10 Comando de Repetição for

Ronaldo F. Hashimoto e Carlos H. Morimoto

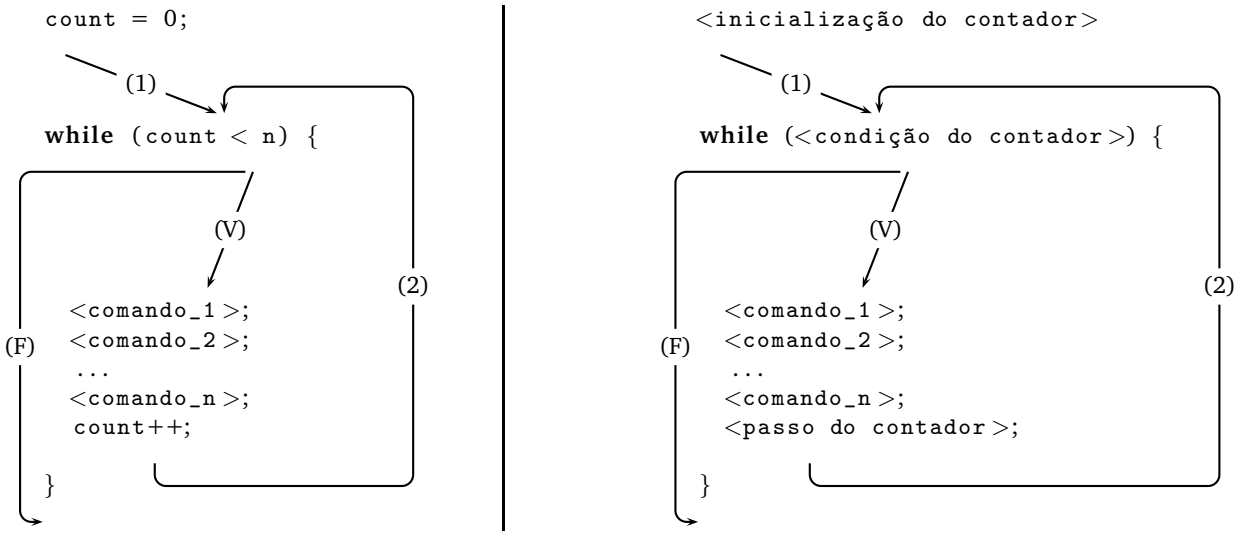
Essa aula introduz o comando de repetição `for`, que pode ser entendida como uma notação compacta do comando `while`.

Ao final dessa aula você deverá saber:

- Descrever a sintaxe do comando `for`.
- Simular o funcionamento do comando `for`.
- Utilizar o comando `for` em programas.

10.1 Motivação

Existem várias situações em que a repetição que resolve um exercício de programação envolve o uso de um contador que começa com um valor inicial (`<inicialização>`) e aumenta/diminui de um em um (ou de dois em dois, a cada passo). Observe atentamente os seguintes trechos de programa:



No trecho do lado esquerdo, a variável `count` controla o número de vezes em que os comandos dentro da repetição são executados. Como `count` é inicializado com zero, aumenta de um em um até chegar em `n`, então os comandos `<comando_1>`, `<comando_2>`, ..., `<comando_n>` são executados `n` vezes.

No trecho do lado direito, nós temos um padrão para este tipo de repetição controlada por um contador. O contador deve ser inicializado antes da repetição; dentro do laço, o contador deve ser incrementado (ou decrementado) de um valor; e a condição do contador deve garantir a parada da repetição. Vamos chamar este padrão de programação de **repetição com contador**. Este padrão de programação pode ser escrito usando o comando `for`.

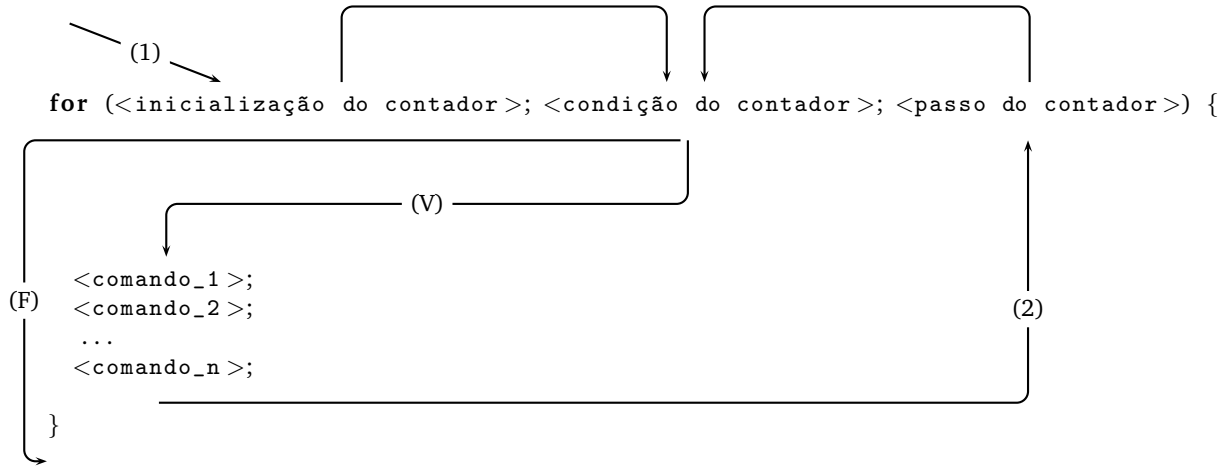
10.2 Sintaxe

O comando `for` é um outro comando de repetição para o padrão de repetição com contador. A sua sintaxe é:

```
for (<inicialização do contador>; <condição do contador>; <passo do contador>) {
  <comando_1>;
  <comando_2>;
  ...
  <comando_n>;
}
```

10.3 Descrição

O fluxo do comando `for` é dado pela seguinte figura:



Basicamente, este comando de repetição tem o seguinte significado: enquanto a <condição> for **verdadeira**, a sequência de comandos <comando_1>, <comando_2>, ..., <comando_n> é executada.

