

SCC 202 - Algoritmos e Estruturas de Dados I

TAD: Tipo Abstrato de Dados (1)
Conceitos

5/8/2010

Objetivos do Curso

◆ Familiarização

- com os principais **Tipos Abstratos de Dados (TAD)**
- com as **estruturas de dados (ED)** que são usadas para representá-los para permitir o armazenamento e busca **eficiente** de dados armazenados em memória principal (RAM) do computador.

◆ No final do curso espera-se que

- os alunos implementem as várias ED, conhecendo as vantagens e desvantagens de cada uma.

◆ Usaremos C para estudar e trabalhar com os TADs; uso também de pseudo-código

Definindo melhor os objetivos do curso

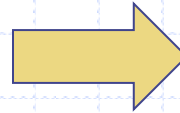


Termos relacionados, mas diferentes

◆ Tipos de Dados:

◆ Pré-definidos/básicos:

- char, int, float, double, void



◆ Definidos pelo usuário:

- enumerações (*enum*)
- Ponteiros

- ◆ **Estruturados**: conjuntos uni e bidimensionais (vetores e matrizes), registros (struct), uniões ou registros variantes (union)

◆ Estruturas de Dados

◆ Tipos Abstratos de Dados

Modificadores dos Tipos Básicos

Tabela: Todos os Tipos de dados definidos pelo Padrão ANSI C, seus tamanhos em bytes e suas faixa de valores.

Tipo	Tamanho em Bytes	Faixa Mínima
<code>char</code>	1	-127 a 127
<code>unsigned char</code>	1	0 a 255
<code>signed char</code>	1	-127 a 127
<code>int</code>	4	-2.147.483.648 a 2.147.483.647
<code>unsigned int</code>	4	0 a 4.294.967.295
<code>signed int</code>	4	-2.147.483.648 a 2.147.483.647
<code>short int</code>	2	-32.768 a 32.767
<code>unsigned short int</code>	2	0 a 65.535
<code>signed short int</code>	2	-32.768 a 32.767
<code>long int</code>	4	-2.147.483.648 a 2.147.483.647
<code>signed long int</code>	4	-2.147.483.648 a 2.147.483.647
<code>unsigned long int</code>	4	0 a 4.294.967.295
<code>float</code>	4	Seis dígitos de precisão
<code>double</code>	8	Dez dígitos de precisão
<code>long double</code>	10	Dez dígitos de precisão

Tipo de Dados

- ◆ Em linguagens de programação, o tipo de uma variável define o conjunto de valores que ela pode assumir (domínio)
 - Por exemplo, uma variável booleana (Pascal) pode ser *true* ou *false*
- ◆ E as possíveis operações
 - Com booleanos temos as operações lógicas and, or, not e a de atribuição, por exemplo.
- ◆ Em programas reais ... **novos tipos de dados** podem ser definidos em função dos existentes
- ◆ Por exemplo, como definiriam o tipo de dados racional?

Racional

◆ Possivelmente como:

- um **vetor** de 2 elementos inteiros, cujo primeiro poderia ser o numerador e o segundo o denominador
- Um **registro** de 2 campos inteiros: numerador e denominador
- ...

Variação de implementação


- ◆ Há diferentes implementações possíveis para o mesmo tipo de dado para melhorar:
 - Velocidade do código
 - Eficiência em termos de espaço
 - Clareza, etc.

- ◆ Todas definem o mesmo domínio e **não mudam o significado das operações**:
 - Para racionais podemos: criar, somar, multiplicar, ver se são iguais, imprimir, etc.

Substituição das implementações

- ◆ As mudanças nas implementações têm grande impacto nos programas dos usuários. Por exemplo:
 - Re-implementação do código
 - Possíveis erros

- ◆ Pergunta principal que o curso tenta responder:



◆ Como podemos modificar as implementações dos tipos com o menor impacto possível para os programas que os usam?

