Notas de aula 28/03

Na aula passada criamos um TAD bem simples chamado Ponto, usando um único arquivo chamado ex_tad_ponto.c:

```c
#include <stdio.h>
#include <math.h>

struct ponto {
	float x;
	float y;
};
typedef struct ponto Ponto;

Ponto cria_ponto(float x, float y) {
	Ponto p;
	p.x = x;
	p.y = y;
	return p;
}

void imprime_ponto(Ponto p){
	printf("%.2f, %.2f\n", p.x, p.y);
}

float distancia(Ponto p1, Ponto p2) {
	float dx, dy, dt;
	dx = p1.x - p2.x;
	dy = p1.y - p2.y;
	dt = sqrt(pow(dx, 2) + pow(dy, 2));
	return dt;
}

main() {
	Ponto p1, p2;
	float d;

	p1 = cria_ponto(10, 20);
	p2 = cria_ponto(10, 30);
	imprime_ponto(p1);
	imprime_ponto(p2);

	d = distancia(p1, p2);
	printf("%.2f", d);
}
```

Vimos que por convenção um TAD é criado usando arquivos separados.

- Arquivo.h: typedefs, protótipos de funções, variáveis globais…
- Arquivo.c: structs e funções.

Sendo assim, vamos pegar o TAD feito anteriormente e separá-lo usando essa convenção. Faremos isso bem passo-a-passo para um melhor entendimento.

1) Vamos criar um arquivo chamado Ponto.h, contendo o typedef para renomear a struct ponto e também os protótipos das funções usadas:

```
typedef struct ponto Ponto;

Ponto cria_ponto(float x, float y);
void imprime_ponto(Ponto p);
float distancia(Ponto p1, Ponto p2);
```

2) Vamos criar um arquivo chamado Ponto.c (o arquivo deve ter o mesmo nome do arquivo anterior, só mudando a extensão). Nele colocaremos a estrutura e as funções que interagem com ela. Também vamos tirar do arquivo ex_tad_ponto.c a inclusão das bibliotecas usadas pelas funções (como a math.h) e vamos colocá-las aqui. Vamos adicionar também o arquivo que criamos, Ponto.h. Diferente das bibliotecas que já vem com a linguagem, este arquivo que foi criado por nós, não vai estar entre < >. Ele deve estar entre “ “.

```
#include <stdio.h>
#include <math.h>
#include "Ponto.h"

struct ponto {
	float x;
	float y;
};

Ponto cria_ponto(float x, float y) {
	Ponto p;
	p.x = x;
	p.y = y;
	return p;
}

void imprime_ponto(Ponto p){
	printf("%.2f, %.2f\n", p.x, p.y);
}

float distancia(Ponto p1, Ponto p2) {
	float dx, dy, dt;
	dx = p1.x - p2.x;
	dy = p1.y - p2.y;
```

```
        dt = sqrt(pow(dx, 2) + pow(dy, 2));
        return dt;
}
```

3) No arquivo ex_tad_ponto.c deve ter sobrado apenas a função main. Adicionaremos também nesse arquivo a inclusão do arquivo Ponto.h que criamos. Ele deve ter ficado assim:

```
#include <stdio.h>
#include "Ponto.h"

main() {
        Ponto p1, p2;
        float d;

        p1 = cria_ponto(10, 20);
        p2 = cria_ponto(10, 30);
        imprime_ponto(p1);
        imprime_ponto(p2);

        d = distancia(p1, p2);
        printf("%.2f", d);

}
```

4) Primeiramente vamos compilar o arquivo Ponto.c usando o comando:
gcc -c Ponto.c
Depois vamos tentar compilar o arquivo ex_tad_ponto.c usando o comando:
gcc -c ex_tad_ponto.c

Ao fazer isso devemos ser notificados de alguns erros, sendo os dois primeiros:

```
usa_tads_ponto.c:5:11: error: storage size of ‘p1’ isn’t known
     Ponto p1, p2;
           ^~
usa_tads_ponto.c:5:15: error: storage size of ‘p2’ isn’t known
     Ponto p1, p2;
               ^
```

