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@edu.ufop.br**

2021

Conteúdo

Conteúdo

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.

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

- 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:

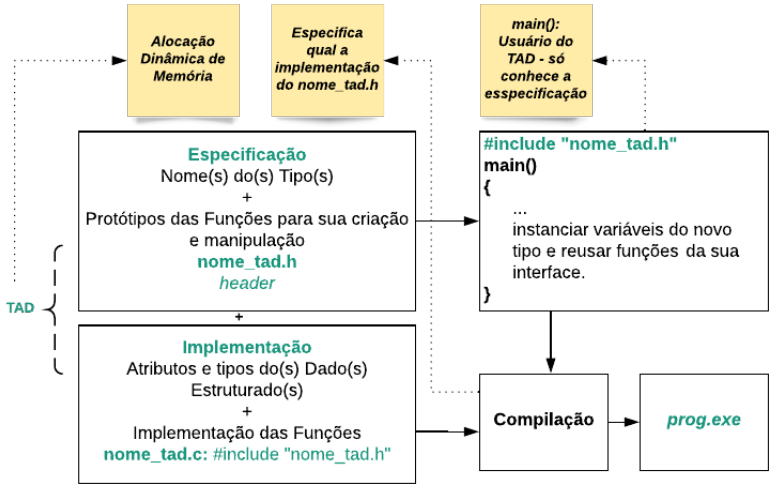

```c
#include "lista.h" /*lista_encadeada.c
    */
void Insere(int x, Lista* L) {
  p = CriaNovaCelula(x);
  L->primeiro= p;
}...
```

Programa do usuário de Lista.

```c
#include "Lista.h"
int main() {
  Lista* L;
  /*chamada de função para alocar o
    Lista* */
  int x;
  x = 20;
  Insere(x,L);
}
```

Conteúdo

A seguir, uma ilustração dos principais elementos de um 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

Definição de TAD (Tipo Abstrato de Dados)

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.

Conteúdo

Ponto no R^2

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no 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 R^2

Passo 1: Definição da especificação do *Ponto*.

ponto.h

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

Ponto no R^2

Passo 2: Implementação da especificação do *Ponto*.

ponto.c

```
1  #include <stdlib.h> /* malloc, free, exit */
2  #include <stdio.h> /* printf */
3  #include <math.h> /* sqrt */
4  #include "ponto.h"
5
6  /*definição dos dados e seus tipos*/
7  struct ponto {
8    float x;
9    float y;
10 };
```

Ponto no R^2

Passo 2: Implementação da especificação do *Ponto*.

ponto.c (*continuação i*)

```
1   Ponto* pto_cria (float x, float y) {
2       Ponto* p = (Ponto*) malloc(sizeof(Ponto));
3     if (p == NULL) {
4       printf("Memória insuficiente!\n");
5       exit(1);
6     }
7     p->x = x;
8     p->y = y;
9     return p;
10  }
11
12  void pto_libera (Ponto** p) {
13    free(*p);
14  }
15
```

Ponto no R^2

Passo 2: Implementação da especificação do *Ponto*.

ponto.c (*continuação ii*)

```
1   void pto_acessa (Ponto* p, float* x, float* y) {
2     *x = p->x;
3     *y = p->y;
4   }
5
6   void pto_atribui (Ponto* p, float x, float y) {
7     p->x = x;
8     p->y = y;
9   }
10
```

Ponto no R^2

Passo 2: Implementação da especificação do *Ponto*.

ponto.c (*continuação iii*)

```
1  float pto_distancia (Ponto* p1, Ponto* p2) {
2    float dx = p2->x - p1->x;
3    float dy = p2->y - p1->y;
4    return sqrt(dx*dx + dy*dy);
5  }
6
```

Ponto no R^2

Passo 3: Importando *Ponto* e suas funções.

principal.c

```
1   #include <stdio.h>
2   #include "ponto.h" /*incluindo a especificação*/
3   int main(){
4     Ponto* p1 = pto_cria(2.0,4.0);/*alocando dinamicamente*/
5     Ponto* p2 = pto_cria(8.0,16.0);
6     /*variáveis auxiliares*/
7     float x,y;
8     /*manipulando os pontos, p1 e p2, instanciados*/
9     pto_acessa(p1, &x, &y);
10    printf("Pi(%f,%f)\n", x, y);
11    printf("Dist(P1 e P2): %f\n", pto_distancia(p1,p2));
12    pto_libera(&p1);
13    pto_libera(p2);
14    return 0;
15  }
16
```

