

MC-202
Curso de C — Parte 5

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

Universidade Estadual de Campinas

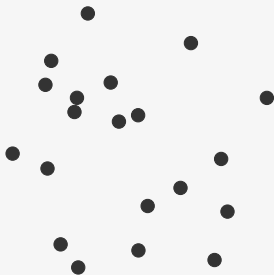
2º semestre/2023

Problema

Como calcular o centroide de um conjunto de pontos?

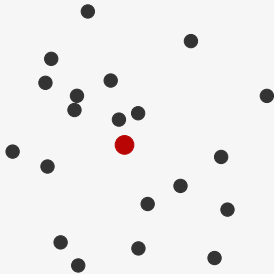
Problema

Como calcular o centroide de um conjunto de pontos?



Problema

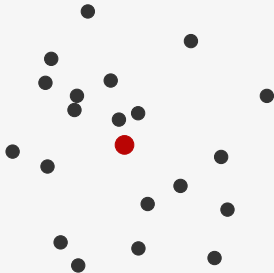
Como calcular o centroide de um conjunto de pontos?



Problema

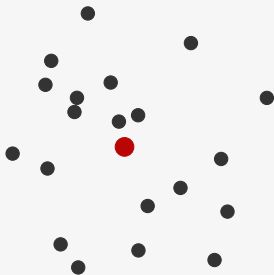
Como calcular o centroide de um conjunto de pontos?

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



Problema

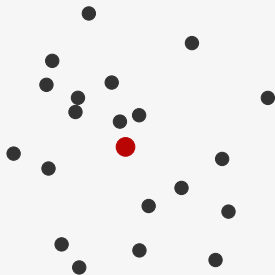
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
```

Problema

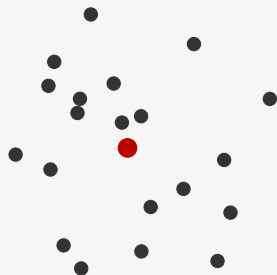
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
10    int i, n;
11    scanf("%d", &n);
```

Problema

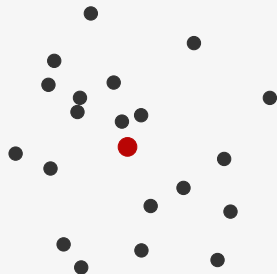
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
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);
```


Problema

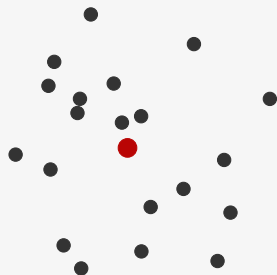
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
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    centro.x = centro.y = 0;
```

Problema

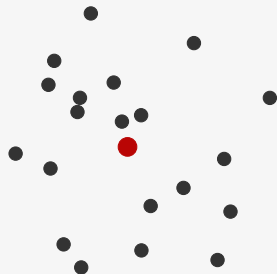
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
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    centro.x = centro.y = 0;
15    for (i = 0; i < n; i++) {
16        centro.x += v[i].x/n;
17        centro.y += v[i].y/n;
18    }
```

Problema

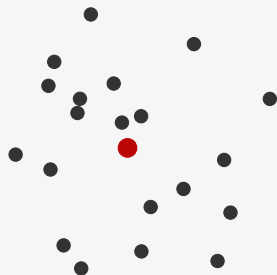
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
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    centro.x = centro.y = 0;
15    for (i = 0; i < n; i++) {
16        centro.x += v[i].x/n;
17        centro.y += v[i].y/n;
18    }
19    printf("%lf %lf\n", centro.x, centro.y);
```

Problema

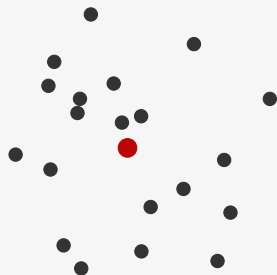
Como calcular o centroide de um conjunto de pontos?



```
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], centro;
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    centro.x = centro.y = 0;
15    for (i = 0; i < n; i++) {
16        centro.x += v[i].x/n;
17        centro.y += v[i].y/n;
18    }
19    printf("%lf %lf\n", centro.x, centro.y);
20    return 0;
21 }
```