Isso acontece porque o arquivo ex_tad_ponto.c não conhece como a estrutura Ponto foi implementada, ou seja, não tem como saber quanto espaço uma variável do tipo Ponto precisa. Essa é justamente a ideia que vimos quando falamos em encapsulamento: o usuário do TAD não deve conhecer ou ter acesso direto a estrutura, para que ele não possa fazer alterações inesperadas sobre ela. O que podemos fazer é permitir que o usuário acesse ponteiros para estrutura. Assim precisaremos fazer uma alteração em nosso código:

```c
#include <stdio.h>
#include "Ponto.h"

main() {
	Ponto *p1, *p2;
	float d;

	p1 = cria_ponto(10, 20);
	p2 = cria_ponto(10, 30);
	imprime_ponto(p1);
	imprime_ponto(p2);

	d = distancia(p1, p2);
	printf("%.2f", d);

}
```

Perceba que p1 e p2 agora são ponteiros do tipo Ponto. Se o programa for compilado novamente, os erros na declaração de p1 e p2 foram resolvidos, embora ainda existam outros erros.

Os próximos erros são:

```
usa_tads_ponto.c:8:10: error: invalid use of incomplete typedef ‘Ponto {aka stru
ct ponto}’
     p1 = cria_ponto(10, 20);
          ^~~~~~~~~~
usa_tads_ponto.c:9:10: error: invalid use of incomplete typedef ‘Ponto {aka stru
ct ponto}’
     p2 = cria_ponto(10, 30);
          ^~~~~~~~~~
```

Perceba que p1 recebe o retorno da função cria_ponto. A função cria ponto retorna um tipo Ponto, mas p1 espera um ponteiro do tipo Ponto. Assim, temos uma incompatibilidade de tipos, que pode ser resolvida alterando a função cria_ponto para que ela retorne um ponteiro.

```c
Ponto * cria_ponto(float x, float y) {
	Ponto *p = malloc(sizeof(Ponto));
	p->x = x;
	p->y = y;
	return p;
}
```

Atente para o uso da função malloc para alocar um endereço de memória para o ponteiro p. Se simplesmente fosse feito isso:

```
Ponto * cria_ponto(float x, float y) {
      Ponto *p;
      p->x = x; // (*p).x = x;
      p->y = y; // (*p).y = y;
      return p;
}
```

Teríamos um erro, pois p ainda não está inicializado, ou seja, ainda não está vinculado a nenhum endereço de memória: o computador estaria recebendo uma instrução para guardar um valor, mas não saberia onde guardar esse valor.

Como a função cria_ponto foi modificada, será necessário mudar seu protótipo em Ponto.h:

```
Ponto * cria_ponto(float x, float y);
```

Salve o arquivo Ponto.h, compile novamente o arquivo Ponto.c e tente compilar novamente o arquivo ex_tad_ponto.c. Perceba que os erros referentes a incompatibilidade de tipos foram resolvidos. Os próximos erros são:

```
usa_tads_ponto.c:10:19: error: type of formal parameter 1 is incomplete
     imprime_ponto(p1);
                   ^~
usa_tads_ponto.c:11:19: error: type of formal parameter 1 is incomplete
     imprime_ponto(p2);
                   ^~
```

A função imprime ponto espera como parâmetro uma variável do tipo Ponto, mas o que está sendo passado para ela em sua chamada é um ponteiro. No arquivo Ponto.c, altere a função que imprime o ponto para que receba um ponteiro:

```
void imprime_ponto(Ponto * p){
      printf("%.2f, %.2f\n", p->x, p->y);
}
```

No arquivo Ponto.h, atualize o protótipo da função:

```
void imprime_ponto(Ponto *p);
```

