MC-202
Curso de C - Parte 1

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

Universidade Estadual de Campinas

2º semestre/2018

Traduzindo de Python para C

```
 1 def maximo(a, b):
 2     if a > b:
 3         return a
 4     else:
 5         return b
 6
 7 def potencia(a, b):
 8     prod = 1
 9     for i in range(b):
10         prod = a * prod
11     return prod
12
13 print("Entre com a e b")
14 a = int(input())
15 b = int(input())
16 maior = maximo(a, b)
17 pot = potencia(a, b)
18 print("Maior:", maior)
19 print("a^b:", pot)
```

Traduzindo de Python para C

```
1  def maximo(a, b):
2      if a > b:
3          return a
4      else:
5          return b
6
7  def potencia(a, b):
8      prod = 1
9      for i in range(b):
10         prod = a * prod
11     return prod
12
13 print("Entre com a e b")
14 a = int(input())
15 b = int(input())
16 maior = maximo(a, b)
17 pot = potencia(a, b)
18 print("Maior:", maior)
19 print("a^b:", pot)
```

Veremos como escrever esse programa em C

Traduzindo de Python para C

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b

def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod

print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Veremos como escrever esse programa em C

- E aprenderemos conceitos da linguagem no processo

Funções

Funções

Em Python

```
1 def maximo(a, b):
2     if a > b:
3         return a
4     else:
5         return b
```

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

A linguagem C é estaticamente tipada:

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

A linguagem C é estaticamente tipada:

- Os tipos das variáveis estão definidos no código

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

A linguagem C é estaticamente tipada:

- Os tipos das variáveis estão definidos no código
- Ao contrário do Python que é dinamicamente tipada

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

A linguagem C é estaticamente tipada:

- Os tipos das variáveis estão definidos no código
- Ao contrário do Python que é dinamicamente tipada
  - Objetos têm tipo, mas variáveis não

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

A linguagem C é estaticamente tipada:

- Os tipos das variáveis estão definidos no código
- Ao contrário do Python que é dinamicamente tipada
  - Objetos têm tipo, mas variáveis não

Existem vários tipos de dados em C:

Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, uma função é declarada como:

- `tipo nome(tipo parametro1, tipo parametro2, ...)`

A linguagem C é estaticamente tipada:

- Os tipos das variáveis estão definidos no código
- Ao contrário do Python que é dinamicamente tipada
  - Objetos têm tipo, mas variáveis não

Existem vários tipos de dados em C:

- `int`, `float`, `double`, `char`, ...

O tipo `int`

O tipo `int` armazena números inteiros

O tipo `int`

O tipo `int` armazena números inteiros

- usualmente de 32 bits, i.e., números em $[-2^{31}, 2^{31} - 1]$

O tipo `int`

O tipo `int` armazena números inteiros

- usualmente de 32 bits, i.e., números em $[-2^{31}, 2^{31} - 1]$
- mas depende do compilador...

O tipo `int`

O tipo `int` armazena números inteiros

- usualmente de 32 bits, i.e., números em $[-2^{31}, 2^{31} - 1]$
- mas depende do compilador...

Algumas operações

`a + b`	soma
`a - b`	subtração
`a * b`	multiplicação
`a / b`	divisão inteira, i.e., `8 / 5` é `1`
`a % b`	resto da divisão, i.e., `8 % 5` é `3`
`a += b`	o mesmo que `a = a + b`
`a -= b`	o mesmo que `a = a - b`
`a *= b`	o mesmo que `a = a * b`
`a /= b`	o mesmo que `a = a / b`
`a %= b`	o mesmo que `a = a % b`
`a++`	o mesmo que `a += 1`
`++a`	o mesmo que `a += 1`
`a--`	o mesmo que `a -= 1`
`--a`	o mesmo que `a -= 1`

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

- Em Python, um bloco começa com `:` e é indentado

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

- Em Python, um bloco começa com `:` e é indentado
- Em C, um bloco é delimitado por `{` e `}`

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

- Em Python, um bloco começa com `:` e é indentado
- Em C, um bloco é delimitado por `{` e `}`
  - Em C, indentação não é obrigatória

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

- Em Python, um bloco começa com `:` e é indentado
- Em C, um bloco é delimitado por `{` e `}`
  - Em C, indentação não é obrigatória
  - Mas é boa pratica de programação

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

- Em Python, um bloco começa com `:` e é indentado
- Em C, um bloco é delimitado por `{` e `}`
  - Em C, indentação não é obrigatória
  - Mas é boa pratica de programação

A maioria das linhas em C são terminadas em `;`

Blocos

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Blocos:

