace std;
template <typename X> void swapargs(X &a, X &b){
  X temp;
  temp = a; a = b; b = temp;
}
int main( ){
  int i = 10, j = 20;
  double x = 10.1, y = 23.3;
  char a = 'x', b = 'z';
  cout << "Original - i, j: " << i << ' ' << j << endl;
  cout << "Original - x, y: " << x << ' ' << y << endl;
  cout << "Original - a, b: " << a << ' ' << b << endl << endl;
  swapargs(i, j);
  swapargs(x, y);
  swapargs(a, b);
  cout << "Trocados - i, j: " << i << ' ' << j << endl;
  cout << "Trocados - x, y: " << x << ' ' << y << endl;
  cout << "Trocados - a, b: " << a << ' ' << b << endl;
  return(0);
}
```

UDESC

Classes

- Mecanismo de C++ que permite aos usuários a construção de tipos, que podem ser usados convenientemente como tipos básicos;
- Um tipo é a representação de um conceito. Por exemplo o tipo float, e suas operações +, -, * e /, corresponde à representação do conceito matemático de número real;
- Uma classe é um tipo definido pelo usuário, que não tem similar entre os tipos nativos.

UDESC

Classes

```
class Date{
  private:
    int d, m, y; //atributos
    static Date default_date;
  public:
    Date(int dd, int mm, int yy);
    //métodos
    void addYear(int n);
    void addMonth(int n);
    void addDay(int n);
    static void set_default(int, int, int);
}
```

UDESC

Membros de uma classe

- Atributos:
  - Podem ser qualquer tipo de dado, simples ou compostos, ou mesmo outras classes previamente definidas;

- Métodos:
  - São quaisquer funções definidas dentro de uma classe e que podem manipular os **atributos** de uma classe.

UDESC

Classes

- Atributos e métodos **private** de uma classe são acessíveis somente pelos outros membros da mesma classe ou por classes **friend**;

- Atributos e métodos **protected** são acessíveis também pelas classes derivadas de uma classe;

- Atributos e métodos **public** são acessíveis a partir de qualquer ponto onde a classe é visível.

UDESC

Construtores e Destrutores

- São funções que atuam implicitamente sobre os objetos no momento de sua declaração (construtores) e de sua remoção (destrutores);
- Construtores:
  - Servem para inicializar os atributos de uma classe durante sua declaração;
  - Podem ser sobrecarregados, ou seja, podem ser fornecidas diferentes versões com diferentes tipos de parâmetros.
  - Aceitam valores *default*:

```
Date(int dd, int mm, int yy=2025)
```

UDESC

Construtores e Destrutores

- Construtor default:
  - É um construtor, sem código e sem parâmetros, criado automaticamente pelo compilador quando não há um codificado pelo programador;
  - Pode também ser escrito pelo programador.
- Construtor de cópia:
  - Usado quando desejamos declarar um objeto e inicializá-lo com os valores guardados em outro objeto já existente da mesma classe;
  - Sintaxe:

    Nome_classe (Nome_classe &Obj, ...);

    &Obj: referência ao objeto que deverá ser copiado;

    Podem existir outros parâmetros, **mas devem ter valores *default***.

Classes

```
class Pessoa{
  private:
    int RG, CPF;
    char Nome[31];
    char Endereco[41];
    char Telefone[13];
  public:
    Pessoa(); // Construtor default
    Pessoa(Pessoa &P); // Construtor de cópia
    int getRG();
    int getCPF();
    ...
```

UDESC

Construtores e Destrutores

- Destrutores:
  - São funções complementares aos construtores;
  - São executados automaticamente quando o escopo de duração do objeto se encerra;
  - São responsáveis por "limpar a casa" quando um objeto for removido da memória;
  - Cada classe pode ter um único destrutor.
  - Sintaxe:

    ~Nome_classe();
  - Observações:
    - Não recebe parâmetros e nem possuem valor de retorno.

Arrays de objetos

- Objetos são tipos abstratos de dados;
  - Portanto, como todo tipo de dado, podem também ser usados na construção de arrays.

