MC-202 — Unidade 2
Revisão: Ponteiros, Alocação Dinâmica e Tipo Abstrato de Dados

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

Universidade Estadual de Campinas

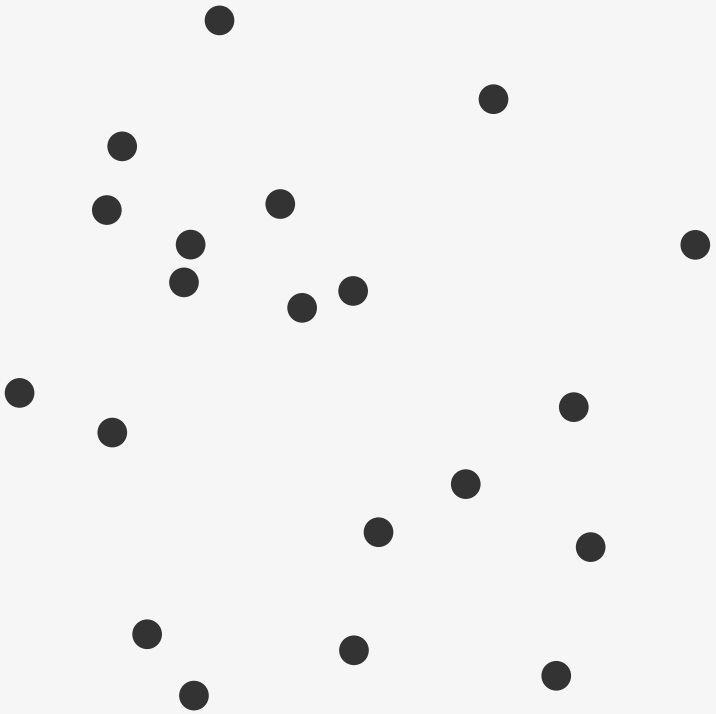
1º semestre/2017

Problema

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

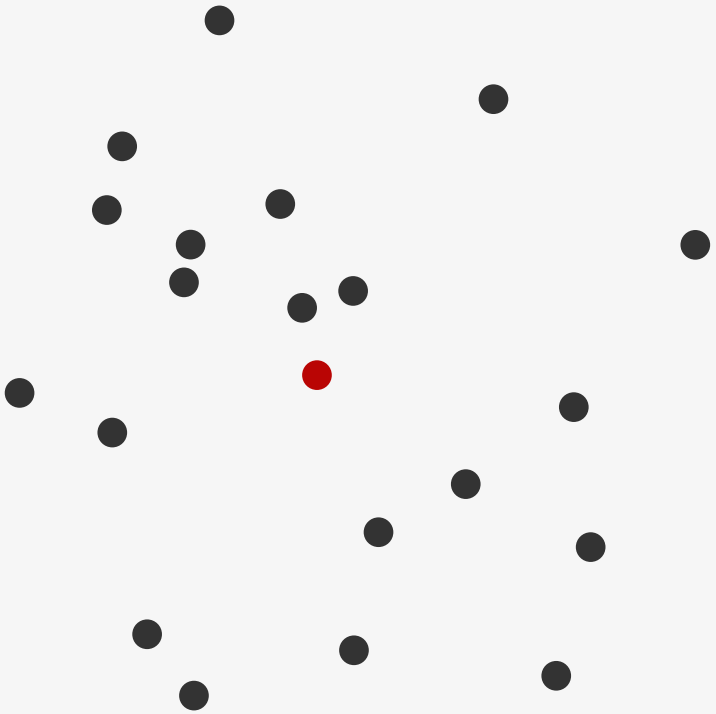
Problema

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



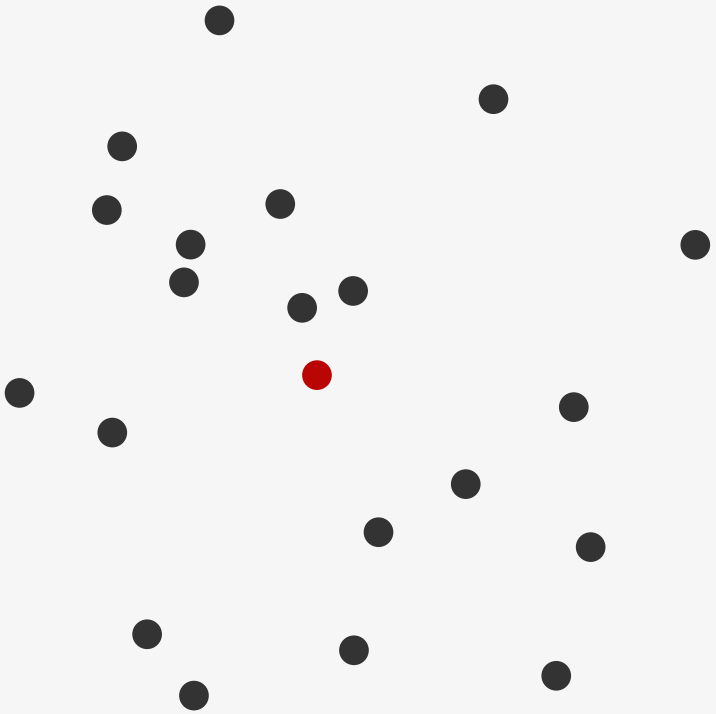
Problema

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



Problema

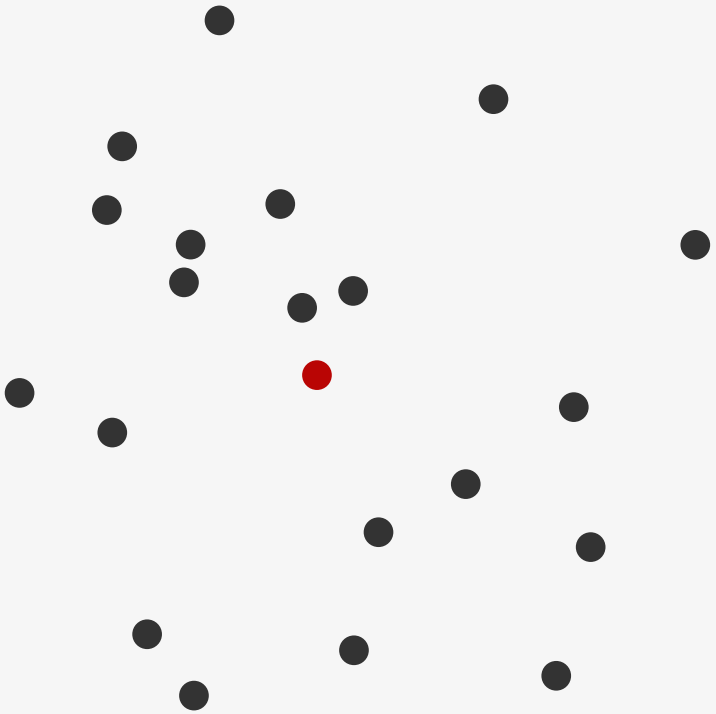
Dados um conjunto de pontos do plano, como calcular o centroide?



