Estruturas de Dados
Aula 8: Tipos Abstratos de Dados

30/03/2011

Variação de implementação

- Há diferentes implementações possíveis para o mesmo tipo de dado
- Todas definem o mesmo domínio e não mudam o significado das operações

Variação de implementação (2)

- Exemplo (frações)
- Fração implementação 1

```
typedef struct
{ int numerador;
  int denominador;} fracao ;

int main()
{ fracao f;
  printf ("Digite o numerador: ");
  scanf ("%d", &f.numerador);
  printf ("\nDigite o denominador: ");
  scanf ("%d", &f.denominador);
  return 0;
}
```

Variação de implementação (3)

- Fração implementação 2

```
#include <stdio.h>
#define numerador 0
#define denominador 1

typedef int fracao[2];

int main()
{ fracao f;
  printf ("Digite o numerador: ");
  scanf ("%d", &f[numerador]);
  printf ("\nDigite o denominador: ");
  scanf ("%d", &f[denominador]);
  return 0;
}
```

Substituição de implementações

- Em programas reais, as implementações dos tipos de dados são modificadas constantemente para melhorar a:
  - Velocidade
  - Eficiência
  - Clareza
  - Etc.
- Essas mudanças têm grande impacto nos programas usuários do tipo de dado. Por exemplo:
  - Re-implementação de código
  - Mais suscetível a erros
  - CUSTO MUITO ALTO!

Substituição de implementações

- Como podemos modificar as implementações dos tipos de dados com o menor impacto possível para os programas?
- Como podemos **encapsular** (esconder) de quem usa um determinado tipo de dado a forma concreta como este tipo foi implementado?

  – TIPOS ABSTRATOS DE DADOS (TAD)

Tipos Abstratos de Dados

- Um TAD especifica o tipo de dado (domínio e operações) sem referência a detalhes da implementação
- Minimiza código do programa que usa detalhes de implementação
  - Dando mais liberdade para mudar implementação com menor impacto nos programas
  - Minimiza custos
- Os programas que usam o TAD não "conhecem" as implementações dos TADs
  - Fazem uso do TAD através de operações

TAD Fracao (operações principais)

cria_fracao(N,D)

Pega dois inteiros e retorna a fracao N/D.

acessa_numerador(F)

Pega a fracao e retorna o numerador.

acessa_denominador(F)

Pega a fracao e retorna o denominador.

```
fracao Soma(fracao F1, fracao F2)
{ int n1 = get_numerador(F1);
  n2 = acessa_numerador(F2);
  d1 = acessa_denominador(F1);
  d2 = acessa_denominador(F2);
  return cria_fracao( n1*d2+n2*d1 , d1*d2 ); }
```