Ponto no R^2

Passo 4: Compilando

Separadamente

```
1  gcc -c ponto.c
2  gcc -c principal.c
3  gcc -o exe ponto.o  principal.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 (*net*/*set*))
  - diretivas para pré-processamento (*#ifndef* /*#def* /*#endif*)

Recapitulando

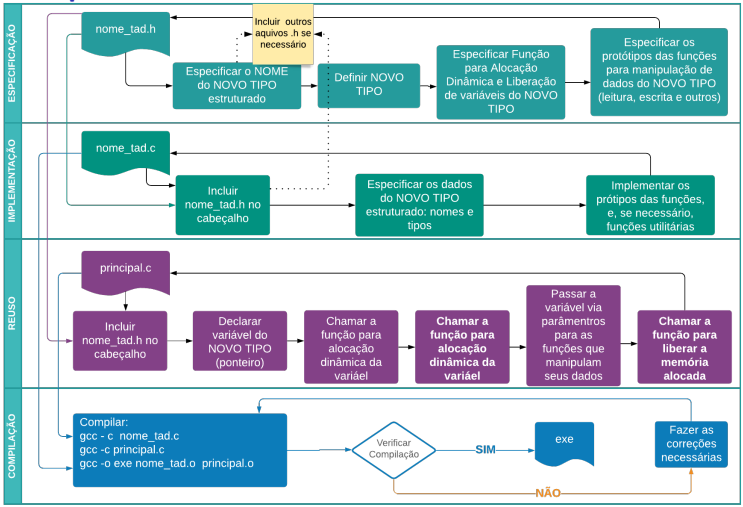
- A implementação, ponto.c:
  - inclui as bibliotecas padrões (e.g., para alocação dinamicamente de memória) e o Ponto.h.
  - implementa TODOS os protótipos definidos em ponto.h.
  - pode implementar funções utilitárias, se necessário.

Recapitulando

- A implementação do *main*(): principal.c:
  - inclui as bibliotecas padrões (e.g., para alocação dinamicamente de memória) 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.

Conteúdo

Como especificar, implementar e reutilizar um TAD



Conteúdo

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

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

Passo 1: Definição da especificação do *Circulo*.

circulo.h

```c
/*TAD Circulo*/
#ifndef circulo_h
#define circulo_h
  #include <stdio.h>
  /*dependência de módulos*/
  #include "ponto.h"
  /*tipo exportado*/
  typedef struct circulo Circulo; /*forward declaration*/

  #define PI 3.14159 /*constante PI*/

  /*funções exportadas*/
  Circulo* circ_cria(Ponto* centro, float raio);
  void circ_libera(Circulo** circ);
  float circ_area(Circulo *circ);
  int circ_interior(Circulo *circ, Ponto* pt);
#endif /* circulo_h */
```

Passo 2: Implementação da especificação do *Circulo*.

circulo.c

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

Passo 2: Implementação da especificação do *Circulo*.

circulo.c (*continuação i*)

```
1   /*alternativa: receber (x,y) e raio: instanciar o ponto*/
2   Circulo* circ_cria(Ponto* centro, float raio){
3     Circulo* circ = (Circulo*) malloc (sizeof(Circulo));
4     if (circ == NULL) {
5       printf("Memória insuficiente!\n");
6       exit(1);
7     }
8     circ->centro = centro;
9     circ->raio = raio;
10    return circ;
11  }
12
13  void circ_libera(Circulo** circ){
14    /*primeiro libera a(s) parte(s)*/
15    pto_libera((*circ)->centro);
16    /*depois libera o todo*/
17    free(*circ);
18  }
```

Passo 2: Implementação da especificação do *Circulo*.

circulo.c (*continuação ii*)

```
1  float circ_area(Circulo *circ){
2    return PI*circ->raio*circ->raio;
3  }
4
5  int circ_interior(Circulo *circ, Ponto* pt){
6    float d = pto_distancia(circ->centro, pt);
7    return(d < (circ->raio)); /*1 se menor, 0 caso contrário*/
8  }
```

Passo 3: Importando *Circulo* e suas funções.

principal_circ.c

```
1  #include <stdio.h>
2  /*basta incluir circulo.h ( contém ponto.h ) */
3  #include "circulo.h"
4  int main(){
5    /*instanciando e manipulando variáveis do tipo Ponto*/
6    Ponto* p1 = pto_cria(8.0,15.0);
7    Ponto* p2 = pto_cria(8.0,16.0);
8    /*instanciando e manipulando variáveis do tipo Circulo*/
9    Circulo* circ = circ_cria(p2, 4.9);
10   printf("Area(cir): %f\n", circ_area(circ));
11   if(circ_interior(circ,p1))
12   printf("P1 está em Circ\n");
13   /*liberando memória alocada*/
14   circ_libera(&circ);
15   pto_libera(p1);
16   /*p2 já foi liberado ao liberar o circ*/
17   return 0;
18 }
19
```

Passo 4: Compilando

Separadamente

```
1  gcc -c  ponto.c
2  gcc -c circulo.c
3  gcc -c principal_circ.c
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 ponto.h
- principal_circ.c não tem acesso aos dados de *Ponto* e de *Circulo*, logo precisa reutilizar as funções para manipulações presentes em ponto.h e circulo.h

Conteúdo

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"*

Conteúdo

Motivação para Definição de TADs

- A idéia 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, *man*() ou mesmo outro módulo, só "enxerga"a interface, não a implementação - baixo acoplamento.

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

- 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.

Conteúdo

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.

Conteúdo

Exercício 1

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