

MC-202  
Curso de C - Parte 4

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

Universidade Estadual de Campinas

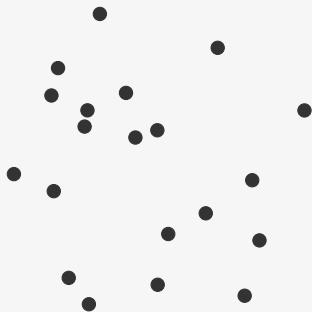
2º semestre/2020

## Problema

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

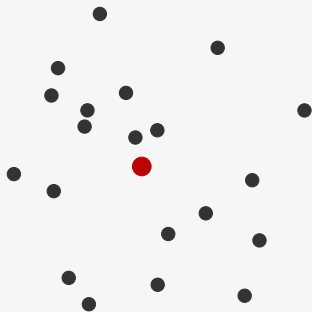
## Problema

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



## Problema

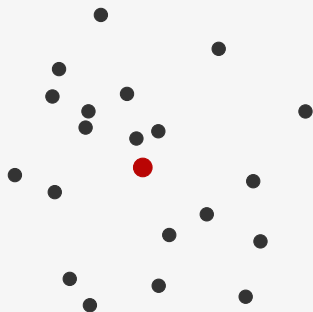
Dado um conjunto de pontos do plano, como calcular o centroide?



# Problema

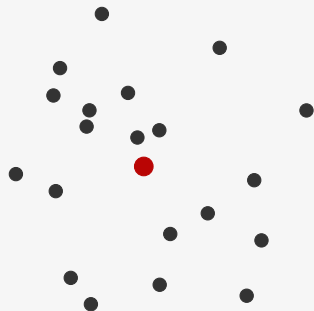
Dado um conjunto de pontos do plano, como calcular o centroide?

```
1 #include <stdio.h>
2 #define MAX 100
```



# Problema

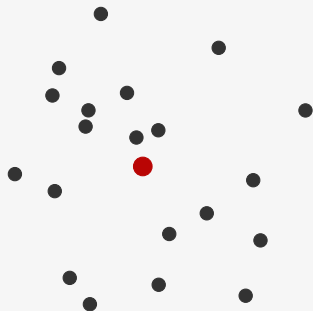
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
```

# Problema

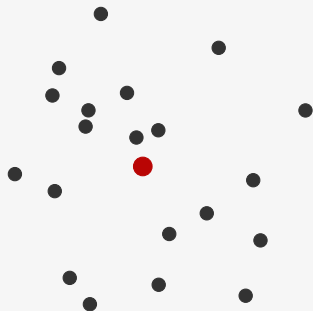
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
```

# Problema

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

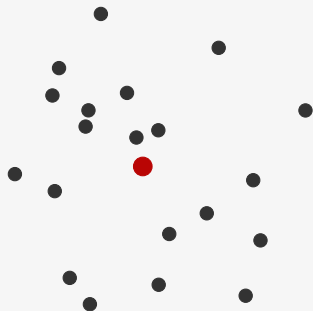


```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
```



# Problema

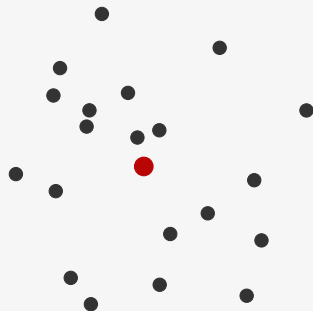
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
8     scanf("%d", &n);
```

# Problema

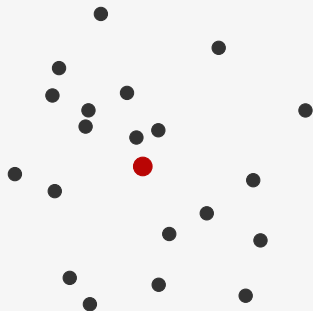
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
8     scanf("%d", &n);
9     for (i = 0; i < n; i++)
10        scanf("%lf %lf", &x[i], &y[i]);
```

