

Linguagens de Programação

Conceitos e Técnicas



Polimorfismo

Prof.

Verificação Estática de Tipos

- Todos os parâmetros e variáveis devem possuir um tipo fixo identificável a partir da análise do texto do programa
- Tipo de cada expressão pode ser identificado e cada operação pode ser verificada em tempo de compilação
 - C faz poucas verificações
 - Reduz confiabilidade
 - MODULA-2 faz muitas verificações
 - Reduz redigibilidade

Verificação Dinâmica de Tipos

- Em tempo de execução
- Somente os valores dessas LPs têm tipo fixo
 - Cada valor tem associado a ele um `tag` indicando o seu tipo
- Uma variável ou parâmetro não possui um tipo associado
 - Pode designar valores de diferentes tipos em pontos distintos da execução
 - Verificação dos tipos de operandos imediatamente antes da execução da operação
- LISP, BASIC, APL e SMALLTALK

Verificação de Tipos Mista

- Maior parte das verificações de tipos em tempo de compilação
- Algumas verificações em tempo de execução
 - Programas em LPs orientadas a objetos podem precisar verificar a conversão de tipos de objetos durante a execução
- C++, ADA e JAVA

Sistemas de Tipos Monomórficos

- Todas constantes, variáveis e subprogramas devem ser definidos com um tipo específico
- PASCAL e MODULA-2
- Simples mas com baixa reusabilidade e redigibilidade
- Muitos algoritmos e estruturas de dados são inerentemente genéricos
 - Algoritmo para ordenação depende parcialmente do tipo do elemento a ser ordenado
 - Tipo do elemento precisa ter operação de comparação
 - Conjuntos e suas operações são totalmente independentes do tipo dos elementos

Sistemas de Tipos Monomórficos

- Nenhuma LP é totalmente monomórfica
- PASCAL
 - *read, readln, write, writeln e eof*
 - Vetores, conjuntos, arquivos
- PASCAL e MODULA-2
 - Operadores (como +) atuam sobre diversos tipos
- Linguagens monomórficas exigem que se crie representação e operações distintas para cada tipo de elemento
 - Lista de inteiros, lista de cadeia de caracteres, etc

Sistemas de Tipos Polimórficos

- Favorecem a construção e uso de estruturas de dados e algoritmos que atuam sobre elementos de tipos diversos
 - Subprogramas polimórficos são aqueles cujos parâmetros (e, no caso de funções, o tipo de retorno) podem assumir valores de mais de um tipo
 - Tipos de dados polimórficos são aqueles cujas operações são aplicáveis a valores de mais de um tipo
- C
 - *void** possibilita a criação de variáveis e parâmetros cujos valores podem ser ponteiros para tipos quaisquer
 - possibilita a construção de estruturas de dados genéricas
 - funções com lista de parâmetros variável também são polimórficas

Tipos de Polimorfismo

- Classificação de Cardelli e Wegner

Polimorfismo	Ad-hoc	Coerção
		Sobrecarga
	Universal	Paramétrico
		Inclusão

Tipos de Polimorfismo

- Ad-hoc
 - Se aplica apenas a subprogramas
 - Aparentemente proporciona reuso do código de implementação do subprograma, mas na realidade não o faz
 - São criados subprogramas específicos para operar sobre cada tipo admissível
 - Somente proporciona reuso do código de chamada dos subprogramas

Tipos de Polimorfismo

- Universal
 - Se aplica a subprogramas e estruturas de dados
 - estrutura de dados pode ser criada incorporando elementos de tipos diversos
 - mesmo código pode ser executado e atuar sobre elementos de diferentes tipos
 - Proporciona reuso de código tanto na chamada quanto na implementação
 - Considerado o verdadeiro polimorfismo

Coerção

- Conversão implícita de tipos

```
void f (float i) { }  
main() {  
    long num;  
    f (num);  
}
```

- *f* aparenta lidar com *float* e *long*
- De fato, *f* lida apenas com *float*
- Compilador se encarrega de embutir código para transformar *long* em *float*
- *C* possui tabela de conversões permitidas

Coerção

- Ampliação
 - Tipo de menor conjunto de valores para tipo de maior conjunto
 - Operação segura pois valor do tipo menor necessariamente tem correspondente no tipo maior
- Estreitamento
 - Tipo de maior conjunto de valores para tipo de menor conjunto
 - Operação insegura pois pode haver perda de informação
- Nem sempre é ampliação ou estreitamento
 - *int* para *unsigned* em C

Coerção

- Dão a entender que determinada operação ou subprograma pode ser realizada com operandos de tipos diferentes mas não é isso o que ocorre

```
main() {  
    int w = 3;  
    float x;  
    float y = 5.2;  
    x = x + y;           // x = somafloat (x, y)  
    x = x + w;          // x = somafloat (x, intToFloat (w) );  
}
```

Coerção

- Ocorre com grande frequência em atribuições e expressões em C

```
main() {  
    int i;  
    char c = 'a';  
    float x;  
    i = c;  
    c = i + 1;  
    x = i;  
    i = x / 7;  
}
```