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

Problema

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

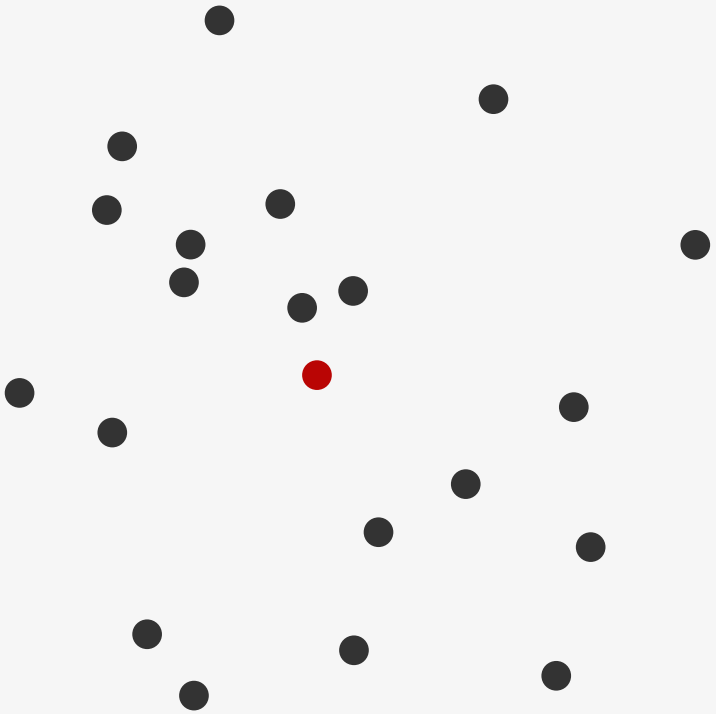


```c
#include <stdio.h>
#define MAX 100

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

Problema

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

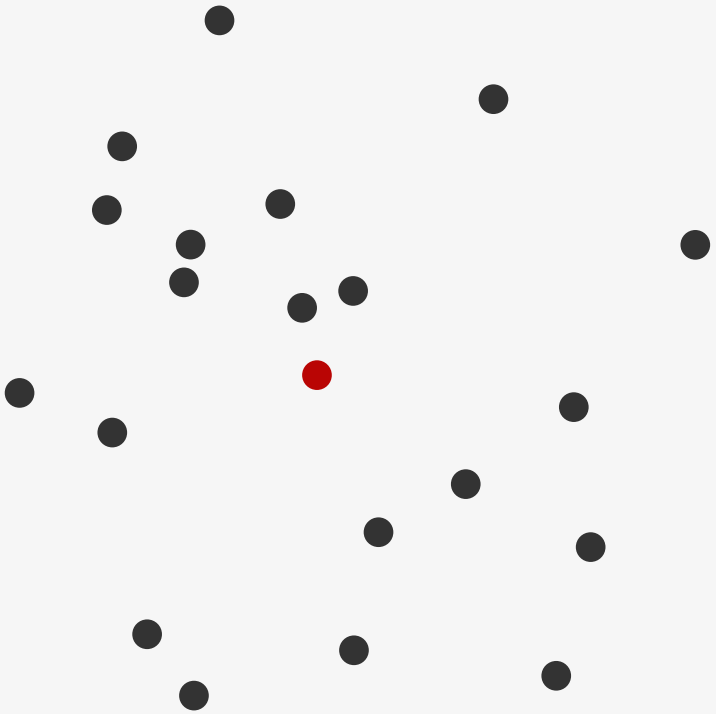


```c
#include <stdio.h>
#define MAX 100

int main() {
  float x[MAX], y[MAX];
  float cx, cy;
```

Problema

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

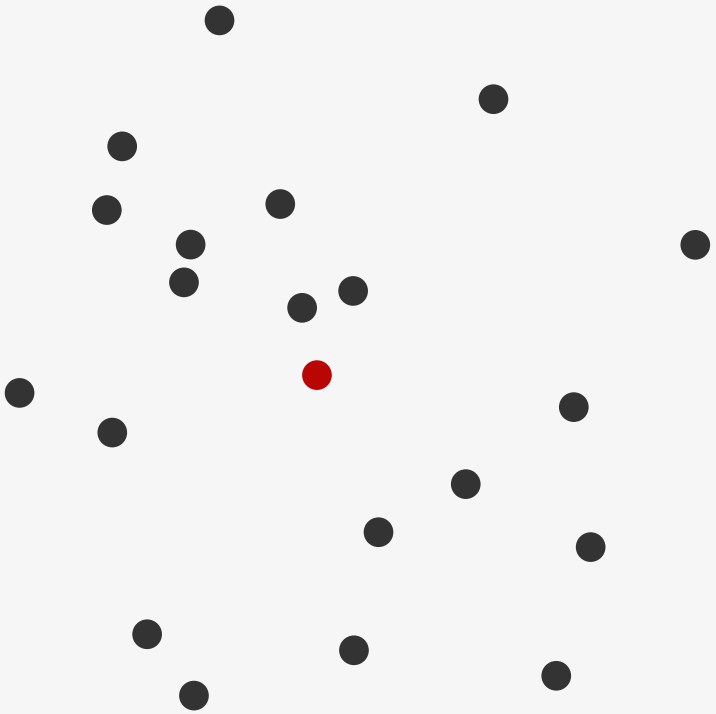


```c
#include <stdio.h>
#define MAX 100

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

Problema

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

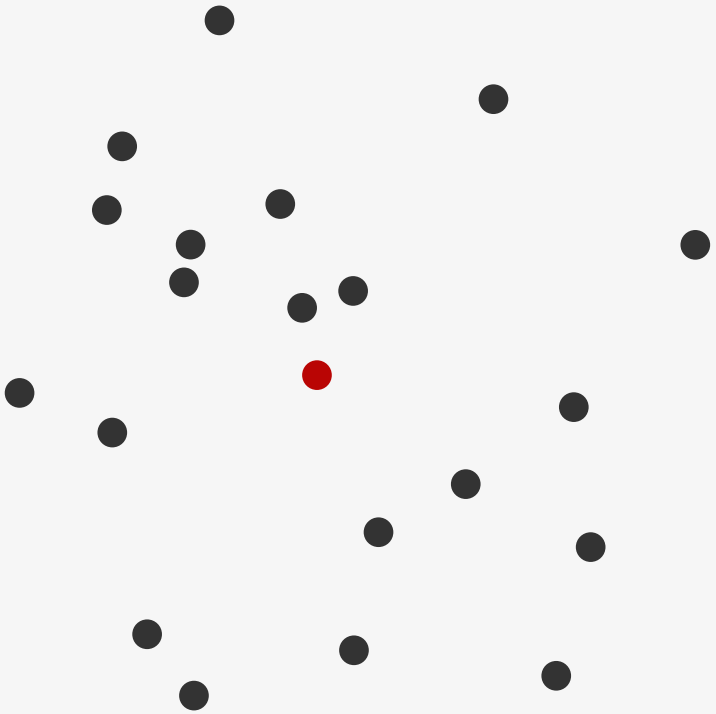


```c
#include <stdio.h>
#define MAX 100

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

Problema

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

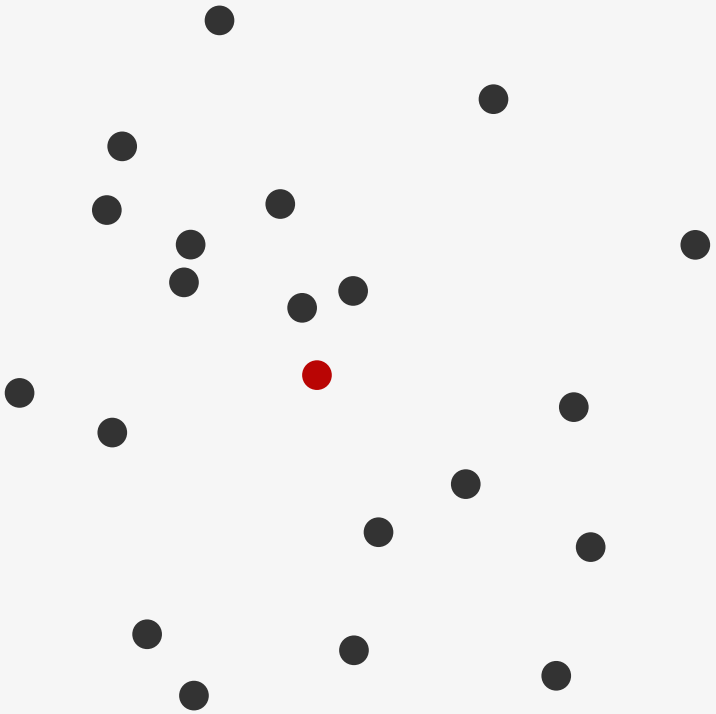