- Em Python, um bloco começa com `:` e é indentado
- Em C, um bloco é delimitado por `{` e `}`
  - Em C, indentação não é obrigatória
  - Mas é boa pratica de programação

A maioria das linhas em C são terminadas em `;`

- Blocos são exceção

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código

- é a função sem o bloco, com a linha terminando com `;`
  exemplo:

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código

- é a função sem o bloco, com a linha terminando com `;` exemplo:

```c
int maximo(int a, int b);
```

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código

- é a função sem o bloco, com a linha terminando com `;` exemplo:

```c
int maximo(int a, int b);
```

- é uma “promessa” que a função existirá no programa

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código

- é a função sem o bloco, com a linha terminando com `;` exemplo:

```c
int maximo(int a, int b);
```

- é uma “promessa” que a função existirá no programa
- permite chamar uma função que será definida depois

Protótipos de Funções

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Muitas vezes é útil definirmos o protótipo da função antes de apresentar o seu código

- é a função sem o bloco, com a linha terminando com `;` exemplo:

```c
int maximo(int a, int b);
```

- é uma “promessa” que a função existirá no programa
- permite chamar uma função que será definida depois
- também será útil para definir Tipos Abstratos de Dados

Condicionais

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Condicionais

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, temos três opções de `if`:

Condicionais

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, temos três opções de `if`:

```c
if (condicao) {
  ...
}
```

Condicionais

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, temos três opções de `if`:

```c
if (condicao) {
   ...
}
```

```c
if (condicao) {
   ...
} else {
   ...
}
```

Condicionais

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, temos três opções de `if`:

```c
if (condicao) {
  ...
}
```

```c
if (condicao) {
  ...
} else {
  ...
}
```

```c
if (condicao) {
  ...
} else if (condicao) {
  ...
} else {
  ...
}
```

Condicionais

Em Python

```python
def maximo(a, b):
    if a > b:
        return a
    else:
        return b
```

Em C

```c
int maximo(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Em C, temos três opções de `if`:

```c
if (condicao) {
  ...
}
```

```c
if (condicao) {
  ...
} else {
  ...
}
```

```c
if (condicao) {
  ...
} else if (condicao) {
  ...
} else {
  ...
}
```

Podemos ter quantos `else if`'s forem necessários

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

Variáveis

Em Python

```
1 def potencia(a, b):
2     prod = 1
3     for i in range(b):
4         prod = a * prod
5     return prod
```

Em C

```
1 int potencia(int a, int b) {
2   int i, prod = 1;
3   for (i = 0; i < b; i++) {
4     prod = a * prod;
5   }
6   return prod;
7 }
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Em C, a variável precisa ser declarada antes de ser usada

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Em C, a variável precisa ser declarada antes de ser usada

- Fazemos isso no início da função

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Em C, a variável precisa ser declarada antes de ser usada

- Fazemos isso no início da função
- `int i;` - declara uma variável de nome `i` do tipo `int`

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Em C, a variável precisa ser declarada antes de ser usada

- Fazemos isso no início da função
- `int i;` - declara uma variável de nome `i` do tipo `int`
- `int i, prod = 1;` - declara `i` e `prod` do tipo `int`

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Em C, a variável precisa ser declarada antes de ser usada

- Fazemos isso no início da função
- `int i;` - declara uma variável de nome `i` do tipo `int`
- `int i, prod = 1;` - declara `i` e `prod` do tipo `int`
  - inicializa `prod` com `1` (opcional)

Variáveis

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em Python, uma variável é declarada automaticamente

- basta atribuir para ela ou definí-la como parâmetro

Em C, a variável precisa ser declarada antes de ser usada

- Fazemos isso no início da função
- `int i;` - declara uma variável de nome `i` do tipo `int`
- `int i, prod = 1;` - declara `i` e `prod` do tipo `int`
  - inicializa `prod` com `1` (opcional)
- Importante: variáveis não inicializadas começam com lixo!

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```c
while (condicao) {
  ...
}
```

← executa enquanto `condicao` for verdadeiro

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```c
while (condicao) {
  ...
}
```

```c
do {
  ...
} while (condicao);
```

← executa enquanto `condicao` for verdadeiro

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```c
while (condicao) {
  ...
}
```

```c
do {
  ...
} while (condicao);
```

← mas sempre executa a primeira vez

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```c
while (condicao) {
  ...
}
```

```c
do {
  ...
} while (condicao);
```

```c
for (inicializacao; condicao; atualizacao) {
  ...
}
```

executada apenas a primeira vez

Laços

Em Python

```
1 def potencia(a, b):
2     prod = 1
3     for i in range(b):
4         prod = a * prod
5     return prod
```

