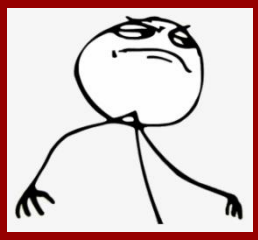
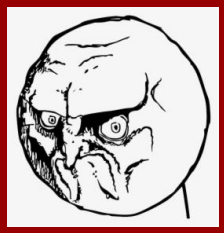
Algoritmos

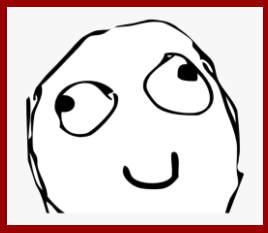
Primeiros Passos



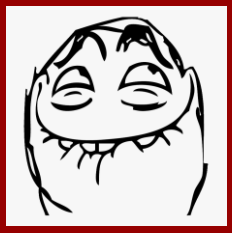
Português Estruturado



C

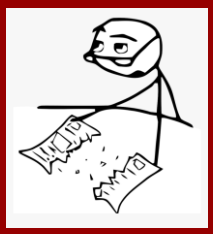


Python

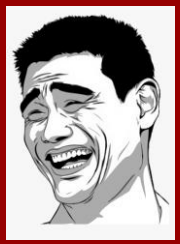


Go

C++



Ruby



Paulo Augusto Nardi

Sumário

Prefácio

Afinal, o que é algoritmo? E programa?

Segundo a ISO 24765:2017, algoritmo é uma sequência de operações (que chamaremos de instruções) para se realizar uma tarefa específica. Já, um programa pode envolver um grupo de projetos relacionados, possivelmente dividido em partes gerenciadas e coordenadas. Deste modo, observa-se que um programa possui um escopo maior, e abrange diversos algoritmos que realizam diversas tarefas.

Um algoritmo é independente de linguagem de programação. Ele pode até mesmo ser escrito em uma pseudolinguagem (também chamada de pseudocódigo). Isto é especialmente vantajoso para principiantes, para que estes se concentrem no fundamento a ser ensinado e não se preocupem com especificidades de uma linguagem de programação, como C e Java.

Por exemplo, imagine o primeiro dia de aula em que você irá aprender como apresentar uma mensagem na tela. Provavelmente é mais simples para o seu entendimento que qualquer linguagem de programação terá um início, um fim e uma instrução que realiza a apresentação da mensagem, como:

```
1.    inicio
2.       escreva(“Olá, mundo”)
3.    fim
```

Por outro lado, se você aprender algoritmo utilizando uma linguagem como Java, o mesmo código será escrito da seguinte maneira:

```
1.    public class Inicio{
2.        public static void main(String args[]){
3.            System.out.println(“Olá, mundo”);
4.        }
5.    }
```

Você terá muita informação para absorver utilizando Java. Você provavelmente entenderá que as chaves { e } representam início e fim de alguma coisa. E entenderá que a instrução para escrever algo é *System.out.println()*. Mas também irá se perguntar: o que é classe, o que é pública, o que é aquele *static*, *void*, *main*, *String*, etc...

Por isso muitos professores preferem ensinar algoritmos utilizando algum pseudocódigo. Como o objetivo é apresentar (e focar) os conceitos básicos que estão presentes nas linguagens de programação, usaremos pseudocódigo. E, ao final de cada seção, darei exemplos daquilo que aprendemos em cinco linguagens bastante utilizadas: C, C++, Go, Ruby e Python. Todas linguagens de propósito geral.

C e C++ são linguagens "atemporais", que são exigidas em muitas empresas pelo mundo. Por isso são muitas vezes as primeiras linguagens a serem ensinadas. C foi criada em 1972 e C++, em 1985 com atualizações até hoje. A principal diferença entre as duas é que C++ suporta orientação a objetos. Ela é compatível com C. Assim, tudo o que se faz em C pode ser feito em C++. Mas o inverso não é verdadeiro.

Go foi criada pela Google em 2007. Hoje é usada por grandes empresas (além da própria Google) como Netflix, American Express, Meta, Microsoft e outras. Ela é semelhante ao C++, mas foi criada com foco em gerenciamento de memória seguro e execução concorrente, que são características importantes quando se deseja criar sistemas em nuvem, por exemplo.

Ruby, apesar de ser de propósito geral, foi criada pensando-se no desenvolvimento Web. É usada por empresas como AirBnB, Hulu e Github. É considerada uma linguagem versátil e portável.

Python é uma linguagem, de certa forma, parecida com o Ruby (você perceberá isso neste livro). Elas se parecem mais entre si do que com C, C++ ou Go. Ela é considerada mais fácil de aprender por iniciantes e também é muito usada para desenvolvimento Web. Exemplos de empresas que usam esta linguagem são Netflix (sim, uma mesma empresa usa mais de uma linguagem de programação! Tudo depende do propósito), Pixar, NASA, IBM e Youtube.

O importante é ter em mente que um algoritmo feito com pseudocódigo é facilmente reescrito na linguagem de programação que lhe for apropriada.

Pronto! Abordamos um pouco sobre Algoritmos e as linguagens de programação que veremos neste livro. Agora vamos abordar como devemos usar este livro!

Um ponto muito importante para quem está aprendendo (qualquer que seja o conteúdo de qualquer que seja a disciplina) é entender as etapas do aprendizado. Isto é útil para nós nos analisarmos e sabermos o que mais precisamos para dominar um conteúdo. É assim que aprendemos a nos tornar autodidatas! Característica, esta, muito importante para a nossa vida profissional. Assistiu ao Karatê Kid? Em qualquer que seja a versão, o senhor Miyagi ou o senhor Han ensinam seus respectivos pupilos a lutar. E como ensinam? Pense nisso enquanto lê os passos do aprendizado a seguir.

O primeiro passo do aprendizado é o entendimento. Muitas vezes entender o assunto é difícil e leva algum tempo até "cair a nossa ficha" (aquela sensação de: ahhhh, captei!). Há vezes, porém, em que é fácil entender o assunto ensinado. E por que este comentário é relevante a ponto de ser escrito no prefácio de um livro? Porque em ambos os casos é fácil nos enganarmos e acharmos que já sabemos um assunto. Na verdade, o entendimento implica que você conseguiu ter uma visão geral sobre o conteúdo. Para quem está iniciando Algoritmos, os primeiros passos parecem simples porque geralmente entendemos com facilidade o novo conteúdo. Isso nos dá a falsa sensação de que já estamos sabendo e nos sentimos tentados a seguir para o próximo conteúdo. É aí que faço um alerta! Uma vez que você tenha entendido, não deixe de fazer os exercícios. Entender é o primeiro passo!

O segundo passo do aprendizado é desenvolver a sua própria linha de raciocínio. Isso mesmo! Entender o assunto é uma parte "passiva" do aprendizado. Você leu (ou viu, ou ouviu) algo que fez sentido e compreendeu o que estava sendo passado. Agora é a hora de fazer exercícios para que você desenvolva ativamente o que entendeu. Você precisará "pensar" em como o quebra-cabeça será montado. Não está apenas observando alguém montar o quebra-cabeça. Nessa etapa, você eventualmente precisará rever alguns pontos que acabou esquecendo ou que não pareciam muito importantes para o momento. Quando você faz os exercícios iniciais, assim que entende algo, você começa a ter a visão dos pormenores, do passo a passo, daquilo que entendeu.

Uma vez que você tenha feito os exercícios iniciais e tenha passado pelos detalhes do que está aprendendo, agora é hora de fixar! E essa é outra parte que muitas vezes erradamente acabamos pulando. É importantíssimo treinar ainda mais até que o conteúdo se torne automático! Nesse momento, enquanto estamos treinando, estamos criando nossas sinapses! Neurônios estão se conectando para

forma uma rede de informação mais duradoura que também se conectará a outras redes de forma a criar relações que nunca imaginamos que seríamos capazes de adquirir!

É nessa hora que nosso cérebro ganha a tão “misteriosa” intuição! A partir de quanto você domina um assunto e vai resolver um outro algoritmo um pouco diferente, seu cérebro, mesmo que você não esteja consciente disso, vai te avisar que você pode ter uma resposta! É aquela hora que você diz: “hum, acho que posso resolver isso”. Ou “eu sei que sei, só não sei como”.

E o que tem o Karatê Kid com isso? Treino, treino, treino. Encera o carro, pinta a cerca... ou tira o casaco, põe o casado! Em resumo: não deixe de treinar só porque entendeu; e não deixe de treinar só porque conseguiu fazer uma lista de exercícios. Enquanto você treina, seu cérebro cria sinapses. E isso não acontece de um dia para o outro.

Uma boa forma de treinar, é tentar criar você mesmo o enunciado de um exercício. De outra forma: crie seu próprio desafio!

Para ajudá-lo a entender o conteúdo, é interessante ter diferentes estímulos: seja a explicação do professor em aula, leitura de livros, visualização de vídeos. Quanto mais estímulos diferentes, melhor. Caso tenha interesse, o link abaixo aborda o conteúdo dado no livro:

https://www.youtube.com/channel/UC96Gf3paZctR3PKuzn99X_w

No primeiro capítulo, iremos nos concentrar em como construir nossos algoritmos, isto é, qual a estrutura básica de um algoritmo e suas instruções. No segundo capítulo, veremos como organizar nossos dados. No terceiro capítulo, melhoraremos nossa organização do código, dividindo-o em pequenas partes, chamados módulos (funções!!!). E, no último capítulo, criaremos um pequeno projeto de Jogo da Velha com aquilo que aprendemos.

Capítulo 1: Estrutura Básica

Um algoritmo é formado por uma sequência de instruções, que por sua vez, precisam ser estruturadas de alguma forma. Linguagens de programação como C e Java possuem indicadores que representam o início e o fim do código. No nosso pseudocódigo, usaremos as palavras *início* e *fim,* como apresentado abaixo*:*

```
1.    inicio
2.       <instrução>
3.       <instrução>
4.       ...
5.    fim
```

Deste modo, todas as instruções que usaremos em nosso algoritmo devem estar entre as palavras ***inicio*** e ***fim***. Chamamos este espaço de bloco principal do programa.

Repare, também, que as instruções que estão entre estas palavras estão ligeiramente deslocadas para a direita. Isso se chama indentação e será mais explorado em breve. É importante realizar a indentação para que o código seja lido com mais facilidade. Assim, quando você olhar o código, notará rapidamente que as instruções estão "dentro" do bloco principal do programa.

Instruções Escreva, Leia e Variáveis.

Uma das ações mais básicas ao se criar um programa é apresentar alguma mensagem na tela. Este é provavelmente o primeiro meio de interação entre o programa e o usuário. Em nosso pseudocódigo, a instrução responsável por essa ação é ***escreva***. Assim, se quisermos criar um algoritmo que apresente a mensagem "Olá, mundo!", deveremos fazê-lo usando esta instrução dentro de início e fim:

```
1.    inicio
2.        escreva("Olá, mundo!!!")
3.    fim
```

Fizemos nosso primeiro algoritmo completo! Ele tem início, realiza uma tarefa e tem um fim! É interessante notar um detalhe: repare que o texto a ser apresentado está entre aspas.

Podemos escrever vários textos e juntá-los (concatená-los) em um só. Fazemos essa junção de textos (concatenação) com o uso de uma vírgula. Por exemplo:

```
escreva("Olá! ", "Sou um texto ", "concatenado")
```

Repare que a vírgula está fora das aspas. Quando ela está fora das aspas representa uma operação de concatenação. Esse exemplo tem exatamente o mesmo efeito de:

```
escreva("Olá! Sou um texto concatenado")
```

Outro ponto importante: tudo o que estiver entre aspas é reconhecido como um texto fixo, que não se altera. E porque é importante reparar que o texto não se altera? Não é isso que sempre queremos?

Nem sempre! E se quisermos que o usuário digite o seu nome e, em seguida, o algoritmo apresente a mensagem *Seja bem vindo, <nome>*? Repare que, nesse caso, o nome que será apresentado na mensagem irá **variar** de acordo com o que for digitado.

Para conseguirmos realizar esta ação do usuário digitar um nome e esse nome ser apresentado, precisamos:

1. Pedir para o usuário digitar seu nome;
2. Ter uma instrução que espera o usuário digitar algo;
3. Ter algum recurso para que o nome digitado seja momentaneamente armazenado na memória do computador para que ele seja usado em algum outro momento;
4. Usar a instrução que já conhecemos de apresentar mensagens na tela.

A instrução que usaremos para o algoritmo aguardar o usuário digitar algo é ***leia***. Mas repare que apenas esperar que algo seja digitado não basta. O que for digitado precisa ser armazenado na memória para que seja usado depois. É como em nós, seres humanos. Se alguém se apresentar e falar seu nome, você precisará reter esse dado em sua memória, para usá-lo depois. Mesmo que para dizer "Olá, <fulano>".

Então a pergunta é: como fazemos isso em um algoritmo? Como guardamos dados que possam variar de acordo com o que o usuário digitar? Usamos um recurso chamado de variável!

Uma **variável** é uma representação da memória que permite armazenar um dado que pode variar. A variável também permite que esse dado seja recuperado para que possa ser usado quando necessário.

Uma variável possui as seguintes características:

- Um nome (que chamamos de **identificador**);
- Representa um **espaço da memória** onde o dado será armazenado (que chamamos de endereço de memória);
- E o **valor** a ser armazenado (o nosso dado).

A próxima pergunta é: então como criamos uma variável para podermos usá-la?

Antes de respondermos a esta pergunta, precisamos entender uma quarta propriedade, o **tipo da variável**. Muitas linguagens de programação exigem (ou simplesmente permitem) que se especifique qual o tipo de dado será armazenado pela variável: se é um número inteiro, um número real, um caractere, uma cadeia de caracteres (como uma frase) ou um valor tipo sim/não (ou verdadeiro e falso), que chamamos de lógico.

Esse recurso de se especificar o tipo da variável tem algumas vantagens, como reduzir possíveis maus usos das variáveis por parte do programador. Isso pode não ficar claro agora, no início do aprendizado, mas não iremos a fundo na explicação para mantermos o foco no que é importante.

O importante, no momento, é entender que para usarmos uma variável, precisamos criá-la. E para criá-la, precisamos declarar o seu tipo de dados e o seu nome (identificador). Após declarada, a variável está pronta para ser usada.

As regras para dar nomes a uma variável são:

- O nome da variável (a partir deste ponto chamaremos de identificador da variável) deve começar com uma letra;
- Os caracteres seguintes podem ser letras ou números;

- O identificador da variável deve ser algo significativo para que o programador se lembre do que ela se trata.

Estas são as três regras básicas. Há linguagens de programação que permitem o uso de alguns caracteres especiais, como _ (*underline*). Mas, para nosso aprendizado, seguiremos estas três regras básicas.

Os tipos de variáveis são:

- Inteiro: use esse tipo quando o valor da variável sempre for um número inteiro, como idade, ano e RA;
- Real: use esse tipo quando o valor puder ser um número com valores decimais, como temperatura, nota e salário;
- Caractere: use esse tipo quando o valor da variável sempre for uma única letra (ou outro caractere, como: opção **m** para masculino e **f** para feminino);
- Cadeia: use esse tipo quando o valor da variável for um texto;
- Lógico: use esse tipo quando o valor a ser armazenado possa ser representado apenas por dois possíveis valores, como sim e não, verdadeiro e falso ou 0 e 1.

A declaração da variável segue abaixo:

```
<Tipo da variável> <identificador da variável>
```

Muito bem! Agora que conhecemos as regras sobre as variáveis, vamos aprender como utilizá-las no algoritmo. Ainda que não seja obrigatório, sempre declararemos as variáveis logo no **início** do algoritmo. Isso é importante por motivos de facilidade de leitura do código. Se declararmos variáveis ao longo do algoritmo, fica mais difícil identificar quais variáveis criamos e onde estão.

Para o problema inicial, em que desejamos que o usuário digite o seu nome para ser apresentado na tela, podemos fazer o seguinte algoritmo:

```
1.    início
2.       cadeia nome
3.       escreva(“Digite o seu nome:”)
4.       leia(nome)
5.       escreva(“Seja bem vindo ” , nome)
6.    fim
```

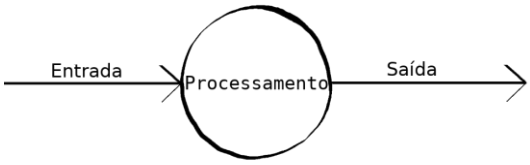
Há vários detalhes interessantes a serem observados:

- Declaramos a variável como descrito anteriormente: <tipo> <identificador>
- O identificador da variável é significativo. Quando olhamos para o código, sabemos que a variável tem o objetivo de armazenar nomes. Se chamássemos apenas de *a*, precisaríamos entender o contexto para saber que a variável armazena nomes;
- Usamos a variável *nome* dentro dos parênteses da instrução leia. E sem aspas! Por quê? Porque tudo que está entre aspas é entendido como um texto, e não uma variável!
- Ao se executar a instrução, o que o usuário digitar será automaticamente armazenado na variável *nome*;
- Para podermos apresentar o nome do usuário na tela, precisamos juntar um texto (*seja bem-vindo*) com uma variável. Isso se chama concatenação e se faz usando uma vírgula.

Agora que sabemos armazenar valores e apresentá-los, seguimos adiante para o próximo passo: podemos manipular os valores das variáveis! Podemos realizar processamentos!

Um algoritmo simples pode ser grosseiramente dividido em três partes: uma para entrada de dados, uma para processamento e outra para apresentar dados Figura 1. Ou, de maneira mais geral: entrada de dados, processamento e saída.

Figura 1 - Representação de um algoritmo



Vamos supor que queiramos criar um algoritmo que peça a idade do usuário, calcule o ano de nascimento (aproximado) e apresente-o na tela.

Então, como se faz o cálculo e o que é necessário? Podemos representar a expressão aritmética da seguinte forma:

ano = 2022 – idade

Repare que temos o uso de duas variáveis. Precisamos da variável **idade** que possui um valor digitado pelo usuário e temos a variável **ano**, que armazenará o resultado do cálculo. No linguajar de algoritmos, dizemos que a variável **ano** *recebe* o resultado de 2022 menos a idade.

Em nosso pseudocódigo, substituiremos o sinal = por <- (como uma seta, feita com os caracteres < e -).

Então, o mesmo código, em nosso algoritmo se torna:

```
ano <- 2022 – idade
```

O algoritmo completo é apresentado a seguir:

```
1.    inicio
2.        inteiro ano
3.        inteiro idade
4.        escreva(“Digite a sua idade:”)
5.        leia(idade)
6.        ano <- 2022 - idade
7.        escreva(“Você nasceu em ”, ano)
8.    fim
```

Como *ano* e *idade* são do mesmo tipo, é possível simplificar suas declarações em apenas uma linha:

```
inteiro ano, idade
```

Por fim, os **operadores aritméticos** básicos que podem ser usados em nossos algoritmos são: + (soma), - (subtração), * (multiplicação) e / (divisão).

Assim como na matemática, os operadores * e / têm precedência em relação a + e -. Portanto:

```
2 + 3 * 5= 17
```

e

```
(2 + 3) * 5 = 25
```

São expressões diferentes!

Até o momento, vimos a estrutura básica de um algoritmo e duas instruções: ***escreva*** e ***leia***. Vimos, também, o conceito de variáveis. Agora é hora de uma pausa para fixar o aprendizado.

Dica: Lembre-se que o aprendizado se divide em três partes: o entendimento, a fixação e a capacidade de solucionar novos problemas que possam ser resolvidos com aquilo que já aprendeu. Por isso, é importante resolver o máximo de problemas que puder. Quanto mais problemas resolver, mais você terá capacidade de resolver outros novos problemas. Uma última dica é desenvolver seus próprios problemas. Use aquilo que aprendeu para criar seus próprios exercícios ou criar algo que tenha vontade de fazer.

Exercícios de Instruções escreva, leia e variáveis

1. Faça um algoritmo que apresente a mensagem “Olá, mundo!!!”
2. Faça um algoritmo que concatene as seguintes mensagens em uma só:
    - “Eu estou ”
    - “Fazendo “
    - “concatenação!”

3. Faça um algoritmo que concatene as seguintes mensagens em uma só:
    - “Olá, amigo! ”
    - “Este texto tem uma vírgula dentro das aspas. ”
    - “Mas eu sei que quando está dentro das aspas ela representa apenas uma vírgula. ”
    - “E quando está fora representa o operador de concatenação.”

4. Faça um algoritmo que peça o nome do usuário e, em seguida, digite a mensagem: “Seja bem vindo, <nome do usuário>.

5. Faça um algoritmo que peça a idade do usuário e, em seguida, digite a mensagem: “Legal! Você tem <idade> anos!”.

6. Faça um algoritmo que peça o nome do usuário, depois a idade do usuário e, em seguida, digite a mensagem: “Legal, <nome do usuário>! Você tem <idade> anos!”.

7. Faça um algoritmo que peça a idade do usuário, depois o ano de nascimento do usuário, e apresente a mensagem: “Você tem <idade> anos e nasceu em <ano de nascimento>!”

8. Faça um algoritmo que peça o ano de nascimento do usuário e apresente sua provável idade.

9. Faça um algoritmo que peça a idade do usuário e apresente seu provável ano de nascimento.

10. Faça um algoritmo que peça uma temperatura em graus célsius e apresente seu respectivo valor em farenheits. O cálculo de conversão é: F= (9*C/5) + 32.

11. Faça um algoritmo que peça uma temperatura em graus farenheits e apresente seu respectivo valor em celsius. O cálculo de conversão é: C= 5*(F-32)/9.

12. Faça um algoritmo que peça um valor inteiro (e o armazene em uma variável de nome a). Peça um segundo valor inteiro (e o armazene em uma variável de nome b). E, troque o valor da variável a por b e vice-versa. Por exemplo, se o usuário digitar primeiro 2, depois 6 (a=2 e b=6), no fim do algoritmo, o valor de a deverá ser 6 e o de b deverá ser 2. Por fim, apresentar os resultados para o usuário.

Como é em algumas linguagens

Sugestões:

- Se você está aprendendo Algoritmos agora e não tem conhecimento prévio, fique apenas no Português Estruturado e use esta seção para ter uma noção de como é em outras linguagens, sem se preocupar em aprendê-las (e perceber como cada Linguagem tem suas particularidades e ao mesmo tempo, algo em comum). Quando terminar o livro, retorne às seções "Como é nas linguagens" e escolha apenas uma linguagem para aprender;
- Se você está aprendendo Algoritmos em conjunto com alguma outra linguagem (geralmente em C ou C++), use esta seção para aprender como fazer o que viu nas seções anteriores com a linguagem que está usando caso ela seja uma destas;
- Se você já tem noção de Algoritmos e quer aprender alguma das linguagens mostradas nesta seção, vá em frente!

Português Estruturado

```
inicio
    inteiro ano
    inteiro idade
    escreva("Digite a sua idade:")
    leia(idade)
    ano <- 2020 - idade
    escreva("Você nasceu em " , ano, " e tem ", idade, " anos")
fim
```

C

```
#include<stdio.h>

int main(){
    int ano, idade;
    printf("\nDigite a sua idade: ");
    scanf("%d",&idade);
    ano = 2022 - idade;
    printf("Você nasceu em %d e tem %d anos",ano, idade);
    return 0;
}
```

Obs: em C e C++, **int main(){** é o nosso INÍCIO. E o **}** do fim do programa representa o nosso **FIM**.

O **#include<stdio.h>** serve para podermos usar as instruções **printf** e **scanf**. Nestas linguagens, as instruções estão no arquivo **stdio.h**. Caso não usemos o include, a linguagem achará que as instruções não existem.

O **printf** (nosso **escreva**) tem um recurso interessante de formatação. Para não ficar tendo que abrir e fechar aspas o tempo todo quando quisermos concatenar variáveis, usamos **especificadores de formato**, onde gostaríamos que os valores destas variáveis sejam colocados. No nosso exemplo, usamos %d que representa o formato decimal de números inteiros. Em seguida ao texto entre aspas, colocamos as variáveis na ordem em que quisermos que apareçam. Então, o valor de ano aparecerá no lugar do primeiro %d e o valor de idade aparecerá no lugar do segundo %d. Se quiséssemos que os mesmos valores fossem apresentados em hexadecimal, por exemplo (em vez de decimal), usaríamos %a no lugar de %d.

Especificadores de formato comumente usados são:

- %d para valores de variáveis do tipo int;
- %f para valores de variáveis do tipo float;
- %lf para valores de variáveis do tipo double;
- %c para valores de variáveis do tipo char
- %s para valores de variáveis de string.

O **scanf** (nosso leia) também usa marcadores. Mas repare que, além disso, ele tem um **&** antes do nome da variável. Não precisa se preocupar com isso neste momento. Mas o & significa que estamos passando o endereço inicial do espaço de memória que nossa variável representa. Com esse endereço inicial, a instrução coloca o valor que o usuário digitar no espaço de memória correto (lembre-se de que uma variável representa um espaço de memória).

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    int ano, idade;
    cout << "Digite a sua idade:";
    cin >> idade;
    ano = 2022 - idade;
    cout << "Voce nasceu em " << ano << " e tem " << idade << " anos";
    return 0;
}
```

Obs: em C e C++, **int main(){** é o nosso INÍCIO. E o **}** do fim do programa representa o nosso **FIM**.

O **#include<iostream>** serve para podermos usar as instruções **cout** e **cin**. Nestas linguagens, as instruções estão no arquivo **iostream**. Caso não usemos o include, a linguagem achará que as instruções não existem.

As instruções que existem em C podem ser usadas em C++. Então, poderíamos usar printf e scanf como no exemplo do C.

cout (nosso escreva) é uma instrução específica do C++. O << representa um **operador de inserção**. Não se preocupe em entender isso agora. Caso esteja muito curioso: o que << faz é inserir os dados no stream de saída (cout) e que consequentemente será apresentado na tela.

cin (nosso leia) também é uma instrução específica do C++. O >> representa um operador de extração. Da mesma forma, não se preocupe com isso, agora. Mas este operador extrai os valores da stream de entrada do usuário.

Go

```
package main

import (
    "fmt"
)

func main() {
    var idade, ano int
    fmt.Print("Digite a sua idade: ")
    fmt.Scanf("%d", &idade)
    ano = 2022 - idade
    fmt.Println("Voce nasceu em", ano, "e tem", idade, "anos")

}
```

Obs: em Go, **func main(){** é o nosso INÍCIO. E o **}** do fim do programa representa o nosso **FIM**. Essa linguagem é fortemente baseada em C.

A declaração das variáveis é um pouco diferente do nosso Português estruturado e do C/C++.

Em Go, os arquivos de programa são organizados em pacotes (packages). O **package main** "diz" ao compilador Go que o pacote em questão deve ser compilado como um programa executado, e não como uma biblioteca compartilhável.

O **import("fmt")** serve para podermos usar as instruções P**rint** e **Scanf**. Nestas linguagens, as instruções estão em **fmt**. Caso não usemos o import, a linguagem achará que as instruções não existem.

fmt.Print e **fmt.Println** são a nossa instrução **escreva**. A diferença entre as duas é que print não pula linha assim que é executada e println pula uma linha após ser executada (o ln significa line). Como queremos digitar a idade ao lado da mensagem, então usamos fmt.Print da primeira vez. Essa linguagem também possui fmt.Printf, para quem prefere algo parecido com C.

Fmt.Scanf é a nossa instrução leia e funciona de forma parecida com o scanf do C/C++. Ele também tem um **&** antes do nome da variável. Como observado em **C**, não precisa se preocupar com isso neste momento. Mas o **&** significa que estamos passando o endereço inicial do espaço de memória que nossa variável representa. Com esse endereço inicial, a instrução coloca o valor que o usuário digitar no espaço de memória correto (lembre-se de que uma variável representa um espaço de memória).

Ruby

```
print "Digite a sua idade: "
idade = gets.chomp.to_i
ano = 2022 - idade
puts "Voce nasceu em " + ano.to_s + " e tem " + idade.to_s + " anos"
```

Obs: em Ruby, não há um marcador de INÍCIO-FIM do programa! O programa começa a partir do início do texto.

Também não há declaração de variável! O tipo da variável será determinado durante a atribuição de um valor. Se um número inteiro for atribuído a uma variável, ela passará a ser do tipo inteiro, e assim por diante. Linguagens desse tipo são chamadas dinâmicas (E outras como C/C++ e Go são chamadas de estáticas porque o tipo deve ser determinado durante a programação, e não durante a execução do programa). Essa linguagem também permite que uma variável troque de tipo. Então, se inicialmente uma variável recebeu um valor inteiro, ela será do tipo inteiro, mas se depois receber um valor de cadeia de caracteres, será do tipo cadeia (string).

print e **puts** são nosso **escreva**. As principais diferenças entre os dois é que o primeiro não pula linha (exceto se usar "\n" que é um marcador para pular linha) e puts automaticamente pula linha após ser executado. Há também diferença em como tratam a concatenação. Há três sinais de concatenação: a vírgula (,), o operador de adição (+) e <<. Exemplos:

`print "a", "b", "c"` gera a saída **abc**
`puts "a","b","c""` gera a saída
a
b
c

`puts "a" + "b" + "c""` e `puts "a" << "b" << "c"` geram a saída **abc**

gets é o nosso leia. Essa instrução sempre retorna uma cadeia de caracteres do teclado. Mesmo que o usuário digite apenas números, eles serão considerados cadeia de caracteres. Então, assim que o valor for digitado pelo usuário, precisamos convertê-lo para inteiro. Por isso o **.to_i**. Assim, **gets.to_i** lê uma cadeia de caracteres do teclado e a converte em um número inteiro. Outras conversões podem ser feitas como abaixo:
a = gets.to_f para converter uma cadeia de caracteres para real.
a = gets.to_i para converter uma cadeia de caracteres para inteiro.
nome = gets.chomp para remover o <enter> do teclado. Caso contrário, a cadeia de caracteres terá como último caractere o próprio <enter>.
nome = gets.chop para remover o último caractere do teclado. Se esse último caractere for o <enter>, ele removerá apenas o <enter> e se comportará como o chomp.
Nome = get.strip para remover todos os espaços antes e depois da cadeia de caracteres, mas não no meio. Então se o usuário digitar:

gosto de Algoritmos <enter>

O resultado apresentado na tela será:
gosto de Algoritmos<fim da cadeia de caracteres>

Python

```
idade = int(input('Digite sua idade: '))
ano = 2022 - idade
print('Você nasceu em', ano, 'e tem', idade, 'anos')
```

Obs: em Python, também não há um marcador de INÍCIO-FIM do programa! O programa começa a partir do início do texto.

E também não há declaração de variável! Assim como em Ruby, o tipo da variável será determinado durante a atribuição de um valor. Se um número inteiro for atribuído a uma variável, ela passará a ser do tipo inteiro, e assim por diante. Linguagens desse tipo são chamadas dinâmicas (E outras como C/C++ e Go são chamadas de estáticas porque o tipo deve ser determinado durante a programação, e não durante a execução do programa). Essa linguagem também permite que uma variável troque de tipo. Então, se inicialmente uma variável recebeu um valor inteiro, ela será do tipo inteiro, mas se depois receber um valor de cadeia de caracteres, será do tipo cadeia (string).

input é a nossa instrução **leia** junto com **escreva**. Isto é, o que estiver dentro dos parênteses e entre apóstrofos será exibido na tela (como o **escreva** faria), e irá esperar o usuário digitar algo (como o **leia** faria). Como a instrução input sempre lê um valor do teclado como uma cadeia de caracteres, precisamos converter o valor digitado para um inteiro. É por isso que a instrução input está dentro dos parênteses da instrução int().

int(<um valor>) tentará converter o valor que estiver entre parênteses em um número inteiro.

print é nossa instrução escreva. Repare que usamos apóstrofos, mas poderia ser aspas.

Tipos de variáveis (em negrito estão os tipos básicos)

Português Estruturado	C/C++	GO	Ruby	Python
inteiro	**int**	**int** int8 int16 int32 int64	Apesar de Ruby e Python possuirem tipos como inteiro, real e strings, as variáveis não são declaradas. Quando um valor é atribuído a uma variável, ela passa a ser do tipo ao qual o valor pertence. Isso é chamado de tipo dinâmico (o tipo da variável é definido em tempo de execução). Ainda, uma mesma variável pode ter diferentes tipos durante a execução do programa. Se em um momento uma variável receber um valor inteiro, ela será do tipo inteiro. Se, em seguida, essa mesma variável receber uma cadeia de caracteres, ela passará a ser do tipo cadeia. C/C++ e GO possuem tipos estáticos, isto é, o tipo da variável deve ser declarado inicialmente, durante a programação, e não pode mudar durante a execução do programa.	
real	**float** (4 bytes) **double** (8 bytes)	**float32** (4 bytes) **float64** (8 bytes) **complex64** c**omplex128**		
caractere	**char**	(não tem, usa string)		
lógico	**bool**	**bool**		
cadeia	(não tem, precisa criar um vetor de caracteres)	**string**		

Estrutura de Controle de Decisão se-senão

Nesta seção, veremos como criar algoritmos que tomam decisões conforme os dados de entrada! Por exemplo, como criar um algoritmo que identifique se o usuário pode dirigir e, depois, enviar uma mensagem de despedida?

O conceito é simples. O algoritmo deve:

- Pedir para que o usuário digite sua idade;
- Leia a idade do usuário e a armazene em uma variável;
- SE a idade do usuário for maior ou igual a 18, o algoritmo deve apresentar a mensagem "você pode dirigir" e a mensagem "Que legal!";
- Independente se for maior ou menor de idade, o algoritmo deverá apresentar a mensagem "Tchau!".

Se prestar atenção, verá que você sabe resolver mais da metade do problema. Precisamos apenas conhecer a instrução que representa SE (e que por um acaso é ***se***).

Mas, antes de abordarmos a instrução, precisamos conhecer os **operadores relacionais**, isto é, como representar comparadores como *maior*, *menor*, *maior* ou *igual*, *menor ou igual*, *diferente* e *igual*. Os respectivos operadores são apresentados na tabela abaixo:

Tabela 1: Operadores Relacionais

Operador	Significado durante a comparação
==	Igual
!=	Diferente
<	Menor
>	Maior
<=	Menor ou igual
>=	Maior ou igual

Uma vez que conhecemos os operadores relacionais, agora é possível desenvolver o algoritmo:

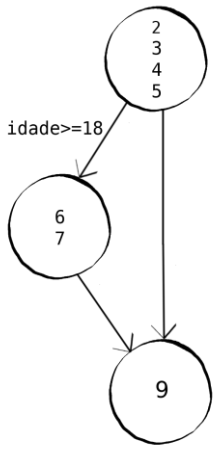
```
1.    inicio
2.        inteiro idade
3.        escreva("Digite a sua idade:")
4.        leia(idade)
5.        se(idade>=18) então
6.            escreva("Você pode dirigir!")
7.            escreva("Que legal!")
8.        fimse
9.        escreva("Tchau!")
10.   fim
```

A instrução ***se*** é bastante simples e intuitiva. Mas há um detalhe interessante a se reparar: para marcar o fim da instrução **se**, usamos o marcador ***fimse***. Por que esse marcador é importante? Sem ele, não é possível saber quantas instruções devem ser executadas quando a condição for verdadeira!

Deste modo, no nosso exemplo, se a idade for maior ou igual a 18, duas instruções deverão ser executadas. Dizemos que as instruções que estão entre a condição e o marcador ***fimse*** fazem parte do bloco da instrução ***se*** (ou apenas, bloco **se**).

Em seguida, após o bloco ***se*** ser executado, o algoritmo apresentará a mensagem "tchau!".

Figura 2 - Grafo SE



E se a condição for falsa? Isto é, e se a idade for menor do que 18? Então as instruções do bloco ***se*** não serão executadas, e o algoritmo continuará na próxima instrução da linha 9, apresentando a mensagem "Tchau!".

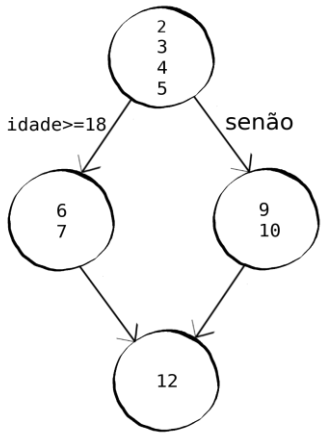
O fluxo entre as instruções, que chamamos de fluxo de controle, pode ser representado como no grafo da Figura 2. O primeiro nó (círculo) representa as instruções das linhas 2, 3, 4 e 5. É possível afirmar que todas estas instruções serão executadas em sequência. O nó mais à esquerda representa as linhas 6 e 7, e só serão executadas se a condição da linha 5 for verdadeira. E o último nó representa a linha 9 e será executada, independente da condição da linha 5.

Agora, vamos incrementar um pouco o mesmo exemplo. Imagine que queremos fazer um algoritmo que escreva “Você pode dirigir” e “Que legal!” caso o usuário digite uma idade igual ou maior que 18, mas que escreva “Você não pode dirigir” e “Que pena!” caso tenha idade inferior a 18:

```
1.    inicio
2.        inteiro idade
3.        escreva(“Digite a sua idade:”)
4.        leia(idade)
5.        se(idade>=18) então
6.            escreva(“Você pode dirigir!”)
7.            escreva(“Que legal!”)
8.        senão
9.            escreva(“Você não pode dirigir!”)
10.           escreva(“Que pena!”)
11.       fimse
12.       escreva(“Tchau!”)
13.   fim
```

Neste exemplo, usamos a palavra ***senão*** para indicar o que deve ser executado caso a condição da linha 5 seja falsa.

Figura 3 - Grafo de um se-senão



Ao mesmo tempo, essa palavra indica o fim do bloco ***se***. As instruções das linhas 6 e 7 apenas serão executadas se a condição da linha 5 for verdadeira. E as instruções das linhas 9 e 10 apenas serão executadas se a condição da linha 5 for falsa. Independente da condição ser falsa ou verdadeira, depois do respectivo bloco ser executado, a instrução da linha 12 será executada. O fluxo do Algoritmo pode ser visualizado na Figura 3.

Exercícios de se-senão

1. Faça um algoritmo que peça o salário de um funcionário. Se o salário for maior do que R$ 10.000, o programa deverá apresentar a mensagem “você ganha bem!”

2. Faça um algoritmo que peça o salário de um funcionário. Se o salário for menor do que R$ 1.045, o algoritmo deve apresentar a mensagem “você ganha menos do que um salário mínimo”. Do contrário, o algoritmo deve apresentar a mensagem “você ganha pelo menos um salário mínimo”.

3. Faça um algoritmo que peça a temperatura corporal do usuário. Se a temperatura for menor que 37, apresentar a mensagem “você está sem febre”. Do contrário, apresentar a mensagem “você está com febre!”.

4. Faça um algoritmo que peça uma senha (apenas numérica). Se a senha for diferente de 123456, apresentar a mensagem “senha não confere!”.

5. Faça um algoritmo que peça uma senha (apenas numérica). Em seguida, o algoritmo deve pedir a confirmação da senha. Se a senha repetida for igual à original, apresentar a mensagem “Senha criada com sucesso”. Do contrário, apresentar a mensagem “senha não confere”.

6. Faça um algoritmo que peça uma nota N1 e, em seguida, uma nota N2. O algoritmo deve calcular a média das duas notas e apresentar a mensagem “Aprovado”, caso o resultado seja maior ou igual a 6; ou “Reprovado”, caso o resultado seja inferior a 6.

7. Faça um algoritmo que peça o nome de um usuário. Em seguida, o algoritmo deve pedir o gênero do usuário (‘f’ para feminino e ‘m’ para masculino). Se o usuário for do sexo feminino, apresentar a mensagem “Olá, senhora <nome>“. Do contrário, apresentar a mensagem “Olá, senhor <nome>“. Observação: faça de conta que o usuário sempre digitará ‘f’ ou ‘m’.

8. Faça um algoritmo que peça dois valores inteiros. Em seguida, o algoritmo deve apresentar primeiro o valor mais baixo e, depois, o valor mais alto. Observação: faça de conta que o usuário nunca digitará o mesmo valor nas duas vezes.

9. Em uma indústria de parafusos, um algoritmo determina se a produção está boa ou ruim. Uma produção é ruim quando o número de parafusos com defeito for maior do que 10% do total produzido. Faça um algoritmo que peça o número total de parafusos produzidos e, em seguida, peça o número de parafusos com defeito. O algoritmo, então, deve calcular o limite de peças com defeito permitidas (com base no total de peças produzidas e a porcentagem limite) e apresentar se a produção está boa ou ruim.

10. Faça um programa que peça um número inteiro e apresente a mensagem "é par" ou "é ímpar". Dica: use o operador %. Este operador retorna o resto de uma divisão. Por exemplo: 5%2 retornará o resto da divisão de 5 por 2. Então, 5%2=1 (porque 5 dividido por 2 é igual a 2 e resta 1). Por outro lado, 6%2=0, porque 6 dividido por 2 é igual a 3 e resta 0.

Como é em algumas linguagens

Português Estruturado

```
inicio
    inteiro idade
    escreva(“Digite a sua idade:”)
    leia(idade)
    se(idade>=18) então
        escreva(“Você pode dirigir!”)
        escreva(“Que legal!”)
    senão
        escreva(“Você não pode dirigir!”)
        escreva(“Que pena!”)
    fimse
    escreva(“Tchau!”)
fim
```

C

```
#include<stdio.h>

int main(){
    int idade;
    printf("Digite a sua idade: ");
    scanf("%d",&idade);
    if(idade>=18){
        printf("Você pode dirigir!\n");
        printf("Que legal!\n");
    }else{
        printf("Você não pode dirigir!\n");
        printf("Que pena!\n");
    }
    printf("Tchau!\n");
    return 0;
}
```

Obs: a condição a ser testada dentro do if precisa estar entre parênteses, como no nosso Português Estruturado. O abre e fecha chaves ({}) representa o bloco da condição if ou else (mais ou menos como o nosso então e fimse). Deste modo, na condição if, tudo o que estiver entre chaves deverá ser executado quando a condição for verdadeira, quando. E no else, tudo o que estiver entre chaves será executado caso a condição if for falsa. Caso o bloco if ou else tenha apenas uma instrução, não é necessário usar as chaves.

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    int idade;
    cout << "Digite a sua idade: ";
    cin >> idade;
    if(idade>=18){
        cout << "Você pode dirigir!\n";
        cout << "Que legal!\n";
    }else{
        cout << "Você não pode dirigir!\n";
        cout << "Que pena!\n";
    }
    cout << "Tchau!\n";
    return 0;
}
```

Obs: o mesmo que na linguagem C: a condição a ser testada dentro do if precisa estar entre parênteses, como no nosso Português Estruturado. O abre e fecha chaves ({}) representa o bloco da condição if ou else (mais ou menos como o nosso então e fimse). Deste modo, na condição if, tudo o que estiver entre chaves deverá ser executado quando a condição for verdadeira, quando. E no else, tudo o que estiver entre chaves será executado caso a condição if for falsa. Caso o bloco if ou else tenha apenas uma instrução, não é necessário usar as chaves.

Go

```
package main

import (
    "fmt"
)

func main() {
    var idade int

    fmt.Print("Digite a sua idade: ")
    fmt.Scanf("%d", &idade)
    if idade >= 18 {
        fmt.Println("Você pode dirigir!")
        fmt.Println("Que legal!")
    } else {
        fmt.Println("Você não pode dirigir!")
        fmt.Println("Que pena!")
    }
    fmt.Println("Tchau!")
}
```

Obs: muito parecido com C/C++, mas sem necessidade de parênteses (mas pode por, se quiser ou se o editor permitir). O abre e fecha chaves ({}) representa o bloco da condição if ou else (mais ou menos como o nosso então e fimse). Deste modo, na condição if, tudo o que estiver entre chaves deverá ser executado quando a condição for verdadeira, quando. E no else, tudo o que estiver entre chaves será executado caso a condição if for falsa. Diferente de C/C++, sempre há necessidade do uso das chaves, mesmo que o bloco tenha apenas uma linha.

Ruby

```
print "Digite a sua idade: "
idade = gets.to_i
if idade >= 18
    puts "Você pode dirigir!"
    puts "Que legal!"
else
    puts "Você não pode dirigir!"
    puts "Que pena!"
end
puts "Tchau!"
```

Obs: não há necessidade de parênteses (mas pode ser usado, caso queira). Não há símbolo de início do bloco. O símbolo do fim do bloco será o else (caso tenha) ou end.

Python

```
idade = int(input('Digite sua idade: '))
if idade >= 18:
    print('Você pode dirigir!')
    print('Que legal!')
else:
    print('Você não pode dirigir!')
    print('Que pena!')
print('Tchau!')
```

Obs: não há necessidade de parênteses (mas pode ser usado, caso queira). O símbolo de dois-pontos (:) deve ser usado como marcador de início do bloco tanto no **if** quanto no **else**. O que define o fim do bloco é a indentação. Na maioria das linguagens, a indentação serve apenas como recurso para facilitar a leitura. Se o programador não quiser indentar o código, ele ficará apenas muito ruim de ser lido e feita a manutenção. Mas, em Python, a indentação é um recurso usado também para o compilador identificar começo e fim de blocos de instruções.

Estrutura de Controle de Decisão se-senão encadeado

Uma vez que aprendemos a instrução ***se-senão***, podemos avançar um pouco em seu uso. Repare que até o momento, vimos exemplos de problemas que se resumem ao máximo a duas possibilidades: se uma condição for verdadeira, então algo é realizado; caso seja falsa, então outra tarefa é realizada.

Porém, muitos outros problemas possuem mais de duas possibilidades. Por exemplo, imagine um algoritmo que peça a nota de um aluno e apresente uma mensagem de acordo com as seguintes condições:

1. Se o aluno tirou nota 6 ou superior, então está aprovado;
2. Senão, se o aluno tirou nota inferior a 4, então, está automaticamente reprovado;
3. Senão, se o aluno tirou nota maior ou igual a 4 e menor que 6, está de recuperação.

Perceba que a última opção sempre ocorrerá se as opções 1 e 2 não ocorrem, porque se a pessoa não tirou nota igual ou superior a 6, nem inferior a 4, então só pode ter tirado nota entre 4 e 6. As mesmas condições podem ser reescritas da seguinte forma:

1. Se o aluno tirou nota 6 ou superior, então está aprovado;
2. Senão, se o aluno tirou nota inferior a 4, então, está automaticamente reprovado;
3. Senão, está de recuperação.

O algoritmo para o problema proposto é apresentado a seguir:

```
1.    inicio
2.        real nota
3.        escreva(“Digite a sua nota:”)
4.        leia(nota)
5.        se(nota>=6) então
6.            escreva(“Aprovado!”)
7.        senão se(nota<4) então
8.            escreva(“Reprovado!”)
9.        senão
```

```
10.            escreva(“Recuperação!”)
11.        fimse
12.        escreva(“Tchau!”)
13.    fim
```

Agora é hora de exercitarmos o uso de ***se-senão*** encadeado.

Exercícios de se-senão Encadeado

1. Faça um algoritmo que peça um número inteiro. Em seguida, o algoritmo deve apresentar uma mensagem conforme a seguinte regra:
    a. Se o valor for positivo: “Valor positivo”;
    b. Se o valor for negativo: “Valor negativo”;
    c. Se não for nenhuma das opções (se for 0): “Valor neutro”.

2. Faça um algoritmo que peça o salário do usuário e apresente uma mensagem de qual classe social você pertence de acordo com a faixa salarial:
    a. Se ganha mais de R$ 18.000: “você é classe A”;
    b. Se ganha mais de R$ 9.000: “você é classe B”;
    c. Se não for nenhuma das anteriores (se ganha menos de R$ 9.000): “você é classe C ou menor”.

3. Faça um algoritmo que peça a temperatura da água de um recipiente, em Célsius. De acordo com a temperatura, o algoritmo deve apresentar a seguinte mensagem:
    a. Se for superior a 100 ºC: “A água está em estado de vapor”;
    b. Se for superior a 0 ºC: “A água está em estado líquido”;
    c. Se não for nenhuma das anteriores, “A água está em estado sólido”.

4. Faça um algoritmo que peça a idade do usuário e apresente uma mensagem de acordo com sua faixa etária:
    a. Se for maior ou igual a 65 anos: “você está na melhor idade”;
    b. Se for maior ou igual a 40 anos (e menor que 65): “você está na meia idade”;
    c. Se for maior ou igual a 30 anos (e menor que 40): “você é adulto”;
    d. Se for maior ou igual a 18 anos (e menor que 30): “você é adulto jovem”;
    e. Se for maior ou igual a 12 anos (e menor que 18): “você é adolescente”;
    f. Se for maior ou igual a 2 anos (e menor que 12): “você é criança”;
    g. Se não for nenhuma das anteriores (menor de 2 anos): “você é bebê”.

5. Faça um algoritmo que peça a altura (em metros) e a sua massa (em quilos). Em seguida, deve-se calcular o Índice de Massa Corporal, da seguinte forma:

   IMC=massa/(altura * altura)

   De acordo com o resultado, o algoritmo deve apresentar uma mensagem, como abaixo:

   a. Abaixo de 17: “muito abaixo do peso”;
   b. Abaixo de 18.5 (e maior ou igual a 17): “abaixo do peso”;
   c. Abaixo de 25 (e maior ou igual a 18.5): “peso normal”;
   d. Abaixo de 30 (e maior ou igual a 25): “acima do peso”;
   e. Abaixo de 35 (e maior ou igual a 30): “obesidade I”;
   f. Abaixo de 40 (e maior ou igual a 35): “obesidade II (severa)”;
   g. Se não for nenhuma das opções (igual ou maior que 40): “obesidade III (mórbida)”.

Como é em algumas linguagens

Português Estruturado

```
inicio
   real nota
   escreva(“Digite a sua nota:”)
   leia(nota)
   se(nota>=6) então
      escreva(“Aprovado!”)
   senão se(nota<4) então
      escreva(“Reprovado!”)
   senão
      escreva(“Recuperação!”)
   fimse
   escreva(“Tchau!”)
fim
```

C

```
#include<stdio.h>

int main(){
    float nota;
    printf("\nDigite a sua nota: ");
    scanf("%f",&nota);
    if(nota>=6){
        printf("\nAprovado!");
    }else if(nota<4){
        printf("\nReprovado!");
    }else{
        printf("\nRecuperação!");
    }
    printf("\nTchau!");
    return 0;
}
```

Obs: neste exemplo, os blocos IF e ELSE não precisam de chaves porque quando há apenas uma instrução dentro do bloco, não há necessidade delas. Mas podem ser colocadas, caso o programador prefira.

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    float nota;
    cout << "\nDigite a sua nota: ";
    cin >> nota;
    if(nota>=6){
        cout << "\nAprovado!";
    }else if(nota<4){
        cout << "\nReprovado!";
    }else{
        cout << "\nRecuperação!";
    }
    cout << "\nTchau!";
    return 0;
}
```

Obs: neste exemplo, os blocos IF e ELSE não precisam de chaves porque quando há apenas uma instrução dentro do bloco, não há necessidade delas. Mas podem ser colocadas, caso o programador prefira.

Go

```
package main

import (
    "fmt"
)

func main() {
    var nota float32
    fmt.Print("Digite a sua nota: ")
    fmt.Scanf("%f", &nota)
    if nota >= 6 {
        fmt.Println("Aprovado!")
    } else if nota < 4 {
        fmt.Println("Reprovado!")
    } else {
        fmt.Println("Recuperação!")
    }
    fmt.Println("Tchau!")
}
```

Obs: muito parecido com C/C++, mas sem necessidade de parênteses (mas pode por, se quiser ou se o editor permitir).

Ruby

```
print "Digite a sua nota: "
nota = gets.to_f
if nota >= 6
    puts "Aprovado!"
elsif nota<4
    puts "Reprovado!"
else
    puts "Recuperação!"
end
puts "Tchau!"
```

Obs: o else seguido de if é uma instrução chamada elsif.

Python

```
nota = float(input('Digite a sua nota: '))
if nota >= 6:
    print('Aprovado!')
elif nota<4:
    print('Reprovado!')
else:
    print('Recuperação!')
print('Tchau!')
```

Obs: o else seguido de if é uma instrução chamada elif.

Estrutura de Controle de Decisão Escolha-Caso

Em todos os exemplos que vimos de ***se-senão*** encadeado, a variável a ser comparada pode assumir diversos intervalos de valores. Mas vamos supor que queiramos criar um algoritmo em que os valores a serem comparados devam ser específicos, como no enunciado abaixo:

1- O algoritmo deve pedir dois valores reais;
2- Em seguida, o algoritmo deve apresentar o seguinte menu:
    a. "Digite 1 para somar";
    b. "Digite 2 para subtrair";
    c. "Digite 3 para multiplicar";
    d. "Digite 4 para dividir".
3- Uma vez digitado um valor de 1 a 4, o algoritmo deverá calcular o resultado e apresentá-lo na tela.

Este algoritmo é facilmente elaborado com o que já conhecemos. Primeiro devemos pensar quantas variáveis serão necessárias: (i) uma variável para o primeiro valor real; (ii) uma variável para o segundo valor real; (iii) uma variável para armazenar o resultado da conta; e, (iv) uma variável para armazenar a opção de 1 a 4.

Em seguida, pensamos em dividir o problema em partes para iniciarmos a escrita do algoritmo: declarar as variáveis; pedir e ler os dois valores reais; em seguida, devemos nos preocupar em pedir e ler a opção do menu; por fim, dependendo da opção, realizar a respectiva operação aritmética e apresentar o resultado:

```
1. inicio
2.    real operando1, operando2, resultado
3.    int opcao
4.    escreva("Digite o valor do 1º operando:")
5.    leia(operando1)
6.    escreva("Digite o valor do 2º operando:")
7.    leia(operando2)
8.    escreva("Digite 1 para somar")
9.    escreva("Digite 2 para subtrair")
10.   escreva("Digite 3 para multiplicar")
11.   escreva("Digite 4 para dividir")
```

```
12.   leia(opcao)
13.   se(opcao==1) então
14.       resultado <- operando1 + operando2
15.   senão se(opcao==2) então
16.       resultado <- operando1 – operando2
17.   senão se (opcao==3) então
18.       resultado <- operando1 * operando2
19.   senão se(opcao==4) então
20.       resultado <- operando1 / operando2
21.   senão
22.       escreva(“Opção inválida.Resultado será 0”)
23.       resultado<-0
24.   fimse
25.   escreva(“O resultado é ”, resultado)
26. fim
```

Este algoritmo está correto. A variável opcao foi usada nas comparações de *se-senão* encadeados. Perfeito. Mas há uma característica diferente neste exemplo em relação aos anteriores: aqui, a variável opcao deve receber apenas valores inteiros específicos, diferente de outros exemplos, em que a comparação era feita em intervalos de valores (como maior que 65). E todas as condições usam o operador de igualdade.

Quando o valor da variável só pode assumir determinados valores inteiros específicos e as comparações são sempre de igualdade (como no nosso exemplo em que só deve receber um valor entre 1, 2 3 ou 4), é possível usar uma outra instrução, chamada *escolha-caso*. Esta instrução é, também, bastante intuitiva e o código fica mais fácil de ler. O mesmo algoritmo, pode ser escrito da seguinte forma:

```
1.  inicio
2.     real operando1, operando2, resultado
3.     int opcao
4.     escreva(“Digite o valor do primeiro operando:”)
5.     leia(operando1)
6.     escreva(“Digite o valor do segundo operando:”)
7.     leia(operando2)
8.     escreva(“Digite 1 para somar”)
9.     escreva(“Digite 2 para subtrair”)
10.    escreva(“Digite 3 para multiplicar”)
11.    escreva(“Digite 4 para dividir”)
12.    leia(opcao)
13.    escolha (opcao)
14.       caso 1:
15.           resultado <- operando1 + operando2
16.       caso 2:
17.           resultado <- operando1 – operando2
18.       caso 3:
```

```
19.           resultado <- operando1 * operando2
20.       caso 4:
21.           resultado <- operando1 / operando2
22.       outrocaso:
23.           escreva("Opção inválida.")
24.           escreva("O resultado será 0")
25.           Resultado <- 0
26.    fimescolha
27.    escreva("O resultado é ", resultado)
28. fim
```

A lógica é a mesma de se usar ***se-senão*** encadeado. A única diferença é sua facilidade de leitura e o fato da instrução só poder ser usada quando o valor da variável a ser comparada for específico.

Há, ainda, situações em que o algoritmo deve se comportar da mesma forma, **caso** o valor da variável seja igual. Por exemplo, imagine um algoritmo que peça o dia da semana (1 representa domingo, 2 representa segunda-feira, 3 representa terça-feira e assim por diante). Em seguida, o algoritmo deve escrever se o dia digitado é fim de semana ou dia útil.

Neste exemplo, o algoritmo também pode ser escrito usando-se ***escolha-caso***, como a seguir:

```
1. inicio
2.     int dia
3.     escreva("Digite o dia da semana:")
4.     escreva("1 é domingo")
5.     escreva("2 é segunda-feira")
6.     escreva("3 é terça-feira")
7.     escreva("E assim por diante")
8.     Leia(dia)


9.      escolha (dia)
10.         caso 1, 7:
11.             Escreva("É fim de semana!")
12.         caso 2, 3, 4, 5, 6:
13.             Escreva("É dia útil")
14.         caso padrão:
15.             escreva("Dia inválido.")
16.             escreva("A semana só tem 7 dias!")
17.     fimescolha
18. fim
```

Repare que no caso do usuário ter digitado 1 ou 7 (linha 10), então a linha 11 será executada. Caso o usuário digite 2, 3, 4, 5 ou 6 (linha 12), a linha 13 será executada. Em quaisquer outros casos, as linhas 15 e 16 serão executadas.

Há linguagens de programação que permitem que esta instrução seja usada também com caracteres e cadeia de caracteres. Aqui, vamos considerar todas as possibilidades.

Exercícios de escolha-caso

Para a lista abaixo, use apenas a instrução ***escolha-caso***.

1. Faça um algoritmo de Monstrodex que apresente 3 opções:
    - 1-Zikachu;
    - 2-Zulbassauro;
    - 3-Zharmander.

    Caso o usuário escolha a opção 1, o algoritmo deve apresentar a mensagem "Monstrinho elétrico da categoria rato".

    Caso o usuário escolha a opção 2, o algoritmo deve apresentar a mensagem "Monstrinho de grama da categoria semente".

    Caso o usuário escolha a opção 3, o algoritmo deve apresentar a mensagem "Monstrinho de fogo da categoria lagarto".

    Caso o usuário escolha outra opção, o algoritmo deve apresentar a mensagem "Monstrinho não cadastrado. Há 8900 monstros! Temos que pegar!"

2. Faça um algoritmo que apresente as seguintes opções de menu:
    - A- Avião
    - B- Carro
    - C- Cruzeiro

    Caso o usuário digite A, o algoritmo deve apresentar a mensagem "É mais rápido!"

    Caso o usuário digite B, o algoritmo deve apresentar a mensagem "É mais barato!"

    Caso o usuário digite C, o algoritmo deve apresentar a mensagem "É mais bonito!"

3. Faça um algoritmo que apresente o seguinte menu:
    - 1-Calcular a área de um retângulo;
    - 2-Calcular a área de um círculo;

- 3-Calcular a área de um triângulo.

Conforme a opção, o algoritmo deverá pedir os respectivos valores (altura e largura para retângulo, altura e base para o triângulo e raio para círculo). As fórmulas para os respectivos cálculos são:

- Área_retângulo = altura * base
- Área_círculo = PI * raio^2
- Área_triângulo = base * altura / 2

Por fim, deve ser apresentado o resultado.

4. Faça um algoritmo que peça o mês do ano (de 1 a 12). O algoritmo deve apresentar a mensagem de quantos dias o mês tem, ou “mês inválido”, caso digite um mês inválido.

Como é em algumas Linguagens

Português Estruturado

```
inicio
    real operando1, operando2, resultado
    int opcao
    escreva(“Digite o valor do primeiro operando:”)
    leia(operando1)
    escreva(“Digite o valor do segundo operando:”)
    leia(operando2)
    escreva(“"1-somar; 2-subtrair; 3-multiplicar; 4-dividir:")
    leia(opcao)

    escolha (opcao)
        caso 1:
            resultado <- operando1 + operando2
        caso 2:
            resultado <- operando1 – operando2
        caso 3:
            resultado <- operando1 * operando2
        caso 4:
            resultado <- operando1 / operando2
        caso 5, 6, 7:
             escreva(“Será disponibilizado futuramente”)
             resultado <- 0
        caso padrão:
            escreva(“Opção inválida. O resultado será 0”)
            resultado <- 0
    fimescolha
    escreva(“O resultado é ”, resultado)
fim
```

C

```
#include<stdio.h>

int main(){
    float num1, num2, res;
    int opc;

    printf("\nDigite o valor do primeiro operador: ");
    scanf("%f",&num1);
    printf("\nDigite o valor do segundo operador: ");
    scanf("%f",&num2);

    printf("\n1-somar; 2-subtrair; 3-multiplicar; 4-dividir:");
    scanf("%d",&opc);

    switch(opc){
        case 1:
            res = num1 + num2;
            break;
        case 2:
            res = num1 - num2;
            break;
        case 3:
            res = num1 * num2;
            break;
        case 4:
            res = num1 / num2;
            break;
        case 5, 6, 7:
            printf("\nSerá disponibilizado futuramente");
            res = 0;
            break;
        default:
            printf("\nOpção inválida! O resultado será 0");
            res = 0;
            break;
    }
    printf("\nO resultado é %.2f", res);
    return 0;
}
```

Obs: repare que ao fim de cada bloco CASE, há a necessidade de se usar a instrução **break**. Ela termina o bloco para que o programa continue sua execução a partir do fim da instrução SWITCH-CASE.

Suponha que, neste exemplo, o usuário digitou a opção 1. Então, entrará no **case 1:** e executará as suas respectivas instruções. Assim que o break for executado, ele pulará o resto da instrução SWITCH-CASE, e executará a instrução **printf** que apresentará o resultado.

Se o programador esquecer de colocar aquele break, então, o algoritmo executaria o que está no **case 1:**, e continuaria executando o bloco do **case 2:** (mesmo que tenha digitado 1) até encontrar um break e pular para o fim do bloco **switch.** Se neste momento você está com um "ué" em sua cabeça, é compreensível. Afinal, quando usamos IF-ELSE, não se usa break no fim de um bloco IF para que se pule os demais ELSE que possam existir. Mas é assim que funciona o SWITCH-CASE do C/C++.

Se mais de um valor executar o mesmo bloco, estes podem ser separados por vírgulas. Então, se o usuário digitar 5, 6 ou 7, o mesmo bloco será executado.

O caso **default** será executado se nenhum outro caso for verdadeiro.

C/C++ permitem o uso de char dentro de um SWITCH-CASE, como abaixo:

```
  ...
  char c;
  printf("\nDigite um caractere:");
 scanf("%c",&c);
    switch(c){
        case 'a':
            printf("\nVocê digitou a");
            break;
        case 'b':
            printf("\nVocê digitou a");
            break;
    }
...
```

Temos também algo diferente no último **printf**. O **.2** entre **%** e **f** indica que a saída deve apresentar duas casas decimais.

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>
using namespace std;

int main(){
    float num1, num2, res;
    int opc;

    cout << "\nDigite o valor do primeiro operador: ";
    cin >> num1;
    cout << "\nDigite o valor do segundo operador: ";
    cin >> num2;

    cout << "1-somar; 2-subtrair; 3-multiplicar; 4-dividir:";
    cin >> opc;
    switch(opc){
        case 1:
            res = num1 + num2;
            break;
        case 2:
            res = num1 - num2;
            break;
        case 3:
            res = num1 * num2;
            break;
        case 4:
            res = num1 / num2;
            break;
        case 5, 6, 7:
            cout << “\nSerá disponibilizado futuramente”;
            res = 0;
            break;
        default:
            cout << "Opção inválida! O resultado será 0";
            res = 0;
            break;
    }
    cout << "\nO resultado é " << res;
    return 0;
}
```

Obs: as mesmas observações do C++.

Go

```
package main

import (
    "fmt"
)

func main() {
       var num1, num2, res float32
    var opc int

    fmt.Print("Digite o valor do primeiro operador: ")
    fmt.Scanf("%f\n", &num1)
    fmt.Print("Digite o valor do segundo operador: ")
    fmt.Scanf("%f\n", &num2)

    fmt.Print("1-somar; 2-subtrair; 3-multiplicar; 4-dividir:")
    fmt.Scanf("%d", &opc)
    switch opc {
    case 1:
        res = num1 + num2
    case 2:
        res = num1 - num2
    case 3:
        res = num1 * num2
    case 4:
        res = num1 / num2
    case 5, 6, 7:
        fmt.Println(“Será disponibilizado futuramente”)
        res = 0
    default:
        fmt.Println("Opção inválida! O resultado será 0")
        res = 0
    }
    fmt.Println("\nO resultado é ", res)
}
```

Obs: muito parecido com C/C++, mas sem necessidade de parênteses (mas pode por, se quiser ou se o editor permitir) e não há **break**.

Se mais de um valor executar o mesmo bloco, estes podem ser separados por vírgulas. Então, se o usuário digitar 5, 6 ou 7, o mesmo bloco será executado.

O caso **default** será executado se nenhum outro caso for verdadeiro.

A instrução switch em GO é mais versátil que em C e C++. Por exemplo, pode ser usada sem uma condição lógica, como alternativa a IF-ELSE IF:

```
    ...
    var idade int
    fmt.Print("Digite sua idade")
    fmt.Scanf("%d", &idade)
```

```
    switch {
    case idade >= 65:
        fmt.Println("Terceira idade")
    case idade < 18:
        fmt.Println("Menor de idade")
    default:
        fmt.Println("Adulto")
    }
```

Também é possível usar string variáveis string:

```
    ...
    var nome string
    fmt.Print("Digite seu nome:")
    fmt.Scanf("%s", &nome)
    switch nome {
    case "Nuvem":
        fmt.Println("Nome legal")
    case "Rex":
        fmt.Println("Nome comum")
    default:
        fmt.Println("Ok.")
    }
   ...
```

Ruby

```
print "Digite o valor do primeiro operador: "
num1 = gets.to_f
print "Digite o valor do segundo operador: "
num2 = gets.to_f
print "1-somar; 2-subtrair; 3-multiplicar; 4-dividir:"
opc = gets.to_i

case opc
when 1
    res = num1 + num2
when 2
    res = num1 - num2
when 3
    res = num1 * num2
when 4
    res = num1 / num2
when 5..7
    puts “Será disponibilizado futuramente”
    res = 0
else
    puts "Opção inválida! O resultado será 0"
    res = 0
end
print "O resultado é ", res
```

Obs: O **case** é nosso escolha e o **when** é nosso caso. Não há dois-pontos (:) ao fim da instrução when.

Se mais de um valor executar o mesmo bloco, estes podem ser separados por vírgulas. Também é possível representar um intervalo de valores. Por exemplo, em vez de separar 5, 6 e 7 por vírgulas, pode ser usado 5..7, que representa um intervalo de valores entre 5 e 7.

Como a linguagem define o tipo da variável durante a atribuição de um valor, a instrução case-when permite a comparação com diferentes tipos, inclusive no mesmo bloco.

```
case opc
when 1
    print("Isso é um número")
when "Nuvem"
    printf("Isso é um nome da minha Shi tzu")
end
```

Python

```
num1 = float(input('Digite o valor do primeiro operador: '))
num2 = float(input('Digite o valor do segundo operador: '))
opc = int(input('1-somar; 2-subtrair; 3-multiplicar; 4-dividir:'))

match opc:
    case 1:
        res = num1 + num2
    case 2:
        res = num1 - num2
    case 3:
        res = num1 * num2
    case 4:
        res = num1 / num2
    case 5 | 6 | 7:
        print('Será disponibilizado futuramente')
        res = 0
    case _:
        print('Opção inválida! O resultado será 0')
        res = 0
print('O resultado é ', res)
```

Obs: o **match** é nosso escolha e o **case** é nosso caso.

Se mais de um valor executar o mesmo bloco, estes podem ser separados por pipeline (|).

O caso **_** será executado se nenhum outro caso for verdadeiro.

Como a linguagem define o tipo da variável durante a atribuição de um valor, a instrução match-case permite a comparação com diferentes tipos, inclusive no mesmo bloco.

```
match opc:
    case 1:
        print('Isso é um número')
    case 'Nuvem':
        print('Esse é o nome da minha Shi tzu')
```

Operadores Lógicos e Condições Compostas

Até o momento, para cada algoritmo, apenas uma condição foi utilizada por vez nas instruções ***se-senão***. Agora, podemos nos aprofundar em problemas que exijam a verificação de mais de uma condição ao mesmo tempo, também chamada de **condição composta**.

Como exemplo, imagine que em um determinado país, os homens podem se aposentar se tiverem trabalhado pelo menos 30 anos. E, no caso das mulheres, elas podem se aposentar se tiverem trabalhado pelo menos 25 anos.

Deste modo, a aposentadoria só acontece:

- **SE** for homem **e** tiver pelo menos 30 anos de trabalho;
- **SE** for mulher **e** tiver pelo menos 20 anos de trabalho.

Perceba que é necessário verificar duas condições: o gênero e o tempo de trabalho.

Para tanto, podemos utilizar os operadores lógicos **E** e **OU**. Na nossa pseudolinguagem utilizaremos **&&** para E, e **||** para ou. Usamos o operador && quando todas as condições precisarem ser verdadeiras. Usamos o operador || quando apenas uma das condições precisar ser verdadeira.

O algoritmo para o problema anterior é:

```
1.  inicio
2.     caractere genero
3.     int tempo
4.     escreva("Digite o seu gênero (m ou f):")
5.     leia(genero)
6.     escreva("Digite o seu tempo de trabalho:")
7.     leia(tempo)
8.     se (genero=='m' && tempo>=30) então
9.         escreva("Você pode se aposentar!")
10.    senão se (genero=='f' && tempo>=25) então
11.        escreva("Você pode se aposentar!")
12.    senão
13.        escreva("Você não pode se aposentar!")
14.    fimse
15. fim
```

É um algoritmo bastante intuitivo. Repare na linha 8 que o usuário só poderá se aposentar se as duas condições forem verdadeiras, por isso foi usado o operador lógico &&. O mesmo para a linha 10.

Como exemplo de operador lógico ||, podemos imaginar um problema para verificar se o usuário pode pedir auxílio-moradia. A regra é: se o usuário morar a mais de 100km da Universidade ou se possuir renda inferior a R$ 1000.00. Neste caso, basta que apenas uma das condições seja verdadeira. Um algoritmo para o problema é descrito a seguir:

```
1.    inicio
2.      real distancia, renda
3.      escreva("A que distância da Universidade você mora?")
4.      leia(distancia)
5.      escreva("Qual sua renda?")
6.      leia(renda)
7.      se (renda<1000 || distancia>100) então
8.          escreva("Você pode pedir auxílio-moradia!")
9.      senão
10.         escreva("você não pode pedir auxílio-moradia!")
11.     fimse
12.   fim
```

O uso do operador lógico || também é bastante intuitivo.

É interessante notar a seguinte situação no exemplo anterior: se o usuário não pode pedir moradia, é porque ele tem renda igual ou superior a R$ 1000 e, ao mesmo tempo, mora a uma distância igual ou inferior a 100 km. Percebeu? Isso porque se a renda fosse inferior a R$1000, mesmo morando perto, ele teria direito de pedir moradia. E, da mesma forma, se tivesse renda alta, mas morasse a mais de 100 km, também poderia pedir moradia. Então, não seria errado criar um algoritmo como segue:

```
1.   inicio
2.     real distancia, renda
3.     escreva("A que distância da Universidade…")
4.     escreva("você mora?")
5.     leia(distancia)
6.     escreva("Qual sua renda?")
7.     leia(renda)
```

```
8.     se (renda>=1000 && distancia<=100) então
9.         escreva(“Você não pode pedir moradia!”)
10.    senão
11.        escreva(“você pode pedir moradia!”)
12.    fimse
13.  fim
```

Ambas as soluções estão corretas! Então, por qual optar? Tanto faz. Mas, por enquanto, o ideal é usar a lógica que siga o mais próximo o enunciado do problema. No nosso caso, como o enunciado descrevia as condições que deveriam ocorrer para pedir moradia, então é interessante traduzirmos diretamente as condições apresentadas: se o usuário morar a mais de 100km da Universidade **OU** se possuir renda inferior a R$ 1000.00.

Há, também, o operador de negação ! (o símbolo que usamos para negar uma expressão lógica é uma exclamação) que nega (ou inverte) o significado de uma expressão.

Deste modo, a expressão:

```
se(renda>1000)então
```

Também poderia ser escrita como:

```
se(!(renda<=1000))então
```

Isso porque dizer “renda é maior que 1000” é o mesmo que dizer “renda não é menor ou igual a 1000”. Mas nota-se que é mais difícil pensar de forma negacionista.

