BCC202 - Estruturas de Dados I

Aula 04: Tipos Abstratos de Dados (TADs)

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

Conteúdo

- **Introdução**
  - Módulos e Compilação em Separado
- **TAD**
  - Abstração
  - Definição
- **Exemplo de uma TAD**
  - Especificação
  - Implementação
  - Principal: main()
- **Fluxograma do desenvolvimento de um TAD**
- **Composição**
- **Bibliotecas**
- **Considerações Finais**
- **Bibliografia**
- **Exercícios**

Introdução

Módulos e Compilação em Separado

Um programa em C pode ser dividido em vários arquivos fontes (arquivos com extensão ".c" e ".h").

- Funções afins são agrupadas por arquivos.
- Um arquivo com funções que representam parte da implementação de um programa é chamado de módulo.
- A implementação de um programa pode ser composta por um ou mais módulos.

Módulos vs Compilação

No caso de um programa composto por vários módulos:

- Cada um dos módulos é compilado separadamente.
- Cada módulo compilado gera um arquivo objeto (extensão .o ou .obj)
- Após a compilação de todos os módulos, outra ferramenta, denominada ligador (linker), é usada para juntar todos os arquivos em um único arquivo executável.
- Durante a ligação dos objetos, os códigos objetos das funções da biblioteca padrão de C também são incluídos (e.g., stdio.h[1]).

Mas o que incluímos no arquivo com a função *main*(): stdio.h ou stdio.c? **Por quê?**

[1] http://www.cplusplus.com/reference/cstdio/

Especificando novos módulos

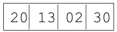
E se pudéssemos criar módulos específicos de acordo com o domínio de cada programa que implementamos?

- O que precisaríamos implementar?
- Como cada módulo seria "ligado" ao *main*()?
- O que precisaríamos definir?
- Quais serão as vantagens de criar módulos de acordo com o domínio/contexto da aplicação?
- E que tal separar especificação da implementação? O que teríamos em cada um desses elementos?

Exemplo Parcial

- Um programa para armazenar uma lista de inteiros.

Implementação por Vetor:




```c
#include "lista.h" /* lista_vetor.c */
void Insere(int x, Lista* L) {
  L->vetor[L->ultimo] = x;
  L->ultimo++;
}...
```

Implementação por Lista Encadeada:


```c
#include "lista.h" /* lista_encadeada.c */
void Insere(int x, Lista* L) {
  p = CriaNovaCelula(x);
  p->prox = L->primeiro->prox;
  L->primeiro->prox = p;
}...
```

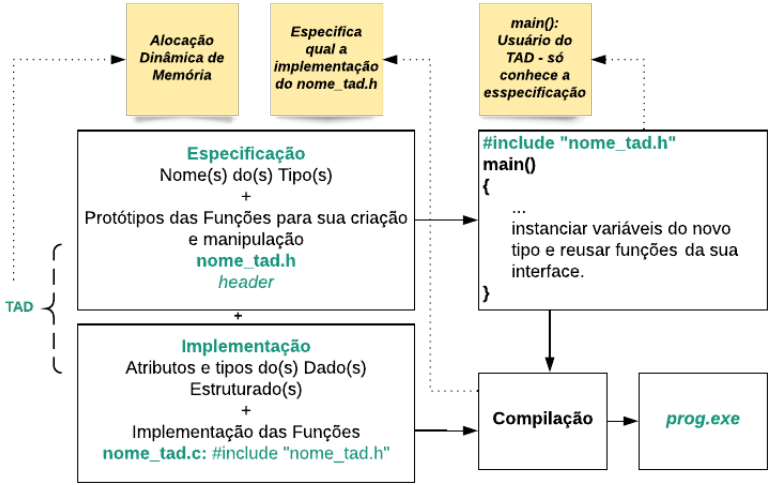
Programa do usuário de Lista.

```c
#include "lista.h"

int main() {
  Lista* L;
  /* chamada de função para
   * alocar o TAD Lista
   */
  int x;
  x = 20;
  Insere(x, L);
}
```

TAD

Ilustração dos principais elementos de um TAD.



Tipo Abstrato de Dados (TAD)

Abstração

"É a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais". [2]

Abstrato

"Abstraída a forma de implementação."

[2] https://pt.wikipedia.org/wiki/Abstracao

Mas o que é um TAD?

Definição

Especificação de um conjunto de dados mais funções que manipulam esses dados.

Em outras palavras

Especificação de um tipo estruturado mais funções para sua criação e manipulação.

Exemplo de uma TAD

Ponto no $\mathbb{R}^2$

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no $\mathbb{R}^2$. Para isso, devemos definir um tipo abstrato, que denominaremos de *Ponto*, e o conjunto de funções que operam sobre esse tipo, descritas a seguir.