Programa usuário do TAD fracao

- Usa o TAD apenas através de suas operações

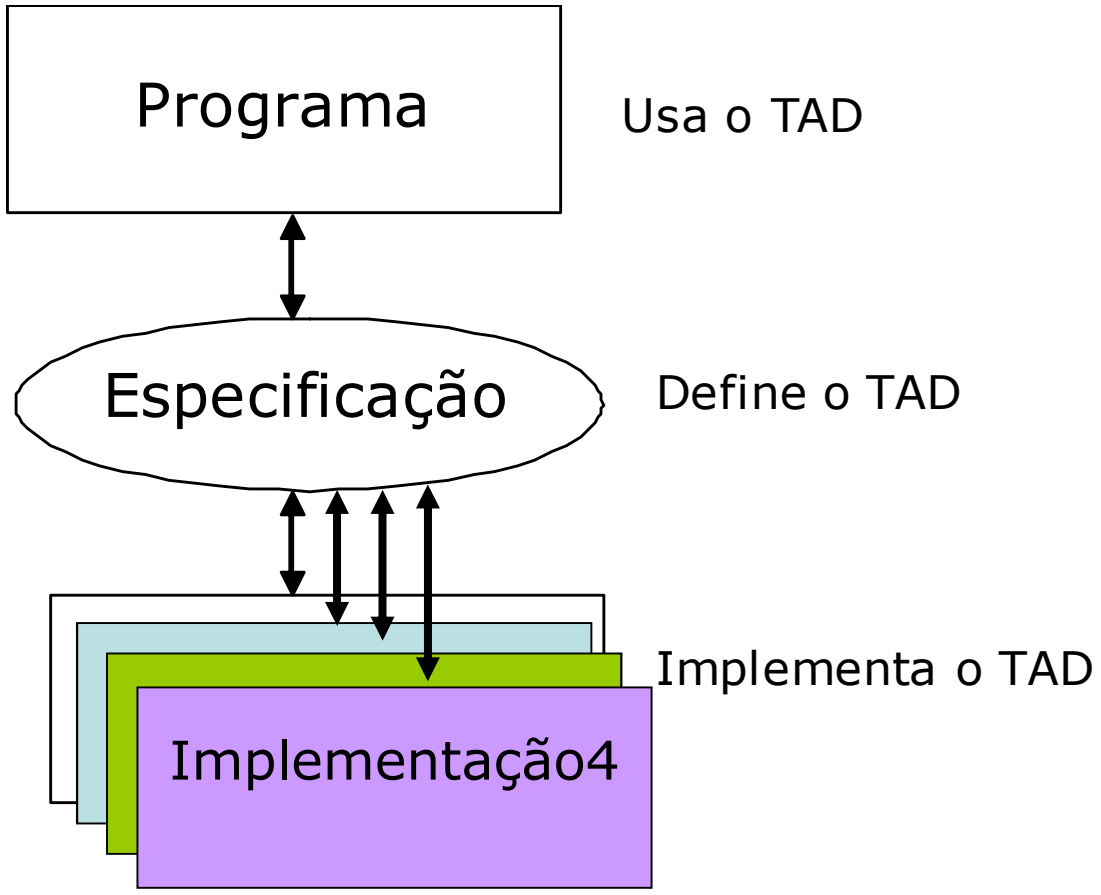
```
#include "fracao.h"

int main()
{ int n, d;
  printf ("Digite o numerador: ");
  scanf ("%d", &n);
  printf ("\nDigite o denominador: ");
  scanf ("%d", &d);
  fracao f = cria_fracao(n, d);
  fracao soma_fracao = soma (f, f);
  return 0;
}
```

Resumindo (TAD)

- Um TAD especifica tudo que se precisa saber para usar um determinado tipo de dado
- Não faz referência à maneira com a qual o tipo de dado será (ou é) implementado

- Quando usamos TAD's, nossos sistemas ficam divididos em:
  - Programas usuários: A parte que usa o TAD
  - Implementação: A parte que implementa o TAD

Resumindo (TAD)



Exemplo TAD Pilha

- Pilha de livros, pilha de pratos, etc.
- Estrutura de dados muito usada em computação (ex., arquitetura de computadores)
- Em uma pilha temos acesso ao elemento do topo apenas, exceto quando retiramos blocos de elementos de uma vez

TAD Pilha (1)

- Uma pilha pode estar vazia ou deve consistir de duas partes:
  - Um elemento do topo
  - Uma pilha (o restante dos elementos)
- Os elementos da pilha podem ser de qualquer tipo, desde que sejam do mesmo tipo
- Operações do TAD Pilha
  - Apresentadas aqui são operações básicas
  - Outras operações podem ser definidas em termos das básicas
- Como podem ver, o TAD pilha não utiliza nenhuma linguagem de programação

Operações do TAD Pilha

- **cria_pilha**
  - Inputs: nenhum
  - Outputs: P (a pilha criada)
  - Pré-condição: nenhuma
  - Pós-condição: P está definida e vazia

- **destroi_pilha**
  - Inputs: P (a pilha)
  - Outputs: P'
  - Pré-condição: none
  - Pós-condição: P' não definida. Todos os recursos de memória alocados para P estão liberados.

Operações do TAD Pilha (2)

- **esta_vazia**
  - Inputs: P (a pilha)
  - Outputs: esta_vazia (boolean)
  - Pré-condição: nenhuma
  - Pós-condição: esta_vazia é true se e somente se P está vazia.
- **top**
  - Inputs: P (a pilha)
  - Outputs: E (um elemento da pilha)
  - Pré-condição: P não está vazia
  - Pós-condição: E é o elemento do topo da pilha (P não é modificada)

Operações do TAD Pilha (3)

- **pop**
  - **Inputs: P (a pilha)**
  - **Outputs: P'**
  - **Pré-condição: P não está vazia**
  - **Pós-condição: um elemento que é o topo da pilha e o restante da pilha (R), onde P′ = R**
- **push**
  - **Inputs: P (uma pilha) e E (um elemento)**
  - **Outputs: P′**
  - **Pré-condição: E é um tipo apropriado da pilha P**
  - **Pós-condição: P′ tem E como o elemento do topo e P como o restante dos elementos**

