Métodos Computacionais

Tipos Abstratos de Dados

Tipos Abstratos de Dados

- Em C:
  - Arquivos-fontes que agrupam funções afins são geralmente denominados de Módulos
  - Em programas modulares, cada módulo deve ser compilado separadamente e depois “linkados” para gerar executável
  - Quando módulo define um novo tipo de dado e o conjunto de operações para manipular dados deste tipo, dizemos que o módulo representa um Tipo Abstrato de Dado (TAD)

Tipos Abstratos de Dados

- Objetivos
  - Encapsular (esconder) de quem usa um determinado tipo, a forma concreta com que ele foi implementado
  - O cliente utiliza o TAD de forma abstrata, apenas com base nas funcionalidades oferecidas pelo tipo (interface)
  - Desacoplar a implementação do uso, facilitando a manutenção e aumentando o potencial de reuso do tipo criado

Tipos Abstratos de Dados

- Geralmente a interface de um TAD é descrita em C nos arquivos .h
- O cliente só precisa dar um “include” no .h que contém a interface do TAD
  - Ao cliente também é dado o arquivo objeto (.o) com a implementação do TAD
    - Esconde (Abstrai) a implementação
- A interface de um TAD contém os protótipos das funções que ele oferece e contém a definição do tipo de dado
  - Funciona como um “manual de uso” do tipo que está sendo definido

Exemplo de TAD: Ponto

```
/* Interface dada pelo arquivo ponto.h */

/* TAD: Ponto (x, y) */


/* Tipo exportado */

typedef struct ponto Ponto;

/* Funções exportadas */

/* Função cria: Aloca e retorna um ponto com
coordenadas (x, y) */

Ponto*   pto_cria  ( float  x ,  float  y ) ;

/* Função libera: Libera a memória de um ponto
previamente criado*/

void   pto_libera   ( Ponto   *p ) ;
```

Só é declarado o tipo, não os campos que fazem parte do tipo

Exemplo de TAD: Ponto

```
/* Função acessa: Retorna os valores das coordenadas de um ponto */
void pto_acessa(Ponto *p,float *x,float *y);


/* Função atribui: Atribui novos valores às coordenadas de um ponto */
void pto_atribui(Ponto *p,float x,float y);


/* Função distancia: Retorna a distância entre dois pontos */


float pto_distancia(Ponto *p1,Ponto *p2 ) ;
```

Importante colocar comentários explicando o que a função faz

Usando o TAD Ponto

```
#include  <stdio.h>
#include  "ponto.h"

int  main( ) {
  Ponto *p  =  pto_cria(2.0,1.0) ;
  Ponto *q  =   pto_cria(3.4,2.1 ) ;
  float   d  =  pto_distancia ( p , q ) ;
  printf  ( "Distancia entre pontos: %f\n" ,  d ) ;
  pto_libera ( q ) ;
  pto_libera ( p ) ;
  return   0 ;
 }
```

Quem utiliza o TAD, precisa incluir o arquivo .h e ter o arquivo objeto com a implementação do TAD

Implementando o TAD Ponto

- Implementação é feita em arquivos .c pelo criador do TAD
  - Cliente não precisa ter acesso ao arquivo.c, e sim ao arquivo compilado (objeto)

```
/* Implementação dada pelo arquivo  ponto.c */

#include <stdlib.h>    /* malloc ,  free ,  exit */
#include <stdio.h>    /* printf */
#include <math.h>   /* sqrt */
#include "ponto.h"

struct   ponto  {
   float   x ;
   float   y ;} ;
```

Estrutura ponto é definida neste arquivo

Implementando o TAD Ponto

```
Ponto* pto_cria(float x,float y){
      Ponto *p = (Ponto*) malloc (sizeof( Ponto ));
      if (p == NULL) {
         printf (“Memória insuficiente\n”);
         exit (1) ;
      }
      p->x = x ;
      p->y = y ;
      return  p ;
 }
```

Implementando o TAD Ponto

```
void pto_libera(Ponto *p){
    free  (p);
}
void pto_acessa(Ponto *p,float *x,float *y){
   *x  =  p->x ;
   *y  =  p->y ;
}
void pto_atribui(Ponto *p,float x,float y){
     p->x  =  x ;
     p->y  =  y ;
}

float pto_distancia( Ponto *p1,Ponto *p2 ){
    float  dx = p2->x - p1->x ;
    float  dy = p2->y - p1->y ;
    return  sqrt(dx * dx + dy * dy);
}
```

Exemplo de TAD: Circulo

```
/* Interface dada pelo arquivo circulo.h */
/* TAD: Circulo */

/* Dependência de Módulos */
#include   "ponto.h"

/* Tipo exportado */
typedef   struct   circulo   Circulo ;
```

