MC-202
Curso de C — Parte 4

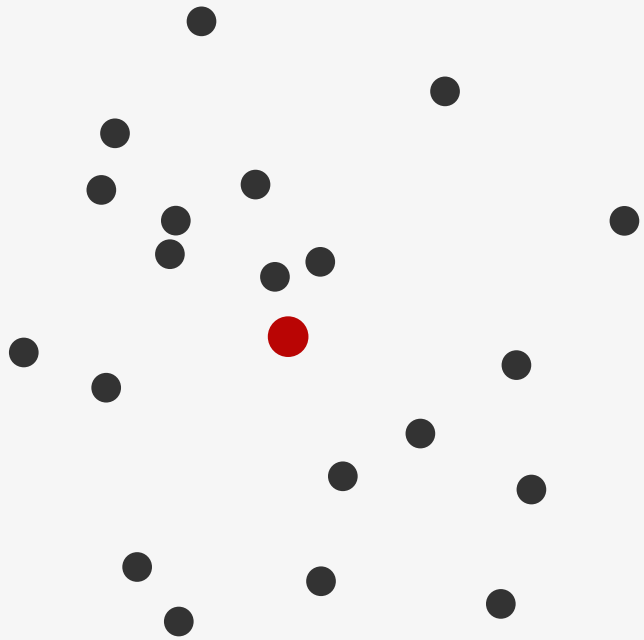
Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2023

Problema

Dado um conjunto de pontos do plano, como calcular o centroide?



```c
#include <stdio.h>
#define MAX 100

int main() {
  double x[MAX], y[MAX];
  double cx, cy;
  int i, n;
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    scanf("%lf %lf", &x[i], &y[i]);
  cx = cy = 0;
  for (i = 0; i < n; i++) {
    cx += x[i] / n;
    cy += y[i] / n;
  }
  printf("%lf %lf\n", cx, cy);
  return 0;
}
```

E se tivéssemos mais dimensões?

- Precisaríamos de um vetor para cada dimensão...

Registro

Registro é:

- uma coleção de variáveis relacionadas de vários tipos
- organizadas em uma única estrutura
- e referenciadas por um nome comum

registro
↳ escopo pra agrupar, mas não tem método, é só um agrupador

Características:

- Cada variável é chamada de membro do registro
- Cada membro é acessado por um nome na estrutura
- Cada estrutura define um novo tipo, com as mesmas características de um tipo padrão da linguagem

Não é uma classe! → Tipo primitivo de classe

- Não tem funções associadas
- C não é Orientada a Objetos como Python

Declaração de estruturas e registros

Declarando uma estrutura com N membros

```
1 struct identificador {
2   tipo1 membro1;
3   tipo2 membro2;
4   ...
5   tipoN membroN;
6 };
```

nome do identificador

Variáveis

Declarando um registro:

```
struct identificador nome_registro;
```

Em C:

- Declaramos um tipo de uma estrutura apenas uma vez
- Podemos declarar vários registros da mesma estrutura

Exemplo de estrutura

Ficha de dados cadastrais de um aluno

```
1  struct data {
2    int dia;
3    int mes;
4    int ano;
5  };
6
7  struct ficha_aluno {
8    int ra;
9    int telefone;
10   char nome[30];
11   char endereco[100];
12   struct data nascimento;
13 };
```

Dentro de uma struct pode diferentes tipos ex: int, char

Ou seja, podemos ter estruturas aninhadas

Usando um registro

Acessando um membro do registro

- `registro.membro`

Imprimindo o nome de um aluno

```
1 struct ficha_aluno aluno;
2 ...
3 printf("Aluno: %s\n", aluno.nome);
```

→ Acessar o campo com .nome

↳ Exemplo

Imprimindo o aniversário

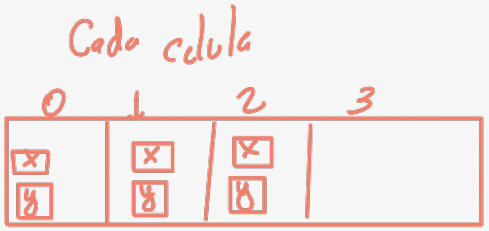
```
1 struct ficha_aluno aluno;
2 ...
3 printf("Aniversario: %d/%d\n", aluno.nascimento.dia,
4                                 aluno.nascimento.mes);
```

Copiando um aluno

```
1 aluno1 = aluno2;
```

→ Copiar 1 registro por outro

Centroide revisitado




```
1  #include <stdio.h>
2  #define MAX 100
3
4  struct ponto {
5    double x, y;
6  };
7
8  int main() {
9    struct ponto v[MAX], centroide;
10   int i, n;
11   scanf("%d", &n);
12   for (i = 0; i < n; i++)
13     scanf("%lf %lf", &v[i].x, &v[i].y);
14   centroide.x = 0;
15   centroide.y = 0;
16   for (i = 0; i < n; i++) {
17     centroide.x += v[i].x / n;
18     centroide.y += v[i].y / n;
19   }
20   printf("%lf %lf\n", centroide.x, centroide.y);
21   return 0;
22 }
```

A palavra-chave typedef

O `typedef` permite dar um novo nome para um tipo...

