MC-202
Curso de C - Parte 6

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

Universidade Estadual de Campinas

2º semestre/2018

Relembrando: ponteiros

Toda informação usada pelo programa está em algum lugar

Relembrando: ponteiros

Toda informação usada pelo programa está em algum lugar

- Toda variável tem um endereço de memória

Relembrando: ponteiros

Toda informação usada pelo programa está em algum lugar

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

Relembrando: 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

Relembrando: 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

Relembrando: 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

Relembrando: 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

Relembrando: 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:

Relembrando: 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`

Relembrando: 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`

Relembrando: 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 *`

Relembrando: 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`

Relembrando: 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`

Relembrando: 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`

Relembrando: 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`

Relembrando: operações com ponteiros

Operações básicas:

Relembrando: operações com ponteiros

Operações básicas:

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

Relembrando: operações com ponteiros

Operações básicas:

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

Relembrando: 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`)

Relembrando: 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;`)

Relembrando: 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

Relembrando: 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

Relembrando: 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;`)

Relembrando: 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;`)

Relembrando: 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;`)

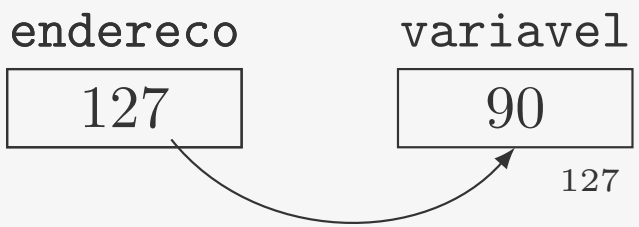
```
1 int *endereco;
2 int variavel = 90;
3 endereco = &variavel;
4 printf("Variavel: %d\n", variavel);
5 printf("Variavel: %d\n", *endereco);
6 printf("Endereço: %p\n", endereco);
7 printf("Endereço: %p\n", &variavel);
```

Relembrando: 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);
```



Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

```
1 def adiciona(lista):
2     lista.append(3)
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

```
1 def adiciona(lista):
2     lista.append(3)
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

```
1 def adiciona(lista):
2     lista = [1, 2, 3]
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

```
1 def adiciona(lista):
2     lista.append(3)
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

```
1 def adiciona(lista):
2     lista = [1, 2, 3]
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

O que é impresso?

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

```
1 def adiciona(lista):
2     lista.append(3)
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

```
1 def adiciona(lista):
2     lista = [1, 2, 3]
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

O que é impresso?

esquerda: 10

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

```
1 def adiciona(lista):
2     lista.append(3)
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

```
1 def adiciona(lista):
2     lista = [1, 2, 3]
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

O que é impresso?

esquerda: `10`

meio: `[1, 2, 3]`

Passagem de parâmetros em Python

Vamos analisar três códigos escritos em Python

```
1 def adiciona(x):
2     x = x + 1
3
4 x = 10
5 adiciona(x)
6 print(x)
```

```
1 def adiciona(lista):
2     lista.append(3)
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

```
1 def adiciona(lista):
2     lista = [1, 2, 3]
3
4 lista = [1, 2]
5 adiciona(lista)
6 print(lista)
```

O que é impresso?

esquerda: `10`

meio: `[1, 2, 3]`

direita: `[1, 2]`

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?

- 0 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?

- `0` 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?

- `0` 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 de parâmetros em C

Vamos analisar o código anterior modificado:

Passagem de parâmetros em C

Vamos analisar o código anterior modificado:

```
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

Vamos analisar o código anterior modificado:

```
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

Vamos analisar o código anterior modificado:

```
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?

- 0 ou 10?

Alterando variáveis com funções

Quando passamos um vetor para um função

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis
  - ex: `scanf("%d %lf", &n, &x);`

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis
  - ex: `scanf("%d %lf", &n, &x);`
- o `scanf` recebe um ponteiro com o endereço da variável

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis
  - ex: `scanf("%d %lf", &n, &x);`
- o `scanf` recebe um ponteiro com o endereço da variável
  - e usa o operador `*` para modificar a variável

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis
  - ex: `scanf("%d %lf", &n, &x);`
- o `scanf` recebe um ponteiro com o endereço da variável
  - e usa o operador `*` para modificar a variável

Por que quando fazemos `scanf` de uma string não usamos `&`?