- *cria*: operação que aloca dinamicamente memória para um ponto com coordenadas x e y;
- *libera*: operação que libera a memória alocada para um ponto;
- *acessa*: operação que devolve as coordenadas de um ponto;
- *atribui*: operação que atribui novos valores às coordenadas de um ponto;
- *distancia*: operação que calcula a distância entre dois pontos.

Ponto no $\mathbb{R}^2$: Definição da especificação do *Ponto*

ponto.h

```
1   /* TAD: Ponto (x,y) */
2   #ifndef ponto_h
3   #define ponto_h
4
5   /* Tipo exportado: somente o nome do NOVO TIPO */
6   typedef struct ponto Ponto; /*forward declaration*/
7
8   /* Funções exportadas */
9   Ponto* PontoCria (float x, float y);
10  void PontoLibera (Ponto** p);
11  void PontoAcessa (Ponto* p, float* x, float* y);
12  void PontoAtribui (Ponto* p, float x, float y);
13  float PontoDistancia (Ponto* p1, Ponto* p2);
14
15  #endif /* ponto_h */
```

Ponto no $\mathbb{R}^2$: Implementação da especificação do *Ponto*

ponto.c

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

/*definição dos dados e seus tipos*/
struct ponto {
    float x;
    float y;
};
```

Ponto no $\mathbb{R}^2$: Implementação da especificação do *Ponto*

ponto.c

```
12      Ponto* PontoCria (float x, float y) {
13          Ponto* p = (Ponto*) malloc(sizeof(Ponto));
14          if (p == NULL) {
15              printf("Memória insuficiente!\n");
16              exit(1);
17              // return NULL;
18          }
19          p->x = x;
20          p->y = y;
21          return p;
22      }
23
24      void PontoLibera (Ponto** p) {
25          free(*p);
26      }
27
```

Ponto no $\mathbb{R}^2$: Implementação da especificação do *Ponto*

ponto.c

```
28      void PontoAcessa (Ponto* p, float* x, float* y) {
29          *x = p->x;
30          *y = p->y;
31      }
32
33      void PontoAtribui (Ponto* p, float x, float y) {
34          p->x = x;
35          p->y = y;
36      }
37
38      float PontoDistancia (Ponto* p1, Ponto* p2) {
39          float dx = p2->x - p1->x;
40          float dy = p2->y - p1->y;
41          return sqrt(dx*dx + dy*dy);
42      }
43
```

Ponto no $\mathbb{R}^2$: Importando *Ponto* e suas funções

principal.c

```c
#include <stdio.h>
#include "ponto.h" /* incluindo a especificação */

int main(){
    Ponto* p1 = PontoCria(2.0, 4.0); /* alocando dinamicamente */
    Ponto* p2 = PontoCria(8.0, 16.0);
    float x, y; /* variáveis auxiliares */
    PontoAcessa(p1, &x, &y); /* manipulando o ponto p1 instanciado */
    printf("P1(%f, %f)\n", x, y);
    printf("Dist(P1 e P2): %f\n", PontoDistancia(p1, p2));
    PontoLibera(&p1);
    PontoLibera(&p2);
    return 0;
}
```

Ponto no $\mathbb{R}^2$: Compilando

Separadamente

```
1  gcc -c ponto.c -Wall
2  gcc -c main.c -Wall
3  gcc -o exe ponto.o main.o
```

"Tudo junto"

```
1  gcc -o exe *.c
```

Ponto no $\mathbb{R}^2$: Compilando

Separadamente

```
1  gcc -c ponto.c -Wall
2  gcc -c main.c -Wall
3  gcc -o exe ponto.o main.o
```

"Tudo junto" (não recomendado)

