ACH 2003 - Computação Orientada a Objetos

Introdução

Prof. Flávio Luiz Coutinho
flcoutinho@usp.br

Motivação

Problema: escrever um programa que leia uma sequência de valores inteiros, e os imprima na ordem inversa à ordem de leitura.

Motivação

Problema: escrever um programa que leia uma sequência de valores inteiros, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema!

Motivação

Problema: escrever um programa que leia uma sequência de valores inteiros, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema! Para realizar a inversão, basta inserir, um a um, os valores lidos na pilha e, em seguida, removê-los um a um.

Motivação

Problema: escrever um programa que leia uma sequência de valores inteiros, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema! Para realizar a inversão, basta inserir, um a um, os valores lidos na pilha e, em seguida, removê-los um a um.

Apesar de termos clareza de como resolver o problema, e qual estrutura de dados é a mais adequada para isso, ainda assim são possíveis escrever várias versões de um programa que resolva este problema, com graus distintos de qualidade (e aqui não estamos falando sobre resolver ou não o problema, ou se é eficiente ou não, mas sim se o programa é bem escrito).

Motivação

Problema: escrever um programa que leia uma sequência de valores inteiros, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema! Para realizar a inversão, basta inserir, um a um, os valores lidos na pilha e, em seguida, removê-los um a um.

Apesar de termos clareza de como resolver o problema, e qual estrutura de dados é a mais adequada para isso, ainda assim são possíveis escrever várias versões de um programa que resolva este problema, com graus distintos de qualidade (e aqui não estamos falando sobre resolver ou não o problema, ou se é eficiente ou não, mas sim se o programa é bem escrito).

Implementações em C (uma linguagem que não possui recursos OO)...

Considerações sobre as implementações em C

Versão 1: ausência de separação do código “usuário” da pilha, da implementação da estrutura em si. Duas responsabilidades misturadas na função main.

Versão 2: código responsável pelas operações da pilha separado em funções. Há alguma separação de responsabilidade. Mas o conhecimento de que as variáveis “free” e “values” representam juntas o estado de uma única estrutura do tipo pilha ainda é papel do “usuário”.

Versão 3: criação de uma *struct* para representar o tipo Pilha. Bem melhor que as versões anteriores, diminui ainda mais o conhecimento que o “usuário” tem que ter a respeito do funcionamento da pilha. Mas o estado interno da pilha está exposto para quem a manipula. Tal exposição é de fato necessária? Daria para evitar?

Considerações sobre as implementações

O uso de orientação a objetos permite dar um passo além ao:

- Vincular o tipo de dado (struct) às suas próprias operações (funções).
- Esconder do “usuário” o estado interno da estrutura.
- Expor apenas o essencial.
- Favorecer a uma melhor modularização/separação do código (isso não é exclusividade de linguagens OO, mas elas acabam favorecendo isso).

- Conceitos centrais: classe e objeto.

Classe e objeto (ou instância)

Classe:

- Define de um tipo. Composto por dados (atributos) e as operações (métodos) associadas. Uma classe é análoga a um molde.

Objeto (ou instância):

- Um representante concreto de uma certa classe (criado a partir do molde).
- Cada instância possui seu conjunto particular de atributos, ou seja, cada instância possui seu próprio contexto.
- Embora a definição de um método de uma dada classe seja única (ou seja, todas as instâncias daquela classe executam o mesmo código ao invocar o método), o escopo (ou seja, o que é “visível”) de cada execução do método inclui apenas os atributos da instância que efetivamente faz a chamada (além dos parâmetros recebidos, e eventuais variáveis locais).

Linguagens procedurais x orientadas a objetos

Em uma linguagem procedural, como o C, a unidade básica de programação é a função. Um programa procedural realiza uma tarefa através da execução coordenada de diversas funções.

Em uma linguagem OO (Java), a unidade básica de programação é a classe. Um programa OO executa uma tarefa através da criação de objetos (instâncias) de tipos variados que interagem entre si (através das chamadas de métodos).

