Nim básico

Original em inglês

Para verificar se a instalação foi bem-sucedida, escreveremos um programa que é tradicionalmente usado como exemplo introdutório: ***Hello World***.

Em um novo arquivo de texto chamado por exemplo `helloworld.nim` precisamos escrever apenas uma linha de código:

```
echo "Hello World!"
```

> A frase que você deseja imprimir deve seguir o comando ***echo*** e deve estar entre aspas duplas (**"**).

Primeiro, precisamos compilar nosso programa e, em seguida, executá-lo para ver se funciona conforme o esperado.

Abra seu terminal no mesmo diretório onde seu arquivo está (no Linux você pode obter 'Abrir Terminal aqui' se clicar com o botão direito do mouse no diretório em seu gerenciador de arquivos, no Windows você deve usar Shift + clique com o botão direito para obter a opção de menu para Abrir a linha de comando).

Compilamos nosso programa digitando no terminal:

```
nim c helloworld.nim
```

Após uma compilação bem-sucedida, podemos executar nosso programa. No Linux, podemos executar nosso programa digitando ***./helloworld*** no terminal, e no Windows, digitando ***helloworld.exe***.

Também existe a possibilidade de compilar e executar o programa com apenas um comando. Precisamos digitar:

```
nim c -r helloworld.nim
```

> ***c*** está dizendo ao Nim para compilar o arquivo e ***-r*** está dizendo para executá-lo imediatamente.
> Para ver todas as opções do compilador, digite ***nim --help*** em seu terminal.

Se você estiver usando o ***VSCode*** com a extensão ***Code Runner*** mencionada antes, você apenas terá que pressionar ***Ctrl + Alt + N*** e seu arquivo será compilado e executado.

Qualquer que seja a forma que você escolheu para executar seu programa, após um breve momento na janela de saída (ou em seu terminal), você verá:

```
Hello World!
```

Parabéns, você executou com sucesso seu primeiro programa Nim!

Agora você sabe como imprimir algumas coisas na tela (usando o comando echo), compilar seu programa (digitando nim c nomedoprograma.nim em seu terminal) e executá-lo (várias possibilidades).

Agora podemos começar a explorar os elementos básicos que nos ajudarão a escrever programas Nim simples.

Naming values (Valores de nomenclatura)

Muitas vezes é útil dar nomes aos valores em nossos programas para nos ajudar a controlar as coisas. Se pedirmos o nome de um usuário, queremos armazená-lo para uso posterior, sem solicitá-lo repetidamente toda vez que precisarmos fazer algum cálculo com ele.

No exemplo ***pi = 3.14***, o nome pi está conectado ao valor ***3.14***. Com base em nossa experiência, podemos dizer que o tipo de variável ** `pi` ** é um número (decimal).

Outro exemplo seria ***firstName = Alice***, onde ***firstName*** é o nome de uma variável com o valor ***Alice***. Diríamos que o tipo desta variável é uma palavra.

Em linguagens de programação, isso funciona de forma semelhante. Essas atribuições de nome têm seu nome, o valor e um tipo.

Declaração de variável

Nim é uma linguagem de programação com tipagem estática, o que significa que o tipo de uma atribuição precisa ser declarado antes de usar o valor.

Em Nim, também distinguimos valores que podem mudar, ou sofrer mutação, daqueles que não podem, mas mais sobre isso mais tarde. Podemos declarar uma variável (uma atribuição mutável) usando a palavra-chave var, apenas declarando seu nome e tipo (o valor pode ser adicionado posteriormente) usando esta sintaxe:

```
var <name>: <type>
```

Se já sabemos seu valor, podemos declarar uma variável e dar a ela um valor imediatamente:

```
var <name>: <type> = <value>
```

> Os colchetes angulares (**<>**) são usados para mostrar algo que você pode alterar.
> Portanto, não é literalmente a palavra ***name*** entre colchetes angulares, mas qualquer nome.

Nim também possui capacidade de inferência de tipo: o compilador pode detectar automaticamente o tipo de atribuição de um nome a partir de seu valor, sem declarar explicitamente o tipo. Veremos mais sobre os vários tipos no próximo capítulo.

Portanto, podemos atribuir uma variável sem um tipo explícito como este:

```
var <name> = <value>
```

Um exemplo disso em Nim é assim:

```
var a: int  # (1)
var b = 7   # (2)
```

1. A variável ***a*** é do tipo ***int*** (inteiro) sem nenhum valor definido explicitamente.
2. A variável ***b*** tem o valor ***7***. Seu tipo é detectado automaticamente como um inteiro.

Ao atribuir nomes, é importante escolher nomes que signifiquem algo para o seu programa. Simplesmente nomeá-los como a, b, c, e assim por diante, rapidamente se tornará confuso. Não é possível usar espaços em um nome, pois isso iria dividi-lo em dois. Portanto, se o nome que você escolher consistir em mais de uma palavra, a maneira usual é escrevê-lo no estilo camelCase (observe que a primeira letra de um nome deve ser minúscula).

Observe, entretanto, que Nim não faz distinção entre maiúsculas e minúsculas e sublinhados, o que significa que ***helloWorld*** e ***hello_world*** teriam o mesmo nome. A exceção a isso é o primeiro caractere, que diferencia maiúsculas de minúsculas. Os nomes também podem incluir números e outros caracteres ***UTF-8***, até mesmo emojis, caso deseje, mas lembre-se de que você e possivelmente outras pessoas terão de digitá-los.

Em vez de digitar ***var*** para cada variável, várias variáveis (não necessariamente do mesmo tipo) podem ser declaradas no mesmo bloco ***var***. No Nim, os blocos são partes do código com o mesmo recuo (mesmo número de espaços antes do primeiro caractere) e o nível de recuo padrão é de dois espaços. Você verá esses blocos em todos os lugares em um programa Nim, não apenas para atribuir nomes.

```
var
  c = -11
  d = "Hello"
  e = '!'
```

> No Nim, as guias de indentação não são permitidas como recuo (tabulação).
> Você pode configurar seu editor de código para converter o pressionamento de Tab em qualquer número de espaços.
> No VS Code, a configuração padrão é converter Tab em quatro espaços. Isso é facilmente substituído nas configurações (Ctrl +,) definindo 'editor.tabSize': 2.

Como as variáveis mencionadas anteriormente são mutáveis, ou seja, seu valor pode mudar (várias vezes), mas seu tipo deve permanecer o mesmo que o declarado.

```
var f = 7 # (1)

f = -3    # (2)
f = 19    # (3)
f = "Hello" # error (4)
```

1. A variável ***f*** tem um valor inicial de ***7*** e seu tipo é inferido como ***int***.
2. O valor de ***f*** é alterado primeiro para ***-3*** e depois para ***19***. Ambos são inteiros, iguais ao valor original.
3. Tentar alterar o valor de ***f*** para ***'Hello'*** produz um erro porque ***Hello*** não é um número e isso mudaria o tipo de ***f*** de um inteiro para uma ***string***.
4. ***# erro*** é um comentário. Os comentários no código Nim são escritos após um caractere **#**. Tudo o que vier depois na mesma linha será ignorado.

Atribuição imutável

Ao contrário das variáveis declaradas com a palavra-chave ***var***, mais dois tipos de atribuição existem em Nim, cujo valor não pode ser alterado, um declarado com a palavra-chave ***const*** e o outro declarado com a palavra-chave ***let***.

Const

O valor de uma atribuição imutável declarada com a palavra-chave ***const*** deve ser conhecido em tempo de compilação (antes que o programa seja executado).

Por exemplo, podemos declarar a aceleração da gravidade como ***const g = 9.81*** ou ***pi*** como ***const pi = 3.14***, pois sabemos seus valores de antemão e esses valores não mudarão durante a execução de nosso programa.

```
const g = 35
g = -27         # error (1)

var h = -5
const i = h + 7 # error (2)
```

1. O valor de uma constante não pode ser alterado.
2. A variável ***h*** não é avaliada em tempo de compilação (é uma variável e seu valor pode mudar durante a execução de um programa), conseqüentemente o valor da constante ***i*** não pode ser conhecido em tempo de compilação, e isso gerará um erro.

Em algumas linguagens de programação, é uma prática comum ter os nomes das constantes escritos em ***ALL_CAPS*** (caixa alta). As constantes em Nim são escritas como qualquer outra variável.