◆ Podemos esconder (**encapsular**) de quem usa o tipo de dado **a forma como foi implementado??**

Sim, com Tipos Abstratos de Dados

- ◆ Um tipo abstrato de dado, ou TAD,
 - especifica um conjunto de operações (ou métodos) e
 - a semântica das operações (o que elas fazem)
- ◆ mas não especifica a implementação das operações. Isto é o que o faz abstrato.
 - Tipo de dados divorciado da implementação !
- ◆ Os programas que usam o TAD não “conhecem” as implementações
 - Fazem uso do TAD através de suas operações

TAD

◆ Definido pelo par (v,o)

- ◆ v: valores, dados a serem manipulados
- ◆ o: operações sobre os valores/dados

◆ Os programas que usam o TAD não “conhecem” as implementações

- Fazem uso do TAD através de suas operações

Estruturas de Dados (ED)

- ◆ A **implementação** de um TAD escolhe uma ED para representá-lo.
- ◆ Cada ED é construída dos **tipos básicos** (*int, real, char, ...*) e/ou dos **tipos estruturados** (*array, record, ...*) de uma linguagem de programação.
- ◆ Por muitos anos, os projetistas de software consideraram importante identificar as ED e os algoritmos bem cedo no ciclo de vida do software.
- ◆ Com o aparecimento de linguagens como Módulo-2 e ADA, tornou-se possível **deixar as decisões sobre as ED para bem mais tarde** no projeto de desenvolvimento de software (**abstração**).

Ocultamento de Informação

- ◆ A característica essencial de um TAD é a separação entre conceito e implementação.
- ◆ O termo "ocultamento de informação" é utilizado para descrever esta habilidade.
- ◆ Ao usuário são fornecidos a descrição dos valores e o conjunto de operações do TAD, mas a **implementação é invisível e inacessível.**

Compilação em separado

- ◆ A separação da **definição** do TAD de sua **implementação**
 - permite que a mudança de implementação não altere o programa que usa o TAD.
- ◆ O **TAD** é **compilado separadamente**, e uma mudança força somente a compilação, de novo, do arquivo envolvido e uma nova link-edição do programa
 - (mais rápida que uma nova compilação do programa).

Exemplo: Definição do TAD Racional (Pascal)

```
Procedure Cria_Racional(r1: integer; r2: integer; var r: Rac);  
  {Cria um número racional a partir de dois inteiros r1 e r2 se r2 <> 0, }  
  {caso contrário exibe mensagem explicativa}
```

```
Procedure Soma_Racional (r1, r2: Rac; var soma_r: Rac);  
  { Soma dois números racionais r1 e r2 e retorna }  
  { o resultado em soma_r }
```

```
Procedure Mult_Racional (r1, r2: Rac; var mult_rac: Rac);  
  { Multiplica dois números racionais r1 e r2 e }  
  { retorna o resultado em multi_rac }
```

```
Function Teste_Igualdade (r1, r2: Rac): boolean;  
  { Verifica se 2 números racionais r1 e r2 são iguais, e se sim retorna }  
  { True, caso contrário False }
```

```
Procedure Imprimi_Racional( r: Rac);  
  {imprime um numero racional r}
```

