MC-202
Curso de C — Parte 4

Lehilton Pedrosa
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

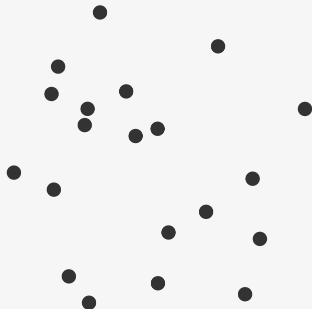
Segundo semestre de 2024

Exercício

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

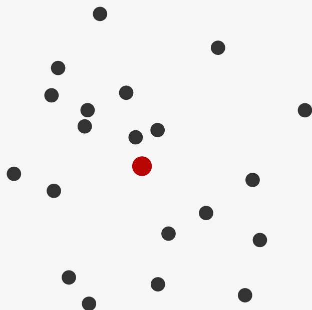
Exercício

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



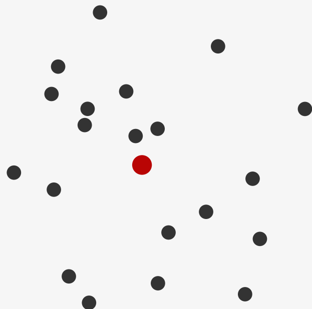
Exercício

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



Exercício

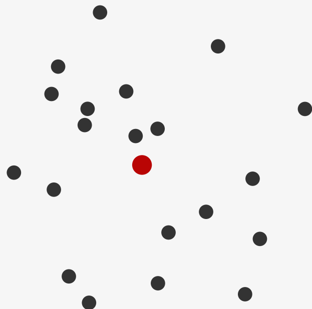
Dado um conjunto de pontos do plano, como calcular o centroide?



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

Exercício

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

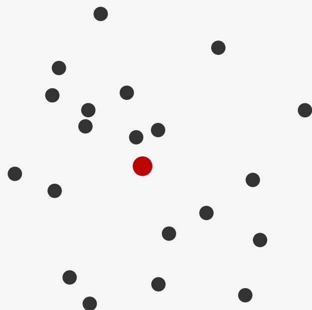


```c
#include <stdio.h>
#define MAX 100

int main() {
  double x[MAX], y[MAX];
```

Exercício

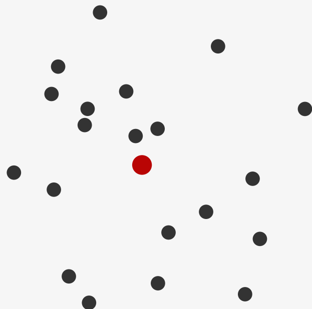
Dado um conjunto de pontos do plano, como calcular o centroide?



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

Exercício

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

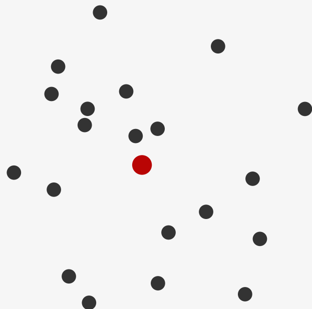


```c
#include <stdio.h>
#define MAX 100

int main() {
  double x[MAX], y[MAX];
  double cx, cy;
  int i, n;
```

Exercício

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

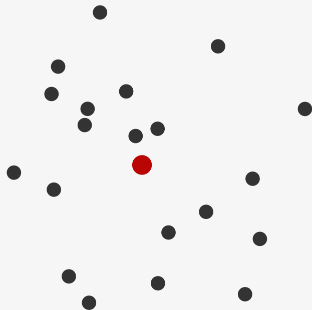


```c
#include <stdio.h>
#define MAX 100

int main() {
  double x[MAX], y[MAX];
  double cx, cy;
  int i, n;
  scanf("%d", &n);
```