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis
  - ex: `scanf("%d %lf", &n, &x);`
- o `scanf` recebe um ponteiro com o endereço da variável
  - e usa o operador `*` para modificar a variável

Por que quando fazemos `scanf` de uma string não usamos `&`?

- A string é um vetor de `char`

Alterando variáveis com funções

Quando passamos um vetor para um função

- sabemos onde o vetor está na memória
- assim conseguimos modificá-lo...

Podemos fazer o mesmo para qualquer variável!

- basta saber onde a variável está na memória
  - usando o operador `&`
- por exemplo, o `scanf` modifica o valor das variáveis
  - porque recebe os endereços da variáveis
  - ex: `scanf("%d %lf", &n, &x);`
- o `scanf` recebe um ponteiro com o endereço da variável
  - e usa o operador `*` para modificar a variável

Por que quando fazemos `scanf` de uma string não usamos `&`?

- A string é um vetor de `char`
- e o vetor já é um ponteiro

Referências em Python

O Python envia a referência do objeto para a função

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória
  - e por isso pode modificá-lo

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória
  - e por isso pode modificá-lo
- e pode ser mais rápido do que copiar todo o objeto

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória
  - e por isso pode modificá-lo
- e pode ser mais rápido do que copiar todo o objeto

É o que chamamos de passagem por referência

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória
  - e por isso pode modificá-lo
- e pode ser mais rápido do que copiar todo o objeto

É o que chamamos de passagem por referência

- E que no C é simulado usando ponteiros

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória
  - e por isso pode modificá-lo
- e pode ser mais rápido do que copiar todo o objeto

É o que chamamos de passagem por referência

- E que no C é simulado usando ponteiros
- O Python está fazendo o mesmo que fazemos em C

Referências em Python

O Python envia a referência do objeto para a função

- e por isso que podemos modificar o objeto
  - se o tipo for mutável..
- é algo parecido com o que o C faz com vetores
- a função sabe onde o objeto está na memória
  - e por isso pode modificá-lo
- e pode ser mais rápido do que copiar todo o objeto

É o que chamamos de passagem por referência

- E que no C é simulado usando ponteiros
- O Python está fazendo o mesmo que fazemos em C
  - mas esconde isso do programador

Funções com várias saídas

Por que usar passagem por referência?

Funções com várias saídas

Por que usar passagem por referência?

- Porque queremos fazer uma função com várias saídas

Funções com várias saídas

Por que usar passagem por referência?

- Porque queremos fazer uma função com várias saídas

```
1  void min_max(int *v, int n, int *p_min, int *p_max) {
2    int i;
3    *p_max = *p_min = v[0];
4    for (i = 1; i < n; i++) {
5      if (*p_max < v[i])
6        *p_max = v[i];
7      if (*p_min > v[i])
8        *p_min = v[i];
9    }
10 }
11
12 int main() {
13   int v[MAX], n, min, max;
14   scanf("%d", &n);
15   le_vetor(v, n);
16   min_max(v, n, &min, &max);
17   printf("Min: %d, Max: %d\n", min, max);
18   return 0;
19 }
```

Passagem por referência pode ser mais rápida

Por que usar passagem por referência?

Passagem por referência pode ser mais rápida

Por que usar passagem por referência?

- Porque pode ser mais rápido do que copiar a informação

Passagem por referência pode ser mais rápida

Por que usar passagem por referência?

- Porque pode ser mais rápido do que copiar a informação

```
1  typedef struct data {
2    int dia, mes, ano;
3  } data;
4
5  typedef struct aluno {
6    int ra, curso;
7    double cr;
8    char nome[50], endereco[200], email[50];
9    data nascimento, ingresso;
10 } aluno;
11
12 void imprime_aluno(aluno a) {
13   ...
14 }
```

Passagem por referência pode ser mais rápida

Por que usar passagem por referência?