Em C

```
1 int potencia(int a, int b) {
2   int i, prod = 1;
3   for (i = 0; i < b; i++) {
4     prod = a * prod;
5   }
6   return prod;
7 }
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```
1 while (condicao) {
2   ...
3 }
```

```
1 do {
2   ...
3 } while (condicao);
```

```
1 for (inicializacao; condicao; atualizacao) {
2   ...
3 }
```

caso falhe, o laço para

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```c
while (condicao) {
  ...
}
```

```c
do {
  ...
} while (condicao);
```

```c
for (inicializacao; condicao; atualizacao) {
  ...
}
```

na primeira vez, executado após `inicializacao`

Laços

Em Python

```python
def potencia(a, b):
    prod = 1
    for i in range(b):
        prod = a * prod
    return prod
```

Em C

```c
int potencia(int a, int b) {
  int i, prod = 1;
  for (i = 0; i < b; i++) {
    prod = a * prod;
  }
  return prod;
}
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```c
while (condicao) {
  ...
}
```

```c
do {
  ...
} while (condicao);
```

```c
for (inicializacao; condicao; atualizacao) {
  ...
}
```

executada após o bloco

Laços

Em Python

```
1 def potencia(a, b):
2     prod = 1
3     for i in range(b):
4         prod = a * prod
5     return prod
```

Em C

```
1 int potencia(int a, int b) {
2   int i, prod = 1;
3   for (i = 0; i < b; i++) {
4     prod = a * prod;
5   }
6   return prod;
7 }
```

Em C, não há `for ... in`

- mas temos `while`, `do...while` e `for`

```
1 while (condicao) {
2   ...
3 }
```

```
1 do {
2   ...
3 } while (condicao);
```

```
1 for (inicializacao; condicao; atualizacao) {
2   ...
3 }
```

antes de testar `condicao`

Função `main`

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Função `main`

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Em C, a execução do programa começa pela função `main`

Função `main`

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Em C, a execução do programa começa pela função `main`

- Sempre devolve um `int`

Função `main`

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Em C, a execução do programa começa pela função `main`

- Sempre devolve um `int`
- Se devolver `0` significa que não houve erros

Função `main`

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Em C, a execução do programa começa pela função `main`

- Sempre devolve um `int`
- Se devolver `0` significa que não houve erros
  - Valores diferentes indicam o erro que ocorreu

Função `main`

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Em C, a execução do programa começa pela função `main`

- Sempre devolve um `int`
- Se devolver `0` significa que não houve erros
  - Valores diferentes indicam o erro que ocorreu

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

- O `%d` significa substituir por um inteiro

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

- O `%d` significa substituir por um inteiro
  - Existem outras substituições também: `%f`, `%s`, etc...

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

- O `%d` significa substituir por um inteiro
  - Existem outras substituições também: `%f`, `%s`, etc...
- recebe um parâmetro com a string a ser impressa

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

- O `%d` significa substituir por um inteiro
  - Existem outras substituições também: `%f`, `%s`, etc...
- recebe um parâmetro com a string a ser impressa
  - e um parâmetro adicional para cada `%d`, `%f`, `%s`, ...

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

- O `%d` significa substituir por um inteiro
  - Existem outras substituições também: `%f`, `%s`, etc...
- recebe um parâmetro com a string a ser impressa
  - e um parâmetro adicional para cada `%d`, `%f`, `%s`, ...
- a substituição é feita da esquerda para a direita na string

Impressão

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A impressão no C é feita pela função `printf`:

- O `%d` significa substituir por um inteiro
  - Existem outras substituições também: `%f`, `%s`, etc...
- recebe um parâmetro com a string a ser impressa
  - e um parâmetro adicional para cada `%d`, `%f`, `%s`, ...
- a substituição é feita da esquerda para a direita na string
- Não adiciona a quebra de linha `'\n'` automaticamente

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

- String diz quantos valores serão lidos e os seus tipos

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

- String diz quantos valores serão lidos e os seus tipos
- Precisa passar o endereço da variável usando operador `&`

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

- String diz quantos valores serão lidos e os seus tipos
- Precisa passar o endereço da variável usando operador `&`
  - veremos mais sobre isso em breve

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

- String diz quantos valores serão lidos e os seus tipos
- Precisa passar o endereço da variável usando operador `&`
  - veremos mais sobre isso em breve
  - por enquanto, não se esqueça do `&`

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

- String diz quantos valores serão lidos e os seus tipos
- Precisa passar o endereço da variável usando operador `&`
  - veremos mais sobre isso em breve
  - por enquanto, não se esqueça do `&`
- Ignora espaços em branco, tabs e quebras de linha

Leitura

Em Python