Assim como na Matemática, há uma relação de precedência entre os operadores:

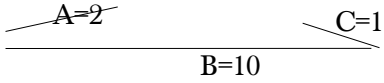
- A negação tem precedência sobre os operadores && e ||
- && tem precedência sobre ||

Hora de praticar.

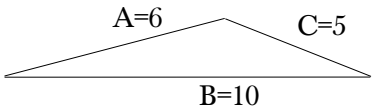
Exercícios de Operadores Lógicos e Condições Compostas

1. Faça um algoritmo que peça para o usuário digitar um número entre 1 e 6 (inclusive). Se o valor digitado estiver no intervalo, apresentar a mensagem "Valor digitado com sucesso". Do contrário, apresentar a mensagem "Valor fora do intervalo permitido".
2. Faça um algoritmo que peça para o usuário digitar um número inferior a 1 ou superior a 6. Se o valor digitado estiver correto, apresentar a mensagem "Valor digitado com sucesso". Do contrário, apresentar a mensagem "Valor não pode estar entre 1 e 6".
3. Faça um algoritmo que peça a idade do usuário. Se ele tiver entre 18 e 65 anos, apresentar a mensagem: "Você é obrigado a votar". Do contrário, apresentar a mensagem: "você não é obrigado a votar".
4. Faça um algoritmo que peça o usuário e senha. Se o usuário for "chefe" e a senha for "123456", então apresentar a mensagem "login realizado". Do contrário, apresentar a mensagem "usuário e/ou senha incorretos".
5. Imagine que em um determinado país, os homens podem se aposentar se tiverem trabalhado pelo menos 30 anos e tenham mais de 65 anos. E, no caso das mulheres, elas podem se aposentar se tiverem trabalhado pelo menos 25 anos e tenham mais de 60 anos. Reparem que foi usado **E** no enunciado, então, deve-se seguir a risca o que foi pedido. Faça um algoritmo que peça o gênero, a idade e tempo de contribuição. Em seguida, o algoritmo deve apresentar se a pessoa pode ou não se aposentar.
6. Faça um algoritmo que peça três lados de um triângulo (pode ser apenas números inteiros). O programa deve apresentar se os lados apresentados formam ou não um triângulo. As mensagens devem ser: "é um triângulo" ou "não é um triângulo". Para saber se é possível criar um triângulo, a regra é: cada lado precisa ser menor que a soma dos outros dois lados (do contrário, o triângulo não fecha). Veja a explicação ilustrada abaixo:

   Repare que o lado B do desenho não respeita a regra, porque B > A + C. Por isso, não tem como o triângulo ser fechado.



Mas no desenho a seguir, é possível montar um triângulo:



Isso porque:

- A < B + C
- B < A + C
- C < A + B

Como é em algumas linguagens

Português Estruturado

```
início
    real nota, frequencia

    escreva("Digite sua nota:")
    leia(nota)
    escreva("Digite sua frequência:")
    leia(frequencia)

    //Uma forma de verificar se foi aprovado
    SE(nota>=6 && frequencia>=75)ENTÃO
        escreva("Aprovado")
    SENÃO
        escreva("Reprovado")
    FIMSE

    //Outra forma de se verificar a mesma coisa
    SE(!(nota<6 || frequencia<75))ENTÃO
        escreva("Aprovado")
    SENÃO
        escreva("Reprovado")
    FIMSE
fim
```

Obs: repare que é importante usarmos os parênteses no segundo exemplo de SE para que primeiro seja feita a operação OU para depois ser negado o resultado. Isso porque queremos dizer: "se a nota NÃO for menor que 6 OU se a frequência NÃO for menor que 75...". Também poderia ser possível escrever a condição como abaixo:

```
SE(!(nota<6) || !(frequencia<75))ENTÃO
```

C

```c
#include<stdio.h>

int main(){
    float nota, frequencia;

    printf("\nDigite sua nota: ");
    scanf("%f",&nota);
    printf("\nDigite sua frequência: ");
    scanf("%f",&frequencia);

    //Uma forma de verificar se foi aprovado
    if(nota>=6 && frequencia>=75){
        printf("\nAprovado!");
    }else{
        printf("\nReprovado!");
    }

    //Outra forma de se verificar a mesma coisa
    if(!(nota<6 || frequencia<75)){
        printf("\nAprovado!");
    }else{
        printf("\nReprovado!");
    }
}
```

Obs: Em C/C++, usamos os mesmos símbolos do nosso Português Estruturado para expressar negação, E e OU (respectivamente !, && e ||).

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    float nota, frequencia;

    cout << "\nDigite sua nota: ";
    cin >> nota;
    cout << "\nDigite sua frequência: ";
    cin >> frequencia;

    //Uma forma de verificar se foi aprovado
    if(nota>=6 && frequencia>=75){
        cout << "\nAprovado!";
    }else{
        cout << "\nReprovado!";
    }

    //Outra forma de se verificar a mesma coisa
    if(!(nota<6 || frequencia<75)){
        cout << "\nAprovado!";
    }else{
        cout << "\nReprovado!";
    }
}
```

Obs: Em C/C++, usamos os mesmos símbolos do nosso Português Estruturado para expressar negação, E e OU (respectivamente !, && e ||).

Go

```
package main

import "fmt"

func main() {
    var nota, frequencia float32

    fmt.Print("Digite sua nota:")
    fmt.Scanf("%f", &nota)
    fmt.Print("Digite sua frequência:")
    fmt.Scanf("\n%f", &frequencia)

    //Uma forma de verificar se foi aprovado
    if nota >= 6 && frequencia >= 75 {
        fmt.Println("Aprovado!")
    } else {
        fmt.Println("Reprovado!")
    }

    //Outra forma de se verificar a mesma coisa
    if !(nota < 6 || frequencia < 75) {
        fmt.Println("Aprovado!")
    } else {
        fmt.Println("Reprovado!")
    }
}
```

Obs: assim como em C/C++, usamos os mesmos símbolos do nosso Português Estruturado para expressar negação, E e OU (respectivamente !, && e ||).

Ruby

```
print "Digite a nota: "
nota = gets.to_f
print "Digite a frequência: "
frequencia = gets.to_f

#Uma forma de verificar se foi aprovado
if nota>=6 and frequencia>=75
    puts "Aprovado!"
else
    puts "Reprovado!"
end

#Outra forma de se verificar a mesma coisa
if not(nota<6 or frequencia<75)
    puts "Aprovado!"
else
    puts "Reprovado!"
end
```

Obs: em Ruby, podemos usar os operadores **not**, **and** e **or**; ou os mesmos símbolos de C/C++.

Python

```
nota = float(input('Digite a nota: '))
frequencia = float(input('Digite a frequencia: '))

#Uma forma de verificar se foi aprovado
if nota>=6 and frequencia>=75:
    print('Aprovado!')
else:
    print('Reprovado!')

#Outra forma de se verificar a mesma coisa
if not(nota<6 or frequencia<75):
    print('Aprovado!')
else:
    print('Reprovado!')
```

Obs: em Python, os símbolos de **negação**, **e** e **ou** são **not**, **and** e **or**.

Se-Senão Aninhado

Até o momento, para cada algoritmo, as condições ***se-senão*** aprendidas eram simples ou compostas por mais de uma variável.

Agora, podemos nos aprofundar em problemas que exijam a verificação de variáveis que possuam relação de dependência. Vamos relembrar o exercício anterior em que o usuário deveria digitar sua nota e o algoritmo indicaria se o aluno foi aprovado, reprovado ou se ficaria de recuperação. Repare que a única variável a ser verificada era a nota.

Imagine que o problema deva levar em consideração, também, a frequência do aluno. Se o aluno tiver frequência igual ou superior a 75%, então deve-se levar em consideração a sua nota para verificar se foi aprovado, reprovado por nota ou se está de recuperação. Mas, se o aluno tiver sua frequência inferior a 75%, está reprovado por frequência.

Tente imaginar como você faria o algoritmo acima. Há várias possibilidades.

Antes de começarmos a escrever o novo algoritmo, precisamos pensar um pouco para que nossa solução seja bem elaborada. Uma boa forma de começarmos é levantarmos os seguintes dados: qual a condição que menos possibilidades há? Existe alguma condição que depende da outra (ou alguma condição mais "forte" que outra)? Estas são duas dicas iniciais a serem pensadas antes de fazermos nosso algoritmo.

No caso do nosso exemplo, não importa a nota do aluno, se ele não tiver pelo menos 75% de presença, estará reprovado. A nota só é importante, caso o aluno tenha pelo menos 75% de presença. E para a nota, existem três possibilidades (aprovado, recuperação ou reprovado). Intuitivamente, podemos dizer que a frequência é uma condição mais forte que nota porque ela só precisa ser verificada se o aluno tiver frequentado 75% ou mais das aulas.

Deste modo, uma regra possível é:

- O algoritmo deve pedir a nota do aluno;
- O algoritmo deve pedir a frequência do aluno;
- Se a frequência for igual ou maior que 75 (por cento), então

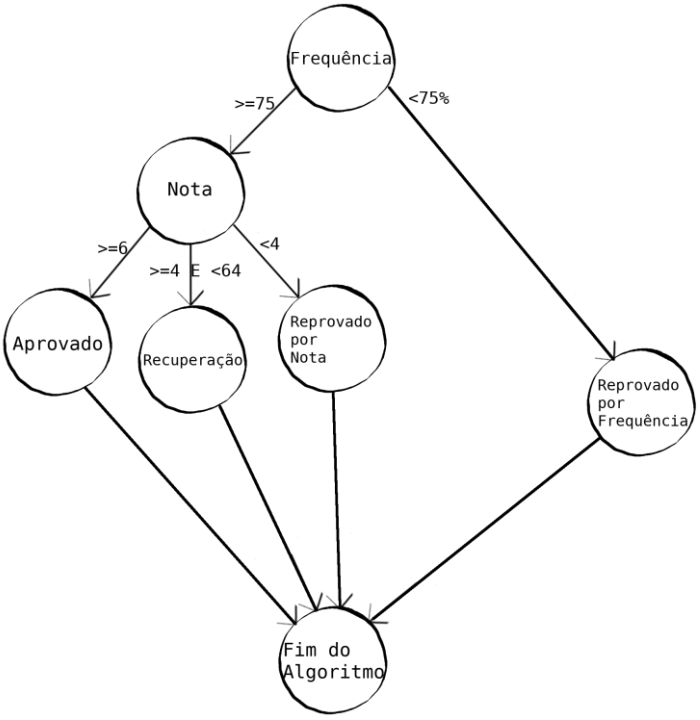
    - Se a nota for maior ou igual a 6, está aprovado;
    - Se a nota for maior ou igual a 4 (e menor que 6), está de recuperação;
    - Se a nota for inferior a 4, está reprovado por falta;
- Se a frequência for inferior a 75, então o aluno está reprovado por frequência.

Repare que, no exemplo acima, a nota só precisa ser verificada se a frequência for maior ou igual a 75%. Outra forma de se pensar o que acabou de ler é que a verificação da nota só será escrita dentro do bloco de condição ***se*** da frequência. O algoritmo para este problema está na sequência. Estude-o com bastante atenção.

```
1.    Inicio
2.        real nota, frequencia
3.        escreva("Digite a sua nota:")
4.        leia(nota)
5.        escreva("Digite a sua frequência")
6.        leia(frequência)
7.        se(frequencia>=75) então
8.              se(nota>=6) então
9.                  escreva("Aprovado!")
10.             senão se(nota<4) então
11.                 escreva("Reprovado por nota!")
12.             senão
13.                 escreva("Recuperação!")
14.             fimse
15.       senão
16.           escreva("Reprovado por frequência!")
17.       fimse
18.   fim
```

Uma forma de se visualizar o algoritmo acima é com o grafo da Figura 4:

Figura 4 - Se-senão aninhado



A explicação a seguir exige um pouco de concentração. Se precisar voltar a ler o mesmo parágrafo, faça-o! Não há problema nenhum nisso! Neste algoritmo apresentado, as notas (da linha 8 a 14) só serão verificadas se a frequência for maior ou igual a 75%. Se a frequência for menor que 75%, a execução do algoritmo deverá pular da linha 7 para 15, sem precisar verificar as notas. Dizemos que há dois blocos ***se***: um que se refere às notas e é o bloco mais interno; e outro bloco que se refere à frequência, o mais externo. O bloco ***se*** das notas é interno ao bloco de frequência porque ele só será verificado se a condição de frequência da linha 7 for satisfeita.

Repare que o ***fimse*** da linha 14 refere-se ao ***se*** da linha 8; da mesma forma, o ***fimse*** da linha 17 refere-se ao ***se*** da linha 7. A ideia por trás de ***se-senão*** dentro de outro é: o primeiro ***fimse*** que aparecer refere-se ao último ***se*** que aparecer. O último ***fimse*** que aparecer refere-se ao primeiro ***se*** que aparecer.

Vamos a uma outra dúvida comum: há dois ***senão***, um na linha 12 e um na linha 15. Como saber a qual bloco cada um pertence? É simples! Enquanto o **senão** estiver entre um ***se*** e um ***fimse***, ele pertencerá a este bloco mais interno. No nosso exemplo, o ***senão*** da linha 12 está entre o ***se*** da linha 8 e o ***fimse*** da linha 14, então ele pertence a este bloco mais interno. Já o ***senão*** da linha 15 aparece após o ***fimse*** da linha 14, então ele não pertence mais a este bloco interno. Conclusão: este ***senão*** da linha 15 pertence ao ***se*** da linha 7.

Em resumo: você pode colocar vários blocos de instruções, um dentro do outro um **ANINHADO** ao outro). O que delimita o fim do bloco é o marcador de fim de bloco (no nosso caso, fimse). Segue um exemplo genérico abaixo:

```
1.        se (<condição 1>) então
2.              <aqui pode ter mais instruções do bloco da condição 1>
3.              se(<condição2>) então
4.                    <instruções do bloco da condição 2>
5.                    se(<condição 3>) então
6.                       <instruções do bloco da condição 3>
7.                    fimse <-representa o fim do bloco da condição 3
8.              fimse       <-representa o fim do bloco da condição 2
9.        fimse             <-representa o fim do bloco da condição 1
```

Com este exemplo, se houvessem várias linhas de código entre o ***se*** da linha 5 e o ***fimse*** da linha 7, estas só seriam executadas se a condição da linha 5 fosse verdadeira. Do mesmo modo, todas as instruções entre a condição da linha 3 e o ***fimse*** da linha 8 só seriam executadas se a condição da linha 3 fosse verdadeira. A mesma ideia segue para a condição do bloco mais externo (demarcado pelas linhas 1 e 9).

E se não planejarmos o nosso código até o momento?

Quando temos mais de uma variável e percebemos que a condição de uma variável influencia na condição da outra, é interessante planejarmos como aninhar uma condição dentro da outra.

Anteriormente, foi dada uma dica: qual a condição que menos possibilidades há? Existe alguma condição que depende da outra (ou alguma

condição mais “forte” que outra)? Estas são duas dicas iniciais a serem pensadas antes de sairmos fazendo nosso algoritmo.

Se não seguirmos esta dica, ainda podemos fazer um algoritmo que funcione, mas ele provavelmente ficará mais complicado de se escrever. No exemplo anterior de notas e frequência, primeiro verificamos a frequência para depois analisar a nota. Isso porque seguimos a dica dada: só existem duas possibilidades de frequência (maior ou igual a 75% e maior que 75%), mas existem três opções de notas, que reflete em aprovado, reprovado ou recuperação. E a nota só precisa ser verificada se a frequência for maior ou igual a 75%.

Se nós optarmos por verificar primeiro a nota e depois a frequência, também conseguiremos montar um algoritmo que funcione. Mas provavelmente não ficará tão bom. Segue o exemplo:

```
1.     inicio
2.        real nota, frequencia
3.        escreva(“Digite a sua nota:”)
4.        leia(nota)
5.        escreva(“Digite a sua frequência”)
6.        leia(frequência)
7.        se(nota>=6) então
8.            se(frequencia>=75) então
9.               escreva(“Aprovado!”)
10.           senão
11.              escreva(“Reprovado por falta!”)
12.           fimse
13.      senão se(nota>=4) então
14.           se(frequencia>=75) então
15.              escreva(“Recuperação!”)
16.           senão
17.              escreva(“Reprovado por falta!”)
18.           fimse
19.      senão
20.           se(frequencia>=75) então
21.              escreva(“Reprovado por nota!”)
22.           senão
23.              escreva(“Reprovado por falta!”)
24.           fimse
25.      fimse
26.  fim
```

Repare que, neste último exemplo, o algoritmo também faz o que deveria, mas precisa de mais linhas de código. Tente analisar o porquê.

Para cada grupo de notas, será necessário verificar se o aluno frequentou o mínimo necessário! Então, se tirou nota igual ou superior a 6, deverá ser analisado se o aluno tem o mínimo necessário de frequência; se o aluno estiver de recuperação, idem; se for reprovado, foi por nota ou frequência?

É por isso que a dica é interessante: quanto menos condições precisarem ser verificadas, mais simples ficará o código. Ou ainda, quanto menor o número de combinações de condições, mais simples ficará o código.

No exemplo de notas e frequência, é simples fazer a análise: as notas só precisam ser analisadas se o aluno tiver o mínimo de frequência. Então, este é o caminho: primeiro verificar a frequência para depois verificar a nota.

Você verá que, em muitos problemas, é possível criar um algoritmo usando operadores lógicos ou usando se-senão aninhado. Qual opção deverá escolher? A que considerar mais simples! Por exemplo, na seção anterior sobre aposentadoria, usamos operadores lógicos &&. Mas podemos usar se-senão aninhado. Usando operadores lógicos (condições compostas):

```
1.    inicio
2.        caractere genero
3.        int tempo
4.        escreva(“Digite (m ou f):”)
5.        leia(genero)
6.        escreva(“Tempo de trabalho:”)
7.        leia(tempo)
8.        se (genero==’m’ && tempo>=30) então
9.            escreva(“Você pode se aposentar!”)
10.       senão se (genero==’f’ && tempo>=25) então
11.           escreva(“Você pode se aposentar!”)
12.       senão
13.           escreva(“Você não pode se aposentar!”)
14.       fimse
15.   fim
```

E usando ***se-senão*** aninhado:

```
1.   inicio
2.       caractere genero
3.       int tempo
4.       escreva(“Digite (m ou f):”)
5.       leia(genero)
6.       escreva(“Tempo de trabalho:”)
7.       leia(tempo)
8.       se (genero==’m’) então
9.           se (tempo>=30) então
10.             escreva(“Você pode se aposentar!”)
11.          senão
12.             escreva(“Você não pode se aposentar!”)
13.          fimse
14.      senão se (genero==’f’) então
15.          se (tempo>=25)
16.              escreva(“Você pode se aposentar!”)
17.          senão
18.              escreva(“Você não pode se aposentar!”)
19.           fimse
20.      fimse
21.  fim
```

Neste caso, parece mais simples o primeiro algoritmo. Se você prestar atenção, conseguirá entender o porquê. Gênero e tempo possuem o mesmo número de comparações e são interdependentes. Além disso, no primeiro exemplo, em particular, precisamos verificar três situações: duas condições compostas e um senão. Repare que o algoritmo só precisa verificar se não pode se aposentar uma vez (no senão). Se formos usar *se-senão* aninhado, neste exemplo, teremos que verificar se não pode se aposentar duas vezes, nas linhas 12 e 18!

O divertido em Algoritmos é justamente a possibilidade de se obter várias soluções diferentes para um mesmo problema. Como exercício de imaginação, você conseguiria refazer este exemplo de outra forma?

Exercícios de se-senão Aninhado

Para todos os exercícios abaixo, use se-senão aninhado.

1. Imagine um parque infantil. Faça um algoritmo que peça a altura do usuário e a idade.
    a. Se tiver altura menor que 1,6m, pode entrar no parque. Neste caso:
        i. Se tiver menos de 5 anos, escreva a mensagem: “Pode brincar no pula-pula e Casinha”;
        ii. Se tiver entre 5 e 8 anos, escreva a mensagem: “Pode brincar na prancha do pirata e piscina de bolinhas”.
        iii. Se tiver mais de 8 anos, escreva a mensagem: “Pode brincar de pebolim, ping-pong e basquete”.
    b. Se tiver altura maior ou igual a 1,6m, escreva a mensagem: “Você é muito grande para entrar no parque”.

2. Faça um algoritmo que peça o semestre em que está cursando a faculdade.
    a. Se estiver no sétimo período ou superior, o algoritmo deve apresentar o menu: 1-matricular na disciplina de Jogos Digitais; 2-Matricular na disciplina de Design de Jogos; 3-Matricular na disciplina de Realidade Virtual.
        i. Se a opção for 1, apresentar a mensagem: “Jogos Digitais confirmado”;
        ii. Se a opção for 2, apresentar a mensagem: “Design confirmado”;
        iii. Se a opção for 3, apresentar a mensagem: “RV confirmado”.
    b. Caso não esteja pelo menos no sétimo período, apresentar a mensagem: “Você ainda não pode se matricular em disciplinas optativas”.

3. Faça um algoritmo que peça para o usuário digitar se é “brasileiro” ou “estrangeiro”.
    a. Se for brasileiro, perguntar a região em que nasceu.
        i. Se for “sul”, apresentar a mensagem “você é está acostumado com frio”;
        ii. Se for “sudeste”, apresentar a mensagem “você está acostumado com chuva”;
        iii. Se for “centro-oeste”, apresentar a mensagem “você está acostumado com clima abafado”;
        iv. Se for “nordeste”, apresentar a mensagem “você está acostumado com praias bonitas”;
        v. Se for “norte”, apresentar a mensagem “você está acostumado com chuvas no começo da tarde”.

   b. Se for estrangeiro, apresentar a mensagem “seja bem-vindo ao Brasil”.

4. Imagine um problema de gerenciamento de fila de um banco que emite senhas para aguardar na fila. Faça um algoritmo que:
   a. Peça o número da senha para o usuário;
   b. Apresente o menu 1-Prioridade; 2-Aposentado; 3-Comum.
   c. Se o número da senha for menor que 100, o algoritmo deve apresentar a mensagem: “aguarde para ser atendido”.
   d. Senão, o algoritmo deve apresentar uma mensagem, conforme a opção:
      i. Se for 1, “Você será reagendado para amanhã”;
      ii. Se for 2, “Você será reagendado para depois de amanhã”;
      iii. Se for 3, “Você deve tentar outro dia”.
5. Faça um algoritmo que apresente o menu: “1-Conversão de temperatura; 2-Conversão de distância”. Se o usuário escolher a opção 1, apresentar o menu: “1-Converter Celsius para Farenheit; 2-Converter Farenheit para Celsius; 3-Converter Celsius para Kelvin”. O algoritmo deverá, então, pedir uma temperatura, realizar a conversão escolhida e apresentar a mensagem com o valor convertido.
   Por outro lado, se o usuário escolher a conversão de distância (opção 2 do primeiro menu), então, apresentar o menu: “1-Converter km para milhas; 2-Converter milhas para km”. Então, o algoritmo deverá realizar a respectiva conversão e apresentar a mensagem com o valor convertido.

   Os cálculos para conversão são:

   - Farenheit = (9*Celsius/5) + 32
   - Celsius = 5*(Farenheit-32)/9
   - Kelvin = Celsius + 273
   - Milha = km * 0.62137
   - Km = milha / 0.62137

   Para este exercício, crie duas soluções de algoritmo. Você pode tentar qual SE-SENÃO achar melhor (aninhado, encadeado, SE-SENÃO SE). Ou mesmo, ESCOLHA-CASO. Depois que fizer as duas versões do algoritmo, compare-os e responda: qual o melhor? Por quê?

   DICA: o enunciado parece grande. Mas pense por partes. Há um **menu inicial** e, dependendo da escolha, um diferente menu. Então, perceba que várias linhas do exercício se resumem a uma explicação de uma linha e meia.

Como é em algumas Linguagens

Português Estruturado

```
inicio
    real altura
    inteiro idade

    escreva("Digite a sua altura: ")
    leia(altura)
    escreva("Digite a sua idade: ")
    leia(idade)

    SE(altura<1.6)ENTÃO
        SE(idade<5)ENTÃO
            escreva("Pode brincar de pula-pula e Casinha")
        SENÃO SE(idade<=8)ENTÃO
            escreva("Pode brincar na prancha do pirata")
        SENÃO
            escreva("Pode brincar de pebolim e ping-pong")
        FIMSE
    SENÃO
        escreva("Você é muito grande para entrar no parque")
    FIMSE
fim
```

C

```c
#include <stdio.h>

int main(){
    float altura;
    int idade;

    printf("\nDigite a sua altura: ");
    scanf("%f",&altura);
    printf("\nDigite a sua idade: ");
    scanf("%d",&idade);

    if(altura<1.6){
        if(idade<5){
            printf("\nPode brincar de pula-pula e Casinha");
        }else if(idade<=8){
            printf("\nPode brincar na prancha do pirata");
        }else{
            printf("\nPode brincar de pebolim e ping-pong");
        }
    }else{
        printf("\nVocê é muito grande para entrar no parque");
    }
}
```

Obs: vale lembrar que em C/C++, quando um bloco possui apenas uma instrução, as chaves são opcionais. É interessante notar que uma instrução IF ou IF-ELSE também é considerada uma única instrução. Deste modo, o trecho acima de código de IF-ELSE aninhado também poderia ser escrito como segue:

```c
    ...
    if(altura<1.6)
        if(idade<5)
            printf("\nPode brincar de pula-pula e Casinha");
        else if(idade<=8)
            printf("\nPode brincar na prancha do pirata");
        else
            printf("\nPode brincar de pebolim e ping-pong");
    else
        printf("\nVocê é muito grande para entrar no parque");
    ...
```

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    float altura;
    int idade;

    cout << "\nDigite a sua altura: ";
    cin >> altura;
    cout << "\nDigite a sua idade: ";
    cin >> idade;

    if(altura<1.6){
        if(idade<5){
            cout << "\nPode brincar de pula-pula e Casinha";
        }else if(idade<=8){
            cout << "\nPode brincar na prancha do pirata";
        }else{
            cout << "\nPode brincar de pebolim e ping-pong";
        }
    }else{
        cout << "\nVocê é muito grande para entrar no parque";
    }
}
```

Obs: vale lembrar que em C/C++, quando um bloco possui apenas uma instrução, as chaves são opcionais. É interessante notar que uma instrução IF ou IF-ELSE também é considerada uma única instrução. Deste modo, o trecho acima de código de IF-ELSE aninhado também poderia ser escrito como segue:

```
    ...
    if(altura<1.6)
        if(idade<5)
            cout << "\nPode brincar de pula-pula e Casinha";
        else if(idade<=8)
            cout << "\nPode brincar na prancha do pirata";
        else
            cout << "\nPode brincar de pebolim e ping-pong";
    else
        cout << "\nVocê é muito grande para entrar no parque";
    ...
```

Go

```
package main

import (
    "fmt"
)

func main() {
    var altura float32
    var idade int

    fmt.Print("Digite a sua altura: ")
    fmt.Scanf("%f", &altura)
    fmt.Print("Digite a sua idade: ")
    fmt.Scanf("\n%d", &idade)

    if altura < 1.6 {
        if idade < 5 {
            fmt.Print("Pode brincar de pula-pula e casinha")
        } else if idade <= 8 {
            fmt.Print("Pode brincar na prancha do pirata")
        } else {
            fmt.Print("Pode brincar de pebolim e ping-pong")
        }
    } else {
        fmt.Print("Você é muito grande para entrar no parque")
    }
}
```

OBS: em Go, é obrigatório o uso de chaves, mesmo que exista apenas uma instrução dentro do bloco.

Ruby

```
print "Digite a sua altura: "
altura = gets.chomp.to_f
print "Digite a sua idade: "
idade = gets.chomp.to_i

if altura < 1.6
    if idade < 5
        puts "Pode brincar de pula-pula e Casinha"
    elsif idade <=8
        puts "Pode brincar na prancha do pirata"
    else
        puts "Pode brincar de pebolim e ping-pong"
    end
else
    puts "Você é muito grande para entrar no parque"
end
```

Python

```
altura = float(input('Digite a sua altura: '))
idade = int(input('Digite a sua idade: '))

if altura < 1.6:
    if idade < 5:
        print('Pode brincar de pula-pula e Casinha')
    elif idade <=8:
        print('Pode brincar na prancha do pirata')
    else:
        print('Pode brincar de pebolim e ping-pong')
else:
    print('Você é muito grande para entrar no parque')
```

Obs: lembre-se que em Python, as instruções dentro de um bloco são marcadas exclusivamente pela indentação.

Obs: lembre-se que em Python, as instruções dentro de um bloco são marcadas exclusivamente pela indentação.

Estruturas de Repetição

Neste ponto, já temos instruções suficientes para fazer algoritmos que permitam tomadas de decisões, como realizar cálculos de acordo com a opção do usuário. Agora é o momento de aprendermos como podemos repetir um mesmo código diversas vezes.

Imagine que você queira fazer um algoritmo que conte de 1 a 10. Você poderia criá-lo com o que já aprendeu até o momento:

```
1.    inicio
2.        escreva(“1”)
3.        escreva(“2”)
4.        escreva(“3”)
5.        escreva(“4”)
6.        escreva(“5”)
7.        escreva(“6”)
8.        escreva(“7”)
9.        escreva(“8”)
10.       escreva(“9”)
11.       escreva(“10”)
12.   fim
```

Ok. Mas e se precisássemos criar um algoritmo que contasse de 1 a 1000? Não seria muito prático escrever 1000 instruções ***escreva***. A situação do programador ficaria ainda mais difícil se ele precisasse repetir uma sequência de instruções até que o usuário digitasse 0. Por exemplo, suponha um algoritmo parecido com o que já fizemos nos exercícios anteriores, mas com essa pequena mudança de menu:

1-Converter Celsius para Farenheit;
2-Converter Farenheit para Celsius;
3-Converter Celsius para Kelvin;
0-Sair.

Agora, o algoritmo deve esperar que o usuário digite um dos quatro valores apresentados. Se o usuário digitar 1, 2 ou 3, o algoritmo deve pedir a temperatura, fazer o cálculo, apresentar na tela E VOLTAR A APRESENTAR O MENU PRINCIPAL NOVAMENTE PARA QUE O USUÁRIO POSSA VOLTAR A

FAZER MAIS CONVERSÕES!!! Apenas se ele digitar 0 o algoritmo deve ser encerrado.

Repare que neste algoritmo, o programador não tem nem como saber a quantidade de vezes que o usuário irá querer realizar conversões. O usuário mais animado poderia passar dias e dias querendo converter temperaturas antes de encerrar o algoritmo.

Nesta seção, veremos as instruções que podem ser usadas para repetirmos sequências de instruções: ***para***, ***enquanto*** e ***faça-enquanto***. Costumamos chamar estas instruções de laços ou estruturas de repetições.

Laço PARA

Esta instrução é a ideal para situações em que sabemos (nós programadores e, consequentemente nosso algoritmo) a quantidade de vezes que precisamos repetir uma sequência de instruções.

Por exemplo, como fazemos para um algoritmo apresentar na tela 5 vezes as mensagens "Iaba", "Daba" e "du"? Novamente, poderíamos fazer as três instruções escreva, depois copiá-las e colá-las 5 vezes, o que nos daria 15 linhas de código.

Mas, usando a instrução ***para***, podemos pedir para que o algoritmo repita estas três instruções por 5 vezes. Muito mais simples! Para isso, precisamos criar uma variável contadora, cuja função é armazenar quantas vezes já foram repetidas as instruções que se deseja repetir. Essa variável, então, serve para o algoritmo controlar a quantidade de repetições já realizadas. Precisamos, também, fornecer os seguintes dados para a instrução ***para***:

- Qual o valor inicial da contagem (vai começar do 0? Do 1? De que valor?). Chamamos isso de inicialização.
- Em que condição as instruções devem ser repetidas (a gente também chama isso de condição para se "entrar no laço"). Por exemplo: se a variável contadora for menor que 6, repita!
- De quanto em quanto a variável deve ser incrementada ou decrementada. Por exemplo, conte de 1 em 1 ou de 2 em 2, etc...

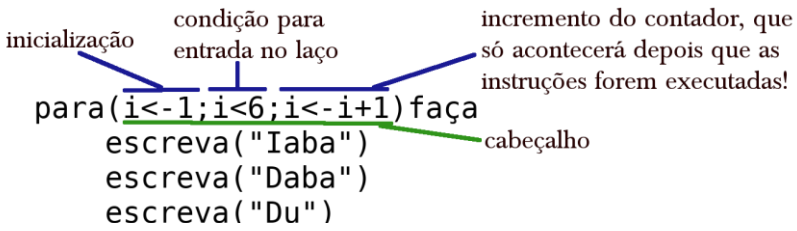
No nosso exemplo, podemos criar uma variável contadora chamada *i*. Então, colocamos o valor inicial 1. Informamos à instrução que as instruções devem ser repetidas enquanto i<6 (ou i<=5). Por fim, informamos que a variável *i* deve ser incrementada em 1 depois de repetir as três instruções. Todas estas informações vão no que chamamos de cabeçalho da instrução ***para***.

A ideia é simples, certo? Agora precisamos apenas saber como escrevemos isso na nossa linguagem:

```
1.    inicio
2.         inteiro i
3.         PARA(i<-1; i<6; i<- i + 1) FAÇA
4.              escreva(“Iaba”)
5.              escreva(“Daba”)
6.              escreva(“Du!”)
7.         fimpara
8.    fim
```

Vamos, agora, analisar este laço, isoladamente (Figura 5).

Figura 5 - Laço ***para***



O que determina o fim do laço ***para*** é a palavra ***fimpara***. E chamamos de bloco do laço ***para*** tudo aquilo que estiver entre o ***para*** e o ***fimpara***. Com este laço, todas as instruções entre ***para*** e ***fimpara*** irão se repetir enquanto a condição do cabeçalho for verdadeira! Pode-se colocar quantas e quais instruções forem necessárias.

A execução do laço ***para*** ocorrerá da seguinte forma (estou sendo propositalmente repetitivo. Leia todos os passos!):

1. A inicialização representa qual o primeiro valor que a variável *i* terá. Uma regra importante é que esta inicialização só será realizada uma vez. No nosso exemplo, então, o primeiro valor de *i* é 1.
2. Em seguida, é verificada a condição descrita no cabeçalho (*i*<6?). Sim, porque *i* começa valendo 1 e 1 é menor que 6. Sendo a condição verdadeira, as instruções dentro do laço serão executadas.
3. Assim que as instruções forem executadas, só no final a variável contadora *i* será incrementada em 1, como indicado no cabeçalho. Então, da primeira vez que o código for executado, após as três instruções serem executadas, a variável *i* passará a valer 2 (1 + 1).
4. Quando chegar no ***fimpara*** o algoritmo retornará para o cabeçalho e fará a verificação novamente (*i*<6?). Como *i* desta vez vale 2, a condição é verdadeira e as instruções dentro do laço serão executadas mais uma vez.
5. Assim que as três instruções forem executadas, novamente a variável *i* será incrementada em 1 (passará a valer 3).
6. Quando chegar no ***fimpara*** o algoritmo retornará para o cabeçalho e fará a verificação novamente (*i*<6?). Como *i* desta vez vale 3, a condição é verdadeira e as instruções dentro do laço serão executadas mais uma vez.
7. Assim que as três instruções forem executadas, novamente a variável *i* será incrementada em 1 (passará a valer 4).
8. Quando chegar no ***fimpara*** o algoritmo retorna para o cabeçalho e faz a verificação novamente (*i*<6?). Como *i* desta vez vale 4, a condição é verdadeira e as instruções dentro do laço serão executadas mais uma vez.
9. Assim que as três instruções forem executadas, novamente a variável *i* será incrementada em 1 (passará a valer 5).
10. Quando chegar no ***fimpara*** o algoritmo retornará para o cabeçalho e fará a verificação novamente (*i*<6?). Como *i* desta vez vale 5, a condição é verdadeira e as instruções dentro do laço serão executadas mais uma vez.
11. Assim que as três instruções forem executadas, novamente a variável *i* será incrementada em 1 (passará a valer 6).
12. Quando chegar no ***fimpara*** o algoritmo retornará para o cabeçalho e fará a verificação novamente (*i*<6?). Como *i* desta vez vale 6, a

condição é falsa (porque 6 não é menor que 6!). Neste ponto, o algoritmo pulará para o fim do laço (***fimpara***) e continuará o algoritmo. Dizemos também que o algoritmo saiu do laço. Como não há mais nada depois de ***fimpara***, o algoritmo terminará.