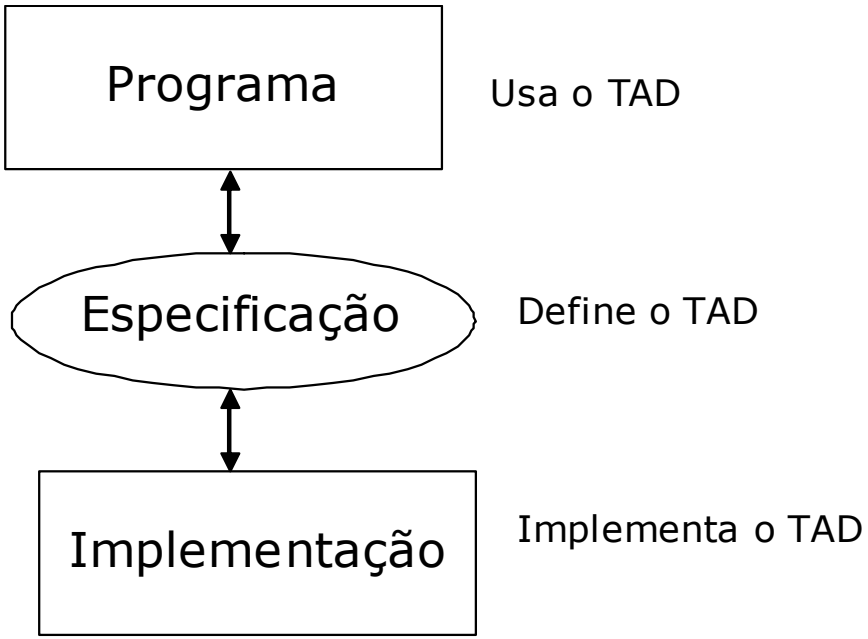
Especificação do TAD

- Devemos definir para cada operação:
  - Inputs, outputs
    - valores de entrada e a saída esperada como resultado da execução da operação
  - Pré-condições
    - Propriedades dos inputs que são assumidas pela operações. Se satisfeitas, é garantido que a operação funcione. Caso contrário, não há garantias e o comportamento é inesperado
  - Pós-condições
    - Define os efeitos causados como resultado da execução da operação
  - Invariantes
    - Propriedades que devem ser sempre verdadeiras (antes, durante e após a execução da operação)

Checagem de pré condições

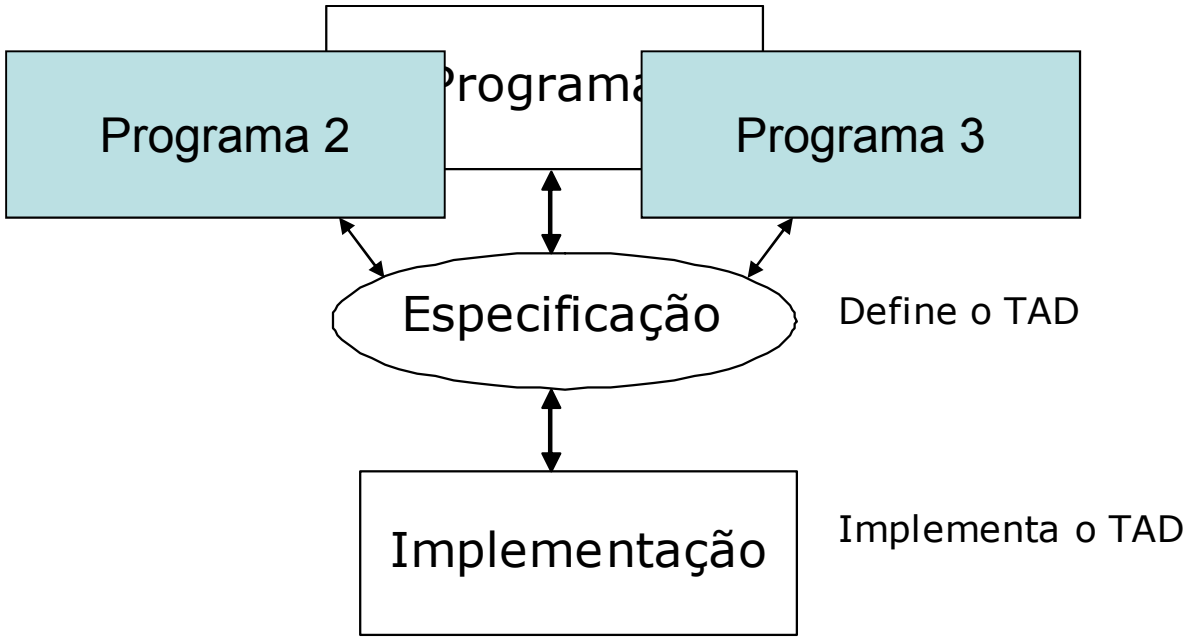
- No programa usuário do TAD
  - Algumas vezes pode ser mais eficiente
- Na implementação do TAD
  - Modificações nas pré-condições são mais facilmente implementadas
  - Erros relacionados a detalhes de implementação são mais facilmente detectados

