Algoritmos e Estruturas de Dados CES-11

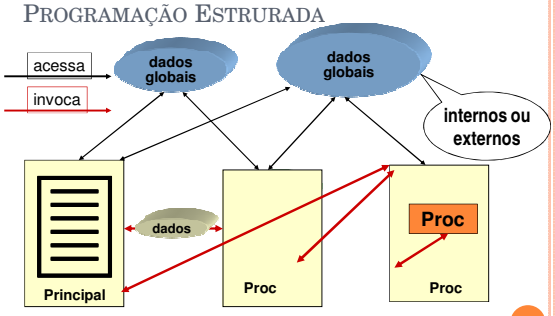
Prof. Paulo André Castro
pauloac@ita.br
Sala 110 – Prédio da Computação
www.comp.ita.br/~pauloac
IECE - ITA

Sumário

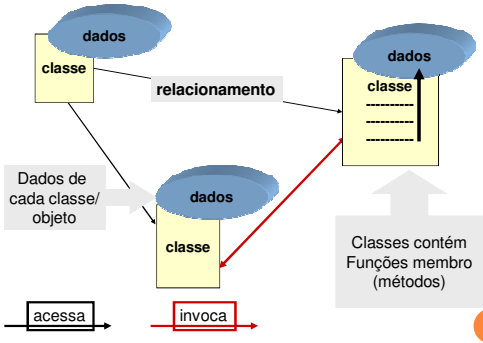
Introdução

- Programação Estruturada x Programação Orientada a Objetos
  - Modelagem com base no conceito de módulo ou sub-programa
  - Modelagem com base no conceito de classe e seus relacionamentos
- Linguagens Orientadas a Objetos
  - Simula, SmallTalk
  - C++,
  - C#, (VB?), Java,etc.

Programação Estrurada



Programação Orientada a Objetos



Conceitos Básicos de OO

- Classe: um categoria de entidades (“coisas”)
  - Corresponde a um tipo, ou coleção, ou conjunto de entidades afins
- Objeto: Uma entidade com existência física que pertence a um determinado conjunto de entidades afins (classe)
- Um objeto é uma **instância** (exemplar) de uma classe

Exemplos de Classe e Objeto

- Classes:
  - Carro, Avião, Pessoa
  - Lista, Grafo, Árvore,
- Objetos:
  - Carro: Porsche 910 Placa XXXX
  - Avião: Boeing 737-300 Prefixo: PY-XXX
  - Pessoa: José da Silva CPF: XXXXXXX
  - Lista: 32,454,13,90
  - Grafo:
  - Pilha:

7

Classes x Tipos de Dados

- Uma classe é um tipo definido pelo usuário que contém uma estrutura de dados e um conjunto de operações que atuam sobre estes dados
- Analogamente, é o mesmo que o tipo inteiro significa para as variáveis declaradas como inteiros: acesso a valores através de operações
- A classe encapsula dados e operações e controla o acesso a estas propriedades

8

Funções de E/S em C++

```
#include <iostream>
main() {
int i;     double d;          long l;

cout << "Digite um inteiro: ";
cin >> i;

cout << "Digite um double: ";
cin >> d;

cout << "Digite um long: ";
cin >> l;

cout << " inteiro:"<< i << " double:" << d << " long:" << l;
cin>>i;
}
```

9

Classes em C++

Sintaxe similar a criação de structs mas não igual:

```
class Circulo {
public:
   int x,y,raio;  // dados públicos
};

class Data {
public:
  int dia,mes,ano;
  void alteraData(int d,int m,int a);
};
```

10

Acessando variáveis de instância

- Use um ponto entre o nome da variável e o campo
  - objectName.fieldName;
- Por exemplo, usando uma classe Ponto
  - Ponto p(2,3); //criação de objeto p
  - int x2= p.x*p.x; // x2 é 4
  - int xPlusY=p.x+p.y; // xPlusY é 5
  - p.x=3;
  - x2=p.x* p.x; // x2 agora é 9
- Dentro de um objeto, suas funções membro podem acessar as variáveis de instância (e de Classe) sem utilizar o ponto