Esboço da classe Stack

```
classe Stack {


    int values[100];

    int free;


    void init() { ... }

    void push(int value) { ... }

    int pop() { ... }

    int empty() { ... }

}
```

Esboço da classe Stack

```
classe Stack {

    int values[100];              // atributos que toda pilha precisa ter.
    int free;

    void init() { ... }
    void push(int value) { ... }
    int pop() { ... }
    int empty() { ... }
}
```

Esboço da classe Stack

```
classe Stack {

    int values[100];                // atributos que toda pilha precisa ter.
    int free;

    void init() { ... }             // conjunto de operações (comportamentos)
    void push(int value) { ... }    // que poderão ser chamados em cada objeto
    int pop() { ... }               // do tipo Pilha.
    int empty() { ... }
}
```

Esboço do novo programa principal

```
main() {

    Stack stack;                // criação de uma instância do tipo Pilha
    int value;

    stack.init();               // chamada de método que prepara a pilha
    while( ... ) {
        stack.push(value);      // chamada de método que guarda um valor
    }
    ...
```

Esboço do novo programa principal

```
    ...


    while( !stack.empty() ) {     // chamada que verifica pilha vazia
        value = stack.pop();      // chamada que remove um valor do topo
        // imprime value
    }


} // fim main
```

Esboço do novo programa principal

```
    ...


    while( !stack.empty() ) {      // chamada que verifica pilha vazia
        value = stack.pop();       // chamada que remove um valor do topo
        // imprime value
    }


} // fim main
```

Observem que isso é apenas um esboço de como o código referente ao exemplo da pilha deverá se parecer quando implementarmos a próxima versão usando o Java. Diversos detalhes, em especial em relação à instanciação do objeto, foram omitidos.

Antes de traduzir o esboço para a linguagem Java…

Vamos falar um pouco sobre a linguagem:

- Linguagem OO criada em 1995
- Virtual machine: programas java compilados para bytecode
- *Write once, run everywhere*
- Sintaxe parecida com C/C++ (mas com menos recursos de baixo nível)
- Extensa biblioteca de classes
- Garbage collector (facilita o gerenciamento de memória)

Hello World em Java

```
class HelloWorld {


    public static void main(String [] args){


        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

```
class HelloWorld {

    public static void main(String [] args){

        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

```
class HelloWorld {


    public static void main(String [] args){


        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

```
class HelloWorld {

    public static void main(String [] args){

        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

```
class HelloWorld {


    public static void main(String [] args){


        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

```
class HelloWorld {


    public static void main(String [] args){


        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

```
class HelloWorld {


    public static void main(String [] args){


        System.out.println("Hello, World!");
    }
}
```

Hello World em Java

Para compilar:

```
> javac HelloWord.java
```

E para rodar:

```
> java HelloWorld
```

Hello World em Java

Dá pra deixar o código mais enxuto?

- remover declaração da classe
- remover “`String [] args`”
- remover “`public static`”
- remover apenas “`static`”
- remover apenas “`public`”

O que acontece em cada cenário?

Antes de retomar o exemplo da Pilha, um outro exemplo…

Classe Pessoa:

- Criando a classe (atributos apenas).

- Programa de teste (usando Pessoa como se fosse uma *struct* do C).
  - Como instanciar (criar) objetos do tipo Pessoa?
  - Como iniciar as instâncias de Pessoa?

Antes de retomar o exemplo da Pilha, um outro exemplo…

Classe Pessoa:

- Adicionando construtor na classe Pessoa.
  - Como o construtor funciona?
  - Por que é melhor usar o construtor do que iniciar os atributos manualmente?

- Adicionando métodos na classe Pessoa.

- Modificando programa de teste.

Antes de retomar o exemplo da Pilha, um outro exemplo…

Classe Pessoa:

- Protegendo atributos na classe Pessoa.
  - getters/setters.

Retomando o exemplo envolvendo o uso da pilha

Versão 4 (escrita em Java): conceito de classe aplicado de forma melhor, mas ainda com alguns detalhes em aberto. Uma classe define um tipo (atributos) e o conjunto de operações associados ao tipo (métodos). Melhor organização do código, mas estrutura interna da pilha ainda exposta para seus “usuários”.

