

# Algoritmos

Paulo Torrens

paulotorrens@gnu.org

Departamento de Ciência da Computação  
Centro de Ciências e Tecnologias  
Universidade do Estado de Santa Catarina

2020/1

# Conceitos Básicos de Algoritmos

- O que é um “algoritmo”?
- Embora definido formalmente há menos de um século, o conceito de algoritmo é muito mais antigo
- Algoritmos descrevem um método, a ser resolvido por um computador, para a solução de um problema
  - Quando originalmente definido, “computador” se referia às pessoas que efetuavam os cálculos, e não a uma máquina
  - Costumamos descrever os problemas em termos de entradas e de saídas; por exemplo, se temos um algoritmo para efetuarmos uma divisão, temos o divisor e o dividendo como entradas, e o resultado será nossa saída
- É possível representarmos algoritmos, informalmente, através de fluxogramas e pseudocódigo

# Conceitos Básicos de Algoritmos

- O que é um “algoritmo”?
- Embora definido formalmente há menos de um século, o conceito de algoritmo é muito mais antigo
- Algoritmos descrevem um método, a ser resolvido por um computador, para a solução de um problema
  - Quando originalmente definido, “computador” se referia às pessoas que efetuavam os cálculos, e não a uma máquina
  - Costumamos descrever os problemas em termos de entradas e de saídas; por exemplo, se temos um algoritmo para efetuarmos uma divisão, temos o divisor e o dividendo como entradas, e o resultado será nossa saída
- É possível representarmos algoritmos, informalmente, através de fluxogramas e pseudocódigo

# Conceitos Básicos de Algoritmos

- O que é um “algoritmo”?
- Embora definido formalmente há menos de um século, o conceito de algoritmo é muito mais antigo
- Algoritmos descrevem um método, a ser resolvido por um computador, para a solução de um problema
  - Quando originalmente definido, “computador” se referia às pessoas que efetuavam os cálculos, e não a uma máquina
  - Costumamos descrever os problemas em termos de entradas e de saídas; por exemplo, se temos um algoritmo para efetuarmos uma divisão, temos o divisor e o dividendo como entradas, e o resultado será nossa saída
- É possível representarmos algoritmos, informalmente, através de fluxogramas e pseudocódigo

# Conceitos Básicos de Algoritmos

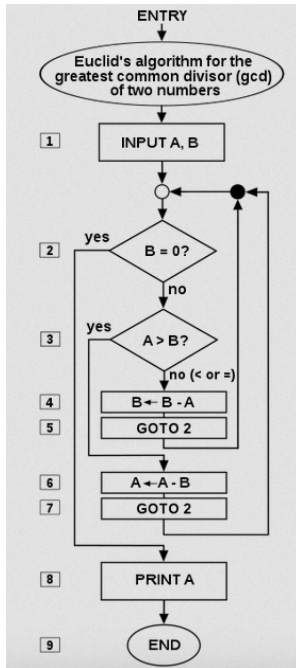
- O que é um “algoritmo”?
- Embora definido formalmente há menos de um século, o conceito de algoritmo é muito mais antigo
- Algoritmos descrevem um método, a ser resolvido por um computador, para a solução de um problema
  - Quando originalmente definido, “computador” se referia às pessoas que efetuavam os cálculos, e não a uma máquina
  - Costumamos descrever os problemas em termos de entradas e de saídas; por exemplo, se temos um algoritmo para efetuarmos uma divisão, temos o divisor e o dividendo como entradas, e o resultado será nossa saída
- É possível representarmos algoritmos, informalmente, através de fluxogramas e pseudocódigo

# Conceitos Básicos de Algoritmos

- O que é um “algoritmo”?
- Embora definido formalmente há menos de um século, o conceito de algoritmo é muito mais antigo
- Algoritmos descrevem um método, a ser resolvido por um computador, para a solução de um problema
  - Quando originalmente definido, “computador” se referia às pessoas que efetuavam os cálculos, e não a uma máquina
  - Costumamos descrever os problemas em termos de entradas e de saídas; por exemplo, se temos um algoritmo para efetuarmos uma divisão, temos o divisor e o dividendo como entradas, e o resultado será nossa saída
- É possível representarmos algoritmos, informalmente, através de fluxogramas e pseudocódigo

# Conceitos Básicos de Algoritmos

- O que é um “algoritmo”?
- Embora definido formalmente há menos de um século, o conceito de algoritmo é muito mais antigo
- Algoritmos descrevem um método, a ser resolvido por um computador, para a solução de um problema
  - Quando originalmente definido, “computador” se referia às pessoas que efetuavam os cálculos, e não a uma máquina
  - Costumamos descrever os problemas em termos de entradas e de saídas; por exemplo, se temos um algoritmo para efetuarmos uma divisão, temos o divisor e o dividendo como entradas, e o resultado será nossa saída
- É possível representarmos algoritmos, informalmente, através de fluxogramas e pseudocódigo