Salve o arquivo Ponto.h, compile novamente o arquivo Ponto.c e compile o arquivo ex_tad_ponto.c. Veja que os erros anteriores foram resolvidos. Entretanto, ainda temos dois erros:

```
usa_tads_ponto.c:13:19: error: type of formal parameter 1 is incomplete
     d = distancia(p1, p2);
                   ^~
usa_tads_ponto.c:13:23: error: type of formal parameter 2 is incomplete
     d = distancia(p1, p2);
                       ^~
```

É o mesmo caso do erro anterior: a função espera um tipo Ponto mas está recebendo um ponteiro do tipo Ponto. Assim, a função distancia precisa ser alterada para receber ponteiros:

```
float distancia(Ponto *p1, Ponto *p2) {
      float dx, dy, dt;
      dx = p1->x - p2->x;
      dy = p1->y - p2->y;
      dt = sqrt(pow(dx, 2) + pow(dy, 2));
      return dt;
}
```

Assim como o seu protótipo no arquivo Ponto.h:

```
float distancia(Ponto *p1, Ponto *p2);
```

Salve o arquivo Ponto.h, compile novamente o arquivo Ponto.c e compile novamente o arquivo ex_tad_ponto.c. O arquivo ex_tad_ponto.c agora é compilado sem erros.

A partir dos códigos fonte Ponto.c e ex_tad_ponto.c, é possível gerar o executável do programa (chamado de teste_tad nesse exemplo). Use o comando:

gcc -o teste_tad Ponto.o ex_tads_ponto.o

Você poderá perceber um erro com as seguintes informações:

```
Ponto.o: Na função "distancia":
Ponto.c:(.text+0xcc): referência não definida para "pow"
Ponto.c:(.text+0xe3): referência não definida para "pow"
Ponto.c:(.text+0xed): referência não definida para "sqrt"
collect2: error: ld returned 1 exit status
```

Tente agora o comando:

gcc -o teste_tad Ponto.o ex_tads_ponto.o -lm

O erro deve ser resolvido e você terá em sua pasta um arquivo chamado test_tad. Execute este arquivo pelo comando:

./teste_tad

Você deve ter uma saída semelhante a esta:

```
10.00, 20.00
10.00, 30.00
Distancia: 10.00
```

Para finalizar precisamos liberar p1 e p2. Como eles foram alocados dinamicamente usando a função malloc, eles precisam ser liberados manualmente. Vamos criar uma nova função no arquivo Ponto.c:

```
void libera_ponto(Ponto *p) {
      free(p);
```

```
}
```

No arquivo Ponto.h devemos adicionar o protótipo da nova função:

```
void libera_ponto(Ponto *p);
```

Ao final da função main, no arquivo ex_tad_ponto.c, adicione:

```
libera_ponto(p1);
libera_ponto(p2);
```

Salve o arquivo Ponto.h, compile novamente o arquivo Ponto.c e o arquivo ex_tad_ponto.c e então gere um novo executável. Não haverá alteração na saída do programa, mas agora temos a garantia que os endereços estão sendo liberados.

Considerações

Vimos que TADs prezam pelo encapsulamento dos dados, por isso o usuário nunca vai conseguir criar uma variável desta forma:

```
Ponto p;
```

Ele conseguirá apenas criar um ponteiro para um tipo Ponto:

```
Ponto *p;
```

Também não lhe será permitido atribuir valores diretamente para um tipo Ponto:

```
(*p).x = 10;
```

ou:

```
p->x = 10;
```

Ao tentar fazer isso irá ocorrer um erro. A atribuição só poderá ser feita através da função criar_ponto, por exemplo.

Perceba que você decidir mudar algo na estrutura ou otimizar o funcionamento de alguma função, somente o arquivo Ponto.c irá precisar ser recompilado. O arquivo onde estão as implementações do usuário do seu TAD não vai ser afetado.

Usando o CodeBlocks

Para que o CodeBlocks reconheça arquivos .h criados por nós, é necessário fazer algumas configurações, veja o tutorial:
https://www.learncpp.com/cpp-tutorial/a3-using-libraries-with-codeblocks/