Coerção

- Existem opiniões controversas a seu respeito
 - Maior redigibilidade por não demandar chamada de funções de conversão
 - Menor confiabilidade pois podem impedir a detecção de certos tipos de erros por parte do compilador

```
main() {  
    int a, b = 2, c = 3;  
    float d;  
    d = a * b;           // d = (float) (a * b) ;  
    a = b * d;           // a = b * c;  
}
```

Coerção

- ADA e MODULA-2 não admitem coerções
- Python admite apenas entre tipos numéricos
- C adota uma postura bastante permissiva
- JAVA busca um meio termo só admitindo a realização de coerções para tipos mais amplos

```
byte a, b = 10, c = 10;
```

```
int d;
```

```
d = b;
```

```
c = (byte) d;
```

```
a = (byte) (b + c);
```


Sobrecarga

- Quando identificador ou operador é usado para designar duas ou mais operações distintas
- Aceitável quando o uso do operador ou identificador não é ambíguo
 - A operação apropriada pode ser identificada usando-se as informações do contexto de aplicação

Sobrecarga

- Operador - de C pode designar
 - Negações inteiras
 - (*int* → *int* ou *short* → *short* ou *long* → *long*)
 - Negações reais
 - (*float* → *float* ou *double* → *double*)
 - Subtrações inteiras
 - (*int* × *int* → *int*)
 - Subtrações reais
 - (*float* × *float* → *float*)
- Conjunto de operações pode ser muito maior

Sobrecarga

- Sugere que determinada operação ou subprograma pode ser realizada com operandos de tipos diferentes mas não é isso que ocorre

```
main(){
    int a = 2, b = 3;
    float x = 1.5, y = 3.4;
    a = a + b;           // a = somaint (a, b);
    x = x + y;          // x = somafloat (x, y);
}
```

Sobrecarga

- MODULA-2 e C
 - Embutem sobrecarga em seus operadores
 - Programadores não podem implementar novas sobrecargas de operadores
 - Não existe qualquer sobrecarga de subprogramas
- PASCAL
 - Existem subprogramas sobrecarregados na biblioteca padrão
 - *read e write*
 - Programadores não podem implementar novas sobrecargas de subprogramas

Sobrecarga

- **JAVA**
 - Embute sobrecarga em operadores e em subprogramas de suas bibliotecas
 - Só subprogramas podem ser sobrecarregados pelo programador
- **ADA e C++**
 - Adotam a postura mais ampla e ortogonal
 - Realizam e permitem que programadores realizem sobrecarga de subprogramas e operadores
 - Não admitem a criação de novos operadores
 - Sintaxe e precedência não pode ser alterada

Sobrecarga em C++

```
class umValor {
    int v;
public:
    umValor() { v = 0; }
    umValor(int j) { v = j; }
    const umValor operator+(const umValor& u) const {
        return umValor(v + u.v);
    }
    umValor& operator+=(const umValor& u) {
        v += u.v;
        return *this;
    }
}
```

Sobrecarga em C++

```
const umValor& operator++() { // prefixado
    v++;
    return *this;
}
const umValor operator++(int) { // posfixado
    umValor antes(v);
    v++;
    return antes;
}
};
```

Sobrecarga em C++

```
main() {  
    int a = 1, b = 2, c = 3;  
    c += a + b;  
    umValor r(1), s(2), t;  
    t += r + s;  
    r = ++s;  
    s = t++;  
}
```

- Útil na criação de objetos em diferentes contextos
- Sintaxe esquisita para sobrecarga de `++` e `--`
- Nem todos operadores podem ser sobrecarregados
 - `::` (resolução de escopo), `.` (seleção de membro), `sizeof` (tamanho do objeto/tipo) não podem

Sobrecarga

- Independente de Contexto
 - Lista de parâmetros diferenciada em número ou tipo dos parâmetros
 - Tipo de retorno não pode ser usado para diferenciação
 - JAVA e C++
- Dependente de Contexto
 - Necessário apenas uma assinatura diferenciada
 - Tipo de retorno pode ser usado para diferenciação
 - Exige mais esforço dos compiladores
 - Pode provocar erros de ambigüidade
 - ADA

Sobrecarga Independente de Contexto

```
void f(void) { }
void f(float) { }
void f(int, int) { }
void f(float, float) { }
// int f(void) { }
main() {
    f();
    f(2.3);
    f(4, 5);
    f(2.2f, 7.3f);
    // f(3, 5.1f);
    // f(1l, 2l);
}
```

⊠ Não pode diferenciar apenas por tipo de retorno

⊠ // int f(void) { }

⊠ Pode gerar ambigüidade quando combinada com coerção

⌘ // f(3, 5.1f);

⌘ // f(1l, 2l);

Sobrecarga Dependente de Contexto

- / designa
 - divisão real (float x float → float)
 - divisão inteira (integer x integer → integer)
- Ambigüidade mesmo sem coerção
- Sobrecarga de / (integer x integer → float)

```
function "/" (m,n : integer) return float is begin  
    return float (m) / float (n);
```

```
end "/";
```

```
n : integer; x : float;
```

```
x: = 7.0/2.0;          -- calcula 7.0/2.0 = 3.5
```

```
x: = 7/2;             -- calcula 7/2 = 3.5
```

```
n: = 7/2;             -- calcula 7/2 = 3
```

```
n: = (7/2) / (5/2); -- calcula (7/2)/(5/2) = 3/2 = 1
```

```
x: = (7/2) / (5/2); -- erro: ambigüidade (pode ser 1.4 ou 1.5)
```