# Linguagens de programação

- Quando nos referimos à programação, utilizamos uma linguagem de programação para descrever essa “receita de bolo”
- Para executarmos um algoritmo em um computador, entretanto, precisamos de uma forma de instruí-lo, isto é, uma forma de representar os passos que queremos que sejam efetuados de uma forma que a máquina seja capaz de entender
- Linguagens de programação são linguagens formais (com regras e significado definidos) que descrevem formas pelas quais podemos representar algoritmos
- Todo sistema de *software* que usamos foi, então, programado através de uma linguagem de programação, instruindo um computador a como se comportar

# Linguagens de programação

- Quando nos referimos à programação, utilizamos uma linguagem de programação para descrever essa “receita de bolo”
- Para executarmos um algoritmo em um computador, entretanto, precisamos de uma forma de instruí-lo, isto é, uma forma de representar os passos que queremos que sejam efetuados de uma forma que a máquina seja capaz de entender
- Linguagens de programação são linguagens formais (com regras e significado definidos) que descrevem formas pelas quais podemos representar algoritmos
- Todo sistema de *software* que usamos foi, então, programado através de uma linguagem de programação, instruindo um computador a como se comportar

# Linguagens de programação

- Quando nos referimos à programação, utilizamos uma linguagem de programação para descrever essa “receita de bolo”
- Para executarmos um algoritmo em um computador, entretanto, precisamos de uma forma de instruí-lo, isto é, uma forma de representar os passos que queremos que sejam efetuados de uma forma que a máquina seja capaz de entender
- Linguagens de programação são linguagens formais (com regras e significado definidos) que descrevem formas pelas quais podemos representar algoritmos
- Todo sistema de *software* que usamos foi, então, programado através de uma linguagem de programação, instruindo um computador a como se comportar

# Linguagens de programação

- Quando nos referimos à programação, utilizamos uma linguagem de programação para descrever essa “receita de bolo”
- Para executarmos um algoritmo em um computador, entretanto, precisamos de uma forma de instruí-lo, isto é, uma forma de representar os passos que queremos que sejam efetuados de uma forma que a máquina seja capaz de entender
- Linguagens de programação são linguagens formais (com regras e significado definidos) que descrevem formas pelas quais podemos representar algoritmos
- Todo sistema de *software* que usamos foi, então, programado através de uma linguagem de programação, instruindo um computador a como se comportar

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...



# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Linguagens de programação

- Nosso objetivo é aprender os conceitos básicos necessários para desenvolver algoritmos e programas de computador, além do básico de linguagens de programação imperativas
- Constantes e variáveis...
- Testes de mesa...
- Identificadores e palavras reservadas...
- Indentação e notação secundária...
- Conceitos básicos de tipos...
- Operadores lógicos e aritméticos...
- Estruturas básicas de controle...

# Constantes e variáveis

- Linguagens imperativas representam algoritmos utilizando o conceito de **variáveis**, nomes usados pelo programador para solicitar que o computador se lembre de um valor
- Em várias linguagens, as variáveis devem ser **declaradas**, quer dizer, precisamos informar que utilizaremos tal variável antes de realmente a utilizarmos, além de informar seu **tipo**
- Após uma variável ser declarada, ela pode ser **atribuída**: podemos solicitar que o programa altere o valor de memória de uma variável, colocando outro valor em seu lugar
- Podemos visualizar a memória como uma **tabela**, onde temos o nome atribuído pelo programador e o último valor atribuído
- Caso uma variável vá ser atribuída uma única vez, e não será posteriormente alterada, podemos considerá-la uma **constante**

# Constantes e variáveis

- Linguagens imperativas representam algoritmos utilizando o conceito de **variáveis**, nomes usados pelo programador para solicitar que o computador se lembre de um valor
- Em várias linguagens, as variáveis devem ser **declaradas**, quer dizer, precisamos informar que utilizaremos tal variável antes de realmente a utilizarmos, além de informar seu **tipo**
- Após uma variável ser declarada, ela pode ser **atribuída**: podemos solicitar que o programa altere o valor de memória de uma variável, colocando outro valor em seu lugar
- Podemos visualizar a memória como uma **tabela**, onde temos o nome atribuído pelo programador e o último valor atribuído
- Caso uma variável vá ser atribuída uma única vez, e não será posteriormente alterada, podemos considerá-la uma **constante**

# Constantes e variáveis

- Linguagens imperativas representam algoritmos utilizando o conceito de **variáveis**, nomes usados pelo programador para solicitar que o computador se lembre de um valor
- Em várias linguagens, as variáveis devem ser **declaradas**, quer dizer, precisamos informar que utilizaremos tal variável antes de realmente a utilizarmos, além de informar seu **tipo**
- Após uma variável ser declarada, ela pode ser **atribuída**: podemos solicitar que o programa altere o valor de memória de uma variável, colocando outro valor em seu lugar
- Podemos visualizar a memória como uma **tabela**, onde temos o nome atribuído pelo programador e o último valor atribuído
- Caso uma variável vá ser atribuída uma única vez, e não será posteriormente alterada, podemos considerá-la uma **constante**