```c
#include <stdio.h>
#define MAX 100

int main() {
  float x[MAX], y[MAX];
  float cx, cy;
  int i, n;
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    scanf("%f %f", &x[i], &y[i]);
```

Problema

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

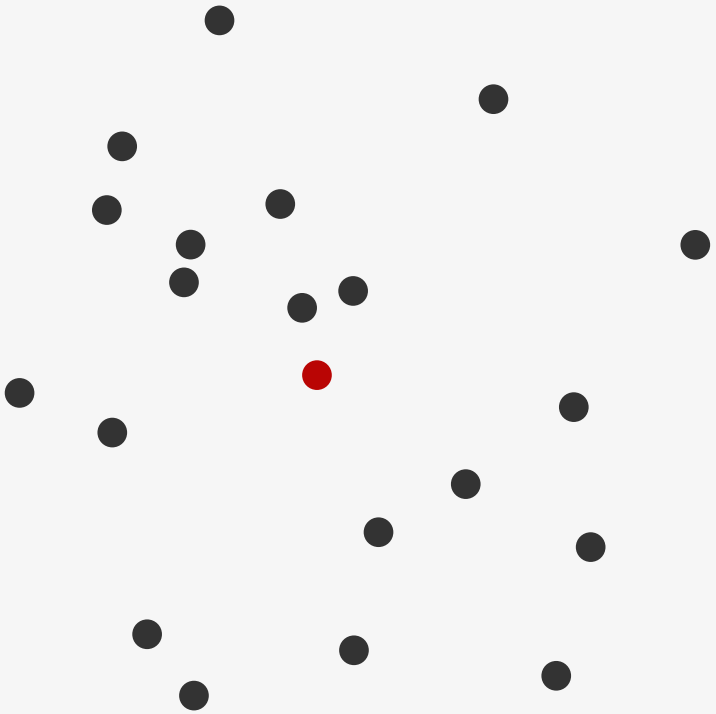


```c
#include <stdio.h>
#define MAX 100

int main() {
  float x[MAX], y[MAX];
  float cx, cy;
  int i, n;
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    scanf("%f %f", &x[i], &y[i]);
  cx = cy = 0;
```

Problema

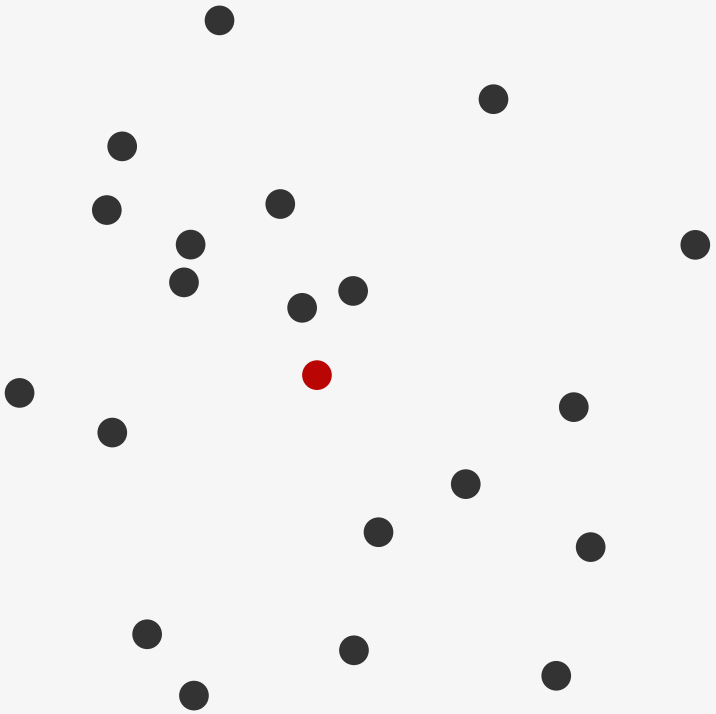
Dados um conjunto de pontos do plano, como calcular o centroide?



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

Problema

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

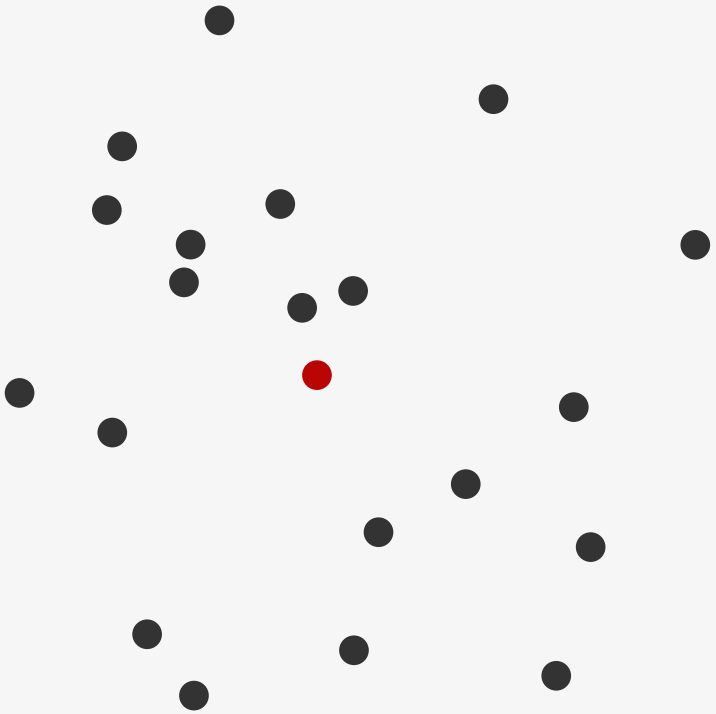


```c
#include <stdio.h>
#define MAX 100

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

Problema

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

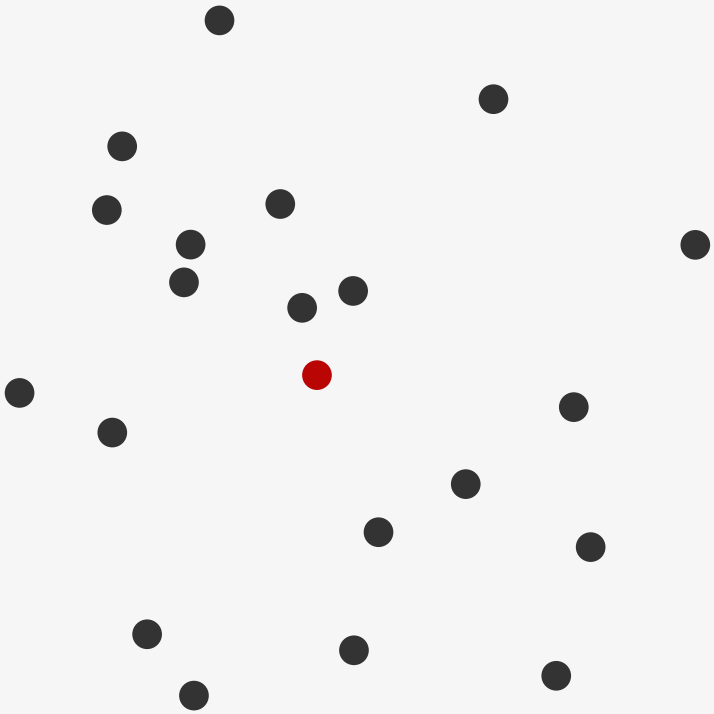


```c
#include <stdio.h>
#define MAX 100

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

Problema

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

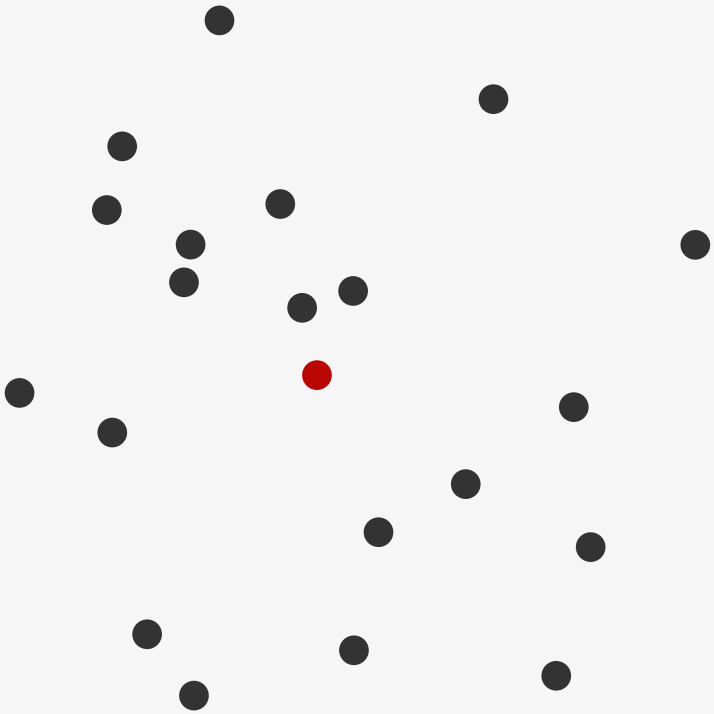


```c
#include <stdio.h>
#define MAX 100

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

Problema

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



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

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

E se tivermos mais do que `MAX` pontos?