- Porque pode ser mais rápido do que copiar a informação

```
1  typedef struct data {
2    int dia, mes, ano;
3  } data;
4
5  typedef struct aluno {
6    int ra, curso;
7    double cr;
8    char nome[50], endereco[200], email[50];
9    data nascimento, ingresso;
10 } aluno;
11
12 void imprime_aluno(aluno a) {
13   ...
14 }
```

`aluno` é uma struct de `344` bytes, mas `aluno *` tem `8` bytes

Passagem por referência pode fazer mais sentido

Por que usar passagem por referência?

Passagem por referência pode fazer mais sentido

Por que usar passagem por referência?

- Se queremos modificar uma `struct`

Passagem por referência pode fazer mais sentido

Por que usar passagem por referência?

- Se queremos modificar uma `struct`
- então talvez faça mais sentido ter uma função `void`

Passagem por referência pode fazer mais sentido

Por que usar passagem por referência?

- Se queremos modificar uma `struct`
- então talvez faça mais sentido ter uma função `void`
- e passar a `struct` por referência

Passagem por referência pode fazer mais sentido

Por que usar passagem por referência?

- Se queremos modificar uma `struct`
- então talvez faça mais sentido ter uma função `void`
- e passar a `struct` por referência

```
1  typedef struct data {
2    int dia, mes, ano;
3  } data;
4
5  typedef struct aluno {
6    int ra, curso;
7    double cr;
8    char nome[50], endereco[200], email[50];
9    data nascimento, ingresso;
10 } aluno;
```

Passagem por referência pode fazer mais sentido

Por que usar passagem por referência?

- Se queremos modificar uma `struct`
- então talvez faça mais sentido ter uma função `void`
- e passar a `struct` por referência

```
1  typedef struct data {
2    int dia, mes, ano;
3  } data;
4
5  typedef struct aluno {
6    int ra, curso;
7    double cr;
8    char nome[50], endereco[200], email[50];
9    data nascimento, ingresso;
10 } aluno;
```

```
1  void atualiza_email(aluno *a) {
2    ...
3  }
```

Alternativa

Mas muitas vezes podemos evitar passagem por referência

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

```
1  void soma_um(int *x) {
2    *x = *x + 1;
3  }
4
5  int main() {
6    int y = 10;
7    soma_um(&y);
8    printf("%d\n", y);
9    return 0;
10 }
```

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

```c
void soma_um(int *x) {
  *x = *x + 1;
}

int main() {
  int y = 10;
  soma_um(&y);
  printf("%d\n", y);
  return 0;
}
```

```c
int soma_um(int x) {
  return x + 1;
}

int main() {
  int y = 10;
  y = soma_um(y);
  printf("%d\n", y);
  return 0;
}
```

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

```c
void soma_um(int *x) {
  *x = *x + 1;
}

int main() {
  int y = 10;
  soma_um(&y);
  printf("%d\n", y);
  return 0;
}
```

```c
int soma_um(int x) {
  return x + 1;
}

int main() {
  int y = 10;
  y = soma_um(y);
  printf("%d\n", y);
  return 0;
}
```

Idem para o exemplo anterior:

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

```c
void soma_um(int *x) {
  *x = *x + 1;
}

int main() {
  int y = 10;
  soma_um(&y);
  printf("%d\n", y);
  return 0;
}
```

```c
int soma_um(int x) {
  return x + 1;
}

int main() {
  int y = 10;
  y = soma_um(y);
  printf("%d\n", y);
  return 0;
}
```

Idem para o exemplo anterior:

- ao invés de `atualiza_email(&mc202[i]);`

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

```c
void soma_um(int *x) {
  *x = *x + 1;
}

int main() {
  int y = 10;
  soma_um(&y);
  printf("%d\n", y);
  return 0;
}
```

```c
int soma_um(int x) {
  return x + 1;
}

int main() {
  int y = 10;
  y = soma_um(y);
  printf("%d\n", y);
  return 0;
}
```

Idem para o exemplo anterior:

- ao invés de `atualiza_email(&mc202[i]);`
- fazemos `mc202[i] = atualiza_email(mc202[i]);`