Let

Atribuições imutáveis declaradas com let não precisam ser conhecidas em tempo de compilação, seu valor pode ser definido a qualquer momento durante a execução de um programa, mas uma vez definido, seu valor não pode ser alterado.

```
let j = 35
j = -27 # error (1)

var k = -5
let l = k + 7 # (2)
```

1. O valor de um imutável não pode ser alterado.
2. Em contraste com o exemplo const acima, isso funciona.

Na prática, você verá ou usará ***let*** com mais freqüência do que ***const***.

Embora você possa usar ***var*** para tudo, sua escolha padrão deve ser ***let***. Use ***var*** apenas para as variáveis que serão modificadas.

Tipos de dados básicos

Inteiros (integers)

Como visto no capítulo anterior, inteiros são números que são escritos sem um componente fracionário e sem um ponto decimal.

Por exemplo: ***32, -174, 0, 10_000_000*** são todos inteiros. Observe que podemos usar _ como um separador de milhares, para tornar os números maiores mais legíveis (é mais fácil ver que estamos falando de ***10 milhões*** quando é escrito como ***10_000_000*** em vez de ***10000000***).

Os operadores matemáticos usuais - ***adição (+), subtração (-), multiplicação (*) e divisão (/)*** - funcionam como esperado. As três primeiras operações sempre produzem inteiros, enquanto a divisão de dois inteiros sempre dá um número de ponto flutuante (um número com um ponto decimal) como resultado, mesmo se dois números puderem ser divididos sem resto.

A divisão inteira (divisão em que a parte fracionária é descartada) pode ser obtida com o operador ***div***. Um operador ***mod*** é usado se alguém estiver interessado no resto (módulo) de uma divisão inteira. O resultado dessas duas operações é sempre um número inteiro.

integers.nim

```
let
  a = 11
  b = 4

echo "a + b = ", a + b
echo "a - b = ", a - b
echo "a * b = ", a * b
echo "a / b = ", a / b
echo "a div b = ", a div b
echo "a mod b = ", a mod b
```

O comando ***echo*** imprimirá na tela tudo o que o segue separado por vírgulas. Nesse caso do primeiro ***echo***, primeiro imprime a ***string a + b =*** e, a seguir, na mesma linha, imprime o resultado da expressão ***a + b***.

Podemos compilar e executar o código acima, e a saída deve ser:

```
a + b = 15
a - b = 7
a * b = 44
a / b = 2.75
a div b = 2
a mod b = 3
```

Números em ponto fluante (Floats)

Os números de vírgula flutuante, ou simplesmente flutuantes, são uma representação aproximada de números reais.

Por exemplo: ***2.73, -3.14, 5.0, 4e7*** são flutuantes. Observe que podemos usar notação científica para grandes flutuadores, onde o número após o e é o expoente. Neste exemplo, ***4e7*** é uma notação que representa ***4 * 10 ^ 7***.

Também podemos usar as quatro operações matemáticas básicas entre dois flutuadores. Operadores ***div*** e ***mod*** não são definidos para flutuadores.

floats.nim

```
let
  c = 6.75
  d = 2.25

echo "c + d = ", c + d
echo "c - d = ", c - d
echo "c * d = ", c * d
echo "c / d = ", c / d
```

```
c + d = 9.0
c - d = 4.5
c * d = 15.1875
c / d = 3.0
```

Observe que nos exemplos de ***adição*** e ***divisão***, embora obtenhamos um número sem uma parte decimal, o resultado ainda é do tipo flutuante.

A precedência das operações matemáticas é como seria de esperar: ***multiplicação*** e ***divisão*** têm prioridade mais alta do que ***adição*** e ***subtração***.

```
echo 2 + 3 * 4
echo 24 - 8 / 4
```

```
14
22.0
```

Convertendo floats e inteiros

Operações matemáticas entre variáveis de diferentes tipos numéricos não são possíveis no Nim e irão produzir um erro:

```
let
  e = 5
  f = 23.456

echo e + f   # error
```

Os valores das variáveis precisam ser convertidos para o mesmo tipo. A conversão é direta: para converter para um inteiro, usamos a função ***int***, e para converter para um ***float***, a função ***float*** é usada.

```
let
  e = 5
  f = 23.987

echo float(e) # (1)
echo int(f)   # (2)

echo float(e) + f # (3)
echo e + int(f)   # (4)
```

1. Imprimir uma versão flutuante de um inteiro e. (e permanece do tipo inteiro)
2. Imprimindo uma versão inteira de um float f.
3. Ambos os operandos são flutuantes e podem ser adicionados.
4. Ambos os operandos são inteiros e podem ser adicionados.

```
5.0
23
28.987
28
```

> Ao usar a função ***int*** para converter um ***float*** em um ***inteiro***, nenhum arredondamento será executado. O número simplesmente elimina quaisquer casas decimais.
> Para realizar o arredondamento devemos chamar outra função, mas para isso devemos saber um pouco mais sobre como usar o Nim.

Characters

O tipo ***char*** é usado para representar um único caractere ***ASCII***.

Os caracteres são escritos entre duas aspas simples (**'**). Os caracteres podem ser letras, símbolos ou dígitos únicos. Vários dígitos ou várias letras produzem um erro.

```
let
  h = 'z'
  i = '+'
  j = '2'
  k = '35' # error
  l = 'xy' # error
```

Strings

Strings podem ser descritos como uma série de caracteres. Seu conteúdo é escrito entre duas aspas duplas (**"**).

Podemos pensar em ***strings*** como palavras, mas elas podem conter mais de uma palavra, alguns símbolos ou dígitos.

```
let
  m = "word"
  n = "A sentence with interpunction."
  o = ""    # (1)
  p = "32"  # (2)
  q = "!"   # (3)
```

1. Uma ***string*** vazia.
2. Este não é um número (***int***). Ele está entre aspas duplas, o que o torna uma ***string***.
3. Embora seja apenas um caractere, não é um caractere porque está entre aspas duplas.

Caracteres especiais

Se tentarmos imprimir a seguinte string:

```
echo "some\nim\tips"
```

O resultado pode nos surpreender:

```
some
im  ips
```

Isso ocorre porque existem vários caracteres que têm um significado especial. Eles são usados colocando o caractere de escape **** antes deles.

- **\ n** é um caractere de nova linha
- **\ t** é um caractere de tabulação
- **** é uma barra invertida (uma vez que **** é usado como o caractere de escape)

Se quisermos imprimir o exemplo acima como foi escrito, temos duas possibilidades:

- Use **** em vez de **** para imprimir barras invertidas ou
- Use ***strings*** brutas com sintaxe ***r '...'*** (colocando uma letra ***r*** imediatamente antes da primeira aspa), nas quais não haja caracteres de escape e nenhum significado especial: tudo é impresso como está.

```
echo "some\\nim\\tips"
echo r"some\nim\tips"
```

```
some\nim\tips
some\nim\tips
```

Existem mais caracteres especiais do que os listados acima, e todos eles são encontrados no manual do Nim.

Concatenação de string

Strings em ***Nim*** são mutáveis, o que significa que seu conteúdo pode mudar. Com a função ***add***, podemos adicionar (anexar) outra ***string*** ou um caractere a uma ***string*** existente. Se não quisermos alterar a ***string*** original, também podemos ***concatenar*** (juntar) ***strings*** com o operador ***&***, isso retorna uma nova ***string***.

stringConcat.nim

```
var # (1)
  p = "abc"
  q = "xy"
```

```
  r = 'z'

p.add("def") # (2)
echo "p is now: ", p

q.add(r) # (3)
echo "q is now: ", q

echo "concat: ", p & q  # (4)

echo "p is still: ", p
echo "q is still: ", q
```

1. Se planejamos modificar ***strings***, eles devem ser declarados como ***var***.
2. Adicionar outra ***string*** modifica a ***string*** existente ***p*** no local, alterando seu valor.
3. Também podemos adicionar um ***char*** a uma ***string***.
4. Concatenar duas ***strings*** produz uma nova ***string***, sem modificar as ***strings*** originais.

```
p is now: abcdef
q is now: xyz
concat: abcdefxyz
p is still: abcdef
q is still: xyz
```

Boleano

Um tipo de dado ***boolean*** (ou apenas ***bool***) pode ter apenas dois valores: ***true*** ou ***false***. Os booleanos são geralmente usados para controlar o fluxo (consulte o próximo capítulo) e geralmente são o resultado de operadores relacionais.