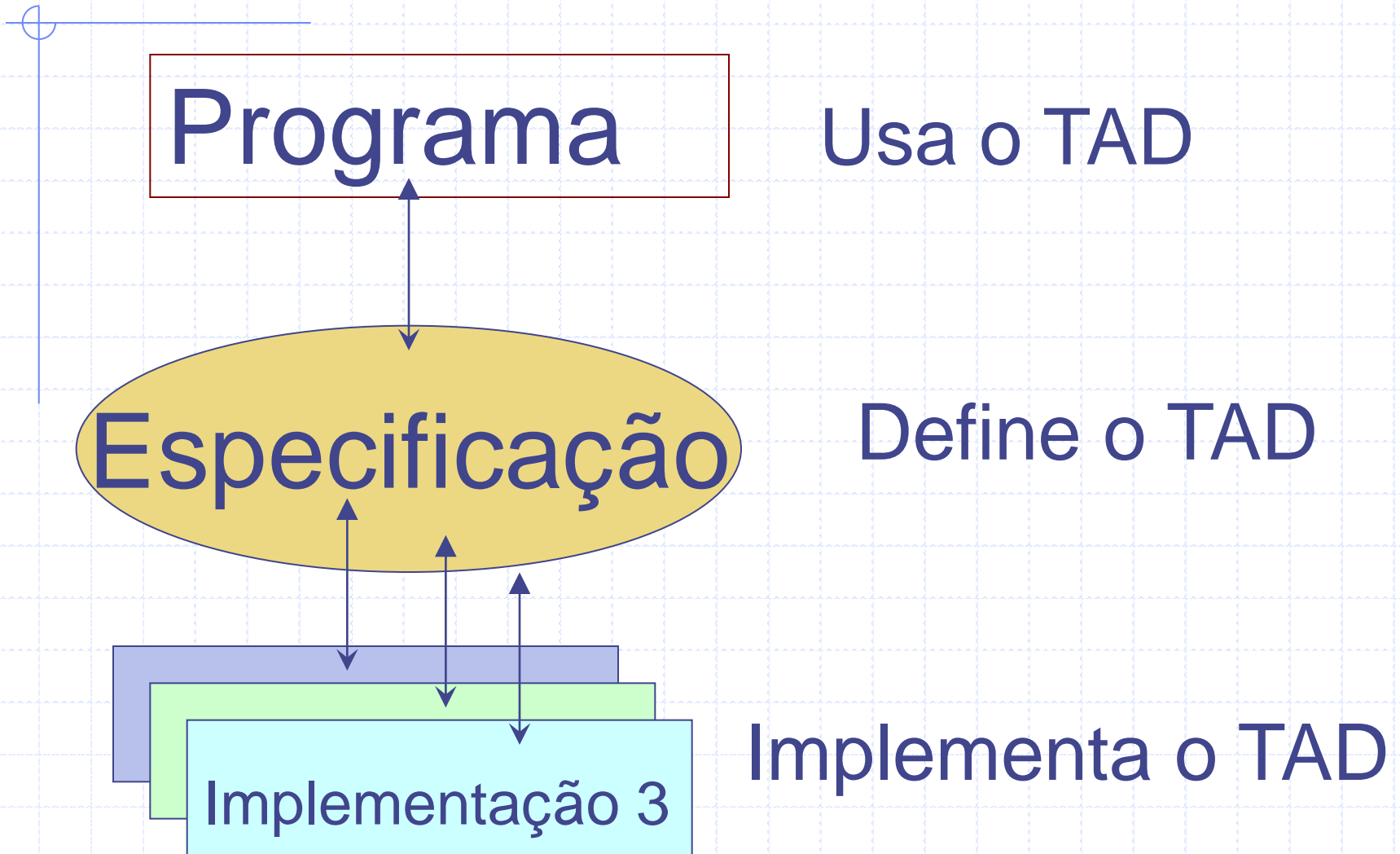
Por que isto é útil?

- ◆ Simplifica a tarefa de especificar um algoritmo
 - se você pode denotar as operações que você precisa sem ter que pensar, ao mesmo tempo, como as operações são executadas.
- ◆ Uma vez que existem geralmente muitas maneiras de implementar um TAD, pode ser útil escrever um algoritmo que pode ser usado com qualquer das possíveis implementações.
- ◆ TADs bastante conhecidos, como o TAD Pilha, já estão implementados em bibliotecas padrão, então eles podem ser escritos uma vez e usado por muitos programadores (**REUSO**).
- ◆ As operações em TADs fornecem uma linguagem de alto nível comum para especificar e falar sobre algoritmos.

Resumindo TAD ...

- ◆ Quando falamos sobre TADs, geralmente distinguimos
 - o código que usa o TAD, chamado **cliente ou programa usuário**,
 - do código que implementa o TAD, chamado código **fornecedor ou implementação**.

Resumindo TAD....software em camadas independentes => permite reuso



Resumindo...

- ◆ Para especificar um TAD não precisamos de nenhuma linguagem de programação
- ◆ Para cada operação, precisamos pensar:
 - Nas entradas, saídas, pré-condição e pós-condição

Especificação do TAD

- ◆ Para cada operação:
 - Entradas e saídas
 - Pré-condições (propriedades das entradas que são assumidas pelas operações. Se satisfeitas, é garantido que a operação funcione, caso contrário não há garantias.
 - Pós-condições (efeitos da execução da operação)

Para implementar TADs

1. Modularização

- ◆ Divisão do código em vários arquivos/compilação em separado

2. Definição de tipos (typedef)

- juntamente com a implementação de funções que agem sobre aquele tipo

3. Passagem de parâmetros

- ◆ Um parâmetro pode especificar um objeto em particular, deixando a operação genérica