Exercício

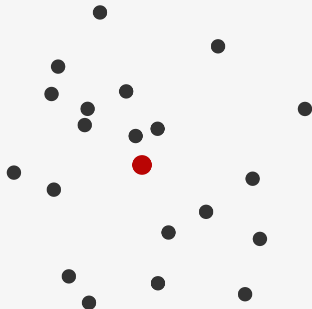
Dado um conjunto de pontos do plano, como calcular o centroide?



```
 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]);
```

Exercício

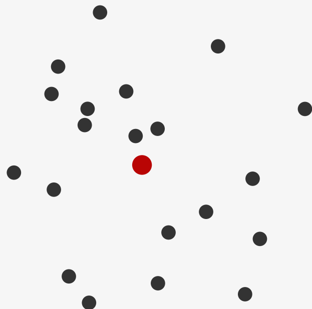
Dado um conjunto de pontos do plano, como calcular o centroide?



```
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;
```

Exercício

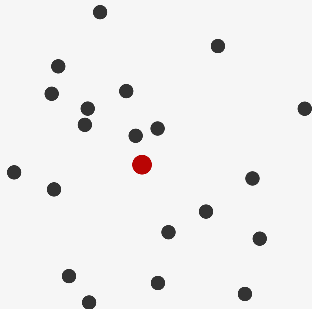
Dado um conjunto de pontos do plano, como calcular o centroide?



```
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++) {
```

Exercício

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

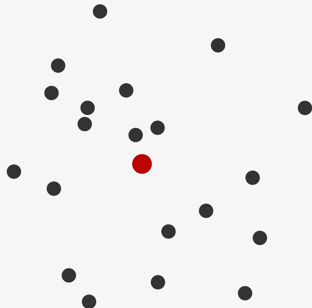


```c
#include <stdio.h>
#define MAX 100

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;
  }
```

Exercício

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

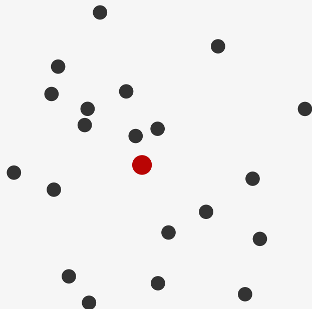


```c
#include <stdio.h>
#define MAX 100

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);
```

Exercício

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

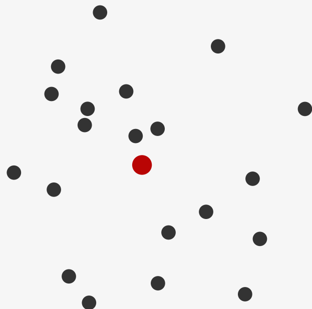


```c
#include <stdio.h>
#define MAX 100

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;
}
```

Exercício

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



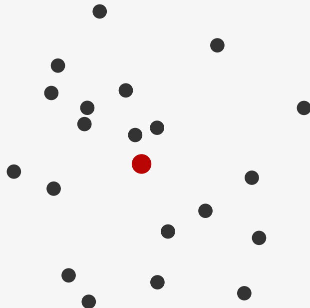
```c
#include <stdio.h>
#define MAX 100

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?

Exercício

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:

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos
- organizadas em uma única estrutura

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

- Cada dado é chamado de membro do registro

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

- Cada dado é chamado de membro do registro
- Cada membro é acessado por um nome na estrutura

Registro

Registro é:

- uma coleção de dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

- Cada dado é chamado 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 dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

- Cada dado é chamado 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 dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

- Cada dado é chamado 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 dados relacionados de vários tipos
- organizadas em uma única estrutura
- e referenciados por um nome comum

Características:

- Cada dado é chamado 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

```c
struct data {
  int dia;
  int mes;
  int ano;
};

struct ficha_aluno {
  int ra;
  int telefone;
  char nome[30];
  char endereco[100];
  struct data nascimento;
};
```

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;
```

Centroide revisitado

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

struct ponto {
  double x, y;
};
```

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;
```

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);
```

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);
```

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;
```

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   }
```

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);
```

Centroide revisitado

```c
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...