```
int main(){
  const int MAX = 4;
  Pessoa Turma[MAX];
  for (int i = 0; i < MAX; i++) { //Utilizando os objetos
    cout << endl << "Entre com os dados" << i << ": \n ";
    Turma[i].setRG();
    Turma[i].setCPF();
    Turma[i].setNome();
    Turma[i].setEndereco();
    Turma[i].setTelefone();
  }
}
```

UDESC

Funções Friend

- São funções externas a uma classe que possuem a capacidade de acessar seus membros privados ou protegidos;
- Podem ser de dois tipos:
  - Funções *friend* não-membros de classe;
  - Funções *friend* membros de outras classes.
- Funções *friend* não-membros de classe:
  - Geralmente são funções genéricas de manipulação de dados às quais desejamos enviar atributos de uma classe;
  - Sintaxe:
    - `Friend Nome_função([parâmetros]);`

Funções Friend

```
class linha;
class box{
  int cor; //cor do box
  int upx, upy; //canto superior esquerdo
  int lowx, lowy; //canto inferior direito
 public:
  friend int mesma_cor(linha l, box b);
  void indica_cor(int c);
  void define_box(int x1,int y1,int x2,int y2);
  void exibe_box(void);
};
class linha{
  int cor;
  int comecox, comecoy;
  int tamanho;
 public:
  friend int mesma_cor(linha l, box b);
  void indica_cor(int c);
  void define_linha(int x, int y, int l);
  void exibe_linha();
};
```

```
//retorna verdadeiro
//se linha e box
//têm a mesma cor
int mesma_cor(linha l, box b){
  if (l.cor == b.cor)
    return (1);
  return (0);
}
```

UDESC

Funções Friend

- Comutatividade em classes *friend*:
  - A relação de amizade não é comutativa;
  - Se a classe A é *friend* da classe B então A pode acessar os dados privados de B, porém B não pode acessar os dados privados de A.

- Transitividade em classes *friend*:
  - A relação de amizade não é transitiva:
  - Se A é *friend* de B, e B é *friend* de C, isto não implica que A é *friend* de C!

UDESC

Sobrecarga de operadores

- É o mecanismo da linguagem C++ que permite a redefinição ou sobrecarga dos operadores da linguagem para permitir que certas operações possam ser escritas de forma natural;
- Consiste no acréscimo de significados aos operadores já existentes na linguagem;
- Operadores sobrecarregáveis:
  - Todos os operadores (inclusive new e delete[]), exceto: “.”, “::”, “.*”, “?”;

UDESC

Sobrecarga de operadores

- Restrições na sobrecarga:
  - Não se podem criar novos operadores além dos já existentes na linguagem;
  - O operador sobrecarregado não pode alterar as regras de precedência e associatividade estabelecidas na sua definição original;
  - Ao menos um parâmetro da função operadora deverá ser objeto de classe.
  - Não se pode combinar sobrecargas para gerar uma 3ª função operadora:
    - Sobrecarregar "+" e "=" não sobrecarrega "+=";

UDESC

Sobrecarga de operadores

- Outras restrições na sobrecarga:
  - Os seguintes operadores só podem ser sobrecarregados como método da classe:
    - Atribuição “=”;
    - Apontador-membro “->”;
    - Parênteses “( )”.

- Formas de implementar a sobrecarga:
  - Como método de classe;
  - Como função *friend* de classe.

UDESC

Templates

```
#include <iostream>
using namespace std;
class Vec3D{
  int x, y, z; //coordenadas 3-D
public:
  Vec3D operator+ (tres_d op2); //op1 está implícito
  Vec3D operator= (tres_d op2); //op1 está implícito
  Vec3D operator++ (void);      //op1 também está implícito
  void mostra(void);
  void atribui(int mx, int my, int mz);
};