Versão 4b (escrita em Java): uso dos modificadores de acesso, protegendo o estado interno, e expondo apenas o realmente essencial para os “usuários” da classe Stack. Reforça a razão de os comportamentos serem declarados na própria classe (seria impossível restringir o estado interno da pilha se as operações fossem implementadas separadas da classe).

Retomando o exemplo envolvendo o uso da pilha

Versão 4 (escrita em Java): conceito de classe aplicado de forma melhor, mas ainda com alguns detalhes em aberto. Uma classe define um tipo (atributos) e o conjunto de operações associados ao tipo (métodos). Melhor organização do código, mas estrutura interna da pilha ainda exposta para seus “usuários”.

Versão 4b (escrita em Java): uso dos modificadores de acesso, protegendo o estado interno, e expondo apenas o realmente essencial para os “usuários” da classe Stack. Reforça a razão de os comportamentos serem declarados na própria classe (seria impossível restringir o estado interno da pilha se as operações fossem implementadas separadas da classe).

Aproveitando o exemplo para falar um pouquinho sobre a classe Scanner…

Conceito importante: encapsulamento

O conceito de encapsulamento está ligado à ideia de ocultar o estado interno de um objeto perante o mundo exterior (código “usuário”), expondo apenas o mínimo necessário para que os “usuários” possam usufruir de suas funcionalidades.

No exemplo da pilha, queremos que seus “usuários” possam criar uma pilha vazia, guardar coisas no topo dela, remover coisas do topo, e verificar se está vazia ou não. Ou seja, que saibam apenas o que a pilha pode fazer.

Mas não queremos que os usuários precisem saber como a pilha implementa sua funcionalidade. E nem que tenham acesso ao seu estado interno.

O encapsulamento adequado é obtido com o bom uso dos modificadores de acesso (`public`, `protected`, `<default>`, `private`), para se restringir o que não deve ser visto/usado, e permitir acesso ao que pode ser usado.

Conceito importante: encapsulamento

“*Ah, mas se o ‘usuário’ sabe como é implementado, apesar de não ser necessário, isso não seria um bônus?*”

- É um bônus saber como é implementado para embasar o bom uso.
- Mas ainda assim o acesso ao estado interno deve ser proibido.
- Exemplo:

    Código “usuário” itera pelos valores armazenados na pilha através do acesso direto ao vetor `values`. Em um momento posterior, a implementação da pilha muda. Ao invés de vetor como estrutura de armazenamento, passe-se a usar uma lista ligada. O código “usuário” que “tirou vantagem” do conhecimento e visibilidade do estado interno da pilha não funciona mais. Uma alteração de um ponto do código afeta o funcionamento de outro!

    Sem falar no risco adicional de corromper o estado da pilha.

Tipos primitivos e objetos

No Java, um atributo ou variável pode ser de dois tipos:

- Tipos primitivos: tipos mais elementares de dados (boolean, byte, char, int, long, float, double). Equivalência razoável com tipos de dados nativos do processador, e também da linguagem C.

- Objetos: tudo aquilo que é instância de uma classe. Strings e vetores (*arrays*) de tipos primitivos também são objetos. Vetores possuem atributos e são alocados com o *new*. A alocação de Strings pode ser implícita, mas possuem diversos métodos.

Tipos primitivos e objetos

Por que é importante saber a diferença?

- Variáveis e atributos cujo tipo é uma classe, são na verdade são ponteiros para as instâncias alocadas. No Java estes ponteiros são chamados de referências.

- Saber esta diferença é especialmente importante para entender o que acontece na passagem de parâmetros a métodos:

    - Tipos primitivos são passados como cópia (parâmetro recebido é uma cópia do valor passado na chamada).

    - Objetos são passados como referência (na realidade, o parâmetro recebido também é uma cópia do valor passado na chamada. Mas como o valor representa um endereço, na prática o objeto que o método manipula é o mesmo que foi passado para chamada).