Software em Camadas



- As camadas de software são independentes
- Modificações na implementação do TAD não geram (grandes) mudanças no programa

Software em Camadas (2)



- Essa abordagem também permite o reuso de códico
- A mesma implementação pode ser usada por vários programas

Exemplos de TAD

TAD Ponto (plano bidimensional)

- **cria_pto**
  - **Input: x e y (coordenadas no plano)**
  - **Output: P (ponto criado)**
  - **Pre: nenhuma**
  - **Pos: P é definido e suas coordenadas são x e y**
- **destroi_pto**
  - **Input: P (o ponto)**
  - **Output: P'**
  - **Pre: nenhuma**
  - **Pos: P' não definido. Todos os recusos de memória alocadores para P estão liberados**

TAD Ponto (2)

- **acessa_x**
  - Input: P (ponto)
  - Output: x
  - Pre: ponto válido e não vazio
  - Pos: P não é modificado

- **acessa_y**
  - Input: P (ponto)
  - Output: y
  - Pre: ponto válido e não vazio
  - Pos: P não é modificado

TAD Ponto (3)

- **atribui_pto**
  - Input: P (ponto), x, y (coordenadas)
  - Output: P'
  - Pre: P válido e não vazio
  - Pos: P'contém valores x e y

- **distancia_pto**
  - Input: P1 (ponto), P2 (ponto)
  - Output: V (valor da distância)
  - Pre: P1 e P2 válidos e não vazios
  - Pos: P1 e P2 não modificados e V contendo o valor da distância entre os pontos

TAD Circulo

- **cria_circ (opção 1)**
  - Input: x, y (coordenadas do centro) e r (raio do círculo)
  - Output: C (o círculo)
  - Pre: r positivo
  - Pos: C é definido, seu centro está nas coordenadas x e y, e seu raio é r
- **cria_circ (opção 2)**
  - Input: PC (o Ponto centro) e r (raio)
  - Output: C (o círculo)
  - Pre: P é definido e não vazio e r positivo
  - Pos: C é definido, seu centro é o ponto PC, e seu raio é r

TAD Circulo

- **destroi_circ**
  - Input: C (o círculo)
  - Output: C′
  - Pre: nenhuma
  - Pos: C′ não definido. Todos os recusos de memória alocadores para C estão liberados
- **area_circ**
  - Input: C (o círculo)
  - Output: V (valor da área)
  - Pre: C é definido e não vazio
  - Pos: C não é modificado

TAD Circulo (2)

- **interior_circ (opção 1)**
  - **Input: C (o círculo) e x, y (coordenadas do ponto)**
  - **Output: B (true se as coordenadas estiverem no interior de C e false caso contrário)**
  - **Pre: C é definido e não vazio**
  - **Pos: C, x e y não são modificados**
- **interior_circ (opção 2)**
  - **Input: C (o círculo) e P (ponto)**
  - **Output: B (true se P estiver interior de C e false caso contrário)**
  - **Pre: C e P são definidos e não vazios**
  - **Pos: C e P não são modificados**