11

Funções membro e Atributos

- Classes podem ter várias funções membro e atributos
  - Função membro: define um comportamento de uma classe
  - Atributo: define uma informação a ser mantida por cada instância de uma classe
- Escopo
  - Escopo de Classe:
    - Ex. Boeing 737 atributo: número de motores
    - Declarado com o modificador ***static***
  - Escopo de Objeto ("Instância")
    - Ex. Boeing 737 atributo: número de assentos

12

Funções Membro

```
#include <iostream>
class Circulo {
public:
    int x,y,raio;
void deslocar(int dx,int dy);
void aumentarRaio(int dR);
void imprimir();
};
void Circulo::imprimir() {
  cout<<"Circulo de centro em ("<<x<<","<<y<<") e raio="<<raio;      }
void Circulo::deslocar(int dx,int dy) {
   x=x+dx;   y=y+dy;        }
void Circulo::aumentarRaio(int dR) {
  raio=raio+dR;   }
```

13

Funções Membro - 2

```
void main() {
   Circulo c1;
   c1.x=50;    c1.y=50;     c1.raio=20;

   c1.imprimir();
   c1.deslocar(5,5);
   c1.imprimir();
   c1.aumentarRaio(3);
   c1.imprimir();

}
```

14

Encapsulamento

- Encapsulamento: É a capacidade de "esconder" parte do código e dos dados do restante do programa
- Pode-se definir um grau de visibilidade as funções membro s e atributos de cada Classe.
- Há vários graus de visibilidade mas todas as linguagens implementam pelo menos os seguintes:
  - Público: Todos podem acessar (ler e escrever)
  - Privado: Apenas a própria classe pode acessar.

15

Encapsulamento

- Diretivas de Visibilidades do C++
  - **public:** visível em qualquer ponto do programa;
  - **private:** visível por membros da classe e friends;
  - **protected:** visível por membros da classe, friends, classes derivadas e friends das mesmas.

16

Encapsulamento - 2

```
class Circulo {
private:
    int x,y,raio;   // dados privativos
protected:  //dados que seriam protected
public:    //dados públicos
void set(int x,int y,int raio);
void deslocar(int dx,int dy);
void aumentarRaio(int dR);
void imprimir();
};
void Circulo::set(int nx,int ny,int nr) {
    x= nx>0 && nx<1000?nx:x;
    y= ny>0 && ny<1000?ny:y;
    raio= nr>0 && nr<1000?nr:raio;
}
```

17

Construtores e Destrutores

- Construtores são funções membro especiais utilizadas para inicializar objetos e são chamados no momento da criação de um objeto
- Destrutores são funções membro especiais para liberar memória e/ou fazer operações de deslocação de recursos ao destruir objetos

18

Construtor

```
class Fila {
private:
   int *valores;
   int num;
public:
Fila(int tam);
void colocarNoFinal(int v);
int removerInicio();
};
Fila::Fila(int tam) {
    valores=(int*)malloc(tam*sizeof(int)); num=0;
}
```

19

Destrutor

```
class Fila {
private:
   int *valores; int num;
public:
Fila(int tam);
~ Fila ();
void colocarNoFinal(int v);
int removerInicio();
};
Fila::~ Fila () {
    free(valores);
}
```

20

Exercício

- Criar uma classe que implemente a estrutura de dados conhecida como pilha (para inteiros) com operações de push e pop
- Criar um programa simples que instancie e utilize a classe Pilha

21

Pilhas (*stacks*)

- Operações sobre Pilhas:
  - push(x): insere x no topo
  - pop(): retira o elemento topo
  - top(): retorna o topo sem desempilhá-lo
  - size(): retorna o tamanho atual da pilha
  - isEmpty(): verifica se a pilha está vazia

Pilha

```
class Pilha {
 private:
   int *valores;     int topo;
int TAM;    int ERRO;
public:
int push(int x); //insere x no topo
int pop(); // retira o elemento topo
int top();  // retorna o topo sem desempilhá-lo
int size();  //retorna o tamanho atual da pilha
bool isEmpty(); // verifica se a pilha está vazia
//construtor e destrutor
 Pilha();
 ~Pilha();
};
```