4. *Flag* para erro, sem emissão de mensagem no código principal

- ◆ Independência do TAD

TADs em C

- ◆ Implementação dos TADs em arquivos separados do programa principal
- ◆ Para isso geralmente separa-se a declaração e a implementação do TAD em dois arquivos:
 - NomeDoTAD.h : com a declaração
 - NomeDoTAD.c : com a implementação
- ◆ O programa ou outros TADs que utilizam o seu TAD devem dar um `#include` no arquivo `.h`

1. Modularização em C

- ◆ Programa em C pode ser dividido em vários arquivos
 - Arquivos **fonte** com extensão **.c**
 - ◆ Denominados de módulos
- ◆ Cada módulo deve ser compilado separadamente
 - Para tanto, usa-se um **compilador**
 - Resultado: **arquivos objeto** não executáveis
 - ◆ Arquivos em linguagem de máquina com extensão **.o** ou **.obj**
- ◆ Arquivos objeto devem ser juntados em um **executável**
 - Para tanto, usa-se um *ligador* ou **link-editor**
 - Resultado: um único arquivo em linguagem de máquina
 - ◆ Usualmente com extensão **.exe**

- ◆ Módulos são muito úteis para construir bibliotecas de funções inter-relacionadas. Por exemplo:
 - Módulos de funções matemáticas
 - Módulos de funções para manipulação de strings
 - etc

- ◆ Em C, é preciso listar no início de cada módulo aquelas funções de outros módulos que serão utilizadas:
 - Isso é feito através de uma lista denominada **cabeçalho**

- ◆ **Exemplo:** considere um arquivo STR.c contendo funções para manipulação de strings, dentre elas:
 - **int** comprimento (**char*** strg)
 - **void** copia (**char*** dest, **char*** orig)
 - **void** concatena (**char*** dest, **char*** orig)

- ◆ **Exemplo (cont):** Qualquer módulo que utilizar essas funções deverá incluir no início o cabeçalho das mesmas, como abaixo.

```
/* Programa Exemplo.c */
#include <stdio.h>
int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);
int main (void) {
    char str[101], str1[51], str2[51];
    printf("Entre com uma seqüência de caracteres: ");
    scanf(" %50s[^\n]", str1);
    printf("Entre com outra seqüência de caracteres: ");
    scanf(" %50s[^\n]", str2);
    copia(str, str1); concatena(str, str2);
    printf("Comprimento total: %d\n", comprimento(str));
    return 0; }
```

◆ Exemplo (cont):

- A partir desses dois fontes (Exemplo.c e STR.c), podemos gerar um executável compilando cada um separadamente e depois ligando-os
- Por exemplo, com o compilador Gnu C (gcc) utilizaríamos a seguinte seqüência de comandos para gerar o arquivo executável Teste.exe:

```
> gcc -c STR.c
```

```
> gcc -c Exemplo.c
```

```
➤ gcc -o Teste.exe STR.o Exemplo.o
```

Questão:

É preciso inserir manualmente e individualmente todos os cabeçalhos de todas as funções usadas por um módulo?

E se forem muitas e de diferentes módulos?

◆ Solução

- **Arquivo de cabeçalhos** associado a cada módulo, com:
 - ◆ cabeçalhos das funções oferecidas pelo módulo e,
 - ◆ eventualmente, os tipos de dados que ele **exporta**
 - typedefs, structs, etc.
- Segue o mesmo nome do módulo ao qual está associado
 - ◆ porém com a **extensão .h**

◆ Exemplo:

- Arquivo STR.h para o módulo STR.c do exemplo anterior

```
/* Arquivo STR.h */
```

```
/* Função comprimento:
```

```
Retorna o no. de caracteres da string str */
```

```
int comprimento (char* str);
```

```
/* Função copia:
```

```
Copia a string orig para a string dest */
```

```
void copia (char* dest, char* orig);
```

```
/* Função concatena:
```

```
Concatena a string orig na string dest */
```

```
void concatena (char* dest, char* orig);
```

◆ O programa Exemplo.c pode então ser reescrito como:

```
/* Programa Exemplo.c */
#include <stdio.h> /* Módulo da Biblioteca C Padrão */
#include "STR.h" /* Módulo Próprio */
int main (void) {
    char str[101], str1[51], str2[51];
    printf("Entre com uma seqüência de caracteres: ");
    scanf(" %50s[^\n]", str1);
    printf("Entre com outra seqüência de caracteres: ");
    scanf(" %50s[^\n]", str2);
    copia(str, str1); concatena(str, str2);
    printf("Comprimento total: %d\n", comprimento(str));
    return 0; }
```

Nota: O uso dos delimitadores < > e " " indica onde o compilador deve procurar os arquivos de cabeçalho, na biblioteca interna (<>) ou no diretório indicado (" " – *default* se ausente).