Alternativa

Mas muitas vezes podemos evitar passagem por referência

- Se não estamos preocupados com tempo
- E não precisamos modificar mais de uma variável

```
1  void soma_um(int *x) {
2    *x = *x + 1;
3  }
4
5  int main() {
6    int y = 10;
7    soma_um(&y);
8    printf("%d\n", y);
9    return 0;
10 }
```

```
1  int soma_um(int x) {
2    return x + 1;
3  }
4
5  int main() {
6    int y = 10;
7    y = soma_um(y);
8    printf("%d\n", y);
9    return 0;
10 }
```

Idem para o exemplo anterior:

- ao invés de `atualiza_email(&mc202[i]);`
- fazemos `mc202[i] = atualiza_email(mc202[i]);`

Durante o curso, sempre que possível, usaremos essa opção!

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor
  - com 50 posições

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor
  - com 50 posições
  - uma das linhas da matriz

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor
  - com 50 posições
  - uma das linhas da matriz

E vetores são ponteiros em C

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor
  - com 50 posições
  - uma das linhas da matriz

E vetores são ponteiros em C

- ou seja, cada linha é um ponteiro

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor
  - com 50 posições
  - uma das linhas da matriz

E vetores são ponteiros em C

- ou seja, cada linha é um ponteiro
- e a matriz é um vetor de ponteiros

Alocação dinâmica de matrizes

Uma matriz em C é um vetor de vetores

- `int matriz[30][50];` é um vetor
  - com `30` posições
- cada `matriz[i]` é um vetor
  - com 50 posições
  - uma das linhas da matriz

E vetores são ponteiros em C

- ou seja, cada linha é um ponteiro
- e a matriz é um vetor de ponteiros

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

- um vetor de `int` alocado dinamicamente é do tipo `int *`

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

- um vetor de `int` alocado dinamicamente é do tipo `int *`
- uma matriz é um vetor de vetores

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

- um vetor de `int` alocado dinamicamente é do tipo `int *`
- uma matriz é um vetor de vetores
  - um vetor de `int *`

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

- um vetor de `int` alocado dinamicamente é do tipo `int *`
- uma matriz é um vetor de vetores
  - um vetor de `int *`
- portanto, uma matriz é do tipo `int **`

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

- um vetor de `int` alocado dinamicamente é do tipo `int *`
- uma matriz é um vetor de vetores
  - um vetor de `int *`
- portanto, uma matriz é do tipo `int **`
  - armazena um endereço de ponteiro de `int`

Alocando

Qual o tipo de uma matriz alocada dinamicamente?

- um vetor de `int` alocado dinamicamente é do tipo `int *`
- uma matriz é um vetor de vetores
  - um vetor de `int *`
- portanto, uma matriz é do tipo `int **`
  - armazena um endereço de ponteiro de `int`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5    int i, n, m, **matriz;
6    scanf("%d %d", &n, &m);
7    matriz = malloc(n * sizeof(int *));
8    for (i = 0; i < n; i++)
9      matriz[i] = malloc(m * sizeof(int));
10   ...
11   for (i = 0; i < n; i++)
12     free(matriz[i]);
13   free(matriz);
14   return 0;
15 }
```

Ponteiros para ponteiros

Uma matriz é um vetor de vetores

Ponteiros para ponteiros

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

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 }
```

Simular passagem por referência de um ponteiro

Ponteiros para ponteiros

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 }
```

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

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 }
```

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 }
```

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

Ponteiros para ponteiros

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 }
```

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 }
```

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

- No primeiro caso, temos um vetor de vetores

Ponteiros para ponteiros

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 }
```

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 }
```

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

- No primeiro caso, temos um vetor de vetores
- No segundo caso, queremos apontar para outro vetor

Exercício

Faça uma função que dado um vetor v de `int`s e um `int` n, seleciona apenas os elementos de v que são positivos

- guarde os elementos positivos em um outro vetor
- você precisará saber o tamanho desse novo vetor...

Exercício

Faça uma função que dados n, m e k, aloca e devolve uma matriz tridimensional $n \times m \times k$ de `double`s.