Tipos Abstratos de Dados e Encapsulamento

Paradigmas de Programação – BCC/UFRPE
Lucas Albertins – lucas.albertins@deinfo.ufrpe.br

Agenda

- O conceito de Abstração
- Vantagens
- Decisões de Projeto
- Exemplos
- Tipos de Dados Parametrizados
- Construtores de Encapsulamento
- Encapsulamentos de Nomes

O Conceito de Abstração

- Uma abstração é uma visão ou representação de uma entidade que inclui apenas seus atributos mais significativos
- Fundamental em programação (e ciência da computação)
  - Abstração de Processo -> Subprogramas
  - Abstração de Dados

O Conceito de Abstração

- Tipo Abstrato de Dados (TAD) é um tipo de dado definido pelo usuário que satisfaz duas condições:
  - A representação de objetos do tipo é escondida de unidades que o utilizam
  - As declarações do tipo e dos protocolos das operações sobre objetos deste tipo estão contidas numa única unidade sintática. Outras unidades podem criar variáveis do tipo definido
- Encapsulamento: invólucro (construtor) que agrupa dados e operações de objetos de um tipo abstrato de dados

Vantagens de Abstração de Dados

- Primeira condição
  - Confiabilidade – esconder representação, códigos usuários não podem acessar diretamente objetos do tipo permitindo que a representação seja alterada sem afetar o código do usuário.
  - Reduz preocupações com códigos e variáveis
  - Conflitos de nomes também são reduzidos
- Segunda Condição
  - Um método para organização de programas
  - Facilita alteração (tudo associado com o tipo está junto)
  - Compilação em separado

Requisitos de Linguagem para TAD

- Uma unidade sintática na qual se encapsula a definição do tipo
- Um método de criar nomes de tipo e cabeçalhos de subprogramas visíveis para clientes escondendo as definições
- Operações primitivas devem vir prontas no processador da linguagem

Decisões de Projeto

- Qual é a forma do contêiner para a interface do tipo?
- TAD podem ser parametrizados?
- Quais controles de acesso são fornecidos?
- A especificação do tipo é separada da sua implementação?

Exemplos: Ada

- Dispositivo de Encapsulamento: pacote (*package*)
  - Especificação do pacote (interface)
  - Corpo do Pacote (implementação)
- Ocultação de Informação (*Information Hiding*)
  - Especificação do pacote em duas partes, pública e privada
  - O nome do tipo abstrato aparece na parte pública da especificação.
  - A representação do tipo abstrato aparece na parte privada da especificação
    - O compilador deve ver a representação após ver o pacote de especificação
    - Clientes devem ver o nome do tipo mas não a representação

Exemplos: Ada

```
package Stack_Pack is
   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; elem: in
Integer);
   procedure pop(stk: in out stack_type);
   function top(stk: in stack_type) return Integer;

   private  -- hidden from clients
   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
```

Exemplos: Ada

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  function Empty(Stk : in Stack_Type) return Boolean is
    begin
    return Stk.Topsub = 0;
    end Empty;
  procedure Push(Stk: in out Stack_Type;
    Element : in Integer) is
    begin
    if Stk.Topsub >= Max_Size then
      Put_Line("ERROR - Stack overflow");
    else
      Stk.Topsub := Stk.Topsub + 1;
      Stk.List(Topsub) := Element;
    end if;
  end Push;
...
 end Stack_Pack;
```

Exemplos: C++

- Baseados no tipo **struct** de C e classes de Simula 67
- Dispositivo de encapsulamento: classes (tipo)
- Todas as instâncias de uma classe compartilham uma única cópia das funções membro e dos dados membro
- Instâncias podem ser estáticas, dinâmicas na pilha ou dinâmicas na heap
- Ocultamento de Informações (*Information Hiding)*:
  - cláusulas *private* para entidades escondidas
  - Cláusulas *public* para entidades de interface
  - Cláusulas *protected* para herança

Exemplos: C++

- Construtores:
  - Funções que inicializam os dados de uma instância
  - Podem alocar espaço de armazenamento se parte do objeto é dinâmico na pilha
  - Pode incluir parâmetros para parametrização de objetos
  - Nome é o mesmo da classe
- Destrutores:
  - Funções de limpeza após uma instância é destruída
  - Implicitamente chamada quando o tempo de vida do objeto acaba, mas também pode ser explicitamente invocado
  - Nome da classe precedido de um til (~)

Exemplos: C++

```
class Stack {
   private:
        int *stackPtr, maxLen, topPtr;
   public:
        Stack() { // a constructor
                stackPtr = new int [100];
                maxLen = 99;
                topPtr = -1;
        };
        ~Stack () {delete [] stackPtr;};
        void push (int number) {
           if (topSub == maxLen)
              cerr << "Error in push - stack is full\n";
           else stackPtr[++topSub] = number;
        };
        void pop () {…};
        int top () {…};
        int empty () {…};
```

Exemplos: C++ (header file)

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private: //** These members are visible only to other
//** members and friends
  int *stackPtr;
  int maxLen;
  int topPtr;
public: //** These members are visible to clients
  Stack(); //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```

Exemplos: C++ (código)

```
// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** A destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Exemplos: Java