E como fazemos um algoritmo que conte de 1 a 10?

```
1.    inicio
2.         inteiro i
3.         para(i<-1; i<11; i<- i + 1) faça
4.              escreva(i)
5.         fimpara
6.    fim
```

Desta vez, simplesmente pedimos para o algoritmo escrever o valor de `i`:

1. Inicialmente `i` vale 1;
2. Será, então feita a comparação. Como `i` vale 1, então a condição `i<11` é verdadeira. Por isso, será executada a instrução do bloco do laço ***para***;
3. A instrução ***escreva*** apresentará o valor de `i`, que é 1;
4. Só após a instrução ser executada é que `i` será incrementada e passará a valer 2.
5. O código voltará para o cabeçalho, será verificado se `i<11` (`i` vale 2, agora). Como a condição é verdadeira, será executada a instrução ***escreva***;
6. A instrução ***escreva*** apresentará o valor de `i`, que é 2;
7. Só após a instrução ser executada é que `i` será incrementada e passará a valer 3.
8. O código voltará para o cabeçalho, será verificado se `i<11` (`i` vale 3, agora). Como a condição é verdadeira, será executada a instrução ***escreva***;
9. <e assim sucessivamente até que `i` passe a valer 11>.
10. Quando `i` for igual a 11, e o código voltar ao cabeçalho, então a condição será falsa e o algoritmo sairá do laço ***para***.

Para fixar o conceito, repasse os dois exemplos anteriores até que fique automático em sua mente.

Incremento e Decremento de variáveis

Muitas linguagens permitem que o incremento ou decremento das variáveis seja feito com uma notação diferente, com o objetivo de simplificar a escrita. Por exemplo, se quisermos incrementar uma variável em 1, podemos fazê-lo, como abaixo:

```
i <- i + 1(geralmente, i=i+1, em outras linguagens)
i +<- 1  (geralmente, i+=1, em outras linguagens)
i++
++i
```

Os dois primeiros casos são mais flexíveis no sentido de que podemos trocar o número do incremento por qualquer valor, inclusive uma outra variável:

`i +<- 2` (ou **`i+=2`**) incrementa *`i`* de 2 em 2.
`i +<- x` (ou **`i+=x`**) incrementa *`i`* de x em x, e é o mesmo que **`i <- i + x`**.

Já os dois últimos casos são mais rígidos neste sentido. Também pode-se usar o análogo para o decremento, como:

```
i--
--i
```

Outro aspecto importante é o significado da posição dos operadores:

Em **`++i`**, a primeira operação a ser realizada é o incremento da variável.
Em **`i++`**, o incremento apenas será realizado depois que a linha toda for executada.

No caso do laço ***para***, não há diferença entre **`++i`** ou **`i++`** porque o incremento é a única coisa a ser realizada quando o bloco terminar de ser executado. Mas preste atenção nos exemplos abaixo. Suponha que o valor de *`i`* seja 2:

```
i = 2
a = 3 * ++i
```

Neste caso, o *`++i`* é executado primeiro. Então a variável *`i`* passa a valer 3, para depois ser executada a multiplicação e a atribuição do resultado à variável ***a***. A variável ***a*** receberá 9.

Agora, em:

```
i=2
a = 3 * i++
```

A multiplicação e atribuição serão executadas primeiro. Então o valor original de *i* será usado na multiplicação, produzindo o resultado 6. Em seguida **a variável *a* receberá esse valor 6**. Apenas quando a linha terminar de ser executada, o valor de *i* será incrementado.

Repare que, nos dois casos, ao final da respectiva linha, o valor de *i* será 3. O que muda é o momento em que o incremento da variável será realizado.

Outro aspecto importante é que normalmente, a variável contadora é declarada dentro do próprio cabeçalho do laço ***para***, como segue abaixo:

```
para(int i<-1; i<=max;i++)faça
    escreva(i)
fimpara
```

A vantagem de se declarar a variável no próprio cabeçalho é que essa variável apenas será visível dentro do bloco do laço ***para***. Deste modo, se tentássemos usar a variável *i* fora do laço, esta variável não seria reconhecida por não existir fora dele.

Agora é a hora de realizar os exercícios de fixação.

Exercícios de Laço para

1. Faça um algoritmo que apresente os valores de 1 a 10.

2. Faça um algoritmo que apresente os valores de 10 a 1.

3. Faça um algoritmo que apresente os valores de 5 a 15.

4. Faça um algoritmo que apresente os valores de 15 a 5.

5. Faça um algoritmo que apresente os valores pares de 1 a 10.

6. Faça um algoritmo que apresente os valores ímpares de 1 a 10.

7. Faça um algoritmo que peça um número positivo maior que 0 e apresente os valores de 1 a este valor digitado.

8. Faça um algoritmo que peça um número menor que 10 e apresente os valores deste número digitado a 10. Por exemplo, se o usuário digitar 8, deverá apresentar os valores: 8, 9 e 10.

9. Faça um algoritmo que peça um número inicial e um número final. Em seguida, deve ser apresentada a contagem do número inicial ao número final. Suponha que o número inicial é sempre menor que o final.

10. Faça um algoritmo que peça um número inicial e um número final. Em seguida, deve ser apresentada a contagem do número inicial ao número final. Se o número inicial for menor que o final, então a contagem será em ordem crescente. Do contrário, a contagem deverá ser em ordem decrescente. Portanto, se o usuário digitar primeiro 2, depois 5, deverá ser apresentado: 2, 3, 4 e 5. Mas se o usuário digitar primeiro 5, depois 2, deverá ser apresentado: 5, 4, 3, 2 e 1.

Como é em algumas linguagens

Português Estruturado

```
inicio
    inteiro i, max
    para(i<-1;i<11;i<-i + 1)faça
        escreva(i)
    fimpara
fim
```

C

```
#include <stdio.h>

int main(){
    int i, max;

    printf("\nDigite o valor máximo: ");
    scanf("%d",&max);

    for(i=1; i<=max;i++){
        printf("\n%d",i);
    }
}
```

OBS: em C, o incremento/decremento da variável pode ser feito de diversas formas, como citado no final da seção sobre laço ***para***. E a variável também pode ser declarada dentro do próprio cabeçalho, como abaixo:

```
int main(){
    int max;

    printf("\nDigite o valor máximo: ");
    scanf("%d",&max);

    for(int i=1; i<=max;i++){
        printf("\n%d",i);
    }
}
```

A variável é, ao mesmo tempo, declarada e inicializada com o valor 1.

Apesar de não ser comum, é possível colocar mais de uma atribuição inicial, ou condição de continuidade ou, até mesmo, incremento no cabeçalho do laço ***for***. Para isso, separa-se por vírgulas:

```
 for(i=1, x=10; i<=max, i<=10;i++, x--){
        printf("\n%d",i);
        printf("\n%d, x);
 }
```

No exemplo acima, duas variáveis são iniciadas, *i* e *x*.
Na condição de continuidade, a vírgula representa um E. Deste modo, enquanto *i<=max* E enquanto *i<=10*, deve-se continuar. Por isso, se o valor de *max* for superior a 10, o laço ***for*** será encerrado quando *i* valer 11.
E, duas variáveis terão seus valores alterados: *i* será incrementado e *x* será decrementado.

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    int i, max;

    cout << "\nDigite o valor máximo: ";
    cin >> max;

    for(i=1; i<=max;i++){
        printf("\n%d",i);
    }
}
```

Obs: as mesmas observações feitas em C se aplicam a C++.

Mas em C++, há variações do laço ***for*** que não se aplicam ao C. Por exemplo, a variação normalmente chamada de ***foreach***. Em C++11, versão da linguagem publicada a partir de 2011, é possível percorrer um vetor da seguinte forma:

```
float meuvetor[] = {9.2f, 1.3f, 6.4f, 3.2f, 4.1f};
for (float elemento : meuvetor) {
    cout << elemento << '\n';
 }
```

Vetores são variáveis que armazenam mais de um valor. É como um gaveteiro em que em cada gaveta armazena um valor. Então, o gaveteiro possui um nome, como **meuvetor**, e cada gaveta possui uma posição para poder colocar um valor. A primeira posição do meuvetor armazena o valor 9.2f, a segunda posição do vetor meuvetor armazena o valor 1.3f e assim por diante. Mais sobre vetores na Seção XYZ.

No nosso exemplo, o vetor possui 5 elementos, que são valores do tipo float (real). O f ao lado de cada número indica ao compilador que você está representando um valor do tipo float. Sem o f, o compilador interpretará o valor como um double.

O importante é que esse laço ***for***, automaticamente reconhece o tamanho do vetor e, em cada volta do laço, também automaticamente atribuirá o valor do próximo elemento do vetor à variável `elemento`.

Assim, na primeira vez que o laço for executado, `elemento` valerá 9.2f; na segunda vez que o laço for executado, `elemento` valerá 1.3f; e assim sucessivamente até que o último valor seja atribuído para `elemento`.
O bacana é que não precisamos usar uma variável contadora. Esta variação do laço for simplesmente retorna o valor de cada elemento.

Go

```
package main

import (
    "fmt"
)

func main() {
    var i, max int

    fmt.Print("Digite o valor máximo: ")
    fmt.Scanf("%d", &max)

    for i = 1; i <= max; i++ {
        fmt.Println(i)
    }
}
```

OBS: em GO, o incremento/decremento da variável pode ser feito de diversas formas, como citado no final da seção sobre laço ***para***. E a variável também pode ser declarada dentro do próprio cabeçalho.

Nesta linguagem, quando a variável é declarada ao mesmo tempo em que é inicializada por algum valor, pode se usar duas notações, por exemplo: `var i = 1` ou `i := 1` (nesta última, usa-se o símbolo `:=` no lugar de `=` e não se usa `var`).

```
func main() {
    var max int

    fmt.Print("Digite o valor máximo: ")
    fmt.Scanf("%d", &max)

    for var i = 1; i <= max; i++ {
        fmt.Println(i)
    }
}
```

Repare que no exemplo acima, usamos `var i = 1`. Isso significa que a variável está sendo declarada naquele momento e o valor 1 será atribuído a ela. O tipo da variável será correspondente ao valor passado. Essa forma de declaração pode ser usada em qualquer lugar do código, quando se deseja declarar uma variável ao mesmo tempo em que se deseja inicializá-la com algum valor.

```
func main() {
    x := “Miguel”
    y := 1
    ...
```

Já nos exemplos mostrados neste último código acima, duas variáveis estão sendo declaradas ao mesmo tempo em que são inicializadas, usando a segunda notação. A variável *x* será do tipo cadeia, porque está sendo inicializada com uma cadeia de caracteres. E a variável *y* será do tipo inteiro porque está sendo inicializada com um número inteiro.
Uma vez que a variável seja inicializada, em Go, não se pode alterar o seu tipo.
Go também possui variações do laço ***for***, como C++ para recuperar valores de um vetor:

```
x := []int{3, 1, 4, 2}

for i, elemento := range x {
    fmt.Println(i, elemento)
}
```

Neste exemplo, foi criado primeiro um vetor de quatro elementos chamado `x`. O primeiro elemento é 3, o segundo é 1 e assim por diante. Esse laço ***for*** possui um contador `i` (cujo valor inicial será 0 e o valor final será a quantidade de elementos de `x` menos 1 (3).
Então, a instrução Println apresentará os seguintes valores:

0 3
1 1
2 4
3 2

Isso porque o elemento 0 é o número 3, o elemento 1 é o número 1, o elemento 2 é o número 4 e o elemento 3 é o número 2. Repare que a primeira posição do vetor é 0.

Ruby

```
print "Digite o valor máximo: "
max = gets.chomp.to_i

for i in 1..max  do
    puts i
end
```

OBS: em Ruby, o laço ***for*** por padrão realiza incremento da variável contadora de 1 em 1. Deste modo, no cabeçalho deve-se indicar o valor inicial e final da variável. No exemplo acima, o valor inicial da variável será 1 e ela deverá ser incrementada até chegar no valor armazenado em *max*. O valor inicial pode ser qualquer um, inclusive uma variável.

Existem diversos métodos diferentes para se atingir um mesmo objetivo. Por exemplo, o mesmo resultado do ***for*** poderia ser escrito com o método upto():

```
1.upto(max)  do
    |i| puts i
end
```

Repare que não foi usada a palavra ***for***, mas um método chamado upto().
Caso se deseje fazer um incremento de 2 em 2 ou outro valor, deve-se usar o método ***step***, como a seguir:

```
for i in (1..max).step(2)  do
    puts i
end
```

E para decremento:

```
for i in (1..max).to_a.reverse  do
    puts i
end
```

OU

```
max.downto(i)  do
    |i| puts i
end
```

E para decremento de 2 em 2:

```
for i in (1..max).step(2).to_a.reverse  do
    puts i
end
```

E para percorrer os elementos de um array:

```
nomes = ['Miguel', 'Ana', 'Paulo']
 for elemento in nomes
    puts elemento
end
```

ou

```
nomes = ['Miguel', 'Ana', 'Paulo']
 nomes.each do |elemento|
    puts elemento
end
```

É interessante notar que aquilo que pode ser considerado uma facilidade em um linguagem, também pode ser considerado uma barreira. O laço *for* do C/C++ e GO são padrões usados em outras linguagens, como Java, C#, etc... Ruby simplificou o cabeçalho para a situação em que precisamos apenas incrementar de 1 em 1. Mas para outros objetivos, o cabeçalho pode parecer mais poluído para quem está aprendendo a programar.

Python

```
max = int(input('Digite o valor máximo: '))

for i in range(1,max+1):
    print(i)
```

Obs: em Python, o primeiro valor da função range é o valor inicial (inclusive ele) e o segundo valor é o final(exclusive ele). Por isso precisamos somar 1 à variável *max*. De outra forma, sem se somar 1, a contagem iria de 1 até o valor anterior à *max*.

Se quisermos contar de 2 em 2, deve-se colocar o incremento em seguida ao valor máximo:

```
for i in range(1, max+1, 2):
    print(i)
```

Se quisermos contar de forma decremental, basta colocar o terceiro valor como negativo. No exemplo abaixo, será contado de 10 a 1 (porque o segundo limite é do tipo **exclusive**), de forma descendente e de 1 em 1:

```
for i in range(max, 0, -1):
    print(i)
```

Ou

```
for i in reversed(range(1,max+1)):
    print(i)
```

Para contar na ordem descendente e de 2 em 2:

```
for i in range(max, 0, -2):
    print(i)
```

Ou

```
for i in reversed(range(1,max+1, 2)):
    print(i)
```

Também é possível usar uma variação do ***for*** para percorrer elementos de um vetor:

```
lista = [1,3,7,2,9]
for i in lista:
    print(i)
```

Laços ENQUANTO e FAÇA-ENQUANTO

No capítulo anterior, aprendemos a instrução ***para*** que é usada quando sabemos a quantidade de vezes que precisamos repetir uma sequência de instruções, como contar de 1 a 10 ou aquilo que o usuário digitar. Mas há situações em que não podemos ter certeza quantas vezes uma sequência de instruções deve ser repetidas.

Por exemplo, imagine que desejamos pedir uma senha numérica para o usuário e que a senha correta seja 123456. Se a senha estiver errada, o algoritmo deve escrever uma mensagem de erro e pedir para o usuário digitá-la novamente. Repare que este processo de mostrar a mensagem de erro e permitir que o usuário digite a senha deve ser repetida ***enquanto*** a senha estiver errada (***enquanto*** a senha for diferente de 123456).

Deste modo, não é possível determinar previamente quantas vezes o usuário errará a senha. Pode ser que ele acerte de primeira. Pode ser que acerte de segunda. Ou pode ser que ele fique tentando até cansar (claro que aqui não estamos considerando bloquear o aplicativo após a terceira tentativa errada!!!). Repare uma última coisa importante deste exemplo: se o usuário acertar de primeira, então a mensagem de erro nunca será apresentada!

O que precisamos é de uma instrução de repetição que permita que uma sequência de instruções se repitam 0 (se acertar de primeira, as instruções de aviso de erro não serão executadas) ou indefinidas vezes (até que o usuário acerte a senha). O nome desta instrução é ***enquanto***!

Vamos colocar os passos que definimos em uma ordem coerente:

- Pedir a senha para o usuário;
- ***Enquanto*** a senha for diferente de 123456 repita:
    - Apresentar a mensagem “Senha incorreta”;
    - Pedir nova senha;
- Apresentar a mensagem “Seja bem vindo”.

Traduzindo para o nosso algoritmo, teremos:

```
1.   inicio
2.      inteiro senha
3.      escreva(“Digite a sua senha”)
4.      leia(senha)
5.      enquanto(senha!=123456) faça
6.          escreva(“Senha incorreta!”)
7.          escreva(“Tente novamente: ”)
8.          leia(senha)
9.      fimenquanto
10.     escreva(“Seja bem vindo”)
11.  fim
```

Vamos entender o código: na linha 4, o algoritmo esperará o usuário digitar a senha e a armazenará o valor digitado na variável de mesmo nome. Na linha 5, a instrução ***enquanto*** fará a comparação. Se a senha for diferente de 123456, deverão ser executadas as linhas 6, 7 e 8 (que chamamos de instruções do bloco ***enquanto***). Ao fim do bloco, o algoritmo voltará para a linha 5, para fazer a comparação novamente. Apenas quando a condição for falsa, o algoritmo irá ignorar o bloco e continuar a partir da linha 10. De forma mais detalhada:

- Suponhamos que o usuário digitou 654321 na linha 4.
- Então, na linha 5, a condição é verdadeira (porque 654321 é diferente de 123456) e as instruções das linhas 6, 7 e 8 serão executadas.
- Na linha 8, o valor da variável *senha* passará a ser o novo número digitado (digamos, 0000).
- Ao alcançar a linha 9, o algoritmo retornará à linha 5 e fará a nova comparação.
- Novamente, a condição será verdadeira, porque 0000 é diferente de 123456. Por isso, as linhas 6, 7 e 8 serão executadas mais uma vez.
- Mas digamos que ao executar a linha 8, desta vez o usuário digite 123456.
- Ao alcançar a linha 9, novamente o algoritmo retornará à linha 5.
- Mas desta vez, quando a comparação for realizada, a condição será falsa, afinal 123456 não é diferente de 123456 (123456 é igual a 123456!!!).

- Como a condição é falsa, agora o algoritmo irá ignorar as instruções do bloco ***enquanto*** e continuará a executar a partir da linha 10 apresentando a mensagem "Seja bem-vindo".

Para terminar este exemplo, vamos imaginar que o algoritmo vai ser executado pela primeira vez e, logo na linha 4, o usuário digite a senha 123456. É importante reparar que, neste caso, logo na primeira comparação da linha 5, a condição já será falsa, isto é, 123456 não é diferente de 123456. Por isso, logo na primeira comparação, o algoritmo irá ignorar o bloco do laço ***enquanto*** e continuar a partir da linha 10.

É por isso que dizemos que esta instrução permite que um bloco seja executado de 0 a indefinidas vezes.

É bastante simples! A ideia é: enquanto uma determinada condição for verdadeira, faça o que está dentro do bloco.

Agora, vamos supor que queiramos repetir instruções indefinidamente, como antes, mas com uma diferença: temos certeza de que queremos que as instruções dentro do bloco sejam executadas pelo menos uma vez. Neste caso, usamos a instrução ***faça-enquanto***.

Por exemplo, imagine que queiramos fazer um algoritmo que apresente o menu:

1-Dizer "oi"
2-Dizer "Tudo bem?"
0-SAIR

Se o usuário digitar 1, o algoritmo deve dizer a mensagem "OI!" e voltar a apresentar o menu para que o usuário digite uma nova opção. Se o usuário digitar 2, o algoritmo deve dizer a mensagem "Tudo bem?" e voltar a apresentar o menu para que o usuário digite uma nova opção. Se o usuário digitar 0, o algoritmo deve terminar. Neste exemplo, conseguimos identificar que o menu deve ser apresentado pelo menos uma vez, assim como a instrução leia e as mensagens a serem exibidas. Neste caso, usaremos a instrução ***faça-enquanto***, da seguinte forma:

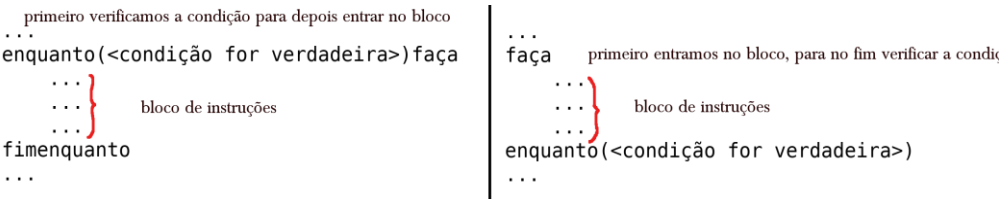
```
1.    inicio
2.          inteiro opc
3.          faça
4.                escreva(“1-Dizer oi”)
5.                escreva(“2-Dizer Tudo bem?”)
6.                escreva(“0-SAIR”)
7.                leia(opc)
8.                escolha(opc)
9.                      caso 1:
10.                           escreva(“OI!!!”)
11.                     caso 2:
12.                           escreva(“Tudo bem?”)
13.               fimescolha
14.         enquanto(opc!=0)
15.   fim
```

Neste código, repare que a comparação só é feita no fim do bloco ***faça-enquanto***. É por isso que podemos garantir que as instruções de dentro do bloco serão executadas pelo menos uma vez. Assim, quando o algoritmo começar, as instruções das linhas 1, 2, 3, 4, 5, 6 e 7 serão executas em sequência até que o algoritmo aguarde o usuário digitar um número, de acordo com o que o menu sugere. Se o usuário digitar 1, o bloco ***escolha-caso*** irá apresentar a mensagem OI, da linha 10. Uma vez que o bloco ***escolha-caso*** termine de ser executado, a condição da linha 14 será analisada. Como o valor de `opc` é 1, e 1 é diferente de 0, a condição será verdadeira e o algoritmo retornará a executar da linha 3 (começo do bloco ***faça-enquanto***). E as instruções de dentro serão executadas novamente: o menu será mostrado novamente, um novo valor será digitado na linha 7, e o bloco ***escolha-caso*** apresentará uma das duas mensagens, caso `opc` valha 1 ou 2, respectivamente.

Mas vamos supor que o usuário tenha digitado 0, desta vez. Então, o bloco ***escolha-caso*** não apresentará mensagens, porque não há um caso para 0. E, ao comparar a condição da linha 14, a comparação será falsa, porque 0 não é diferente de 0. Como a condição é falsa, desta vez o bloco não se repetirá e o algoritmo continuará após a linha 14 (e, consequentemente, chegaremos ao fim do algoritmo).

A diferença entre as instruções ***enquanto*** e ***faça-enquanto*** é que na primeira, a condição é analisada antes do bloco ser executado; e em ***faça-enquanto***, a condição só é analisada no fim do bloco (Figura 6).

Figura 6 - Diferença entre enquanto e faça-enquanto



Em resumo: tanto a instrução ***enquanto*** quanto a instrução ***faça-enquanto*** devem ser usadas se não se sabe quantas vezes uma sequência de instruções deve ser executada. Se sabemos que essa sequência deve ser executada pelo menos uma vez, usamos ***faça-enquanto***. Se é possível que esta sequência nunca seja executada, usamos ***enquanto***.

Hora de praticar.

Exercícios de enquanto e faça-enquanto

1. Faça um algoritmo que peça uma senha numérica. Em seguida, o algoritmo deverá verificar a senha. E, enquanto a senha for diferente de 123456, o algoritmo deverá escrever a mensagem "Senha inválida, tente novamente", para em seguida aguardar a nova senha ser digitada. Ao fim do algoritmo, deverá ser escrito "Senha correta. Seja bem-vindo".

2. Faça um algoritmo que peça para o usuário digitar um número inteiro entre 1 e 6. Após ser digitado, enquanto esse número for menor que 1 ou maior que 6, escreva a mensagem "valor inválido. Tente novamente". E o algoritmo deverá esperar por um novo número. Neste exemplo, se o usuário digitar um número dentro do intervalo correto de primeira, então o algoritmo não precisará escrever a mensagem de valor inválido.

3. Faça um algoritmo que conte de 1 a 6, mas no lugar de laço PARA, use ENQUANTO. Se um outro programador for dar manutenção no seu algoritmo, qual das duas opções seria mais fácil de dar manutenção, PARA ou ENQUANTO?

4. Faça um algoritmo de adivinhação. Primeiro, o algoritmo deve pedir para o usuário 1 digitar um número. Em seguida, o algoritmo deve pedir para o usuário 2 digitar um número. O algoritmo deverá continuar pedindo para o usuário 2 digitar um número enquanto o valor digitado pelo usuário 2 for diferente do valor digitado pelo usuário 1. Por fim, depois que o usuário 2 acertar o valor digitado, o algoritmo deverá escrever "Parabéns, você acertou".

5. Faça um algoritmo que apresente o menu abaixo, enquanto a opção digitada for diferente de 0.

MENU:
1-Dizer "Oi"
2-Dizer "Olá"
0-SAIR

Se o usuário digitar 1, o algoritmo deverá apresentar a mensagem “Oi”. Se o usuário digitar 2, o algoritmo deve apresentar a mensagem “Olá“. Se o algoritmo digitar 0, deve terminar.

6. Faça um algoritmo que simule um Monstrodex! Ele deve apresentar o seguinte menu:

Monstrodex:
1-Zikachu
2-Zulbassauro
3-Zharmander
0-Sair do Monstrodex

- Caso o usuário escolha a opção 1, o algoritmo deve apresentar a mensagem “Monstro elétrico da categoria rato”.
- Caso o usuário escolha a opção 2, o algoritmo deve apresentar a mensagem “Monstro de grama da categoria semente”.
- Caso o usuário escolha a opção 3, o algoritmo deve apresentar a mensagem “Monstro de fogo da categoria lagarto”.
- Caso o usuário escolha outra opção, o algoritmo deve apresentar a mensagem “Monstro não cadastrado. Há 8900 monstros! Temos que pegar!”
- Após apresentar o resultado, o algoritmo deve retornar ao menu inicial, enquanto a opção for diferente de 0.

Como é em algumas linguagens

Português Estruturado

```
inicio
    inteiro senha, opc

    escreva("Digite sua senha:")
    leia(senha)

    enquando(senha!=123456)faça
        escreva("Senha incorreta. Tente de novo")
        leia(senha)
    fimenquanto

    escreva(“Seja bem vindo!”)

    faça
        escreva("MENU")
        escreva("1-Dizer Oi")
        escreva("2-Dizer Olá")
        escreva("3-Dizer Opa!")
        escreva(“0-Sair”)
        leia(opc)

        escolha(opc)
            caso 1:
                escreva("Oi, tudo bem???")
            caso 2:
                escreva("Olá!!! Como vai?")
            caso 3:
                escreva("Opa! Fala!")
        fimescolha
    enquanto(opc!=0)
fim
```

C

```c
#include <stdio.h>

int main(){
    int senha, opc;

    printf("\nDigite sua senha:");
    scanf("%d",&senha);

    while(senha!=123456){
        printf("\nSenha incorreta. Tente de novo");
        scanf("%d",&senha);
    }

    printf("\nSeja bem vindo!");

    do{
        printf("\nMENU:");
        printf("\n1-Dizer oi");
        printf("\n2-Dizer olá");
        printf("\n3-Dizer Opa!");
        printf("\n0-Sair");
        scanf("%d",&opc);

        switch(opc){
            case 1:
                printf("\nOi, tudo bem???");
                break;
            case 2:
                printf("\nOlá!!! Como vai???");
                break;
            case 3:
                printf("\nOpa! Fala!");
                break;
        }
    }while(opc!=0);
}
```

Obs: as instruções ***while*** e ***do-while*** são muito próximas ao aprendido nas seções anteriores. Este mesmo padrão é também aplicado a linguagens como Java e C#.

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){

    int senha, opc;

    cout << "\nDigite sua senha:";
    cin >>senha;

    while(senha!=123456){
        cout << "\nSenha incorreta. Tente de novo";
        cin >> senha;
    }

    cout << "\nSeja bem vindo!";

    do{
        cout << "\nMENU:";
        cout << "\n1-Dizer oi";
        cout << "\n2-Dizer olá";
        cout << "\n3-Dizer Opa!";
        cout << "\n0-Sair";
        cin >> opc;

        switch(opc){
            case 1:
                cout << "\nOi, tudo bem???";
                break;
            case 2:
                cout << "\nOlá!!! Como vai???";
                break;
            case 3:
                cout << "\nOpa! Fala!";
                break;
        }
    }while(opc!=0);
}
```

Obs: as instruções ***while*** e ***do-while*** são muito próximas ao aprendido nas seções anteriores. Este mesmo padrão é também aplicado a linguagens como Java e C#.

Go

```
package main

import (
    "fmt"
)

func main() {
    var senha, opc int

    fmt.Print("\nDigite sua senha:")
    fmt.Scanf("%d", &senha)

    for senha != 123456 {
        fmt.Print("\nSenha incorreta. Tente de novo")
        fmt.Scanf("\n%d", &senha)
    }

    fmt.Print("\nSeja bem vindo!")

    for {
        fmt.Print("\nMENU:")
        fmt.Print("\n1-Dizer oi")
        fmt.Print("\n2-Dizer olá")
        fmt.Print("\n3-Dizer Opa!")
        fmt.Print("\n0-Sair")
        fmt.Scanf("\n%d", &opc)

        switch opc {
        case 1:
            fmt.Print("\nOi, tudo bem???")
        case 2:
            fmt.Print("\nOlá!!! Como vai???")
        case 3:
            fmt.Print("\nOpa! Fala!")
        }
        if opc == 0 {
            break
        }
    }
}
```

Obs: em Go, não há as instruções ***while*** e ***do-while***. Usa-se ***for*** para as duas situações.
-No lugar de:

```
while(<condição>)
```

-Usa-se:

```
for <condição>
```

Nota-se que o conceito é o mesmo. Muda-se apenas o nome da instrução.
Já, em ***do-while***, há uma pequena diferença em relação à lógica. Não há exatamente uma instrução igual a ***do-while***. Usa-se uma instrução ***if*** no fim do laço para INTERROMPER a repetição.

Em C:

```
do{
…
}while(<condição>);
```

A condição é verificada ao fim do bloco e, se for VERDADEIRA, o bloco é executado novamente desde a instrução ***do***.
Já, em:

```
for{
…
    if <condição> {
        break;
    }
}
```

É um pouco diferente porque se usa uma instrução **if** para SAIR da repetição, e não para CONTINUAR a repetição. Deste modo, Go usa uma lógica mais parecida com **ATÉ QUE** do que **ENQUANTO**. Isso porque em C, enquanto uma condição for verdadeira deve-se repetir. Mas no caso de Go, deve-se repetir ATÉ QUE a condição seja verdadeira (e por consequência, saia da repetição).

Por causa disso, observe que a condição em Go deve ser invertida.
Em C, a condição era:

```
while(opc!=0);
```

Porque deve CONTINUAR A REPETIR ENQUANTO *opc* for diferente de 0.

Em Go, é:

```
for{
    ...
    if (opc==0){
        break;
    }
}
```

Porque deve PARAR DE REPETIR quando *opc* for igual a 0.
A instrução break faz justamente isso: obriga o laço a parar. Ele interrompe o laço.

Ruby

```
print "Digite sua senha:"
senha = gets.chomp.to_i

while senha!=123456
    print "Senha incorreta. Tente de novo"
    senha = gets.chomp.to_i
end

puts "Seja bem vindo!"

loop do
    puts "MENU"
    puts "1-Dizer Oi"
    puts "2-Dizer Olá"
    puts "3-Dizer Opa!"
    puts "0-Sair"
    opc = gets.chomp.to_i

    case opc
    when 1
        puts "Oi, tudo bem???"
    when 2
        puts "Olá!!! Como vai?"
    when 3
        puts "Opa! Fala!"
    end

    if opc==0
        break
    end
end
```

Obs: a instrução ***while*** é muito semelhante a C e C++. Já, ***faça-enquanto*** em Ruby é mais parecido com Go e Python. Não há exatamente uma instrução igual a ***do-while***. Usa-se uma instrução **if** no fim do laço para INTERROMPER a repetição. Por causa dessa diferença, repare que a condição do **if** é inversa ao que se usa em ***do-while*** no C ou C++.

Em C, a condição era:

```
  ...
while(opc!=0);
```

Porque deve CONTINUAR A REPETIR ENQUANTO `opc` for diferente de 0. Em Ruby, é:

```
loop do
    ...
    if opc==0
        break
    end
end
```

Porque deve PARAR DE REPETIR quando `opc` for igual a 0.
A instrução break faz justamente isso: obriga o laço a parar. Ele interrompe o laço.

Python

```
senha = int(input('Digite sua senha:'))

while senha!=123456:
    senha = int(input('Senha incorreta. Tente de novo'))

print('Seja bem vindo!')

while True:
    print('MENU')
    print('1-Dizer Oi')
    print('2-Dizer Olá')
    print('3-Dizer Opa!')
    print('0-Sair')
    opc = int(input())

    match opc:
        case 1:
            print('Oi, tudo bem???')
        case 2:
            print('Olá!!! Como vai?')
        case 3:
            print('Opa! Fala!')

    if opc==0:
        break
```

Obs: a instrução ***while*** é muito semelhante a C e C++. Já, ***faça-enquanto*** é mais parecido com Go e Ruby. Não há exatamente uma instrução igual a ***do-while***. Usa-se uma instrução **if** no fim do laço para INTERROMPER a repetição. Por causa dessa diferença, repare que a condição do **if** é inversa em relação ao **do-while** do C ou C++.

Em C, a condição era:

```
  ...
while(opc!=0);
```

Porque deve CONTINUAR A REPETIR ENQUANTO `opc` for diferente de 0.

Em Python, é:

```
while True:
    ...
    if opc==0:
        break
```

Porque deve PARAR DE REPETIR quando `opc` for igual a 0.
A instrução break faz justamente isso: obriga o laço a parar. Ele interrompe o laço.

Teste de Mesa

Até o momento, aprendemos como desenvolver nossos primeiros algoritmos com instruções básicas. Vocês devem ter percebido que, mesmo com poucas instruções, nosso algoritmo pode ficar realmente desafiador para ser escrito quando muitas tarefas precisam ser representadas.

Como fazemos então para identificar se o algoritmo está realizando o que deveria?

Você pode pensar: é só criarmos um programa e testarmos! Certo. Você pode descobrir facilmente se um pequeno algoritmo faz o que deveria, executando-o no computador. Mas! Mas pode não ser tão fácil descobrir o porquê seu código não faz o que você gostaria que ele fizesse. Lembre-se: o programa faz o que você pediu, não o que você gostaria que ele fizesse! São coisas diferentes.

Para descobrirmos porquê o código se comporta diferente do que gostaríamos, até o momento temos "executado" nosso algoritmo "de cabeça". Esta seção apresenta uma maneira de formalizarmos essa execução que fazemos apenas mentalmente. Chamamos de teste de mesa (trace table).

O objetivo de se realizar o teste de mesa é entender o comportamento do algoritmo, usando uma técnica mais organizada do que a execução mental. E quando precisamos entender um algoritmo? Quando cometemos um erro de lógica que não conseguimos identificar ou quando queremos entender um algoritmo de terceiros.

O básico

Para realizarmos um teste de mesa, criamos uma tabela que possua uma coluna para a instrução, uma coluna para o que será apresentado na tela e uma coluna para cada variável do algoritmo. Nesta tabela, iremos:

- Colocar na ordem de execução, cada instrução do código;
- Podemos ignorar linhas que contenham apenas declarações sem inicialização, ou apenas sinalização de início e fim de blocos, como ***início***, ***fim***, ***fimse***, etc... (mas podem ser incluídos, caso seja considerado conveniente);

- Se a instrução for do tipo ***escreva***, colocamos na coluna "tela" a mensagem a ser apresentada;
- Se a instrução for do tipo ***leia***, o desenvolvedor deve assumir um valor a ser digitado, como se fosse o usuário. Esse valor deve estar descrito na coluna "tela";
- Se uma variável não tiver sido inicializada, podemos colocar um símbolo de interrogação (?) ou deixar em branco;
- Se uma variável receber um novo valor, marcaremos esse valor na coluna da respectiva variável;
- Se uma variável não receber um novo valor mas tiver sido inicializada, repetiremos na próxima linha, o valor da linha anterior.