# Constantes e variáveis

- Linguagens imperativas representam algoritmos utilizando o conceito de **variáveis**, nomes usados pelo programador para solicitar que o computador se lembre de um valor
- Em várias linguagens, as variáveis devem ser **declaradas**, quer dizer, precisamos informar que utilizaremos tal variável antes de realmente a utilizarmos, além de informar seu **tipo**
- Após uma variável ser declarada, ela pode ser **atribuída**: podemos solicitar que o programa altere o valor de memória de uma variável, colocando outro valor em seu lugar
- Podemos visualizar a memória como uma **tabela**, onde temos o nome atribuído pelo programador e o último valor atribuído
- Caso uma variável vá ser atribuída uma única vez, e não será posteriormente alterada, podemos considerá-la uma **constante**



- Linguagens imperativas representam algoritmos utilizando o conceito de **variáveis**, nomes usados pelo programador para solicitar que o computador se lembre de um valor
- Em várias linguagens, as variáveis devem ser **declaradas**, quer dizer, precisamos informar que utilizaremos tal variável antes de realmente a utilizarmos, além de informar seu **tipo**
- Após uma variável ser declarada, ela pode ser **atribuída**: podemos solicitar que o programa altere o valor de memória de uma variável, colocando outro valor em seu lugar
- Podemos visualizar a memória como uma **tabela**, onde temos o nome atribuído pelo programador e o último valor atribuído
- Caso uma variável vá ser atribuída uma única vez, e não será posteriormente alterada, podemos considerá-la uma **constante**

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     → int a = 42;
6       int b = 51;
7       // Soma a e b
8       int c = a + b;
9       print("c = ", c);
10      return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
------	-------

Terminal:

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     → int a = 42;
6       int b = 51;
7       // Soma a e b
8       int c = a + b;
9       print("c = ", c);
10      return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
------	-------

Terminal:

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     ✓ int a = 42;
6     → int b = 51;
7     // Soma a e b
8     int c = a + b;
9     print("c = ", c);
10    return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
a	42

Terminal:

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     ✓ int a = 42;
6     ✓ int b = 51;
7     // Soma a e b
8     → int c = a + b;
9     print("c = ", c);
10    return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
a	42
<b>b</b>	<b>51</b>

Terminal:

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     ✓ int a = 42;
6     ✓ int b = 51;
7     // Soma a e b
8     ✓ int c = a + b;
9     → print("c = ", c);
10    return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
a	42
b	51
c	93

Terminal:

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     ✓ int a = 42;
6     ✓ int b = 51;
7     // Soma a e b
8     ✓ int c = a + b;
9     ✓ print("c = ", c);
10    → return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
a	42
b	51
c	93

Terminal:

**c = 93**

# Constantes e variáveis

Conforme vamos executando, passo a passo, um algoritmo, podemos atualizar sua memória. Anotamos a próxima linha.

Código:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     ✓ int a = 42;
6     ✓ int b = 51;
7     // Soma a e b
8     ✓ int c = a + b;
9     ✓ print("c = ", c);
10    ✓ return EXIT_SUCCESS;
11 }
```

Memória:

Nome	Valor
a	42
b	51
c	93

Terminal:

c = 93

**O programa terminou!**



# Constantes e variáveis

- Em C, nós **declaramos** variáveis usando um tipo (int,char, float, etc) seguido do nome da variável
- Nós **atribuímos** uma variável através do nome da variável seguido de um símbolo de igual e uma **expressão** com o valor que queremos que seja salvo
- É possível declarar e imediatamente atribuir um valor a uma variável; uma que foi declarada, porém ainda não foi definida, possui um valor indeterminado

```
1 // Declaração
2 int meu_int;
3 // Declaração e atribuição
4 int x = 10;
5 int y = 20;
6 // Atribuição
7 meu_int = x + y;
```

# Constantes e variáveis

- Em C, nós **declaramos** variáveis usando um tipo (int,char, float, etc) seguido do nome da variável
- Nós **atribuímos** uma variável através do nome da variável seguido de um símbolo de igual e uma **expressão** com o valor que queremos que seja salvo
- É possível declarar e imediatamente atribuir um valor a uma variável; uma que foi declarada, porém ainda não foi definida, possui um valor indeterminado