Sobrecarga

- Nem todos consideram uma característica desejável a possibilidade dos programadores sobrecarregarem os operadores
- Programas podem ficar mais fáceis de serem lidos e redigidos com a sobrecarga
- Aumenta a complexidade da LP
- Pode ser mal utilizado, tendo efeito contrário a legibilidade

Sobrecarga

- JAVA não inclui a sobrecarga de operadores por considerá-la capaz de gerar confusões e aumentar a complexidade da LP
- Python só permite sobrecarga de operadores em métodos de classes
 - Considera que lista de parâmetros variável é suficiente para prover funcionalidade equivalente a sobrecarga

Paramétrico

- Parametrização das estruturas de dados e subprogramas com relação ao tipo do elemento sobre o qual operam
 - Abstrações recebem um parâmetro implícito adicional especificando o tipo sobre o qual elas agem
- Subprogramas específicos para cada tipo do elemento

```
int identidade (int x) {  
    return x;  
}
```
- Subprogramas genéricos

```
T identidade (T x) {  
    return x;  
}
```

Politipo

- Subprogramas genéricos possuem parâmetro tipo
 - T é parâmetro tipo em T identidade ($T \ x$)
 - Tipo retornado por identidade será o mesmo usado na chamada

```
x = identidade (3.2); // x receberá um float
```

 - Tipo de identidade é sua assinatura $T \rightarrow T$
 - Tipo como $T \rightarrow T$ é chamado de politipo porque pode derivar uma família de muitos tipos
- Não existe impedimento em se usar mais de um parâmetro tipo em um politipo
 - $U \times T \rightarrow T$ indica que parâmetros podem ser de tipos distintos e que o tipo retornado será o tipo do segundo

Paramétrico em C++

- *Uso de template*

```
template <class T>
T identidade (T x) {
    return x;
}

class tData {
    int d, m, a;
};
```

```
main () {
    int x;
    float y;
    tData d1, d2;
    x = identidade (1);
    y = identidade (2.5);
    d2 = identidade (d1);
    // y = identidade (d2);
}
```


Paramétrico em C++

- Muitas funções são parcialmente genéricas

```
template <class T>
T maior (T x, T y) {
    return x > y ? x : y;
}
class tData {
    int d, m, a;
};

main () {
    tData d1, d2;
    printf ("%d", maior (3, 5));
    printf ("%f", maior (3.1, 2.5));
    // d1 = maior (d1, d2);
}
```

- Erro de compilação porque o operador > não está definido para a classe tData
- Necessário sobrecarregar o operador > para a classe tData

Paramétrico em C++

- É possível parametrizar classes

```
template <class T, int tam>
class tPilha {
    T elem[tam];
    int topo;
public:
    tPilha(void) { topo = -1; }
    int vazia (void) { return topo == -1; }
    void empilha (T);
    void desempilha (void);
    T obtemTopo (void);
};
```

Paramétrico em C++

```
template <class T, int tam>
void tPilha<T, tam>::empilha (T el){
    if (topo < tam-1)
        elem[++topo] = el;
}
template <class T, int tam>
void tPilha<T, tam>::desempilha (void){
    if (!vazia()) topo--;
}
template <class T, int tam>
T tPilha<T, tam>::obtemTopo (void) {
    if (!this->vazia()) return elem[topo];
}
```

Paramétrico em C++

```
class tData {  
    int d, m, a;  
};
```

```
void main () {  
    tData d1, d2;  
    tPilha <int, 3> x;  
    tPilha <tData, 2> y;  
    x.empilha (1);  
    y.empilha (d1);  
    x.empilha (3);  
    y.empilha (d2);  
    while (!x.vazia() || !y.vazia()) {  
        x.desempilha();  
        y.desempilha();  
    }  
}
```

Paramétrico em C++

- Implementação de template em C++
 - Só possibilita a reutilização de código fonte
 - Não é possível compilar o código usuário separadamente do código de implementação
 - O compilador C++ necessita saber quais tipos serão associados ao template
 - Faz varredura do código usuário
 - replica todo o código de implementação para cada tipo utilizado
 - Compilador necessita verificar se o tipo usado é compatível com as operações definidas nas implementações e saber o tamanho a ser alocado

Paramétrico



- ADA
 - Pacotes Genéricos
- JAVA
 - Generics
- Python
 - Totalmente polimórfico paramétrico
 - Valores carregam seus parâmetros tipo

Inclusão

- Característico de linguagens orientadas a objetos
- Uso de hierarquia de tipos para criação de subprogramas e estruturas de dados polimórficas
- Idéia Fundamental
 - Elementos dos subtipos são também elementos do supertipo
 - Abstrações formadas a partir do supertipo podem também envolver elementos dos seus subtipos

Inclusão

- S subtipo de T implica
 - S formado por um sub-conjunto dos valores de T
 - Todo valor de S é também um valor de T
 - As operações associadas ao tipo T são aplicáveis ao subtipo S
 - S herda todas as operações do tipo T
- Conceito de tipo implementado através de classes em linguagens orientadas a