TADs em C

◆ Módulos podem ser usados para definir um novo tipo de dado e o conjunto de operações para manipular dados desse tipo:

- Tipo Abstrato de Dados (TAD)

◆ Definindo um tipo *abstrato*, pode-se “esconder” a implementação

- Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como ele é implementado
- Facilita manutenção e re-uso de códigos, entre outras vantagens

2. Typedef

◆ O comando **typedef** permite ao programador definir um novo nome para um determinado tipo:

- `typedef int Tamanho`
- `typedef char *Cadeia`
- `typedef struct tipo_endereco {
 char rua [50];
 int numero;
 char bairro [20];
 char cidade [30];
 char sigla_estado [3];
 long int CEP;
} TEndereco;`

...continuação

```
struct data {  
    int dia;  
    char mes[10];  
    int ano;  
};  
  
typedef struct data novo_tipo;  
  
novo_tipo natal, ano_novo;
```

Modularização em Pascal

◆ Em Pascal

■ *units*

- ◆ Coleção de dados (variáveis, constantes, etc.) e procedimentos e funções que manipulam os dados
- ◆ É uma biblioteca ativada pelo programa principal pela diretiva *uses*
- ◆ Possui duas partes
 - *Interface*: declarações e protótipos
 - *Implementation*

◆ Meio caminho andado,

- pois a interface e implementação não estão em arquivos separados.

TAD RACIONAL em PASCAL -- Unit Elemento.PAS

Unit elemento;

Interface

{definicao de tipo racional}

Type Rac = array [1..2] of integer;

{ ou Type Rac = Record

Nun, Den : integer

End;

}

Implementation

End.

Unit Racional.PAS

Unit Racional;

Interface

uses elemento;

{Type Rac}

Procedure Cria_Racional(r1, r2: integer; var r: Rac);
{ Cria um número racional a partir de dois inteiros se $r2 \neq 0$, }
{ caso contrário exibe mensagem explicativa }

Procedure Soma_Racional (r1, r2: Rac; var soma_rac: Rac);
{ Soma dois números racionais R1 e R2 e retorna }
{ o resultado em Result }

Procedure Mult_Racional (r1, r2: Rac; var mult_rac: Rac);
{ Multiplica dois números racionais R1 e R2 e }
{ retorna o resultado em Result }

Function Teste_Igualdade (r1, r2: Rac): boolean;
{ Verifica se 2 números racionais R1 e R2 são iguais, e se sim retorna True, }
{ caso contrário False }

Procedure Imprimi_Racional(r: Rac);
{ imprime um numero racional }

Unit Racional.PAS

```
{ Continuação... }
```

Implementation

```
{Implementacao de procedimentos e funcoes}
```

```
Procedure Cria_Racional(r1, r2: integer; var r: Rac);
```

```
Begin
```

```
  if r2<>0 then { pre-condicao}
```

```
    begin      { pos-condicao}
```

```
      r[1]:= r1;
```

```
      r[2]:=r2;
```

```
    end
```

```
  else writeln ('Erro: Denominador Zero')
```

```
End;
```

```
Procedure Imprimi_Racional( r: Rac);
```

```
{Imprime racional - pos-condicao}
```

```
Begin
```

```
  writeln('racional criado:',r[1], '/',r[2]);
```

```
  writeln;
```

```
End;
```

Unit Racional.PAS

{ Continuação... }

```
Procedure Soma_Racional (r1, r2: Rac; var soma_rac: Rac);
```

```
Begin
```

```
  soma_rac[2]:= r1[2] * r2[2];  {pos-condicao}
```

```
  soma_rac[1]:= r1[1] * r2[2] + r2[1] * r1[2];
```

```
End;
```

```
Procedure Mult_Racional (r1, r2: Rac; var mult_rac: Rac);
```

```
Begin
```

```
  mult_rac[1]:= r1[1] * r2[1];  {pos-condicao}
```

```
  mult_rac[2]:= r1[2] * r2[2];
```

```
End;
```

```
Function Teste_Igualdade (r1, r2: Rac): boolean;
```

```
Begin
```

```
  Teste_Igualdade:=false;
```

```
  {pos-condicao}
```

```
  If (r1[1] * r2[2] = r1[2]* r2[1]) then
```

```
    Teste_Igualdade := true;
```

```
End;
```

```
End.
```