Exemplo: `typedef unsigned int u32;`

- Com isso, é possível declarar uma variável: `u32 x;`
- Escrever `unsigned int` ou `u32` é a mesma coisa

Vamos usar o `typedef` para dar nome para a `struct`

```
1 typedef struct identificador {
2   tipo1 membro1;
3   tipo2 membro2;
4   ...
5   tipoN membroN;
6 } novonome;
```

Com isso, ao invés de declarar uma variável dessa forma

- `struct identificador var;`

podemos declarar dessa forma

- `novonome var;`

Números Complexos

Vamos criar um programa que lida com números complexos

- Um número complexo é da forma $a + b\,i$
  - a e b são números reais
  - $i = \sqrt{-1}$ é a unidade imaginária

Queremos somar dois números complexos lidos e calcular o valor absoluto ($\sqrt{a^2 + b^2}$)

```
1  typedef struct {
2    double real;
3    double imag;
4  } Complexo;
5
6  int main() {
7    Complexo a, b, c;
8    scanf("%lf %lf", &a.real, &a.imag);
9    scanf("%lf %lf", &b.real, &b.imag);
10   c.real = a.real + b.real;
11   c.imag = a.imag + b.imag;
12   printf("%lf\n", sqrt(c.real * c.real + c.imag * c.imag));
13   return 0;
14 }
```

Maiúsculo por convenção (prof) → adotado por várias empresas

Ex: CadastroAluno

double

função minúsculo com underline

Ex: cadastro_aluno_novo

Raiz quadrada

Reflexão

Quando somamos 2 variáveis `float`:

- não nos preocupamos como a operação é feita
  - internamente o float é representado por um número binário
  - Ex: `0.3` é representado como `00111110100110011001100110011010`
- o compilador esconde os detalhes!

Codificação dos números em bits

E se quisermos lidar com números complexos?

- nos preocupamos com os detalhes

Será que também podemos abstrair um número complexo?

- Sim - usando registros e funções
- Faremos algo que se parece com uma classe

↳ Parece pois não é uma linguagem que possua mas fica quase igual

Números Complexos - Usando funções

```c
Complexo complexo(double real, double imag) {
  Complexo c;
  c.real = real;
  c.imag = imag;
  return c;
}

Complexo complexo_soma(Complexo a, Complexo b) {
  return complexo(a.real + b.real, a.imag + b.imag);
}

Complexo complexo_le() {
  Complexo a;
  scanf("%lf %lf", &a.real, &a.imag);
  return a;
}
```

DRY (Don't Repeat Yourself) vs. WET (Write Everything Twice)

- Funções permitem reutilizar código em vários lugares

Onde a função é usada, só é importante o seu resultado

- Não como o resultado é calculado...

Várias Funções Possíveis

```
1  Complexo complexo(double real, double imag);
2
3  Complexo complexo_le();
4  void complexo_imprime(Complexo a);
5
6  int complexos_iguais(Complexo a, Complexo b);
7
8  Complexo complexo_soma(Complexo a, Complexo b);
9  Complexo complexo_multiplicacao(Complexo a, Complexo b);
10
11 double complexo_absoluto(Complexo a);
12 Complexo complexo_conjugado(Complexo a);
```

E se quisermos usar números complexos em vários programas?

- basta copiar a struct e as funções...
- e se acharmos um bug ou quisermos mudar algo?
- Esse solução não é DRY...

↳ Por isso dá pra separar em módulos separ em partes (slide de baixo)

Ideia

Vamos quebrar o programa em três partes

1. Implementação das funções para os números complexos
   - Definem como calcular soma, absoluto, etc...
   - Chamamos de Implementação
2. Código que utiliza as funções de números complexos
   - Soma dois números complexos sem se importar como
   - Calcula o absoluto sem se importar como
   - mas precisa conhecer o protótipo das funções...
   - Chamamos de Cliente
3. Struct e protótipos das funções para números complexos
   - Declara o que o Cliente pode fazer
   - Declara o que precisa ser implementado
   - Chamamos de Interface

Interface e Implementação podem ser usadas em outros programas

Tipo Abstrato de Dados

Um TAD é um conjunto de valores associado a um conjunto de operações permitidas nesses dados

- Interface: conjunto de operações de um TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realizam as operações
  - A implementação é o único "lugar" que uma variável é acessada diretamente
- Cliente: código que utiliza/chama uma operação
  - O cliente nunca acessa a variável diretamente

Em C:

- um TAD é declarado como uma `struct`
- a interface é um conjunto de protótipos de funções que manipula a `struct`

Números Complexos - Interface

Criamos um arquivo `complexos.h` com a `struct` e os protótipos de função

```
1  #ifndef COMPLEXO_H
2  #define COMPLEXO_H
3
4  struct complexo { double real; double imag; };
5
6  typedef struct complexo Complexo;
7
8  Complexo complexo(double real, double imag);
9
10 Complexo complexo_le();
11 void complexo_imprime(Complexo a);
12
13 int complexos_iguais(Complexo a, Complexo b);
14
15 Complexo complexo_soma(Complexo a, Complexo b);
16 Complexo complexo_multiplicacao(Complexo a, Complexo b);
17
18 double complexo_absoluto(Complexo a);
19 Complexo complexo_conjugado(Complexo a);
20
21 #endif
```

Números Complexos - Implementação

Criamos um arquivo `complexos.c` com as implementações

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

Complexo complexo(double real, double imag) {
  Complexo c;
  c.real = real;
  c.imag = imag;
  return c;
}

Complexo complexo_soma(Complexo a, Complexo b) {
  return complexo(a.real + b.real, a.imag + b.imag);
}

Complexo complexo_le() {
  Complexo a;
  scanf("%lf %lf", &a.real, &a.imag);
  return a;
}
```

← bibliotecas usadas

← tem a definição da `struct`

Números Complexos - Exemplo de Cliente

E quando formos usar números complexos em nossos programas?

```
1  #include <stdio.h>
2  #include "complexos.h"
3
4  int main() {
5    Complexo a, b, c;
6    a = complexo_le();
7    b = complexo_le();
8    c = complexo_soma(a, b);
9    complexo_imprime(c);
10   printf("%lf\n", complexo_absoluto(c));
11   return 0;
12 }
```

tem a `struct` e as funções

Biblioteca do usuário, sem diretório do C, usar (" ") aspas duplas

Como compilar?

Temos três arquivos diferentes:

- `cliente.c` contém a função `main`
- `complexos.c` contém a implementação
- `complexos.h` contém a interface

Vamos compilar por partes:

- `gcc -std=c99 -Wall -Werror -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -c complexos.c`
  - vai gerar o arquivo compilado `complexos.o`
- `gcc cliente.o complexos.o -lm -o cliente`
  - faz a linkagem, gerando o executável `cliente`
  - adicionamos `cliente.o` e `complexos.o`
  - e outras bibliotecas, por exemplo, `-lm`

Makefile

É mais fácil usar um Makefile para compilar

```
1  all: cliente
2
3  cliente: cliente.o complexos.o
4    gcc cliente.o complexos.o -lm -o cliente
5
6  cliente.o: cliente.c complexos.h
7    gcc -std=c99 -Wall -Werror -c cliente.c
8
9  complexos.o: complexos.c complexos.h
10   gcc -std=c99 -Wall -Werror -c complexos.c
```

Basta executar `make` na pasta com os arquivos:

- `cliente.c`
- `complexos.c`
- `complexos.h`
- `Makefile`

Apenas recompila o que for necessário!

Vantagens do TAD

- Reutilizar o código em vários programas
  - `complexos.{c,h}` podem ser usados em outros lugares
  - permite criar bibliotecas de tipos úteis
    - ex: biblioteca de álgebra linear
- Código mais simples, claro e elegante
  - O cliente só se preocupa em usar funções
  - O TAD só se preocupa em disponibilizar funções
- Separa a implementação da interface
  - Podemos mudar a implementação sem quebrar clientes
  - Os resultados das funções precisam ser os mesmos
  - Mas permite fazer otimizações, por exemplo
  - Ou adicionar novas funções
- O código fica modular
  - Mais fácil colaborar com outros programadores
  - Arquivos menores com responsabilidade bem definida
- Permite disponibilizar apenas o `.h` e `.o`
  - Não precisa disponibilizar o código fonte da biblioteca

Como criar um TAD

Construímos o TAD definindo:

- Um nome para o tipo a ser usado
  - Ex: `complexo`
  - Uma `struct` com um `typedef`
- Quais funções ele deve responder
  - `soma`, `absoluto`, etc...
  - Considerando quais são as entradas e saídas
  - E o resultado esperado
  - Idealmente, cada função tem apenas uma responsabilidade

Ou seja, primeiro definimos a interface

- Basta então fazer uma possível implementação

Exercício - Conjunto de Inteiros

→ Até quinta não precisa entregar

Faça um TAD que representa um conjunto de inteiros e que suporte as operações mais comuns de conjunto como adição, união, interseção, etc.

Exercício - Matrizes → Desafio

Faça um TAD que representa uma matriz de reais e que suporte as operações mais comuns para matrizes como multiplicação, adição, etc.