FACAMP
FACULDADES DE CAMPINAS

Linguagens de Programação

Aula 12

Suporte à programação orientada a objetos

2º semestre de 2019

Prof José Martins Junior

Introdução

- Muitas linguagens suportam programação orientada a objeto
  - De Cobol a Lisp
  - C++ e Ada (procedural e orientada a objeto)
  - CLOS (linguagem funcional)
  - Java e C# (orientada a objetos mas usa construções imperativas)
  - Smalltalk (orientada a objeto pura)

- Uma linguagem que suporta POO deve ter três características
  - Tipo abstrato de dados (encapsulamento)
  - Herança
  - Vinculação dinâmica de chamada de métodos (polimorfismo)

Herança

- Limitações dos TAD's
  - Reuso: alguma mudança sempre é necessária
  - Organização: todos TAD's estão no mesmo nível
- A herança permite criar novos tipos baseados em tipos existentes
  - Reuso
    - Um novo tipo herda os dados e funcionalidades de um tipo já existente
    - É possível adicionar novos dados e funcionalidades
    - É possível alterar as funcionalidades existentes
  - Organização
    - É possível criar hierarquia de classes

Herança

- TADs em linguagens OO são chamados de **classes**
- Uma instância de uma classe é chamada de **objeto**
- Uma classe que é definida através de herança é chamada de **classe derivada** ou **subclasse**
- Uma classe da qual uma nova classe é derivada é chamada de **classe pai** ou **superclasse**
- Os subprogramas que definem operações sobre os objetos de uma classe são chamados de **métodos**
  - A chamada de métodos também é conhecida como mensagem

Herança

- Uma classe derivada pode diferenciar-se da classe pai de várias maneiras
  - Alguns membros da classe pai podem ser privados, o que implica que eles não são visíveis nas subclasses
  - A subclasse pode adicionar novos membros
  - A subclasse pode modificar o comportamento dos métodos herdados
    - O novo método sobrescreve o método herdado
    - O método do pai é sobrescrito

Herança

- Classes podem ter dois tipos de variáveis
  - De instância e de classe
- Classes podem ter dois tipos de métodos
  - De instância e de classe

- Herança simples
  - Subclasse de uma única classe pai
- Herança múltipla
  - Subclasse de múltiplas classes pai

- Desvantagem da herança
  - Cria dependências entre as classes da hierarquia

Vinculação dinâmica

- Uma variável polimórfica de uma classe é capaz de referenciar para objetos da classe e objetos de qualquer subclasse
  - Quando uma classe em uma hierarquia de classes sobrescreve um método, e este método é chamado através de uma variável polimórfica, a vinculação para o método correto será dinâmica
  - Permite que softwares sejam mais facilmente modificados

- Classe abstrata
  - Um método abstrato não contém a definição, ele define apenas o protocolo
  - Uma classe abstrata contém pelo menos um método abstrato
  - Uma classe abstrata não pode ser instanciada

Suporte a POO em Smalltalk

- Características gerais
  - Tudo é objeto
  - A computação ocorre através de troca de mensagens
  - Mensagens podem ser parametrizadas
  - Todos os objetos são alocados do heap
  - A desalocação é implícita
  - Os construtores precisam ser chamados explicitamente
  - Não tem a aparência de linguagens imperativas

Suporte a POO em Smalltalk

- Checagem de tipo e polimorfismo
  - A vinculação das mensagens para métodos é dinâmica
    - Quando uma mensagem é enviada para um objeto, o método corresponde é buscado na classe do objeto, se o método não for encontrado a busca continua na classe pai e assim sucessivamente; o processo continua até a classe Object
  - A checagem de tipo é dinâmica e um erro acontece quando uma mensagem é enviada para um objeto que não tem um método correspondente
- Herança
  - Uma subclasse herda todas as variáveis de instância, métodos de instância e métodos de classe da classe pai
  - Uma subclasse pode acessar um método sobrescrito usando a pseudo variável super
  - Herança simples

Suporte a POO em C++

- Característica gerais
  - Baseado em C e em SIMULA 67
  - Primeira linguagem de programação OO amplamente utilizada
  - Dois sistemas de tipo: imperativo e OO
  - Objetos podem ser estáticos, dinâmicos na pilha ou dinâmicos no heap
  - Desalocação explícita usando o operador delete
  - Destrutores
  - Mecanismo de controle de acesso elaborado