A convenção de nomenclatura usual para variáveis booleanas é escrevê-las como perguntas simples sim/não (verdadeiro/falso), por exemplo, ***IsEmpty***, ***isFinished***, ***isMoving***, etc.

Operadores relacionais

Os operadores relacionais testam a relação entre duas entidades, que devem ser comparáveis.

Para comparar se dois valores são iguais, **==** (dois sinais de igual) é usado. Não confunda com **=**, que é usado para atribuição como vimos anteriormente.

Aqui estão todos os operadores relacionais definidos para inteiros:

relationalOperators.nim

```
let
  g = 31
  h = 99

echo "g is greater than h: ", g > h
echo "g is smaller than h: ", g < h
echo "g is equal to h: ", g == h
echo "g is not equal to h: ", g != h
echo "g is greater or equal to h: ", g >= h
echo "g is smaller or equal to h: ", g <= h
```

```
g is greater than h: false
g is smaller than h: true
g is equal to h: false
g is not equal to h: true
g is greater or equal to h: false
g is smaller or equal to h: true
```

Também podemos comparar ***caracteres*** e ***strings***:

relationalOperators.nim

```
let
  i = 'a'
  j = 'd'
  k = 'Z'

echo i < j
echo i < k # (1)
```

```
let
  m = "axyb"
  n = "axyz"
  o = "ba"
  p = "ba "

echo m < n # (2)
echo n < o # (3)
echo o < p # (4)
```

1. Todas as letras maiúsculas vêm antes das letras minúsculas.
2. A comparação de strings funciona caractere por caractere. Os três primeiros caracteres são iguais e o caractere ***b*** é menor que o caractere ***z***.
3. O comprimento da ***string*** não importa para comparação se seus caracteres não forem idênticos.
4. A ***string*** mais curta é menor do que a mais longa.

```
true
false
true
true
true
```

Operadores lógicos

Operadores lógicos são usados para testar a veracidade de uma expressão que consiste em um ou mais valores booleanos.

- Lógico ***and*** retorna verdadeiro apenas se ambos os membros forem verdadeiros
- Lógico ***or*** retorna verdadeiro se houver pelo menos um membro verdadeiro
- O ***xor*** lógico retorna verdadeiro se um membro for verdadeiro, mas o outro não
- O lógico ***not*** nega a veracidade de seu membro: mudando verdadeiro para falso e vice-versa (é o único operador lógico que leva apenas um operando)

logicalOperators.nim

```
echo "T and T: ", true and true
echo "T and F: ", true and false
echo "F and F: ", false and false
echo "---"
echo "T or T: ", true or true
echo "T or F: ", true or false
echo "F or F: ", false or false
echo "---"
echo "T xor T: ", true xor true
echo "T xor F: ", true xor false
echo "F xor F: ", false xor false
echo "---"
echo "not T: ", not true
echo "not F: ", not false
```

Operadores relacionais e lógicos podem ser combinados para formar expressões mais complexas.

Por exemplo: ***(5 < 7) and (11 + 9 == 32 - 2 * 6)*** se tornará verdadeiro e ***(20 == 20)***, que se tornará verdadeiro e verdadeiro, e no final dará o resultado final verdadeiro.

Recapitular

Este foi o capítulo mais longo deste tutorial e cobrimos muito terreno. Reserve um tempo para examinar cada tipo de dados e experimente o que você pode fazer com cada um deles.

Os tipos podem parecer uma restrição no início, mas permitem que o compilador Nim torne seu código mais rápido e certifique-se de que você não esteja fazendo algo errado por acidente - isso é especialmente benéfico em grandes bases de código.

Agora você conhece os tipos de dados básicos e várias operações sobre eles, o que deve ser suficiente para fazer alguns cálculos simples no Nim. Teste seus conhecimentos fazendo os seguintes exercícios.

Exercícios

1. Crie uma variável imutável contendo sua idade (em anos). Imprima sua idade em dias. (1 ano = 365 dias)
2. Verifique se sua idade é divisível por 3. (Dica: use o ***mod***)
3. Crie uma variável imutável contendo sua altura em centímetros. Imprima sua altura em polegadas. (1 pol = 2,54 cm)
4. Um tubo tem 3/8 de polegada de diâmetro. Expresse o diâmetro em centímetros.
5. Crie uma variável imutável contendo seu primeiro nome e outra contendo seu sobrenome. Faça uma variável fullName concatenando as duas variáveis anteriores. Não se esqueça de colocar um espaço em branco no meio. Imprima seu nome completo.
6. Alice ganha R$400 a cada 15 dias. Bob ganha R$3,14 por hora e trabalha 8 horas por dia, 7 dias por semana. Depois de 30 dias, Alice ganhou mais do que Bob? (Dica: use operadores relacionais)

Controle de fluxo

Até agora, em nossos programas, cada linha de código foi executada em algum ponto. As declarações de fluxo de controle nos permitem ter partes de código que serão executadas apenas se alguma condição booleana for satisfeita.

Se pensarmos em nosso programa como uma estrada, podemos pensar no fluxo de controle como vários ramos, e escolhemos nosso caminho dependendo de alguma condição. Por exemplo, compraremos ovos apenas se seu preço for inferior a algum valor. Ou, se estiver chovendo, levaremos um guarda-chuva, caso contrário (***else***) levaremos óculos escuros.

Escritos em pseudocódigo, esses dois exemplos seriam assim:

```
if precoOvos < precoEsperado:
  compreOvos

if estaChovendo:
  traga quarda-chuva
else:
  traga óculos escuros
```

A sintaxe do Nim é muito semelhante, como você verá a seguir.

Declaração If

Uma instrução ***if***, conforme mostrado acima, é a maneira mais simples de ramificar nosso programa.

A sintaxe Nim para escrever a instrução ***if*** é:

```
if <condition>: # (1)
  <indented block> # (2)
```

1. A condição deve ser do tipo booleano: uma variável booleana ou uma expressão relacional ***and*** / ***or*** lógico.
2. Todas as linhas após a linha ***if*** que são indentadas com dois espaços formam o mesmo bloco e serão executadas apenas se a condição for verdadeira.

As instruções ***if*** podem ser aninhadas, ou seja, dentro de um bloco ***if***, pode haver outra instrução ***if***.

if.nim

```
let
  a = 11
  b = 22
  c = 999

if a < b:
  echo "a is smaller than b"
  if 10*a < b: # (1)
    echo "not only that, a is *much* smaller than b"
```

```
if b < c:
  echo "b is smaller than c"
  if 10*b < c: # (2)
    echo "not only that, b is *much* smaller than c"

if a+b > c: # (3)
  echo "a and b are larger than c"
  if 1 < 100 and 321 > 123: # (4)
    echo "did you know that 1 is smaller than 100?"
    echo "and 321 is larger than 123! wow!"
```

1. A primeira condição é verdadeira, a segunda é falsa - o ***echo*** interno não é executado.
2. Ambas as condições são verdadeiras e ambas as linhas são impressas.
3. A primeira condição é falsa - todas as linhas dentro de seu bloco serão ignoradas, nada é impresso.
4. Usando a lógica ***and*** dentro da instrução ***if***.

```
a is smaller than b
b is smaller than c
not only that, b is *much* smaller than c
```

Else

Else segue após um bloco ***if*** e nos permite ter uma ramificação do código que será executada quando a condição na instrução ***if*** não for verdadeira.

else.nim

```
let
  d = 63
  e = 2.718

if d < 10:
  echo "d is a small number"
else:
  echo "d is a large number"

if e < 10:
  echo "e is a small number"
else:
  echo "e is a large number"
```

```
d is a large number
e is a small number
```

> Se você deseja executar um bloco apenas se a declaração for falsa, você pode simplesmente negar a condição com o operador ***not***.

Elif

Elif é a abreviatura de ***'else if'*** e nos permite encadear várias instruções ***if***.