Herança

- Subclasses herdam os atributos e métodos de uma classe e, portanto, implementam subtipos do tipo definido por essa classe
- Herança associa à subclasse
 - uma representação inicial para os objetos dessa classe (os atributos herdados)
 - um conjunto inicial de métodos aplicáveis aos objetos dessa classe (os métodos herdados)
- Subclasse pode conter atributos e métodos adicionais, especializando o estado e o comportamento dos objetos da subclasse

Herança em JAVA

```
public class Pessoa {  
    private String nome;  
    private int idade;  
    public Pessoa (String n, int i) {  
        nome = n;  
        idade = i;  
    }  
    public void aumentarIdade () {  
        idade++;  
    }  
}
```

Herança em JAVA

```
public class Empregado extends Pessoa {  
    private float salario;  
    public Empregado (String n, int i, float s) {  
        super(n, i);  
        salario = s;  
    }  
    public void mudarSalario (float s) {  
        salario = s;  
    }  
}
```

Herança em JAVA

```
public class Empresa {  
    public static void main(String[] args) {  
        Pessoa p = new Pessoa ("Denise", 34);  
        p.aumentarIdade();  
        Empregado e1 = new Empregado ("Rogerio", 28, 1000.00);  
        e1.mudarSalario(2000.00);  
        e1.aumentarIdade();  
    }  
}
```

Vantagens da Herança



- Aumenta a reusabilidade do código
 - Desnecessário redefinir os atributos e métodos da classe Pessoa na classe Empregado
- Herança é polimorfismo universal
 - o método aumentarIdade(), usado para mudar a idade de uma Pessoa, e também aplicado para mudar a idade de um Empregado

Especificador de Acesso para Classes Herdeiras

- Algumas situações requerem que classes herdeiras tenham acesso livre aos atributos da classe herdada
- A alternativa de fazer esses atributos públicos ou criar métodos públicos de acesso pode não ser satisfatória porque torna esses atributos e métodos acessíveis para métodos de qualquer outra classe
- `protected`: novo especificador de acesso para classes herdeiras

Especificador de Acesso para Classes Herdeiras em JAVA

```
public class Pessoa {  
    protected int idade;  
}  
  
public class Empregado extends Pessoa {  
    public Empregado (int i) { idade = i; }  
    public boolean aposentavel() {  
        if (idade >= 65) return true;  
        return false;  
    }  
}
```

Especificador de Acesso para Classes Herdeiras em JAVA

```
public class Empresa {  
    public static void main(String[] args) {  
        Empregado e = new Empregado (32);  
        if (e.aposentavel()) System.out.println("Chega de trabalho!");  
        // e.idade = 70;  
    }  
}
```


Inicialização de Atributos com Herança

- Necessário inicializar atributos da superclasse antes dos da classe
- Em JAVA

```
class Estado {  
    Estado(String s) {  
        System.out.println(s);  
    }  
}  
  
class Pessoa {  
    Estado p = new Estado("Ativo");  
    Pessoa () {  
        System.out.println("Pessoa");  
    }  
}
```

Inicialização de Atributos com Herança

```
class Idoso extends Pessoa {
    Estado i = new Estado("Sabio");
    Idoso () {
        System.out.println("Idoso ");
    }
}

class Avo extends Idoso {
    Estado a1 = new Estado ("Alegre");
    Avo() {
        System.out.println("Avo");
        a3 = new Estado ("Orgulhoso");
    }
}
```

```
Estado a2 = new Estado ("Amigo");
```

Inicialização de Atributos com Herança

```
void fim() {  
    System.out.println("Fim");  
}  
Estado a3 = new Estado ("Satisfeito");  
}  
public class Inicializacao {  
    public static void main(String[] args) {  
        Avo a = new Avo();  
        a.fim();  
    }  
}
```

Sobrescrição

- Método herdado não é adequado para realizar a mesma operação nos objetos das subclasses

```
class XY {  
    protected int x = 3, y = 5;  
    public int soma () {  
        return x + y;  
    }  
}
```

```
class XYZ extends XY {  
    int z = 17;  
    public int soma () {  
        return x + y + z;  
    }  
}
```

Sobrescrição

```
public class Sobrescrita {  
    public static void main (String[] args) {  
        XYZ xyz = new XYZ();  
        System.out.println(xyz.soma());  
    }  
}
```

- Extensão de método na sobrescrição

```
class XYZ extends XY {  
    int z = 17;  
    public int soma () {  
        return super.soma() + z;  
    }  
}
```

Identificação Dinâmica de Tipos



- Forma de identificar o tipo do objeto em tempo de execução
 - útil em situações nas quais o programador desconhece o tipo verdadeiro de um objeto
 - permite elaboração de trechos de código nos quais métodos invocados por um mesmo referenciador de objetos se comportam de maneira diferenciada

Ampliação ou Upcast

- Instância
 - Objetos de uma classe
- Membro
 - Todas as instâncias da classe e de suas subclasses
- É permitido atribuir qualquer membro a uma referência à classe
- Ampliação
 - Termo usado para descrever o movimento de objetos na sua linha de ancestrais no sentido da

Ampliação em JAVA

```
public class Empresa {
    public void paga (Pessoa pes) {}
    public void contrata (Empregado emp) {}
    public static void main(String[] args) {
        Pessoa p = new Pessoa ("Lucas", 30);
        Empregado e = new Empregado ("Luis", 23, 1500.00);
        p = e;
        // e = p;
        Empresa c = new Empresa();
        c.paga(e);
        // c.contrata(p);
    }
}
```