# Problema

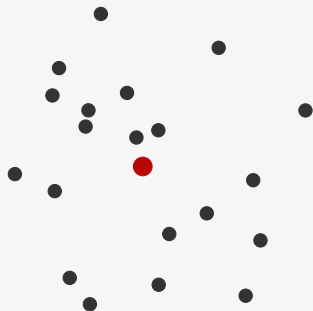
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
8     scanf("%d", &n);
9     for (i = 0; i < n; i++)
10         scanf("%lf %lf", &x[i], &y[i]);
11     cx = cy = 0;
```

# Problema

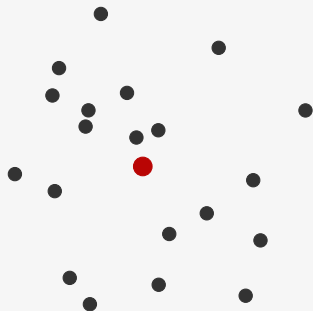
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
8     scanf("%d", &n);
9     for (i = 0; i < n; i++)
10         scanf("%lf %lf", &x[i], &y[i]);
11     cx = cy = 0;
12     for (i = 0; i < n; i++) {
```

# Problema

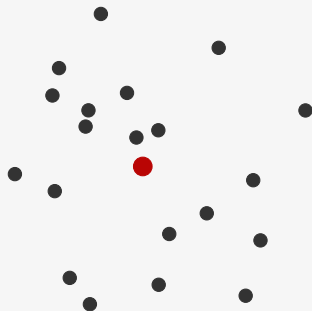
Dado um conjunto de pontos do plano, como calcular o centroide?



```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
8     scanf("%d", &n);
9     for (i = 0; i < n; i++)
10         scanf("%lf %lf", &x[i], &y[i]);
11     cx = cy = 0;
12     for (i = 0; i < n; i++) {
13         cx += x[i]/n;
14         cy += y[i]/n;
15     }
```

# Problema

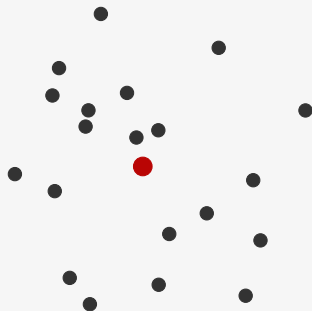
Dado um conjunto de pontos do plano, como calcular o centroide?



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

# Problema

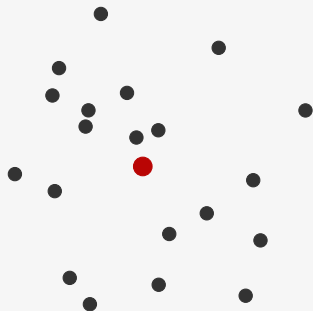
Dado um conjunto de pontos do plano, como calcular o centroide?



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

## Problema

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



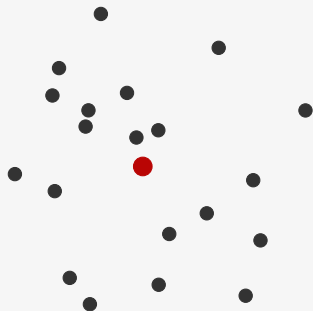
```
1 #include <stdio.h>
2 #define MAX 100
3
4 int main() {
5     double x[MAX], y[MAX];
6     double cx, cy;
7     int i, n;
8     scanf("%d", &n);
9     for (i = 0; i < n; i++)
10         scanf("%lf %lf", &x[i], &y[i]);
11     cx = cy = 0;
12     for (i = 0; i < n; i++) {
13         cx += x[i]/n;
14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

E se tivéssemos mais dimensões?



# Problema

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



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

E se tivéssemos mais dimensões?

- Precisaríamos de um vetor para cada dimensão...

# Registro

Registro é:

# Registro

Registro é:

- uma coleção de variáveis relacionadas de **vários** tipos

# Registro

Registro é:

- uma coleção de variáveis relacionadas de **vários** tipos
- organizadas em uma única estrutura

# Registro

Registro é:

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

# Registro

Registro é:

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

Características:

# Registro

Registro é:

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

Características:

- Cada variável é chamada de **membro** do registro

# Registro

Registro é:

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

Características:

- Cada variável é chamada de **membro** do registro
- Cada membro é acessado por um nome na estrutura



# Registro

Registro é:

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

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

# Registro

Registro é:

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

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!

# Registro

Registro é:

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

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!

- Não tem funções associadas

# Registro

Registro é:

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

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!

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

# Declaração de estruturas e registros

# Declaração de estruturas e registros

Declarando uma **estrutura** com  $N$  membros

# 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 };
```