Com estas regras iniciais, podemos fazer um exemplo simples. Este exemplo é simples demais e não exige a execução de um teste de mesa para entendermos o que faz. Mas é perfeito para aprendermos como fazê-lo. Considere o seguinte código:

```
1.    inicio
2.         inteiro x, dobro
3.         escreva("Digite um valor")
4.         leia(x)
5.         dobro<-x * 2
6.         escreva("O resultado é ", dobro)
7.    fim
```

Podemos ignorar as linha 1, 2 e 7 e começarmos pela linha 3. Iremos, agora, registrar cada linha, na sequência em que será executada, como abaixo:

Instrução	tela	x	dobro
escreva("Digite um valor")	Digite um valor	?	?
leia(x)	4	4	?
dobro <- x * 2		4	8
escreva("O resultado é ", dobro)	O resultado é 8	4	8

Repare que:

- Ao prepararmos a tabela, colocamos as colunas *x* e *dobro*, pois representam as variáveis do algoritmo que rastrearemos.

- A primeira instrução era do tipo ***escreva***, então apresentamos a mensagem na coluna "tela". Isso serve para verificarmos se a mensagem está sendo corretamente apresentada, pois nos força a verificar como estamos concatenando as partes do texto (caso haja concatenação). Serve também, para entendermos o que está sendo pedido para o usuário, por exemplo.

- Ainda na primeira linha, os valores das variáveis *x* e *dobro* são indefinidas, pois não inicializamos as variáveis. Há linguagens como Java, que automaticamente inicializam as variáveis numéricas com o valor 0. Mas há linguagens que não fazem isso, como C/C++, em que o valor inicial é um lixo de memória. Por isso, se o algoritmo for criado sem uma linguagem de programação previamente pensada, consideramos indefinido o valor de uma variável não inicializada.

- Na linha 2 da tabela, há uma instrução *leia(x)* em que o desenvolvedor deverá assumir um valor digitado. Aqui, considerei o número 4, mas poderia ser qualquer outro. E, ao mesmo tempo, a coluna *x* assume o valor 4 a partir deste ponto do código. É interessante, porque quando olhamos para a tabela, passamos a entender exatamente a partir de que momento que uma variável assumiu algum valor.

- Na linha 3 da tabela, a variável *dobro* assume um valor. Para realizar o cálculo (*x*2*), basta o desenvolvedor observar na linha anterior, qual o valor de *x*. O valor está facilmente acessível para ele. Pode parecer irrelevante neste exemplo, mas imagine uma situação em que o cálculo é mais complexo com diversas variáveis sendo usadas no cálculo. Quando o usuário realizar o cálculo da expressão aritmética (para testar se o resultado é o esperado), todos os valores das variáveis estarão facilmente acessíveis, bastando olhar para a linha anterior.

- Na linha 4 da tabela, a mensagem final é apresentada.

Neste ponto, você pode estar pensando: dará muito trabalho fazer um teste de mesa para um sistema completo! E você está certo. Não se aplica teste de mesa a

um sistema inteiro. Usa-se para pequenos trechos de código que precisamos entender. Acredite, há situações em que mesmo um código pequeno de poucas linhas pode ser muito difícil de se entender.

Além disso, as ferramentas de programação possuem mecanismos que permitem rastrear o valor das variáveis e executar um código linha a linha. Na prática, iremos sempre fazer testes de mesa a partir da própria interface gráfica (IDE) da ferramenta de programação.

Aqui, estamos aprendendo a formalizar esse teste manualmente para que possamos compreender o comportamento de nossos algoritmos. É um excelente exercício para visualizarmos melhor os algoritmos que escrevemos. Também é uma boa forma de analisarmos o que fazemos. Pense bem: quando terminamos um exercício das listas, é comum ficarmos inseguros no começo. As vezes, carregamos a insegurança até o próximo dia de aula para tirarmos a dúvida com o professor. Usando teste de mesa, podemos ficar mais seguros sobre a resposta de nossos exercícios se o aplicarmos para confirmar se ele faz o que deveria.

Teste de mesa com SE-SENÃO

Veremos agora como realizar um teste de mesa em que há instruções ***se-senão***. Considere o algoritmo abaixo:

```
1.    inicio
2.         inteiro idade
3.         escreva(“Digite sua idade”)
4.         leia(idade)
5.         se(idade>=18)então
6.              escreva(“Você pode dirigir“)
7.         senão
8.              escreva(“Você não pode dirigir”)
9.         fimse
10.        escreva(“Até mais”)
11.   fim
```

Neste caso, podemos perceber que se executarmos o código apenas uma vez, teremos que escolher qual situação será testada (maior de idade ou menor de

idade). Em casos assim, nos perguntamos o que fazer? Fazemos o teste de mesa duas vezes, uma para cada situação? Ou apenas uma vez aleatoriamente?

Em muitos casos, precisamos fazer o teste de mesa apenas para descobrirmos o comportamento do algoritmo em uma situação específica. Por exemplo, se soubermos de antemão que um programa não funciona quando digitamos 20, mas não conseguimos entender porque não funciona, então fazemos o teste de mesa direcionado para essa situação.

Por outro lado, se queremos fazer um teste de mesa apenas para entender o que ele faz, pode ser interessante realizar testes de mesa para cada caminho do algoritmo, se isso for viável (porque mesmo algoritmos pequenos podem gerar muitos caminhos diferentes, tornando impraticável a realização de um teste de mesa para cada situação).

Aqui, para entendermos como funciona o teste de mesa, faremos um para cada caso:

Instrução	**tela**	**idade**
EXECUÇÃO 1		
escreva(“Digite sua idade”)	**Digite sua idade**	**?**
leia(idade)	**18**	**18**
se(idade>=18)então (V)		**18**
escreva(“Você pode dirigir”)	**Você pode dirigir**	**18**
escreva(“Até mais”)	**Até mais**	**18**
EXECUÇÃO 2		
escreva(“Digite sua idade”)	**Digite sua idade**	**?**
leia(idade)	**9**	**9**
se(idade>=18)então (F)		**9**
escreva(“Você não pode dirigir”)	**Você não pode dirigir**	**9**
escreva(“Até mais”)	**Até mais**	**9**

Na primeira execução, consideramos que o usuário digitou 18. Repare que a linha de execução do ***se*** possui uma marcação (V) do lado. Esta anotação que o

desenvolvedor pode fazer é útil para que ele revise a execução mais tarde e rapidamente identifique o resultado da condição.

Como indicado na seção anterior, nós podemos ignorar instruções de marcação de bloco como ***else*** e ***fimse***. Mas poderia ser descrito, também. O importante é manter a coerência. Se o desenvolvedor decidir desconsiderar estas instruções, então que o faça durante todo o teste de mesa para manter a leitura padronizada.

Na segunda execução, o desenvolvedor assumiu que o usuário digitaria 9 e o resultado da condição foi falsa (por isso a marcação do F ao lado). Em razão da condição ser falsa, a próxima instrução a ser executada foi ***escreva*** do bloco ***senão***, seguido pela mensagem "Até mais".

Já, um ***se-senão*** encadeado pode ser representado no teste de mesa como abaixo:

```
    inicio
1.      inteiro idade
2.      escreva("Digite sua idade")
3.      leia(idade)
4.      se(idade>70)então
5.        escreva("Não precisa votar")
6.      senão se(idade>=18)então
7.        escreva("Você precisa votar")
8.      senão
9.        escreva("Você não pode votar")
10.     fimse
11.     escreva("Até mais")
12. fim
```

Neste exemplo, estamos ignorando o voto opcional de pessoas com 16 e 17 anos para simplificar o exemplo.

Instrução	tela	idade
EXECUÇÃO 1		
escreva(“Digite sua idade”)	Digite sua idade	?
leia(idade)	80	80
se(idade>70)então (V)		80
escreva(“Não precisa votar”)	Não precisa votar	80
escreva(“Até mais”)	Até mais	80
EXECUÇÃO 2		
escreva(“Digite sua idade”)	Digite sua idade	?
leia(idade)	18	18
se(idade>70)então (F)		18
se(idade>=18)então (V)		18
escreva(“Você precisa votar”)	Você precisa votar	18
escreva(“Até mais”)	Até mais	18
EXECUÇÃO 3		
escreva(“Digite sua idade”)	Digite sua idade	?
leia(idade)	9	9
se(idade>70)então (F)		9
se(idade>=18)então (F)		9
escreva(“Você não pode votar”)	Você não pode votar	9
escreva(“Até mais”)	Até mais	9

Na primeira execução, a primeira condição foi verdadeira, portanto, foi executada a instrução de dentro de seu bloco e os demais blocos ***senão*** foram pulados. Assim, quando a condição é verdadeira seu respectivo bloco é executado e o algoritmo continua após o ***fimse***.

Na segunda execução, a primeira condição foi falsa. Isso fez com que a execução pulasse o respectivo bloco e testasse a próxima condição do ***se-senão***

encadeado. Esta segunda condição é verdadeira. Assim, a instrução deste bloco foi executada e o algoritmo seguiu a partir do ***fimse***.

Na terceira execução, a primeira condição foi falsa. A execução pulou para a segunda condição que também foi falsa, fazendo a execução pular para o terceiro bloco e executar a sua respectiva instrução.

Teste de mesa com ESCOLHA-CASO

O teste de mesa para a instrução ***escolha-caso*** é exemplificado com base no código abaixo:

```
1. inicio
2.     inteiro opc
3.     escreva(“MENU:”)
4.     escreva(“1-DIZER OI”)
5.     escreva(“2-DIZER OLÁ”)
6.     leia(opc)
7.     escolha(opc)
8.          caso 1:
9.               escreva(“OI”)
10.         caso 2:
11.              escreva(“OLÁ”)
12.    fimescolha
13.    escreva(“TCHAU”)
14.fim
```

Instrução	tela	opc
EXECUÇÃO 1		
escreva(“MENU:”)	MENU:	?
escreva(“1-DIZER OI”)	1-DIZER OI	?
escreva(“1-DIZER OLÁ”)	2-DIZER OLÁ	?
leia(opc)	1	1
escolha(opc)		1
caso 1: (V)		1
escreva(“OI”)	OI	1
escreva(“TCHAU”)	TCHAU	1

EXECUÇÃO 2		
escreva(“MENU:”)	**MENU:**	**?**
escreva(“1-DIZER OI”)	**1-DIZER OI**	**?**
escreva(“1-DIZER OLÁ”)	**2-DIZER OLÁ**	**?**
leia(opc)	**2**	**2**
escolha(opc)		**2**
caso 1: **(F)**		**2**
caso 2: **(V)**		
escreva(“OLÁ”)	**OLÁ**	**2**
escreva(“TCHAU”)	**TCHAU**	**2**

Neste teste de mesa, repare que optei por escrever a instrução ***escolha(opc)*** para que seja fácil identificar de qual variável se tratam os casos. O comportamento das opções é muito semelhante ao do ***se-senão*** encadeado. Caso a primeira possibilidade seja verdadeira, executa-se o seu bloco e o algoritmo será pulado para a instrução seguinte à ***fimescolha***. Do contrário, cada opção será testada até que se encontre a primeira verdadeira ou o próprio fim da instrução.

Teste de mesa com ENQUANTO

Considere o código abaixo, que usamos para aprender esse laço:

```
1.  inicio
2.      inteiro senha
3.      escreva(“Digite a sua senha”)
4.      leia(senha)
5.      enquanto(senha!=123456)faça
6.          escreva(“Senha incorreta!”)
7.          escreva(“Tente novamente: ”)
8.          leia(senha)
9.      fimenquanto
10.     escreva(“Seja bem vindo”)
11.  fim
```

O teste de mesa para este algoritmo pode ser feito como a seguir:

Instrução	tela	senha
escreva("Digite a sua senha")	Digite a sua senha	?
leia(senha)	0000	0000
enquanto(senha!=123456)faça (V)		0000
escreva("Senha incorreta! Tente novamente: ")	Senha incorreta! Tente novamente:	0000
leia(senha)	1111	1111
enquanto(senha!=123456)faça (V)		1111
escreva("Senha incorreta! Tente novamente: ")	Senha incorreta! Tente novamente:	1111
leia(senha)	123456	123456
enquanto(senha!=123456)faça (F)		123456
escreva("Seja bem vindo")	Seja bem vindo	123456

Aqui, eu, desenvolvedor, resolvi executar o algoritmo com duas senhas incorretas (não corretas) para depois escolher uma senha válida. Então:

- Da primeira vez que o algoritmo pediu para eu digitar a senha (ainda fora do laço), eu escolhi 0000.
- Na terceira linha da tabela, a condição foi verdadeira (porque estamos testando se a senha é INCORRETA!). Deu nó? Lembre-se: o objetivo deste ***enquanto*** é ficar executando ENQUANTO a senha digitada for INCORRETA! Isto é, diferente de 123456. O ***enquanto*** deste exemplo é como uma barreira que impede que o algoritmo continue enquanto a senha **não** for **correta**. Porque, enquanto a senha for "não correta", vai ficar preso no laço. É isso que significa **!=**. A exclamação significa "não". Então **!=** significa "não igual", que é o mesmo que "diferente".
- Como a condição foi verdadeira, entramos no bloco do ***enquanto***, uma nova instrução ***escreva*** pede para digitar de novo e um novo valor é esperado ao se executar o ***leia***.

- Neste ponto escolhi colocar mais uma senha inválida (ou não correta, se achar mais fácil) para entender o que o código faz (1111).
- Assim que o usuário digitou o novo valor, voltamos para o início do ***enquanto*** para testar a condição novamente. Como escolhi uma senha incorreta (isto é, diferente de 123456 ou "não igual" a 123456), então a condição foi verdadeira e entramos novamente no laço.
- O algoritmo pede mais uma vez para digitar um valor. Desta vez, escolhi digitar a senha correta.
- O algoritmo sobe novamente para o início do ***enquanto*** e testa a condição.
- Desta vez a condição é falsa. Então, o algoritmo deve pular o bloco ***enquanto*** e continuar suas instruções. E é isso que queríamos. O nosso ***enquanto*** terminou assim que a senha correta foi digitada. Se deu nó na cabeça, isso acontece por causa do "jogo de palavras". Digitar uma senha incorreta, nesse exemplo, implica uma condição verdadeira. Aqui está o nó. A senha é incorreta sob o ponto de vista do requisito do sistema (não deixe continuar se a senha é incorreta). Já, a condição verdadeira é sob o ponto de vista do que se deve REPETIR quando um valor inválido é digitado (peça a senha novamente ENQUANTO a senha não é correta).
- Por fim, a instrução ***escreva*** de "Seja bem vindo" é executada.

Você deve ter percebido que fazer teste de mesa para algoritmos que possuem laço pode deixar o teste realmente grande. Então, a pergunta é: até quando devo ficar testando o laço? Ou, com o mesmo significado: quantas vezes devo executar valores em que a condição do ***enquanto*** seja verdadeira?

Lembre-se que o objetivo do teste de mesa é entender o algoritmo. Então a resposta é simples: enquanto for necessário para que você entenda o código (seja porque você está tentando descobrir um erro de lógica, seja para entender um algoritmo de terceiros).

Teste de mesa com FAÇA-ENQUANTO

Dado o código abaixo:

```
1.  inicio
2.      inteiro opc
3.       faça
4.           escreva("1-Continuar a ver esse menu")
5.           escreva("Outro número para sair")
6.           leia(opc)
7.       enquanto(opc==1)
8.       escreva("Tchau!")
9.  fim
```

Segue o teste de mesa para este algoritmo.

Instrução	tela	opc
escreva("1-Continuar a ver esse menu")	1-Continuar a ver esse menu	
escreva("Outro número para sair")	Outro número para sair	
leia(opc)	1	1
enquanto(opc==1) (V)		1
escreva("1-Continuar a ver esse menu")	1-Continuar a ver esse menu	1
escreva("Outro número para sair")	Outro número para sair	1
leia(opc)	1	1
enquanto(opc==1) (V)		
escreva("1-Continuar a ver esse menu")	1-Continuar a ver esse menu	1
escreva("Outro número para sair")	Outro número para sair	1
leia(opc)	0	0
enquanto(opc==1) (F)		
escreva("Tchau")	Tchau	0

Um detalhe a se observar é que o marcador ***faça*** não está incluído no teste de mesa. Isso porque estamos seguindo a regra de não usarmos marcadores de início e fim de bloco se eles não possuírem condições, assim como declaração de variáveis.

Mas nada impede do testador incluir estes marcadores se considerar conveniente. O objetivo do teste de mesa é ajudar a entender o código e sua lógica. Diferentes fontes de aprendizado ensinam o teste de mesa com ligeiras diferenças. Por exemplo, há quem não use a coluna "tela". E, realmente, ela pode ser eliminada se não for conveniente. Afinal, fica complicado apresentar o que aparecerá na tela se o sistema tiver interface gráfica.

Da mesma forma, é possível simplificar o teste de mesa enumerando as linhas do algoritmo e descrevendo apenas o respectivo número na coluna instrução.

Como o propósito do teste de mesa, neste livro, é apresentar o estado do algoritmo de forma mais completa durante sua execução, optei por apresentar um modelo de tabela em que precisamos escrever mais em prol da visualização.

Teste de mesa com PARA

O laço ***para*** é interessante porque precisamos "desmembrá-lo". Lembre-se que o laço ***para*** é um ***enquanto*** com uma notação mais clara, como na tabela a seguir:

```
1.    inicio                          inicio
2.       inteiro i                         inteiro i
3.    para(i<-0;i<3;i<-i+1)faça            i<-0
4.        escreva(i)                        enquanto(i<3)faça
5.    fimpara                                   escreva(i)
6.    escreva("FIM")                            i<-i+1
fim                                        fimenquanto
                                           escreva("FIM")
                                      fim
```

Quando se diz que a notação do ***para*** é mais clara, queremos dizer que é possível entender o comportamento do laço de repetição apenas observando o cabeçalho do ***para***. Repare na coluna da direita que, para entendermos o seu comportamento de repetição, precisamos olhar para três linhas (a inicialização da variável `i`, a condição do ***enquanto*** e o incremento no fim do bloco) e isso se torna mais inconveniente quando o bloco do laço ***enquanto*** for grande.

Deste modo, para representarmos um laço ***para*** em um teste de mesa, precisamos tomar o cuidado de lembrar que a inicialização da variável ocorre apenas uma vez e que o incremento (ou decremento) ocorre apenas no fim do bloco. Assim, o teste de mesa do ***para*** se torna igual ao do ***enquanto***.

Segue o teste de mesa do código que possui o laço ***para***:

Instrução	tela	i
i← 0		0
i<3 (V)		0
escreva(i)	0	0
i <- i + 1		1
i<3 (V)		1
escreva(i)	1	1
i <- i + 1		2
i<3 (V)		2
escreva(i)	2	2
i <- i + 1		3
i<3 (F)		3
escreva("FIM")		

No exemplo:

1. A primeira instrução representada foi a inicialização da variável contadora *i*.

2. Em seguida, foi verificada a condição do cabeçalho do ***para***. Como a condição é verdadeira, então seu bloco deverá ser executado.

3. Executa-se a instrução ***escreva(i)*** e o valor 0 é apresentado na tela.

4. Apenas quando todas as instruções do bloco ***para*** são executadas (no caso, só temos uma instrução), é que o incremento da variável contadora é incrementado, de acordo com o cabeçalho.

5. Uma vez incrementada a variável, a condição deve ser testada novamente (no algoritmo, voltamos a execução para o cabeçalho).

6. A condição é verdadeira, e repete-se novamente incrementando a variável (3, depois 4), até que a condição seja falsa e o ***para*** termine.

Há dois pontos importantes a se reparar: (i) não foi necessário representar a palavra ***para*** porque ela está implícita; (ii) o teste de mesa de um laço ***para*** pode ficar realmente grande a ponto de ser impraticável executá-lo por completo. O que fazer nessa situação?

Como o objetivo do teste de mesa é entender o comportamento do algoritmo, podemos: repeti-lo apenas o suficiente para entendermos a lógica; diminuir propositalmente o intervalo da condição para entendermos o que faz em um laço menor; mas se o objetivo é descobrir um valor de uma variável que é alterado dentro do laço e precisamos testar o laço completo, podemos usar os recursos da IDE para rastrear o valor da variável durante a execução e fazer o teste de mesa na própria ferramenta de programação.

Teste de mesa com Expressões Compostas

Quando encontramos expressões mais complexas em que misturamos vários operadores relacionais, lógicos e aritméticos, podemos usar o teste de mesa para nos auxiliar na resposta.

Por exemplo, imagine que você tem a seguinte condição:

```
…
se(a+3*4 > b+4*2 || x!=0 && b>4)então
…
```

Podemos usar o teste de mesa realizando o passo a passo da expressão. Imagine que neste ponto do código, temos os seguintes valores das variáveis: `a <- 2`, `b <- 5` e `x <- 1`. Podemos representar a expressão como segue:

Instrução	tela	a	b	x
...		2	5	1
se(a+3*4 > b+4*2 || x!=0 && b>4)então se(2+3*3 > 4+4*2 || 1!=0 && 5>4)então se(11 > 12 || 1!=0 && 5>4) então se(F || V && V)então se(F || V)então		2	5	1

se(V)então (V)				
...		**2**	**5**	**1**

O que fizemos acima foi resolver a expressão, seguindo a regra de precedência da matemática: primeiro as expressões aritméticas, depois expressões relacionais, depois expressões lógicas. Dentro das expressões aritméticas, primeiro a multiplicação e divisão. Entre as expressões relacionais, primeiro not, depois E(&&), depois OU(||).

Então, na primeira linha, resolvi substituir as variáveis por seus valores naquele momento, para facilitar a minha visualização. Em seguida, realizei apenas as operações aritméticas. Depois, resolvi as operações relacionais (com os símbolos de >, !=). Realizei as operações lógicas, primeiro com o && (por preceder ||) e, na próxima linha, o ||. E, por fim, apresentei o resultado.

Essa solução por partes foi feita para me ajudar a solucionar a expressão com mais segurança e entender o que está acontecendo. Você pode resolver a mesma expressão com menos ou mais linhas, dependendo da sua necessidade.

Agora é hora de praticar. Lembre-se que o tempo investido nos exercícios a seguir não irão apenas fazer com que aprenda a realizar teste de mesa, mas servirá para você adquirir ainda mais familiaridade com a dinâmica do fluxo entre as instruções e a lógica dos algoritmos. Com isso, você treinará o pensamento de algoritmos. Aos poucos, perceberá que não precisará mais traduzir pequenos trechos do programa do Português para um algoritmo porque a solução virá na forma de algoritmo.

Exemplo: quando perceber que precisará repetir uma sequência de instruções, não pensará desta forma “tenho que repetir isso”, mas pensará nessas instruções dentro de um laço automaticamente. E isso é fantástico!

Portanto, não tente fazer os exercícios com pressa.

Exercícios de Teste de Mesa

1. Faça o teste de mesa para o algoritmo abaixo:

```
inicio
      inteiro x, dobro
      escreva(“Digite um valor”)
      leia(x)
      dobro<-x * 2
      escreva(“O resultado é “, dobro)
fim
```

2. Faça o teste de mesa para o algoritmo abaixo:

```
inicio
      inteiro x, res1, y, res2
      escreva(“Digite um valor”)
      leia(x)
      res1<-x * x
      escreva(x, “ ao quadrado é “, res1)
      escreva(“Digite outro valor”)
      leia(y)
      res2<-y*10
      escreva(y, “ vezes 10 é “, res2)
fim
```

3. Faça o teste de mesa para o algoritmo abaixo:

```
inicio
      inteiro x, res
      escreva(“Digite um valor”)
      leia(x)
      res<-x * x
      escreva(x, “ ao quadrado é “, res)
      escreva(“Digite outro valor”)
      leia(x)
      res<-x*10
      escreva(x, “ vezes 10 é “, res)
fim
```

4. Faça o teste de mesa para o algoritmo abaixo:

```
inicio
      inteiro i
      escreva(“Digite um valor entre 1 e 3”)
      leia(i)
      enquanto(i<1 || i>3)faça
            escreva(“Valor inválido. Tente novamente”)
            leia(i)
      fimenquanto
      escolha(i)
            caso 1:
                  escreva(“Você se contenta com pouco”)
            caso 2:
                  escreva(“Você procura o equilíbrio”)
            caso 3:
                  escreva(“Você quer o que puder”)
      fimescolha
fim
```

5. Faça o teste de mesa para o algoritmo abaixo. O símbolo ! representa negação. Por exemplo !(i>5) significa se “i não for maior que 5”, que é o mesmo que escrever (i<=5):

```
inicio
      inteiro i
      escreva(“Digite um valor entre 1 e 3”)
      leia(i)
      enquanto(!(i>=1 && i<=3))faça //é interessante
desmembrar a expressão na tabela, como na seção
anterior
            escreva(“Valor inválido. Tente novamente”)
            leia(i)
      fimenquanto
      escreva(“Eita!”)
fim
```

6. Faça o teste de mesa para o algoritmo abaixo:

```
inicio
       inteiro i
       faça
              escreva("Digite um número entre 1 e 3")
              leia(i)
              escolha(i)
                caso 1:
                     escreva("Você se contenta com pouco")
                caso 2:
                     escreva("Você procura o equilíbrio")
                caso 3:
                     escreva("Você quer o que puder")
                caso padrão:
                     escreva("Nah... valor inválido")
              fimescolha
       enquanto(i<1 || i>3)
fim
```

7. Compare o exercício 6 com o exercício 4. Aí está parte da beleza da programação! Infinita diversidade em infinitas combinações!

8. Faça o teste de mesa para o algoritmo abaixo:

```
inicio
      inteiro i
      para(i<-0;i<4;i<-i+1)faça
            escreva(i)
      fimpara
fim
```

9. Faça um algoritmo que peça uma temperatura em graus célsius e apresente seu respectivo valor em farenheits. O cálculo de conversão é: F= (9*C/5) + 32. Em seguida, faça um teste de mesa.

10. Faça um teste de mesa para o algoritmo abaixo e escreva o que ele faz:

```
inicio
      inteiro x,i
      leia(x)
      para(i<-0;i<x;i<-i+1)faça
            escreva("#")
      fimpara
fim
```

11. Faça um teste de mesa para o algoritmo abaixo e escreva o que ele faz:

```
inicio
      inteiro x, y, i,r
      r<-1
      escreva(“Digite um número:”)
      leia(x)
      escreva(“Digite outro número:”)
      leia(y)
      para(i<-0;i<y;i<-i+1) faça
            r<-r * x
      fimpara
      escreva(“Resultado: ”, r)
fim
```

Capítulo 2: Variáveis Compostas

Neste ponto, vimos praticamente todas as instruções básicas que são comuns a muitas linguagens de programação, como C, C++, C#, Java, Visual Basic, Pascal e diversas outras. Com as instruções que vimos, conseguimos fazer algoritmos de diversos tamanhos e complexidades.

A partir de agora, veremos alguns recursos que nos ajudam a organizar o código e os dados que precisamos manipular dentro do código. Vamos começar agrupando variáveis que representam um grupo em uma única variável. Por isso o nome "compostas"!

Variáveis Compostas HOMOGÊNEAS

Nesta seção, aprenderemos a como agrupar variáveis de um mesmo tipo. Por isso o nome "homogêneas"!

Vetores

Imagine que você precise armazenar a nota de um aluno, como fizemos em outro capítulo. Isso é fácil! Usamos uma variável. Mas, e se precisarmos armazenar quatro notas e apresentar a média? Podemos fazer como em solução anterior: usar uma variável para armazenar a nota da vez, uma variável para armazenar a somatória e, depois que todos os valores forem digitados, dividimos a somatória por quatro e apresentamos na tela, como abaixo:

```
1. inicio
2.     real nota, somatoria, media
3.     inteiro i
4.     somatoria <- 0
5.     para(i <- 0; i < 4; i <- i + 1)faça
6.         escreva("Digite a nota: ")
7.         leia(nota)
8.         somatoria <- somatoria + nota
9.     fimpara
10.    media <- somatoria/4
11.    escreva(media)
12. fim
```

O problema é: e se quisermos manter as notas digitadas armazenadas para usá-las depois? Por exemplo, e se quisermos apresentar as notas digitadas após a média ser apresentada? Do jeito que o código está seria impossível porque estamos usando apenas uma variável para armazenar as notas e quando um valor for atribuído a esta variável, o valor anterior se perderá (porque um valor sobrescreve o outro).

Então, uma saída bastante intuitiva seria criar variáveis para as quatro notas. Mas, neste caso, não poderíamos usar o laço **para**, para pedir para o usuário digitar a nota. Pense bem: como eu faria para o algoritmo usar a variável que eu quero para cada volta no laço? Repare que no exemplo anterior, apenas a variável *nota* está sendo usada.

Bom, até é possível usar o laço ***para***, apenas não seria prático! Veja o exemplo abaixo:

```
1.    inicio
2.        real nota1, nota2, nota3, nota4, media
3.        inteiro i
4.        para(i< - 0;i<4;i <- i+1)faça
5.            escreva(“Digite a nota: “)
6.            escolha(i)
7.                caso 0:
8.                    leia(nota1)
9.                caso 1:
10.                   leia(nota2)
11.               caso 2:
12.                   leia(nota3)
13.               caso 3:
14.                   leia(nota4)
15.           fimescolha
16.       fimpara
17.       media <- (nota1 + nota2 + nota3 + nota4)/4
18.       escreva(media)
19.       escreva(“As notas são: “)
20.       escreva(nota1)
21.       escreva(nota2)
22.       escreva(nota3)
23.       escreva(nota4)
24.   fim
```

Seria muito mais simples pedir para digitar cada nota diretamente:

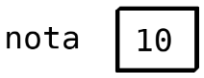
```
1.    inicio
2.        real nota1, nota2, nota3, nota4, media
3.        escreva(“Digite a nota: “)
4.        leia(nota1)
5.        escreva(“Digite a nota: “)
6.        leia(nota2)
7.        escreva(“Digite a nota: “)
8.        leia(nota3)
9.        escreva(“Digite a nota: “)
10.       leia(nota4)
11.       media <- (nota1 + nota2 + nota3 + nota4)/4
12.       escreva(media)
13.       escreva(“As notas são: “)
14.       escreva(nota1)
15.       escreva(nota2)
16.       escreva(nota3)
17.       escreva(nota4)
18.   fim
```

Perfeito! Resolvemos um problema. Mas temos outro problema! E se quisermos digitar as notas de todos os alunos de uma turma? Imagine uma turma de 40 alunos! Ou 100!

Neste caso, seria muito interessante poder usar uma variável que armazenasse mais de um valor ao mesmo tempo. Então, seria uma variável que se parece menos com uma caixa em que só cabe um valor e se pareça mais com um gaveteiro cheio de gavetas em que cada gaveta armazena um valor!

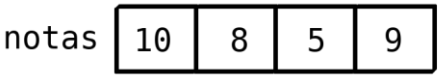
Uma variável como as que a gente usou até o momento armazena apenas um valor (Figura 7).

Figura 7 - Variável simples



A variável para nosso problema, armazena mais de um valor, em posições diferentes!

Figura 8 - Variável composta homogênea (vetor)



Esse tipo de variável é chamada de variável composta homogênea (Figura 8). Ela é composta porque pode armazenar mais de um valor. E é homogênea porque os valores a serem armazenados precisam ser sempre do mesmo tipo. Nesse caso específico, a variável também é chamada de vetor (ou *array*, ou matriz de uma dimensão). O importante é que chamaremos estas variáveis de **vetores** a partir deste ponto.

As perguntas que a gente deve fazer agora são: como declaro uma variável assim? Como eu faço para colocar um valor em cada posição diferente?

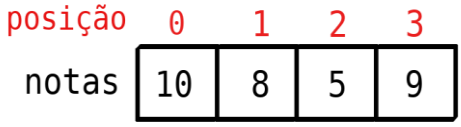
Para declarar um vetor, usaremos uma notação semelhante a C/C++: declaramos a variável normalmente, mas com o tamanho entre colchetes. Como tamanho, entende-se a quantidade de valores (ou elementos) que queremos que este vetor armazene. Então, se quisermos criar um vetor que armazene 4 notas, faremos a declaração como segue:

```
real notas[4]
```

Simples!

Agora precisamos saber como colocar um valor em cada posição deste vetor. Para isso, vamos entender como as posições são nomeadas. A primeira posição de um vetor é 0, a segunda posição é 1 e assim por diante (Figura 9).

Figura 9 – Posições de uma variável composta homogênea (vetor)



Uma vez que compreendemos esta estrutura, fica fácil aprender como colocar um valor no vetor:

```
notas[0] <- 10
notas[1] <- 8
notas[2] <- 5
notas[3] <- 9
```

Agora vem a parte divertida! Vamos voltar ao problema que tínhamos. Quando não conhecíamos os vetores e tentamos usar uma variável para cada nota, tínhamos um problema: não é prático usar um laço ***para*** porque a cada volta do laço, precisaríamos usar uma variável diferente.

Mas agora, temos uma única variável! O nome da variável é sempre o mesmo! O que muda é o número que está entre colchetes. Chamamos este número de **índice**.

Opa, se é um número entre colchetes que VARIA, podemos usar uma VARIÁVEL no índice! A variável contadora do laço ***para***! Assim, uma solução para o nosso problema é:

```
1.    real notas[4], media
2.    inteiro i
3.    para(i< - 0; i < 4; i <- i + 1)faça
4.        escreva("Digite a nota")
5.        leia(notas[i])
6.    fimpara
```

Agora, quando o valor de *i* for 0, no momento em que a linha 5 for executada, o valor digitado pelo usuário será armazenado na posição 0 do vetor notas. Quando o valor de *i* for 1, a linha 5 armazenará o valor digitado pelo usuário na posição 1. E assim por diante!

A beleza deste código é que se quisermos alterá-lo para 20 notas, as modificações serão pequenas:

```
1.    real notas[20], media
2.    inteiro i
3.    para(i< - 0; i < 20; i <- i + 1)faça
4.        escreva("Digite a nota")
5.        leia(notas[i])
6.    fimpara
```

Mudamos apenas os valores na declaração e na condição do laço ***para***.

Agora faremos um algoritmo que pede 50 notas, calcula a média e apresenta média e as notas:

```
1.    inicio
2.        real notas[50], media, somatoria
3.        inteiro i
4.        somatoria <- 0
5.        para(i <- 0; i < 50; i <- i + 1)faça
6.            escreva(“Digite a nota”)
7.            leia(notas[i])
8.            somatoria <- somatoria + notas[i]
9.        fimpara
10.       media <- somatoria/50
11.       escreva(“Média: “, media)
12.       para(i <- 0; i < 50; i <- i + 1)faça
13.           escreva(“Nota: ”, notas[i])
14.       fimpara
15.   fim
```

Repare que, para realizar o cálculo da média, continua sendo mais prático criar uma variável de somatória. Um recurso não elimina o outro.

Agora é hora de praticar.

Exercícios de Vetores

1. Faça um programa que peça 6 valores inteiros e os apresente na tela logo após serem digitados.

2. Faça um programa que peça 6 valores inteiros e, depois, apresente-os na tela.

Observe os exercícios 1 e 2 e responda: no primeiro exercício, era necessário o uso de vetores? Se você fez o primeiro exercício com vetores, tente fazer sem vetores. Se fez o primeiro exercício sem vetores, tente fazer com vetores.

Para os próximos exercícios, use vetor para armazenar os valores digitados pelo usuário.

3. Faça um programa que peça 5 números reais. Depois, o programa deve apresentar o número maior.

4. Faça um programa que peça 5 números reais. Depois, o programa deve apresentar o número menor.

5. Faça um programa que peça 5 números inteiros e, no final, troque a ordem do primeiro número com a do último número digitado.

6. Faça um programa que peça 6 números inteiros e, depois, apresente-os na tela na forma invertida ao que foi digitado. Exemplo, se foi digitado 1 3 2 4 5 9, deve ser apresentado 9 5 4 2 3 1.

7. Faça um programa que peça 6 valores inteiros, mas o primeiro valor deve ser colocado na última posição do vetor, o segundo valor deve ser colocado na penúltima posição e assim por diante.

8. Faça um programa que peça 6 valores inteiros. Após digitados os valores, a ordem deve ser trocada de forma que. o primeiro valor deve ser colocado na última posição do vetor, o segundo valor deve ser colocado na penúltima posição e assim por diante.

OBS: repare que este exercício é diferente do anterior: no exercício 7, os valores devem ser colocados nas posições invertidas logo após cada valor ser digitado. No exercício 8, os valores devem ser trocados apenas depois que todos eles forem digitados.

Desafio 1: faça um programa que peça 5 números reais. Depois, o programa deve apresentar a média. Por último, o programa deve procurar qual número digitado é o mais próximo da média e apresentá-lo na tela.

Desafio 2: faça um programa que peça 5 números inteiros. Após digitados, o programa deve ordenar os valores no vetor e apresentá-los na tela em ordem crescente.