Vamos analisar o "fluxo" do programa usando o comando de repetição **for**. Primeiramente, a execução do programa vem e faz a inicialização do contador (seta marcada com (1)). Depois a <condição do contador> do **for** é testada. Se "de cara" a <condição do contador> é **falsa**, o fluxo do programa ignora a sequência de comandos dentro do **for** e segue a seta marcada com (F). Agora, se a <condição do contador> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) e executa a sequência de comandos dentro do **for**; executado o último comando (<comando_n>), o fluxo do programa segue a seta marcada com (2) e executa <passo do contador> (aumenta/diminui o contador de passo) e volta a testar a <condição do contador>. Se a <condição do contador> é **verdadeira**, então o fluxo do programa segue a seta marcada com (V) repetindo a sequência de comandos dentro do **for**. Se <condição do contador> é **falsa**, o fluxo do programa ignora a sequência de comandos e segue a seta marcada com (F).

Note que o fluxo de execução do **for** é idêntico ao do **while** do padrão de repetição com contador.

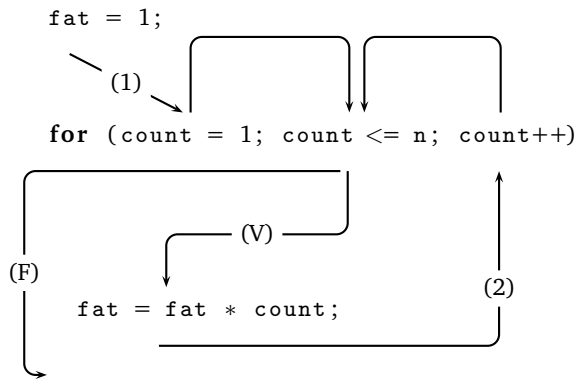
Na verdade, todo comando **for** pode ser escrito usando o comando **while** e vice-versa. A vantagem de usar o comando **for** em repetição com contador é que sua forma é compacta.

10.4 Exemplo

Dado um número inteiro `n>=0`, calcular `n!`

Precisamos gerar a sequência de números $1, 2, \ldots, n$. E acumular a multiplicação para cada número gerado.

Para gerar esta sequência e ao mesmo tempo acumular a multiplicação, vamos usar o comando **for** da seguinte forma:



10.5 Perguntas

1. Este trecho de programa funciona para `n=0`?
2. Por que neste trecho de programa o comando `fat = fat * count;` não está entre chaves?

10.6 Mais um Exercício

Dizemos que um número é triangular se ele é produto de três números naturais consecutivos (e.g.: `120` é triangular pois $120 = 4 \times 5 \times 6$). Dado um natural `n>0`, determinar se `n` é triangular.

Basicamente, precisamos definir 3 coisas:

1. Início: qual o primeiro candidado a solução a ser testado?
2. Fim: a partir de que valor não é mais necessário procurar a solução?
3. Incremento: como gerar a próxima solução a ser testada.

Nesse caso, como os números são positivos, a primeria solução a ser testada seria $1 \times 2 \times 3$. A segunda seria $2 \times 3 \times 4$, e assim por diante. Caso o produto for igual ao número que desejamos testar, então encontramos a solução, e a resposta seria positiva. Quando o produto se torna maior que o número desejado, sabemos que o número não pode ser triangular, e podemos encerrar a busca.

Dessa forma, devemos gerar a sequência $1, 2, 3, \ldots,$ `n` e para cada número `count` desta sequência, calcular o produto `count` $\times$ `(count+1)` $\times$ `(count+2)` e verificar se ele é igual a `n`.

Vejamos um programa que faz isso usando **for**:

```
triangular = FALSE;

for (count=1; count<=n && triangular == FALSE; count++)
  if (count * (count+1) * (count+2) == n)
    triangular = TRUE;

if (triangular == TRUE)
  printf("O numero %d e triangular\n", n);
else
  printf("O numero %d nao eh triangular\n", n);
```

10.7 Perguntas e Comentários

1. `triangular` é um indicador de passagem?
2. O comando **if** dentro do **for** não está entre chaves pois somente tem um comando dentro do **for**.
3. O comando `triangular = TRUE` dentro do **if** não está entre chaves pois somente tem um comando dentro do **if**.
4. Para que serve a condição adicional `triangular == FALSE` dentro do comando **for**?

10.8 Exercícios Recomendados

1. Dado um inteiro `p>1`, verificar se `p` é primo.

   Use indicador de passagem, o comando **for** e interrompa a repetição quando descobrir que `p` não é um número primo.

2. Dados um número inteiro `n>0`, e uma sequência com `n` números inteiros, determinar o maior inteiro da sequência. Por exemplo, para a sequência

   `6, -2, 7, 0, -5, 8, 4`

   o seu programa deve escrever o número `8`.