Problema

Como calcular o centroide de um conjunto de pontos?



```
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], centro;
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    centro.x = centro.y = 0;
15    for (i = 0; i < n; i++) {
16        centro.x += v[i].x/n;
17        centro.y += v[i].y/n;
18    }
19    printf("%lf %lf\n", centro.x, centro.y);
20    return 0;
21 }
```

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
 - cada membro de um registro também

Ponteiros

Toda informação usada pelo programa está em algum lugar

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

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação

Ponteiros

Toda informação usada pelo programa está em algum lugar

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

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Ponteiros

Toda informação usada pelo programa está em algum lugar

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

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

Ponteiros

Toda informação usada pelo programa está em algum lugar

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

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara 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
 - cada membro de um registro também

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara um ponteiro para **int**
 - seu nome é **p**

Ponteiros

Toda informação usada pelo programa está em algum lugar

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

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara um ponteiro para **int**
 - seu nome é **p**
 - 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
 - cada membro de um registro também

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara um ponteiro para **int**
 - seu nome é **p**
 - seu tipo é **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
 - cada membro de um registro também

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara um ponteiro para **int**
 - seu nome é **p**
 - seu tipo é **int ***
 - armazena um endereço de um **int**
- **double *q;** declara 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
 - cada membro de um registro também

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara um ponteiro para **int**
 - seu nome é **p**
 - seu tipo é **int ***
 - armazena um endereço de um **int**
- **double *q;** declara um ponteiro para **double**
- **char *c;** declara 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
 - cada membro de um registro também

Um ponteiro é uma variável que armazena um **endereço**

- para um tipo específico de informação
 - **int**, **char**, **double**, structs declaradas, etc.

Exemplos:

- **int *p;** declara um ponteiro para **int**
 - seu nome é **p**
 - seu tipo é **int ***
 - armazena um endereço de um **int**
- **double *q;** declara um ponteiro para **double**
- **char *c;** declara um ponteiro para **char**
- **struct data *d;** declara um ponteiro para **struct data**

Operações com ponteiros

Operações básicas:

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)
- `*` acessa o conteúdo no endereço indicado pelo ponteiro

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)
- `*` acessa o conteúdo no endereço indicado pelo ponteiro
 - `*p` onde `p` é um ponteiro

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)
- `*` acessa o conteúdo no endereço indicado pelo ponteiro
 - `*p` onde `p` é um ponteiro
 - podemos ler (ex: `x = *p;`) ou escrever (ex: `*p = 10;`)

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)
- `*` acessa o conteúdo no endereço indicado pelo ponteiro
 - `*p` onde `p` é um ponteiro
 - podemos ler (ex: `x = *p;`) ou escrever (ex: `*p = 10;`)

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)
- `*` acessa o conteúdo no endereço indicado pelo ponteiro
 - `*p` onde `p` é um ponteiro
 - podemos ler (ex: `x = *p;`) ou escrever (ex: `*p = 10;`)

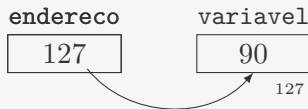
```
1 int *endereco;  
2 int variavel = 90;  
3 endereco = &variavel;  
4 printf("Variavel: %d\n", variavel);  
5 printf("Variavel: %d\n", *endereco);  
6 printf("Endereço: %p\n", endereco);  
7 printf("Endereço: %p\n", &variavel);
```

Operações com ponteiros

Operações básicas:

- `&` retorna o endereço de memória de uma variável (ex: `&x`)
 - ou posição de um vetor (ex: `&v[i]`)
 - ou campo de uma struct (ex: `&data.mes`)
 - podemos salvar o endereço em um ponteiro (ex: `p = &x;`)
- `*` acessa o conteúdo no endereço indicado pelo ponteiro
 - `*p` onde `p` é um ponteiro
 - podemos ler (ex: `x = *p;`) ou escrever (ex: `*p = 10;`)

```
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);
```



O que é um vetor em C?