```python
print("Entre com a e b")
a = int(input())
b = int(input())
maior = maximo(a, b)
pot = potencia(a, b)
print("Maior:", maior)
print("a^b:", pot)
```

Em C

```c
int main() {
  int a, b, maior, pot;
  printf("Entre com a e b\n");
  scanf("%d %d", &a, &b);
  maior = maximo(a, b);
  pot = potencia(a, b);
  printf("Maior: %d\n", maior);
  printf("a^b: %d\n", pot);
  return 0;
}
```

A leitura no C é feita pela função `scanf`:

- String diz quantos valores serão lidos e os seus tipos
- Precisa passar o endereço da variável usando operador `&`
  - veremos mais sobre isso em breve
  - por enquanto, não se esqueça do `&`
- Ignora espaços em branco, tabs e quebras de linha
  - veremos alguns casos onde isso não acontece...

O programa inteiro

```
 1 #include <stdio.h>
 2
 3 int maximo(int a, int b) {
 4   if (a > b) {
 5     return a;
 6   } else {
 7     return b;
 8   }
 9 }
10
11 int potencia(int a, int b) {
12   int i, prod = 1;
13   for (i = 0; i < b; i++) {
14     prod = a * prod;
15   }
16   return prod;
17 }
18
19 int main() {
20   int a, b, maior, pot;
21   printf("Entre com a e b\n");
22   scanf("%d %d", &a, &b);
23   maior = maximo(a, b);
24   pot = potencia(a, b);
25   printf("Maior: %d\n", maior);
26   printf("a^b: %d\n", pot);
27   return 0;
28 }
```

No começo, colocamos as bibliotecas a serem usadas

- Usamos `stdio.h` por causa de `printf` e `scanf`

Executando o programa

Python é interpretada, C é compilada

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`: usa o padrão C89 (versão mais portável)

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`: usa o padrão C89 (versão mais portável)

`-Wall`: dá mais *warnings* de compilação

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão
`-Werror`:	*warnings* viram erros de compilação

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão
`-Werror`:	*warnings* viram erros de compilação
`-g`:	permite usar gdb e valgrind

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão
`-Werror`:	*warnings* viram erros de compilação
`-g`:	permite usar gdb e valgrind
`-lm`:	permite usar funções matemáticas

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão
`-Werror`:	*warnings* viram erros de compilação
`-g`:	permite usar gdb e valgrind
`-lm`:	permite usar funções matemáticas
`-o`:	define o nome do programa

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão
`-Werror`:	*warnings* viram erros de compilação
`-g`:	permite usar gdb e valgrind
`-lm`:	permite usar funções matemáticas
`-o`:	define o nome do programa

Executando o programa:

Executando o programa

Python é interpretada, C é compilada

- O interpretador do Python abre e executa o seu código
- O compilador do C gera um arquivo executável
  - Depois não depende mais do compilador

Compilando (no terminal):

```
gcc -ansi -Wall -pedantic-errors -Werror -g -lm programa.c -o programa
```

Flags:

`-ansi`:	usa o padrão C89 (versão mais portável)
`-Wall`:	dá mais *warnings* de compilação
`-pedantic-errors`:	força a seguir o padrão
`-Werror`:	*warnings* viram erros de compilação
`-g`:	permite usar gdb e valgrind
`-lm`:	permite usar funções matemáticas
`-o`:	define o nome do programa

Executando o programa:

- `./programa`

O programa refatorado

```
 1 #include <stdio.h>
 2
 3 int maximo(int a, int b) {
 4   if (a > b)
 5     return a;
 6   else
 7     return b;
 8 }
 9
10 int potencia(int a, int b) {
11   int prod = 1, i;
12   for (i = 0; i < b; i++)
13     prod *= a;
14   return prod;
15 }
16
17 int main() {
18   int a, b;
19   printf("Entre com a e b\n");
20   scanf("%d %d", &a, &b);
21   printf("Maximo: %d\na^b: %d\n", maximo(a, b), potencia(a, b));
22   return 0;
23 }
```

Alguns outros detalhes:

O programa refatorado

```
 1 #include <stdio.h>
 2
 3 int maximo(int a, int b) {
 4   if (a > b)
 5     return a;
 6   else
 7     return b;
 8 }
 9
10 int potencia(int a, int b) {
11   int prod = 1, i;
12   for (i = 0; i < b; i++)
13     prod *= a;
14   return prod;
15 }
16
17 int main() {
18   int a, b;
19   printf("Entre com a e b\n");
20   scanf("%d %d", &a, &b);
21   printf("Maximo: %d\na^b: %d\n", maximo(a, b), potencia(a, b));
22   return 0;
23 }
```

Alguns outros detalhes:

- Quando o bloco de um `if`, `else`, `for` ou `while` tiver apenas uma linha, podemos omitir o `{` e `}`

O programa refatorado

