FACAMP
FACULDADES DE CAMPINAS

Linguagens de Programação

Aula 11

Tipos de Dados Abstratos e Construções de Encapsulamento

2º semestre de 2019

Prof José Martins Junior

O conceito de abstração

- Uma **abstração** é uma visão ou representação de uma entidade que incluí apenas os atributos mais significativos
- A abstração é uma ferramenta poderosa contra a complexidade, o seu propósito é simplificar a programação
- Abstração de processos
  - Subprogramas
  - Exemplo: em um procedimento de ordenação o algoritmo usado fica oculto

```
sortInt(list, listLen);
```

- Abstração de dados

Introdução à abstração de dados

- Um tipo abstrato de dado satisfaz duas condições
  - Encapsulamento
    - A representação de seus elementos é ocultada de quem usa o tipo
    - As únicas operações diretas possíveis sobre os elementos são aquelas fornecidas na definição do tipo
    - Vantagens
      - Confiabilidade, os clientes não podem mudar a representação dos elementos diretamente, aumentando a integridade dos objetos
      - Evita colisões de nomes
      - Possibilidade de alterar a representação e implementação sem afetar os clientes
  - Interface do tipo
    - Deve conter a declaração do tipo e dos protocolos das operações sobre seus elementos em uma única unidade sintática
    - Sua implementação pode ser realizada em outra unidade sintática
    - Vantagem
      - Compilação separada

Exemplo de tipo: pilha

create(stack)	Cria e possivelmente inicializa um objeto de pilha
destroy(stack)	Libera o armazenamento para a pilha
empty(stack)	Um predicado ou função booleana que retorna verdadeiro (*true*) se a pilha especificada é vazia e falso (*false*) caso contrário.
push(stack, element)	Insere o elemento especificado na pilha especificada
pop(stack)	Remove o elemento do topo da pilha especificada
top(stack)	Retorna uma cópia do elemento do topo da pilha especificada

- Exemplo de uso por um cliente

```
...
create(stk1);
push(stk1, color1);
push(stk1, color2);
if (!empty(stk1))
   temp = top(stk1);
...
```

Questões de projeto

- Requisitos da linguagem para TAD
  - Uma unidade sintática que encapsula a definição do tipo e dos protótipos das operações
  - Uma maneira de tornar o nomes de tipos visíveis para clientes do código e ocultar a implementação
  - Poucas operações padrões (se alguma) deve ser fornecidas (além das fornecidas na definição do tipo): atribuição, comparação..
  - Qual é a forma do “invólucro” para a interface do tipo?
  - Os tipos abstratos podem ser parametrizados?
  - Quais mecanismos de controle de acesso são fornecidos e como eles são especificados?

Exemplos de linguagens: Ada

- A construção de encapsulamento é chamada de pacote
  - Pacote de especificação (define a interface do tipo)
  - Pacote de corpo (implementação)
- Ocultação de informação
  - O pacote de especificação tem uma parte visível ao cliente e uma parte oculta (private)
  - Na parte visível ao cliente é feito a declaração do tipo abstrato, que pode conter também a representação dos tipos não ocultos
  - Na parte privada é especificada a representação do tipo abstrato

Exemplos de linguagens: Ada

- Motivos para ter uma parte privada no pacote de especificação
  - O compilador precisa saber a representação vendo apenas o pacote de especificação
  - O clientes precisam enxergar o nome do tipo, mas não a representação
- Ter parte dos detalhes da implementação (a representação) no pacote de especificação não é bom
  - Uma solução é fazer todos os TADs serem ponteiros, mas esta solução tem problemas
  - Dificuldades com ponteiros
  - Comparação de objetos
  - O controle da alocação é perdido

Exemplo de tipo pilha em Ada

```
package Stack_Pack is
-- As entidades visíveis (interface)  type
   Stack_Type is limited private;
   Max_Size: constant := 100;
   function Empty(Stk : in Stack_Type) return Boolean;
   procedure Push(Stk : in out Stack_Type;
                  Element : in Integer);
   procedure Pop(Stk: in out Stack_Type);
   function Top(Stk: in Stack_Type) return Integer;
-- Parte que é oculta aos clientes
   private
      type List_Type is array (1..Max_Size) of Integer;
      type Stack_Type is record
        List: List_Type;
        Topsub: Integer range 0..Max_Size := 0;
      end record;
end Stack_Pack;
```