As dependências entre módulos devem ser explicitadas

Exemplo de TAD: Circulo

```
/* Funções exportadas */

/* Função cria: Aloca e retorna um circulo com centro
(x, y) e raio r */

Circulo* circ_cria(float x, float  y ,  float   r ) ;

/* Função libera:Libera a memória de um circulo
previamente criado*/

void   circ_libera( Circulo   *c ) ;

/* Função area:Retorna o valor da área do círculo. */
float   circ_area( Circulo  *c ) ;
/* Função interior: Verifica se um dado ponto está
dentro do círculo */
 int   circ_interior ( Circulo   *c , Ponto   *p ) ;
```

Dependência entre módulos

Implementando o TAD Circulo

```
/* Implementação dada pelo arquivo  circulo.c */

#include  <stdlib.h>
#include <stdio.h>
#include  “circulo.h”
#define   PI   3.14159

struct   circulo  {
   Ponto   *p ;
   float      r ;   } ;
```

Implementando o TAD Circulo

```
Circulo* circ_cria(float x,float y,float r){
    Circulo *c =(Circulo*) malloc(sizeof(Circulo)) ;
    c->p  =  pto_cria(x, y) ;
    c->r  =  r ;
    return   c  ;  }

void circ_libera(Circulo *c){
     pto_libera(c->p) ;
      free(c) ;
}
```

Função oferecida pelo TAD Ponto

Implementando o TAD Circulo

```
float circ_area(Circulo *c){
    return   PI * c->r * c->r ;
}

int circ_interior(Circulo *c,Ponto *p) {
    float   d =  pto_distancia(c->p,p) ;
    return (d < c->r) ;
}
```

Exemplo de TAD: String

```
/* Interface dada pelo arquivo String.h */
/* TAD: String */
/* Tipo exportado */
typedef    struct   string   String ;
/* Funções exportadas */
String* str_cria(char* str);
String*  str_copia (String* dest, String* orig) ;
int str_comprimento(String* str);
String* str_concatena(String* dest, String* orig);
```

Implementando o TAD String

- Podemos utilizar mais de uma estratégia para implementar Strings
  - Vetores de char com tamanho fixo
  - Ponteiros para char
- Uso do conceito de TAD, esconde do cliente do módulo a estratégia escolhida
  - Só quem sabe é o implementador
  - Assinaturas das funções serão as mesmas independentemente da implementação

Implementando String com Vetor

```
/* Implementação dada pelo arquivo  String.c */

#include <stdlib.h>
#include <stdio.h>
#include "String.h"
#include <string.h>
#define N 4000

/* Estratégia 1 : Vetores Fixos */
    struct   string  {
        char s[N] ;
        int  tam /* diz o comprimento */
    };
```

Estrutura string possui um vetor de tamanho fixo

Implementando String com Vetor

```
String* str_cria(char* c){
      String *str; int comp;
      comp = strlen(c);
      if (comp > N - 1) {
        printf( "String muito grande\n" );
        return NULL;
      }
      str = (String*) malloc(sizeof(String));
      if (str == NULL){
        printf("Memória insuficiente\n");
        return NULL;
      }
      strcpy(str->s,c);
      str->tam = comp;
      return   str  ;
}
```

Testa o tamanho do char* de entrada

Alocação da estrutura String

Implementando String com Vetor

```
String*  str_copia(String* dest, String* orig ) {
     strcpy(dest->s,orig->s);
     dest->tam = orig->tam;
     return dest;
}

int  str_comprimento(String* str) {
     return str->tam;
}
```

Implementando String com Vetor

```
String*  str_concatena(String* dest, String* orig ) {

    if ((dest->tam + orig->tam) > N){
            printf("String origem muito grande");
            return NULL;
    }
    strcat(dest->s,orig->s);
    dest->tam = dest->tam + orig->tam;
    return dest;
}
```

Testa se as strings concatenadas cabem no vetor

Implementando String com Ponteiro

```
/* Implementação dada pelo arquivo  String.c */

#include <stdlib.h>
#include <stdio.h>
#include “String.h”
#include <string.h>

/* Estratégia 2 : Ponteiro */
    struct   string  {
        char* s ;
    };
```

Estrutura string possui um ponteiro para char

Implementando String com Ponteiro

```
String* str_cria(char* c){
      String *str; int comp;
      comp = strlen(c)+ 1;
      str = (String*) malloc(sizeof(String));
      if (str == NULL){
        printf("Memória insuficiente\n");
        return NULL;
      }
      str->s = (char*) malloc(comp * sizeof(char));
      if (str->s == NULL){
        printf("Memória insuficiente\n");
        return NULL;
      }
      strcpy(str->s,c);
      return   str  ;
}
```

Alocação da estrutura String

Alocação para armazenar um vetor de caracteres (endereço inicial guardado em um ponteiro)

Implementando String com Ponteiro

```
String*  str_copia(String* dest, String* orig ) {
  int comp = strlen(orig->s);
  if (strlen(dest->s) < comp) {
    comp++;
    dest->s =(char*)realloc(dest->s,comp* sizeof(char));
    if (dest->s == NULL){
            printf("Memoria insuficiente");
            return NULL;
     }
  }
  strcpy(dest->s,orig->s);
  return dest;
}

int  str_comprimento(String* str) {
     return strlen(str->s);
}
```