```
 1 #include <stdio.h>
 2
 3 int maximo(int a, int b) {
 4   if (a > b)
 5     return a;
 6   else
 7     return b;
 8 }
 9
10 int potencia(int a, int b) {
11   int prod = 1, i;
12   for (i = 0; i < b; i++)
13     prod *= a;
14   return prod;
15 }
16
17 int main() {
18   int a, b;
19   printf("Entre com a e b\n");
20   scanf("%d %d", &a, &b);
21   printf("Maximo: %d\na^b: %d\n", maximo(a, b), potencia(a, b));
22   return 0;
23 }
```

Alguns outros detalhes:

- Quando o bloco de um `if`, `else`, `for` ou `while` tiver apenas uma linha, podemos omitir o `{` e `}`
- Podemos escrever `prod *= a;` na linha 13

O programa refatorado

```
 1 #include <stdio.h>
 2
 3 int maximo(int a, int b) {
 4   if (a > b)
 5     return a;
 6   else
 7     return b;
 8 }
 9
10 int potencia(int a, int b) {
11   int prod = 1, i;
12   for (i = 0; i < b; i++)
13     prod *= a;
14   return prod;
15 }
16
17 int main() {
18   int a, b;
19   printf("Entre com a e b\n");
20   scanf("%d %d", &a, &b);
21   printf("Maximo: %d\na^b: %d\n", maximo(a, b), potencia(a, b));
22   return 0;
23 }
```

Alguns outros detalhes:

- Quando o bloco de um `if`, `else`, `for` ou `while` tiver apenas uma linha, podemos omitir o `{` e `}`
- Podemos escrever `prod *= a;` na linha 13
- O `printf` pode imprimir os resultados de funções

Um programa com listas/vetores

Um programa com listas/vetores

Em Python

```
1 print("Digite 10 números")
2 lista = []
3 for i in range(10):
4     lista.append(int(input()))
5 print("Positivos")
6 for x in lista:
7     if x > 0:
8         print(x)
```

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Em C, as listas são bem diferentes em relação ao Python:

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Em C, as listas são bem diferentes em relação ao Python:

- São chamadas de vetores ou *arrays*

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Em C, as listas são bem diferentes em relação ao Python:

- São chamadas de vetores ou *arrays*
- Todos os elementos são sempre do mesmo tipo

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Em C, as listas são bem diferentes em relação ao Python:

- São chamadas de vetores ou *arrays*
- Todos os elementos são sempre do mesmo tipo
- Têm tamanho fixo definido na declaração da variável

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Em C, as listas são bem diferentes em relação ao Python:

- São chamadas de vetores ou *arrays*
- Todos os elementos são sempre do mesmo tipo
- Têm tamanho fixo definido na declaração da variável
- Exemplo de declaração: `int lista[10];`

Um programa com listas/vetores

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Em C, as listas são bem diferentes em relação ao Python:

- São chamadas de vetores ou *arrays*
- Todos os elementos são sempre do mesmo tipo
- Têm tamanho fixo definido na declaração da variável
- Exemplo de declaração: `int lista[10];`
  - Define uma lista de 10 `int`s

Um programa com listas

Em Python

```
1 print("Digite 10 números")
2 lista = []
3 for i in range(10):
4     lista.append(int(input()))
5 print("Positivos")
6 for x in lista:
7     if x > 0:
8         print(x)
```

Em C

```
1  #include <stdio.h>
2
3  int main() {
4    int i, lista[10];
5    printf("Digite 10 números\n");
6    for (i = 0; i < 10; i++)
7      scanf("%d", &lista[i]);
8    printf("Positivos\n");
9    for (i = 0; i < 10; i++) {
10     if (lista[i] > 0)
11       printf("%d\n", lista[i]);
12   }
13   return 0;
14 }
```

Um programa com listas

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Cada `lista[i]` é um `int`

Um programa com listas

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Cada `lista[i]` é um `int`

- Imprimir `lista[i]`:

Um programa com listas

Em Python

```
1 print("Digite 10 números")
2 lista = []
3 for i in range(10):
4     lista.append(int(input()))
5 print("Positivos")
6 for x in lista:
7     if x > 0:
8         print(x)
```

Em C