- Similar a C++, exceto:
  - Todos os tipos definidos pelo usuário são classes
  - Todos os objetos são alocados na heap e acessados através de referência
  - Atributos individuais de classes podem ter modificadores de controle de acesso (*private*, *public*, *protected*) ao invés de cláusulas
  - Java tem um segundo mecanismo de escopo através de pacotes
    - Entidades em classes de um mesmo pacote que não possuem modificadores de controle de acesso são visíveis dentro do pacote

Exemplos: Java

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

Exemplos: C#

- Baseado em C++ e Java
- Tem dois novos modificadores, *internal* e *protected internal*
- Todas as instâncias são dinâmicas na heap
- Construtores default estão disponíveis para todas as classes
- Coleta de lixo é usada para a maioria dos objetos na heap, então destrutores são raramente utilizados
- **structs** são classes mais leves que não suportam herança
- Métodos de acesso a atributos (getters e setters) através da definição de propriedades

Exemplos: C#

```
public class Weather {
   public int DegreeDays { //** DegreeDays is a property
      get {return degreeDays;}
      set {
        if (value < 0 || value > 30)
          Console.WriteLine(
              "Value is out of range: {0}", value);
        else degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Exemplos: Ruby

- Dispositivo de Encapsulamento: Classe
- Atributos de instância começam com arroba (@)
- Atributos de classe começam com dois arrobas (@@)
- Métodos tem a sintaxe de funções (`def ... end`)
- Construtores são nomeados `initialize`s (um por classe)
- Membros de classe podem ser marcados privados ou públicos (default)

Exemplos: Ruby

```
class StackClass {
   def initialize
     @stackRef = Array.new
     @maxLen = 100
     @topIndex = -1
   end

   def push(number)
     if @topIndex == @maxLen
       puts "Error in push - stack is full"
     else
       @topIndex = @topIndex + 1
       @stackRef[@topIndex] = number
     end
   end
   def pop … end
   def top … end
   def empty … end
end
```

Tipos abstratos de Dados Parametrizados

- TAD parametrizados permitem o projeto de um TAD que pode armazenar qualquer elemento tipado
- Também conhecidos como classes genéricas
- Nas linguagens
  - C++ - Templates
  - Ada – Pacotes genéricos
  - Java 5.0 e C# - Classes genéricas

Classes genéricas em Java

- Parâmetros genéricos devem ser classes
- Mais comuns tipos genéricos são os tipos de coleção (ex: LinkedList e ArrayList)
- Elimina a necessidade de realizar casts em objetos
- Elimina o problema de ter múltiplos tipos numa estrutura
- Usuários podem definir classes genéricas
- Não podem armazenar tipos primitivos

Classes genéricas em Java

```
import java.util.*;
  public class Stack2<T> {
    private ArrayList<T> stackRef;
    private int maxLen;
    public Stack2() {
      stackRef = new ArrayList<T> ();
      maxLen = 99;
    }
    public void push(T newValue) {
      if (stackRef.size() == maxLen)
        System.out.println("Error in push - stack is full");
      else
        stackRef.add(newValue);
    ...
  }
```

- Instantiation: `Stack2<String> myStack = new Stack2<String> ();`

Construtores de Encapsulamento

- Programas grandes tem duas necessidades especiais:
  - Uma forma de organização além de divisão em subprogramas
  - Uma forma de compilação parcial
- Solução óbvia: agrupas subprogramas relacionados em unidades que podem ser separadamente compiladas
- Tais coleções são chamadas encapsulamentos

Subprogramas aninhados

- Organizar programas através de aninhamento de subprogramas
- Permitidos em Ada, Fortran 95+, Python, JavaScript e Ruby

Encapsulamento em C

- Arquivos contendo um ou mais subprogramas podem ser independentemente compilados
- A interface é colocada num arquivo de cabeçalho (*header file)*
- Problema: o *linker* não checa tipos entre cabeçalhos e suas implementações
- #include usado para incluir arquivos de cabeçalhos em aplicações

Encapsulamento em C++ e Ada

- C++
  - Pode definir arquivos de cabeçalho e código similar a C
  - Ou classes podem ser usadas para encapsulamento
    - A classe é usada como interface (prototypes)
    - As definições são especificadas em um arquivo separado
- Ada
  - Pacotes de especificação de Ada podem incluir declarações de dados e subprogramas
  - Podem ser compilados separadamente

Encapsulamentos de Nome

- São usadas para criar escopos para nomes
- C++ namespaces
  - Podem alocar cada biblioteca no seu próprio namespace e qualificar nomes usados fora com o namespace
  - C# também inclui namespaces
- Pacotes Java
  - Podem conter mais de uma classe; classes no mesmo pacote tem visibilidade *friendly*
  - Clientes de um pacote podem usá-lo através do seu nome completo ou usando *import*

Encapsulamentos de Nome

- Pacotes Ada
  - São definidos em hierarquias que correspondem a hierarquias de arquivos
  - Visibilidade de uma unidade de programa é recebida com a cláusula *with*
- Ruby
  - Classes são encapsulamentos mas Ruby também tem módulos
  - Eles encapsulam coleções de constantes e métodos
  - Não podem ser instanciados, herdados ou definir variáveis
  - Acesso ao conteúdo de um módulo é através do método *require*

Leitura Adicional

- Capítulo 11– Tipos Abstratos de Dados e Encapsulamento. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011
- Próxima aula
  - Capítulo 13 e 14 – Concorrência e Tratamento de Exceções. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011