Ampliação em C++

- Só através de ponteiros ou referências

```
Pessoa p, *q;
```

```
Empregado e, *r;
```

```
q = r;
```

```
// r = q;
```

```
// p = e;
```

```
// e = p;
```

- Limitação é consequência do mecanismo de cópia de objetos utilizado pela operação de atribuição e para passagem de parâmetros por valor

Amarração Tardia de Tipos

- Definição dinâmica do método a ser executado
 - Depende do objeto que invoca o método

```
class Pessoa {  
    String nome;  
    int idade;  
    public Pessoa (String n, int i) {  
        nome = n;  
        idade = i;  
    }  
    public void aumentaIdade () { idade++; }  
    public void imprime(){  
        System.out.print(nome + " , " + idade);  
    }  
}
```

Amarração Tardia de Tipos

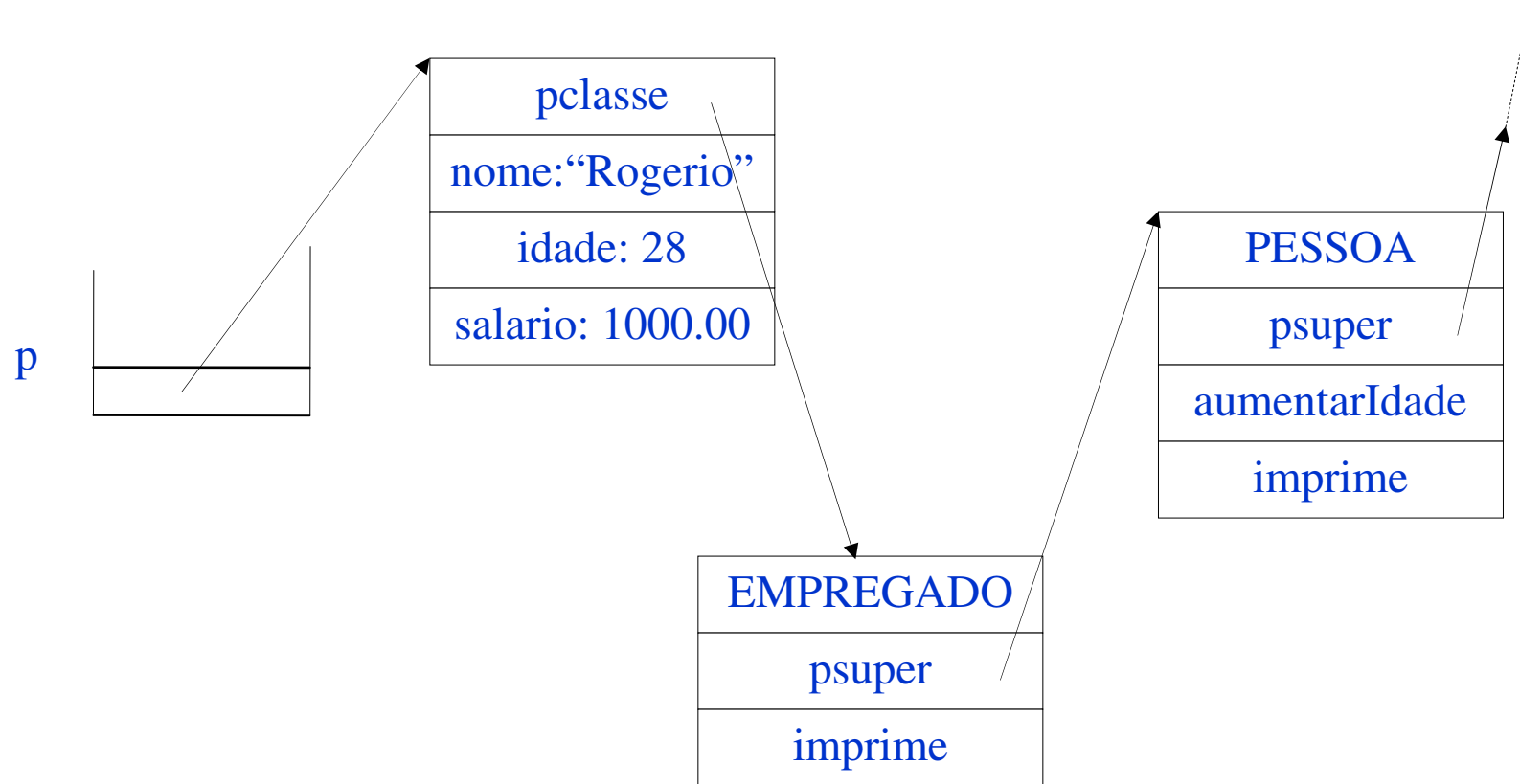
```
class Empregado extends Pessoa {  
    float salario;  
    Empregado (String n, int i, float s) {  
        super(n, i);  
        salario = s;  
    }  
    public void imprime(){  
        super.imprime();  
        System.out.print(" , " + salario);  
    }  
}
```

Amarração Tardia de Tipos

```
public class Empresa {  
    public static void main(String[] args) {  
        Pessoa p = new Empregado ("Rogerio", 28, 1000.00);  
        p.aumentaIdade();  
        p.imprime();  
    }  
}
```