Como é em algumas linguagens

Português Estruturado

```
inicio
    real notas[4], media, somatoria
    inteiro i
    somatoria <- 0

    PARA(i <- 0; i < 4; i <- i + 1)FAÇA
        escreva(“Digite a nota: ”)
        leia(notas[i])
        somatoria <- somatoria + notas[i]
    FIMPARA

    media <- somatoria/4
    escreva(“Média: “, media)

    PARA(i <- 0; i < 4;i <- i + 1)FAÇA
        escreva(“Nota: ”, notas[i])
    FIMPARA
fim
```

C

```c
#include <stdio.h>

int main(){
    float notas[4], media, somatoria;
    int i;
    somatoria=0;

    for(i=0;i<4;i++){
        printf("\nDigite sua nota: ");
        scanf("%f",&notas[i]);
        somatoria += notas[i];
    }

    media = somatoria/4;
    printf("\nMédia: %.2f",media);

    for(i=0;i<4;i++){
        printf("\nNota: %.2f",notas[i]);
    }
}
```

OBS: o uso de vetores em C é basicamente o mesmo usado em nosso Português Estruturado.

É possível declarar um vetor com elementos em uma única linha. Por exemplo, se quiser criar um vetor *a*, de 3 elementos inteiros (2, 4 e 6), pode-se fazer da seguinte forma:

int a[] = {2, 4, 6};

C++

Pode ser o mesmo código do C, ou:

```
#include <iostream>

using namespace std;

int main(){
    float notas[4], media, somatoria;
    int i;
    somatoria=0;

    for(i=0;i<4;i++){
        cout << "\nDigite sua nota: ";
        cin >> notas[i];
        somatoria += notas[i];
    }

    media = somatoria/4;
    cout << "\nMédia: " « media;

    for(i=0;i<4;i++){
        cout << "\nNota: " << notas[i];
    }
}
```

Obs: as mesmas observações feitas em C se aplicam a C++.

É possível declarar um vetor com elementos em uma única linha. Por exemplo, se quiser criar um vetor *a*, de 3 elementos inteiros (2, 4 e 6), pode-se fazer da seguinte forma:

```
int a[] = {2, 4, 6};
```

Pode-se substituir o último laço for do código acima pelo do padrão C11, para recuperar valores dos vetores da seguinte maneira:

```
for(float e : notas){
    cout << “\nNota: “ << e
}
```

O laço ***for*** percorrerá todos os valores do vetor notas. A cada volta, o valor da respectiva posição será atribuído à variável *e*. Repare que a variável *e*, neste laço, não tem a função de contadora, mas armazenará o valor da posição do vetor notas.

Deste modo, na primeira volta, a variável *e* armazenará o valor da posição 0 do vetor; na segunda volta, *e* armazenará o valor da posição 1 do vetor; e assim sucessivamente até a última posição.

Go

```
package main

import (
    "fmt"
)

func main() {
    var notas [4]float32
    var media, somatoria float32
    var i int

    for i = 0; i < 4; i++ {
        fmt.Print("Digite sua nota: ")
        fmt.Scanf("%f\n", &notas[i])
        somatoria += notas[i]
    }

    media = somatoria / 4
    fmt.Print("\nMédia: ", media)

    for i = 0; i < 4; i++ {
        fmt.Print("\nNota: ", notas[i])
    }
}
```

OBS: é possível declarar um vetor com elementos em uma única linha. Por exemplo, se quiser criar um vetor *a*, de 3 elementos inteiros (2, 4 e 6), pode-se fazer da seguinte forma:

```
var a = [3]int{2, 4, 6}
```

ou

```
a := [3]int{2,4,6}
```

Em Go, também é possível usar uma variação do laço for para recuperar valores de vetores. Assim, o último laço for do exemplo acima poderia ser substituído por:

```
for _, e := range notas {
    fmt.Print("\nNota: ", e)
}
```

O traço (underline) é usado no lugar da variável contadora, quando não nos interessa seu valor, que é exatamente nosso caso. Repare que no exemplo acima, não queremos imprimir o valor do contador, apenas o valor da nota. Mas caso quiséssemos recuperar o valor do contador, poderíamos fazê-lo da seguinte forma:

```
for i, e := range notas {
    fmt.Print("\nNota ", i, ": ", e)
}
```

Assim, a variável *i* recebe o valor do contador e a variável *e* recebe o valor da nota, na posição de *i*.

Em Go, além dos vetores (arrays), é possível usar *slice*. A diferença entre vetor e *slice* é que este último não tem um tamanho fixo. Por isso, caso queira criar um "vetor" que possa aumentar de tamanho, é possível usar *slice* no lugar.

A declaração de um *slice* é muito parecida com a de um vetor, mas não se usa o tamanho entre colchetes:

```
var teste []float32
```

Para inserir valores(por exemplo, 3,5), usa-se:

```
teste = append(teste, 3.5)
```

E o uso é o mesmo:

```
fmt.Print("Valor inserido: ", teste[0])
```

O uso de *slices* tem impacto negativo na performance em relação aos vetores. Isso porque ***append*** é uma função que, ao ser chamada, criará um novo espaço para caber todos os elementos que estão na variável teste mais o novo valor, e retornará esse novo *slice* para a variável `teste`. É como pegar um vetor, criar um novo espaço para ele com uma posição a mais e chamar esse novo vetor com o mesmo nome.

Em programas que não exigem performance, como aplicações comerciais simples, o impacto não é visível.

Ruby

```
notas = Array.new(4)
somatoria = 0

for i in 0..3 do
    print "Digite sua nota: "
    notas[i] = gets.chomp.to_f
    somatoria += notas[i]
end

media = somatoria/4
puts "Média: " << media.to_s

for i in 0..3 do
    puts "Nota: " << notas[i].to_s
end
```

OBS: é possível declarar um vetor com elementos em uma única linha. Por exemplo, se quiser criar um vetor *a*, de 3 elementos inteiros (2, 4 e 6), pode-se fazer da seguinte forma:

a = [2, 4, 6] ou **a = Array[2,4,6]** ou **a = Array.[](2,4,6)**

Caso se deseje criar um vetor com vários elementos de mesmo valor, é possível fazê-lo da seguinte maneira:

```
a = Array.new(4, 3)
```

O código acima cria um vetor de 4 posições, em que cada posição começará com o valor 3.

Em Ruby, também é possível usar uma variação do laço ***for*** para recuperar valores de vetores. Assim, o último laço for do exemplo acima poderia ser substituído por:

```
for e in notas do
    puts “Nota: “<< e.to_s
end
```

Ou:

```
notas.each do |e|
    puts “Nota: “ << e.to_s
end
```

É possível declarar um vetor com um tamanho determinado em tempo de execução. Por exemplo, é possível criar um vetor com o tamanho definido pelo usuário, como abaixo:

```
print “Digite o tamanho do vetor”
i = gets.chomp.to_i
teste = Array.new(i)
```

Outras linguagens possuem recursos que também permitem a criação de um vetor com um tamanho definido pelo usuário. Em C/C++, é possível criar um vetor alocado dinamicamente com o uso da instrução malloc e calloc, mas foge do escopo deste livro.

Python

```
import numpy

notas = numpy.empty(4, float)
somatoria = 0

for i in range(4):
    notas[i] = float(input('Digite sua nota: '))
    somatoria += notas[i]

media = somatoria/4
print ('Média:', media)

for i in notas:
    print('Nota:',i)
```

Obs: para se criar um vetor vazio de quatro elementos, usa-se a biblioteca numpy. Muitas vezes, esta biblioteca não vem instalada junto com a linguagem. Se for o caso, deve-se instalá-la, usando-se um comando no prompt. Caso o gerenciador de pacotes pip esteja instalado, o comando é:

```
pip install numpy
```

No código acima, usamos uma variação do laço for para percorrer todo o vetor. Em cada volta, a variável `e` receberá o próximo valor do vetor.

É possível declarar um vetor com elementos em uma única linha. Por exemplo, se quisermos criar um vetor `a`, de 3 elementos inteiros (2, 4 e 6), pode-se fazer da seguinte forma:

```
a = [2, 4, 6]
```

É possível declarar um vetor com um tamanho determinado em tempo de execução. Por exemplo, é possível criar um vetor com o tamanho definido pelo usuário, como abaixo:

```
i = int(input('Digite o tamanho:'))
teste = np.empty(i,int)
print (teste.size)
```

Outras linguagens possuem recursos que também permitem a criação de um vetor com um tamanho definido pelo usuário. Em C/C++, é possível criar um vetor alocado dinamicamente com o uso da instrução malloc e calloc, mas foge do escopo deste livro.

Interrompemos a programação... (pré-requisito para a próxima seção)

Vamos brincar um pouco com laço ***para*** aninhado!

Tente executar no papel com um teste de mesa (ou mentalmente) o algoritmo abaixo. Qual será o resultado?

```
1. início
2.     inteiro i, j
3.     para(i <- 0;i<3;i <- i+1)faça
4.         para(j <- 0; j<4; j <- j+1)faça
5.             escreva(i , " ", j)
6.         fimpara
7.     fimpara
8. fim
```

Temos um código com laços ***para*** aninhados. Chamamos o primeiro ***para*** da linha 3 de laço mais externo (ou ***para*** mais externo) e o de dentro como o mais interno. Repare que usamos uma variável contadora diferente para cada laço.

De forma resumida, assim que entrarmos no laço ***para*** mais externo e entrarmos em seu bloco, todo o laço ***para*** interno tem que terminar para que voltemos à linha 3. Então:

a) As linhas 1, 2 e 3 são executadas em sequência;
b) Como a condição da linha 3 é verdadeira (porque o valor de `i` é 0), então entramos no bloco do laço externo para executar o que estiver dentro (as linhas 4, 5 e 6);
c) Assim, a linha 4 será executada. E como a condição do laço mais interno também é verdadeira (porque o valor inicial de `j` é 0), entraremos no bloco do laço ***para*** interno para executar a linha 5.
d) A linha 5 será executada e mostrará os valores de `i` e `j`: 0 0;
e) Agora vem a parte legal. Como chegamos ao fim do bloco do laço ***para*** mais internos, `j` será incrementado em 1 e voltaremos para o começo do laço ***para*** mais INTERNO! Porque o ***fimpara*** da linha 6 pertence ao ***para*** da linha 4;
f) Então, voltamos para a linha 4 e testamos a condição. Como `j` ainda é menor que 4 (porque `j` agora vale 1), então entramos novamente

na linha 5 e a executamos. Agora os valores de *i* e *j* são, respectivamente, 0 e 1;

g) Então, voltamos ao passo **e** desta nossa lista. A variável *j* será incrementada em 1 (passa a valer 2) e vamos para a linha 4 testas a condição;
h) Como *j* continua sendo menor que 4 (porque agora *j* vale 2), executamos a linha 5 e apresentamos os valores 0 e 2;
i) Incrementamos a variável *j* e ela passa a valer 3;
j) Voltamos para a linha 4 e a condição continua sendo verdadeira;
k) Executamos a linha 5 e apresentamos os valores de *i* e *j*: 0 3;
l) Incrementamos a variável *j* e ela passa a valer 4;
m) Voltamos para a linha 4. Mas agora a condição do laço ***para*** mais interno é falsa porque *j* não é menor que 4 (já que 4 não é menor que 4). Por causa disso, saímos do laço mais interno para a linha 7;
n) Como a linha 7 é o fim do laço ***para*** mais EXTERNO, então incrementamos a variável pertencente ao laço mais EXTERNO. Isto é, incrementamos o valor de *i* e ele passa a valer 1;
o) Como 1 é menor que 3, então, entramos no bloco do laço mais externo e executamos a linha 4;
p) Como voltamos a executar o laço ***para*** interno novamente do começo, então o valor de *j* passa a valer 0 para, em seguida, testarmos se `j<4`;
q) Como 0 é menor que quatro, entramos na linha 5 e apresentamos na tela os valores de *i* e *j*: 1 0;

Agora, o raciocínio é o mesmo. Então, o laço interno inteiro será executado e serão mostrados os valores:

```
1 0
1 1
1 2
1 3
```

Quando o laço interno terminar, será incrementado o valor de *i* (a variável *i* passa a valer 2). A condição do laço externo continuará a ser verdadeira e o laço ***para*** interno será novamente executado, gerando as saídas:

```
2 0
2 1
2 2
2 3
```

E *i* será incrementado de novo. Como a condição do laço externo é falsa quando *i* for igual a 3, o algoritmo termina.

Deste modo, as saídas são 0 0, 0 1, 0 2, 0 3, 1 0, 1 1, 1 2, 1 3, 2 0, 2 1, 2 2, 2 3.

Matrizes

Imagine, agora, que em uma disciplina, cada aluno realiza quatro provas.

Se eu quiser armazenar as quatro notas de um aluno, eu poderia criar um vetor, como abaixo:

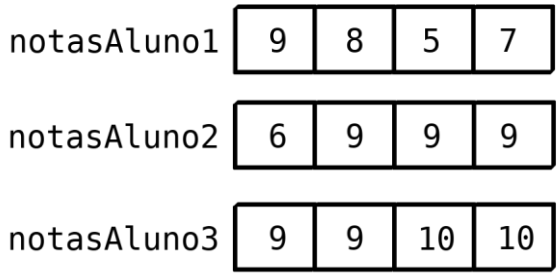
```
real notas[4]
```

Mas e se eu quiser armazenas as 4 notas de três alunos? Eu poderia criar 3 vetores:

```
real notasAluno1[4], notasAluno2[4],
notasAluno3[4]
```

A representação destas notas seria como na Figura 10.

Figura 10 - Três vetores

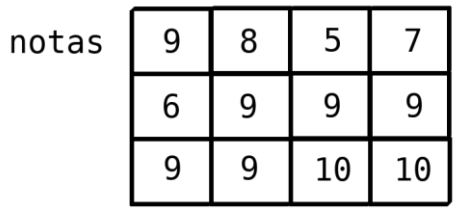


Mas começamos a ter um problema. Do jeito que está, como faríamos um algoritmo para preencher cada vetor? Precisaríamos de um laço **para** para cada

vetor. Isso começa a ficar inconveniente se eu quiser armazenar as quatro notas de cada aluno de uma turma de 60 pessoas!

A solução, então, é criarmos uma variável que pareça uma tabela (Figura 11):

Figura 11 - Matriz



Este tipo de variável também é composta e homogênea, pelos mesmos motivos de um vetor: composta porque é uma variável que armazena mais de um valor e homogênea porque todos os valores precisam ser de um mesmo tipo.

Esta variável agora possui linhas e colunas! Por isso dizemos que ela possui duas dimensões. O nome que damos a ela é **matriz de duas dimensões**.

Um vetor também é uma matriz! Mas uma matriz de uma dimensão, porque só possui linhas. Geralmente chamamos uma matriz de duas dimensões simplesmente de matriz. E uma matriz de uma dimensão, de vetor.

E novamente temos as perguntas: como declaro uma variável assim? Como eu faço para colocar um valor em cada posição diferente?

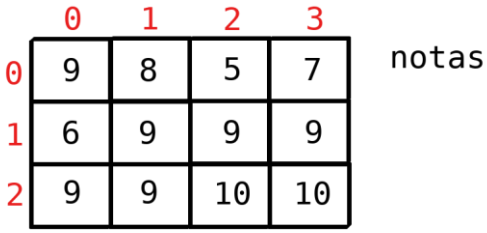
Para declarar uma matriz de duas dimensões, usamos dois pares de colchetes: um para declarar a quantidade de linhas e um para a quantidade de colunas. Seguindo o exemplo acima, faremos:

```
real notas[3][4]
```

Declaramos uma matriz que possui 3 linhas por 4 colunas.

E como coloco valores em cada posição? Para isso, vamos adotar que as matrizes possuem dois índices (um para cada dimensão, como em um jogo de xadrez, batalha naval... ou bingo!), como na Figura 12:

Figura 12 - Posições de uma matriz



Então, para colocarmos um valor na coordenada (0,0), fazemos:

```
notas[0][0] <- 9
```

Para colocarmos um valor na coordenada (0,1):

```
notas[0][1] <- 8
```

E assim por diante.

Na hora de preencher a matriz, não queremos criar uma instrução ***leia*** para cada posição. O que faremos é usar o laço ***para***, usando a mesma lógica que usamos com vetores.

Hum... quando trabalhamos com vetores, primeiro colocamos um valor na posição 0, depois na 1, depois na 2 e assim por diante. Como faremos com a matriz? Primeiro vamos colocar um valor na posição (0,0), depois na posição (0,1), depois (0,2), etc... Assim, as posições que serão utilizadas seguirão a ordem:

```
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
```

Se você leu a seção anterior, vai achar estes números familiares. Como faço para usar laço ***para*** e conseguir gerar esses números? Com laços ***para*** aninhados!

Então, para criar um algoritmo que atribua valores a cada posição da minha matriz, fazemos como a seguir:

```
1.  inicio
2.     inteiro i, j
3.     real notas[3][4]
4.     para(i <- 0; i < 3;i <- i + 1)faça
5.         para(j <- 0; j < 4; j <- j + 1)faça
6.             escreva(“Digite uma nota:”)
7.             leia(notas[i][j])
8.         fimpara
9.     fimpara
10.  fim
```

É muito simples! O mesmo para apresentar as notas depois.

Mas, sob o ponto de vista do usuário, estamos acostumados a contar a partir do número 1. Então, pode ficar estranho pedir para o usuário digitar a nota 0 do aluno 0. O mais natural é pedir a nota 1 do aluno 1, a nota 2 do aluno 1, e assim por diante. Por isso, na hora de apresentar na tela o que deve ser digitado, podemos alterar a linha 6 para:

```
escreva(“Digite uma nota “, j+1, “ do aluno “, i+1, “:”)
```

Repare que não alteramos o valor de *j*, nem de *i*. Apenas, na hora de apresentar, somamos 1 ao que será apresentado, mas sem mudar o valor da variável em si.

Precisamos reparar em alguns detalhes (use lápis e papel para acompanhar, se ficar um pouco confuso):

- Estamos digitando a primeira nota do primeiro aluno, depois a segunda nota do primeiro aluno, até a quarta nota do primeiro aluno, para depois digitarmos as notas do segundo aluno e assim por diante;
- Então, *i* representa o aluno e *j* representa a nota;

- Mas na hora de mostrar para o usuário, é mais natural escrevermos que queremos a **nota** do **aluno**. Por isso, o `j` está sendo concatenado antes do `i`;
- Também é mais natural para o usuário se começarmos a contar a partir de 1. Por isso que `i` e `j` está sendo somados a 1 na instrução ***escreva***;
- Por fim, a soma de `i` e de `j` não estão alterando os valores destas variáveis, porque não estamos atribuindo o resultado das somas a elas (estamos fazendo simplesmente `i + 1`, e não `i<-i + 1`);

Agora que vimos como manipular matrizes e quando devemos usá-las, vamos praticar!

Exercícios de Matrizes

1. Crie uma matriz de 2x2, de números inteiros. Populacione-a usando laço PARA e, por fim, apresente todos os valores na tela.

PARA OS EXERCÍCIOS QUE PEDEM PARA O RESULTADO SER APRESENTADO COMO UMA MATRIZ, CONSIDERE QUE A INSTRUÇÃO ESCREVA NÃO PULA DE LINHA AUTOMATICAMENTE E QUE PARA PULAR UMA LINHA É NECESSÁRIO O \n, COMO EM C:

```
escreva("\n") //pular linha
```

2. Crie uma matriz de 3x3, de números reais. Populacione-a usando laço PARA e, por fim, apresente todos os valores na tela, mas na forma de matriz.

3. Crie uma matriz de 3x2, de números inteiros. Populacione-a usando laço para e, por fim, apresente todos os valores, mas de forma invertida. Por exemplo, se: 3 7 1 8 4 1

3	7
1	8
4	1

Então, deve-se apresentar a seguinte ordem na tela. 3 1 4 7 8 1

3	1	4
7	8	1

4. Crie uma matriz de 2x3, de números inteiros e a populacione usando laço para. Por fim, calcule e apresente a soma de cada linha. Por exemplo, se o usuário digitar: 3 1 4 7 8 1 e a matriz for semelhante à última do exercício anterior, então, o algoritmo deverá apresentar respectivamente: 8 e 16

5. Crie três matrizes de 3x3 de inteiros: mat1, mat2 e mult. Em seguida, peça para o usuário digitar os valores para mat1 e mat2. Em seguida, o algoritmo deverá

preencher a matriz mult com a multiplicação das respectivas posições das matrizes mat1 e mat2.

mat1

3	1	4
7	8	1
2	1	3

mat2

2	2	0
3	4	1
5	2	1

mult

6	2	0
21	32	1
10	2	3

Como é em algumas linguagens

Português Estruturado

```
inicio
    real notas[3][4]
    inteiro i, j

    para(i <- 0; i < 3; i <- i + 1)faça
        para(j <- 0; j < 4; j <- j + 1)faça
            escreva("Digite a nota ", i+1, " do aluno ", j+1,": ")
            leia(notas[i][j])
        fimpara
    fimpara

    para(i <- 0;i<3;i <- i+1)faça
        para(j <- 0; j < 4; j <- j + 1)faça
            escreva("Nota ", i+1, " do aluno ", j+1,": ", notas[i][j])
        fimpara
    fimpara
fim
```

C

```c
#include <stdio.h>

int main(){
    float notas[3][4];
    int i, j;

    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            printf("\nDigite a nota %d do aluno %d: ", j+1, i+1);
            scanf("%f",&notas[i][j]);
        }
    }

    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            printf("\nNota %d do aluno %d: %.2f", j+1, i+1, notas[i][j]);
        }
    }
}
```

OBS: o uso de matrizes em C é basicamente o mesmo usado em nosso Português Estruturado.

É possível declarar uma matriz com elementos em uma única linha. Por exemplo, se quiser criar uma matriz *a*, de 2x3 elementos inteiros:

```
2 4
6 8
```

Pode-se fazer da seguinte forma:

```c
int a[2][2] = {{2, 4}, {6, 8}};
```

C++

Pode ser o mesmo código do C, ou:

```
#include<iostream>

using namespace std;

int main(){
    float notas[3][4];
    int i, j;

    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            cout << "\nDigite a nota " << j+1 << " do aluno "
                     << i+1 << ": ";
            cin >> notas[i][j];
        }
    }

    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            cout << "\nNota " << j+1 << " do aluno "
                     << i+1 << ": " << notas[i][j];
        }
    }
}
```

Obs: as mesmas observações feitas em C se aplicam a C++.

É possível declarar uma matriz com elementos em uma única linha. Por exemplo, se quiser criar uma matriz *a*, de 2x3 elementos inteiros:

```
2 4 6
1 3 5
```

Pode-se fazer da seguinte forma:

```
int a[2][3] = {{2, 4, 6}, {1, 3, 5}};
```

Go

```
package main

import (
    "fmt"
)

func main() {
    var notas [3][4]float32
    var i, j int

    for i = 0; i < 3; i++ {
        for j = 0; j < 4; j++ {
            fmt.Print("Digite a nota ", j+1, " do aluno ", i+1, ": ")
            fmt.Scanf("%f\n", &notas[i][j])
        }
    }

    for i = 0; i < 3; i++ {
        for j = 0; j < 4; j++ {
            fmt.Println("Nota ", j+1, " do aluno ", i+1, ": ", notas[i][j])
        }
    }
}
```

OBS: é possível declarar uma matriz com elementos em uma única linha. Por exemplo, se quiser criar uma matriz *a*, de 2x3 elementos inteiros:

```
2 4 6
1 3 5
```

Pode-se fazer da seguinte forma:

```
var a = [2][3]int{{2, 4, 6}, {1, 3, 5}}
```

ou

```
a := [2][3]int{{2, 4, 6}, {1, 3, 5}}
```

Ruby

```
notas = Array.new(3, Array.new(4))

for i in 0..2 do
    for j in 0..3 do
        print "Digite a nota ", j+1, " do aluno ", i+1, ":"
        notas[i][j] = gets.chomp.to_f
    end
end

for i in 0..2 do
    for j in 0..3 do
        print "\nNota ", j+1, " do aluno ", i+1, ": ", notas[i][j]
    end
end
```

OBS: é possível declarar uma matriz com elementos em uma única linha. Por exemplo, se quiser criar uma matriz *a*, de 2x3 elementos inteiros:

```
2 4 6
1 3 5
```

Pode-se fazer da seguinte forma:

```
a = [[2, 4, 6], [1, 3, 5]]
```

ou

```
a  = Array[[2, 4, 6], [1, 3, 5]]
```

Python

```
import numpy as np

notas = np.empty((3, 4))

for i in range(3):
    for j in range(4):
        notas[i][j]=float(input('Digite a nota '+str(j+1)+' do aluno
'+str(i+1)+':'))

for i in range(3):
    for j in range(4):
        print ('Nota '+ str(j+1) + ' do aluno ' + str(i+1)+': ' +
str(notas[i][j]))
```

Obs: para se criar uma matriz vazia de quatro elementos, usa-se a biblioteca numpy. Muitas vezes, esta biblioteca não vem instalada junto com a linguagem. Se for o caso, deve-se instalá-la, usando-se um comando no prompt. Caso o gerenciador de pacotes pip esteja instalado, o comando é:

```
pip install numpy
```

OBS: é possível declarar uma matriz com elementos em uma única linha. Por exemplo, se quiser criar uma matriz *a*, de 2x3 elementos inteiros:

```
2 4 6
1 3 5
```

Pode-se fazer da seguinte forma:

```
a = [[2, 4, 6], [1, 3, 5]]
```

Variáveis Compostas Heterogêneas

Vamos analisar um outro inconveniente relacionado a variáveis e como resolvê-lo apenas com um recurso simples de organização. Imagine que você precise representar uma pessoa no seu código, como um aluno. Em determinado momento, você precisa armazenar o nome, registro acadêmico (RA), endereço e o coeficiente acadêmico (um número real que representa quão bem... ou mal... está o desempenho do aluno até o momento).

É muito simples criar um algoritmo que peça os respectivos valores e os apresente na tela:

```
1. início
2.     inteiro ra
3.     cadeia nome, endereco
4.     real coeficiente
5.     escreva(“Digite o nome do aluno:”)
6.     leia(nome)
7.     escreva(“Digite o endereço do aluno:”)
8.     leia(endereco)
9.     escreva(“Digite o RA do aluno:”)
10.    leia(ra)
11.    escreva(“Digite o coeficiente do aluno:”)
12.    leia(coeficiente)
13.    escreva(“Nome: “, nome)
14.    escreva(“Endereço: “, endereco)
15.    escreva(“RA: “, ra)
16.    escreva(“Coeficiente: “, coeficiente)
17. fim
```

Ótima solução! Mas começamos a ter um inconveniente se precisarmos, ao mesmo tempo, armazenar outros valores. Principalmente se os outros valores têm o mesmo significado. Por exemplo, e se tivermos que, ao mesmo tempo, armazenar os valores do professor, como nome, endereço, salário e função?

Também é possível criar um algoritmo da forma como estávamos acostumados, mas começamos a ter variáveis que têm o mesmo significado (como nome), apesar de pertencerem a grupos de pessoas diferentes. Teríamos que criar variáveis com nomes que fossem fáceis de distinguir, como:

```
1. início
2.     inteiro ra
3.     cadeia nomeA, enderecoA
4.     cadeia nomeP, enderecoP, funcao
5.     real coeficiente, salario
6.     ...
```

Repare no trecho anterior que *nomeA* pertence ao aluno e *nomeP* pertence ao professor. Até é possível criar um algoritmo assim. Mas começa a ficar cada vez mais confuso para o programador (ou para quem for dar manutenção no programa, depois) a qual grupo pertence cada variável. E se além do aluno e professor, precisássemos armazenar valores de... pássaros??? Sim, porque são alunos de um curso de ornitologia!

Então, criar uma variável *nomeP* para o nome do pássaro poderia confundir com o *nomeP* do professor. Precisaria ser *nomePA*.

Felizmente, as linguagens de programação mais usadas possuem um recurso bastante interessante chamado de registro (ou estrutura). A ideia é:

Sempre que o programador identificar que um conjunto de variáveis representa uma categoria, é possível agrupar essas variáveis e dar um nome a elas. Esse agrupamento será visto como um novo tipo de variável que pode ser declarado!

Então, pegando o exemplo de alunos, agrupamos as variáveis relacionadas ao aluno da seguinte forma:

```
1. estrutura aluno
2.     cadeia nome, endereco
3.     inteiro ra
4.     real coeficiente
5. fimestrutura
```

É assim que criaremos a estrutura que representa o aluno. Essa estrutura é vista pela linguagem de programação como um novo tipo de dados! Então, agora, basta declararmos uma variável do tipo aluno! A declaração segue a mesma regra de declaração que fizemos até o momento: o tipo seguido do nome da variável (também chamado de identificador da variável).

```
aluno a
```

No exemplo acima, criamos uma variável *a*, do tipo aluno!

E como fazemos para usar a variável *a*, se ela tem nome, endereço, RA e coeficiente? A propósito, passamos a chamar eles de "campos". Como atribuímos valores aos campos da variável *a*? Fácil!

```
a.nome < - "Miguel"
a.endereco < - "Rua A"
a.ra < - 1323
a.coeficiente < - 9.8
```

Temos então, uma variável *a* do tipo aluno que possui os quatro campos. E para acessá-los, basta um "ponto" e o nome do campo.

Começamos a achar interessante o uso de estruturas (ou registros), por exemplo, se precisarmos representar dois alunos ao mesmo tempo. Basta declarar duas variáveis:

```
aluno a, b
```

Se executarmos:

```
a.nome < - "Miguel"
b.nome < - "Ana"
```

Teremos duas variáveis, cada uma com um valor diferente para o nome.

Também percebemos que o código se torna mais organizado e fácil de ler, quando tivermos uma situação como a apresentada no começo da seção. Se temos conjuntos de variáveis que representam aluno, outro conjunto que representa professor e outro conjunto que representa pássaros, criamos uma estrutura para cada categoria. Uma vez criada cada estrutura, para criar uma variável de cada estrutura, teremos algo como:

```
1. aluno a
2. professor prof
3. passaro pass
```

```
4. a.nome <- “Miguel”
5. prof.nome <- “Paulo”
6. pass.nome < - “Sabiá”
```

O bacana é que cada variável possui o seu próprio campo que pode ser chamado de nome. Porque um é o campo nome de um aluno, o outro é o campo nome de um professor e o outro é o campo nome de um pássaro.

Então, não precisamos declarar um grupo de variáveis para cada aluno que quisermos representar, ou um grupo de variáveis para professores, etc... Precisamos apenas criar uma estrutura para cada grupo, uma vez, e uma variável (pelo menos) para cada estrutura.

Repare no antes:

```
1. início
2.     inteiro ra, categoria
3.     cadeia nomeA, enderecoA
4.     cadeia nomeProf, enderecoProf
5.     cadeia funcao, nomePass, cor, especie
6.     real coeficiente, salario, peso
```

E no depois:

```
início
   aluno a
   professor b
   passaro c
```

Percebeu como a parte de declaração é mais limpa no segundo exemplo? Repare também que não precisamos dar nomes diferentes para as variáveis que têm o mesmo significado, mas que pertencem a grupos diferentes, como nomeA, nomeProf e nomePass. Isso porque se atribuirmos valores como abaixo:

```
a.nome <- “José”
b.nome <- “Maria”
c.nome <- “Pica-pau”
```

O campo nome da variável *a* representa o nome de um aluno, porque a variável *a* é do tipo aluno. Da mesma forma, o campo nome da variável *b* representa o nome de um professor, porque *b* é uma variável do tipo professor. O mesmo para o pássaro!

Isso tudo porque organizamos as variáveis do primeiro exemplo em grupos (as estruturas), tornando-as um campo de um novo tipo de variável.

E, por falar em variável, essas três variáveis (*a*, *b* e *c*), chamamos de **variáveis compostas HETEROGÊNEAS**! A variável *a* é composta porque ela armazena mais de um valor (como nome, endereço, etc...) e é heterogênea porque os campos podem ser de tipos diferentes! Por exemplo, nome e endereço são do tipo cadeia, mas RA é do tipo inteiro e coeficiente é do tipo real.

Nós criamos uma estrutura ANTES do bloco início-fim!

```
1.  estrutura aluno
2.      cadeia nome, endereco
3.      inteiro ra
4.      real coeficiente
5.  fimestrutura
6.
7.  estrutura professor
8.      cadeia nome, endereco, funcao
9.      real salario
10. fimestrutura
11.
12.  INÍCIO
13.     aluno a
14.     professor p
15.     escreva(“Digite o nome do aluno:”)
16.     leia(a.nome)
17.     escreva(“Digite o endereço:”)
18.     leia(a.endereco)
19.     escreva(“Digite o RA:”)
20.     leia(a.ra)
21.     escreva(“Digite o coeficiente:”)
22.     leia(a.coeficiente)
23.     ...
```

Vamos reforçar aqui a terminologia:

- Um **registro** e uma **estrutura** são sinônimos! Por que existem dois nomes? Porque tem linguagem cuja instrução se chama struct (estrutura) e tem linguagem que chama a mesma instrução de registro (record).

- Uma **estrutura** (ou registro) **é um modelo**, possui um conjunto de campos que **representam alguma abstração**, como pessoa, aluno, funcionário, produto, venda, etc...

- E o que é uma abstração? É o processo de se formular uma visão. Ou, ainda, a visão de um objeto em que se foca na informação relevante para um propósito particular e se ignora o que é irrelevante (ISO 24765). Por exemplo, no nosso propósito particular de desenvolvimento do nosso sistema é relevante armazenar o nome e endereço do aluno, mas é irrelevante armazenar a cor preferida e signo (mas esses dois últimos poderiam ser relevantes em um outro sistema! Por isso o "propósito particular" da definição acima).

- A linguagem de programação entende a estrutura como um novo tipo de dados, chamado **tipo abstrato de dados**. O tipo é abstrato justamente porque o conjunto de campos reunidos na estrutura criada representa a visão de algo maior que elas próprias. Isto é, nome, endereço e RA, juntos, representam um aluno, que é uma abstração.

- Tem linguagem em que precisamos primeiro criar a estrutura, depois indicar que a estrutura é um novo tipo de dados para depois fazer declarações de variáveis a partir deste novo tipo (como a linguagem C e o uso da instrução typedef). Mas há linguagens em que ao criar a estrutura, automaticamente ela é considerada um novo tipo de dados (como C++).

- A variável declarada a partir deste tipo abstrato de dados é chamada **variável composta heterogênea** porque ela é composta por campos que podem possuir diferentes tipos, como cadeia para nome e inteiro para RA.

```
1. estrutura aluno
2.     cadeia nome, endereco
3.     inteiro ra
4.     real coeficiente
5. fimestrutura

6  início
7      aluno a
8      …
```

No exemplo acima, no balão azul criamos a estrutura de um novo tipo abstrato de dados (a estrutura do tipo aluno). No balão vermelho, declaramos uma variável composta heterogênea *a* do tipo *aluno*.

Exercícios de Estruturas

1. Faça um algoritmo que tenha um registro (estrutura) que represente um aluno. O usuário deve digitar o RA, nome e endereço do aluno. Em seguida, o algoritmo deve apresentar os dados do aluno.

2. Faça um algoritmo que tenha um registro (estrutura) que represente um DVD em uma locadora. Então, deve ser armazenado o título, gênero, duração e prateleira. O programa deve pedir os respectivos dados do DVD. Em seguida, deve apresentá-los.

3. Faça um algoritmo que tenha um registro (estrutura) que represente um monstro de bolso. O usuário deve digitar o nome, vida, ataque e defesa do bichinho. Em seguida, os dados devem ser apresentados.

4. Faça um algoritmo que tenha um registro (estrutura) de monstro de bolso (como do exercício anterior) e um registro (estrutura) de treinador. Este último deve ter os campos nome, idade e número de insígnias. O usuário deve digitar os valores para os campos de um monstrinho e os valores dos campos de um treinador. Em seguida, o algoritmo deve apresentar os valores de ambos na tela.