Função realloc tenta alocar mais espaço devolvendo endereço inicial original. Caso não consiga devolve um novo endereço, mas conteúdo original é copiado

Implementando String com Ponteiro

```
String*  str_concatena(String* dest, String* orig ) {
    int compDest = strlen(dest->s);
    int compOrig = strlen(orig->s);
    int comp = compDest + compOrig + 1;

    dest->s =(char*)realloc(dest->s,comp * sizeof(char));

    if (dest->s == NULL){
            printf("Memoria insuficiente");
            return NULL;
     }
     strcat(dest->s,orig->s);
     return dest;
}
```

Exemplo de TAD: Conta

```
/* Interface dada pelo arquivo conta.h */
/* TAD: conta bancária */
typedef    struct   conta   Conta ;
/* Funções exportadas */
Conta*  cont_cria(char* agencia,char* numero) ;
void  cont_libera(Conta *cont);
float  cont_saldo(Conta* cont);
void  cont_deposita(Conta* cont,float valor);
int   cont_saque(Conta* cont, float valor );
int  cont_transfere(Conta* origem,Conta* destino,
                    float valor) ;
```

Implementando o TAD Conta

- Usaremos duas estratégias para implementar o tipo Conta
  - Armazenando o saldo em uma estrutura
  - Armazenando o valor total dos saques e depósitos

Implementando o TAD Conta

```
/* Implementação dada pelo arquivo  conta.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include “conta.h”
/* Existem duas estratégias alternativas definidas a
seguir */

/* Estratégia 1 : Armazenar saldo em variável */
    struct   conta  {
          char  agencia[10] ;
          char  numero[15] ;
          float  saldo ;
    }  ;
```

Implementando o TAD Conta

```
/* Estratégia 1 : Armazenar saldo em variável */
Conta*   cont_cria(char* agencia,char* numero){
      Conta  *cont   ;
      cont = (Conta*)  malloc(sizeof(Conta)) ;
      if (cont == NULL) {
        printf ( "Memória insuficiente\n" ) ;
        exit (1) ;
      }
      strcpy(cont->agencia,agencia)  ;
      strcpy(cont-> numero,numero) ;
      cont->saldo = 0;
      return  cont  ;
}
```

Implementando o TAD Conta

```
/* Estratégia 1 : Armazenar saldo em variável */
void   cont_libera  ( Conta* cont  ){
    free  (cont) ;
}

float  cont_saldo  ( Conta* cont ){
      return     cont-> saldo   ;
}

void cont_deposita (Conta* cont, float valor) {
     cont->saldo += valor;
}
```

Implementando o TAD Conta

```
/* Estratégia 1 : Armazenar saldo em variável */
int   cont_saque  ( Conta* cont, float valor  ){
  if (valor > cont->saldo){
      printf("Valor maior que saldo!");
      return 0;
    }
    cont->saldo - = valor;
    return 1;
}
```

Implementando o TAD Conta

```
/* Estratégia 1 : Armazenar saldo em variável */
int cont_transfere(Conta* origem,Conta* destino,
                   float valor ){
     if (cont_saque(origem,valor)) {
            cont_deposita(destino,valor);
            return 1
     } else
            return 0;
}
```

Implementando o TAD Conta

```
/* Implementação dada pelo arquivo  conta.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include “conta.h”

/* Estratégia 2 : Armazenar saques e depósitos em
variáveis */
    struct   conta  {
         char     agencia[10] ;
         char     numero[15] ;
         float  saque, deposito ;
    }  ;
```

Implementando o TAD Conta

```
/* Estratégia 2 : Armazenar saques e depósitos
em variáveis */
Conta*  cont_cria( char* agencia,  char*   numero  ){
      Conta  *cont   ;
      cont = (Conta*)  malloc(sizeof(Conta)) ;
      if (cont == NULL) {
         printf ( “Memória insuficiente\n” ) ;
         exit (1) ;
      }
     strcpy(cont->agencia,agencia)  ;
     strcpy(cont-> numero,numero) ;
     cont->saque = 0;
     cont->deposito = 0;
     return  cont  ;
}
```

Implementando o TAD Conta

```
/* Estratégia 2 : Armazenar saques e depósitos
em variáveis */
void  cont_libera (Conta* cont){
    free(cont) ;
}

float  cont_saldo  ( Conta* cont ){
      return (cont-> deposito – cont-> saque);
}

void cont_deposita (Conta* cont, float valor) {
     cont->deposito += valor;
}
```

Implementando o TAD Conta

```
/* Estratégia 2 : Armazenar saques e depósitos
em variáveis */
int   cont_saque(Conta* cont,float valor){
    if (valor > cont_saldo(cont)){
      printf("Valor maior que saldo!");
      return 0;
    }
    cont->saque + = saque;
    return 1;
}
```

Implementando o TAD Conta

```
/* Estratégia 2 : Armazenar saques e depósitos
em variáveis */

int cont_transfere(Conta* origem,Conta* destino,
                   float valor ){
     if (cont_saque(origem,valor)) {
            cont_deposita(destino,valor);
            return 1
     } else
            return 0;
}
```