⌘ Menos eficiente que amarração estática

Amarração Tardia de Tipos



Amarração Tardia x Amarração Estática em C++

- Todas chamadas utilizam amarração tardia em JAVA
- Em C++ o implementador da classe pode decidir se deseja o uso de amarração tardia ou não para cada método da classe
 - Visa não comprometer a eficiência de execução desnecessariamente
 - Uso da palavra virtual

```
class Pessoa {  
public:  
    void ler(){}  
    virtual void imprimir() {}
```

Amarração Tardia de Tipos

- Possibilita a criação de código usuário com polimorfismo universal

```
class Militar {  
    void operacao(){}  
}  
class Exercito extends Militar {  
    void operacao(){System.out.println("Marchar");}  
}  
class Marinha extends Militar {  
    void operacao(){System.out.println("Navegar");}  
}  
class Aeronautica extends Militar {  
    void operacao(){System.out.println("Voar");}
```

Amarração Tardia de Tipos

```
public class Treinamento {
    public static void treinar(Militar[] m) {
        for (int i = 0; i < m.length; i++) {
            m[i].operacao();
        }
    }
    public static void main (String[] args) {
        Militar[] m = new Militar[] {
            new Exercito(), new Marinha(), new Aeronautica(),
            new Militar()
        }
        treinar(m);
    }
}
Polimorfismo
```


Classes Abstratas

- Possuem membros, mas não possuem instâncias
 - Membros são as instâncias de suas subclasses concretas (não abstratas)
 - Proibida a criação de instâncias dessas classes
- Deve ser necessariamente estendida
- Úteis quando uma classe, ancestral comum para um conjunto de classes, se torna tão geral a ponto de não ser possível ou razoável ter instâncias dessa classe

Classes Abstratas

- Em JAVA

- Uso de especificador `abstract`

```
abstract class Militar {  
    void operacao(){}  
}
```

- Provocaria erro de compilação no exemplo apresentado anteriormente
 - Na construção de Militar

Métodos Abstratos

- Métodos declarados na classe, mas não implementados
 - A implementação desses métodos é deixada para as subclasses
- Classes abstratas normalmente possuem um ou mais métodos abstratos
- Se uma classe possui um método abstrato, ela deve necessariamente ser uma classe abstrata
- Especificam protocolo entre a classe e suas subclasses

Classes e Métodos Abstratos

- Em JAVA

```
abstract class Militar {  
    String patente;  
    Militar(String p) { patente = p; }  
    String toString() { return patente; }  
    abstract void operacao();  
}
```

Classes e Métodos Abstratos

- Em C++

```
class Militar {  
public:  
    virtual void operacao()=0;  
    void imprime { cout << "Militar"; }  
};
```

Especificação por Classe Abstrata

```
abstract class Forma {
    abstract void mover(int dx, int dy);
    abstract void desenhar();
}
class Circulo extends Forma {
    int x, y, raio;
    Circulo (int x, int y, int r) {
        this.x = x;
        this.y = y;
        raio = r;
    }
}
```

Especificação por Classe Abstrata

```
void mover(int dx, int dy) {
    x += dx;
    y += dy;
}
void desenhar() {
    System.out.println ("Circulo:");
    System.out.println ("  Origem: (" + x + ", " + y +")");
    System.out.println ("  Raio: " + raio);
}
}
```

Classes Abstratas Puras

- Todos os métodos são abstratos
- Usadas para disciplinar a construção de classes
- **JAVA** define o conceito de interface para a sua implementação
 - Não possuem atributos de instância
 - Métodos são todos públicos e abstratos

```
interface Forma {  
    void mover(int dx, int dy);  
    void desenhar();  
}
```


Estreitamento ou Downcast

- Termo usado para descrever a conversão de tipos de objetos no sentido da superclasse para as subclasses
- Não é completamente seguro para o sistema de tipos porque um membro da superclasse não é necessariamente do mesmo tipo da subclasse para a qual se faz a conversão
 - Algumas LPs não permitem o estreitamento
 - Outras exigem que ele seja feito através de uma operação de conversão explícita

Estreitamento ou Downcast

- JAVA somente permite a realização de estreitamento através de conversão explícita
 - Caso a conversão seja feita entre classes não pertencentes a uma mesma linha de descendência na hierarquia, ocorrerá erro de compilação
 - Caso a conversão seja na mesma linha de descendência, mas o objeto designado pela superclasse não seja membro da classe para o qual se faz o estreitamento, ocorrerá uma exceção em tempo de execução
 - O operador `instanceof` permite testar dinamicamente se o objeto designado pela superclasse realmente é da classe para a qual se deseja fazer a conversão

Estreitamento em JAVA

```
class UmaClasse {}
class UmaSubclasse extends UmaClasse {}
class OutraSubclasse extends UmaClasse{}
public class Estreitamento {
    public static void main (String[] args) {
        UmaClasse uc = new UmaSubclasse();
        UmaSubclasse us = (UmaSubclasse) uc;
        OutraSubclasse os;
        // os = (OutraSubclasse) us;
        // os = (OutraSubclasse) uc;
        if (uc instanceof OutraSubclasse) os = (OutraSubclasse) uc;
    }
}
```