O programa testa cada afirmação até encontrar uma que seja verdadeira. Depois disso, todas as outras instruções são ignoradas.

elif.nim

```
let
  f = 3456
  g = 7

if f < 10:
  echo "f is smaller than 10"
elif f < 100:
  echo "f is between 10 and 100"
elif f < 1000:
  echo "f is between 100 and 1000"
```

```
else:
  echo "f is larger than 1000"

if g < 1000:
  echo "g is smaller than 1000"
elif g < 100:
  echo "g is smaller than 100"
elif g < 10:
  echo "g is smaller than 10"
```

```
f is larger than 1000
g is smaller than 1000
```

> No caso de ***g***, embora ***g*** satisfaça todas as três condições, apenas o primeiro ramo é executado, pulando automaticamente todos os outros ramos.

Case

Uma instrução ***case*** é outra maneira de escolher apenas um dos vários caminhos possíveis, semelhante à instrução ***if*** com vários ***elifs***. Uma declaração de caso, no entanto, não leva várias condições booleanas, mas sim qualquer valor com estados distintos e um caminho para cada valor possível.

Código escrito em bloco ***if-elif*** parecido com este:

```
if x == 5:
  echo "Five!"
elif x == 7:
  echo "Seven!"
elif x == 10:
  echo "Ten!"
else:
  echo "unknown number"
```

Pode ser escrito com uma instrução ***case*** como esta:

```
case x
of 5:
  echo "Five!"
of 7:
  echo "Seven!"
of 10:
  echo "Ten!"
else:
  echo "unknown number"
```

Ao contrário da instrução ***if***, a instrução case deve cobrir todos os casos possíveis. Se alguém não estiver interessado em algum desses casos, então: descarte pode ser usado.

case.nim

```
let h = 'y'

case h
of 'x':
  echo "You've chosen x"
of 'y':
  echo "You've chosen y"
of 'z':
  echo "You've chosen z"
else: discard # (1)
```

1. Mesmo que estejamos interessados em apenas três valores de ***h***, devemos incluir esta linha para cobrir todos os outros casos possíveis (todos os outros caracteres). Sem ele, o código não seria compilado.

```
You've chosen y
```

Também podemos usar vários valores para cada ramificação se a mesma ação ocorrer para mais de um valor.

multipleCase.nim

```
let i = 7

case i
  of 0:
    echo "i is zero"
  of 1, 3, 5, 7, 9:
    echo "i is odd"
  of 2, 4, 6, 8:
    echo "i is even"
  else:
    echo "i is too large"
```

```
i is odd
```

Loops

Os ***loops*** são outra construção de fluxo de controle que nos permite executar algumas partes do código várias vezes.

Neste capítulo, conheceremos dois tipos de ***loops***:

- ***loop for***: executa um número conhecido de vezes
- ***loop while***: execute enquanto alguma condição for satisfeita

Loop for

A sintaxe de um loop for é:

```
for <loopVariable> in <iterable>:
  <loop body>
```

Tradicionalmente, ***i*** é freqüentemente usado como um nome de uma variável de loop, mas qualquer outro nome pode ser usado. Essa variável estará disponível apenas dentro do loop. Assim que o loop terminar, o valor da variável é descartado.

O iterável é qualquer objeto por meio do qual possamos iterar. Dos tipos já mencionados, as ***strings*** são objetos iteráveis. (Mais tipos iteráveis serão introduzidos no próximo capítulo.)

Todas as linhas no corpo do ***loop*** são executadas em cada ***loop***, o que nos permite escrever com eficiência partes repetitivas do código.

Se quisermos iterar por meio de um intervalo de números (inteiros) em Nim, a sintaxe do iterável é ***início .. término***, onde início e término são números. Isso irá iterar por todos os números entre o início e o fim, incluindo o início e o fim. Para o intervalo padrão iterável, o início precisa ser menor do que o final.

Se quisermos iterar até um número (sem incluí-lo), podemos usar **`..<`**:

for1.nim

```
for n in 5 .. 9: # (1)
  echo n

echo ""

for n in 5 ..< 9: # (2)
  echo n
```

1. Iterando através de um intervalo de números usando **`..`** - ambas as extremidades estão incluídas no intervalo.
2. Iterando pelo mesmo intervalo usando **`..<`** - itera até a extremidade superior, sem incluí-lo.

```
5
6
7
8
9

5
6
7
8
```

Se quisermos iterar por meio de um intervalo de números com um tamanho de passo diferente de um, a contagem é usada. Com a contagem, definimos o valor inicial, o valor de parada (incluído no intervalo) e o tamanho do passo.

for2.nim

```
for n in countup(0, 16, 4): # (1)
  echo n
```

1. Contando de zero a 16, com um tamanho de etapa de 4. O final (16) está incluído na faixa.

```
0
4
8
12
16
```

Para iterar por um intervalo de números onde o início é maior do que o final, uma função semelhante chamada contagem regressiva é usada. Mesmo que estejamos em contagem regressiva, o tamanho do passo deve ser positivo.

for2.nim

```
for n in countdown(4, 0): # (1)
  echo n

echo ""

for n in countdown(-3, -9, 2): # (2)
  echo n
```

1. Para iterar de um número superior para um número inferior, devemos usar a contagem regressiva (o operador .. só pode ser usado quando o valor inicial é menor que o valor final).
2. Mesmo durante a contagem regressiva, o tamanho do passo deve ser um número positivo.

```
4
3
2
1
0

-3
-5
-7
-9
```

Como a ***string*** é iterável, podemos usar um loop for para iterar por meio de cada caractere da string (esse tipo de iteração às vezes é chamado de loop ***for-each***).

for3.nim

```
let word = "alphabet"

for letter in word:
  echo letter
```

```
a
l
p
h
a
b
e
t
```

Se também precisarmos ter um contador de iteração (começando do zero), podemos conseguir isso usando a syntax **`for <counterVariable>, <loopVariable> in <iterator>:`**. Isso é muito prático se você deseja iterar por meio de um iterável e, simultaneamente, acessar outro iterável no mesmo deslocamento.

for3.nim

```
for i, letter in word:
  echo "letter ", i, " is: ", letter
```

```
letter 0 is: a
letter 1 is: l
letter 2 is: p
letter 3 is: h
letter 4 is: a
letter 5 is: b
letter 6 is: e
letter 7 is: t
```

Loop while

Os loops ***while*** são semelhantes às instruções ***if***, mas eles continuam executando seu bloco de código enquanto a condição permanecer verdadeira. Eles são usados quando não sabemos com antecedência quantas vezes o loop será executado.

Devemos ter certeza de que o loop terminará em algum ponto e não se tornará um loop infinito.

while.nim

```
var a = 1

while a*a < 10: # (1)
  echo "a is: ", a
  inc a # (2)

echo "final value of a: ", a
```

1. Esta condição será verificada todas as vezes antes de entrar no novo loop e executar o código dentro dele.
2. ***Inc*** é usado para incrementar a por um. É o mesmo que escrever ***a = a + 1*** ou ***a += 1***.

```
a is: 1
a is: 2
a is: 3
final value of a: 4
```

Break e continue

A instrução break é usada para sair prematuramente de um loop, geralmente se alguma condição for atendida.

No próximo exemplo, se não houvesse uma instrução ***if*** com ***break***, o loop continuaria a ser executado e impresso até que ***i*** se tornasse 1000. Com a instrução ***break***, quando ***i*** chegar a 3, saímos imediatamente do loop (antes de imprimir o valor de ***i***).

break.nim

```
var i = 1

while i < 1000:
  if i == 3:
    break
  echo i
  inc i
```

```
1
2
```

A instrução continue inicia a próxima iteração de um loop imediatamente, sem executar as linhas restantes da iteração atual. Observe como 3 e 6 estão faltando na saída do seguinte código:

continue.nim

```
for i in 1 .. 8:
  if (i == 3) or (i == 6):
    continue
  echo i
```

```
1
2
4
5
7
8
```