```
1  gcc -o exe *.c
```

Por que não é recomendado?

Recapitulando

A seguir, algumas observações acerca do novo tipo, *Ponto*, implementado:

- A especificação (ponto.h) define apenas:
  - o nome do novo tipo
  - os protótipos das funções para:
    - alocar e liberar memória dinamicamente
    - manipular os dados de *Ponto* (pelos menos para ler e escrever (*get*/*set*))
  - diretivas para pré-processamento (*#ifndef* /*#def* /*#endif*)
    - Definição do mesmo *.h* duas vezes.

Recapitulando

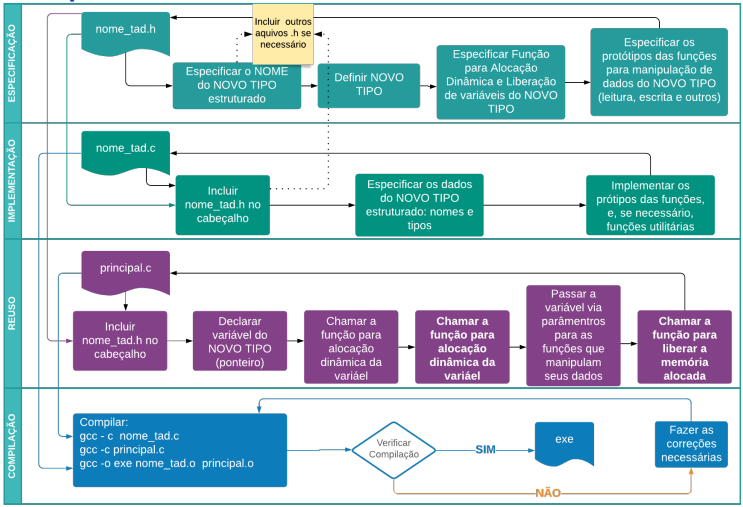
- A implementação (ponto.c):
  - inclui as bibliotecas padrões (e.g., para alocação dinamicamente de memória) e o ponto.h.
  - implementa TODOS os protótipos definidos em ponto.h.
  - pode implementar funções utilitárias, se necessário (não presentes no .h).

Recapitulando

- A implementação do *main* (principal.c):
  - inclui as bibliotecas padrões (e.g., para entrada e saída de dados) e o ponto.h.
  - necessariamente declara variáveis como sendo ponteiro para o novo tipo Ponto.
  - não tem acesso aos campos de Ponto
    - os campos estão definidos no arquivo *ponto.c*
    - *main*() contém #include "ponto.h" (não há campos da(s) *struct(s)* no *ponto.h*)
  - toda manipulação das variáveis do tipo *Ponto*$*$ são realizadas via funções declaradas no ponto.h
  - deve sempre chamar as funções para alocar e liberar memória dinamicamente.

Fluxograma do desenvolvimento de um TAD

Como especificar, implementar e reutilizar um TAD



Composição

Composição

Principais Conceitos

- Um TAD pode ser reutilizado para definir outros novos tipos.
- Criar dependências entre módulos.
- Em outras palavras:
  - Um TAD pode reutilizar zero, um ou mais TADs.
  - structs que "contém" outro(s) struct(s).

TAD Círculo

Criar um TAD Circulo com as seguintes operações:

- *cria*: aloca memória dinamicamente para um círculo com centro (x, y) e raio r.
- *libera*: libera a memória alocada para um círculo.
- *area*: calcula a área do círculo.
- *interior*: verifica se um dado ponto está dentro do círculo.

TAD Círculo

Criar um TAD Circulo com as seguintes operações:

- *cria*: aloca memória dinamicamente para um círculo com centro (x, y) e raio r.
- *libera*: libera a memória alocada para um círculo.
- *area*: calcula a área do círculo.
- *interior*: verifica se um dado ponto está dentro do círculo.

Por onde começar? Há tipos abstratos de dados que podemos reutilizar?

Especificação do TAD *Circulo*

circulo.h

```
1   /*TAD Circulo*/
2   #ifndef circulo_h
3   #define circulo_h
4
5   #include <stdio.h> /* dependência de módulos */
6   #include "ponto.h" /* tipo exportado/importado */
7
8   typedef struct circulo Circulo; /* forward declaration */
9
10  #define PI 3.14159 /* constante PI */
11
12  /* funções exportadas */
13  Circulo* CirculoCria(Ponto* centro, float raio);
14  void CirculoLibera(Circulo** circ);
15  float CirculoArea(Circulo *circ);
16  int CirculaInterior(Circulo *circ, Ponto* pt);
17
18  #endif /* circulo_h */
```

Implementação do TAD *Circulo*

circulo.c

```
1  #include <stdlib.h> /* malloc, free, exit */
2  #include <stdio.h> /* printf */
3  #include "circulo.h"
4
5  /* definição dados e seus tipos */
6  struct circulo{
7      Ponto* centro;
8      float raio;
9  };
```

Implementação do TAD *Circulo*

circulo.c

```
11      /* alternativa: receber (x,y) e raio: instanciar o ponto */
12      Circulo* CirculoCria(Ponto* centro, float raio) {
13          Circulo* circ = (Circulo*) malloc (sizeof(Circulo));
14          if (circ == NULL) {
15              printf("Memória insuficiente!\n");
16              exit(1);
17          }
18          circ->centro = centro;
19          circ->raio = raio;
20          return circ;
21      }
22
23      void CirculoLibera(Circulo** circ){
24          PontoLibera(&(*circ)->centro); /* primeiro libera a(s) parte(s) */
25          free(*circ); /* depois libera o todo */
26      }
```

Implementação do TAD *Circulo*

circulo.c