Como é em algumas linguagens

Português Estruturado

```
estrutura aluno
    cadeia nome, endereco
    inteiro ra
    real coeficiente
fim estrutura

início
    aluno a
    a.nome < - “Miguel”
    a.endereco < - “Rua A”
    a.ra < - 1323
    a.coeficiente < - 9.8

    escreva("Seu nome: ", a.nome)
    escreva("Seu endereço: ", a.endereco)
    escreva("Seu RA: ", a.ra)
    escreva("Seu coeficiente: ", a.coeficiente)
fim
```

C

```c
#include <stdio.h>
#include <string.h>

struct aluno{
    char nome[90], endereco[250];
    int ra;
    float coeficiente;
};

typedef aluno aluno;

int main(){
    aluno a;

    strcpy(a.nome, "Miguel");
    strcpy(a.endereco, "Rua A");
    a.ra = 1323;
    a.coeficiente = 0.9f;

    printf("\nSeu nome: %s", a.nome);
    printf("\nSeu endereço: %s", a.endereco);
    printf("\nSeu RA: %d", a.ra);
    printf("\nSeu coeficiente: %.2f", a.coeficiente);
}
```

OBS: em C, primeiro criamos uma estrutura para, em seguida, definir esta estrutura como um novo tipo. É por isso que, após criarmos a struct aluno, temos uma linha:

```c
typedef aluno aluno;
```

Esta linha define um novo tipo (aqui também chamado aluno) a partir da estrutura aluno. Mas poderíamos criar o novo tipo com outro nome, como:

```c
typedef aluno alu;
```

Neste caso, o novo tipo se chama alu, criado a partir da estrutura aluno. Assim, para declarar uma variável a partir do tipo alu:

```c
int main(){
    alu a;
    ...
}
```

Há, também, outras formas de criar a estrutura e o tipo, como:

```c
typedef struct aluno{
    char nome[90], endereco[250];
    int ra;
    float coeficiente;
} aluno;
```

Esta é uma forma simplificada de se criar a estrutura e defini-la como um novo tipo, ao mesmo tempo.

Em C, não há o tipo *string*. Como substituto, usamos um vetor de caracteres. Por isso, precisamos usar instruções especiais para atribuir valores a `nome` e `endereço`. No nosso exemplo, `strcpy` é uma instrução que copia um valor para um vetor.

Se quisermos que o usuário digite os valores, devemos substituir as atribuições pelo seguinte código:

```
printf("\nDigite o nome: ");
gets(a.nome);
printf("\nDigite o endereço: ");
gets(a.endereco);
printf("\nDigite o RA: ");
scanf("%d",&a.ra);
printf("\nDigite o coeficiente: ");
scanf("%f",&a.coeficiente);
```

Quando se usa vetor de caracteres em substituição ao tipo *string*, usamos a instrução `gets` no lugar de `scanf`.

C++

Pode ser o mesmo código do C, ou:

```
#include <iostream>
using namespace std;

struct aluno{
    string nome, endereco;
    int ra;
    float coeficiente;
};

int main(){
    aluno a;

    a.nome = "Miguel";
    a.endereco = "Rua A";
    a.ra = 1323;
    a.coeficiente = 0.9f;

    cout << "\nSeu nome: " << a.nome;
    cout << "\nSeu endereço: " << a.endereco;
    cout << "\nSeu RA: " << a.ra;
    cout << "\nSeu coeficiente: " << a.coeficiente;
}
```

Obs: em C++, a criação de uma estrutura pode ser simplificada como no exemplo acima. Mesmo sem a instrução typedef, o C++ cria um novo tipo com o mesmo nome da estrutura automaticamente.

Diferente de C, em C++ temos cadeia de caracteres!!! As strings!!!

Se quisermos que o usuário digite os valores, devemos substituir as atribuições pelo seguinte código:

```
    cout << "\nDigite o nome: ";
    getline (cin,a.nome);
    cout << "\nDigite o endereço: ";
    getline (cin, a.endereco);
    cout << "\nDigite o RA: ";
    cin >> a.ra;
    cout << "\nDigite o coeficiente: ";
    cin >> a.coeficiente;
```

Se usarmos apenas cin >> a.nome, teremos um problema caso digitemos mais de uma palavra: apenas a primeira será usada. Isso ocorre por causa da maneira como os *buffers* são gerenciados. É para evitar esse problema que usamos a instrução ***getline***.

Go

```
package main

import (
    "fmt"
)

type aluno struct {
    nome, endereco string
    ra int
    coeficiente float32
}

func main() {
    var a aluno

    a.nome = "Miguel"
    a.endereco = "Rua A"
    a.ra = 1323
    a.coeficiente = 0.9

    fmt.Println("Seu nome:", a.nome)
    fmt.Println("Seu endereço:", a.endereco)
    fmt.Println("Seu RA:", a.ra)
    fmt.Println("Seu coeficiente:", a.coeficiente)
}
```

OBS: A criação de estruturas em Go é muito semelhante a C e C++.

Se quisermos que o usuário digite os valores, devemos substituir as atribuições pelo seguinte código:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

type aluno struct {
    nome, endereco string
    ra             int
    coeficiente    float32
}

func main() {
    var a aluno

    teclado := bufio.NewReader(os.Stdin)
    fmt.Print("Digite o nome:")
    a.nome, _ = teclado.ReadString('\n')
    fmt.Print("Digite o endereço:")
    a.endereco, _ = teclado.ReadString('\n')
    fmt.Print("Digite o RA:")
    fmt.Scanf("%d\n", &a.ra)
```

```
    fmt.Print("Digite o coeficiente: ")
    fmt.Scanf("%f\n", &a.coeficiente)
    a.nome = strings.Replace(a.nome, "\n", "", -1)
    a.endereco = strings.Replace(a.endereco, "\n", "", -1)
    ...
```

Em Go, quando se pede a leitura de cadeia de caracteres, evita-se usar *scanf* ou *scanln*, entre outros, também por causa do espaço entre palavras e o gerenciamento do *buffer* de entrada. Sugere-se criar um objeto de leitura (no nosso exemplo, chamamos de teclado) a partir do *buffer* de entrada/saída.

Este objeto que criamos tem um método (instrução) que realiza a leitura de uma cadeia de caracteres até que encontre um <enter> representado pelo símbolo '\n'.

Mas como o <enter> também passa a fazer parte da cadeia de caracteres, nas duas últimas linhas do exemplo estamos justamente removendo-as, onde substituímos '\n' (que é o enter) por "".

Ruby

```
aluno = Struct.new(:nome, :endereco, :ra, :coeficiente)

a = aluno.new("Miguel", "Rua A", 1323, 0.9)

print "\nSeu nome: ", a.nome
print "\nSeu endereço: ", a.endereco
print "\nSeu RA: ", a.ra
print "\nSeu coeficiente: ", a.coeficiente
```

OBS: Como variáveis em Ruby possuem o tipo de acordo com o valor a ser atribuído, ao se criar uma estrutura, apenas escrevemos o nome dos campos, sem seus tipos!

Quando criamos a variável *a*, podemos atribuir os valores de uma única vez, como no exemplo acima. Mas poderíamos criar a variável e, depois atribuir os valores, como abaixo:

```
a = aluno.new()

a.nome = "Miguel"
a.endereco = "Rua A"
a.ra = 1323
a.coeficiente = 0.9
```

E caso desejemos que o usuário digite os valores, podemos fazer do seguinte modo:

```
a = aluno.new()
print "\nDigite o nome: "
a.nome = gets.chomp
print "\nDigite o endereço: "
a.endereco = gets.chomp
print "\nDigite o RA: "
a.ra = gets.chomp.to_i
print "\nDigite o coeficiente: "
a.coeficiente = gets.chomp.to_f
```

Python

```
from dataclasses import dataclass
import string

@dataclass
class aluno:
    nome: string
    endereco: string
    ra: int
    coeficiente: float

a = aluno('Miguel', 'Rua A', 1323, 0.9)
print('Seu nome: ' + a.nome)
print('Seu endereço: ' + a.endereco)
print('Seu RA: ' + str(a.ra))
print('Seu coeficiente: ' + str(a.coeficiente))
```

Obs: em Python, não temos um equivalente direto a estrutura, como em C. Mas podemos usar uma classe de dados (*dataclass*). E é exatamente o que fazemos no nosso exemplo acima.

Assim como em Ruby, podemos atribuir os valores aos campos em uma única linha. Mas poderíamos fazer separadamente, como abaixo. Não se preocupe agora com o símbolo @. Mas ele significa que criaremos uma classe aluno usando o *decorator dataclass* (classe apenas de dados). E por que usamos isso antes de criar a classe? Para simplificar a escrita. Sem o *decorator*, precisaríamos escrever a classe aluno da seguinte forma:

```
class aluno:
    def __init__(self, nome, endereco, ra, coeficiente):
        self.nome = nome
        self.endereco = endereco
        self.ra = ra
        self.coeficiente = coeficiente
```

E teríamos uma sintaxe mais "verborrágica". Quando usamos o `@dataclass`, as funcionalidades da classe são implementadas automaticamente. E como fazemos, se quisermos que o usuário entre com os dados?

```
from dataclasses import dataclass, field
import string

@dataclass
class aluno:
    nome: string = field(init=False)
    endereco: string = field(init=False)
    ra: int = field(init=False)
    coeficiente: float = field(init=False)

a = aluno()
a.nome = input('Digite o nome: ')
a.endereco = input('Digite o endereço: ')
a.ra = int(input('Digite o RA: '))
a.coeficiente = float(input('Digite o coeficiente: '))
```

Para criarmos uma variável `a` sem inicializar seus campos na mesma linha, precisamos informar que cada campo não precisa ser inicializado quando a variável for declarada. Por isso temos o `=field(init=False)` em cada campo.

Misturando Variáveis Compostas Homogêneas e Heterogêneas

Uma das belezas em Algoritmos é a capacidade de se combinar diferentes instruções e recursos. Já fizemos isso quando precisamos colocar um bloco ***se-senão*** dentro de um bloco ***para***, por exemplo.

Em relação às variáveis compostas, algo semelhante acontece. Nas duas próximas subseções, veremos dois exemplos de combinações.

Variável Composta Homogênea cujos elementos são Variáveis Compostas Heterogêneas

Imagine um grupo de variáveis que possam ser agrupadas em uma estrutura, como um monstrinho que batalha contra outros monstrinhos. Um monstro de bolso... um pocket monster, ou pocketmon para encurtar (não queremos violar direitos autorais, aqui). Imagine que esse pocketmon possui nome, cor, vida, ataque e defesa. Então, podemos criar uma estrutura, como segue:

```
estrutura pocketmon
    cadeia nome
    cadeia cor
    inteiro vida
    inteiro ataque
    inteiro defesa
fimestrutura
```

Lembrando que poderíamos declarar as variáveis de mesmo tipo em uma única linha. Agora, podemos criar uma variável do tipo `pocketmon` e preencher os seus respectivos valores:

```
início
     pocketmon p
     escreva(“Digite o nome do pocketmon:”)
     leia(p.nome)
     escreva(“Digita a cor do pocketmon:”)
     leia(p.cor)
     escreva(“Digite a vida do pocketmon:”)
     leia(p.vida)
     escreva(“Digite o ataque do pocketmon:”)
```

```
        leia(p.ataque)
        escreva("Digite a defesa do pocketmon:")
        leia(p.defesa)
...
```

Mas, e se quiséssemos representar todos os pocketmons de um torneio? Ou todos os pocketmons que existem? E agora vem a magia! Lembra quando queríamos representar todas as notas de uma turma? O que fizemos? Criamos um vetor de nota em que cada posição do vetor armazena um elemento do tipo real.

Aqui, faremos a mesma coisa! Criaremos um vetor de pocketmon, onde cada posição do vetor precisa armazenar um elemento do tipo pocketmon! A ideia é simples!

Duas perguntas devem vir à nossa cabeça: como fazemos para declarar um vetor de pocketmon? E como podemos manipular os elementos deste vetor?

Para a primeira pergunta, basta usar o conhecimento que temos da declaração de qualquer vetor! Se quiséssemos declarar um vetor de inteiros, primeiro declararíamos o tipo, depois damos um nome para a variável e o tamanho entre colchetes:

```
inteiro a[5]
```

Com pocketmons é a mesma coisa!

```
pocketmon a[5]
```

Deste modo, acabamos de criar um vetor de pocketmon de cinco elementos, chamado *a[]*.

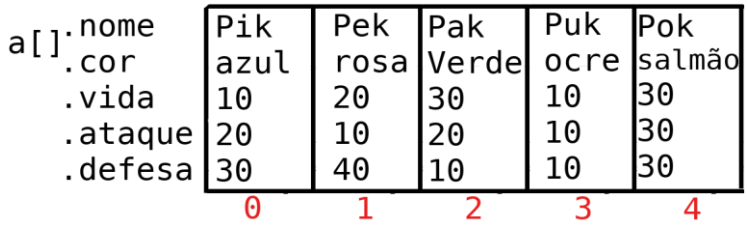
Em termos abstratos, podemos enxergar esse vetor da seguinte forma (Figura 13):

Figura 13 - Visualização de um vetor de pocketmons

Repare que o vetor *a* continua sendo uma variável composta homogênea porque todos os elementos são de um mesmo tipo (pocketmon). Mas, agora, cada elemento é uma variável composta heterogênea. Isso porque cada elemento, individualmente, é uma variável que possui campos de diferentes tipos. Então temos uma variável composta homogênea (vetor) que é COMPOSTA por 5 elementos que são variáveis compostas heterogêneas (os pocketmons).

De forma um pouco menos abstrata, podemos enxergar o vetor acima como segue (Figura 14):

Figura 14 - Vetor de pocketmons



Para atribuirmos valores aos campos do elemento da posição 0, por exemplo, fazemos da seguinte forma:

```
a[0].nome <- “Pik”
a[0].cor <- “azul”
a[0].vida <- 10
a[0].ataque <- 20
a[0].defesa <- 30
```

O mesmo para cada outro elemento, atualizando o índice entre colchetes.

Neste ponto, você pode estar pensando: se é um vetor e se eu quiser atribuir valores a todos os pocketmons, posso usar um laço ***para***! Exato!

O nome deste vetor não é muito apropriado porque *a* é apenas uma letra do alfabeto. Poderíamos chamar este vetor de um índice de pocketmons, ou um... pocketdex! Então, em vez de chamar o vetor de *a*, no exemplo a seguir chamaremos o vetor de *pocketdex*.

Vamos cria um algoritmo que atribua valores a todos os seus elementos e, em seguida, apresente todos os valores para o usuário:

```
1.    estrutura pocketmon
2.        cadeia nome
3.        cadeia cor
4.        inteiro vida
5.        inteiro ataque
6.        inteiro defesa
7.    fimestrutura
8.
9.    início
10.       pocketmon pocketdex[5]
11.       inteiro i
12.       para(i <- 0; i < 5; i++)faça
13.           escreva(“Digite o nome do pocketmon:”)
14.           leia(pocketdex[i].nome)
15.           escreva(“Digita a cor do pocketmon:”)
16.           leia(pocketdex[i].cor)
17.           escreva(“Digite a vida do pocketmon:”)
18.           leia(pocketdex[i].vida)
19.           escreva(“Digite o ataque do pocketmon:”)
20.           leia(pocketdex[i].ataque)
21.           escreva(“Digite a defesa do pocketmon:”)
22.           leia(pocketdex[i].defesa)
23.       fimpara
24.       para(i <- 0; i < 5; i++)faça
25.           escreva(“NOME: “, pocketdex[i].nome)
26.           escreva(“COR: “, pocketdex[i].cor)
27.           escreva(“VIDA: “, pocketdex[i].vida)
28.           escreva(“ATAQUE: “, pocketdex[i].ataque)
29.           escreva(“Defesa: “, pocketdex[i].defesa)
30.       fimpara
31.   fim
```

Agora precisamos praticar!

Exercícios de Vetores e Variáveis Compostas Heterogêneas

1. Crie um algoritmo que represente funcionários. Cada funcionário possui nome, endereço, cargo e salário (crie uma estrutura para isso). No algoritmo, declare um vetor de 100 funcionários. Em seguida, faça com que o algoritmo peça os dados de cada funcionário (use o laço PARA). Por fim, o algoritmo deve apresentar os dados de todos os 100 funcionários.

OBS: os exercícios a seguir são incrementais. Mas faça cada um desde o início, sem "copiar e colar".

2. Faça um algoritmo de pocketdex. O pocketdex deve armazenar cinco pocketmons (para isso, crie um vetor de pocketmon de 5 posições). Cada pocketmon deve possuir os campos nome, vida, ataque e defesa. O algoritmo deve pedir os dados dos cinco pocketmons (um pocketmon por vez). E, no fim, apresentá-los todos.

3. Faça um algoritmo de pocketdex. O pocketdex deve armazenar cinco pocketmons (para isso, crie um vetor de pocketmon de 5 posições). Cada pocketmon deve possuir os campos como o do exercício 2. O algoritmo deve pedir os dados dos cinco pocketmons (um pocketmon por vez). E, no fim, o algoritmo deve apresentar a seguinte mensagem: "Digite de 0 a 4, de qual pocketmon deseja ver os dados". Em seguida, o algoritmo deve apresentar os dados do pocketmon, conforme o número digitado.

4.Faça um algoritmo de pocketdex. O pocketdex deve armazenar cinco pocketmons (para isso, crie um vetor de pocketmon de 5 posições). Cada pocketmon deve possuir os campos como o do exercício 2. O algoritmo deve pedir os dados dos cinco pocketmons (um pocketmon por vez). E, no fim, o algoritmo deve apresentar a seguinte mensagem: "Digite de 0 a 4, de qual pocketmon deseja ver os dados". Em seguida, o algoritmo deve apresentar os dados do pocketmon, conforme o número digitado. Por fim, o algoritmo deve voltar a perguntar novamente qual pocketmon o usuário deseja ver. O algoritmo deverá continuar esta ação até que o usuário digite um valor que não esteja no intervalo entre 0 e 4.

5. Faça um algoritmo de pocketdex. O pocketdex deve armazenar cinco pocketmons (para isso, crie um vetor de pocketmon de 5 posições). Cada pocketmon

deve possuir os campos como o do exercício 2. O algoritmo deve pedir os dados dos cinco pocketmons (um pocketmon por vez). Em seguida, o algoritmo deve apresentar o seguinte menu: “1- Alterar dados do pocketmon; 2-Ver dados do pocketmon; Outro número para sair”. Uma vez que o usuário escolha entre 1 ou 2, o algoritmo deve pedir qual pocketmon (entre 0 e 4) o usuário deseja alterar/ver. Se o usuário digitar 1, o algoritmo deverá pedir para ele digitar os novos valores do pocketmon e retornar ao menu inicial. Se o usuário digitar 2, o algoritmo deverá apresentar os dados do usuário e voltar ao menu inicial. Isto deve se repetir até que o usuário digite um valor diferente de 1 e 2, no menu inicial.

Como é em algumas linguagens

Português Estruturado

```
estrutura pocketmon
    cadeia nome
    inteiro vida
    inteiro ataque
    inteiro defesa
fimestrutura

início
    pocketmon pocketdex
    inteiro i
    para(i <- 0; i < 5; i++)faça
        escreva(“Digite o nome do pocketmon:”)
        leia(pocketdex[i].nome)
        escreva(“Digite a vida do pocketmon:”)
        leia(pocketdex[i].vida)
        escreva(“Digite o ataque do pocketmon:”)
        leia(pocketdex[i].ataque)
        escreva(“Digite a defesa do pocketmon:”)
        leia(pocketdex[i].defesa)
    fimpara

    para(i <- 0; i < 5; i++)faça
        escreva(“NOME: “, pocketdex[i].nome)
        escreva(“VIDA: “, pocketdex[i].vida)
        escreva(“ATAQUE: “, pocketdex[i].ataque)
        escreva(“Defesa: “, pocketdex[i].defesa)
    fimpara
fim
```

C

```c
#include <stdio.h>
#include <string.h>

struct pocketmon{
    char nome[90];
    int vida;
    int ataque;
    int defesa;
};

typedef pocketmon pocketmon;

int main(){
    pocketmon pocketdex[5];
    int i;

    for(i=0; i<5; i++){
        printf("\nDigite o nome do pocketmon: ");
        gets(pocketdex[i].nome);
        printf("\nDigite a vida do pocketmon: ");
        scanf("%d",&pocketdex[i].vida);
        printf("\nDigite o ataque do pocketmon: ");
        scanf("%d",&pocketdex[i].ataque);
        printf("\nDigite a defesa do pocketmon: ");
        scanf("%d",&pocketdex[i].defesa);
        fflush(stdin);
    }
    for(i=0; i<5; i++){
        printf("\nNome: %s", pocketdex[i].nome);
        printf("\nVida: %d", pocketdex[i].vida);
        printf("\nAtaque: %d", pocketdex[i].ataque);
        printf("\nDefesa: %d", pocketdex[i].defesa);
    }
}
```

OBS: em C/C++, a criação de variáveis compostas homogêneas cujos elementos são variáveis compostas heterogêneas é uma transcrição literal do que vimos na seção.

A instrução ***fflush(stdin)*** é necessária sempre que se identifica que será lido um número e, em sequência, uma cadeia de caracteres. Entre as duas leituras, deve-se colocar essa instrução. No nosso exemplo, a última leitura no primeiro laço é de um número inteiro. Mas quando o laço reiniciar, será lido um novo nome, que é uma cadeia de caracteres. Então, usa-se *fflush*.

Isso ocorre porque a leitura de um número não remove o *<enter>* do *buffer* de leitura (porque o *<enter>* não é considerado um número). Já, a leitura de uma cadeia de caracteres considera o *<enter>* como um caractere. Então, na próxima leitura do nome, haverá o *<enter>* do número anteriormente digitado ainda sobrando no *buffer*. E, automaticamente, esse *<enter>* será usado como primeira entrada do usuário. E será entendido como se o usuário tivesse teclado o *<enter>* sem escrever nada antes.

C++

Pode ser o mesmo código do C, ou:

```
#include <iostream>
#include <string.h>

using namespace std;

struct pocketmon{
    string nome;
    int vida;
    int ataque;
    int defesa;
};

int main(){
    pocketmon pocketdex[5];
    int i;

    for(i=0; i<5; i++){
        cout << "\nDigitee o nome do pocketmon: ";
        getline(cin, pocketdex[i].nome);
        cout << "\nDigite a vida do pocketmon: ";
        cin >> pocketdex[i].vida;
        cout << "\nDigite o ataque do pocketmon: ";
        cin >> pocketdex[i].ataque;
        cout << "\nDigite a defesa do pocketmon: ";
        cin >> pocketdex[i].defesa;
        fflush(stdin);
    }
    for(i=0; i<5; i++){
        cout << "\nNome: " << pocketdex[i].nome;
        cout << "\nVida: " << pocketdex[i].vida;
        cout << "\nAtaque: " << pocketdex[i].ataque;
        cout << "\nDefesa: " << pocketdex[i].defesa;
    }
}
```

OBS: as mesmas observações de C se aplicam aqui.

Go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

type pocketmon struct {
    nome   string
    vida   int
    ataque int
    defesa int
}

func main() {

    var pocketdex [5]pocketmon
    var i int
    teclado := bufio.NewReader(os.Stdin)

    for i = 0; i < 5; i++ {
        fmt.Print("Digite o nome do pocketmon:")
        pocketdex[i].nome, _ = teclado.ReadString('\n')
        pocketdex[i].nome=strings.Replace(pocketdex[i].nome, "\n", "", -1)
        fmt.Print("Digite a vida do pocketmon:")
        fmt.Scanf("%d\n", &pocketdex[i].vida)
        fmt.Print("Digite o ataque do pocketmon:")
        fmt.Scanf("%d\n", &pocketdex[i].ataque)
        fmt.Print("Digite a defesa do pocketmon:")
        fmt.Scanf("%d\n", &pocketdex[i].defesa)
    }

    for i = 0; i < 5; i++ {
        fmt.Println("Nome:", pocketdex[i].nome)
        fmt.Println("Vida:", pocketdex[i].vida)
        fmt.Println("Ataque:", pocketdex[i].ataque)
        fmt.Println("Defesa:", pocketdex[i].defesa)
    }
}
```

OBS: em Go, a criação de variáveis compostas homogêneas cujos elementos são variáveis compostas heterogêneas é muito semelhante ao que vimos na seção, em Português Estruturado.

Ruby

```
pocketmon = Struct.new(:nome, :vida, :ataque, :defesa)
pocketdex = Array.new(5){pocketmon.new()}

for i in 0..4 do
    print "\nDigite o nome do pocketmon: "
    pocketdex[i].nome = gets.chomp
    print "\nDigite a vida do pocketmon: "
    pocketdex[i].vida = gets.chomp.to_i
    print "\nDigite o ataque do pocketmon: "
    pocketdex[i].ataque = gets.chomp.to_i
    print "\nDigite a defesa do pocketmon: "
    pocketdex[i].defesa = gets.chomp.to_i
end

for i in 0..4 do
    print "\nNome: ", pocketdex[i].nome
    print "\nVida: ", pocketdex[i].vida
    print "\nAtaque: ", pocketdex[i].ataque
    print "\nDefesa: ", pocketdex[i].defesa
end
```

OBS: a primeira linha cria a estrutura. Na segunda linha, é criado um vetor de 5 elementos de pocketmons. A instrução ***new*** cria um objeto do tipo pocketmon para cada posição do vetor. Sem essa instrução, cada posição estaria vazia, sem um objeto em cada posição.

Python

```
from dataclasses import dataclass, field
import string
import numpy as np

@dataclass
class pocketmon:
    nome: string = field(init=False)
    vida: int = field(init=False)
    ataque: int = field(init=False)
    defesa: int = field(init=False)

pocketdex = np.empty(5, dtype=pocketmon)

for i in range(5):
    pocketdex[i] = pocketmon()
    pocketdex[i].nome = input('Digite o nome do pockemon: ')
    pocketdex[i].vida = input('Digite a vida do pockemon: ')
    pocketdex[i].ataque = input('Digite o ataque do pockemon: ')
    pocketdex[i].defesa = input('Digite a defesa do pockemon: ')

for i in range(5):
    print('Nome: ' + pocketdex[i].nome)
    print('Vida: ' + pocketdex[i].vida)
    print('Ataque: ' + pocketdex[i].ataque)
    print('defesa: ' + pocketdex[i].defesa)
```

Obs: criamos um vetor de 5 pocketmons. Mas, inicialmente, cada posição está vazia, sem nenhum pocketmon. Então, para cada posição do vetor, primeiro cria-se um novo pocketmon e o coloca na posição respectiva do laço:

```
pocketdex[i] = pocketmon()
```

Para, depois, atribuir o nome, vida, ataque e defesa.

A ideia é:

-Criar um novo pocketmon e colocá-lo na posição 0 de pocketdex.

-Agora, atribuir um nome, vida, ataque e defesa a esse novo pocketmon da posição 0 de pocketdex.

-Criar um novo pocketmon e colocá-lo na posição 1 de pocketdex.

-Agora, atribuir um nome, vida, ataque e defesa a esse novo pocketmon da posição 1 de pocketdex.

-E assim até a posição 4.

Variável Composta Heterogênea com um campo de vetor

O inverso da seção passada também é muito comum! A criação de uma estrutura que tenha um dos campos que é um vetor.

Nós já vimos um exemplo em que é interessante criar um vetor que represente as notas de um aluno. Agora, podemos pensar em criar um algoritmo que represente um aluno, com nome, RA e suas quatro notas do semestre!

Podemos pensar de forma separada: uma estrutura para nome e ra que se chame aluno e um vetor de notas de tamanho 4:

```
1.    estrutura aluno
2.        cadeia nome
3.        inteiro RA
4.    fimestrutura
5.
6.    início
7.        aluno a
8.        real notas[4]
9.        …
```

Esta é uma solução. Mas teremos um inconveniente se precisarmos representar dois alunos, por exemplo, porque precisaremos declarar dois vetores. É gerenciável, é possível, mas não é conveniente.

Se pensarmos bem, como estas 4 notas pertencem ao aluno, faz muito sentido declarar este vetor dentro da própria estrutura aluno, de modo que o vetor seja mais um campo de aluno:

```
1.    estrutura aluno
2.        cadeia nome
3.        inteiro RA
4.        real notas[4]
5.    fimestrutura
6.
7.    início
8.        aluno a
```

Uma vez que declaramos o aluno *a*, automaticamente teremos o campo notas, que permite o armazenamento de 4 valores! Bem mais prático. No exemplo anterior vimos como se cria a estrutura e como declaramos uma variável a partir dela (a variável *a* é uma variável composta heterogênea que tem, como um de seus

campos, uma variável composta homogênea, que é o vetor de notas). Para atribuir valores aos seus respectivos campos, faremos da seguinte maneira:

```
1.     a.nome <- “Miguel”
2.     a.RA <- 123456
3.     a.notas[0] <- 10
4.     a.notas[1] <- 9
5.     a.notas[2] <- 8
6.     a.notas[3] <- 7.5
```

Ou, como estamos usando um vetor e queremos que o usuário digite os valores:

```
1.  início
2.     aluno a
3.     inteiro i
4.     escreva(“Digite o nome: ”)
5.     leia(a.nome)
6.     escreva(“Digite o RA: ”)
7.     leia(a.RA)
8.     para(i <- 0;i<4;i++)faça
9.         escreva(“Digite a nota “, i, “: “)
10.        leia(a.notas[i])
11.    fimpara
12. fim
```

Podemos seguir realizando mais combinações!

Imagine que você quer representar uma turma inteira de 30 alunos, cada um com suas 4 notas!

Não é difícil. Primeiro, imagine como você iria representar uma turma de 30 alunos. Fizemos algo semelhante na seção anterior com pocketmons: criamos uma estrutura pocketmon. Quando formos declarar a variável, nós a declaramos como um vetor de pocketmons. Aqui fazemos o mesmo: criamos uma estrutura aluno e declararemos um vetor de 30 alunos.

Agora, precisamos apenas perceber que as notas são um novo campo de aluno.

```
1.   estrutura aluno
2.     cadeia nome
3.     inteiro RA
4.     real notas[4]
5.   fimestrutura
6.
7.   início
8.      aluno a[30]
9.      inteiro i, j
10.     para(i <- 0; i < 30; i++)faça
11.         escreva(“Digite o nome: ”)
12.         leia(a[i].nome)
13.         escreva(“Digite o RA: ”)
14.         leia(a[i].RA)
15.         para(j <- 0; j < 4; j++)faça
16.             escreva(“Digite a nota “, j, “: “)
17.             leia(a[i].notas[j])
18.         fimpara
19.     fimpara
20.     para(i <- 0; i < 30; i++)faça
21.         escreva(“Nome do aluno “, i, “: “, a[i].nome)
22.         escreva(“RA do aluno “, i, “: “, a[i].RA)
23.         para(j <- 0; j < 4; j++)faça
24.             escreva(“Nota “, j, “: “, a[i].notas[j])
25.         fimpara
26.     fimpara
27.  fim
```

Precisamos apenas reparar em alguns pontos:

Na estrutura aluno, consideramos as notas como um campo de aluno. Isso é muito interessante em termos de organização. Se pensássemos em separar as notas da estrutura aluno, ou precisaríamos criar 30 vetores de notas ou precisaríamos criar uma matriz de 30x4, em que cada linha representasse a nota de um aluno e cada coluna a nota. Isso seria melhor que a opção de criar 30 vetores, com certeza! Mas teríamos duas variáveis separadas (um vetor de aluno e uma matriz de notas), o que é menos organizado do que está o código acima.

Como criamos um vetor de alunos *a*, para acessar cada aluno, precisamos usar colchetes. Então, para acessar os campos do aluno da posição 4, precisamos fazer o seguinte:

```
a[4].nome <- “Murilo”
a[4].RA <- 12345
```

E como notas é um vetor, precisamos usar colchete para acessar qual nota queremos atribuir (portanto teremos que usar colchetes depois de notas) e colchetes

para acessar o respectivo aluno. Assim, se quisermos acessar cada nota do aluno da posição 4, precisaríamos fazer:

```
a[4].notas[0] <- 10
a[4].notas[1] <- 8
a[4].notas[2] <- 7.5
a[4].notas[3] <- 9
```

Não é complicado, mas é importante treinarmos para fixarmos na memória.

Exercícios de Estruturas com Vetores como Campos

1. Imagine que você queira criar um algoritmo em que uma pessoa (que possui um nome e endereço) possa comprar 5 números de uma rifa. Então, essa pessoa possui um nome, um endereço e um vetor de inteiro de 5 posições para representar os números de suas rifas. Crie um algoritmo que possua uma estrutura como a descrita, uma variável declarada a partir dessa pessoa, que peça cada valor para o usuário e, por fim, apresente todos os valores na tela.

2. Faça um algoritmo que represente um cliente de moda. Esse cliente precisa ter nome, endereço e 4 cores preferidas para serem usadas mais tarde pelo designer de moda. Então, o algoritmo terá uma estrutura com os campos nome, endereço e um vetor de 4 posições para armazenar as cores (um vetor do tipo cadeia, em que cada posição representa uma cor). O algoritmo terá uma variável declarada a partir desta estrutura. O algoritmo precisará pedir os valores de cada campo e, por fim, apresentá-los na tela.

3. Faça um algoritmo que represente um personagem de RPG. Todo personagem tem um nome, Idade, Vida, Ataque, Defesa e 3 habilidades. Por exemplo, O Alexander tem 35 anos, 10 pontos de vida, 20 pontos de ataque, 5 pontos de defesa e suas habilidades são: invisibilidade, magia e teletransporte. O algoritmo deve pedir estes dados e apresentá-los na tela. As habilidades devem ser consideradas como um vetor de 3 posições de cadeia.

4. Faça um algoritmo que represente uma lista de 5 personagens de RPG. Os personagens devem seguir a estrutura do exercício anterior. A diferença é que o algoritmo deverá pedir os dados dos 5 personagens para, depois, apresentá-los na tela.

Como é em algumas linguagens

Português Estruturado

```
estrutura aluno
    cadeia nome
    inteiro RA
    real notas[4]
fimestrutura

início
    aluno a
    inteiro i
    escreva(“Digite o nome: ”)
    leia(a.nome)
    escreva(“Digite o RA: ”)
    leia(a.RA)
    para(i <- 0; i < 4; i++)faça
        escreva(“Digite a nota “, i+1, “: “)
        leia(a.notas[i])
    fimpara

    escreva(“Nome: “, a.nome)
    escreva(“RA: “, a.nome)
    para(i <- 0; i < 4; i++)faça
        escreva(“Nota “, i+1, “:  “, a.notas[i])
    fimpara
fim
```

C

```c
#include <stdio.h>
#include <string.h>

struct aluno{
    char nome[250];
    int ra;
    float notas[4];
};

typedef aluno aluno;

int main(){
    aluno a;
    int i;

    printf("\nDigite o nome: ");
    gets(a.nome);
    printf("\nDigite o RA: ");
    scanf("%d",&a.ra);

    for(i=0; i<4; i++){
        printf("\nDigite a nota %d: ", i+1);
        scanf("%f", &a.notas[i]);
    }

    printf("\nNome: %s", a.nome);
    printf("\nRA: %d", a.ra);
    for(i=0; i<4; i++){
        printf("\nNota %d: %.2f", i+1, a.notas[i]);
    }
}
```

OBS: Em C, a criação de vetores dentro de estruturas é semelhante ao apresentado na seção anterior, em Português Estruturado. Aliás!!! Aliás, já estávamos usando vetores dentro de estruturas em C desde o primeiro exemplo. Afinal, C não possui o tipo *string* e usamos um vetor de caracteres em seu lugar!

C++

Pode ser o mesmo código do C, ou:

```
#include <iostream>
#include <string>

using namespace std;

struct aluno{
    string nome;
    int ra;
    float notas[4];
};

int main(){
    aluno a;
    int i;

    cout << "\nDigite o nome: ";
    getline(cin, a.nome);
    cout << "\nDigite o RA: ";
    cin >> a.ra;

    for(i=0; i<4; i++){
        cout << "\nDigite a nota: " << i+1 << ": ";
        cin >> a.notas[i];
    }

    cout << "\nNome: " << a.nome;
    cout << "\nRA: " << a.ra;
    for(i=0; i<4; i++){
        cout << "\nNota " << i+1 << ": " << a.notas[i];
    }
}
```

Obs: Diferente de C, C++ possui tipo *string*. E a criação de estruturas com campos que são vetores é similar ao apresentado na seção anterior, em Português Estruturado.