Ponteiros

Toda informação usada pelo programa está em algum lugar

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`
  - seu tipo é `int *`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`
  - seu tipo é `int *`
- `double *q` é um ponteiro para `double`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`
  - seu tipo é `int *`
- `double *q` é um ponteiro para `double`
- `char **r` é um ponteiro para um ponteiro para `char`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`
  - seu tipo é `int *`
- `double *q` é um ponteiro para `double`
- `char **r` é um ponteiro para um ponteiro para `char`
  - armazena um endereço de um ponteiro para `char`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`
  - seu tipo é `int *`
- `double *q` é um ponteiro para `double`
- `char **r` é um ponteiro para um ponteiro para `char`
  - armazena um endereço de um ponteiro para `char`
  - é um ponteiro para um `char *`

Ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória
  - cada posição de um vetor também
  - espaços de memória alocados dinamicamente também

Um ponteiro é uma variável que armazena um endereço

- para um tipo específico de informação (ex: `int`)
  - veremos o porquê disso

Por exemplo:

- `int *p` é um ponteiro para `int`
  - armazena um endereço de um `int`
  - seu tipo é `int *`
- `double *q` é um ponteiro para `double`
- `char **r` é um ponteiro para um ponteiro para `char`
  - armazena um endereço de um ponteiro para `char`
  - é um ponteiro para um `char *`
- `int ***s` é ponteiro de ponteiro de ponteiro para `int`

Operações com ponteiros

Operações básicas:

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável
  - ou posição de um vetor

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável
  - ou posição de um vetor
- `*` acessa o conteúdo do endereço indicado pelo ponteiro

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável
  - ou posição de um vetor
- `*` acessa o conteúdo do endereço indicado pelo ponteiro

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável
  - ou posição de um vetor
- `*` acessa o conteúdo do endereço indicado pelo ponteiro

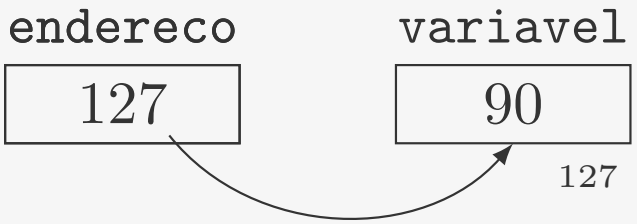
```
1 int *endereco;
2 int variavel = 90;
3 endereco = &variavel;
4 printf("Variavel: %d\n", variavel);
5 printf("Variavel: %d\n", *endereco);
6 printf("Endereço: %p\n", endereco);
7 printf("Endereço: %p\n", &variavel);
```

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável
  - ou posição de um vetor
- `*` acessa o conteúdo do endereço indicado pelo ponteiro

```
1 int *endereco;
2 int variavel = 90;
3 endereco = &variavel;
4 printf("Variavel: %d\n", variavel);
5 printf("Variavel: %d\n", *endereco);
6 printf("Endereço: %p\n", endereco);
7 printf("Endereço: %p\n", &variavel);
```



Alocação dinâmica

Às vezes queremos armazenar mais dados

Alocação dinâmica

Às vezes queremos armazenar mais dados

Mas por quê não declarar mais variáveis?

Alocação dinâmica

Às vezes queremos armazenar mais dados

Mas por quê não declarar mais variáveis?

- Não sabemos quantas variáveis e quando declará-las

Alocação dinâmica

Às vezes queremos armazenar mais dados

Mas por quê não declarar mais variáveis?

- Não sabemos quantas variáveis e quando declará-las
- Uma função pode ter que armazenar uma informação para outras funções usarem

Alocação dinâmica

Às vezes queremos armazenar mais dados

Mas por quê não declarar mais variáveis?

- Não sabemos quantas variáveis e quando declará-las
- Uma função pode ter que armazenar uma informação para outras funções usarem
- Queremos usar uma organização mais complexa da memória (*estrutura de dados*)

Organização da memória

A memória de um programa é dividida em duas partes:

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido
- O espaço é liberado quando a função termina

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido
- O espaço é liberado quando a função termina

Alocação dinâmica:

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido
- O espaço é liberado quando a função termina

Alocação dinâmica:

- `malloc` reserva um número de bytes no heap

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido
- O espaço é liberado quando a função termina

Alocação dinâmica:

- `malloc` reserva um número de bytes no heap
  - Ex: `malloc(sizeof(int))` aloca o espaço para um `int`

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido
- O espaço é liberado quando a função termina

Alocação dinâmica:

- `malloc` reserva um número de bytes no heap
  - Ex: `malloc(sizeof(int))` aloca o espaço para um `int`
- Devemos guardar o endereço da variável com um ponteiro

Organização da memória

A memória de um programa é dividida em duas partes:

- Pilha: onde são armazenadas as variáveis
  - Em geral, espaço limitado (ex: 8MB)
- Heap: onde são armazenados dados que não estão em variáveis
  - Do tamanho da memória RAM

Variáveis:

- O compilador reserva um espaço na pilha
  - alocação estática
- A variável é acessada por um nome bem definido
- O espaço é liberado quando a função termina

Alocação dinâmica:

- `malloc` reserva um número de bytes no heap
  - Ex: `malloc(sizeof(int))` aloca o espaço para um `int`
- Devemos guardar o endereço da variável com um ponteiro
- O espaço deve ser liberado usando `free`

Criando uma variável `int` dinamicamente

Criando uma variável `int` dinamicamente

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
```

Criando uma variável `int` dinamicamente

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   int *ponteiro;
```

Criando uma variável `int` dinamicamente

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   int *ponteiro;
6
7   ponteiro = malloc(sizeof(int));
```

Criando uma variável `int` dinamicamente

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main() {
 5   int *ponteiro;
 6
 7   ponteiro = malloc(sizeof(int));
 8   if (ponteiro == NULL) {
 9       printf("Não há mais memória!\n");
10       exit(1);
11   }
```

Criando uma variável `int` dinamicamente

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main() {
 5   int *ponteiro;
 6
 7   ponteiro = malloc(sizeof(int));
 8   if (ponteiro == NULL) {
 9       printf("Não há mais memória!\n");
10       exit(1);
11   }
12
13   *ponteiro = 13;
```

Criando uma variável `int` dinamicamente

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main() {
 5   int *ponteiro;
 6
 7   ponteiro = malloc(sizeof(int));
 8   if (ponteiro == NULL) {
 9       printf("Não há mais memória!\n");
10       exit(1);
11   }
12
13   *ponteiro = 13;
14   printf("Endereco %p com valor %d.\n", ponteiro, *ponteiro);
```

Criando uma variável `int` dinamicamente

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

int main() {
  int *ponteiro;

  ponteiro = malloc(sizeof(int));
  if (ponteiro == NULL) {
      printf("Não há mais memória!\n");
      exit(1);
  }