Programa que usa o TAD

```
{Programa que utiliza a Unit Racional}  
program racio;
```

```
uses Elemento, Racional;
```

```
Var a, a1: integer;
```

```
    b, b1: integer;
```

```
    r0, r1, soma: Rac;
```

```
Begin
```

```
  writeln('entre com os valores inteiros');
```

```
  readln(a,b);
```

```
  readln (a1,b1);
```

```
  {Cria um nro racional}
```

```
  Cria_Racional(a,b,r0);
```

```
  Imprimi_Racional(r0);
```

```
  {Cria um nro racional}
```

```
  Cria_Racional(a1, b1, r1);
```

```
  Imprimi_Racional(r1);
```

```
  {Cria uma soma}
```

```
  Soma_Racional(r0,r1,soma);
```

```
  Imprimi_Racional(soma);
```

```
  readln;
```

```
End.
```


TAD RACIONAL em C

◆ Elemento.h

```
/*definicao de tipo racional – comentar um dos 2*/
```

```
/* como struct */
```

```
typedef struct{  
    int Num;  
    int Den;  
} Rac;
```

```
/* como vetor */
```

```
typedef int Rac[2];
```

3. parâmetros

Racional.h

```
#include <stdbool.h>
```

```
#include "Elemento.h"
```

```
void Cria_Racional(int r1, int r2, Rac *r);
```

```
/* Cria um número racional a partir de dois inteiros se r2 <> 0,  
caso contrário exibe mensagem explicativa */
```

```
void Soma_Racional (Rac *r1, Rac *r2, Rac *soma_rac);
```

```
/* Soma dois números racionais R1 e R2 e retorna  
o resultado em Result */
```

```
void Mult_Racional (Rac *r1, Rac *r2, Rac *mult_rac);
```

```
/* Multiplica dois números racionais R1 e R2 e  
retorna o resultado em Result */
```

Onde vamos
colocar a
mensagem?????
QUAL outra
opção para indicar
sucesso ou não?

Note que as funções recebem e retornam PONTEIROS (para o tipo Rac). Isso porque o cliente não conseguirá declarar uma variável do TAD, pois seu tamanho e composição são desconhecidos.

No entanto, o cliente consegue declarar um ponteiro para uma variável do TAD, pois o ponteiro é uma variável cujo tamanho independe do tipo de dado que aponta, já que armazena apenas um endereço de memória.

Racional.h

{continua...}

```
bool Teste_Igualdade (Rac *r1, Rac *r2);
```

```
/* Verifica se 2 números racionais R1 e R2 são iguais, e se sim retorna True,  
caso contrário False */
```

```
void Imprimi_Racional(Rac *r);
```

```
/* Imprime um numero racional */
```

Obs.: A interface é a mesma independente da estrutura implementada no Elemento.h

4. Flag para erros

Racional.c

◆ tipo Rac: implementado com Struct

```
#include "Racional.h"
#include <stdio.h>
void Cria_Racional(int r1, int r2, Rac *r){
    if (r2!=0) /* pre-condicao*/
    {
        /* pos-condicao*/
        r->Num = r1;
        r->Den = r2;
    }
}
```

Não seria melhor retornar uma indicação do sucesso ou não da operação???

Mensagem de Erro

◆ tipo Rac: implementado com Struct

```
#include "Racional.h"
#include <stdio.h>
void Cria_Racional(int r1, int r2, Rac *r){
    if (r2!=0) /* pre-condicao*/
    {          /* pos-condicao*/
        r->Num = r1;
        r->Den = r2;

    }
    printf ("Erro: Denominador zero \n"); exit(1);
}
```

Não é uma boa solução,
Pois assim imprimimos
mensagem
no código do cliente.