Exercícios

1. A conjectura de Collatz é um problema matemático popular com regras simples. Primeiro escolha um número. Se for ímpar, multiplique por três e some um; Se for par, divida por dois. Repita este procedimento até chegar a um. Por exemplo. 5 → ímpar → 3 * 5 + 1 = 16 → par → 16/2 = 8 → par → 4 → 2 → 1 → fim!
   Escolha um inteiro (como uma variável mutável) e crie um loop que imprimirá todas as etapas da conjectura de Collatz. (Dica: use ***div*** para divisão)
2. Crie uma variável imutável contendo seu nome completo. Escreva um laço for que irá iterar através daquela string e imprimir apenas as vogais (a, e, i, o, u). (Dica: use a instrução ***case*** com vários valores por ramo)
3. Fizz buzz é um jogo infantil às vezes usado para testar conhecimentos básicos de programação. Contamos os números de um para cima. Se um número for divisível por 3, substitua-o por fizz, se for divisível por 5 substitua-o por buzz e se um número for divisível por 15 (ambos 3 e 5) substitua-o por fizzbuzz. As primeiras rodadas ficariam assim: 1, 2, fizz, 4, buzz, fizz, 7, ...
   Crie um programa que irá imprimir as primeiras 30 rodadas de Fizz Buzz. (Dica: cuidado com a ordem dos testes de divisibilidade)
4. Nos exercícios anteriores, você converteu polegadas em centímetros e vice-versa. Crie uma tabela de conversão com vários valores. Por exemplo, a tabela pode ser assim:

```
in  | cm
---------------
1   | 2.54
4   | 10.16
7   | 17.78
10  | 25.4
13  | 33.02
16  | 40.64
19  | 48.26
```

Containers

Os contêineres são tipos de dados que contêm uma coleção de itens e nos permitem acessar esses elementos. Normalmente, um contêiner também é iterável, o que significa que podemos usá-los da mesma forma que usamos strings no capítulo de loops.

Por exemplo, uma lista de compras é um contêiner de itens que queremos comprar, e uma lista de números primos é um contêiner de números. Escrito em pseudocódigo:

```
listaCompras = [presunto, ovos, pão, maçãs]
primos = [1, 2, 3, 5, 7]
```

Arrays

Uma matriz é o tipo de contêiner mais simples. As matrizes são homogêneas, ou seja, todos os elementos em uma matriz devem ter o mesmo tipo. Os arrays também têm um tamanho constante, o que significa que a quantidade de elementos (ou melhor: a quantidade de elementos possíveis) deve ser conhecida em tempo de compilação. Isso significa que chamamos arrays de 'contêiner homogêneo de comprimento constante'.

O tipo de array é declarado usando array [comprimento, tipo], onde comprimento é a capacidade total do array (número de elementos que pode caber) e tipo é um tipo de todos os seus elementos. A declaração pode ser omitida se o comprimento e o tipo puderem ser inferidos dos elementos passados.

Os elementos de uma matriz são colocados entre colchetes.

```
var
  a: array[3, int] = [5, 7, 9]
  b = [5, 7, 9] # (1)
  c = []  # error (2)
  d: array[7, string] # (3)
```

1. Se fornecermos os valores, o comprimento e o tipo do array b são conhecidos em tempo de compilação. Embora correto, não há necessidade de declará-lo especificamente como o array a.
2. Nem o comprimento nem o tipo dos elementos podem ser inferidos desse tipo de declaração - isso produz um erro.
3. A maneira correta de declarar um array vazio (que será preenchido posteriormente) é fornecer seu comprimento e tipo, sem fornecer os valores de seus elementos - o array d pode conter sete strings.

Uma vez que o comprimento de uma matriz deve ser conhecido em tempo de compilação, isso não funcionará:

```
const m = 3
let n = 5

var a: array[m, char]
var b: array[n, char] # error (1)
```

1. Isso produz um erro porque n é declarado usando let - seu valor não é conhecido em tempo de compilação. Só podemos usar valores declarados com const como um parâmetro de comprimento para uma inicialização de array.

Sequences

Sequências são contêineres semelhantes a arrays, mas seu comprimento não precisa ser conhecido no tempo de compilação e pode mudar durante o tempo de execução: declaramos apenas o tipo dos elementos contidos com seq [type]. As sequências também são homogêneas, ou seja, cada elemento em uma sequência deve ser do mesmo tipo.

Os elementos de uma sequência são colocados entre **@ [** e **]**.

```
var
  e1: seq[int] = @[] # (1)
  f = @["abc", "def"] # (2)
```

1. O tipo de uma sequência vazia deve ser declarado.
2. O tipo de sequência não vazia pode ser inferido. Nesse caso, é uma sequência contendo strings.

Outra maneira de inicializar uma sequência vazia é chamar o procedimento newSeq. Veremos mais chamadas de procedimento no próximo capítulo, mas por enquanto apenas saiba que esta também é uma possibilidade:

```
var
  e = newSeq[int]() # (1)
```

1. Fornecer o parâmetro de tipo entre colchetes permite que o procedimento saiba que deve retornar uma sequência de um determinado tipo.
   Um erro frequente é a omissão do final (), que deve ser incluído.

Podemos adicionar novos elementos a uma sequência com a função add, semelhante ao que fizemos com strings. Para que isso funcione, a sequência deve ser mutável (definida com var) e o elemento que estamos adicionando deve ser do mesmo tipo que os elementos da sequência.

seq.nim

```
var
  g = @['x', 'y']
  h = @['1', '2', '3']

g.add('z') # (1)
echo g

h.add(g) # (2)
echo h
```

1. Adicionando um novo elemento do mesmo tipo (char).
2. Adicionando outra sequência contendo o mesmo tipo.

```
@['x', 'y', 'z']
@['1', '2', '3', 'x', 'y', 'z']
```

Tentar passar tipos diferentes para as sequências existentes produzirá um erro:

```
var i = @[9, 8, 7]

i.add(9.81) # error (1)
g.add(i)    # error (2)
```

1. Tentando adicionar um ***float*** a uma sequência de ***int***.
2. Tentando adicionar uma sequência de ***int*** a uma sequência de ***char***.

Como as sequências podem variar em comprimento, precisamos encontrar uma maneira de obter seu comprimento, para isso podemos usar a função ***len***.

```
var i = @[9, 8, 7]
echo i.len

i.add(6)
echo i.len
```

```
3
4
```

Indexar e fatiar (Indexing and slicing)

A indexação nos permite obter um elemento específico de um contêiner por meio de seu índice. Pense no índice como uma posição dentro do contêiner.

Nim, como muitas outras linguagens de programação, tem indexação baseada em zero, o que significa que o primeiro elemento em um contêiner tem o índice zero, o segundo elemento tem o índice um, etc.

Se quisermos indexar "para trás", isso é feito usando o prefixo **`^`. O último elemento (primeiro na parte de trás) tem o índice **`^ 1`.

A sintaxe para indexação é **`<container>[<index>]`**.

indexing.nim

```
let j = ['a', 'b', 'c', 'd', 'e']

echo j[1] # (1)
echo j[^1] # (2)
```

1. Indexação baseada em zero: o elemento no índice 1 é b.
2. Obtendo o último elemento.

```
b
e
```

O fatiamento nos permite obter uma série de elementos com uma chamada. Ele usa a mesma sintaxe dos intervalos (introduzidos na seção de ***loop for***).

Se usarmos a sintaxe ***start .. stop***, ambas as extremidades serão incluídas na fatia. Usando a sintaxe ***start ..< stop***, o índice de parada não é incluído na fatia.

A sintaxe para fatiar é **`<container>[<start> .. <stop>]`**.

indexing.nim

```
echo j[0 .. 3]
echo j[0 ..< 3]
```

```
@[a, b, c, d]
@[a, b, c]
```

Tanto a indexação quanto o fracionamento podem ser usados para atribuir novos valores aos contêineres e strings mutáveis existentes.

assign.nim

```
var
```

```
  k: array[5, int]
  l = @['p', 'w', 'r']
  m = "Tom and Jerry"

for i in 0 .. 4: # (1)
  k[i] = 7 * i
echo k

l[1] = 'q' # (2)
echo l

m[8 .. 9] = "Ba" # (3)
echo m
```

1. A matriz de comprimento 5 possui índices de zero a quatro. Iremos atribuir um valor a cada elemento do array.
2. Atribuir (alterar) o segundo elemento (índice 1) de uma sequência.
3. Alterar caracteres de uma string nos índices 8 e 9.

```
[0, 7, 14, 21, 28]
@['p', 'q', 'r']
Tom and Barry
```

Tuplas

Ambos os contêineres que vimos até agora são homogêneos. Por outro lado, as tuplas contêm dados heterogêneos, ou seja, os elementos de uma tupla podem ser de tipos diferentes. Da mesma forma que as matrizes, as tuplas têm tamanho fixo.