  *ponteiro = 13;
  printf("Endereco %p com valor %d.\n", ponteiro, *ponteiro);

  free(ponteiro);
  return 0;
}
```

Regras da alocação dinâmica

- Incluir a biblioteca `stdlib.h`

Regras da alocação dinâmica

- Incluir a biblioteca `stdlib.h`
- O tamanho gasto por um tipo pode ser obtido com `sizeof`

Regras da alocação dinâmica

- Incluir a biblioteca `stdlib.h`
- O tamanho gasto por um tipo pode ser obtido com `sizeof`
- Informar o tamanho a ser reservado para `malloc`

Regras da alocação dinâmica

- Incluir a biblioteca `stdlib.h`
- O tamanho gasto por um tipo pode ser obtido com `sizeof`
- Informar o tamanho a ser reservado para `malloc`
- Verificar se acabou a memória comparando com `NULL`

Regras da alocação dinâmica

- Incluir a biblioteca `stdlib.h`
- O tamanho gasto por um tipo pode ser obtido com `sizeof`
- Informar o tamanho a ser reservado para `malloc`
- Verificar se acabou a memória comparando com `NULL`
- Liberar a memória após a utilização com `free`

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float media, *notas; /* será usado como um vetor */
6   int i, n;
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float media, *notas; /* será usado como um vetor */
6   int i, n;
7   scanf("%d", &n);
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float media, *notas; /* será usado como um vetor */
6   int i, n;
7   scanf("%d", &n);
8   notas = malloc(n * sizeof(float));
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5    float media, *notas; /* será usado como um vetor */
6    int i, n;
7    scanf("%d", &n);
8    notas = malloc(n * sizeof(float));
9    for (i = 0; i < n; i++)
10     scanf("%d", &notas[i]);
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

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

int main() {
  float media, *notas; /* será usado como um vetor */
  int i, n;
  scanf("%d", &n);
  notas = malloc(n * sizeof(float));
  for (i = 0; i < n; i++)
    scanf("%d", &notas[i]);
  media = 0;
  for (i = 0; i < n; i++)
    media += notas[i]/n;
  printf("Média: %f\n", media);
```

Ponteiros e vetores

- O nome de um vetor é um ponteiro para o início do mesmo
- Podemos usar ponteiros como se fossem vetores

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

int main() {
  float media, *notas; /* será usado como um vetor */
  int i, n;
  scanf("%d", &n);
  notas = malloc(n * sizeof(float));
  for (i = 0; i < n; i++)
    scanf("%d", &notas[i]);
  media = 0;
  for (i = 0; i < n; i++)
    media += notas[i]/n;
  printf("Média: %f\n", media);
  free(notas);
  return 0;
}
```

Aritmética de ponteiros

- Vetores são ponteiros constantes: o endereço para onde apontam não pode ser alterado

Aritmética de ponteiros

- Vetores são ponteiros constantes: o endereço para onde apontam não pode ser alterado
- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento

Aritmética de ponteiros

- Vetores são ponteiros constantes: o endereço para onde apontam não pode ser alterado
- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento
  - O compilador considera o tamanho do tipo apontado

Aritmética de ponteiros

- Vetores são ponteiros constantes: o endereço para onde apontam não pode ser alterado
- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento
  - O compilador considera o tamanho do tipo apontado
  - Ex: somar `1` em um ponteiro para `int` faz com que o endereço pule `sizeof(int)` bytes

Aritmética de ponteiros

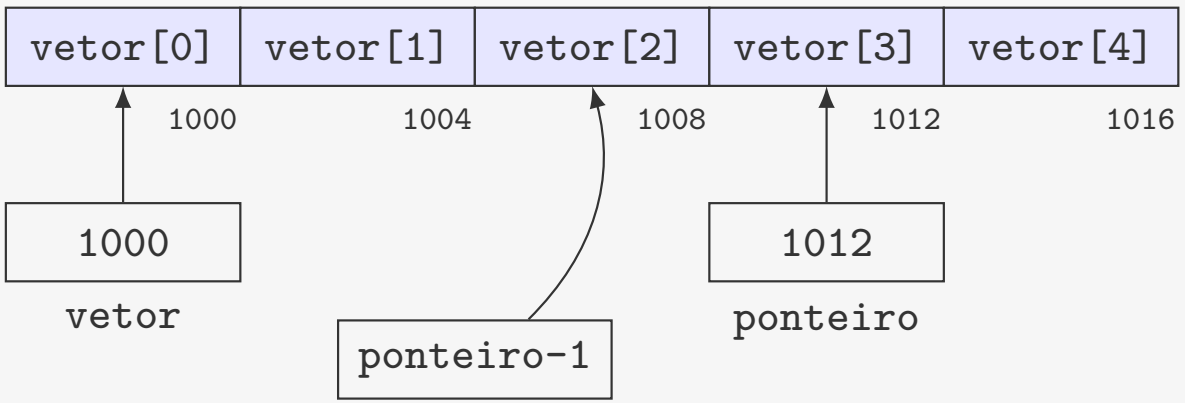
- Vetores são ponteiros constantes: o endereço para onde apontam não pode ser alterado
- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento
  - O compilador considera o tamanho do tipo apontado
  - Ex: somar 1 em um ponteiro para `int` faz com que o endereço pule `sizeof(int)` bytes

```
1 int vetor[5] = {1, 2, 3, 4, 5};
2 int *ponteiro;
3 ponteiro = vetor + 2;
4 ponteiro++;
5 printf("%d %d %d", *vetor, *(ponteiro - 1), *ponteiro);
```

Aritmética de ponteiros

- Vetores são ponteiros constantes: o endereço para onde apontam não pode ser alterado
- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento
  - O compilador considera o tamanho do tipo apontado
  - Ex: somar `1` em um ponteiro para `int` faz com que o endereço pule `sizeof(int)` bytes

```
1 int vetor[5] = {1, 2, 3, 4, 5};
2 int *ponteiro;
3 ponteiro = vetor + 2;
4 ponteiro++;
5 printf("%d %d %d", *vetor, *(ponteiro - 1), *ponteiro);
```



Exemplo: centroide com `malloc`

Exemplo: centroide com `malloc`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
```

Exemplo: centroide com `malloc`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float *x, *y,
```

Exemplo: centroide com `malloc`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float *x, *y, cx, cy;
6   int i, n;
```

Exemplo: centroide com `malloc`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float *x, *y, cx, cy;
6   int i, n;
7   scanf("%d", &n);
```

Exemplo: centroide com `malloc`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float *x, *y, cx, cy;
6   int i, n;
7   scanf("%d", &n);
8   x = malloc(n*sizeof(float));
```

Exemplo: centroide com `malloc`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5   float *x, *y, cx, cy;
6   int i, n;
7   scanf("%d", &n);
8   x = malloc(n*sizeof(float));
9   y = malloc(n*sizeof(float));
```

Exemplo: centroide com `malloc`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main() {
 5   float *x, *y, cx, cy;
 6   int i, n;
 7   scanf("%d", &n);
 8   x = malloc(n*sizeof(float));
 9   y = malloc(n*sizeof(float));
10   if (x == NULL || y == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
```

Exemplo: centroide com `malloc`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main() {
 5   float *x, *y, cx, cy;
 6   int i, n;
 7   scanf("%d", &n);
 8   x = malloc(n*sizeof(float));
 9   y = malloc(n*sizeof(float));
10   if (x == NULL || y == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
14   for (i = 0; i < n; i++)
15     scanf("%f %f", &x[i], &y[i]);
```

Exemplo: centroide com `malloc`

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

int main() {
  float *x, *y, cx, cy;
  int i, n;
  scanf("%d", &n);
  x = malloc(n*sizeof(float));
  y = malloc(n*sizeof(float));
  if (x == NULL || y == NULL) {
    printf("Não há mais memória\n");
    exit(1);
  }
  for (i = 0; i < n; i++)
    scanf("%f %f", &x[i], &y[i]);
  cx = cy = 0;
  for (i = 0; i < n; i++) {
    cx += x[i]/n;
    cy += y[i]/n;
  }
  printf("%f %f\n", cx, cy);
```

Exemplo: centroide com `malloc`

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

int main() {
  float *x, *y, cx, cy;
  int i, n;
  scanf("%d", &n);
  x = malloc(n*sizeof(float));
  y = malloc(n*sizeof(float));
  if (x == NULL || y == NULL) {
    printf("Não há mais memória\n");
    exit(1);
  }
  for (i = 0; i < n; i++)
    scanf("%f %f", &x[i], &y[i]);
  cx = cy = 0;
  for (i = 0; i < n; i++) {
    cx += x[i]/n;
    cy += y[i]/n;
  }
  printf("%f %f\n", cx, cy);
  free(x);
  free(y);
  return 0;
}
```

Passagem de parâmetros em C

Considere o seguinte código:

Passagem de parâmetros em C

Considere o seguinte código:

```
1  void imprime_invertido(int v[10], int n) {
2    while(n >= 0) {
3      printf("%d ", v[n-1]);
4      n--;
5    }
6  }
7
8  int main() {
9    int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};
10   imprime_invertido(v, n);
11   printf("%d\n", n);
12   return 0;
13 }
```

Passagem de parâmetros em C

Considere o seguinte código:

```
1  void imprime_invertido(int v[10], int n) {
2    while(n >= 0) {
3      printf("%d ", v[n-1]);
4      n--;
5    }
6  }
7
8  int main() {
9    int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};
10   imprime_invertido(v, n);
11   printf("%d\n", n);
12   return 0;
13 }
```

O que é impresso na linha 11?

Passagem de parâmetros em C

Considere o seguinte código:

```
1  void imprime_invertido(int v[10], int n) {
2    while(n >= 0) {
3      printf("%d ", v[n-1]);
4      n--;
5    }
6  }
7
8  int main() {
9    int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};
10   imprime_invertido(v, n);
11   printf("%d\n", n);
12   return 0;
13 }
```

O que é impresso na linha 11?

- `-1` ou `10`?

Passagem de parâmetros em C

Considere o seguinte código:

```
1  void imprime_invertido(int v[10], int n) {
2    while(n >= 0) {
3      printf("%d ", v[n-1]);
4      n--;
5    }
6  }
7
8  int main() {
9    int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};
10   imprime_invertido(v, n);
11   printf("%d\n", n);
12   return 0;
13 }
```

O que é impresso na linha 11?

- `-1` ou `10`?

O valor da variável local `n` da função `main` é copiado para o parâmetro (variável local) `n` da função `imprime_invertido`

Passagem de parâmetros em C

Considere o seguinte código:

```
1  void imprime_invertido(int v[10], int n) {
2    while(n >= 0) {
3      printf("%d ", v[n-1]);
4      n--;
5    }
6  }
7
8  int main() {
9    int n = 10, v[10] = {0,1,2,3,4,5,6,7,8,9};
10   imprime_invertido(v, n);
11   printf("%d\n", n);
12   return 0;
13 }
```

O que é impresso na linha 11?

- `-1` ou `10`?

O valor da variável local `n` da função `main` é copiado para o parâmetro (variável local) `n` da função `imprime_invertido`

- O valor de `n` em `main` não é alterado, continua `10`

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {
2   int i;
3   for (i = 0; i < n; i++)
4     v[i]++;
5 }
```

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {
2   int i;
3   for (i = 0; i < n; i++)
4     v[i]++;
5 }
```

Quando passamos um vetor:

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {
2   int i;
3   for (i = 0; i < n; i++)
4     v[i]++;
5 }
```

Quando passamos um vetor:

- na verdade, passamos o endereço do vetor por cópia

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {
2   int i;
3   for (i = 0; i < n; i++)
4     v[i]++;
5 }
```

Quando passamos um vetor:

- na verdade, passamos o endereço do vetor por cópia
- ou seja, o conteúdo do vetor não é passado para a função

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {
2   int i;
3   for (i = 0; i < n; i++)
4     v[i]++;
5 }
```

Quando passamos um vetor:

- na verdade, passamos o endereço do vetor por cópia
- ou seja, o conteúdo do vetor não é passado para a função
- por isso, mudanças dentro da função afetam o vetor

Passagem de parâmetros por cópia

Toda passagem de parâmetros em C é feita por cópia

- O valor da variável (ou constante) da chamada da função é copiado para o parâmetro
  - `imprime_invertido(v, n)` não altera o valor de `n`
- Mesmo se variável e parâmetro tiverem o mesmo nome

Mas e se passarmos um vetor?

```
1 void soma_um(int v[10], int n) {
2   int i;
3   for (i = 0; i < n; i++)
4     v[i]++;
5 }
```

Quando passamos um vetor:

- na verdade, passamos o endereço do vetor por cópia
- ou seja, o conteúdo do vetor não é passado para a função
- por isso, mudanças dentro da função afetam o vetor

Toda passagem de parâmetros em C é feita por cópia

Passagem por referência

Outras linguagens permitem passagem por referência

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Em C, isso pode ser simulado usando ponteiros

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Em C, isso pode ser simulado usando ponteiros

- Passamos um ponteiro para a variável que gostaríamos de alterar

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Em C, isso pode ser simulado usando ponteiros

- Passamos um ponteiro para a variável que gostaríamos de alterar
- O valor do ponteiro é passado por cópia

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Em C, isso pode ser simulado usando ponteiros

- Passamos um ponteiro para a variável que gostaríamos de alterar
- O valor do ponteiro é passado por cópia
- Mas ainda podemos acessar o valor da variável usando *

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Em C, isso pode ser simulado usando ponteiros

- Passamos um ponteiro para a variável que gostaríamos de alterar
- O valor do ponteiro é passado por cópia
- Mas ainda podemos acessar o valor da variável usando $*$

Exemplo:

Passagem por referência

Outras linguagens permitem passagem por referência

- As alterações feitas na variável dentro da função de fato alteram o valor da variável usada na chamada da função
- Pode ser mais rápido do que copiar toda a informação

Em C, isso pode ser simulado usando ponteiros

- Passamos um ponteiro para a variável que gostaríamos de alterar
- O valor do ponteiro é passado por cópia
- Mas ainda podemos acessar o valor da variável usando `*`

Exemplo:

```
1 void imprime_e_remove_ultimo(int v[10], int *n) {
2   printf("%d\n", v[*n - 1]);
3   (*n)--;
4 }
```

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ex: simular passagem por referência de um ponteiro

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ex: simular passagem por referência de um ponteiro

```
1 void aloca_e_zera(int **v, int n) {
2   int i;
3   *v = malloc(n * sizeof(int));
4   for (i = 0; i < n; i++)
5     (*v)[i] = 0;
6 }
```

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ex: simular passagem por referência de um ponteiro

```
1 void aloca_e_zera(int **v, int n) {
2   int i;
3   *v = malloc(n * sizeof(int));
4   for (i = 0; i < n; i++)
5     (*v)[i] = 0;
6 }
```

Solução mais elegante: devolver o ponteiro do novo vetor

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ex: simular passagem por referência de um ponteiro

```
1 void aloca_e_zera(int **v, int n) {
2   int i;
3   *v = malloc(n * sizeof(int));
4   for (i = 0; i < n; i++)
5     (*v)[i] = 0;
6 }
```

Solução mais elegante: devolver o ponteiro do novo vetor

Precisa ficar claro qual é o objetivo na hora de programar:

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ex: simular passagem por referência de um ponteiro

```
1 void aloca_e_zera(int **v, int n) {
2   int i;
3   *v = malloc(n * sizeof(int));
4   for (i = 0; i < n; i++)
5     (*v)[i] = 0;
6 }
```

Solução mais elegante: devolver o ponteiro do novo vetor

Precisa ficar claro qual é o objetivo na hora de programar:

- No primeiro caso, temos um vetor de vetores

Ponteiros para ponteiros

Ex: uma matriz é um vetor de vetores

```
1 int **aloca_matriz(int n, int m) {
2   int i, **matriz;
3   matriz = malloc(n * sizeof(int *));
4   for (i = 0; i < n; i++)
5     matriz[i] = malloc(m * sizeof(int));
6   return matriz;
7 }
```

Ex: simular passagem por referência de um ponteiro

```
1 void aloca_e_zera(int **v, int n) {
2   int i;
3   *v = malloc(n * sizeof(int));
4   for (i = 0; i < n; i++)
5     (*v)[i] = 0;
6 }
```

Solução mais elegante: devolver o ponteiro do novo vetor

Precisa ficar claro qual é o objetivo na hora de programar:

- No primeiro caso, temos um vetor de vetores
- No segundo caso, queremos alterar um vetor

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

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 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`
- `ponteiro_registro->membro`