# 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 };
```

Declarando **um registro**:



# 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 };
```

Declarando **um registro**:

```
struct identificador nome_registro;
```

# 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 };
```

Declarando **um registro**:

```
struct identificador nome_registro;
```

Em C:

# 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 };
```

Declarando **um registro**:

```
struct identificador nome_registro;
```

Em C:

- Declaramos um tipo de uma estrutura apenas uma vez

# 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 };
```

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

# 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 };
```

# 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 };
```

Ou seja, podemos ter estruturas **aninhadas**

# Usando um registro

Acessando um membro do registro



## Usando um registro

Acessando um membro do registro

- `registro.membro`

## Usando um registro

Acessando um membro do registro

- `registro.membro`

Imprimindo o nome de um aluno

# 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);
```

# 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);
```

Imprimindo o aniversário

# 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);
```

Imprimindo o aniversário

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

# 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);
```

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

# 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);
```

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;
```

## O comando typedef

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



## O comando typedef

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

Exemplo: `typedef int meu_int;`

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

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

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

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;
```

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

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

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

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;`

## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

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



## O comando typedef

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

Exemplo: `typedef int meu_int;`

- Com isso, é possível declarar uma variável: `meu_int x;`
- O tipo `int` continua existindo

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;`

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
```

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     ponto v[MAX], centroide;
```

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     ponto v[MAX], centroide;
10    int i, n;
11    scanf("%d", &n);
```

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     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);
```

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     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;
```

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     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    }
```

# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     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);
```



# Centroide revisitado

```
1 #include <stdio.h>
2 #define MAX 100
3
4 typedef struct ponto {
5     double x, y;
6 } ponto;
7
8 int main() {
9     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 }
```

# Números Complexos

Vamos criar um programa que lida com números complexos

# Números Complexos

Vamos criar um programa que lida com números complexos

- Um número complexo é da forma  $a + bi$

# Números Complexos

Vamos criar um programa que lida com números complexos

- Um número complexo é da forma  $a + bi$ 
  - $a$  e  $b$  são números reais

# Números Complexos

Vamos criar um programa que lida com números complexos

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

# Números Complexos

Vamos criar um programa que lida com números complexos

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

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

# Números Complexos

Vamos criar um programa que lida com números complexos

- Um número complexo é da forma  $a + bi$ 
  - $a$  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 }
```

# Reflexão

Quando somamos 2 variáveis `float`:



# Reflexão

Quando somamos 2 variáveis `float`:

- não nos preocupamos como a operação é feita

# Reflexão

Quando somamos 2 variáveis `float`:

- não nos preocupamos como a operação é feita
  - internamente o float é representado por um número binário

# 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`

# 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!

# 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!

E se quisermos lidar com números complexos?

# 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!

E se quisermos lidar com números complexos?

- nos preocupamos com os detalhes

# 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!

E se quisermos lidar com números complexos?

- nos preocupamos com os detalhes

Será que também podemos abstrair um número complexo?

