Insper

12 - Tipos abstratos de dados

Sistemas Hardware-Software - 2020/1

Igor Montagner

Parte 1 - revisando TADs

Vamos considerar primeiro uma estrutura usada para guardar um ponto 2D. Este tipo de estrutura seria útil ao trabalhar com algoritmos de Geometria Computacional ou mesmo em aplicações de física.

```
typedef struct {
    double x, y;
} Point2D;
```

Vamos listar algumas operações que podem ser feitas com um ponto:

1. Inicialização e finalização - todo ponto deve ser inicializado com algum valor para x e y.
2. Somar dois pontos (e obter um terceiro);
3. Calcular o coeficiente angular de uma reta que passe pelos dois pontos;
4. Multiplicar ambas as coordenadas de um ponto (recebendo um novo em troca) - esta operação equivale a mudanças de escala
5. Retornar os valores das componentes x e y do ponto;

A ideia de um *Tipo Abstrato de Dados* é formalizar um "contrato" que lista quais operações podem ser feitas com este dado. Estas operações não dependem de nenhuma implementação em particular do tipo. Por exemplo, declarar o ponto com contendo um `double coords[2]` não muda os resultados de nenhuma das operações acima mas mudaria o código de acesso a coordenada `x` (`p.x` vs `p.coords[0]`). Veja um exemplo concreto de como fazer isto abaixo (arquivo *point2d.h*).

```
#ifndef __POINT2D__
#define __POINT2D__

struct _p;
typedef struct _p Point2D;

Point2D *point2D_new(double x, double y);
void point2D_destroy(Point2D *p);

double point2D_get_x(Point2D *p);
double point2D_get_y(Point2D *p);

Point2D *point2D_add(Point2D *p1, Point2D *p2);
double point2D_theta(Point2D *p1, Point2D *p2);
Point2D *point2D_scale(Point2D *p, double s);


#endif
```

Ponteiros opacos

Note que no exemplo acima a definição do `struct Point2D` não está inclusa no arquivo `point2d.h`! Por conta de ponteiro se um tipo de dado em *C*, podemos declarar ponteiros para `Point2D` e passá-los para as funções de nosso *TAD mesmo sem saber a definição exata de onde eles apontam!*. Só não podemos tentar acessar *a variável apontada* por este ponteiros (usando `*` ou `->`). Por isso, **alocação dinâmica de memória**

é essencial em *TAD*s: ela permite que todos os detalhes da implementação interna estejam **encapsulados** e que só possamos interagir com o tipo via as funções definidas para isso.

Exercício 1: Abra o arquivo `teste_point2d.c`. Você consegue entender seu conteúdo?

Exercício 2: Compile o arquivo `teste_point2d.c` usando a seguinte linha de comando. Rode-o logo em seguida. O que significa sua saída?

```
$ gcc -Og -Wall -g teste_point2d.c point2d.c -o teste_ponto
```

Exercício 3: Abra o arquivo `point2d.c` e complete as partes faltantes. Verifique se tudo funciona corretamente usando `teste_point2d.c`. Você deve aproveitar ao máximo as funções já criadas (ou seja, pode usar `point2d_new` nas outras funções).

Entrega: agora que sua implementação do *TAD* `Point2D` está completa, compile o arquivo `teste_point2d.c` e execute-o usando o *Valgrind*. Se seu programa rodar sem erros, entregue-o no Blackboard.

Parte 2 - vetores dinâmicos

Agora que já vimos vetores dinâmicos em aula, vamos implementá-los. Nosso vetor tem a seguinte interface:

```c
#ifndef __VECINT_H__
#define __VECINT_H__

struct _vec_int;

typedef struct _vec_int vec_int;

vec_int *vec_int_create();
void vec_int_destroy(vec_int **v);

int vec_int_size(vec_int *v);

/* As seguinte operações devolvem
 * 1 se pos é uma posição válida e a operação foi bem sucedida
 * 0 caso contrário
 *
 * No caso de at, o valor é retornado na variável apontada por vi.
 */
int vec_int_at(vec_int *v, int pos, int *vi);
int vec_int_insert(vec_int *v, int pos, int val);
int vec_int_remove(vec_int *v, int pos);

#endif
```

Entrega: implemente a estrutura *vetor dinâmico* no arquivo *vec_int.c* e compile o programa de testes da mesma maneira que foi feito na parte anterior. Para sua entrega estar 100% seu programa de testes deverá rodar sem erros no *valgrind*.