```
1 // Declaração
2 int meu_int;
3 // Declaração e atribuição
4 int x = 10;
5 int y = 20;
6 // Atribuição
7 meu_int = x + y;
```

# Constantes e variáveis

- Em C, nós **declaramos** variáveis usando um tipo (int,char, float, etc) seguido do nome da variável
- Nós **atribuímos** uma variável através do nome da variável seguido de um símbolo de igual e uma **expressão** com o valor que queremos que seja salvo
- É possível declarar e imediatamente atribuir um valor a uma variável; uma que foi declarada, porém ainda não foi definida, **possui um valor indeterminado**

```
1 // Declaração
2 int meu_int;
3 // Declaração e atribuição
4 int x = 10;
5 int y = 20;
6 // Atribuição
7 meu_int = x + y;
```

# Constantes e variáveis

- Em C, nós **declaramos** variáveis usando um tipo (int,char, float, etc) seguido do nome da variável
- Nós **atribuímos** uma variável através do nome da variável seguido de um símbolo de igual e uma **expressão** com o valor que queremos que seja salvo
- É possível declarar e imediatamente atribuir um valor a uma variável; uma que foi declarada, porém ainda não foi definida, **possui um valor indeterminado**

```
1 // Declaração
2 int meu_int;
3 // Declaração e atribuição
4 int x = 10;
5 int y = 20;
6 // Atribuição
7 meu_int = x + y;
```

# Teste de mesa

- A fim de verificar o funcionamento de um algoritmo, podemos efetuar um **teste de mesa**, executando o algoritmo com “papel e caneta”, manualmente
- Como representado no exemplo anterior, para cada passo, anotamos a **próxima instrução a ser executada** e o **registro de memória**, o qual representamos como uma tabela ou lista
- Seguindo o fluxo de um programa, seja ele representado por um fluxograma ou código, podemos testar nosso raciocínio, encontrar falhas, e garantir que nosso algoritmo se comporta como esperamos

# Teste de mesa

- A fim de verificar o funcionamento de um algoritmo, podemos efetuar um **teste de mesa**, executando o algoritmo com “papel e caneta”, manualmente
- Como representado no exemplo anterior, para cada passo, anotamos a **próxima instrução a ser executada** e o **registro de memória**, o qual representamos como uma tabela ou lista
- Seguindo o fluxo de um programa, seja ele representado por um fluxograma ou código, podemos testar nosso raciocínio, encontrar falhas, e garantir que nosso algoritmo se comporta como esperamos

- A fim de verificar o funcionamento de um algoritmo, podemos efetuar um **teste de mesa**, executando o algoritmo com “papel e caneta”, manualmente
- Como representado no exemplo anterior, para cada passo, anotamos a **próxima instrução a ser executada** e o **registro de memória**, o qual representamos como uma tabela ou lista
- Seguindo o fluxo de um programa, seja ele representado por um fluxograma ou código, podemos testar nosso raciocínio, encontrar falhas, e garantir que nosso algoritmo se comporta como esperamos

## Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     → int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

## Estado:

PL	a	b
3		



Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     → int b = 5;
5
6     while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

Estado:

PL	a	b
3		
4	10	

Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     → while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

Estado:

PL	a	b
3		
4	10	
6	10	5

## Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         →     if(a > b) {
8                 a = a - b;
9             } else {
10                b = b - a;
11            }
12        }
13
14        return a;
15    }
```

PL = Próxima Linha

## Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5

## Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         if(a > b) {
8             →      a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

## Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5

## Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     → while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

## Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5
6	5	5

## Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         →     if(a > b) {
8                 a = a - b;
9             } else {
10                b = b - a;
11            }
12        }
13
14    return a;
15 }
```

PL = Próxima Linha

## Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5
6	5	5
7	5	5

Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10      →      b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5
6	5	5
7	5	5
10	5	5

Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     → while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5
6	5	5
7	5	5
10	5	5
6	5	0



Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     → return a;
15 }
```

PL = Próxima Linha

Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5
6	5	5
7	5	5
10	5	5
6	5	0
14	5	0

## Código:

```
1 // Máximo divisor comum
2 int euclides() {
3     int a = 10;
4     int b = 5;
5
6     while(b != 0) {
7         if(a > b) {
8             a = a - b;
9         } else {
10             b = b - a;
11         }
12     }
13
14     return a;
15 }
```

PL = Próxima Linha

## Estado:

PL	a	b
3		
4	10	
6	10	5
7	10	5
8	10	5
6	5	5
7	5	5
10	5	5
6	5	0
14	5	0
Resultado:		5

# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**

# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**

# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**

# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**

# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**

# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**



# Estruturas de fluxo de controle

- Embora a execução funcione de forma sequencial, muitas vezes temos a necessidade de **ramificar** (do inglês, *branch*) nosso código, agindo de forma diferente para situações diferentes
- Linguagens como C e Python oferecem **estruturas de fluxo de controle** que permitem desviar a execução de um algoritmo
- Ramificações podem acontecer de forma **condicional**, que dependem do valor de uma expressão durante a execução, ou **incondicionais**
  - Um exemplo de estrutura de fluxo condicional é o `if`
  - Um exemplo de estrutura de fluxo incondicional é o `goto`
  - Nós não falamos sobre o `goto`
- Em C, estruturas condicionais são seguidas de **blocos de código**, que são delimitados por **chaves**

