Capítulo 11

Tipos Abstratos de Dados e
Construções de Encapsulamento

Tópicos

- O conceito de abstração
- Introdução à abstração de dados
- Questões de projeto para os tipos abstratos de dados
- Exemplos em algumas linguagens
- Tipos abstratos de dados parametrizados
- Construções de encapsulamento
- Nomeação de encapsulamentos

O conceito de abstração

- Uma **abstração** é uma visão ou representação de uma entidade que inclui apenas suas partes mais importantes

- É um conceito fundamental na computação

- Dois grandes tipos:
  - **Abstração de processos** (capítulo 9)
  - **Abstração de dados** (este capítulo)

- Praticamente todas as linguagens suportam os dois tipos de abstração

Introdução à abstração de dados

- Um **tipo abstrato de dados** é um tipo de dados definido pelo usuário que satisfaz duas condições fundamentais:
  - A **representação interna dos valores e operações sobre esses valores é escondida** dos programas que usarão esse tipo abstrato de dado (o tipo abstrato de dado tem uma interface que explica como deve ser utilizado)
  - A **declaração** do tipo e os **protocolos das operações** são contidos em uma **única unidade sintática** (a interface do tipo não depende da implementação interna)

Por que usar TADs?

- Vantagem de ocultar detalhes internos:
  - Confiabilidade: usuários não têm acesso direto aos valores e operações, permitindo que a representação possa ser alterada sem afetar o código do usuário
  - Reduz a quantidade de código e variáveis que um programador deve estar ciente (permite focar no essencial)
  - Reduz conflitos de nomes

- Vantagem de unidade sintática única:
  - Organização dos programas
  - Mantém tudo em um único local (facilita a manutenção)
  - Permite compilação independente

Exemplo: pilha (stack)

create(stack)	Creates and possibly initializes a stack object
destroy(stack)	Deallocates the storage for the stack
empty(stack)	A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise
push(stack, element)	Pushes the specified element on the specified stack
pop(stack)	Removes the top element from the specified stack
top(stack)	Returns a copy of the top element from the specified stack

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

Requisitos de linguagem para TADs

- Ter uma unidade sintática que encapsula a definição do tipo abstrato
- Um método para fazer com que o cabeçalho do TAD seja visível aos clientes, escondendo a definição interna
- Agumas operações primitivas no processador da linguagem

Questões de projeto

- Os TADs podem ser parametrizados?
- Que controles de acesso estarão disponíveis?
- A especificação do TAD pode ser separada fisicamente de sua implementação?

Exemplo: C++

- Baseado no conceito de **`struct`** do C; usa classes como mecanismo de encapsulamento
- Uma classe é um tipo
- Todas as instâncias da classe compartilham uma cópia única das operações
- Cada instância tem sua própria cópia de dados
- Instâncias podem ser estáticas, dinâmicas de pilha ou dinâmicas de heap

Exemplo: C++

- Ocultar as informações
  - *Private:* para ocultar entidades
  - *Public:* para criar a interface às entidades
  - *Protected:* para herança (capítulo 12)

Exemplo: C++

- Construtores:
  - Funções para **inicializar os dados das instâncias** (inicializar os valores). Atenção: os construtores **não criam as instâncias** dos objetos!
  - Podem alocar armazenamento se o objeto (ou parte dele) é heap-dynamic
  - Podem ter parâmetros
  - São chamados implicitamente quando uma instância do objeto é criada
  - Podem ser explicitamento chamados
  - Nome é o mesmo da classe

Exemplo: C++

- Destrutores
  - Limpam tudo depois que a instância for destruída (liberam espaço na heap)
  - Chamados implicitamente quando o tempo de vida da instância termina
  - Podem ser explicitamente chamados
  - O nome é o mesmo da classe, precedido por um til (~)

Exemplo: C++ - Classe Pilha

```
class Stack {
   private:
          int *stackPtr, maxLen, topPtr;
   public:
          Stack() {
                 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 () {…};
}
```

Exemplo: C++ - header file

```
#include <iostream.h>
class Stack {
private:
  int *stackPtr;
  int maxLen;
  int topPtr;
public:
  Stack();
  ~Stack();
  void push(int);
  void pop();
  int top();
  int empty();
}
```

Exemplo: C++ - implementação

```
// Stack.cpp

#include <iostream.h>
#include "Stack.h"

using std::cout;

Stack::Stack() {
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}

Stack::~Stack() {delete [] stackPtr;};
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Exemplo: Java

- Parecido com C++, exceto:
  - Todos os objetos são alocados na heap e acessados através de variáveis de referência
  - Coletor de lixo é implícito

Exemplo: Java

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

TADs parametrizados

- Permitem passar argumentos aos construtores
- Permitem criar TAD para armazenar qualquer tipo de elemento
- Também conhecidos por classes genéricas
- C++, Java, C#

TAD parametrizado em C++

- Passagem de argumentos ao construtor

```
Stack (int size) {
   stk_ptr = new int [size];
   max_len = size - 1;
   top = -1;
};
```

Ao declarar o objeto Stack:

```
Stack stk(150);
```

TAD parametrizado em C++

- O elemento do stack pode ser parametrizado tornando a classe um **template**

```
template <class Type>
class Stack {
  private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
  public:
    Stack() {
      stackPtr = new Type[100];
      maxLen = 99;
      topPtr = -1;
    }
    Stack(int size) {
      stackPtr = new Type[size];
      maxLen = size – 1;
      topSub = -1;
    }
  ...
}
```

  - Para criar a instância: `Stack<int> myIntStack;`

Construção de encapsulamentos

- Programas grandes e complexos precisam de duas coisas importantes:
  - **Meio de organização** além da simples divisão em sugprogramas
  - **Meio de realizar compilação parcial** (compilar unidades que são menores do que o programa como um todo)
- Solução: agrupar subprogramas relacionados em uma unidade que possa ser compilada de forma independente
- Esses agrupamentos são chamados de encapsulamentos

Encapsulamento por aninhamento

- Meio de organizar programas colocando subprogramas aninhados em subprogramas maiores

- Python, JavaScript, Ruby

Encapsulamento em C

- Arquivos com um ou mais subprogramas podem ser compilados de forma independente

- A interface é colocada no *header file*

Nomeação de encapsulamentos

- Grandes programas definem muitos nomes globais; é necessário um mecanismo para dividir esses nomes em grupos relacionados

- A nomeação de encapsulamentos cria escopos de nomes (espaços de nomes, namespaces)

E agora?

- Acabou, aproveite para estudar!