Vec3D Vec3D::operator+ (Vec3D op2){
  Vec3D temp;
  temp.x = x + op2.x; //estas são adições de inteiros
  temp.y = y + op2.y; //e o + retém o seu significado
  temp.z = z + op2.z; //original relativo a eles
  return(temp);
}
```

Templates

```
Vec3D Vec3D::operator= (Vec3D op2){
  x = x + op2.x; //estas são adições de inteiros
  y = y + op2.y; //e o + retém o seu significado
  z = z + op2.z; //original relativo a eles
  return(*this);
}

//sobrecarrega um operador unário
Vec3D Vec3D::operator++ (void){
  x++;
  y++;
  z++;
  return(*this);
}

//mostra as coordenadas x, y, z
void Vec3D::mostra(void){
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}
```

UDESC

Herança

- O C++ implementa:
  - Herança simples: herdar de uma classe base;
  - Herança múltipla: herdar de várias classes base.
- Herança Simples:
  - Em C++ é implementada a partir de uma declaração de hereditariedade, na qual a classe derivada declara sua classe base, o modificador de acesso da herança e a seguir declara seus membros particulares.
  - Sintaxe:

```
class Nome_classe:[modificador] Nome_classe_base{
//declaração dos membros da classe derivada
}
```

UDESC

Herança

- Construtores e destrutores:
  - Uma classe derivada pode ter seus próprios construtores e destrutores, independentemente de sua classe base;
  - As classes derivadas não herdam os construtores e destrutores das classes base;
  - Cabe ao programador decidir como os construtores e destrutores das classes derivadas se relacionam com os das classes base;
  - Os objetos são construídos na seguinte ordem: classe base, os membros da classe e a classe derivada.

Herança

- Relacionamentos entre construtores de classes derivadas e classes base. Podem ser implícitos ou explícitos:
  - Relacionamentos implícitos:
    - Caso nada seja especificado pelo programador, o construtor da classe derivada realiza uma chamada implícita ao construtor default da classe base.
  - Relacionamentos explícitos:
    - Deve ser especificada na definição do construtor da classe derivada. Sua sintaxe é:
    - Nome_classe_derivada(parâmetros classe derivada, parâmetros classe base): Nome_classe_base(parâmetros classe base){

      //código do construtor da classe derivada

      }

Herança – Overriding

- É o mecanismo que permite às classes derivadas modificar métodos herdados das classes base;
- Possibilita introduzir alterações na forma de funcionamento das classes derivadas sem alterar a interface;
- A função sobrescrita ou substituída (*Overriding*) deve combinar em tipo de retorno, nome e lista de parâmetros com a função antecessora da classe base.

Herança múltipla

- Ocorre quando uma classe derivada possui duas ou mais classes base;
- Utilizada quando se precisa combinar características de duas ou mais classes diferentes em uma única classe;
- Sintaxe:

```
class Nome_classe: [modif_1] classe_base1, [modif_2] classe_base2, ..., [modif_n] classe_base_n
{
//Atributos e métodos da classe
}
```

UDESC

Ambiguidade em herança múltipla

- Quando duas ou mais classes base possuem atributos ou métodos homônimos, e a classe derivada não sobrescreveu estes atributos ou métodos, isto pode causar ambiguidade;
- Para evitá-la pode-se utilizar o operador de resolução de escopo para referir-se ao método, da classe base correta, ao qual se deseja referenciar.

UDESC

Class x Struct

- Uma struct em C++ não é apenas uma struct como em C ANSI (embora seja sempre bom manter o padrão para não confundir programadores incautos).
- Classes podem também serem criadas usando struct. Ambos os construtores são válidos. A diferença é que se usarmos a palavra **struct** os atributos e membros serão **públicos por padrão**, enquanto nas classes eles são privados por padrão.

Estruturas de dados II

Revisão de C++

André Tavares da Silva

andre.silva@udesc.br