```
1  #include <stdio.h>
2
3  int main() {
4    int i, lista[10];
5    printf("Digite 10 números\n");
6    for (i = 0; i < 10; i++)
7      scanf("%d", &lista[i]);
8    printf("Positivos\n");
9    for (i = 0; i < 10; i++) {
10     if (lista[i] > 0)
11       printf("%d\n", lista[i]);
12   }
13   return 0;
14 }
```

Cada `lista[i]` é um `int`

- Imprimir `lista[i]`:
  `printf("%d", lista[i]);`

Um programa com listas

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Cada `lista[i]` é um `int`

- Imprimir `lista[i]`:
  `printf("%d", lista[i]);`
- Ler um número e guardar em `lista[i]`:

Um programa com listas

Em Python

```python
print("Digite 10 números")
lista = []
for i in range(10):
    lista.append(int(input()))
print("Positivos")
for x in lista:
    if x > 0:
        print(x)
```

Em C

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Cada `lista[i]` é um `int`

- Imprimir `lista[i]`:
  `printf("%d", lista[i]);`
- Ler um número e guardar em `lista[i]`:
  `scanf("%d", &lista[i]);`

Refatoração

```
 1 #include <stdio.h>
 2
 3 int main() {
 4   int i, lista[10];
 5   printf("Digite 10 números\n");
 6   for (i = 0; i < 10; i++)
 7     scanf("%d", &lista[i]);
 8   printf("Positivos\n");
 9   for (i = 0; i < 10; i++) {
10     if (lista[i] > 0)
11       printf("%d\n", lista[i]);
12   }
13   return 0;
14 }
```

Refatoração

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Podemos melhorar esse código:

Refatoração

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Podemos melhorar esse código:

- Ter uma função que lê vetores

Refatoração

```c
#include <stdio.h>

int main() {
  int i, lista[10];
  printf("Digite 10 números\n");
  for (i = 0; i < 10; i++)
    scanf("%d", &lista[i]);
  printf("Positivos\n");
  for (i = 0; i < 10; i++) {
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
  }
  return 0;
}
```

Podemos melhorar esse código:

- Ter uma função que lê vetores
- Ter uma função que imprime apenas os positivos

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

A função recebe um vetor chamado `lista`:

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

A função recebe um vetor chamado `lista`:

- Não precisamos especificar o tamanho entre o `[]`

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

A função recebe um vetor chamado `lista`:

- Não precisamos especificar o tamanho entre o `[]`
  - apenas quando é um parâmetro

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

A função recebe um vetor chamado `lista`:

- Não precisamos especificar o tamanho entre o `[]`
  - apenas quando é um parâmetro
- É nossa responsabilidade saber o tamanho do vetor

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

A função recebe um vetor chamado `lista`:

- Não precisamos especificar o tamanho entre o `[]`
  - apenas quando é um parâmetro
- É nossa responsabilidade saber o tamanho do vetor
  - Por isso precisamos do parâmetro `n`

Imprimindo números positivos

```
1 void imprime_positivos(int lista[], int n) {
2   int i;
3   printf("Positivos\n");
4   for (i = 0; i < n; i++)
5     if (lista[i] > 0)
6       printf("%d\n", lista[i]);
7 }
```

A função é do tipo `void`:

- Significa que a função não devolve valor

A função recebe um vetor chamado `lista`:

- Não precisamos especificar o tamanho entre o `[]`
  - apenas quando é um parâmetro
- É nossa responsabilidade saber o tamanho do vetor
  - Por isso precisamos do parâmetro `n`
  - No C, não há o equivalente ao `len()` do Python

Lendo um vetor

Em C, não é possível devolver um vetor...

Lendo um vetor

Em C, não é possível devolver um vetor...

- Passamos um vetor como parâmetro

Lendo um vetor

Em C, não é possível devolver um vetor...

- Passamos um vetor como parâmetro
- Modificamos o seu conteúdo

Lendo um vetor

Em C, não é possível devolver um vetor...

- Passamos um vetor como parâmetro
- Modificamos o seu conteúdo

```
1 void le_vetor(int lista[], int n) {
2   int i;
3   printf("Digite %d números\n", n);
4   for (i = 0; i < n; i++)
5     scanf("%d", &lista[i]);
6 }
```

Lendo um vetor

Em C, não é possível devolver um vetor...

- Passamos um vetor como parâmetro
- Modificamos o seu conteúdo

```
1 void le_vetor(int lista[], int n) {
2   int i;
3   printf("Digite %d números\n", n);
4   for (i = 0; i < n; i++)
5     scanf("%d", &lista[i]);
6 }
```

A função modifica o conteúdo do vetor `lista`

Lendo um vetor

Em C, não é possível devolver um vetor...

- Passamos um vetor como parâmetro
- Modificamos o seu conteúdo

