MAT A40 - Estrutura de Dados e Algoritmos I

Dr. George Lima
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade Federal da Bahia

Módulo 2

Tipos abstratos de dados e recursos da linguagem C

O que é um TAD?

Tipo de dados vs. Tipos abstrato de dados

- Tipo de dados
  - Variáveis e constantes geralmente são *tipadas* em linguagens de programação, ex.: em C/C++, `float`, `int`, `char`.
  - Definem domínio de valores e espaço em memória.
- Tipos abstrato de dados (TAD)
  - Abstração matemática útil à modelagem e estruturação de programas
  - Definem a modelo e suas operações sem associação com implementação, ex.: ponto num plano (coordenadas, distância à origem), estudante (nome, matrícula, inclusão, exclusão etc).

TAD

Exemplo: ponto num plano

- Atributos: coordenadas (x, y)
- Operação: distância à origem $(0, 0)$

Sem uso de TAD:

```
1
2  float x, y;
3  float dist(float x, float y) {
4   return sqrt(x*x+y*y);
5  }
```

Com uso de TAD:

```
1
2  struct point {
3      float x, y;
4   } p;
5
6  float dist(struct point p) {
7   return sqrt(p.x*p.x+p.y*p.y);
8  }
```

TAD em C/C++

Alguns aspectos da linguagem C/C++ são importantes para implementação de TAD.

- Estruturas (`struct`): Para encapsular atributos numa única entidade. C++ oferece ainda a possibilidade de encapsular as operações sobre tais atributos, com o uso de `class`, que define tipos de *objetos*.
- Definição de tipos (`typedef`): Definição de novos tipos fornece ao código maior clareza, facilitando o entendimento e a manutenção.
- Ponteiros (`*`): C/C++ oferece a possibilidade de definir variáveis para armazenar endereços de outras. Tais variáveis são chamadas de ponteiros.
- Alocação dinâmica de memória (`malloc`): Espaços de memória podem ser requisitados em tempo de execução em C através da função `malloc`. C++ também oferece `new`.

Encapsulamento de atributos: `struct`

```
1  struct point {
2      float x, y;
3  };
4
5  struct segment {
6    struct point p1, p2;
7  };
8
9  float segment_size(struct segment s) {
10   return( sqrt( (s.p1.x-s.p2.x)*(s.p1.x-s.p2.x)
11         + (s.p1.y-s.p2.y)*(s.p1.y-s.p2.y) ) );
12
13 }
14
15 int main() {
16   struct segment r;
17
18   r.p1.x = 3; r.p1.y = 5; r.p2.x = 6; r.p2.y = 9;
19   printf("\n %f: ",segment_size(r));
20  }
```

Definição de tipos: typedef

```c
typedef struct point {
    float x, y;
} point_t;

typedef struct segment {
  point_t p1, p2;
} segment_t;

float segment_size(struct segment s) {
  return( sqrt( (s.p1.x-s.p2.x)*(s.p1.x-s.p2.x)
       + (s.p1.y-s.p2.y)*(s.p1.y-s.p2.y) ) );
}

int main() {
  segment_t r;

  r.p1.x = 3; r.p1.y = 5; r.p2.x = 6; r.p2.y = 9;
  printf("\n %f: ",segment_size(r));
 }
```

Exercícios de fixação

Estenda o exemplo anterior:

1. Inclua uma função para verificar se dois segmentos de retas são paralelos, retornando `TRUE` ou `FALSE`. Para tanto, defina o tipo `boolean_t`. Utilize `enum` para definir os possíveis valores deste tipo.
2. Defina uma entidade `square` e um novo tipo a ela associado, `square_t`, composta de um vetor `p` com quatro pontos, que representam os vértices do quadrado. Crie uma função nomeada `min_square`, que recebe dois quadrados como parâmetro e retorna o quadrado de menor área.

Referência à memória: ponteiro

```
1  int main() {
2    int i = 3, j, *pi = &i, v[5] = {1,2,3,4,5};
3    struct { int a; int b; } s, *sp = &s;
4
5    s.a = 5; s.b = 10;
6
7    printf("\n (a) i= %d  s.a = %d s.b= %d",i,s.a,s.b);
8    for(j = 0; j<5; j++) printf("\n v[%d] = %d",j,v[j]);
9
10   *pi = 5;  sp->a = 1; sp->b = 2;
11   for(j = 0; j<5; j++) *(v+j) = 0;
12
13   printf("\n (b) i= %d  s.a = %d s.b= %d",i,s.a,s.b);
14   for(j = 0; j<5; j++) printf("\n v[%d] = %d",j,v[j]);
15
16   pi = v;
17   for(j = 0; j<5; j++) *(pi++) = 10;
18   printf("\n (c)");
19   for(j = 0; j<5; j++) printf("\n v[%d] = %d",j,v[j]);
20 }
```

Referência à memória: ponteiro e alocação dinâmica de memória

```
1  int main() {
2   int j,*pv, *pm[2];
3
4    pv = malloc(5*sizeof(int));
5    for (j = 0; j < 5; j++) pv[j] = j*2;
6    *pm = pv;
7
8    pv = malloc(10*sizeof(int));
9    for (j = 0; j < 10; j++) pv[j] = j*3;
10   *(pm+1) = pv;
11
12   printf("\n (a) ");
13   for(j = 0; j<5; j++)
14     printf("\n v[%d] = %d",j,*(*pm+j));
15
16   printf("\n (b) ");
17   for(j = 0; j<10; j++)
18     printf("\n v[%d] = %d",j,*(*(pm+1)+j));
19 }
```

Informações relevantes sobre alocação de memória

- Após alocar memória em tempo de execução, deve-se testar se a alocação foi bem sucedida:

```
1 ....
2 int *ptr;
3 ptr = malloc(sizeof(int));
4 if (ptr == 0) { printf("\n *** ERRO *** \n");
5   return 1;
6 }
```

- Em C, as seguintes funções são usadas para alocar e liberação memória:

```
1 void *malloc(size_t size);
2 void free(void *ptr);
3 void *calloc(size_t nmemb, size_t size);
4 void *realloc(void *ptr, size_t size);
```

- Memória alocada deve ser devolvida quando não é mais necessária: `free(ptr)`.
- O uso de `calloc` é geralmente mais seguro que `malloc`.

Exercícios de fixação

1. Explique o trecho de código abaixo, comparando-o com o apresentado anteriormente.

```
1    int **M, m=5,n=10;
2    M = malloc (m * sizeof (int *));
3    for (int i = 0; i < m; ++i)
4       M[i] = malloc (n * sizeof (int));
```

2. Substitua `malloc` por `calloc` e certifique-se que alocações de memória não falharam.
3. Escreva um código para desalocar a memória alocada na linha 2 do código acima.
4. Escreva uma função para alocar memória em tempo de execução para armazenar n pontos (tipo `point_t`). Esta função deve retornar o endereço inicial da memória alocada. Todos os n pontos devem ser inicialmente a origem $(0, 0)$.