Exemplo de tipo pilha em Ada

```
package body Stack_Pack is
   function Empty(Stk : in Stack_Type) return Boolean is  begin
      return Stk.Topsub = 0;
      end Empty;
   ...
end Stack_Pack;
```

```
with Stack_Pack;
procedure Use_Stacks is  Topone : Integer;  Stack : Stack_Type;
   begin
   Push(Stack, 42);
   Push(Stack, 17);  Topone := Top(Stack);  Pop(Stack);
   ...
   end Use_Stacks;
```

Exemplos de linguagens: C++

- Linguagem criada para adicionar orientação a objetos em C
  - Portanto, suporta TADs
- Os mecanismos de encapsulamento são as classes e estruturas
  - Os dados são chamados de dados membros
  - As funções são chamadas de funções membros
  - Os membros podem ser da classe ou da instância
  - Todas as instâncias de uma classe compartilham uma cópia das funções membros
  - Cada instância da classe tem sua cópia dos dados membros
  - As instâncias podem ser estáticas, dinâmica na pilha ou dinâmicas no heap (new e delete)
  - Uma função membro pode ser inline (cabeçalho e corpo juntos)

Exemplos de linguagens: C++

- Ocultação de informação
  - `private`, para entidades ocultas
  - `public`, para entidades públicas
  - `protected`, para entidades visíveis apenas para as subclasses
- Construtores
  - Utilizados para inicializar uma instância da classe
- Destrutores
  - Chamado implicitamente quando o tempo de vida da instância acaba

Exemplo de tipo pilha em C++

```
class Stack {
  private:
    int *stackPtr;  int maxLen;  int topPtr;
  public:
    Stack() {     // construtor
      stackPtr = new int [100];
      maxLen = 99;
      topPtr = -1;
    };
    ~Stack () { // destrutor
      delete [] stackPtr;
    };
    void push (int num) {...};
    void pop () {...};
    int top () {...};
    int empty () {...};
}
```

Exemplo de tipo pilha em C++

```
void main() {
  int topOne;
  Stack stk;
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
  ...
}
```

Separando definição de implementação

```
// Stack.h - arquivo de cabeçalho para classe Stack
class Stack {
// Membros visíveis apenas para outros
// membros da classe
private:
   int *stackPtr;  int
   maxLen;  int topPtr;
// Membros visíveis para os clientes
public:
   Stack();    // Construtor
   ~Stack(); // Destrutor
   void push(int);  void
   pop();  int top();
   int empty();
}
```

Separando definição de implementação

```
// Stack.cpp - implementação para a classe Stack
#include <iostream>
#include "Stack.h"
using std::count;

Stack::Stack() {
   stackPtr = new int[100];
   maxLen = 99;
   topPtr = -1;
}
...
void Stack::push(int number) {
...
}
...
```

Exemplos de linguagens: Java

- Similar ao C++, exceto que
  - Todos os tipos definidos pelos usuários são classes
  - Todos os objetos são alocados no heap e acessados através de referência
  - Os métodos precisam ser definidos na classe
  - Modificadores de acesso são especificados em entidades, não em cláusulas
  - Não tem destrutor

Exemplo de tipo pilha em Java

```
class Stack {
    private int [] stackRef;
    private int maxLen;  private int topIndex;
    public Stack() { // a constructor
      stackRef = new int [100];
      maxLen = 99;
      topPtr = -1;
    };
    public void push(int num) {...};
    public void pop() {...};
    public int top() {...};
    public boolean empty() {...};
}
```

Exemplo de tipo pilha em Java

```
class TestStack {
  public static void main(String[] args) {
    Stack myStack = new Stack();
    myStack.push(42);
    myStack.push(29);
    ...
    myStack.pop();
    ...
  }
}
```

Exemplos de linguagens: C#

- Baseado em C++ e Java
- As instâncias de classe são dinâmicas no heap
- Destrutores são raramente usados
- Estruturas são semelhantes as classes, mas não podem ter herança, são alocadas na pilha e acessadas como valores
- Suporte a propriedades, que é uma maneira de implementar getters e setters sem requerer a chamada de método explícita

Exemplos de linguagens: Ruby

- O mecanismo de encapsulamento são as classes
  - Capacidade semelhante as classes em C++ eJava
  - Os nomes das variáveis de instâncias começam com @ e de classes com @@
  - Os métodos são declarados com as mesma sintaxe que as funções
  - O construtor é o initialize, que é chamado quando o método new da classe é chamado
  - As classes são dinâmicas
- Ocultação de informação
  - Os membros das classes podem ser públicos ou privados

Exemplo de tipo pilha em Ruby

```
class StackClass
  def initialize
    @stackRef = Array.new
    @maxLen = 100
    @topIndex = -1
  end
  def push(number)
    ...
  end
  def pop
    ...
  end
  def top
    ...
  end
  def empty
    ...
  end  end
```