# 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!

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



# 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!

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

## Números Complexos - Usando funções

```
1 complexo complexo_novo(double real, double imag) {
2     complexo c;
3     c.real = real;
4     c.imag = imag;
5     return c;
6 }
7
8 complexo complexo_soma(complexo a, complexo b) {
9     return complexo_novo(a.real + b.real, a.imag + b.imag);
10 }
11
12 complexo complexo_le() {
13     complexo a;
14     scanf("%lf %lf", &a.real, &a.imag);
15     return a;
16 }
```

## Números Complexos - Usando funções

```
1 complexo complexo_novo(double real, double imag) {
2     complexo c;
3     c.real = real;
4     c.imag = imag;
5     return c;
6 }
7
8 complexo complexo_soma(complexo a, complexo b) {
9     return complexo_novo(a.real + b.real, a.imag + b.imag);
10 }
11
12 complexo complexo_le() {
13     complexo a;
14     scanf("%lf %lf", &a.real, &a.imag);
15     return a;
16 }
```

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

## Números Complexos - Usando funções

```
1 complexo complexo_novo(double real, double imag) {
2     complexo c;
3     c.real = real;
4     c.imag = imag;
5     return c;
6 }
7
8 complexo complexo_soma(complexo a, complexo b) {
9     return complexo_novo(a.real + b.real, a.imag + b.imag);
10 }
11
12 complexo complexo_le() {
13     complexo a;
14     scanf("%lf %lf", &a.real, &a.imag);
15     return a;
16 }
```

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

- Funções permitem reutilizar código em vários lugares

## Números Complexos - Usando funções

```
1 complexo complexo_novo(double real, double imag) {
2     complexo c;
3     c.real = real;
4     c.imag = imag;
5     return c;
6 }
7
8 complexo complexo_soma(complexo a, complexo b) {
9     return complexo_novo(a.real + b.real, a.imag + b.imag);
10 }
11
12 complexo complexo_le() {
13     complexo a;
14     scanf("%lf %lf", &a.real, &a.imag);
15     return a;
16 }
```

**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

# Números Complexos - Usando funções

```
1 complexo complexo_novo(double real, double imag) {
2     complexo c;
3     c.real = real;
4     c.imag = imag;
5     return c;
6 }
7
8 complexo complexo_soma(complexo a, complexo b) {
9     return complexo_novo(a.real + b.real, a.imag + b.imag);
10 }
11
12 complexo complexo_le() {
13     complexo a;
14     scanf("%lf %lf", &a.real, &a.imag);
15     return a;
16 }
```

**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_novo(double real, double imag);
2
3 complexo complexo_soma(complexo a, complexo b);
4
5 double complexo_absoluto(complexo a);
6
7 complexo complexo_le();
8
9 void complexo_imprime(complexo a);
10
11 int complexos_iguais(complexo a, complexo b);
12
13 complexo complexo_multiplicacao(complexo a, complexo b);
14
15 complexo complexo_conjugado(complexo a);
```

## Várias Funções Possíveis

```
1 complexo complexo_novo(double real, double imag);
2
3 complexo complexo_soma(complexo a, complexo b);
4
5 double complexo_absoluto(complexo a);
6
7 complexo complexo_le();
8
9 void complexo_imprime(complexo a);
10
11 int complexos_iguais(complexo a, complexo b);
12
13 complexo complexo_multiplicacao(complexo a, complexo b);
14
15 complexo complexo_conjugado(complexo a);
```

E se quisermos usar números complexos em vários programas?



## Várias Funções Possíveis

```
1 complexo complexo_novo(double real, double imag);
2
3 complexo complexo_soma(complexo a, complexo b);
4
5 double complexo_absoluto(complexo a);
6
7 complexo complexo_le();
8
9 void complexo_imprime(complexo a);
10
11 int complexos_iguais(complexo a, complexo b);
12
13 complexo complexo_multiplicacao(complexo a, complexo b);
14
15 complexo complexo_conjugado(complexo a);
```

E se quisermos usar números complexos em vários programas?

- basta copiar a struct e as funções...

## Várias Funções Possíveis

```
1 complexo complexo_novo(double real, double imag);
2
3 complexo complexo_soma(complexo a, complexo b);
4
5 double complexo_absoluto(complexo a);
6
7 complexo complexo_le();
8
9 void complexo_imprime(complexo a);
10
11 int complexos_iguais(complexo a, complexo b);
12
13 complexo complexo_multiplicacao(complexo a, complexo b);
14
15 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?