Usando um registro

Acessando um membro do registro

- `registro.membro`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Usando um registro

Acessando um membro do registro

- `registro.membro`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Imprimindo o nome de um aluno

Usando um registro

Acessando um membro do registro

- `registro.membro`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Imprimindo o nome de um aluno

```
1 struct ficha_aluno aluno;
2 struct ficha_aluno *ponteiro_aluno;
3 ...
4 printf("Aluno: %s\n", aluno.nome);
5 printf("Outro aluno: %s\n", ponteiro_aluno->nome);
```

Usando um registro

Acessando um membro do registro

- `registro.membro`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Imprimindo o nome de um aluno

```
1 struct ficha_aluno aluno;
2 struct ficha_aluno *ponteiro_aluno;
3 ...
4 printf("Aluno: %s\n", aluno.nome);
5 printf("Outro aluno: %s\n", ponteiro_aluno->nome);
```

Imprimindo o aniversário

Usando um registro

Acessando um membro do registro

- `registro.membro`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Imprimindo o nome de um aluno

```
1 struct ficha_aluno aluno;
2 struct ficha_aluno *ponteiro_aluno;
3 ...
4 printf("Aluno: %s\n", aluno.nome);
5 printf("Outro aluno: %s\n", ponteiro_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`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Imprimindo o nome de um aluno

```
1 struct ficha_aluno aluno;
2 struct ficha_aluno *ponteiro_aluno;
3 ...
4 printf("Aluno: %s\n", aluno.nome);
5 printf("Outro aluno: %s\n", ponteiro_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`
- `ponteiro_registro->membro`
  - o mesmo que `(*ponteiro_registro).membro`

Imprimindo o nome de um aluno

```
1 struct ficha_aluno aluno;
2 struct ficha_aluno *ponteiro_aluno;
3 ...
4 printf("Aluno: %s\n", aluno.nome);
5 printf("Outro aluno: %s\n", ponteiro_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

```
1 typedef struct ponto {
2   float x, y;
3 } ponto;
```

Centroide revisitado

```
1 typedef struct ponto {
2   float x, y;
3 } ponto;
4
5 int main() {
6   ponto *v, centroide;
```

Centroide revisitado

```
1 typedef struct ponto {
2   float x, y;
3 } ponto;
4
5 int main() {
6   ponto *v, centroide;
7   int i, n;
8   scanf("%d", &n);
```

Centroide revisitado

```
1 typedef struct ponto {
2   float x, y;
3 } ponto;
4
5 int main() {
6   ponto *v, centroide;
7   int i, n;
8   scanf("%d", &n);
9   v = malloc(n * sizeof(ponto));
```

Centroide revisitado

```
 1 typedef struct ponto {
 2   float x, y;
 3 } ponto;
 4
 5 int main() {
 6   ponto *v, centroide;
 7   int i, n;
 8   scanf("%d", &n);
 9   v = malloc(n * sizeof(ponto));
10   if (v == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
```

Centroide revisitado

```
 1 typedef struct ponto {
 2   float x, y;
 3 } ponto;
 4
 5 int main() {
 6   ponto *v, centroide;
 7   int i, n;
 8   scanf("%d", &n);
 9   v = malloc(n * sizeof(ponto));
10   if (v == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
14   for (i = 0; i < n; i++)
15     scanf("%f %f", &v[i].x, &v[i].y);
```

Centroide revisitado

```
 1 typedef struct ponto {
 2   float x, y;
 3 } ponto;
 4
 5 int main() {
 6   ponto *v, centroide;
 7   int i, n;
 8   scanf("%d", &n);
 9   v = malloc(n * sizeof(ponto));
10   if (v == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
14   for (i = 0; i < n; i++)
15     scanf("%f %f", &v[i].x, &v[i].y);
16   centroide.x = centroide.y = 0;
```

Centroide revisitado

```
 1 typedef struct ponto {
 2   float x, y;
 3 } ponto;
 4
 5 int main() {
 6   ponto *v, centroide;
 7   int i, n;
 8   scanf("%d", &n);
 9   v = malloc(n * sizeof(ponto));
10   if (v == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
14   for (i = 0; i < n; i++)
15     scanf("%f %f", &v[i].x, &v[i].y);
16   centroide.x = centroide.y = 0;
17   for (i = 0; i < n; i++) {
18     centroide.x += v[i].x/n;
19     centroide.y += v[i].y/n;
20   }
```

Centroide revisitado

```
1  typedef struct ponto {
2    float x, y;
3  } ponto;
4
5  int main() {
6    ponto *v, centroide;
7    int i, n;
8    scanf("%d", &n);
9    v = malloc(n * sizeof(ponto));
10   if (v == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
14   for (i = 0; i < n; i++)
15     scanf("%f %f", &v[i].x, &v[i].y);
16   centroide.x = centroide.y = 0;
17   for (i = 0; i < n; i++) {
18     centroide.x += v[i].x/n;
19     centroide.y += v[i].y/n;
20   }
21   printf("%f %f\n", centroide.x, centroide.y);
```

Centroide revisitado

```
1  typedef struct ponto {
2    float x, y;
3  } ponto;
4
5  int main() {
6    ponto *v, centroide;
7    int i, n;
8    scanf("%d", &n);
9    v = malloc(n * sizeof(ponto));
10   if (v == NULL) {
11     printf("Não há mais memória\n");
12     exit(1);
13   }
14   for (i = 0; i < n; i++)
15     scanf("%f %f", &v[i].x, &v[i].y);
16   centroide.x = centroide.y = 0;
17   for (i = 0; i < n; i++) {
18     centroide.x += v[i].x/n;
19     centroide.y += v[i].y/n;
20   }
21   printf("%f %f\n", centroide.x, centroide.y);
22   free(v);
23   return 0;
24 }
```

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
  - mesmo internamente o float sendo representado por um número binário

Reflexão

Quando somamos 2 variáveis `float`:

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

Quando lemos, imprimimos ou somamos 2 pontos:

Reflexão

Quando somamos 2 variáveis `float`:

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

Quando lemos, imprimimos ou somamos 2 pontos:

- nos preocupamos com os detalhes

Reflexão

Quando somamos 2 variáveis `float`:

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

Quando lemos, imprimimos ou somamos 2 pontos:

- nos preocupamos com os detalhes

Será que também podemos abstrair um ponto?

Reflexão

Quando somamos 2 variáveis `float`:

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

Quando lemos, imprimimos ou somamos 2 pontos:

- nos preocupamos com os detalhes

Será que também podemos abstrair um ponto?

- Sim! Usando registros, funções e um pouco de cuidado

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 uma TAD

Tipo Abstrato de Dados

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

- Interface: conjunto de operações de uma 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 uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza as operações

Tipo Abstrato de Dados

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

- Interface: conjunto de operações de uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza 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 uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza 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 uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza 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 uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza 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 uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza 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 um registro (`struct`)

Tipo Abstrato de Dados

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

- Interface: conjunto de operações de uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
- Implementação: conjunto de algoritmos que realiza 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 um registro (`struct`)
- a interface é um conjunto de protótipos de funções que manipula o registro

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #ifndef RETANGULO_H_
2 #define RETANGULO_H_
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #ifndef RETANGULO_H_
2 #define RETANGULO_H_
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #ifndef RETANGULO_H_
2 #define RETANGULO_H_
3
4 typedef struct {
5     char cor[10];
6     float largura, altura;
7 } Retangulo;
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1  #ifndef RETANGULO_H_
2  #define RETANGULO_H_
3
4  typedef struct {
5      char cor[10];
6      float largura, altura;
7  } Retangulo;
8
9  // aloca memória e inicializa
10 Retangulo *criar_retangulo();
11
12 float altura_retangulo(Retangulo *ret);
13 float largura_retangulo(Retangulo *ret);
14 float area_retangulo(Retangulo *ret);
15 void ler_retangulo(Retangulo *ret);
16 void girar_retangulo(Retangulo *ret);
17
18 // finaliza e libera memória
19 void destruir_retangulo(Retangulo *ret);
20
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1  #ifndef RETANGULO_H_
2  #define RETANGULO_H_
3
4  typedef struct {
5      char cor[10];
6      float largura, altura;
7  } Retangulo;
8
9  // aloca memória e inicializa
10 Retangulo *criar_retangulo();
11
12 float altura_retangulo(Retangulo *ret);
13 float largura_retangulo(Retangulo *ret);
14 float area_retangulo(Retangulo *ret);
15 void ler_retangulo(Retangulo *ret);
16 void girar_retangulo(Retangulo *ret);
17
18 // finaliza e libera memória
19 void destruir_retangulo(Retangulo *ret);
20
21 #endif /* RETANGULO_H_ */
```