# Estruturas de fluxo de controle (if)

- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)

- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)

- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)

- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)

- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)

- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)

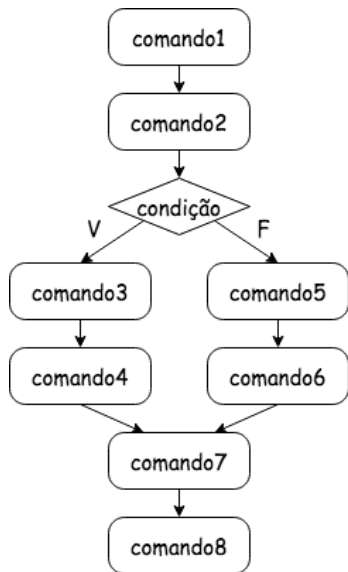
- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)



# Estruturas de fluxo de controle (if)

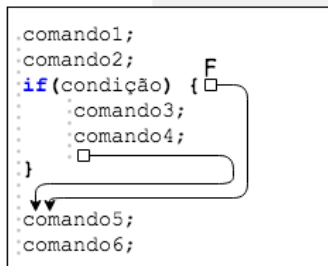
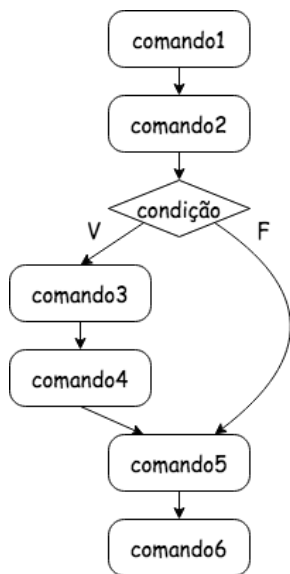
- Uma estrutura de controle **condicional** é representada na linguagem C pela *keyword* `if`
- A **sintaxe necessária** é: *keyword* `if`, abre parênteses, condição, fecha parênteses, bloco 1, *keyword* `else`, bloco 2
  - O segundo bloco, junto ao `else`, pode ser omitido
- Ao encontrar um comando `if`, o programa verificará o valor da expressão entre parênteses
  - Caso ela seja **verdadeira**, a próxima linha de execução será a primeira linha dentro do bloco 1
  - Caso ela seja **falsa**, a próxima linha de execução será a primeira linha dentro do bloco 2
  - Após executar todos os comandos dentro do bloco em questão, a execução segue com a próxima linha **após** a estrutura `if`
  - Caso o `else` (e o bloco 2) estejam ausentes, e a condição seja falsa, a execução segue imediatamente para o comando após a estrutura `if` (como se o bloco 2 estivesse vazio!)

# Estruturas de fluxo de controle (if)



```
comando1;  
comando2;  
if(condição) {  
    comando3;  
    comando4;  
} else {  
    comando5;  
    comando6;  
}  
comando7;  
comando8;
```


# Estruturas de fluxo de controle (if, sem else)



# Estruturas de fluxo de controle (if, sem else)

Há várias formas de se fazer um algoritmo, algumas ilegíveis!

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^(a-z\d){2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



# Estruturas de fluxo de controle (switch)

- Em algumas situações, podemos desejar ramificar o fluxo de um programa de acordo com o valor de uma expressão, para múltiplos casos, como, por exemplo, ao pedir ao usuário que escolha uma opção em um menu
- A linguagem C oferece uma estrutura de switch estruturado para isso (por hora, ignore a versão não-estruturada!)
- A **sintaxe necessária** é: *keyword* switch, abre parênteses, uma expressão, fecha parênteses, e, entre chaves, uma **sequência de opções** com valores a serem comparados
  - Uma opção pode começar com a *keyword* case, uma constante numérica, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula
  - É possível adicionar uma opção para um **caso geral**, para valores não cobertos por opções case, utilizando a *keyword* default, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula

# Estruturas de fluxo de controle (switch)

- Em algumas situações, podemos desejar ramificar o fluxo de um programa de acordo com o valor de uma expressão, para múltiplos casos, como, por exemplo, ao pedir ao usuário que escolha uma opção em um menu
- A linguagem C oferece uma estrutura de switch estruturado para isso (por hora, ignore a versão não-estruturada!)
- A **sintaxe necessária** é: *keyword switch*, abre parênteses, uma expressão, fecha parênteses, e, entre chaves, uma **sequência de opções** com valores a serem comparados
  - Uma opção pode começar com a *keyword case*, uma constante numérica, dois pontos, um bloco, a *keyword break*, e um ponto e vírgula
  - É possível adicionar uma opção para um **caso geral**, para valores não cobertos por opções case, utilizando a *keyword default*, dois pontos, um bloco, a *keyword break*, e um ponto e vírgula