A palavra-chave `typedef`

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

Exemplo: `typedef unsigned int u32;`

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;`

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

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`

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`

```
typedef struct identificador {
  tipo1 membro1;
  tipo2 membro2;
  ...
  tipoN membroN;
} novonome;
```

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

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;`

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`

```
typedef struct identificador {
  tipo1 membro1;
  tipo2 membro2;
  ...
  tipoN membroN;
} novonome;
```

Com isso, ao invés de declarar uma variável dessa forma

- `struct identificador var;`

podemos declarar dessa forma

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`

```
typedef struct identificador {
  tipo1 membro1;
  tipo2 membro2;
  ...
  tipoN membroN;
} 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

Números Complexos

Vamos criar um programa que lida com números complexos

- Um número complexo é da forma $a + b\,i$

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

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

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 ($\sqrt{a^2 + b^2}$)

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 }
```

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

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

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

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

Números Complexos - Usando funções

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

complexo complexo_soma(complexo a, complexo b) {
  return complexo_novo(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)

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

```c
complexo complexo_novo(double real, double imag);

complexo complexo_soma(complexo a, complexo b);

double complexo_absoluto(complexo a);

complexo complexo_le();

void complexo_imprime(complexo a);

int complexos_iguais(complexo a, complexo b);

complexo complexo_multiplicacao(complexo a, complexo b);

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

```c
complexo complexo_novo(double real, double imag);

complexo complexo_soma(complexo a, complexo b);

double complexo_absoluto(complexo a);

complexo complexo_le();

void complexo_imprime(complexo a);

int complexos_iguais(complexo a, complexo b);

complexo complexo_multiplicacao(complexo a, complexo b);

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?

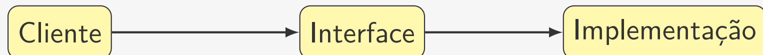
- basta copiar a struct e as funções...
- e se acharmos um bug ou quisermos mudar algo?
- Esse solução não é DRY...

Ideia

Vamos quebrar o programa em três partes

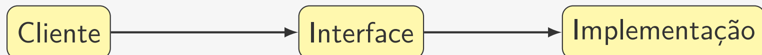
Ideia

Vamos quebrar o programa em três partes



Ideia

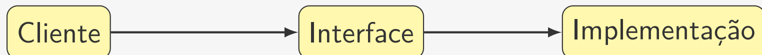
Vamos quebrar o programa em três partes



1. Implementação das funções para os números complexos

Ideia

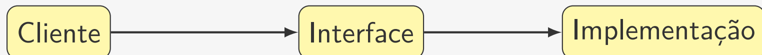
Vamos quebrar o programa em três partes



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

Ideia

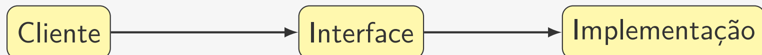
Vamos quebrar o programa em três partes



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

Ideia

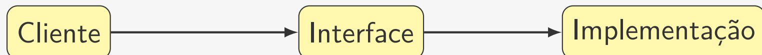
Vamos quebrar o programa em três partes



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

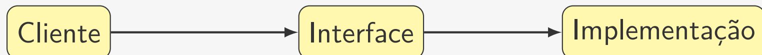
Vamos quebrar o programa em três partes



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

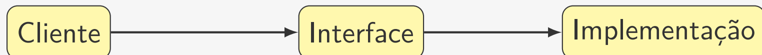
Vamos quebrar o programa em três partes



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

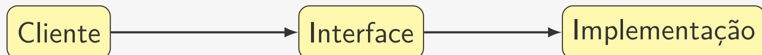
Vamos quebrar o programa em três partes



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

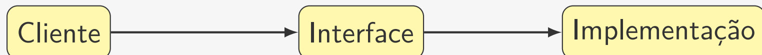
Vamos quebrar o programa em três partes



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

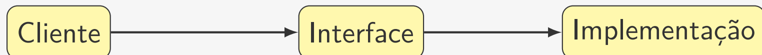
Vamos quebrar o programa em três partes



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

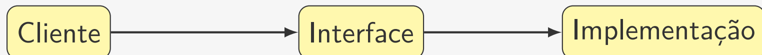
Vamos quebrar o programa em três partes



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

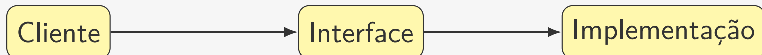
Vamos quebrar o programa em três partes



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

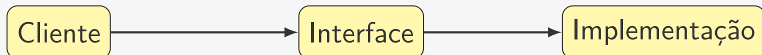
Vamos quebrar o programa em três partes



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

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

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

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

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

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

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

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:

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`

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  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>
2  #include <math.h>
3  #include "complexos.h"
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 }
```

- Linhas 1–2 ← bibliotecas usadas
- Linha 3 ← 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

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 -Wvla -g -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 -Wvla -g -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 -Wvla -g -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -Wvla -g -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 -Wvla -g -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -Wvla -g -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 -Wvla -g -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -Wvla -g -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 -Wvla -g -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -Wvla -g -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 -Wvla -g -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -Wvla -g -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 -Wvla -g -c cliente.c`
  - vai gerar o arquivo compilado `cliente.o`
- `gcc -std=c99 -Wall -Werror -Wvla -g -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 -g -lm -o cliente
5
6  cliente.o: cliente.c complexos.h
7    gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9  complexos.o: complexos.c complexos.h
10   gcc -std=c99 -Wall -Werror -Wvla -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 -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7   gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10   gcc -std=c99 -Wall -Werror -Wvla -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 -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7   gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10  gcc -std=c99 -Wall -Werror -Wvla -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 -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7   gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10  gcc -std=c99 -Wall -Werror -Wvla -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 -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7   gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10  gcc -std=c99 -Wall -Werror -Wvla -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 -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7   gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10   gcc -std=c99 -Wall -Werror -Wvla -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 -g -lm -o cliente
5
6 cliente.o: cliente.c complexos.h
7   gcc -std=c99 -Wall -Werror -Wvla -g -c cliente.c
8
9 complexos.o: complexos.c complexos.h
10  gcc -std=c99 -Wall -Werror -Wvla -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

Um último detalhe

Pode ser que você tenha dois tipos abstratos de dados

Um último detalhe

Pode ser que você tenha dois tipos abstratos de dados

- E um precise incluir o outro...

Um último detalhe

Pode ser que você tenha dois tipos abstratos de dados

- E um precise incluir o outro...
- O que leva a um loop de inclusão

Um último detalhe

Pode ser que você tenha dois tipos abstratos de dados

- E um precise incluir o outro...
- O que leva a um loop de inclusão

Podemos usar o `#ifndef` para evitar isso

Um último detalhe

Pode ser que você tenha dois tipos abstratos de dados

- E um precise incluir o outro...
- O que leva a um loop de inclusão

Podemos usar o `#ifndef` para evitar isso

```
1 #ifndef ARQUIVO_H // trocamos TAD pelo nome do arquivo
2 #define ARQUIVO_H
3
4 // Conteúdo do arquivo.h
5
6 #endif
```

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.

Solução — conjunto.h

```
1 #ifndef CONJUNTO_H
2 #define CONJUNTO_H
3 #define CONJUNTO_MAX 100
4
5 typedef struct {
6     int elementos[CONJUNTO_MAX];
7     int tamanho;
8 } Conjunto;
9
10 Conjunto conjunto_vazio();
11 Conjunto conjunto_uniao(Conjunto a, Conjunto b);
12 Conjunto conjunto_intersecao(Conjunto a, Conjunto b);
13 Conjunto conjunto_diferenca(Conjunto a, Conjunto b);
14 Conjunto conjunto_adiciona(Conjunto a, int elemento);
15 Conjunto conjunto_remove(Conjunto a, int elemento);
16 int conjunto_pertence(Conjunto a, int elemento);
17 void conjunto_imprime(Conjunto a);
18 Conjunto conjunto_le();
19
20 #endif
```

Solução — conjunto.c

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

Conjunto conjunto_vazio() {
    Conjunto c;
    c.tamanho = 0;
    return c;
}

void conjunto_imprime(Conjunto a) {
    printf("{");
    for (int i = 0; i < a.tamanho; i++) {
        printf("%d", a.elementos[i]);
        if (i < a.tamanho - 1)
            printf(", ");
    }
    printf("}\n");
}

int conjunto_pertence(Conjunto a, int elemento) {
    for (int i = 0; i < a.tamanho; i++)
        if (a.elementos[i] == elemento)
            return 1;
    return 0;
}

Conjunto conjunto_adiciona(Conjunto a, int elemento) {
    Conjunto c = a;
    if (!conjunto_contem(c, elemento)) {
        c.elementos[c.tamanho] = elemento;
        c.tamanho++;
    }
    return c;
}
```