Em C, se fizermos `int v[100];`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`
 - normalmente 400 bytes

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`
 - normalmente 400 bytes
- dizemos que `v` foi alocado estaticamente

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`
 - normalmente 400 bytes
- dizemos que `v` foi alocado estaticamente
 - o compilador fez o trabalho

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`
 - normalmente 400 bytes
- dizemos que `v` foi alocado estaticamente
 - o compilador fez o trabalho

Podemos alocar vetores dinamicamente

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`
 - normalmente 400 bytes
- dizemos que `v` foi alocado estaticamente
 - o compilador fez o trabalho

Podemos alocar vetores dinamicamente

- nós alocamos e nós liberamos a região de memória

O que é um vetor em C?

Em C, se fizermos `int v[100];`

- temos uma variável chamada `v`
- que é, de fato, do tipo `int * const`
 - `const` significa que não podemos fazer `v = &x;`
 - i.e., não podemos mudar o endereço armazenado em `v`
- e que aponta para o primeiro `int` do vetor
 - ou seja, `v == &v[0]`
- de uma região da memória de 100 `int`
 - normalmente 400 bytes
- dizemos que `v` foi alocado estaticamente
 - o compilador fez o trabalho

Podemos alocar vetores dinamicamente

- nós alocamos e nós liberamos a região de memória
- do tamanho que desejarmos

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória
 - a região é sempre contígua

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória
 - a região é sempre contígua
- `malloc(sizeof(struct data))` aloca a quantidade de bytes necessária para representar uma `struct data`

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória
 - a região é sempre contígua
- `malloc(sizeof(struct data))` aloca a quantidade de bytes necessária para representar uma `struct data`
- `malloc(10 * sizeof(int))` aloca a quantidade de bytes necessária para representar 10 `ints`

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória
 - a região é sempre contígua
- `malloc(sizeof(struct data))` aloca a quantidade de bytes necessária para representar uma `struct data`
- `malloc(10 * sizeof(int))` aloca a quantidade de bytes necessária para representar 10 `ints`

`free` libera uma região de memória alocada dinamicamente

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória
 - a região é sempre contígua
- `malloc(sizeof(struct data))` aloca a quantidade de bytes necessária para representar uma `struct data`
- `malloc(10 * sizeof(int))` aloca a quantidade de bytes necessária para representar 10 `ints`

`free` libera uma região de memória alocada dinamicamente

- precisa ser um endereço que foi devolvido por `malloc`

sizeof, malloc e free

`sizeof` devolve o tamanho em bytes de um tipo dado

- `sizeof(int)` (normalmente) devolve 4
- `sizeof(struct data)` - tamanho da `struct data`
 - é a soma dos tamanhos dos seus membros
 - e possivelmente mais alguns bytes para alinhamento

`malloc` aloca dinamicamente a quantidade de bytes informada

- devolve o endereço inicial da região de memória
 - a região é sempre contígua
- `malloc(sizeof(struct data))` aloca a quantidade de bytes necessária para representar uma `struct data`
- `malloc(10 * sizeof(int))` aloca a quantidade de bytes necessária para representar 10 `ints`

`free` libera uma região de memória alocada dinamicamente

- precisa ser um endereço que foi devolvido por `malloc`
- evita que vazemos memória (*memory leak*)

Aritmética de ponteiros

Podemos realizar operações aritméticas em ponteiros:

Aritmética de ponteiros

Podemos realizar operações aritméticas em ponteiros:

- somar ou subtrair um número inteiro

Aritmética de ponteiros

Podemos realizar operações aritméticas em ponteiros:

- somar ou subtrair um número inteiro
- também incremento (`++`) e decremento (`--`)

Aritmética de ponteiros

Podemos realizar operações aritméticas em ponteiros:

- somar ou subtrair um número inteiro
- também incremento (**++**) e decremento (**--**)
- o compilador considera o tamanho do tipo apontado

Aritmética de ponteiros

Podemos realizar operações aritméticas em ponteiros:

- somar ou subtrair um número inteiro
- também 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

Podemos realizar operações aritméticas em ponteiros:

- somar ou subtrair um número inteiro
- também 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

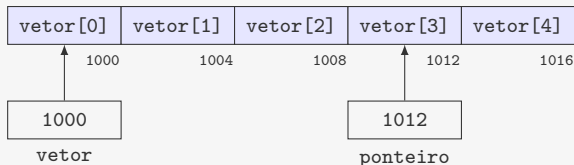
```
1 int vetor[5] = {1, 2, 3, 4, 5};
2 int *ponteiro;
3 ponteiro = vetor + 2;
4 ponteiro++;
5 printf("%d %d %d", *vetor, *(ponteiro - 1), *ponteiro);
```