23

Funções Membro da Pilha

```
int Pilha::size() { return topo+1; }

bool Pilha::isEmpty() {return (topo<0); }

int Pilha::top() {
   if (isEmpty())
     return ERRO;
  return valores[topo];
}
```

24

Funções Membro da Pilha - 2

```
int Pilha::push(int x) {
  if (size() == TAM)
     return ERRO;
  valores[++topo]=x;
  return !ERRO;
}

int Pilha::pop() {
if (isEmpty())   return ERRO;
return valores[topo--]; }
```

25

Construtor e Destrutor

```
Pilha::Pilha() {
  TAM=100; ERRO=-1;  topo = -1;
 valores=(int*)malloc(TAM*sizeof(int));
}
Pilha::~Pilha() {
 free(valores);
}
```

26

Utilização da PILHA

```
void main() {
 Pilha pilha;

 pilha.push(3);
 cout<<"Empilha "<<3<<endl;
 pilha.push(4);
 cout<<"Empilha "<<4<<endl;
 pilha.push(5);
 cout<<"Empilha "<<5<<endl;

 cout<<"Desempilha "<<pilha.pop()<<endl;
 cout<<"Desempilha "<<pilha.pop()<<endl;
 cout<<"Desempilha "<<pilha.pop()<<endl;
 getch();
}
```

27

Resultado da Execução

- Empilha 3
- Empilha 4
- Empilha 5
- Desempilha 5
- Desempilha 4
- Desempilha 3

28

Exercício 2

- Faça com que a pilha dobre de tamanho sempre que seja necessário mais espaço
- Faça isto com o mínimo de alterações no código que utiliza a pilha.

29

Alteração - 1

```
int Pilha::push(int x) {
  if (size() == Pilha::TAM) {
     TAM=TAM*2;
      int *aux=(int*)malloc(TAM*sizeof(int));
      copiar_dados(aux,valores);
      free(valores);
      valores=aux;
  }
     valores[++topo]=x;
  return !ERRO;
}
void Pilha::copiar_dados(int* aux,int *valores) {
  for(int i=0;i<=topo;i++)
      aux[i]=valores[i];
}
```

30

ALTERAÇÃO - 2

```
class Pilha {
 private:
   int *valores;     int topo;
int TAM;            int ERRO;
   void copiar_dados(int *, int*);
public:
int push(int x); //insere x no topo
int pop(); // retira o elemento topo
int top();  // retorna o topo sem desempilhá-lo
int size();  //retorna o tamanho atual da pilha
bool isEmpty(); // verifica se a pilha está vazia
//construtor e destrutor
 Pilha();
 ~Pilha();
};
```

31

UTILIZAÇÃO DA PILHA – NENHUMA ALTERAÇÃO

```
void main() {
 Pilha pilha;

 pilha.push(3);
 cout<<"Empilha "<<3<<endl;
 pilha.push(4);
 cout<<"Empilha "<<4<<endl;
 pilha.push(5);
 cout<<"Empilha "<<5<<endl;

 cout<<"Desempilha "<<pilha.pop()<<endl;
 cout<<"Desempilha "<<pilha.pop()<<endl;
 cout<<"Desempilha "<<pilha.pop()<<endl;
 getch();
}
```

32

SOBRECARGA DE FUNÇÕES-MEMBRO

- Muitas vezes, desejamos que uma função possa receber diferentes tipos de parâmetros.
- A Sobrecarga de funções permite a existência de vários funções de mesmo nome, porém com protótipos levemente diferentes ou seja variando no número e tipo de parâmetros e no valor de retorno
- Exemplo: Classe que calcula logaritmos

```
#include <math.h>
class Logaritmo {
public:
double log(double x) {//retorna por default log base 10
    return log10(x);
 }
double log(double x, double b) {
    return log10(x)/log10(b);
    }
}
```

33

SOBRECARGA DE FUNÇÕES-MEMBRO

- A sobrecarga também pode ser usada em construtores (lembre-se construtores são também funções-membro)
- Exemplo: Criar um tamanho default para inicialização da nossa classe Pilha, mas permitir que o programador define o tamanho se desejar….

```
class Pilha {
 private:
  .........
//construtor e destrutor
 Pilha();          Pilha(int t);
 ...................
};
```

34

EXERCÍCIO 3

- Alterar a classe pilha para que o programador que a utiliza possa especificar qual o tamanho inicial da pilha se assim o desejar.