```
1 void le_vetor(int lista[], int n) {
2   int i;
3   printf("Digite %d números\n", n);
4   for (i = 0; i < n; i++)
5     scanf("%d", &lista[i]);
6 }
```

A função modifica o conteúdo do vetor `lista`

- Entenderemos isso melhor em breve...

Código completo

```c
#include <stdio.h>

void le_vetor(int lista[], int n) {
  int i;
  printf("Digite %d números\n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &lista[i]);
}

void imprime_positivos(int lista[], int n) {
  int i;
  printf("Positivos\n");
  for (i = 0; i < n; i++)
    if (lista[i] > 0)
      printf("%d\n", lista[i]);
}

int main() {
  int lista[10];
  le_vetor(lista, 10);
  imprime_positivos(lista, 10);
  return 0;
}
```

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

No C, um vetor é um bloco contíguo de memória

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

No C, um vetor é um bloco contíguo de memória

- E o C assume que você usará o bloco corretamente

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

No C, um vetor é um bloco contíguo de memória

- E o C assume que você usará o bloco corretamente
- Não há checagem dos limites do vetor

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

No C, um vetor é um bloco contíguo de memória

- E o C assume que você usará o bloco corretamente
- Não há checagem dos limites do vetor

O que ocorre muitas vezes é *off-by-one*

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

No C, um vetor é um bloco contíguo de memória

- E o C assume que você usará o bloco corretamente
- Não há checagem dos limites do vetor

O que ocorre muitas vezes é *off-by-one*

- Se o vetor tem `n` posições,

Cuidados com vetores em C

A responsabilidade de acessar apenas posições válidas é sua!

- Se você declarou um vetor com 10 posições
- E acessar a posição 10, 11, 12, etc...
  - Ou você terá um erro de execução: `segmentation fault`
  - Ou não...
    - Se for impressão, pode imprimir o valor de outra variável
    - Se for escrita, pode mudar o valor de outra variável

No C, um vetor é um bloco contíguo de memória

- E o C assume que você usará o bloco corretamente
- Não há checagem dos limites do vetor

O que ocorre muitas vezes é *off-by-one*

- Se o vetor tem `n` posições,
- você não deve acessar a posição `n`

Variáveis globais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`global` é uma variável global:

Variáveis globais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`global` é uma variável global:

- pode ser acessada em qualquer função

Variáveis globais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`global` é uma variável global:

- pode ser acessada em qualquer função
- variáveis globais só são usadas em casos específicos

Variáveis globais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`global` é uma variável global:

- pode ser acessada em qualquer função
- variáveis globais só são usadas em casos específicos
- podem levar a erros difíceis de encontrar no programa

Variáveis locais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

Variáveis locais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`local`, `local1`, `local2` e `parametro` são variáveis locais:

Variáveis locais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`local`, `local1`, `local2` e `parametro` são variáveis locais:

- existem apenas dentro da função onde foram definidas

Variáveis locais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`local`, `local1`, `local2` e `parametro` são variáveis locais:

- existem apenas dentro da função onde foram definidas
- `local1` de `funcao1` é diferente de `local1` de `funcao2`

Variáveis locais

```
1  #include <stdio.h>
2
3  int global;
4
5  void funcao1(int parametro) {
6    int local1, local2;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int local1, local2;
12   ...
13 }
14
15 int main() {
16   int local;
17 }
```

`local`, `local1`, `local2` e `parametro` são variáveis locais:

- existem apenas dentro da função onde foram definidas
- `local1` de `funcao1` é diferente de `local1` de `funcao2`
- quando a função acaba, o valor é perdido

Sobreposição de escopo

```
1  #include <stdio.h>
2
3  int x;
4
5  void funcao1(int parametro) {
6    x = 10;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int x;
12   x = 10;
13 }
```

Sobreposição de escopo

```
1  #include <stdio.h>
2
3  int x;
4
5  void funcao1(int parametro) {
6    x = 10;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int x;
12   x = 10;
13 }
```

Variáveis locais têm precedência sobre variáveis globais

Sobreposição de escopo

```
1  #include <stdio.h>
2
3  int x;
4
5  void funcao1(int parametro) {
6    x = 10;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int x;
12   x = 10;
13 }
```

Variáveis locais têm precedência sobre variáveis globais

- Em `funcao1`, a variável global `x` tem seu valor alterado

Sobreposição de escopo

```
1  #include <stdio.h>
2
3  int x;
4
5  void funcao1(int parametro) {
6    x = 10;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int x;
12   x = 10;
13 }
```

Variáveis locais têm precedência sobre variáveis globais

- Em `funcao1`, a variável global `x` tem seu valor alterado
- Em `funcao2`, a variável local `x` tem seu valor alterado

Sobreposição de escopo