Aritmética de ponteiros

Podemos realizar operações aritméticas em ponteiros:

- somar ou subtrair um número inteiro
- também 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);
```



Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

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

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

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

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

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

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

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


Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

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

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     double media, *notas; /* será usado como um vetor */
6     int i, n;
7     scanf("%d", &n);
8     notas = malloc(n * sizeof(double));
9     if (notas == NULL) {
10        printf("Nao ha memoria suficiente!\n");
11        exit(1);
12    }
13    for (i = 0; i < n; i++)
14        scanf("%lf", &notas[i]);
```

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     double media, *notas; /* será usado como um vetor */
6     int i, n;
7     scanf("%d", &n);
8     notas = malloc(n * sizeof(double));
9     if (notas == NULL) {
10        printf("Nao ha memoria suficiente!\n");
11        exit(1);
12    }
13    for (i = 0; i < n; i++)
14        scanf("%lf", &notas[i]);
15    media = 0;
16    for (i = 0; i < n; i++)
17        media += notas[i] / n;
18    printf("Média: %lf\n", media);
```

Ponteiros e vetores

Se tivermos um ponteiro `p`, podemos escrever `p[i]`

- como se fosse um vetor
- é o mesmo que escrever `*(p + i)`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     double media, *notas; /* será usado como um vetor */
6     int i, n;
7     scanf("%d", &n);
8     notas = malloc(n * sizeof(double));
9     if (notas == NULL) {
10        printf("Nao ha memoria suficiente!\n");
11        exit(1);
12    }
13    for (i = 0; i < n; i++)
14        scanf("%lf", &notas[i]);
15    media = 0;
16    for (i = 0; i < n; i++)
17        media += notas[i] / n;
18    printf("Média: %lf\n", media);
19    free(notas);
20    return 0;
21 }
```

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 os outros dados

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 os outros dados
 - Do tamanho da memória RAM disponível

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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (variáveis):

- O compilador reserva um espaço na pilha
- 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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (variáveis):

- O compilador reserva um espaço na pilha
- 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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (variáveis):

- O compilador reserva um espaço na pilha
- 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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (variáveis):

- O compilador reserva um espaço na pilha
- 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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (variáveis):

- O compilador reserva um espaço na pilha
- 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
- 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 os outros dados
 - Do tamanho da memória RAM disponível

Alocação estática (variáveis):

- O compilador reserva um espaço na pilha
- 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
- Devemos guardar o endereço da variável com um ponteiro
- O espaço deve ser liberado usando **free**

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`
 - ex: `v = malloc(n * sizeof(int));`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`
 - ex: `v = malloc(n * sizeof(int));`
- Verifique se acabou a memória comparando com `NULL`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`
 - ex: `v = malloc(n * sizeof(int));`
- Verifique se acabou a memória comparando com `NULL`
 - use a função `exit` para sair do programa

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`
 - ex: `v = malloc(n * sizeof(int));`
- Verifique se acabou a memória comparando com `NULL`
 - use a função `exit` para sair do programa
 - ex:

```
1 if (v == NULL) {  
2     printf("Nao ha memoria suficiente!\n");  
3     exit(1);  
4 }
```

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`
 - ex: `v = malloc(n * sizeof(int));`
- Verifique se acabou a memória comparando com `NULL`
 - use a função `exit` para sair do programa
 - ex:

```
1 if (v == NULL) {
2     printf("Nao ha memoria suficiente!\n");
3     exit(1);
4 }
```
- Libere a memória após a utilização com `free`

Receita para alocação dinâmica de vetores

- Incluir a biblioteca `stdlib.h`
- Declare o ponteiro com o tipo apropriado
 - ex: `int *v;`
- Aloque a região de memória com `malloc`
 - O tamanho de um tipo pode ser obtido com `sizeof`
 - ex: `v = malloc(n * sizeof(int));`
- Verifique se acabou a memória comparando com `NULL`
 - use a função `exit` para sair do programa
 - ex:

```
1 if (v == NULL) {
2     printf("Nao ha memoria suficiente!\n");
3     exit(1);
4 }
```
- Libere a memória após a utilização com `free`
 - ex: `free(v);`