Solução — conjunto.c (continuação)

```
35 Conjunto conjunto_remove(Conjunto a, int elemento) {
36     Conjunto c = a;
37     for (int i = 0; i < c.tamanho; i++)
38         if (c.elementos[i] == elemento) {
39             c.elementos[i] = c.elementos[c.tamanho - 1];
40             c.tamanho--;
41             break;
42         }
43     return c;
44 }
45
46 Conjunto conjunto_uniao(Conjunto a, Conjunto b) {
47     Conjunto c = a;
48     for (int i = 0; i < b.tamanho; i++)
49         c = conjunto_adiciona(c, b.elementos[i]);
50     return c;
51 }
52
53 Conjunto conjunto_intersecao(Conjunto a, Conjunto b) {
54     Conjunto c = conjunto_vazio();
55     for (int i = 0; i < a.tamanho; i++)
56         if (conjunto_contem(b, a.elementos[i]))
57             c = conjunto_adiciona(c, a.elementos[i]);
58     return c;
59 }
60
61 Conjunto conjunto_diferenca(Conjunto a, Conjunto b) {
62     Conjunto c = conjunto_vazio();
63     for (int i = 0; i < a.tamanho; i++)
64         if (!conjunto_contem(b, a.elementos[i]))
65             c = conjunto_adiciona(c, a.elementos[i]);
66     return c;
67 }
```

Solução — conjunto.c (continuação)

```
68 Conjunto conjunto_le() {
69     Conjunto c = conjunto_vazio();
70     int n, x;
71     printf("Entre com o tamanho do conjunto: ");
72     scanf("%d", &n);
73     printf("Entre com os elementos do conjunto: ");
74     for (int i = 0; i < n; i++) {
75         scanf("%d", &x);
76         c = conjunto_adiciona(c, x);
77     }
78     return c;
79 }
```

Solução — cliente_conjunto.c

```
1  #include <stdio.h>
2  #include "conjunto.h"
3
4  int main() {
5      Conjunto a = conjunto_vazio();
6      Conjunto b = conjunto_vazio();
7      printf("Conjunto A\n");
8      a = conjunto_le();
9      printf("Conjunto B\n");
10     b = conjunto_le();
11
12     printf("União: ");
13     conjunto_imprime(conjunto_uniao(a, b));
14     printf("Interseção: ");
15     conjunto_imprime(conjunto_intersecao(a, b));
16     printf("Diferença: ");
17     conjunto_imprime(conjunto_diferenca(a, b));
18
19     printf("Adicionando 42 ao conjunto A: ");
20     Conjunto c = conjunto_adiciona(a, 42);
21     conjunto_imprime(c);
22
23     printf("42 pertence ao novo conjunto? ");
24     if (conjunto_contem(c, 42))
25         printf("Sim\n");
26     else
27         printf("Não\n");
28
29     return 0;
30 }
```