```
1  #include <stdio.h>
2
3  int x;
4
5  void funcao1(int parametro) {
6    x = 10;
7    ...
8  }
9
10 void funcao2(int parametro) {
11   int x;
12   x = 10;
13 }
```

Variáveis locais têm precedência sobre variáveis globais

- Em `funcao1`, a variável global `x` tem seu valor alterado
- Em `funcao2`, a variável local `x` tem seu valor alterado

Um dos motivos que evitamos o uso de variáveis globais!

Variáveis e Funções

Variáveis e Funções

```
1  #include <stdio.h>
2
3  void soma_um(int x) {
4    x = x + 1;
5  }
6
7  int main() {
8    int x = 1;
9    soma_um(x);
10   printf("%d ", x);
11   return 0;
12 }
```

Variáveis e Funções

```c
#include <stdio.h>

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

int main() {
  int x = 1;
  soma_um(x);
  printf("%d ", x);
  return 0;
}
```

```c
#include <stdio.h>

void soma_um(int v[], int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i]++;
}

int main() {
  int i, v[5] = {1, 2, 3, 4, 5};
  soma_um(v, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  return 0;
}
```

Variáveis e Funções

```c
#include <stdio.h>

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

int main() {
  int x = 1;
  soma_um(x);
  printf("%d ", x);
  return 0;
}
```

```c
#include <stdio.h>

void soma_um(int v[], int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i]++;
}

int main() {
  int i, v[5] = {1, 2, 3, 4, 5};
  soma_um(v, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  return 0;
}
```

No código da esquerda é impresso 1

Variáveis e Funções

```c
#include <stdio.h>

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

int main() {
  int x = 1;
  soma_um(x);
  printf("%d ", x);
  return 0;
}
```

```c
#include <stdio.h>

void soma_um(int v[], int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i]++;
}

int main() {
  int i, v[5] = {1, 2, 3, 4, 5};
  soma_um(v, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  return 0;
}
```

No código da esquerda é impresso `1`

- A variável `x` de `main` é diferente da variável `x` de `soma_um`

Variáveis e Funções

```c
#include <stdio.h>

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

int main() {
  int x = 1;
  soma_um(x);
  printf("%d ", x);
  return 0;
}
```

```c
#include <stdio.h>

void soma_um(int v[], int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i]++;
}

int main() {
  int i, v[5] = {1, 2, 3, 4, 5};
  soma_um(v, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  return 0;
}
```

No código da esquerda é impresso `1`

- A variável `x` de `main` é diferente da variável `x` de `soma_um`

No código da direita é impresso `2 3 4 5 6`

Variáveis e Funções

```c
#include <stdio.h>

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

int main() {
  int x = 1;
  soma_um(x);
  printf("%d ", x);
  return 0;
}
```

```c
#include <stdio.h>

void soma_um(int v[], int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i]++;
}

int main() {
  int i, v[5] = {1, 2, 3, 4, 5};
  soma_um(v, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  return 0;
}
```

No código da esquerda é impresso `1`

- A variável `x` de `main` é diferente da variável `x` de `soma_um`

No código da direita é impresso `2 3 4 5 6`

- A função altera o conteúdo do vetor

Variáveis e Funções

```c
#include <stdio.h>

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

int main() {
  int x = 1;
  soma_um(x);
  printf("%d ", x);
  return 0;
}
```

```c
#include <stdio.h>

void soma_um(int v[], int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i]++;
}

int main() {
  int i, v[5] = {1, 2, 3, 4, 5};
  soma_um(v, 5);
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  return 0;
}
```

No código da esquerda é impresso `1`

- A variável `x` de `main` é diferente da variável `x` de `soma_um`

No código da direita é impresso `2 3 4 5 6`

- A função altera o conteúdo do vetor
- Entenderemos o motivo disso posteriormente...

Exercício

a) Escreva um programa completo em C que lê dois vetores de 100 números inteiros, armazena o produto de Hadamard destes vetores em um terceiro vetor e imprime esse terceiro vetor.

O Produto de Hadamard de dois vetores u e v é o produto ponto-a-ponto de u e v, isto é, o vetor $(u_1v_1, u_2v_2, \ldots, u_nv_n)$.

Exercício

a) Escreva um programa completo em C que lê dois vetores de 100 números inteiros, armazena o produto de Hadamard destes vetores em um terceiro vetor e imprime esse terceiro vetor.

O Produto de Hadamard de dois vetores u e v é o produto ponto-a-ponto de u e v, isto é, o vetor $(u_1v_1, u_2v_2, \ldots, u_nv_n)$.

b) Modifique o programa para calcular o produto de Hadamard de dois vetores de tamanho menor ou igual a 100 (dado na entrada).