Os elementos de uma tupla são colocados entre parênteses.

tuples.nim

```
let n = ("Banana", 2, 'c') # (1)
echo n
```

1. As tuplas podem conter campos de diferentes tipos. Neste caso: ***string***, ***int*** e ***char***.

```
(Field0: "Banana", Field1: 2, Field2: 'c')
```

Também podemos nomear cada campo em uma tupla para distingui-los. Isso pode ser usado para acessar os elementos da tupla, em vez de indexar.

tuples.nim

```
var o = (name: "Banana", weight: 2, rating: 'c')

o[1] = 7 # (1)
o.name = "Apple" # (2)
echo o
```

1. Alterar o valor de um campo usando o índice do campo.
2. Alterar o valor de um campo usando o nome do campo.

```
(name: "Apple", weight: 7, rating: 'c')
```

Exercícios

1. Crie uma matriz vazia que pode conter dez inteiros.
    - Preencha essa matriz com números 10, 20, ..., 100. (Dica: use loops)
    - Imprima apenas os elementos dessa matriz que estão em índices ímpares (valores 20, 40,...).
    - Multiplique os elementos em índices pares por 5. Imprima a matriz modificada.
2. Refaça o exercício de conjectura de Collatz, mas desta vez, em vez de imprimir cada passo, adicione-o a uma sequência.
    - Escolha um número inicial. As escolhas interessantes, entre outras, são 9, 19, 25 e 27.
    - Crie uma sequência cujo único membro seja aquele número inicial
    - Usando a mesma lógica de antes, continue adicionando elementos à sequência até chegar a 1
    - Imprima o comprimento da sequência e a própria sequência
3. Encontre o número em um intervalo de 2 a 100 que produzirá a sequência de Collatz mais longa.

    - Para cada número no intervalo determinado, calcule sua sequência de Collatz
    - Se o comprimento da sequência atual for maior do que o registro anterior, salve o comprimento atual e o número inicial como um novo registro (você pode usar a tupla (longestLength, StartingNumber) ou duas variáveis separadas)
    - Imprime o número inicial que fornece a sequência mais longa e seu comprimento

Procedures

Os procedimentos, ou funções como são chamados em algumas outras linguagens de programação, são partes do código que executam uma tarefa específica, empacotados como uma unidade. A vantagem de agrupar o código assim é que podemos chamar esses procedimentos em vez de escrever todo o código novamente quando quisermos usar o código do procedimento.

Em alguns dos capítulos anteriores, vimos a conjectura de Collatz em vários cenários diferentes. Envolvendo a lógica da conjectura de Collatz em um procedimento, poderíamos ter chamado o mesmo código para todos os exercícios.

Até agora, usamos muitos procedimentos integrados, como ***echo*** para impressão, ***add*** para adicionar elementos a uma sequência, ***inc*** para aumentar o valor de um inteiro, ***len*** para obter o comprimento de um contêiner, etc. Agora veremos Como criar e usar nossos próprios procedimentos.

Algumas das vantagens de usar procedimentos são:

- Reduzindo a duplicação de código
- Mais fácil de ler o código, pois podemos nomear as peças pelo que fazem
- Decompor uma tarefa complexa em etapas mais simples

Conforme mencionado no início desta seção, os procedimentos são freqüentemente chamados de funções em outras linguagens. Na verdade, esse é um nome um tanto impróprio se considerarmos a definição matemática de uma função. As funções matemáticas

recebem um conjunto de argumentos (como f (x, y), onde f é uma função e x e y são seus argumentos) e sempre retornam a mesma resposta para a mesma entrada.

Os procedimentos programáticos, por outro lado, nem sempre retornam a mesma saída para uma determinada entrada. Às vezes, eles não devolvem nada. Isso ocorre porque nossos programas de computador podem armazenar estados nas variáveis mencionadas anteriormente, que os procedimentos podem ler e alterar. No Nim, a palavra ***func*** é atualmente reservada para ser usada como o tipo de função mais matematicamente correto, sem forçar efeitos colaterais.

Declarando um procedure

Antes de podermos usar (chamar) nosso procedimento, precisamos criá-lo e definir o que ele faz.

Um procedimento é declarado usando a palavra-chave ***proc*** e o nome do procedimento, seguido pelos parâmetros de entrada e seu tipo entre parênteses, e a última parte são dois pontos e o tipo do valor retornado de um procedimento, como este:

```
proc <name>(<p1>: <type1>, <p2>: <type2>, ...): <returnType>
```

O corpo de um procedimento é escrito no bloco recuado seguindo a declaração anexada com um sinal **=**.

callProcs.nim

```
proc findMax(x: int, y: int): int = # (1)
  if x > y:
    return x # (2)
  else:
    return y
  # this is inside of the procedure
# this is outside of the procedure
```

1. Procedimento de declaração denominado findMax, que possui dois parâmetros, x e y, e retorna um tipo int.

2. Para retornar um valor de um procedimento, usamos a palavra-chave return.

```
proc echoLanguageRating(language: string) = # (1)
  case language
  of "Nim", "nim", "NIM":
    echo language, " is the best language!"
  else:
    echo language, " might be a second-best language."
```

1. O procedimento ***echoLanguageRating*** apenas ecoa o nome fornecido, não retorna nada, então o tipo de retorno não é declarado.

Normalmente não temos permissão para alterar nenhum dos parâmetros que recebemos. Fazer algo assim gerará um erro:

```
proc changeArgument(argument: int) =
  argument += 5

var ourVariable = 10
changeArgument(ourVariable)
```

Para que isso funcione, precisamos permitir que Nim e o programador usando nosso procedimento alterem o argumento declarando-o como uma variável:

```
proc changeArgument(argument: var int) = # (1)
  argument += 5

var ourVariable = 10
changeArgument(ourVariable)
echo ourVariable
changeArgument(ourVariable)
echo ourVariable
```

1. Observe como o argumento agora é declarado como ***var int*** e não apenas como ***int***.

Isso, é claro, significa que o nome que passamos deve ser declarado também como uma variável, passando algo atribuído com const ou let gerará um erro.

Embora seja uma boa prática passar as coisas como argumentos, também é possível usar nomes declarados fora do procedimento, tanto variáveis quanto constantes:

```
var x = 100

proc echoX() =
  echo x # (1)
  x += 1 # (2)

echoX()
echoX()
```

1. Aqui, acessamos a variável externa ***x***.
2. Também podemos atualizar seu valor, uma vez que é declarado como uma variável.

```
100
101
```

Chamando os procedimentos

Depois de declarar um procedimento, podemos chamá-lo. A maneira usual de chamar procedimentos / funções em muitas linguagens de programação é declarar seu nome e fornecer os argumentos entre parênteses, como este:

```
<procName>(<arg1>, <arg2>, ...)
```

O resultado da chamada de um procedimento pode ser armazenado em uma variável.

Se quisermos chamar nosso procedimento findMax do exemplo acima e salvar o valor de retorno em uma variável, podemos fazer isso com:

callProcs.nim

```
let
  a = findMax(987, 789)
  b = findMax(123, 321)
  c = findMax(a, b) # (1)

echo a
echo b
echo c
```

1. O resultado da função ***findMax*** é aqui denominado ***c***, e é chamado com os resultados de nossas duas primeiras chamadas (***findMax (987, 321)***).

```
987
321
987
```

O Nim, ao contrário de muitas outras linguagens, também oferece suporte à Sintaxe de Chamada de Função Uniforme, que permite muitas maneiras diferentes de chamar procedimentos.