Exercício — Matrizes

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

Solução — matriz.h

```
1  #ifndef MATRIZ_H
2  #define MATRIZ_H
3  #define MATRIZ_MAX 100
4
5
6  typedef struct Matriz {
7      double elementos[MATRIZ_MAX][MATRIZ_MAX];
8      int linhas;
9      int colunas;
10 } Matriz;
11
12
13 Matriz matriz_nova(int linhas, int colunas, double elementos[][MATRIZ_MAX]);
14 void matriz_imprime(Matriz m);
15 Matriz matriz_adiciona(Matriz m1, Matriz m2);
16 Matriz matriz_multiplica(Matriz m1, Matriz m2);
17 Matriz matriz_multiplica_escalar(Matriz m, double escalar);
18
19 #endif
```

Solução — matriz.c

```
1  #include "matriz.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  Matriz matriz_nova(int linhas, int colunas, double elementos[][MATRIZ_MAX]) {
6      Matriz m;
7      m.linhas = linhas;
8      m.colunas = colunas;
9      for (int i = 0; i < linhas; i++)
10         for (int j = 0; j < colunas; j++)
11             m.elementos[i][j] = elementos[i][j];
12     return m;
13 }
14
15 void matriz_imprime(Matriz m) {
16     for (int i = 0; i < m.linhas; i++) {
17         for (int j = 0; j < m.colunas; j++)
18             printf("%f ", m.elementos[i][j]);
19         printf("\n");
20     }
21 }
22
23 Matriz matriz_adiciona(Matriz m1, Matriz m2) {
24     if (m1.linhas != m2.linhas || m1.colunas != m2.colunas) {
25         printf("Erro: soma de matrizes inválida\n");
26         exit(1);
27     }
28     Matriz m;
29     m.linhas = m1.linhas;
30     m.colunas = m1.colunas;
31     for (int i = 0; i < m1.linhas; i++)
32         for (int j = 0; j < m1.colunas; j++)
33             m.elementos[i][j] = m1.elementos[i][j] + m2.elementos[i][j];
34     return m;
35 }
```

Solução — matriz.c (continuação)

```
36 Matriz matriz_multiplica(Matriz m1, Matriz m2) {
37     if (m1.colunas != m2.linhas) {
38         printf("Erro: multiplicação de matrizes inválida\n");
39         exit(1);
40     }
41     Matriz m;
42     m.linhas = m1.linhas;
43     m.colunas = m2.colunas;
44     for (int i = 0; i < m1.linhas; i++)
45         for (int j = 0; j < m2.colunas; j++) {
46             m.elementos[i][j] = 0;
47             for (int k = 0; k < m1.colunas; k++)
48                 m.elementos[i][j] += m1.elementos[i][k] * m2.elementos[k][j];
49         }
50     return m;
51
52 }
53
54 Matriz matriz_multiplica_escalar(Matriz m, double escalar) {
55     Matriz r;
56     r.linhas = m.linhas;
57     r.colunas = m.colunas;
58     for (int i = 0; i < m.linhas; i++)
59         for (int j = 0; j < m.colunas; j++)
60             r.elementos[i][j] = m.elementos[i][j] * escalar;
61     return r;
62 }
```

Solução — cliente_matriz.c

```
1  #include <stdio.h>
2  #include "matriz.h"
3
4  int main() {
5      double elementos1[MATRIZ_MAX][MATRIZ_MAX] = {{1, 2}, {3, 4}};
6      double elementos2[MATRIZ_MAX][MATRIZ_MAX] = {{5, 6}, {7, 8}};
7      Matriz m1 = matriz_nova(2, 2, elementos1);
8      Matriz m2 = matriz_nova(2, 2, elementos2);
9      Matriz m3 = matriz_adiciona(m1, m2);
10     Matriz m4 = matriz_multiplica(m1, m2);
11     Matriz m5 = matriz_multiplica_escalar(m1, 2);
12     printf("Matriz 1:\n");
13     matriz_imprime(m1);
14     printf("Matriz 2:\n");
15     matriz_imprime(m2);
16     printf("Soma:\n");
17     matriz_imprime(m3);
18     printf("Multiplicação:\n");
19     matriz_imprime(m4);
20     printf("Multiplicação da Matriz 1 por 2:\n");
21     matriz_imprime(m5);
22     return 0;
23 }
```

Dúvidas?