# Estruturas de fluxo de controle (switch)

- Em algumas situações, podemos desejar ramificar o fluxo de um programa de acordo com o valor de uma expressão, para múltiplos casos, como, por exemplo, ao pedir ao usuário que escolha uma opção em um menu
- A linguagem C oferece uma estrutura de switch estruturado para isso (por hora, ignore a versão não-estruturada!)
- A **sintaxe necessária** é: *keyword* switch, abre parênteses, uma expressão, fecha parênteses, e, entre chaves, uma **sequência de opções** com valores a serem comparados
  - Uma opção pode começar com a *keyword* case, uma constante numérica, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula
  - É possível adicionar uma opção para um **caso geral**, para valores não cobertos por opções case, utilizando a *keyword* default, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula

# Estruturas de fluxo de controle (switch)

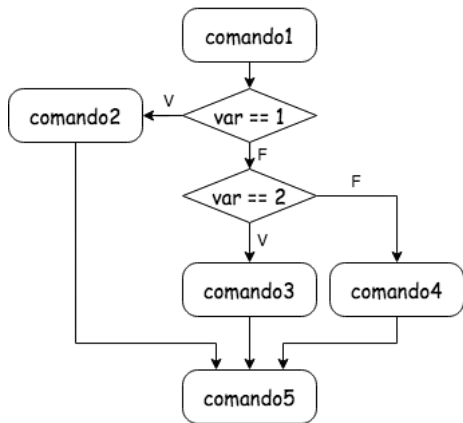
- Em algumas situações, podemos desejar ramificar o fluxo de um programa de acordo com o valor de uma expressão, para múltiplos casos, como, por exemplo, ao pedir ao usuário que escolha uma opção em um menu
- A linguagem C oferece uma estrutura de switch estruturado para isso (por hora, ignore a versão não-estruturada!)
- A **sintaxe necessária** é: *keyword* switch, abre parênteses, uma expressão, fecha parênteses, e, entre chaves, uma **sequência de opções** com valores a serem comparados
  - Uma opção pode começar com a *keyword* case, uma constante numérica, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula
  - É possível adicionar uma opção para um **caso geral**, para valores não cobertos por opções case, utilizando a *keyword* default, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula



# Estruturas de fluxo de controle (switch)

- Em algumas situações, podemos desejar ramificar o fluxo de um programa de acordo com o valor de uma expressão, para múltiplos casos, como, por exemplo, ao pedir ao usuário que escolha uma opção em um menu
- A linguagem C oferece uma estrutura de switch estruturado para isso (por hora, ignore a versão não-estruturada!)
- A **sintaxe necessária** é: *keyword* switch, abre parênteses, uma expressão, fecha parênteses, e, entre chaves, uma **sequência de opções** com valores a serem comparados
  - Uma opção pode começar com a *keyword* case, uma constante numérica, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula
  - É possível adicionar uma opção para um **caso geral**, para valores não cobertos por opções case, utilizando a *keyword* default, dois pontos, um bloco, a *keyword* break, e um ponto e vírgula

# Estruturas de fluxo de controle (switch)



Cuide com a indentação!!

```
comando1;  
switch(var) {  
    case 1: {  
        comando2;  
        break;  
    }  
    case 2: {  
        comando3;  
        break;  
    }  
    default: {  
        comando4;  
        break;  
    }  
}  
comando5;
```

# Estruturas de fluxo de controle (while)

- É comum necessitarmos de **ações que se repetem** de forma determinada, e por isso linguagens imperativas oferecem **estruturas de repetição**
- Uma forma simples de se repetir uma operação é através de uma estrutura `while`: **enquanto** uma condição for verdadeira, repita uma sequência de comandos
- A **sintaxe necessária** é: *keyword* `while`, abre parênteses, condição, fecha parênteses, bloco
- Após a execução do último comando no bloco fornecido para a estrutura `while`, o fluxo **retorna à condição**, e a verifica novamente
  - Se ela for falsa, seguimos para a próxima instrução após o bloco, similar a um `if` sem `else`
  - Se ela for verdadeira novamente, seguimos entrando no bloco fornecido mais uma vez, repetindo isso enquanto for necessário

# Estruturas de fluxo de controle (while)

- É comum necessitarmos de **ações que se repetem** de forma determinada, e por isso linguagens imperativas oferecem **estruturas de repetição**
- Uma forma simples de se repetir uma operação é através de uma estrutura `while`: **enquanto** uma condição for verdadeira, repita uma sequência de comandos
- A **sintaxe necessária** é: *keyword* `while`, abre parênteses, condição, fecha parênteses, bloco
- Após a execução do último comando no bloco fornecido para a estrutura `while`, o fluxo **retorna à condição**, e a verifica novamente
  - Se ela for falsa, seguimos para a próxima instrução após o bloco, similar a um `if` sem `else`
  - Se ela for verdadeira novamente, seguimos entrando no bloco fornecido mais uma vez, repetindo isso enquanto for necessário