## Várias Funções Possíveis

```
1 complexo complexo_novo(double real, double imag);
2
3 complexo complexo_soma(complexo a, complexo b);
4
5 double complexo_absoluto(complexo a);
6
7 complexo complexo_le();
8
9 void complexo_imprime(complexo a);
10
11 int complexos_iguais(complexo a, complexo b);
12
13 complexo complexo_multiplicacao(complexo a, complexo b);
14
15 complexo complexo_conjugado(complexo a);
```

E se quisermos usar números complexos em vários programas?

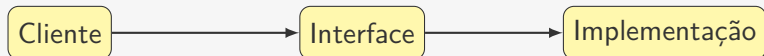
- basta copiar a struct e as funções...
- e se acharmos um bug ou quisermos mudar algo?
- Essa solução não é **DRY**...

## Ideia

Vamos quebrar o programa em três partes

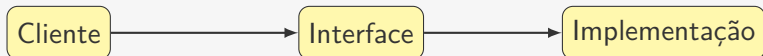
## Ideia

Vamos quebrar o programa em três partes



## Ideia

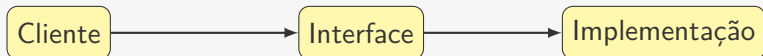
Vamos quebrar o programa em três partes



1. Implementação das funções para os números complexos

## Ideia

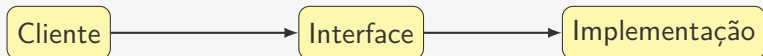
Vamos quebrar o programa em três partes



1. Implementação das funções para os números complexos
  - Definem como calcular soma, absoluto, etc...

## Ideia

Vamos quebrar o programa em três partes

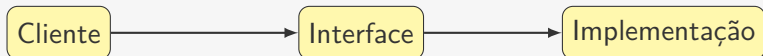


1. Implementação das funções para os números complexos
  - Definem como calcular soma, absoluto, etc...
  - Chamamos de **Implementação**



## Ideia

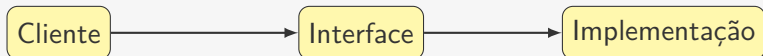
Vamos quebrar o programa em três partes



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

# Ideia

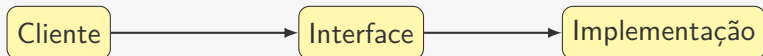
Vamos quebrar o programa em três partes



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

# Ideia

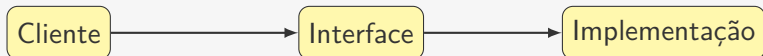
Vamos quebrar o programa em três partes



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

# Ideia

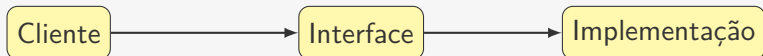
Vamos quebrar o programa em três partes



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...

# Ideia

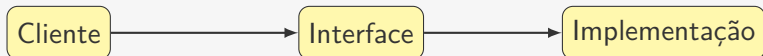
Vamos quebrar o programa em três partes



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**

# Ideia

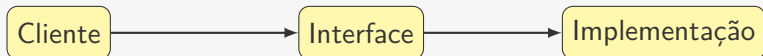
Vamos quebrar o programa em três partes



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

# Ideia

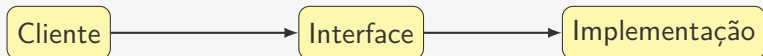
Vamos quebrar o programa em três partes



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
  - Define o que o Cliente pode fazer