Estreitamento em C++

- Conversão explícita tradicional
 - Feita em tempo de compilação sem qualquer verificação
 - `cast`
- Conversão explícita estática
 - Feita em tempo de compilação
 - Verifica se conversão ocorre em uma linha de descendência

Estreitamento em C++

- Conversão explícita dinâmica
 - Feita em tempo de execução
 - Verifica se conversão é para o tipo correto
 - `dynamic_cast`
- Teste dinâmico de tipos
 - Feito em tempo de execução
 - Verifica qual o tipo referenciado por ponteiro

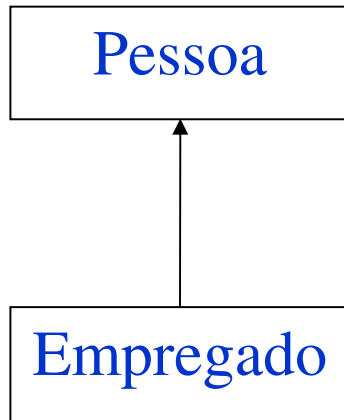
Estreitamento em C++

```
#include <typeinfo>
class UmaClasse {
public:
    virtual void temVirtual () {}
};
class UmaSubclasse: public UmaClasse {};
class OutraSubclasse: public UmaClasse {};
class OutraClasse {};
main () {
    // primeira parte do exemplo
    UmaClasse* pc = new UmaSubclasse;
    OutraSubclasse* pos = dynamic_cast <OutraSubclasse*> (pc);
    UmaSubclasse* ps = dynamic_cast <UmaSubclasse*> (pc);
```

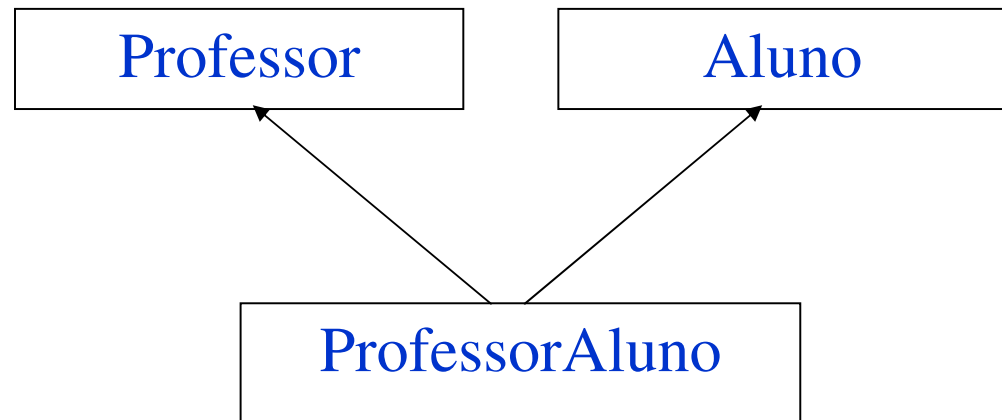
Estreitamento em C++

```
// segunda parte do exemplo
UmaSubclasse us;
pc = static_cast <UmaClasse*> (&us);
pc = &us;
OutraClasse* poc = (OutraClasse*) pc;
// OutraClasse* poc = static_cast <OutraClasse*> (pc);
// terceira parte do exemplo
if (typeid(pc) == typeid(ps))
    ps = static_cast<UmaSubclasse*>(pc);
if (typeid(pc) == typeid(pos))
    pos = static_cast<OutraSubclasse*>(pc);
```

Herança Múltipla



Herança Simples



Herança Múltipla

Herança Múltipla

- LPs com herança simples
 - Composição
 - Uso de métodos envoltórios
 - Impede subtipagem múltipla

```
class Professor {  
    String n = "Marcos";  
    int matr = 53023;  
    public String nome() { return n; }  
    public int matricula() { return matr; }  
}
```

```
class Aluno {
```

```
    String n = "Marcos";  
    int matr = 127890023;
```

Herança Múltipla

```
float coef = 8.3;
public String nome() { return n; }
public int matricula() { return matr; }
public float coeficiente() { return coef; }
}
class ProfessorAluno extends Professor {
    Aluno aluno = new Aluno();
    public float coeficiente() {
        return aluno.coeficiente();
    }
    public int matriculaAluno() {
        return aluno.matricula();
    }
}
```

Herança Múltipla

- LPs com herança múltipla
 - Resolvem problema de subtipagem múltipla
 - Não precisam de métodos envoltórios
 - Problemas
 - Conflito de Nomes
 - Herança Repetida

- Em C++

```
class Aluno {  
    float nota;  
public:
```

```
    void imprime();  
};
```