Voltando ao centroide

Voltando ao centroide

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

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
```

Voltando ao centroide

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


Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
```

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
13    if (v == NULL) {
14        printf("Nao ha memoria suficiente!\n");
15        exit(1);
16    }
17    for (i = 0; i < n; i++)
18        scanf("%lf %lf", &v[i].x, &v[i].y);
```

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
13    if (v == NULL) {
14        printf("Nao ha memoria suficiente!\n");
15        exit(1);
16    }
17    for (i = 0; i < n; i++)
18        scanf("%lf %lf", &v[i].x, &v[i].y);
19    centro.x = centro.y = 0;
```

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
13    if (v == NULL) {
14        printf("Nao ha memoria suficiente!\n");
15        exit(1);
16    }
17    for (i = 0; i < n; i++)
18        scanf("%lf %lf", &v[i].x, &v[i].y);
19    centro.x = centro.y = 0;
20    for (i = 0; i < n; i++) {
21        centro.x += v[i].x/n;
22        centro.y += v[i].y/n;
23    }
```

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
13    if (v == NULL) {
14        printf("Nao ha memoria suficiente!\n");
15        exit(1);
16    }
17    for (i = 0; i < n; i++)
18        scanf("%lf %lf", &v[i].x, &v[i].y);
19    centro.x = centro.y = 0;
20    for (i = 0; i < n; i++) {
21        centro.x += v[i].x/n;
22        centro.y += v[i].y/n;
23    }
24    printf("%lf %lf\n", centro.x, centro.y);
```

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
13    if (v == NULL) {
14        printf("Nao ha memoria suficiente!\n");
15        exit(1);
16    }
17    for (i = 0; i < n; i++)
18        scanf("%lf %lf", &v[i].x, &v[i].y);
19    centro.x = centro.y = 0;
20    for (i = 0; i < n; i++) {
21        centro.x += v[i].x/n;
22        centro.y += v[i].y/n;
23    }
24    printf("%lf %lf\n", centro.x, centro.y);
25    free(v);
```

Voltando ao centroide

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     double x, y;
6 };
7
8 int main() {
9     struct ponto *v, centro;
10    int i, n;
11    scanf("%d", &n);
12    v = malloc(n * sizeof(struct ponto));
13    if (v == NULL) {
14        printf("Nao ha memoria suficiente!\n");
15        exit(1);
16    }
17    for (i = 0; i < n; i++)
18        scanf("%lf %lf", &v[i].x, &v[i].y);
19    centro.x = centro.y = 0;
20    for (i = 0; i < n; i++) {
21        centro.x += v[i].x/n;
22        centro.y += v[i].y/n;
23    }
24    printf("%lf %lf\n", centro.x, centro.y);
25    free(v);
26    return 0;
27 }
```

Ponteiros, vetores e funções

Funções

Ponteiros, vetores e funções

Funções

- não podem devolver vetores

Ponteiros, vetores e funções

Funções

- não podem devolver vetores
 - não podemos escrever `int [] funcao(...)`

Ponteiros, vetores e funções

Funções

- não podem devolver vetores
 - não podemos escrever `int [] funcao(...)`
- mas podem devolver ponteiros

Ponteiros, vetores e funções

Funções

- não podem devolver vetores
 - não podemos escrever `int [] funcao(...)`
- mas podem devolver ponteiros
 - podemos escrever `int * funcao(...)`

Ponteiros, vetores e funções

Funções

- não podem devolver vetores
 - não podemos escrever `int [] funcao(...)`
- mas podem devolver ponteiros
 - podemos escrever `int * funcao(...)`

Nunca devolva o endereço de uma variável local

Ponteiros, vetores e funções

Funções

- não podem devolver vetores
 - não podemos escrever `int [] funcao(...)`
- mas podem devolver ponteiros
 - podemos escrever `int * funcao(...)`

Nunca devolva o endereço de uma variável local

- Ela deixará de existir quando a função terminar

Ponteiros, vetores e funções

Funções

- não podem devolver vetores
 - não podemos escrever `int [] funcao(...)`
- mas podem devolver ponteiros
 - podemos escrever `int * funcao(...)`

Nunca devolva o endereço de uma variável local

- Ela deixará de existir quando a função terminar
- Ou seja, nunca devolva um vetor alocado estaticamente

Exercício - Alocando vetor

Escreva uma função que dado um `int n`, aloca um vetor de `double` com n posições zerado.