```
28      float CirculoArea(Circulo *circ) {
29          return PI * circ->raio * circ->raio;
30      }
31
32      int CirculoInterior(Circulo *circ, Ponto* pt) {
33          float d = PontoDistancia(circ->centro, pt);
34          return(d < (circ->raio)); /*1 se menor, 0 caso contrário*/
35      }
```

Importando o TAD *Circulo* e suas funções

principal_circ.c

```c
#include <stdio.h>
#include "circulo.h" /* basta incluir circulo.h ( contém ponto.h ) */

int main(){
  /* instanciando e manipulando variáveis do tipo Ponto */
  Ponto* p1 = PontoCria(8.0, 15.0);
  Ponto* p2 = PontoCria(8.0, 16.0);
  /* instanciando e manipulando variáveis do tipo Circulo */
  Circulo* circ = CirculoCria(p2, 4.9);
  printf("Area(cir): %f\n", CirculoArea(circ));
  if(CirculoInterior(circ, p1))
  printf("P1 está em Circ\n");
  CirculoLibera(&circ); /* liberando memória alocada */
  PontoLibera(&p1);
  /* p2 já foi liberado ao liberar o circ */
  return 0;
}
```

Compilando

Separadamente

```
1  gcc -c ponto.c -Wall
2  gcc -c circulo.c -Wall
3  gcc -c principal_circ.c -Wall
4  gcc -o exe ponto.o circulo.o principal_circ.o
```

Recapitulando

A seguir, algumas observações acerca do novo tipo, *Circulo*, implementado:

- circulo.h inclui ponto.h: há dependência de módulos
- circulo.c não tem acesso aos dados de *Ponto*, logo precisa reutilizar as funções para manipular variáveis do tipo *Ponto (ponteiro de Ponto) que estão em ponto.h
- principal_circ.c não tem acesso aos dados de *Ponto* e de *Circulo*, logo precisa utilizar as funções para manipulações presentes em ponto.h e circulo.h

Bibliotecas

Bibliotecas

Em algumas situações, não queremos definir novos tipos, mas apenas agrupar funções afins.

- Exemplo: para definir uma biblioteca com métodos de ordenação, teríamos.
  - ordenacao.h: conteria apenas os protótipos dos métodos de ordenação, sem definir novos tipos.
  - ordenacao.c: conteria apenas as implementações dos protótipos.
  - main.c (ou outro módulo dependente): teria o $\#include$ "*ordenacao.h*"

Considerações Finais

Motivação para Definição de TADs

- A ideia central é encapsular ou esconder de quem usa determinado tipo a forma concreta como foi implementado.
- Com isso, desacoplamos a implementação do uso:
  - Facilitando a manutenção e aumentando o potencial de reutilização do tipo criado.
- Agrupar tipos e funções com funcionalidades relacionadas - alta coesão.
- Usuário, *main*() ou mesmo outro módulo, só "enxerga" a interface, não a implementação - baixo acoplamento.

Vantagens da modularização

- Implementar um programa em vários módulos e estabelecer dependência entre eles favorece:
  - manutenção
  - reúso
  - corretude
  - legibilidade
  - encapsulamento

TAD vs Bibliotecas

- Exemplos de módulos:
  - TAD: novo(s) tipo(s) mais funções para manipulá-los - ao menos: criar, liberar, ler/escrever seus dados.
  - Biblioteca: funções afins
- Usuários, *main*() ou outro(s) módulo(s), incluem a especificação do módulo.

Análise de algoritmos.

Bibliografia

Bibliografia

Os conteúdos deste material, incluindo 4-tad/figs/, textos e códigos, foram extraídos ou adaptados do livro-texto indicado a seguir:


Celes, Waldemar and Cerqueira, Renato and Rangel, José
Introdução a Estruturas de Dados com Técnicas de Programação em C.
Elsevier Brasil, 2016.
ISBN 978-85-352-8345-7.

Exercícios

Exercício 1

Sobre o *TAD Circulo*, especifique e implemente funções para ler e atualizar os valores do ponto central e raio de um círculo (*get*/*set*).

Exercício 2

Considerando o *TAD Ponto*, implemente o *TAD Quadrado*. Seu TAD deve conter:

- *cria*: operação que aloca dinamicamente memória para dois pontos com coordenadas x e y: (x_{min}, y_{min}) e (x_{max}, y_{max});
- *libera*: operação que libera a memória alocada para os pontos do quadrado;
- *acessa*: operação que devolve as coordenadas dos dois pontos;
- *atribui*: operação que atribui novos valores às coordenadas dos dois pontos;
- *sobreposicao*: operação que retorna se dois quadrados se sobrepõe (1) ou não (0).