Polimorfismo

Conflito de Nomes



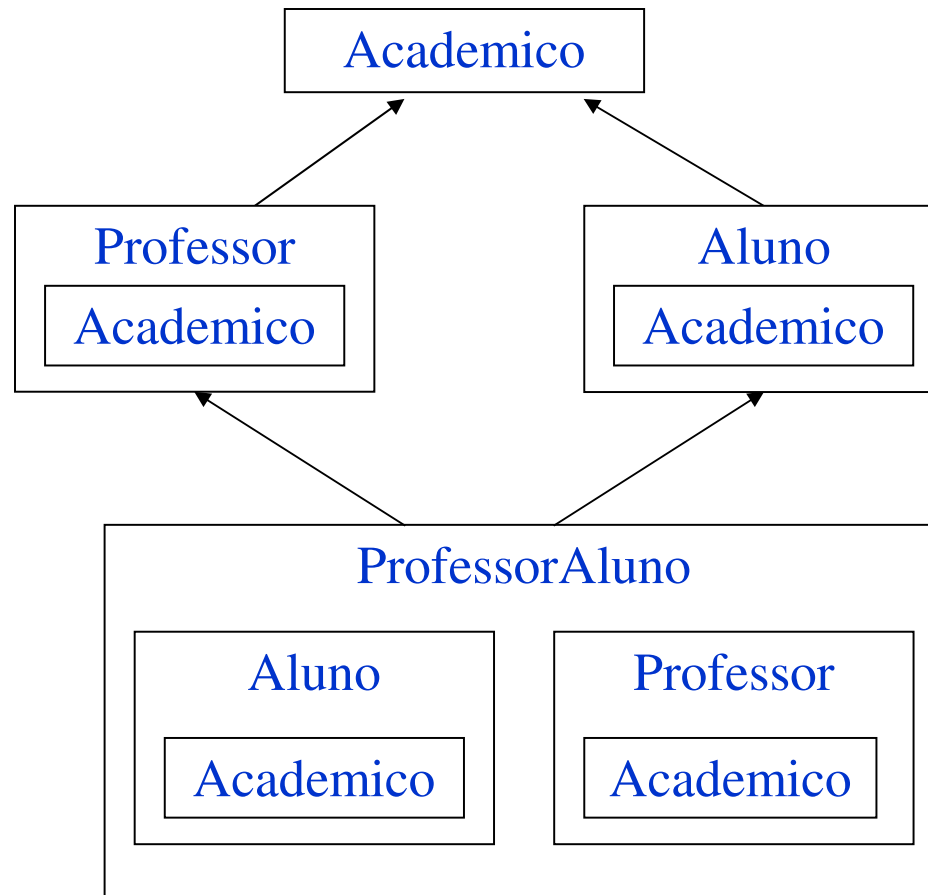
```
class Professor {
    float salario;
public:
    void imprime();
};
class ProfessorAluno: public Professor, public Aluno { };
main() {
    ProfessorAluno indeciso;
    // indeciso.imprime();
}
```

Conflito de Nomes

- Sobrescrição resolve

```
class ProfessorAluno: public Professor, public Aluno {
public:
    void imprime();
};
void ProfessorAluno::imprime() {
    Aluno::imprime();
}
main() {
    ProfessorAluno indeciso;
    indeciso.imprime();
}
```

Herança Repetida



Herança Repetida

- Duplicação desnecessária de atributos
- Conflito de Nomes
- Uso de virtual em C++

```
class Academico {  
    int i;  
    int m;  
public:  
    int idade ( ) { return i; }  
    int matricula ( ) { return m; }  
};
```

Herança Repetida

```
class Professor: virtual public Academico {
    float s;
public:
    float salario ( ) { return s; }
};
class Aluno: virtual public Academico {
    float coef;
public:
    float coeficiente ( ) { return coef; }
};
class ProfessorAluno: public Professor, public Aluno {};
```


Herança Múltipla em JAVA

- Não permite herança múltipla de classes
- Usa o conceito de interfaces para permitir subtipagem múltipla

```
interface Aluno {  
    void estuda();  
    void estagia();  
}  
  
class Graduando implements Aluno {  
    public void estuda() {}  
    public void estagia() {}  
}
```

Herança Múltipla em JAVA

```
interface Cirurgiao { void opera(); }
interface Neurologista { void consulta(); }
class Medico { public void consulta() {} }
class NeuroCirurgiao extends Medico implements Cirurgiao,
                                                    Neurologista {
    public void opera() { }
}
public class Hospital {
    static void plantãoCirurgico (Cirurgiao x) { x.opera(); }
    static void atendimentoGeral (Medico x) { x.consulta(); }
    static void neuroAtendimento (Neurologista x) { x.consulta(); }
    static void neuroCirurgico (NeuroCirurgiao x) { x.opera(); }
```

Herança Múltipla em JAVA

```
public static void main(String[ ] args) {  
    NeuroCirurgiao doutor = new NeuroCirurgiao();  
    plantãoCirurgico(doutor);  
    atendimentoGeral(doutor);  
    neuroAtendimento(doutor);  
    neuroCirurgia(doutor);  
}  
}
```

- Não existem problemas de conflito de nomes e herança repetida
- Dificulta a reutilização de código pois limita a

Composição X Herança

- **Composição**
 - Quando se quer as características de uma classe, mas não sua interface
 - Objeto é utilizado para implementar a funcionalidade da nova classe
 - Relacionamento do tipo **tem-um**
- **Herança**
 - Além de usar as características, a classe herdeira também usa a interface da classe herdada
 - Relacionamento do tipo **é-um**

Estruturas de Dados Genéricas

- Polimorfismo Paramétrico
 - C++
 - Compilador garante homogeneidade dos elementos
- Polimorfismo por Inclusão
 - JAVA
 - Uso de Object
 - Simplifica a LP
 - Necessidade de Estreitamento