Exemplo de tipo pilha em Ruby

```
myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{myStack.top}"
myStack.pop
```

Tipos abstratos de dados parametrizados

- Permite a criação de tipos abstratos de dados "tipados", que podem armazenar dados de algum tipo
- Não é uma questão relacionada as linguagens dinâmicas
- Algumas linguagens com suporte a TAD parametrizados
  - Ada, C++, Java 5, C# 2005

Tipos abstratos de dados parametrizados em Ada

```
generic
   Max_Size: Positive;
   type Element_Type is private;
package Generic_Stack is
   Type Stack_Type is limited private;
   procedure Push(Stk : in out Stack_Type;
                  Element : in Element_Type);
   ...
end Generic_Stack;

Package Integer_Stack is new Generic_Stack(100, Integer);
Package Float_Stack is new Generic_Stack(100, Float);
```

Tipos abstratos de dados parametrizados em C++

```
template <class Type>
class Stack {
  private:
    Type *stackPtr; int maxLen; int topPtr;
  public:
    Stack(int size) {
      stackPtr = new Type[size];
      maxLen = 99;
      topPtr = -1;
    };
    void push(Type value) {...};
    ...
}

Stack<int> s1;
Stack<float> s2;
```

Tipos abstratos de dados parametrizados em Java 5

- Antes da versão 5, as classes como `LinkedList` e `ArrayList` podiam armazenar qualquer objeto
  - Problemas com coleção de objetos
    - Todo objeto da coleção precisa da coerção quando é acessado
    - Não é possível fazer checagem de tipo quando os valores são adicionados
    - Não é possível inserir tipos primitivos nas coleções
  - O Java 5 tentou resolver estes problemas, adicionado genéricos (e autoboxing) a linguagem
    - As classes genéricas resolveram o primeiro e o segundo problema, mas não o terceiro, porque os parâmetros genéricos tem quer classe
    - O autoboxing resolveu o terceiro problema

Tipos abstratos de dados parametrizados em Java 5

```
class Stack<T> {
    private T[] stackRef;
    private int maxLen;
    private int topIndex;
    ...
    public void push(T value) {...};
    ...
}

Stack<Integer> s1 = new Stack<Integer>();
int x = 10;
s1.push(x);
int y = s1.top();

Stack<Float> s2 = new Stack<Float>();
```

Tipos abstratos de dados parametrizados em C# 2005

- Assim como Java, nas primeiras versão do C# as coleção armazenavam objetos de qualquer classe
- Classes genéricas foram adicionadas ao C# 2005.
- Diferente do Java, os elementos de coleções genéricas podem ser acessados através de índices

Construções de encapsulamento

- Programas maiores têm duas necessidades especiais
  - Alguma maneira de organização, além da simples divisão em subprogramas e tipos abstratos de dados
  - Alguma maneira de realizar compilação parcial
- Solução: Encapsulamento
  - Agrupar os códigos e dados logicamente relacionados em uma unidade que possa ser compilada separadamente
  - Existem várias formas de encapsulamento

Construções de encapsulamento

- Encapsulamento em C
  - Um ou mais subprogramas e tipos são colocados em arquivos que podem ser compilados independentemente
  - A interface é colocada em um arquivo de cabeçalho, e a implementação em outro arquivo
  - `#include` é utilizado para incluir o cabeçalho
  - O ligador não checa os tipos entre o cabeçalho e a implementação

Construções de encapsulamento

- Encapsulamento em C++
  - Permite definir arquivos de cabeçalho e implementação (semelhante ao C)
  - Os arquivos de cabeçalho de templates em geral incluem a declaração e a definição
  - `friend` fornece um mecanismo para permitir acesso a membros privados

Construções de encapsulamento

- Pacotes Ada
  - Podem incluir várias declarações de tipos e subprogramas
  - Podem ser compilados separadamente
  - Os pacotes de especificação e corpo podem ser compilados separadamente

Construções de encapsulamento

- Assemblies em C#
  - Uma coleção de arquivos que aparentam ser uma DLL ou executável
  - Cada arquivo define um módulo que pode ser compilado separadamente
  - Uma DLL é uma coleção de classes e subprogramas que são ligados individualmente a um executável
  - Contém outras informações, como dependências e versão
  - Podem ser privados ou públicos
  - O modificador de acesso internal especifica que um membro é visível a todos no mesmo assembly

Encapsulamento de nomes

- Serve para isolar espaços de nomes em projetos
  - Namespaces em C++
  - Pacotes em Java
  - Pacotes em Ada
  - Módulos em Ruby

Bibliografia

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