Exercício - Alocando vetor

Escreva uma função que dado um `int n`, aloca um vetor de `double` com `n` posições zerado.

```
1 double * aloca_e_zera(int n) {
2     int i;
3     double *v = malloc(n * sizeof(double));
4     for (i = 0; i < n; i++)
5         v[i] = 0.0;
6     return v;
7 }
```

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

```
1 void imprime(double *v, int n) {
2     int i;
3     for (i = 0; i < n; i++)
4         printf("%lf", v[i]);
5     printf("\n");
6 }
```

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

```
1 void imprime(double *v, int n) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         printf("%lf", v[i]);  
5     printf("\n");  
6 }
```

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

```
1 void imprime(double *v, int n) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         printf("%lf", v[i]);  
5     printf("\n");  
6 }
```

Alocado dinamicamente

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

```
1 void imprime(double *v, int n) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         printf("%lf", v[i]);  
5     printf("\n");  
6 }
```

Alocado dinamicamente

```
1 v = malloc(n * sizeof(double));  
2 ...  
3 imprime(v, n);
```


Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

```
1 void imprime(double *v, int n) {
2     int i;
3     for (i = 0; i < n; i++)
4         printf("%lf", v[i]);
5     printf("\n");
6 }
```

Alocado dinamicamente

```
1 v = malloc(n * sizeof(double));
2 ...
3 imprime(v, n);
```

Alocado estaticamente

Exercício - Imprimindo vetores

Queremos fazer uma função que imprime um vetor

- para vetores alocados estaticamente ou dinamicamente

Como vetores são ponteiros, basta receber um ponteiro!

```
1 void imprime(double *v, int n) {
2     int i;
3     for (i = 0; i < n; i++)
4         printf("%lf", v[i]);
5     printf("\n");
6 }
```

Alocado dinamicamente

```
1 v = malloc(n * sizeof(double));
2 ...
3 imprime(v, n);
```

Alocado estaticamente

```
1 double w[100];
2 ...
3 imprime(w, 100);
```

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`
 - veja o endereço armazenado em `d`

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`
 - veja o endereço armazenado em `d`
 - vá para essa posição de memória (onde está o registro)

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`
 - veja o endereço armazenado em `d`
 - vá para essa posição de memória (onde está o registro)
 - acesse o campo `mes` deste registro

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`
 - veja o endereço armazenado em `d`
 - vá para essa posição de memória (onde está o registro)
 - acesse o campo `mes` deste registro
- porém isso é tão comum que temos um atalho: `d->mes`

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`
 - veja o endereço armazenado em `d`
 - vá para essa posição de memória (onde está o registro)
 - acesse o campo `mes` deste registro
- porém isso é tão comum que temos um atalho: `d->mes`
 - significa exatamente o mesmo que `(*d).mes`

Ponteiros e Structs

Frequentemente alocamos uma `struct` dinamicamente

- Elas serão o elemento básico de muitas das EDs
- Teremos o ponteiro para uma `struct`
- e precisaremos acessar um dos seus campos...

Imagine que temos um ponteiro `d` do tipo `struct data *`

- acessamos o campo `mes` fazendo `(*d).mes`
 - veja o endereço armazenado em `d`
 - vá para essa posição de memória (onde está o registro)
 - acesse o campo `mes` deste registro
- porém isso é tão comum que temos um atalho: `d->mes`
 - significa exatamente o mesmo que `(*d).mes`
 - é um açúcar sintático do C

Exercício

- Declare uma `struct` que armazena informações de notas de uma turma. Essa estrutura deve armazenar o número de alunos, as notas das provas e a maior nota.

Exercício

- Declare uma `struct` que armazena informações de notas de uma turma. Essa estrutura deve armazenar o número de alunos, as notas das provas e a maior nota.
- Depois faça um programa que leia todos os dados e imprima a maior nota.