Interface: retangulo.h

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #include "retangulo.h"
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #include "retangulo.h"
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #include "retangulo.h"
2
3 Retangulo *criar_retangulo() {
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #include "retangulo.h"
2
3 Retangulo *criar_retangulo() {
4     Retangulo *r = malloc(sizeof(Retangulo));
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #include "retangulo.h"
2
3 Retangulo *criar_retangulo() {
4     Retangulo *r = malloc(sizeof(Retangulo));
5     if (r == NULL) {
6         printf("Faltou memória\n");
7         exit(1);
8     }
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1 #include "retangulo.h"
2
3 Retangulo *criar_retangulo() {
4     Retangulo *r = malloc(sizeof(Retangulo));
5     if (r == NULL) {
6         printf("Faltou memória\n");
7         exit(1);
8     }
9     r->largura = r->altura = 0; // retângulo vazio
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1  #include "retangulo.h"
2
3  Retangulo *criar_retangulo() {
4      Retangulo *r = malloc(sizeof(Retangulo));
5      if (r == NULL) {
6          printf("Faltou memória\n");
7          exit(1);
8      }
9      r->largura = r->altura = 0; // retângulo vazio
10     return r;
11 }
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```c
#include "retangulo.h"

Retangulo *criar_retangulo() {
    Retangulo *r = malloc(sizeof(Retangulo));
    if (r == NULL) {
        printf("Faltou memória\n");
        exit(1);
    }
    r->largura = r->altura = 0; // retângulo vazio
    return r;
}

float area_retangulo(Retangulo *r) {
    return r->largura * r->altura;
}
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```
1  #include "retangulo.h"
2
3  Retangulo *criar_retangulo() {
4      Retangulo *r = malloc(sizeof(Retangulo));
5      if (r == NULL) {
6          printf("Faltou memória\n");
7          exit(1);
8      }
9      r->largura = r->altura = 0; // retângulo vazio
10     return r;
11 }
12
13 float area_retangulo(Retangulo *r) {
14     return r->largura * r->altura;
15 }
16
17 ...
18
```

Um exemplo concreto: retângulo

Criar 2 arquivos: a interface e a implementação

```c
#include "retangulo.h"

Retangulo *criar_retangulo() {
    Retangulo *r = malloc(sizeof(Retangulo));
    if (r == NULL) {
        printf("Faltou memória\n");
        exit(1);
    }
    r->largura = r->altura = 0; // retângulo vazio
    return r;
}

float area_retangulo(Retangulo *r) {
    return r->largura * r->altura;
}

...

void destruir_retangulo(Retangulo *r) {
    free(r);
}
```

Implementação: retangulo.c

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
2 #include "retangulo.h"
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
2 #include "retangulo.h"
3
4 int main() {
5   float area1, area2;
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
2 #include "retangulo.h"
3
4 int main() {
5   float area1, area2;
6   Retangulo *r;
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
2 #include "retangulo.h"
3
4 int main() {
5   float area1, area2;
6   Retangulo *r;
7   r = criar_retangulo();
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1 #include <stdio.h>
2 #include "retangulo.h"
3
4 int main() {
5   float area1, area2;
6   Retangulo *r;
7   r = criar_retangulo();
8   ler_retangulo(r);
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1  #include <stdio.h>
2  #include "retangulo.h"
3
4  int main() {
5    float area1, area2;
6    Retangulo *r;
7    r = criar_retangulo();
8    ler_retangulo(r);
9
10   area1 = area_retangulo(r);
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1  #include <stdio.h>
2  #include "retangulo.h"
3
4  int main() {
5    float area1, area2;
6    Retangulo *r;
7    r = criar_retangulo();
8    ler_retangulo(r);
9
10   area1 = area_retangulo(r);
11   girar_retangulo(r);
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1  #include <stdio.h>
2  #include "retangulo.h"
3
4  int main() {
5    float area1, area2;
6    Retangulo *r;
7    r = criar_retangulo();
8    ler_retangulo(r);
9
10   area1 = area_retangulo(r);
11   girar_retangulo(r);
12   area2 = area_retangulo(r);
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

```
1  #include <stdio.h>
2  #include "retangulo.h"
3
4  int main() {
5    float area1, area2;
6    Retangulo *r;
7    r = criar_retangulo();
8    ler_retangulo(r);
9
10   area1 = area_retangulo(r);
11   girar_retangulo(r);
12   area2 = area_retangulo(r);
13
14   if (area1 != area2) {
15     printf("A implementação está incorreta!\n");
16   }
17
```

Um exemplo concreto: retângulo

Usando o TAD no cliente

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

int main() {
  float area1, area2;
  Retangulo *r;
  r = criar_retangulo();
  ler_retangulo(r);

  area1 = area_retangulo(r);
  girar_retangulo(r);
  area2 = area_retangulo(r);

  if (area1 != area2) {
    printf("A implementação está incorreta!\n");
  }

  destruir_retangulo(r);
  return 0;
}
```

Implementação: cliente.c

Exercício 1

Quando um programa inicia, ele é carregado pelo sistema operacional na memória. Pesquise como essa memória é organizada e responda:

Exercício 1

Quando um programa inicia, ele é carregado pelo sistema operacional na memória. Pesquise como essa memória é organizada e responda:

- Quais são as principais partes da memória?

Exercício 1

Quando um programa inicia, ele é carregado pelo sistema operacional na memória. Pesquise como essa memória é organizada e responda:

- Quais são as principais partes da memória?
- Em que partes são armazenadas: variáveis locais, variáveis estáticas, strings constantes, números constantes no programa, variáveis alocadas com malloc, comandos?

Exercício 2

Reflita e responda:

Exercício 2

Reflita e responda:

- Qual a diferença entre passagem por valor e referência?

Exercício 2

Reflita e responda:

- Qual a diferença entre passagem por valor e referência?
- Quando é vantajoso em passar registros (struct) por referências?

Exercício 2

Reflita e responda:

- Qual a diferença entre passagem por valor e referência?
- Quando é vantajoso em passar registros (struct) por referências?
- E quando é melhor usar passagem por valores?

Exercício 3

Outra maneira de armazenar um conjunto de pontos é salvar todas as coordenadas em uma matriz. Assim, cada ponto é representado por uma linha.

Exercício 3

Outra maneira de armazenar um conjunto de pontos é salvar todas as coordenadas em uma matriz. Assim, cada ponto é representado por uma linha.

1. Reescreva o programa que calcula o centroide com a ideia acima

Exercício 3

Outra maneira de armazenar um conjunto de pontos é salvar todas as coordenadas em uma matriz. Assim, cada ponto é representado por uma linha.

1. Reescreva o programa que calcula o centroide com a ideia acima
   - Utilize alocação dinâmica de memória em seu programa

Exercício 3

Outra maneira de armazenar um conjunto de pontos é salvar todas as coordenadas em uma matriz. Assim, cada ponto é representado por uma linha.

1. Reescreva o programa que calcula o centroide com a ideia acima
   - Utilize alocação dinâmica de memória em seu programa
2. Modifique o programa do item anterior para utilizar um vetor ao invés de uma matriz

Exercício 3

Outra maneira de armazenar um conjunto de pontos é salvar todas as coordenadas em uma matriz. Assim, cada ponto é representado por uma linha.

1. Reescreva o programa que calcula o centroide com a ideia acima
   - Utilize alocação dinâmica de memória em seu programa
2. Modifique o programa do item anterior para utilizar um vetor ao invés de uma matriz
   - Dica: utilize linearização dos indícies da matriz