# Estruturas de fluxo de controle (while)

- É comum necessitarmos de **ações que se repetem** de forma determinada, e por isso linguagens imperativas oferecem **estruturas de repetição**
- Uma forma simples de se repetir uma operação é através de uma estrutura `while`: **enquanto** uma condição for verdadeira, repita uma sequência de comandos
- A **sintaxe necessária** é: *keyword* `while`, abre parênteses, condição, fecha parênteses, bloco
- Após a execução do último comando no bloco fornecido para a estrutura `while`, o fluxo **retorna à condição**, e a verifica novamente
  - Se ela for falsa, seguimos para a próxima instrução após o bloco, similar a um `if` sem `else`
  - Se ela for verdadeira novamente, seguimos entrando no bloco fornecido mais uma vez, repetindo isso enquanto for necessário

# Estruturas de fluxo de controle (while)

- É comum necessitarmos de **ações que se repetem** de forma determinada, e por isso linguagens imperativas oferecem **estruturas de repetição**
- Uma forma simples de se repetir uma operação é através de uma estrutura `while`: **enquanto** uma condição for verdadeira, repita uma sequência de comandos
- A **sintaxe necessária** é: *keyword* `while`, abre parênteses, condição, fecha parênteses, bloco
- Após a execução do último comando no bloco fornecido para a estrutura `while`, o fluxo **retorna à condição**, e a verifica novamente
  - Se ela for falsa, seguimos para a próxima instrução após o bloco, similar a um `if` sem `else`
  - Se ela for verdadeira novamente, seguimos entrando no bloco fornecido mais uma vez, repetindo isso enquanto for necessário

# Estruturas de fluxo de controle (while)

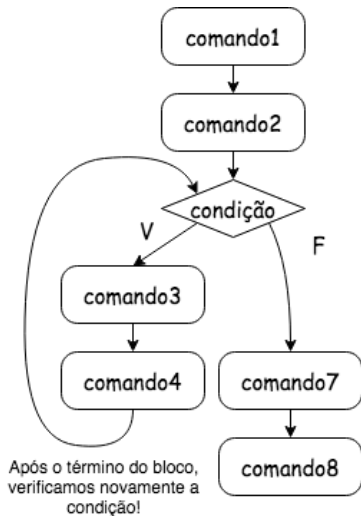
- É comum necessitarmos de **ações que se repetem** de forma determinada, e por isso linguagens imperativas oferecem **estruturas de repetição**
- Uma forma simples de se repetir uma operação é através de uma estrutura `while`: **enquanto** uma condição for verdadeira, repita uma sequência de comandos
- A **sintaxe necessária** é: *keyword* `while`, abre parênteses, condição, fecha parênteses, bloco
- Após a execução do último comando no bloco fornecido para a estrutura `while`, o fluxo **retorna à condição**, e a verifica novamente
  - Se ela for falsa, seguimos para a próxima instrução após o bloco, similar a um `if` sem `else`
  - Se ela for verdadeira novamente, seguimos entrando no bloco fornecido mais uma vez, repetindo isso enquanto for necessário

# Estruturas de fluxo de controle (while)

- É comum necessitarmos de **ações que se repetem** de forma determinada, e por isso linguagens imperativas oferecem **estruturas de repetição**
- Uma forma simples de se repetir uma operação é através de uma estrutura `while`: **enquanto** uma condição for verdadeira, repita uma sequência de comandos
- A **sintaxe necessária** é: *keyword* `while`, abre parênteses, condição, fecha parênteses, bloco
- Após a execução do último comando no bloco fornecido para a estrutura `while`, o fluxo **retorna à condição**, e a verifica novamente
  - Se ela for falsa, seguimos para a próxima instrução após o bloco, similar a um `if` sem `else`
  - Se ela for verdadeira novamente, seguimos entrando no bloco fornecido mais uma vez, repetindo isso enquanto for necessário



# Estruturas de fluxo de controle (while)



```
comando1;  
comando2;  
while(condição) {  
    comando3;  
    comando4;  
}  
comando7;  
comando8;
```

- Muitas vezes, ao desenvolver um algoritmo, temos a necessidade de repetir uma sequência de comandos um número  $n$  de vezes
- Por possuímos a capacidade de alterar a memória, é comum usarmos o conceito de um **contador**, uma variável usada para contar quantas vezes um *loop* foi realizado
- Por exemplo, podemos iniciar uma variável  $i$  de tipo inteiro com valor zero, e, repetir a execução de um bloco enquanto a variável for menor que um valor  $n$ , que representa o número de vezes que queremos executar
  - Ao fim de cada bloco, **aumentamos** o valor do contador
- Então assim, por exemplo, tendo  $n = 5$ , poderemos repetir o bloco cinco vezes, com a variável  $i$  valendo 0, 1, 2, 3 e 4, respectivamente, em cada **iteração**