Suporte a POO em C++

- Herança
  - Tipos de controle de acesso aos membros

    `private, protected e public`
  - Um classe não precisa ter uma classe pai
  - Todos os objetos precisam ser inicializado antes de serem usados
    - No caso de subclasse, os membros herdados precisam ser inicializados quando uma instância da subclasse é criada
  - Suporta herança múltipla
    - Se dois membros herdados tem o mesmo nome, eles podem ser acessados usando o operador de resolução de escopo (::)
  - Um método da subclasse precisa ter os mesmos parâmetros do método da classe pai para sobrescrevê-lo
    - O tipo de retorno deve ser o mesmo ou um tipo derivado (público)

Exemplo de POO em C++

```
class single_linked_list {
  private:  class
    node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0;};
    void insert_at_head(int);
    void insert_at_tail(int);
    int remove_at_head();  bool
    empty();
};
```

Exemplo de POO em C++

```
// Como stack é uma derivação pública, todos os
// métodos públicos da classe single_linked_list
// também são públicos em stack, o que deixa a
// classe com métodos públicos indesejados
// (insert_at_head, insert_at_tail e remove_at_head)

class stack: public single_linked_list {
  public:
     stack() {}
     void push(int value) {
        insert_at_head(value);
     }
     int pop() {
        return remove_at_head();
     }
};
```

Exemplo de POO em C++

```
// stack2 é uma derivação privada de
// single_linked_list, portanto os membros
// públicos e protegidos herdados de
// single_linked_list são privados em stack2.
// stack2 tem que redefinir a visibilidade do
// membro empty para torná-lo público.

class stack2: private single_linked_list {
  public:
     stack2() {}
     void push(int value) {
        insert_at_head(value);
     }
     int pop() {
        return remove_at_head();
     }
     single_linked_list::empty;
};
```

Exemplo de POO em C++

```
// Uma alternativa mais interessante é usar composição,
// o que permite que uma pilha possa ser
// definida com qualquer implementação de lista.
class stack3 {
  private:
    list *li;
  public:
    stack3(list *l) : li(l) {}
    void push(int value) {
      li->insert_at_head(value);
    }
    int pop() {
      return li->remove_at_head();
    }
    boolean empty() {
      return li->empty();
    }
};
```

Suporte a POO em C++

- Vinculação dinâmica
  - Um método pode ser definido como virtual, que significa que ele será vinculado dinamicamente quando chamado em uma variável polimórfica
  - Funções virtuais puras não têm definição
  - Uma classe que tem pelo menos um método virtual puro é uma classe abstrata

Exemplo de vinculação em C++

```
class Shape {
  public:
    virtual void name() = 0;
};
class Rectangle: public Shape {
  public:
    void name() {
        printf("Rectangle\n");
    }
    void code() {
        printf("R\n");
    }
};
class Square: public Rectangle {
  public:
    void name() {
        printf("Square\n");
    }
    void code() {
        printf("S\n");
    }
};
```

O que será impresso?

```
Shape* s = new Rectangle();
s->name();
...
s = new Square();
s->name();
```

```
Rectangle
Square
```

`name` é um método virtual e portanto é vinculado dinamicamente ao método da classe **instanciada** referenciada por `r`.

Exemplo de vinculação em C++

```
class Shape {
  public:
    virtual void name() = 0;
};
class Rectangle: public Shape {
  public:
    void name() {
        printf("Rectangle\n");
    }
    void code() {
        printf("R\n");
    }
};
class Square: public Rectangle {
  public:
    void name() {
        printf("Square\n");
    }
    void code() {
        printf("S\n");
    }
};
```

O que será impresso?

```
Rectangle *r = new Rectangle();
r->code();
...
r = new Square();
r->code();
```

```
R
R
```

`code` **não** é um método virtual e portanto é vinculado estaticamente ao método do tipo declarado de `r`.

Suporte a POO em Java

- Características gerais
  - Similar a C++
  - Todos os tipos são objetos, exceto os primitivos (boolean, char, numéricos)
  - Para os tipos primitivos poderem ser usados com tipos objetos, eles devem ser colocados em objetos que são invólucros (wrappers)
    - Java 5 adicionou autoboxing e autounboxing
  - Todas as classes são descendentes de `Object`
  - Todos objetos são dinâmicos no heap e desalocação é implícita
    - O método `finalize` é executado quando o objeto é desalocado pelo coletor de lixo
    - Como o momento exato que finalize será executado não pode ser determinado, é necessário outro mecanismo definido pelo usuário para desalocação de recursos

Suporte a POO em Java

- Herança
  - Um método pode ser declarado `final`, o que significa que ele não pode ser sobrescrito
  - Uma classe pode ser declarada `final`, o que significa que ela não pode ser a classe pai de nenhuma outra classe
  - O construtor da classe pai deve ser chamado antes do construtor da subclasse
  - Toda subclasse é subtipo
  - Suporta apenas herança simples
  - Uma interface é semelhante a uma classe abstrata, mas pode ter apenas as declarações dos métodos e constantes
  - Uma classe pode implementar mais de uma interface (mix-in)
- Vinculação dinâmica
  - Todas as mensagens são vinculadas dinamicamente a métodos, exceto se o método for `final`, `private` ou `static`