35

EXERCÍCIO 3 - ALTERAÇÃO

```
class Pilha {
 private:
   int *valores;     int topo;
int TAM;            int ERRO;
   void copiar_dados(int *, int*);
public:
int push(int x); //insere x no topo
int pop(); // retira o elemento topo
int top();  // retorna o topo sem desempilhá-lo
int size();  //retorna o tamanho atual da pilha
bool isEmpty(); // verifica se a pilha está vazia
//construtor e destrutor
 Pilha();          Pilha(int t);
 ~Pilha();
};
```

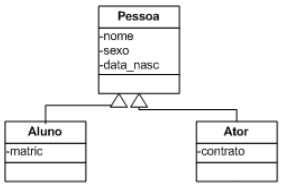
36

Exercício 3 – Alteração 2

```
Pilha::Pilha(int t) {
  topo=-1;
 TAM=t;
 ERRO=-1;
 valores=(int*)malloc(TAM*sizeof(int));
}
```

37

Herança ("Derivação") – relação "é um"



38

Classes Base e Classes Derivadas

- Classe, classe-pai, super-classe, **classe base**:
  - Carro
  - Motor
  - Avião
- Sub-classe, classe-filha, **classe derivada**
  - Carro: Porsche 910
  - Motor: Ford 16V
  - Avião: Boeing 737

39

Relação de Herança

- O filho herda todas as características do pai
  - Comportamento: funções
  - Atributos: valores
- Em linguagens OO, geralmente há meios de restringir o que será ou não herdado
- Diretivas de visibilidade em herança C++
  - public: visibilidade da classe base não é alterada;
  - private: membros public e protected da classe base são private na classe derivada;
  - protected: membros public e protected da classe base são protected na classe derivada.

40

Herança

- Herança quando uma classe tem implicitamente características de outras classes
- Herança Múltipla: quando uma classe pode herdar funções membro e atributos de várias classes
- Uma classe pode herdar apenas de uma outra classe em algumas linguagens (Java)

Exemplo – C++

```
class Pessoa{
    public:
    string nome;
    char sexo;
    int idade;

};
```

```
class Ator: public Pessoa{
        public: string contrato;
        /* campos herdados
        String nome;
        char sexo;
        int idade; */
    };
```

```
class Aluno : public Pessoa{
        public: long matric;
        /* campos herdados
        string nome;
        char sexo;
        int idade; */
 };
```

42

Construtores de classes derivadas

```
class Pessoa{
    public:
    char* nome;
    char sexo;
    int idade ;
   Pessoa( char* n,char s, int i) { nome=n; sexo=s; idade=i;
   }  //uma função membro com corpo dentro da classe é inline
} ;

class Ator: public Pessoa{
   public: char* contrato;
     Ator( char* n,char s, int i, char* c) : Pessoa(n,s,i) { contrato=c;}
};
```

43

Construtores de classes derivadas

```
class Pessoa{
    public:
    char* nome;
    char sexo;
    int idade ;
   Pessoa( char* n,char s, int i) { nome=n; sexo=s; idade=i;
   }  //uma função membro com corpo dentro da classe é inline
} ;

class Aluno : public Pessoa{
   public: long matric;
    Aluno( char* n,char s, int i, long m) : Pessoa(n,s,i) { matric=m;}
 };
```