- Muitas vezes, ao desenvolver um algoritmo, temos a necessidade de repetir uma sequência de comandos um número  $n$  de vezes
- Por possuímos a capacidade de alterar a memória, é comum usarmos o conceito de um **contador**, uma variável usada para contar quantas vezes um *loop* foi realizado
- Por exemplo, podemos iniciar uma variável  $i$  de tipo inteiro com valor zero, e, repetir a execução de um bloco enquanto a variável for menor que um valor  $n$ , que representa o número de vezes que queremos executar
  - Ao fim de cada bloco, **aumentamos** o valor do contador
- Então assim, por exemplo, tendo  $n = 5$ , poderemos repetir o bloco cinco vezes, com a variável  $i$  valendo 0, 1, 2, 3 e 4, respectivamente, em cada **iteração**

- Muitas vezes, ao desenvolver um algoritmo, temos a necessidade de repetir uma sequência de comandos um número  $n$  de vezes
- Por possuímos a capacidade de alterar a memória, é comum usarmos o conceito de um **contador**, uma variável usada para contar quantas vezes um *loop* foi realizado
- Por exemplo, podemos iniciar uma variável  $i$  de tipo inteiro com valor zero, e, repetir a execução de um bloco enquanto a variável for menor que um valor  $n$ , que representa o número de vezes que queremos executar
  - Ao fim de cada bloco, **aumentamos** o valor do contador
- Então assim, por exemplo, tendo  $n = 5$ , poderemos repetir o bloco cinco vezes, com a variável  $i$  valendo 0, 1, 2, 3 e 4, respectivamente, em cada **iteração**

- Muitas vezes, ao desenvolver um algoritmo, temos a necessidade de repetir uma sequência de comandos um número  $n$  de vezes
- Por possuímos a capacidade de alterar a memória, é comum usarmos o conceito de um **contador**, uma variável usada para contar quantas vezes um *loop* foi realizado
- Por exemplo, podemos iniciar uma variável  $i$  de tipo inteiro com valor zero, e, repetir a execução de um bloco enquanto a variável for menor que um valor  $n$ , que representa o número de vezes que queremos executar
  - Ao fim de cada bloco, **aumentamos** o valor do contador
- Então assim, por exemplo, tendo  $n = 5$ , poderemos repetir o bloco cinco vezes, com a variável  $i$  valendo 0, 1, 2, 3 e 4, respectivamente, em cada **iteração**

- Muitas vezes, ao desenvolver um algoritmo, temos a necessidade de repetir uma sequência de comandos um número  $n$  de vezes
- Por possuímos a capacidade de alterar a memória, é comum usarmos o conceito de um **contador**, uma variável usada para contar quantas vezes um *loop* foi realizado
- Por exemplo, podemos iniciar uma variável  $i$  de tipo inteiro com valor zero, e, repetir a execução de um bloco enquanto a variável for menor que um valor  $n$ , que representa o número de vezes que queremos executar
  - Ao fim de cada bloco, **aumentamos** o valor do contador
- Então assim, por exemplo, tendo  $n = 5$ , poderemos repetir o bloco cinco vezes, com a variável  $i$  valendo 0, 1, 2, 3 e 4, respectivamente, em cada **iteração**

# Estruturas de fluxo de controle (for)

- Podemos exemplificar a ideia de contadores com o seguinte código:

```
1 // (1) Declaramos um contador
2 int i = 0;
3 // (2) Enquanto ele for menor que um limite...
4 while(i < n) {
5     // (3) Executamos alguns comandos
6     comandos;
7     // (4) E aumentamos o contador
8     i++;
9 }
```

- Pelo fato desse padrão ser comum, a linguagem C oferece uma forma alternativa, equivalente ao código acima:

```
1 for(int i = 0; i < n; i++) {
2     comandos;
3 }
```

# Estruturas de fluxo de controle (for)

- Podemos exemplificar a ideia de contadores com o seguinte código:

```
1 // (1) Declaramos um contador
2 int i = 0;
3 // (2) Enquanto ele for menor que um limite...
4 while(i < n) {
5     // (3) Executamos alguns comandos
6     comandos;
7     // (4) E aumentamos o contador
8     i++;
9 }
```

- Pelo fato desse padrão ser comum, a linguagem C oferece uma forma alternativa, equivalente ao código acima:

```
1 for(int i = 0; i < n; i++) {
2     comandos;
3 }
```



# Estruturas de fluxo de controle (for)

```
for(int i=0;i<5;i++)
```