Esta é uma chamada em que o primeiro argumento é escrito antes do nome da função e o resto dos parâmetros são declarados entre parênteses:

```
<arg1>.<procName>(<arg2>, ...)
```

Usamos essa sintaxe quando adicionamos elementos a uma sequência existente (**`<seq>.add(<element>)`**), **pois isso a torna mais legível e expressa nossa intenção de maneira mais clara do que escrever** **`add(<seq>, <element>)`**. Também podemos omitir os parênteses em torno dos argumentos:

```
<procName> <arg1>, <arg2>, ...
```

Vimos esse estilo sendo usado quando chamamos o procedimento ***echo*** e ao chamar o procedimento ***len*** sem nenhum argumento. Esses dois também podem ser combinados dessa forma, mas essa sintaxe, entretanto, não é vista com muita frequência:

```
<arg1>.<procName> <arg2>, <arg3>, ...
```

A sintaxe de chamada uniforme permite um encadeamento mais legível de vários procedimentos:

ufcs.nim

```
proc plus(x, y: int): int = # (1)
  return x + y

proc multi(x, y: int): int =
  return x * y

let
  a = 2
  b = 3
  c = 4

echo a.plus(b) == plus(a, b)
echo c.multi(a) == multi(c, a)


echo a.plus(b).multi(c) # (2)
echo c.multi(b).plus(a) # (3)
```

1. Se vários parâmetros forem do mesmo tipo, podemos declarar seu tipo desta forma compacta.
2. Primeiro adicionamos ***a*** e ***b***, então o resultado dessa operação (***2 + 3 = 5***) é passado como o primeiro parâmetro para o procedimento ***multi***, onde é multiplicado por c (***5 * 4 = 20***).
3. Primeiro, multiplicamos ***c*** e ***b***, então o resultado dessa operação (***4 * 3 = 12***) é passado como o primeiro parâmetro para o procedimento ***plus***, onde é adicionado com ***a*** (***12 + 2 = 14***).

```
true
true
20
14
```

Variável de resultado

No Nim, todo procedimento que retorna um valor tem uma variável de resultado declarada implicitamente e inicializada (com um valor padrão). O procedimento retornará o valor desta variável de resultado quando atingir o final de seu bloco indentado, mesmo sem declaração de retorno.

result.nim

```
proc findBiggest(a: seq[int]): int = # (1)
  for number in a:
    if number > result:
      result = number
  # the end of proc (2)

let d = @[3, -5, 11, 33, 7, -15]
echo findBiggest(d)
```

1. O tipo de retorno é ***int***. A variável ***result*** é inicializada com o valor padrão para ***int: 0***.
2. Quando o final do procedimento é alcançado, o valor do ***result*** é retornado.

```
33
```

Observe que este procedimento serve para demonstrar a variável ***result***, e não está 100% correto: se você passasse uma sequência contendo apenas números negativos, este procedimento retornaria ***0*** (que não está contido na sequência).

> ***Cuidado!*** Em versões mais antigas do Nim (antes do ***Nim 0.19.0***), o valor padrão de ***strings*** e ***sequências*** era nulo, e quando íamos usá-los como tipos de retorno, a variável ***result*** precisaria ser inicializada como uma ***string*** vazia (***""***) ou como uma sequência vazia (***@ []***).

result.nim

```
proc keepOdds(a: seq[int]): seq[int] =
  # result = @[] (1)
  for number in a:
    if number mod 2 == 1:
      result.add(number)


let f = @[1, 6, 4, 43, 57, 34, 98]
echo keepOdds(f)
```

1. Na versão ***Nim 0.19.0*** e mais recente, esta linha não é necessária - as sequências são inicializadas automaticamente como sequências vazias.
   Em versões mais antigas do Nim, as sequências devem ser inicializadas e, sem essa linha, o compilador geraria um erro.
   (Observe que ***var*** não deve ser usado, pois o resultado já está declarado implicitamente.)

```
@[1, 43, 57]
```

Dentro de um procedimento, também podemos chamar outros procedimentos.

filterOdds.nim

```
proc isDivisibleBy3(x: int): bool =
  return x mod 3 == 0

proc filterMultiplesOf3(a: seq[int]): seq[int] =
  # result = @[] (1)
  for i in a:
    if i.isDivisibleBy3(): # (2)
      result.add(i)


let
  g = @[2, 6, 5, 7, 9, 0, 5, 3]
  h = @[5, 4, 3, 2, 1]
  i = @[626, 45390, 3219, 4210, 4126]

echo filterMultiplesOf3(g)
```

```
echo h.filterMultiplesOf3()
echo filterMultiplesOf3 i # (3)
```

1. Mais uma vez, esta linha não é necessária nas versões mais recentes do Nim.
2. Chamando o procedimento declarado anteriormente. Seu tipo de retorno é ***bool*** e pode ser usado na instrução ***if***.
3. A terceira forma de chamar um procedimento, como vimos acima.

```
@[6, 9, 0, 3]
@[3]
@[45390, 3219]
```

Declaração de encaminhamento

Conforme mencionado no início desta seção, podemos declarar um procedimento sem um bloco de código. A razão para isso é que temos que declarar os procedimentos antes de podermos chamá-los, fazer isso não funcionará:

```
echo 5.plus(10) # error (1)

proc plus(x, y: int): int = # (2)
  return x + y
```

1. Isso gerará um erro, pois o procedimento ***plus*** ainda não foi definido.
2. Aqui nós definimos ***plus***, mas como é depois de usá-lo, Nim ainda não sabe sobre isso.

A maneira de contornar isso é chamada de declaração de encaminhamento:

```
proc plus(x, y: int): int # (1)

echo 5.plus(10) # (2)

proc plus(x, y: int): int = # (3)
  return x + y
```

1. Aqui, dizemos a Nim que ele deve considerar que o procedimento plus existe com essa definição.
2. Agora estamos livres para usá-lo em nosso código, isso vai funcionar.
3. Isso é onde o plus é realmente implementado, isso deve corresponder à nossa definição anterior.

Exercícios

1. Crie um procedimento que cumprimente uma pessoa (**`imprima "Olá, <nome>"`**) com base no nome fornecido. Crie uma sequência de nomes. Cumprimente cada pessoa usando o procedimento criado.
2. Crie um procedimento findMax3 que retornará o maior dos três valores.
3. Os pontos no plano 2D podem ser representados como tupla [x, y: float]. Escreva um procedimento que receberá dois pontos e retornará um novo ponto que é a soma desses dois pontos (adicione x e y separadamente).
4. Crie dois procedimentos tick e tock que ecoam as palavras 'tick' e 'tock'. Tenha uma variável global para controlar quantas vezes eles foram executados, e execute um do outro até que o contador alcance 20. A saída esperada é obter linhas com 'tick' e 'tock' alternando 20 vezes. (Dica: use declarações de encaminhamento.)

> Você pode pressionar Ctrl + C para interromper a execução de um programa se entrar em um loop infinito.

Teste todos os procedimentos chamando-os com parâmetros diferentes.

Módulos

Até agora, usamos a funcionalidade que está disponível por padrão sempre que iniciamos um novo arquivo Nim. Isso pode ser estendido com módulos, que fornecem mais funcionalidade para algum tópico específico.

Alguns dos módulos Nim mais usados são:

- ***strutils***: funcionalidade adicional ao lidar com strings
- ***sequtils***: funcionalidade adicional para sequências
- ***math***: funções matemáticas (logaritmos, raízes quadradas, ...), trigonometria (sen, cos, ...)
- ***times***: medir e lidar com o tempo

Mas há muitos mais, tanto na chamada biblioteca padrão quanto no gerenciador de pacotes ágil.

Importando um módulo

Se quisermos importar um módulo e todas as suas funcionalidades, basta colocar **import <moduleName>** em nosso arquivo. Isso geralmente é feito na parte superior do arquivo para que possamos ver facilmente o que nosso código usa.

stringutils.nim

```
import strutils # (1)

let
  a = "My string with whitespace."
  b = '!'

echo a.split() # (2)
echo a.toUpperAscii() # (3)
echo b.repeat(5) # (4)
```

1. Importando ***strutils***.
2. Usando o módulo ***split*** de ***strutils***. Ele divide a ***string*** em uma sequência de palavras.
3. ***toUpperAscii*** converte todas as letras ***ASCII*** em maiúsculas.
4. ***repeat*** também é do módulo ***strutils*** e repete um caractere ou uma ***string*** inteira a quantidade de vezes solicitada.

```
@["My", "string", "with", "whitespace."]
MY STRING WITH WHITESPACE.
!!!!!
```

> Para os usuários vindos de outras linguagens de programação (especialmente Python), a maneira como as importações funcionam no Nim pode parecer 'errada'. Se for esse o caso, esta é a leitura recomendada.

maths.nim

```
import math # (1)

let
  c = 30.0 # degrees
  cRadians = c.degToRad() # (2)

echo cRadians
echo sin(cRadians).round(2) # (3)

echo 2^5 # (4)
```