Instanciação

O que acontece quando se executa a linha: `Stack stack = new Stack()?`

- Alocação de memória suficiente para representar um objeto do tipo pilha. De forma simplificada, memória suficiente para guardar uma referência (atributo `values`) e um valor inteiro (atributo `free`).
- Após a alocação, o código do construtor é executado para preparar o objeto recém criado.
- Ao final, da execução do construtor, o endereço de memória do objeto, já alocado e iniciado, é devolvido e atribuído à variável local `stack`.

Instanciação

O que acontece quando se executa a linha: `Stack stack = new Stack()`?

- Alocação de memória suficiente para representar um objeto do tipo pilha. De forma simplificada, memória suficiente para guardar uma referência (atributo `values`) e um valor inteiro (atributo `free`).
- Após a alocação, o código do construtor é executado para preparar o objeto recém criado.
- Ao final, da execução do construtor, o endereço de memória do objeto, já alocado e iniciado, é devolvido e atribuído à variável local `stack`.
- E quanto ao vetor (*array*)? O vetor é um objeto à parte. Ele é alocado separadamente no construtor da classe `Stack` (`new int[100]`) e sua referência guardada no atributo `values`.

Modificadores de acesso

`public`: atributos e métodos visíveis para código em qualquer classe.

`protected`: atributos e métodos visíveis para código das classes que pertencem ao mesmo pacote, e também para classes derivadas (mesmo que em pacotes distintos)

`<default>`: atributos e métodos visíveis para código das classes que pertencem ao mesmo pacote.

`private`: atributos e métodos visíveis apenas para o código da classe que os declara.

Modificador static

Por padrão, todo atributo declarado em uma classe vai existir para cada instância dela que venha a ser criada, e cada instância tem sua “própria cópia” do atributo.

Da mesma forma, os métodos declarados em uma classe enxergam além dos parâmetros recebidos e variáveis locais declaradas no método, o conjunto de atributos da instância que chama o método (independente dos modificadores de acesso associados).

Mas em certas situações (que não devem ser a regra) queremos guardar algum tipo de informação que não está vinculada a instâncias específicas, e que haja apenas “uma cópia” desta informação. Para estes casos, podemos usar o modificador `static` para declarar um atributo. Dizemos que este atributo é um atributo da classe (ao contrário do padrão, que são atributos de instâncias).

Modificador static

Do mesmo modo, para métodos que implementam alguma funcionalidade que independe da existência de um objeto sobre o qual irá atuar, podemos declarar métodos `static`.

Quando se aprende Java sem ter muita bagagem dos conceitos de OO, podemos ficar habituados a abusar do modificador `static`, programando em um estilo procedural/imperativo ao invés de OO (vejam o exemplo Ex3b.java).

Por isso, o uso do **`static`** não deve ser algo frequente em um sistema OO bem projetado.

Exemplos de uso razoável do static

- Contador de instâncias criadas durante a execução (classe Pessoa).

- Classe Math

Resumo

Conceitos OO:

- classe e objeto
- atributos e métodos
- modificadores de acesso e encapsulamento

Java:

- modificadores: public, protected, <default>, private
- modificador static
- tipo primitivo x objeto
- instanciação / construtores

Curiosidade…

Ano de lançamento de algumas linguagens de programação:

- Fortran 1957 # computação numérica e científica.
- LISP 1958 # Longas e Intermináveis Sequências de Parênteses.
- COBOL 1959 # rodando nos mainframes dos bancos até hoje!
- BASIC 1964 # o Python dos anos 70/80! ;)
- C 1972 # a linguagem referência para muitas outras!
- C++ 1985 # C orientado a objetos; perae, mas e o Objective-C?
- Perl 1988 # um pouco fora de moda, mas ainda interessante.
- Python 1991 # mais antigo do que o Java! :-O
- Java 1995 # C++ mais amigável e “sem ponteiros”!
- C# 2000 # o primo (ou seria irmão?) do Java.
- Kotlin 2011 # não é Java, mas roda na JVM.