Suporte a POO em Java

- Classes aninhadas
  - Várias formas de classes aninhadas
  - A classe que encapsula a classe aninhada pode acessar qualquer membro da classe aninhada
  - Uma classe aninhada não estática, tem uma referência para uma instância da classe que a encapsula e portanto pode acessar os membros desta instância
  - A classe aninhada pode acessar qualquer membro da classe que a encapsula
  - Classes aninhadas podem ser anônimas
  - Uma classe aninhada pode ser declarada em um método

Suporte a POO em C#

- Características gerais
  - Inclui classes e estruturas
  - As classes são semelhantes as classes em Java
  - As estruturas são alocadas na pilha e não oferece herança
- Herança
  - Mesma sintaxe utilizada em C++
  - Um método herdado da classe pai pode ser substituído por uma na classe derivada marcando o método com new
  - O método substituído da classe pai pode ser acesso com o prefixo base
  - O suporte a interface é o mesmo que o do Java

Suporte a POO em C#

- Vinculação dinâmica
  - A classe pai precisa marcar o método com `virtual`
  - A subclasse precisa marcar o método com `override`
  - Um método pode ser marcado como `abstract`
  - Uma classe que contém pelo menos um método abstrato precisa ser marcada como `abstract`
  - Todos as classes são descendentes de `Object`
- Classes aninhadas
  - Uma classe aninhada é como uma classe aninhada estática em Java

Registro de instância de classes (RIC)

- Duas questões importantes
  - Estrutura de armazenamento para variáveis de instâncias
  - Vinculação dinâmica de mensagens para métodos

Registro de instância de classes (RIC)

- O RIC armazena o estado do objeto
  - É estático (construído em tempo de compilação)
  - Se uma classe tem pai, as variáveis de instância da subclasse são adicionadas ao RIC da classe pai
  - O acesso as variáveis é feito como em registros (usando um deslocamento)
  - Os métodos que são vinculados estaticamente e não precisam estar envolvidos com o RIC
  - A chamada a métodos vinculados dinamicamente pode ser implementada como um ponteiro no RIC
  - Todos os ponteiros para métodos podem ser armazenados em uma tabela de métodos virtuais (vtable), compartilhada pelas instâncias
  - Um método é representado como um deslocamento a partir do início da tabela

Exemplo de herança simples com Java

```
public class A {
    public int a, b;
    public void draw() { . . . }
    public void area() { . . . }
}

public class B extends A {
    public int c, d;
    public void draw() { . . . }
    public void sift() { . . . }
}
```