TAD's em C

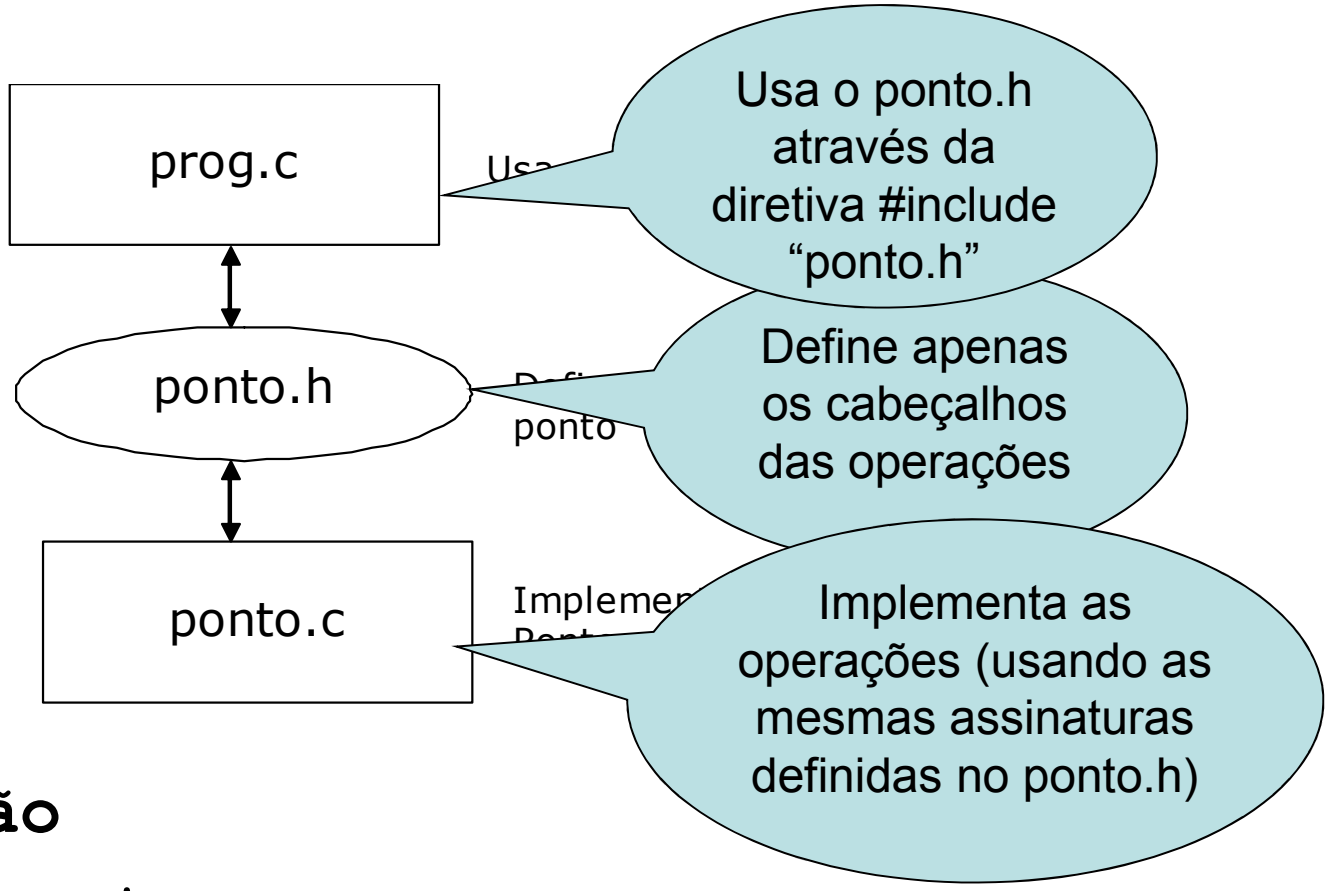
- A linguagem C oferece mecanismos para especificação e uso de TAD's:
  - O uso é possível pois C permite modularização de programas
  - A especificação é possível com o arquivo cabeçalho (.h)
    - O arquivo .h possui apenas os protótipos das operações
    - Usar a #include para incluir o arquivo .h. Inclui o arquivo antes da compilação
  - Os diferentes módulos são incluídos em um único programa executável na "linkagem"

TAD's em C (2)

Exemplo:

- TAD Ponto no arquivo *ponto.h*
- Implementação do tipo ponto no arquivo *ponto.c*
- Módulo que usa a implementação do ponto é *prog.c*
  - **`#include "ponto.h"`**
  - **Inclui o cabeçalho na pré-compilação (chamado pré-processamento)**

TAD's em C (2)



- **Compilação**
  - **gcc -c ponto.c**
  - **gcc -c prog.c**
- **Linkagem**
  - **gcc -o prog.exe ponto.o prog.o**

Abstração

- "É a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais"

- Quando definimos um TAD (Tipo *Abstrato* de Dados), nos concentramos nos aspectos essencias do tipo de dado (operações) e nos abstraímos de como ele foi implementado

Encapsulamento

- "Consiste na separação de aspectos internos e externos de um objeto".

- O TAD provê um mecanismo de encapsulamento de um tipo de dado, no qual separamos a especificação (aspecto externo) de sua implementação (aspecto interno)

Exercício

TAD Matriz (m por n)

- Definir operações básicas para manipulação de elementos (i,j), linhas e colunas
- Para cada operação, definir inputs, outputs, pré-condições, pós-condições
- Quais seriam outras operações interessantes para o TAD matriz (além das básicas)?