Insper

12 - Tipos abstratos de dados

Sistemas Hardware-Software - 2019/1

Igor Montagner

Parte 1 - o tipo `Point2D`

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

```
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. Ler 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 são abstratas e não dependem de nenhuma implementação específica 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
```

Note que no exemplo acima não permitimos no contrato que o usuário do nosso ponto mude suas coordenadas. Mais ainda, a definição de um ponto não está nem inclusa no arquivo! Podemos declarar ponteiros para `Point2D` (pois eles são, em essência, endereços de tamanho 8 bytes) e passá-los para as funções de nosso *TAD* mas não podemos mexer nos campos internos do `Point2D`! 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.

Exercício 4: Os testes escritos em `teste_point2d.c` lembram código escrito em linguagens de mais alto nível, como Java. Uma grande diferença é que toda função começa com `point2D_`. Pesquise por que isto é necessário e explique abaixo.

Parte 2 - um vetor que se autoredimensiona

Como nossa estrutura `Point2D` era muito simples, iremos agora trabalhar com um tipo de dado clássico: *Vetor*. Diferentemente do array de tamanho fixo que podemos criar em *C* com `malloc`, o *Vetor* suporta a inserção e remoção de itens sem limitações de tamanho. Ou seja, ele é capaz de autoredimensionar o espaço de memória em que armazena seus dados.

Para que isto seja possível o nosso tipo *Vetor* guardará os dados em um *array* `data` alocado dinâmicamente de tamanho `capacity`. O número de elementos **ocupados** dentro deste array é o `size` do vetor.

1. Se o array `data` encher, realoque-o para um novo array `data` de tamanho `capacity*2`. O array começa com `capacity = 1;`
2. Se o array `data` estiver com menos de um quarto dos elementos preenchidos, realoque-o com tamanho `capacity/2`.

Exercício 1: abra o arquivo `teste_vec_int.c`. A partir deste arquivo, você conseguiria aprender a usar o tipo `vec_int`?

Exercício 2: Comece a implementação do `vec_int` no arquivo `vec_int.c`. Cheque se sua implementação usando o `valgrind` e os testes em `teste_vec_int.c`.