Exemplo de herança simples com Java

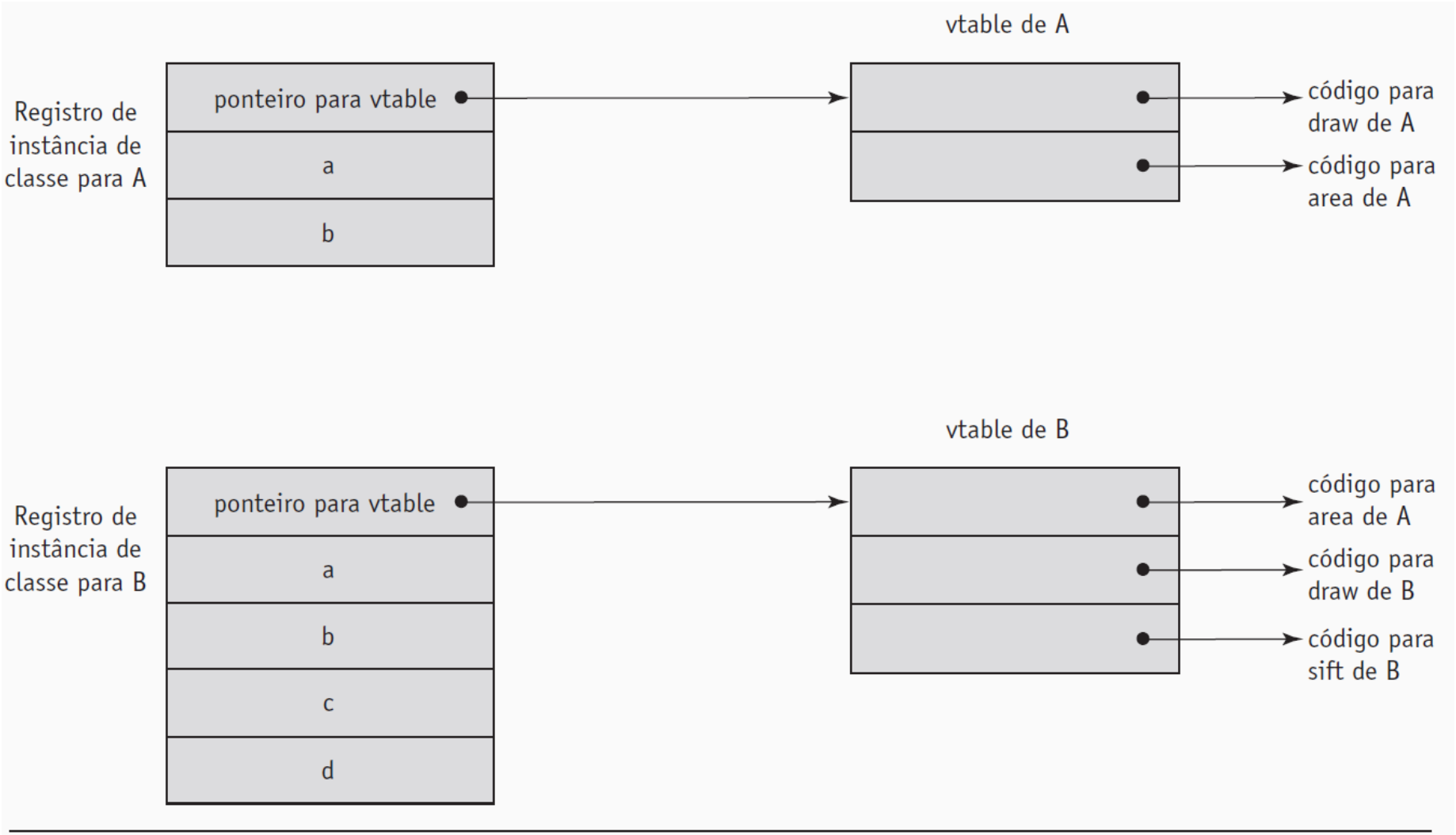

Figura 12.2 Um exemplo das RICs com herança simples.

Exemplo de herança múltipla com C++

```
class A {
    public: int a;
    virtual void fun() { . . . }
    virtual void init() { . . . }
}
class B {
    public: int b;
    virtual void sun() { . . . }
}
class C: public A, public B {
    public: int c;
    void fun() { . . . }
    virtual void dud() { . . . }
}
```

Exemplo de herança múltipla com C++

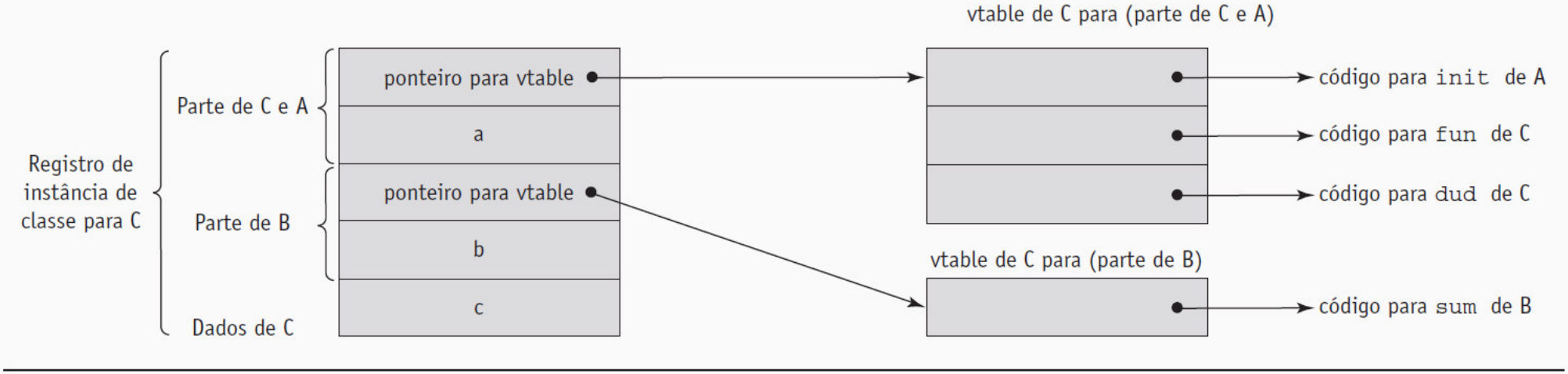

Figura 12.3 Um exemplo de um RIC de uma subclasse com múltiplos pais.

Bibliografia

- SEBESTA, R. W. Conceitos de Linguagens de Programação. Porto Alegre: Bookman, 2011. 792p.