1. Importando ***math***.
2. Convertendo graus em radianos com ***degToRad***.
3. O seno recebe radianos. Arredondamos (também a partir do módulo de ***math***) o resultado para no máximo 2 casas decimais. (Caso contrário, o resultado seria: ***0.4999999999999999***)
4. O módulo ***math*** também possui o operador ^ para calcular as potências de um número.

```
0.5235987755982988
0.5
32
```

Criando o nosso próprio

Freqüentemente, temos tanto código em um projeto que faz sentido dividi-lo em partes para que cada um faça uma determinada coisa. Se você criar dois arquivos lado a lado em uma pasta, vamos chamá-los de firstFile.nim e secondFile.nim, você pode importar um do outro como um módulo:

firstFile.nim

```
proc plus*(a, b: int): int = # (1)
  return a + b

proc minus(a, b: int): int = # (2)
  return a - b
```

1. Observe como o procedimento de adição agora tem um asterisco (*) após seu nome, isso informa ao Nim que outro arquivo importando este poderá usar este procedimento.
2. Por outro lado, isso não estará visível ao importar este arquivo.

secondFile.nim

```
import firstFile # (1)

echo plus(5, 10) # (2)
echo minus(10, 5) # error (3)
```

1. Aqui, importamos ***firstFile.nim***. Não precisamos colocar a extensão ***.nim*** aqui.
2. Isso funcionará bem e produzirá ***15***, conforme declarado no ***firstFile*** e visível para nós.
3. No entanto, isso gerará um erro, pois o procedimento ***minus*** não é visível, pois não tem um asterisco seguido ao seu nome em ***firstFile.nim***.

Caso você tenha mais do que esses dois arquivos, convém organizá-los em um subdiretório (ou mais de um subdiretório). Com a seguinte estrutura de diretório:

```
.
├── myOtherSubdir
│   ├── fifthFile.nim
│   └── fourthFile.nim
├── mySubdir
│   └── thirdFile.nim
├── firstFile.nim
└── secondFile.nim
```

Se você quiser importar todos os outros arquivos em seu ***secondFile.nim***, faça o seguinte:

secondFile.nim

```
import firstFile
import mySubdir/thirdFile
import myOtherSubdir / [fourthFile, fifthFile]
```

Interagindo com a entrada do usuário

Usar as coisas que apresentamos até agora (tipos de dados básicos e contêineres, fluxo de controle, loops) nos permite fazer alguns programas simples.

Neste capítulo, aprenderemos como tornar nossos programas mais interativos. Para isso, precisamos de uma opção para ler os dados de um arquivo ou solicitar uma entrada do usuário.

Lendo de um arquivo

Digamos que temos um arquivo de texto chamado people.txt no mesmo diretório que nosso código Nim. O conteúdo desse arquivo é parecido com este:

people.txt

```
Alice A.
Bob B.
Carol C.
```

readFromFile.nim

```
import strutils

let contents = readFile("people.txt") # (1)
echo contents

let people = contents.splitLines() # (2)
echo people
```

1. Para ler o conteúdo de um arquivo, usamos o procedimento ***readFile*** e fornecemos um caminho para o arquivo a ser lido (se o arquivo estiver no mesmo diretório de nosso programa Nim, basta fornecer um nome de arquivo). O resultado é uma ***string*** multilinha.
2. Para dividir uma ***string*** multilinha em uma sequência de ***strings*** (cada string contém todo o conteúdo de uma única linha), usamos ***splitLines*** do módulo ***strutils***.

```
Alice A.
Bob B.
Carol C.
            # (1)
@["Alice A.", "Bob B.", "Carol C.", ""] # (2)
```

1. Havia uma nova linha final (última linha vazia) no arquivo original, que também está presente aqui.
2. Por causa da nova linha final, nossa sequência é mais longa do que esperávamos.

Para resolver o problema de uma nova linha final, podemos usar o procedimento de tira de strutils depois de ler um arquivo. Tudo o que isso faz é remover os chamados espaços em branco do início e do final de nossa string. O espaço em branco é simplesmente qualquer caractere que crie algum espaço, novas linhas, espaços, tabulações, etc.

readFromFile2.nim

```
import strutils

let contents = readFile("people.txt").strip() # (1)
echo contents

let people = contents.splitLines()
echo people
```

1. O uso de ***strip*** fornece os resultados esperados.

```
Alice A.
Bob B.
Carol C.
@["Alice A.", "Bob B.", "Carol C."]
```

Lendo a entrada do usuário

Se quisermos interagir com um usuário, devemos ser capazes de pedir-lhe uma entrada e, em seguida, processá-la e usá-la. Precisamos ler a entrada padrão (***stdin***) passando ***stdin*** para o procedimento ***readLine***.

interaction1.nim

```
echo "Please enter your name:"
let name = readLine(stdin) # (1)

echo "Hello ", name, ", nice to meet you!"
```

1. O tipo de ***name*** é considerado uma ***string***.

```
Please enter your name:
        (1)
```

1. Aguardando entrada do usuário. Depois de escrever nosso nome e pressionar ***Enter***, o programa continuará.

```
Please enter your name:
Alice
Hello Alice, nice to meet you!
```

Lidando com números

A leitura de um arquivo ou de uma entrada do usuário sempre fornece uma ***string*** como resultado. Se quisermos usar números, precisamos converter ***strings*** em números: novamente usamos o módulo ***strutils*** e ***parseInt*** para converter para inteiros ou ***parseFloat*** para converter em um ***float***.

interaction2.nim

```
import strutils

echo "Please enter your year of birth:"
let yearOfBirth = readLine(stdin).parseInt() # (1)

let age = 2018 - yearOfBirth

echo "You are ", age, " years old."
```

1. Converta uma ***string*** em um inteiro. Quando escrito assim, confiamos em nosso usuário para fornecer um número inteiro válido. O que aconteceria se um usuário inserisse ***'79 ou noventa e três?*** Tente você mesmo.

```
Please enter your year of birth:
1934
You are 84 years old.
```

Se tivermos o arquivo ***numbers.txt*** no mesmo diretório que nosso código Nim, com o seguinte conteúdo:

numbers.txt

```
27.3
98.24
11.93
33.67
55.01
```

E queremos ler esse arquivo e encontrar a soma e a média dos números fornecidos, podemos fazer algo assim:

interaction3.nim

```
import strutils, sequtils, math # (1)

let
  strNums = readFile("numbers.txt").strip().splitLines() # (2)
  nums = strNums.map(parseFloat) # (3)

let
  sumNums = sum(nums) # (4)
  average = sumNums / float(nums.len) # (5)

echo sumNums
echo average
```

1. Importamos vários módulos. ***strutils*** fornece ***strip*** e ***splitLines***, ***sequtils*** fornece ***map*** e ***math*** fornece ***sum***.
2. Retiramos a nova linha final e dividimos as linhas para criar uma sequência de ***strings***.
3. ***map*** funciona aplicando um procedimento (neste caso, ***parseFloat***) a cada membro de um ***contêiner***. Em outras palavras, convertemos cada ***string*** em um ***float***, retornando uma nova sequência de ***floats***.
4. Usando ***sum*** do módulo ***math*** para nos dar a soma de todos os elementos em uma sequência.
5. Precisamos converter o comprimento de uma sequência em ***float***, porque ***sumNums*** é um ***float***.

```
226.15
45.23
```

Exercícios

1. Pergunte a um usuário sua altura e peso. Calcule seu IMC. Relate-lhes o valor do IMC e a categoria.
2. Repita o exercício de conjectura de Collatz para que seu programa peça a um usuário um número inicial. Imprima a sequência resultante.

3. Peça a um usuário uma string que ele deseja que seja invertida. Crie um procedimento que recebe uma string e retorna uma versão reversa. Por exemplo, se o usuário digitar Nim-lang, o procedimento deve retornar gnal-miN. (Dica: use indexação e contagem regressiva)

Conclusão

É hora de concluir este tutorial. Esperançosamente, isso foi útil para você e você conseguiu dar os primeiros passos na programação e/ou na linguagem de programação Nim.

Estes têm sido apenas o básico e nós apenas arranhamos a superfície, mas isso deve ser o suficiente para permitir que você faça programas simples e resolva algumas tarefas ou quebra-cabeças simples. Nim tem muito mais a oferecer e, com sorte, você continuará a explorar suas possibilidades.