Flag de Sucesso/Insucesso

◆ tipo Rac: implementado com Struct

```
#include "Racional.h"
#include <stdio.h>
bool Cria_Racional(int r1, int r2, Rac *r){
/* se a execução foi realizada com sucesso retorna true se não
false*/
if (r2!=0) /* pre-condicao*/
{
/* pos-condicao*/
r->Num = r1;
r->Den = r2;
return true;
}
return false;
}
```

Mais adequado, pois assim o cliente trata o erro da forma que desejar

Racional.c

```
void Imprimi_Racional (Rac *r){
    if(r != NULL) printf("\n Racional criado: %d/%d \n", r->Num, r->Den);
}

void Soma_Racional (Rac *r1, Rac *r2, Rac *soma_rac){
    soma_rac->Den = r1->Den * r2->Den; /*pos-condicao*/
    soma_rac->Num = r1->Num * r2->Den + r2->Num * r1->Den;
}

void Mult_Racional (Rac *r1, Rac *r2, Rac *mult_rac){
    mult_rac->Num = r1->Num * r2->Num; /*pos-condicao*/
    mult_rac->Den = r1->Den * r2->Den;
}

bool Teste_Igualdade (Rac *r1, Rac *r2){
    /*pos-condicao*/
    if (r1->Num * r2->Den == r1->Den * r2->Num) return true;
    return false;
}
```

Racional.c

◆ Tipo Rac: implementado como Vetor

```
#include "Racional.h"
#include <stdio.h>
void Cria_Racional(int r1, int r2, Rac *r){
    if (r2!=0) /* pre-condicao*/
    {          /* pos-condicao*/
        *r[0] = r1;
        *r[1] = r2;
    }
}
void Imprimi_Racional(Rac *r)
{ /*Imprime racional - pos-condicao*/
    if(r != NULL){
        printf("\n Racional criado: %d/%d \n", *r[0], *r[1]);
    }
}
```


Racional.c

```
{continua...}
```

```
void Soma_Racional (Rac *r1, Rac *r2, Rac *soma_rac){  
    *soma_rac[1] = *r1[1] * *r2[1];    /*pos-condicao*/  
    *soma_rac[0] = *r1[0] * *r2[1] + *r2[0] * *r1[1];  
}
```

```
void Mult_Racional (Rac *r1, Rac *r2, Rac *mult_rac){  
    *mult_rac[0] = *r1[0] * *r2[0];    /*pos-condicao*/  
    *mult_rac[1] = *r1[1] * *r2[1];  
}
```

```
bool Teste_Igualdade (Rac *r1, Rac *r2){  
    /*pos-condicao*/  
    if (*r1[0] * *r2[1] == *r1[1] * *r2[0]) return true;  
    return false;  
}
```

Programa que usa o TAD

```
#include <stdlib.h>
#include <stdio.h>
#include "Racional.h"
```

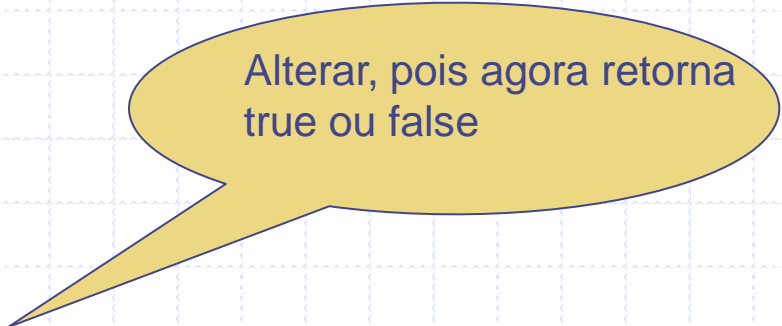
```
int num1, den1;
int num2, den2;
Rac r1, r2, soma;
```

```
int main(){
    printf(" **** Programa Racional **** \n");
    printf("Primeiro Numero (no formato num1/den1): \n");
    scanf("%d/%d",&num1,&den1);

    printf("Segundo Numero (no formato den2/num2): \n");
    scanf("%d/%d",&num2,&den2);
```

Programa que usa o TAD

```
{continua...}  
/*Cria um nro racional*/  
Cria_Racional(num1,den1,&r1);  
Imprimi_Racional(&r1);  
  
/*Cria um nro racional*/  
Cria_Racional(num2, den2, &r2);  
Imprimi_Racional(&r2);  
  
/*Cria uma soma*/  
Soma_Racional(&r1,&r2,&soma);  
Imprimi_Racional(&soma);  
  
system("pause");  
return 0;  
}
```



Alterar, pois agora retorna true ou false

Obs: A estrutura implementada é transparente ao usuário.