44

Exemplo – C++

```
....
Pessoa pessoa("Alan",'m',20);
 Ator ator("Beth",'f',21,"cont. tipo1");
 Aluno aluno("Charlie",'m',22,20000);
cout<<"Nome:"<<pessoa.nome<<" Sexo:"<<pessoa.sexo<<
   "Idade:"<<pessoa.idade<<endl;
cout<<"Nome:"<<ator.nome<<" Sexo:"<<ator.sexo<<
   "Idade:"<<ator.idade<<  "Contrato:"<<ator.contrato<<endl;
cout<<"Nome:"<<aluno.nome<<"  Sexo:"<<aluno.sexo<<
   "Idade:"<<aluno.idade<<" Matricula:"<<aluno.matric<<endl;
```

45

Resultado

- Nome:Alan Sexo:m Idade:20
- Nome:Beth Sexo:f Idade:21 Contrato:cont. tipo1
- Nome:Charlie Sexo:m Idade:22 Matricula:20000

46

Polimorfismo e Sobreescrita de métodos

- Um mesmo comando enviado para objetos diferentes gera (ou pode gerar) ações diferentes.
- Exemplo:
  - Comando: Mover
    - Carro
    - Avião
    - Pessoa
- A chamada de uma mesma função pode levar a ser executadas funções diferentes
- Ex.: EntidadeMovel->mover();
  - //Pode resultar em Carro::mover(); Aviao::mover(); ou Pessoa::mover();
- Em C++, deve-se definir este tipo de função como **virtual**

47

Exemplo de Orientação a Objetos –Pessoa e Casado

```
class Pessoa{
    public:
    char* nome;
    char sexo;
    int idade ;
   Pessoa( char* n,char s, int i) { nome=n; sexo=s; idade=i;
   }  //uma função membro com corpo dentro da classe é inline
   virtual bool ehResponsavel();
} ;

bool Pessoa::ehResponsavel(){
       if(idade>18)  return true;  else return false;
}
```

Exemplo de Orientação a Objetos – Pessoa e Casado

```
class Casado :  public Pessoa  {
public:
 virtual bool ehResponsavel();
   Casado( char* n,char s, int i) : Pessoa(n,s,i) {    }
};

bool Casado::ehResponsavel() {
            return true;
}
```

49

Exemplo 1 – C++

```
void main() {
Pessoa pessoa("Alan",'m',20);
 Casado pessoa_casada("Beth",'f',17);
 Pessoa *p1=new Casado("Charlie",'m',17);
cout<<"\nNome:"<<pessoa.nome;
  if(pessoa.ehResponsavel())     cout<<" Legalmente responsável";
  else     cout<<" Não legalmente responsável";

cout<<"\nNome:"<<pessoa_casada.nome;
  if(pessoa_casada.ehResponsavel())  cout<<" Legalmente responsável";
  else   cout<<" Não legalmente responsável";

cout<<"\nNome:"<<p1->nome;
  if(p1->ehResponsavel())       cout<<" Legalmente responsável";
  else     cout<<" Não legalmente responsável";
//Resultado????
}
```

50

Resultado – Exemplo 1

- Nome:Alan Legalmente responsável
- Nome:Beth Legalmente responsável
- Nome:Charlie Legalmente responsável

51

Exemplo 2 – C++

```
void main() {
Pessoa pessoa("Alan",'m',20);
 Casado pessoa_casada("Beth",'f',17);
 Pessoa *p1=new Pessoa("Charlie",'m',17);
cout<<"\nNome:"<<pessoa.nome;
  if(pessoa.ehResponsavel())     cout<<" Legalmente responsável";
  else     cout<<" Não legalmente responsável";

cout<<"\nNome:"<<pessoa_casada.nome;
  if(pessoa_casada.ehResponsavel())  cout<<" Legalmente responsável";
  else   cout<<" Não legalmente responsável";

cout<<"\nNome:"<<p1->nome;
  if(p1->ehResponsavel())       cout<<" Legalmente responsável";
  else     cout<<" Não legalmente responsável";
//Resultado????
}
```

52

Resultado – Exemplo 2

- Nome:Alan Legalmente responsável
- Nome:Beth Legalmente responsável
- Nome:Charlie Não Legalmente responsável

53