# Ideia

Vamos quebrar o programa em três partes

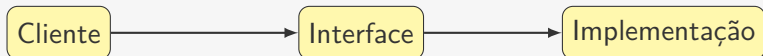


1. Implementação das funções para os números complexos
  - Definem como calcular soma, absoluto, etc...
  - Chamamos de **Implementação**
2. Código que utiliza as funções de números complexos
  - Soma dois números complexos sem se importar como
  - Calcula o absoluto sem se importar como
  - mas precisa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



# Ideia

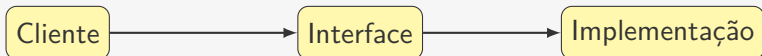
Vamos quebrar o programa em três partes



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
  - Define o que o Cliente pode fazer
  - Define o que precisa ser implementado
  - Chamamos de **Interface**

# 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
  - Define o que o Cliente pode fazer
  - Define o que precisa ser implementado
  - Chamamos de **Interface**

**Interface** e **Implementação** podem ser usadas em outros programas

# Tipo Abstrato de Dados

# Tipo Abstrato de Dados

Um conjunto de valores associado a um conjunto de **operações permitidas** nesses dados

# Tipo Abstrato de Dados

Um conjunto de valores associado a um conjunto de **operações permitidas** nesses dados

- **Interface:** conjunto de operações de um TAD

# Tipo Abstrato de Dados

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

# Tipo Abstrato de Dados

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

# Tipo Abstrato de Dados

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



# Tipo Abstrato de Dados

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

# Tipo Abstrato de Dados

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

# Tipo Abstrato de Dados

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:

# Tipo Abstrato de Dados

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**

# Tipo Abstrato de Dados

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

# Números Complexos - Implementação

Criamos um arquivo `complexos.c` com as implementações

```
1 #include <stdio.h> ← bibliotecas usadas
2 #include <math.h>
3 #include "complexos.h" ← tem a definição da struct
4
5 complexo complexo_novo(double real, double imag) {
6     complexo c;
7     c.real = real;
8     c.imag = imag;
9     return c;
10 }
11
12 complexo complexo_soma(complexo a, complexo b) {
13     return complexo_novo(a.real + b.real, a.imag + b.imag);
14 }
15
16 complexo complexo_le() {
17     complexo a;
18     scanf("%lf %lf", &a.real, &a.imag);
19     return a;
20 }
```

# Números Complexos - Exemplo de Cliente

E quando formos usar números complexos em nossos programas?

```
1 #include <stdio.h>
2 #include "complexos.h" ← tem a struct e as funções
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 }
```



## Como compilar?

Temos três arquivos diferentes:

# Como compilar?

Temos três arquivos diferentes:

- `cliente.c` contém a função `main`

# Como compilar?

Temos três arquivos diferentes:

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

# Como compilar?

Temos três arquivos diferentes:

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

# 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:

# 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`

# 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`

# 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`



# 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`

# 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`

# 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`

# 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`

# 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

# 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
```

# 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:



# 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**

# 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**

# 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

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**

# 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

## Vantagens do TAD

- Reutilizar o código em vários programas
  - `complexos.{c,h}` podem ser usados em outros lugares

# Vantagens do TAD

- Reutilizar o código em vários programas
  - `complexos.{c,h}` podem ser usados em outros lugares
  - permite criar bibliotecas de tipos úteis



# 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

# 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

# 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

# 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

# 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

# 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

# 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

# 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



# 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

# 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

# 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

# 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

# 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`

# 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:

## Como criar um TAD

Construímos o TAD definindo:

- Um nome para o tipo a ser usado



# Como criar um TAD

Construímos o TAD definindo:

- Um nome para o tipo a ser usado
  - Ex: `complexo`

# Como criar um TAD

Construímos o TAD definindo:

- Um nome para o tipo a ser usado
  - Ex: `complexo`
  - Uma `struct` com um `typedef`

# 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

# 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...

# 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

# 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

# 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

# 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



# 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

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

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