Go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

type aluno struct {
    nome  string
    ra    int
    notas [4]float32
}

func main() {
    var a aluno
    var i int
    teclado := bufio.NewReader(os.Stdin)

    fmt.Print("Digite o nome:")
    a.nome, _ = teclado.ReadString('\n')
    a.nome = strings.Replace(a.nome, "\n", "", -1)
    fmt.Print("Digite o RA:")
    fmt.Scanf("%d", &a.ra)

    for i = 0; i < 4; i++ {
        fmt.Print("Digite a nota", i+1, ":")
        fmt.Scanf("\n%f", &a.notas[i])
    }

    fmt.Print("\nNome:", a.nome)
    fmt.Print("\nRA:", a.ra)

    for i = 0; i < 4; i++ {
        fmt.Print("\nNota", i+1, ":", a.notas[i])
    }
}
```

OBS: Assim como em C/C++, em Go, a criação de vetores dentro de estruturas é semelhante ao apresentado na seção anterior, em Português Estruturado.

Ruby

```
aluno = Struct.new(:nome, :ra, :notas)
a = aluno.new()

a.notas = Array.new(4)

print "\nDigite o nome: "
a.nome = gets.chomp
print "\nDigite o RA: "
a.ra = gets.chomp.to_i

for i in 0..3 do
    print "\nDigite a nota ", i+1, ": "
    a.notas[i] = gets.chomp.to_f
end

print "\nNome: ", a.nome
print "\nRA: ", a.ra
for i in 0..3 do
    print "\nNota ", i+1, ": ", a.notas[i]
end
```

OBS: Em Ruby, a definição do vetor não se dá dentro da estrutura. Primeiro criamos a estrutura em si, com o campo notas, sem um tipo definido. Em seguida, criamos uma variável a partir da estrutura (até aí, normal). Após criada a variável, definimos que o campo (notas) será um novo vetor de 4 posições. Há outras formas de se criar um vetor dentro de uma estrutura em Ruby, mas essa é uma forma simples.

Python

```
from dataclasses import dataclass, field
import string
import numpy as np

@dataclass
class aluno:
    nome: string = field(init=False)
    ra: int = field(init=False)
    notas: np.ndarray = np.zeros(4)

a = aluno()

a.nome = input('Digite o nome: ')
a.ra = int(input('Digite o ra: '))
for i in range(4):
    a.notas[i] = input('Digite a nota ' + str(i+1) + ': ')

print('Nome: ', a.nome)
print('RA: ', a.ra)
for i in range(4):
    print('Nota ' + str(i+1) + ": " + str(a.notas[i]))
```

Obs: em Python, podemos criar um vetor usando a instrução ***ndarray*** da biblioteca numpy. Em seguida, definimos que cada posição receberá automaticamente o valor 0. Ou poderíamos usar:

```
notas: np.ndarray = np.empty(4)
```

Mas não é possível criar o vetor diretamente, como fizemos em outros exemplos. A linha abaixo NÃO FUNCIONA!

```
notas: np.empty(4)
```

Variável Composta Heterogênea dentro de Variável Composta Heterogênea

E é possível criar uma variável composta heterogênea que tem um campo que também é uma variável composta heterogênea! E isso é muito comum.

Imagine que você queira fazer um sistema que represente carros e aviões, como um jogo. Nesse sistema, um carro é representado por marca, modelo, cor, consumo e raio do pneu. O avião é representado por modelo, envergadura, altura máxima de vôo e raio do pneu.

É fácil criar uma estrutura para cada veículo:

```
1. estrutura carro
2.     cadeia marca
3.     cadeia modelo
4.     real consumo
5.     real raiopneu
6. fim
```

```
7.   estrutura aviao
8.      cadeia modelo
9.      real envergadura
10.     real altura
11.     real raiopneu
12.  fim
```

Perfeito!

Mas imagine que, neste sistema, queiramos representar mais dados sobre os pneus, como pressão máxima, largura e cor. Então, poderíamos adaptar nossas estruturas da seguinte forma:

```
1. estrutura carro
2.     cadeia marca
3.     cadeia modelo
4.     real consumo
5.     real raiopneu
6.     real pressaopneu
7.     real largurapneu
8.     cadeia corpneu
9. fim
```

```
10. estrutura aviao
11.    cadeia modelo
12.    real envergadura
13.    real altura
14.    real raiopneu
15.    real pressaopneu
16.    real largurapneu
17.    cadeia corpneu
18.  fim
```

Esta é uma solução possível.

Mas começamos a ter um inconveniente: há quatro campos que são comuns tanto ao carro quanto ao avião e que representam um pneu. Se quisermos criar uma

terceira estrutura que também possua pneu (como uma moto, bicicleta, patinete, triciclo, etc...), iremos repetir estes quatro campos em cada estrutura.

Consegue imaginar uma forma de organizar melhor estes dados?

Podemos criar uma estrutura pneu, antes das demais estruturas! E, dentro da estrutura carro e avião, criamos um campo do tipo pneu! Aqui, chamarei a estrutura de pneumático para chamar a variável criada a partir dela de pneu.

```
1.  estrutura pneumatico
2.      real raio
3.      real pressao
4.      real largura
5.      cadeia cor
6.  fim
```

```
7.  estrutura carro
8.      cadeia marca
9.      cadeia modelo
10.     real consumo
11.     pneumatico pneu
12. fim
```

```
13. estrutura aviao
14.     cadeia modelo
15.     real envergadura
16.     real altura
17.     pneumatico pneu
18. fim
```

Então criamos uma estrutura pneumático (que será um novo tipo de dados) que possui os campos raio, pressão, largura e cor; e criamos, em cada outra estrutura, um campo pneu, do tipo pneumático.

Observe como as duas outras estruturas se tornaram mais organizadas. Além disso, há uma vantagem em relação à manutenção e atualização do sistema. Vamos supor que depois que o sistema está completo, você perceba que seria importante criar mais um campo relacionado ao pneu, como marca do pneu.

Na primeira solução, teríamos que criar esse campo em cada estrutura que precisa representar um pneu: um campo a mais em avião, carro, moto, etc...

Por outro lado, com a segunda solução, em que criamos uma estrutura pneumático, basta inserir um campo nesta estrutura! E todos as outras estruturas terão um campo pneu, como os campos definidos da estrutura pneumático!

Agora a pergunta é: como acessamos os campos no nosso algoritmo? Completamos o código anterior como a seguir:

```
1.  início
2.      carro c
3.      aviao a
4.      c.marca <- “Folks”
5.      c.modelo <- “IC21”
6.      c.consumo <- 6.2
```

```
7.      c.pneu.raio <- 15
8.      c.pneu.pressao <- 32
9.      c.pneu.largura <- 12
10.     c.pneu.cor <- “preto”

11.     a.modelo <- “Cisna”
12.     a.envergadura <- 15
13.     a.altura <- 10000
14.     a.pneu.raio <- 25
15.     a.pneu.pressao <- 42
16.     a.pneu.largura <- 22
17.     a.pneu.cor <- “branco”
18. FIM
```

Nota-se, pelas linhas 7 a 10 e 14 a 17, que a lógica do uso é bem consistente!

Se pneu é uma variável composta heterogênea, então, é preciso acessar seus campos separando-os do identificador da variável (pneu) pelo ponto (.), como *pneu.raio*. Seguindo a mesma lógica, se pneu é um campo da variável a, então esse campo também deve ser separado do identificador da variável (a) por um ponto (.), como a.pneu.raio.

Agora é hora de praticar.

Exercícios de Variável Composta Heterogênea dentro de Variável Composta Heterogênea

1. Faça um algoritmo que represente funcionários e alunos. Todo funcionário possui nome, endereço, CPF e salário. Todo aluno possui nome, endereço, CPF e RA. O algoritmo deve pedir os dados de um funcionário e os de um aluno e apresentá-los na tela.

OBS: não é crime criar as estruturas funcionário e aluno, cada uma com um campo nome, endereço e CPF. De fato, é até mais interessante. Mas neste exercício, identifique os campos que são comuns às duas estruturas, crie uma outra estrutura chamada pessoa com estes campos e crie o campo do tipo pessoa tanto em aluno quanto funcionário. Temos duas estruturas que terão um campo que é uma variável composta heterogênea criada do tipo pessoa. Linguagens orientadas a objeto são mais naturais para este tipo de situação.

2. Faça um algoritmo que permita representar pocketmons. Um pocketmon tem espécie, vida, ataque e defesa. O algoritmo deve, também, representar o treinador de pocketmon. O treinador tem nome, idade e pode carregar um pocketmon. O algoritmo deve pedir para o usuário digitar os dados do treinador e de seu respectivo pocketmon. Por fim, os dados devem ser apresentados na tela.

OBS: Como todo treinador carrega seu próprio pocketmon, então, crie primeiro a estrutura pocketmon. Quando for criar a estrutura treinador, um dos campos do treinador será o pocketmon que carrega. Teremos uma estrutura treinador que possui um campo que é uma variável composta heterogênea do tipo pocketmon.

3. Faça um algoritmo como o anterior. Mas, desta vez, cada treinador carrega consigo 5 pocketmons! O algoritmo deve pedir os dados do treinador (e consequentemente dos 5 pocketmons) e apresentá-los na tela.

OBS: a ideia é a mesma do exercício 2. Mas o campo pocketmon que será criado dentro da estrutura treinador será um vetor de pocketmons. Temos uma estrutura treinador que possui um campo que é um vetor (portanto uma variável composta homogênea) de elementos que são variáveis compostas heterogêneas).

4. Faça um algoritmo como o anterior. Mas agora o algoritmo representa um campeonato de pocketmons! Então, cada treinador terá seus 5 pocketmons. O algoritmo deve pedir os dados sobre todos os treinadores e seus respectivos pocketmons. Para isso, no bloco principal do algoritmo (dentro do INÍCIO-FIM), declare um vetor de 10 treinadores.

OBS: temos um vetor de treinadores. Isso significa que temos uma variável composta homogênea cujos elementos são variáveis compostas heterogêneas. O tipo treinador, por sua vez, contém um campo que é uma variável composta homogênea de pocketmons, isto é, uma variável composta homogênea de elementos que são variáveis compostas heterogêneas.

Como é em algumas linguagens

Português Estruturado

```
estrutura pneumatico
    real raio
    real pressao
    real largura
fimestrutura

estrutura carro
    cadeia marca
    cadeia modelo
    real consumo
    pneumatico pneu
fimestrutura

início
    carro c
    escreva("Digite a marca do carro: ")
    leia(c.marca)
    escreva("Digite o modelo do carro: ")
    leia(c.modelo)
    escreva("Digite o consumo: ")
    leia(c.consumo)
    escreva("Digite o raio do pneu")
    leia(c.pneu.raio)
    escreva("Digite a pressão do pneu: ")
    leia(c.pneu.pressao)
    escreva("Digite a largura do pneu: ")
    leia(c.pneu.largura)

    escreva("Marca: ", c.marca)
    escreva("Modelo: ", c.modelo)
    escreva("Consumo: ", c.consumo)
    escreva("Raio: ", c.pneu.raio)
    escreva("Pressão: ", c.pneu.pressao)
    escreva("Largura: ", c.pneu.largura)
fim
```

C

```c
#include <stdio.h>
#include <string.h>


struct pneumatico{
    float raio;
    float pressao;
    float largura;
};

typedef pneumatico pneumatico;

struct carro{
    char marca[10];
    char modelo[10];
    float consumo;
    pneumatico pneu;
};

typedef carro carro;

int main(){
    carro c;
    printf("\nDigite a marca do carro: ");
    gets(c.marca);
    printf("\nDigite o modelo do carro: ");
    gets(c.modelo);
    printf("\nDigite o consumo: ");
    scanf("%f",&c.consumo);
    printf("\nDigite o raio do pneu: ");
    scanf("%f", &c.pneu.raio);
    printf("\nDigite a pressão do pneu: ");
    scanf("%f", &c.pneu.pressao);
    printf("\nDigite a largura do pneu: ");
    scanf("%f", &c.pneu.largura);

    printf("\nMarca: %s", c.marca);
    printf("\nModelo: %s", c.modelo);
    printf("\nConsumo: %.2f", c.consumo);
    printf("\nRaio: %.2f", c.pneu.raio);
    printf("\nPressão: %.2f", c.pneu.pressao);
    printf("\nLargura: %.2f", c.pneu.largura);
    return 0;
}
```

Obs: a criação de variáveis compostas heterogêneas dentro de outra estrutura segue o mesmo padrão do apresentado na seção anterior, em Português Estruturado.

C++

Pode ser o mesmo código do C, ou:

```
#include <iostream>
#include<string>

using namespace std;

struct pneumatico{
    float raio;
    float pressao;
    float largura;
};

struct carro{
    string marca;
    string modelo;
    float consumo;
    pneumatico pneu;
};

int main(){
    carro c;
    cout << "\nDigite a marca do carro: ";
    getline(cin, c.marca);
    cout << "\nDigite o modelo do carro: ";
    getline(cin, c.modelo);
    cout << "\nDigite o consumo: ";
    cin >> c.consumo;
    cout << "\nDigite o raio do pneu: ";
    cin >> c.pneu.raio;
    cout << "\nDigite a pressão do pneu: ";
    cin >> c.pneu.pressao;
    cout << "\nDigite a largura do pneu: ";
    cin >> c.pneu.largura;

    cout << "\nMarca: " << c.marca;
    cout << "\nModelo: " << c.modelo;
    cout << "\nConsumo: " << c.consumo;
    cout << "\nRaio: " << c.pneu.raio;
    cout  << "\nPressão: " << c.pneu.pressao;
    cout << "\nLargura: " << c.pneu.largura;
    return 0;
}
```

Obs: a criação de variáveis compostas heterogêneas dentro de outra estrutura segue o mesmo padrão do apresentado na seção anterior, em Português Estruturado.

Go

```
package main

import (
	"bufio"
	"fmt"
	"os"
)

type pneumatico struct {
	raio    float32
	pressao float32
	largura float32
}

type carro struct {
	marca   string
	modelo  string
	consumo float32
	pneu    pneumatico
}

func main() {
	var c carro
	teclado := bufio.NewReader(os.Stdin)

	fmt.Print("Digite a marca do carro: ")
	c.marca, _ = teclado.ReadString('\n')
	fmt.Print("Digite o modelo do carro: ")
	c.modelo, _ = teclado.ReadString('\n')
	fmt.Print("Digite o consumo: ")
	fmt.Scanf("%f\n", &c.consumo)
	fmt.Print("Digite o raio do pneu: ")
	fmt.Scanf("%f\n", &c.pneu.raio)
	fmt.Print("Digite a pressão do pneu: ")
	fmt.Scanf("%f\n", &c.pneu.pressao)
	fmt.Print("Digite a largura do pneu: ")
	fmt.Scanf("%f\n", &c.pneu.largura)

	fmt.Print("Marca: ", c.marca)
	fmt.Print("Modelo: ", c.modelo)
	fmt.Print("Consumo: ", c.consumo)
	fmt.Print("\nRaio: ", c.pneu.raio)
	fmt.Print("\nPressão: ", c.pneu.pressao)
	fmt.Print("\nLargura: ", c.pneu.largura)
}
```

Obs: o procedimento é muito semelhante ao visto na seção anterior, respeitando-se as pequenas diferenças de declaração desta linguagem.

Ruby

```
pneumatico = Struct.new(:raio, :pressao, :largura)
carro = Struct.new(:marca, :modelo, :consumo, :pneu)

c = carro.new()
c.pneu = pneumatico.new()

print "\nDigite a marca do carro: "
c.marca = gets.chomp
print "\nDigite o modelo do carro: "
c.modelo = gets.chomp
print "\nDigite o consumo: "
c.consumo = gets.chomp.to_f
print "\nDigite o raio do pneu: "
c.pneu.raio = gets.chomp.to_f
print "\nDigite a pressão do pneu: "
c.pneu.pressao = gets.chomp.to_f
print "\nDigite a largura do pneu: "
c.pneu.largura = gets.chomp.to_f

print "\nMarca: ", c.marca
print "\nModelo: ", c.modelo
print "\nConsumo: ", c.consumo
print "\nRaio: ", c.pneu.raio
print "\nPressão: ", c.pneu.pressao
print "\nLargura: ", c.pneu.largura
```

Obs: aqui não há muita novidade se comparado à criação de vetores em estruturas. Como a linguagem não é fortemente tipada, primeiro criamos o campo pneu na estrutura carro (sem definir seu tipo) para, depois, especificar que aquele campo é uma nova estrutura do tipo pneumatico.

Python

```
from dataclasses import dataclass, field
import string

@dataclass
class pneumatico:
    raio: float = field(init=False)
    pressao: float = field(init=False)
    largura: float = field(init=False)

@dataclass
class carro:
    marca: string = field(init=False)
    modelo: string = field(init=False)
    consumo: float = field(init=False)
    pneu: pneumatico = field(init=False)

c = carro()
c.pneu = pneumatico()

c.marca = input('Digite a marca do carro:')
c.modelo = input('Digite o modelo do carro:')
c.consumo = float(input('Digite o consumo do carro:'))
c.pneu.raio = float(input('Digite o raio do pneu:'))
c.pneu.pressao = float(input('Digite a pressão do pneu:'))
c.pneu.largura = float(input('Digite a largura do pneu:'))

print ('Marca: ', c.marca)
print ('Modelo: ', c.modelo)
print ('Consumo: ', c.consumo)
print ('Raio: ', c.pneu.raio)
print ('Pressão: ', c.pneu.pressao)
print ('Largura: ', c.pneu.largura)
```

Obs: em Python, o que usamos como estrutura é, na verdade, uma classe de dados. E classes precisam ser instanciadas ou o campo será considerado como nulo. Isto é, o campo pneu é do tipo pneumático. Mas não há um objeto ainda. É como se, no momento da criação da classe carro, estivéssemos apenas especificando que possui pneu, mas sem colocar um pneu ainda no carro. A linha:

```
c.pneu = pneumatico()
```

Faz com que um novo pneu seja de fato criado e colocado no campo pneu. Antes disso, considere que havia o espaço esperando por um pneu, mas sem o pneu, ainda.

Capítulo 3: FUNÇÕES

Com aquilo que aprendemos até o momento, podemos fazer algoritmos realmente grandes, com diversas funcionalidades. Quanto mais tarefas o algoritmo precisar executar, maior a quantidade de linhas que o programador precisará planejar.

Da forma como estamos desenvolvendo nossos algoritmos, com todas as instruções entre o bloco ***início*** e ***fim***, aos poucos teremos alguns problemas, como organização, manutenção e repetição de instruções. Imagine que seu algoritmo tem tantas tarefas (como pedidos para o usuário digitar dados, vários cálculos, vários menus e diferentes apresentações de valores na tela). Não é difícil um algoritmo ultrapassar centenas ou milhares de instruções. Agora, imagine que todo esse código foi escrito dentro de um único bloco ***início-fim***.

Com essa visão em mente, tente imaginar que você precisará, em algum momento, realizar alguma manutenção no código. É fácil perceber que poderá perder bastante tempo apenas para encontrar a parte que precisa alterar.

Também é muito comum que o algoritmo precise realizar uma mesma tarefa, em diferentes partes do código, como no exemplo abaixo:

```
início
    inteiro num, i
    escreva("**********************")
    escreva("*                    *")
    escreva("* Digite um número *")
    escreva("*                    *")
    escreva("**********************")
    leia(num)

    para(i <- 0; i < num; i++)faça
        escreva(i)
    fimpara
    escreva("**********************")
    escreva("*                    *")
    escreva("* Digite um número *")
    escreva("*                    *")
    escreva("**********************")
    leia(num)
```

```
    se(num>=10 && num<=-10)então
        escreva(“Valor com mais de um dígito”)
    senão
        escreva(“Valor com apenas um dígito”)
    fimse

    escreva(“**********”)
    escreva(“*        *”)
    escreva(“* Digite um número *”)
    escreva(“*        *”)
    escreva(“**********”)
    leia(num)

    para(i <- num; i >= 0; i <- i - 2)faça
        escreva(i)
    fimpara

    escreva(“**********”)
    escreva(“*        *”)
    escreva(“* Digite um número *”)
    escreva(“*        *”)
    escreva(“**********”)
    leia(num)

    se(num%2==0)então
        escreva(“O número é par!”)
    senão
        escreva(“O número é ímpar!”)
    fim
fim
```

Ainda que seja um algoritmo pequeno, nota-se que há uma tarefa repetida quatro vezes no código (pedir para o usuário digitar um número). Além de ser trabalhoso repetir uma mesma sequência de instruções várias vezes, ainda há um problema: imagine que seu cliente não gostou da mensagem e quer que você mude o que está escrito justamente nas instruções que estão repetidas. Você terá que fazer a alteração quatro vezes, neste exemplo! E isso porque é um algoritmo pequeno. E se uma mesma tarefa fosse executada em centenas de partes diferentes do código? Você teria que alterar centenas de vezes!

A solução para evitar estes problemas apresentados até o momento, é dividir o código em partes menores de forma que cada parte deva realizar uma ação pontual. Deste modo, o algoritmo ficará modularizado, isto é, dividido em módulos.

Segundo a ISO 24765, um módulo é uma parte logicamente separável de um programa. Ou, ainda, uma unidade de programa que é discreta e identificável.

Aproveitando o exemplo anterior, vamos separar o código que se repete e escrevê-lo apenas uma vez. Vamos, por enquanto, apenas separar as instruções *escreva*. Também daremos um nome a este módulo para que possa ser identificável, como na definição do parágrafo anterior (chamaremos de apresentarMenu).

```
apresentarMenu()
    escreva(“**********”)
    escreva(“*        *”)
    escreva(“* Digite um número *”)
    escreva(“*        *”)
    escreva(“**********”)
fim
```

Não é importante, neste momento, entender porque há um abre e fecha parênteses depois do nome do módulo. O que importa é que este código foi criado, tem um nome e podemos reaproveitá-lo no nosso novo corpo principal do algoritmo, bastando chamá-lo nos pontos em que queiramos que ele seja executado. Abaixo, segue um exemplo completo com as linhas enumeradas para facilitar a explicação:

```
1. apresentarMenu()
2.     escreva(“**********”)
3.     escreva(“*        *”)
4.     escreva(“* Digite um número *”)
5.     escreva(“*        *”)
6.     escreva(“**********”)
7. fim

8.   inicio
9.       inteiro num, i

10.      apresentarMenu()

11.      leia(num)
12.      para(i<-0; i<num;i<-i+1)faça
13.          escreva(i)
14.      fimpara
```

```
15.      apresentarMenu()

16.      leia(num)
17.      se(num>=10 && num<=-10)então
18.          escreva("Valor com mais de um dígito")
19.      senão
20.          escreva("Você com apenas um dígito")
21.      fimse

22.      apresentarMenu()

23.      leia(num)
24.      para(i<-num; i>=0;i<-i-2)faça
25.          escreva(i)
26.      fimpara

27.      apresentarMenu()

28.      leia(num)
29.      se(num%2==0)então
30.          escreva("O número é par!")
31.      senão
32.          escreva("O número é ímpar!")
33.      fimse
34.  fim
```

Repare que o módulo apresentarMenu() foi criado antes do bloco de instruções ***início-fim***. Se o módulo pode ser criado antes ou depois do ***início-fim*** vai depender da linguagem. Por enquanto, consideremos que os módulos devam ser criados sempre antes deste bloco.

Agora precisamos entender como o código funciona: o algoritmo sempre começará pelo ***início***! Isso mesmo, ainda que haja instruções antes do ***início***, como as linhas de 1 a 7 em verde no nosso exemplo.

O algoritmo continuará pela declaração das variáveis (linha 9) e, quando a linha 10 for executada, a mágica acontecerá. Dizemos que, nesta linha, há uma chamada para o módulo apresentarMenu(). Assim, o algoritmo pára nesta linha e começa a executar o módulo apresentarMenu(), da linha 1 à linha 7. Quando este

módulo terminar de ser executado e chegar à linha 7, vem a outra mágica: o algoritmo volta exatamente para a linha em que o módulo foi chamado, isto é, a linha 10.

O código continuará para as linhas 11, 12, 13, 14 até que o laço ***para*** termine. Quando a linha 15 for executada, novamente o algoritmo pára de ser executado nesta linha para executar o módulo apresentarMenu() da linha 1 à linha 7 (dizemos: fazer a chamada do módulo apresentarMenu()). E, desta vez, quando o módulo terminar de ser executado, o algoritmo voltará para a linha 15!!! Porque desta vez o módulo foi chamado desta linha!

Sensacional!

E, assim, sucessivamente, até o algoritmo terminar.

Uma vez que se entenda como o algoritmo se comporta, é mais fácil conseguir enxergar as vantagens do uso de módulo: desta vez, se o cliente quiser mudar uma frase ou como o texto é apresentado, o programador não precisará alternar quatro locais diferentes, como no nosso algoritmo original. Basta fazer a alteração uma vez, no módulo. Afinal, toda vez que este módulo for chamado, aquela mesma instrução alterada será executada.

Isto também evita um grave problema: no algoritmo original, se o cliente quisesse que uma alteração fosse feita no texto, o programador teria que lembrar que a alteração precisaria ser feita em todas as partes do código que eram repetidas. A chance do programador esquecer de fazer uma alteração é grande. Isso gera o que chamamos de inconsistência, porque o algoritmo se comportará como o cliente quer na maioria das vezes, mas não naqueles locais em que o programador esqueceu de alterar.

Por outro lado, ao se adotar a solução de modularizar o código, basta que a alteração seja feita uma única vez (no módulo) e todas as vezes em que este módulo for chamado, garantidamente o algoritmo executará a alteração realizada.

Escopo e Tempo de Vida das Variáveis em um Módulo

Agora que aprendemos sobre módulos, vamos nos aprofundar um pouco sobre "onde" podemos declarar variáveis. Lembrem-se de que até o momento, declarávamos variáveis sempre logo em seguida ao ***início*** do algoritmo. Mas podemos declará-las em outros locais, como dentro de um módulo ou, até mesmo, antes do módulo e do bloco ***início-fim***!

Porém, há um cuidado que devemos tomar: o local onde a variável é declarada afeta duas de suas propriedades: o escopo (também chamado de visibilidade) e o tempo de vida da variável.

Para poder explicar estas propriedades, primeiro observe o algoritmo a seguir.

```
1.  inteiro idade

2.  lerNome()
3.      cadeia nome
4.      escreva("Digite seu nome")
5.      leia(nome)
6.      escreva("Você se chama ", nome)
7.  fim

8.  início
9.     real salario
10.     escreva("Digite sua idade")
11.     leia(idade)
12.     escreva("Digite seu salário")
13.     leia(salario)
14.     lerNome()
15.     escreva("Você tem ", idade, " anos")
16.     escreva("Seu salário é ", salario")
17. fim
```

Este código funciona! Repare que três variáveis foram declaradas em lugares diferentes.

A variável **salario** da linha 9 foi declarada no local em que estamos acostumados. Mas a variável **idade** da linha 1 foi declarada logo antes do módulo e antes do bloco ***início-fim***. E a variável **nome** da linha 3 foi declarada dentro do módulo lerNome().

Antes de explicar sobre a particularidade dessas variáveis, tente entender o que este algoritmo faz. Lembre-se que o programa começa pelo ***início***.

Pronto? Como o algoritmo começa a ser executado de dentro do bloco ***início-fim***, então:

- O usuário verá uma mensagem para que digite sua idade. O que ele digitar será armazenado na variável idade;
- Em seguida, o usuário verá uma mensagem para que digite seu salário. O que ele digitar será armazenado na variável salario;
- Depois, na linha 14, haverá a chamada do módulo lerNome(). Deste modo, o algoritmo pára de executar nesta linha e começa a executar o módulo na linha 2.
- O usuário verá uma mensagem para digitar o seu nome. O que ele digitar será armazenado na variável nome. E, na linha 6, seu nome será apresentado na tela.
- Assim que o módulo terminar de ser executado, o algoritmo voltará a ser executado da linha em que lerNome() foi chamado, isto é, após a linha 14.
- O algoritmo apresentará a mensagem, mostrando a idade do usuário.
- Por fim, será apresentado o salário do usuário.

Uma pergunta muito comum é: mas por que foi criado um módulo para lerNome() e não foi criado um módulo para ler a idade ou salário? A resposta é simples: para me ajudar a explicar sobre a declaração das variáveis em diferentes lugares. "Quando" devemos criar módulo, veremos mais para a frente.

Agora que entendemos o que o algoritmo faz e porque ele foi criado assim, vamos discutir sobre as particularidades de cada declaração. Para isso, vou apresentar as regras básicas sobre declaração de variáveis:

1. Toda variável criada dentro de um módulo (ou dentro do bloco ***início-fim***) só é visível e só existe dentro do respectivo módulo. Elas são chamadas de **variáveis locais**.

2. Toda variável criada fora de um módulo e fora do bloco ***início-fim*** é visível em todos os módulos e também no bloco ***início-fim***. Elas são chamadas de **variáveis globais**.

Aplicando as duas regras ao algoritmo que criamos, podemos entender que a variável **nome** só poderá ser usada dentro do módulo lerNome(). Ela não poderá ser usada no bloco ***início-fim***, porque ela só é visível dentro do módulo lerNome() e só existirá na memória enquanto este módulo não terminar de ser executado.

A mesma regra se aplica à variável **salario**. Como ela foi criada dentro do bloco ***início-fim***, ela só poderá ser usada dentro deste bloco. E ela não será visível dentro de quaisquer outros módulos que existam. No nosso exemplo, ela não é visível de dentro do módulo lerNome(). E esta variável deixará de existir na memória quando o bloco ***início-fim*** terminar.

Por isso as variáveis **nome** e **salario** são chamadas de locais.

Já, aplicando a regra 2 no nosso exemplo, passamos a entender que a variável **idade**, por ter sido criada fora do módulo e fora do bloco ***início-fim***, é considerada global, isto é, ela pode ser usada em qualquer módulo que o algoritmo venha a ter e dentro do bloco ***início-fim***. E ela só deixará de existir quando o algoritmo terminar.

Rascunho de como Modularizar um Algoritmo

Agora que entendemos que a visibilidade das variáveis depende de onde são declaradas, vamos pensar em um algoritmo um pouco mais completo e como podemos dividi-lo em módulos.

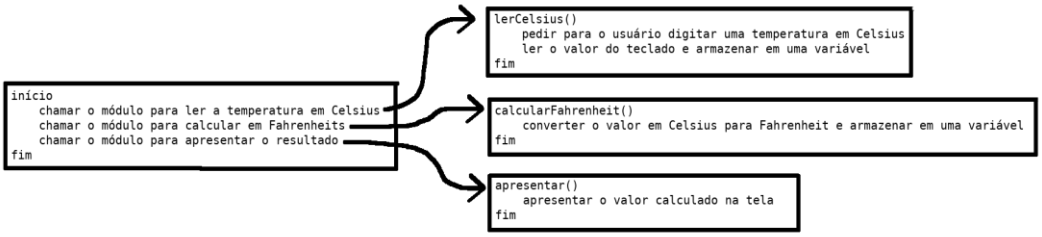
Imagine um algoritmo, como já fizemos, que peça uma temperatura em Celsius, calcule a conversão para Fahrenheit e apresente o resultado para o usuário.

Lendo este problema, podemos facilmente enxergar que o algoritmo precisará realizar três ações bem específicas: a captura dos dados de entrada, o cálculo e a apresentação. Lembre-se que um módulo deve realizar uma única ação específica! Assim, já que é fácil ver que podemos dividir o código em três partes, vamos criar um módulo para cada ação.

- Em um módulo colocaremos o código que pede a temperatura em Célsius. Vamos chamar esse módulo de lerCelsius();
- Em outro módulo, realizaremos o cálculo de conversão para Fahrenheit. Chamemos esse módulo de calcularFahrenheit();
- Em um terceiro módulo, colocaremos o código para apresentar a temperatura. Um bom nome é apresentar();
- No bloco ***início-fim***, apenas chamaremos estes módulos, na ordem correta.

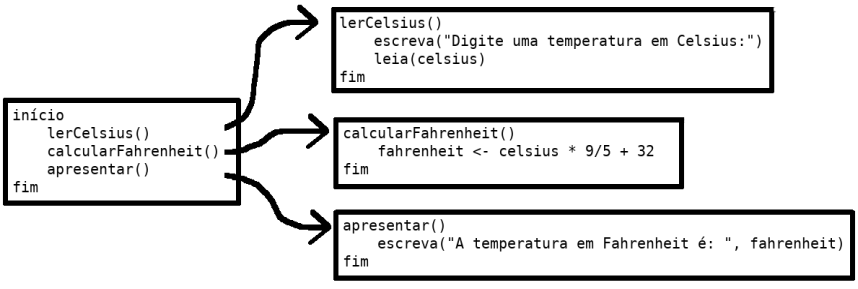
Podemos começar a rascunhar nosso código, como na Figura 15:

Figura 15 - Rascunhando funções



O bacana dessa imagem é que podemos ver como o bloco ***início-fim*** está interagindo com os módulos. Terminado esse primeiro rascunho, vamos substituir as frases em Português pelas nossas instruções. O novo rascunho ficará assim (Figura 16):

Figura 16 - Novo rascunho



Passe um tempo para entender o que está acontecendo neste segundo rascunho. Estamos planejando criar um algoritmo em que cada ação se tornou um

módulo. Essa divisão é necessária na prática? Os módulos calcularFahrenheit() e apresentar(), por exemplo, só possuem uma linha. Preciso criar um módulo para uma ação que só exige uma linha? Depende!!!

Vamos começar pelo módulo calcularFahrenheit(). Ele possui uma expressão aritmética simples. Mas e se precisarmos usar esse cálculo várias vezes em um sistema grande? Há chance de eu escrever essa expressão errada pelo menos uma vez? Se sim, vale a pena criar um módulo. Agora imagine que essa expressão é bem mais complexa. Se tiver que escrevê-la várias vezes em um sistema grande, as chances de errar a escrita pelo menos uma vez aumenta (por exemplo, acidentalmente trocar um sinal de multiplicação por adição).

E o módulo apresentar()? Ele realmente é simples. Desnecessário criar um módulo só para isso na prática. Mas é muito provável que em um sistema grande, você não vá apresentar uma mensagem em um *prompt* de comandos. É bem provável que você fará com que essa mensagem apareça dentro de uma janela de mensagens, com botão de OK, e um tamanho específico, com figurinha. Nesse caso, esse módulo terá mais que uma linha. Então, apesar de, nesse exemplo específico, não ser muito justificável separar uma linha de apresentação em um módulo, é muito interessante exercitarmos dessa forma para que já tenhamos essa noção de separação bem automática em nossa cabeça quando o código se tornar mais complexo!

Moral da história até o momento: por enquanto que o algoritmo é simples, parece desnecessário modularizar o código. Mas esse algoritmo simples é o ponto de partida para algoritmos maiores. Treinar essa divisão com esse exemplo é um bom ponto de partida para adquirirmos a habilidade de saber separar código em módulos.

Onde Declarar as Variáveis?

Vamos, agora, terminar o nosso rascunho. Se repararmos na Figura **Erro! Fonte de referência não encontrada.**, existem variáveis que precisam ser usadas e ainda não as declaramos. Antes de avançarmos, lembrem-se: variáveis locais apenas existem dentro da função em que foram declaradas.

PROBLEMA! Se declararmos a variável `celsius` dentro do módulo lerCelsius(), ela só será visível dentro deste módulo e só existirá dentro deste módulo. Então, quando o módulo terminar de ser executado, a variável não existirá mais na

memória. Isso será um problema quando calcularFahrenheit() for executado, pois a variável *celsius* não existe mais na memória!

Aquela variável *celsius* de calcularFahrenheit() será considerada como outra variável e deverá também ser declarada. Mas aí vem outro problema: se declararmos uma outra variável *celsius* no módulo calcularFahrenheit(), como faremos para o valor que estava na variável do módulo lerCelsius() ir parar na variável celsius de calcularFahrenheit()?

Não é porque as duas variáveis têm o mesmo nome que elas compartilham o mesmo valor. Na memória, são dois espaços diferentes, uma para cada variável!

Temos duas soluções. Uma que é inicialmente mais intuitiva e outra que parece mais burocrática. Vamos começar pela solução mais intuitiva:

Podemos declarar as variáveis como globais! Assim elas serão visíveis em todos os módulos! Deste modo, nosso código completo com variáveis globais ficaria assim:

```
1.   real celsius, fahrenheit

2.   lerCelsius()
3.     escreva("Digite uma temperatura em Celsius:")
4.     leia(celsius)
5.   fim
6.
7.   calcularFahrenheit()
8.     fahrenheit <- celsius * 9 / 5 + 32
9.   fim

10.  apresentar()
11.    escreva("Em Fahrenheit é: ", fahrenheit)
12.  fim

13.  início
14.    lerCelsius()
15.    calcularFahrenheit()
16.    apresentar()
17.  fim
```

O primeiro detalhe a se reparar no código é que as duas variáveis declaradas são globais, pois foram declaradas fora das funções e do bloco ***início-fim***. Deste modo, se um programa fosse feito assim, a execução desse programa entenderia que as variáveis usadas na linha 4, 8 e 11 são aquelas variáveis declaradas como globais.

Então, temos uma primeira solução para o nosso problema! E o pensamento é: como é legal usar variáveis globais! Vou criar todas as minhas variáveis como globais e ser feliz!

Só que não...

Precisamos ter em mente que neste momento, estamos fazendo algoritmos pequenos para aprendermos a lógica básica. Futuramente, quando estes conceitos estiverem bem afixados e começarmos a criar algoritmos maiores, vamos perceber que um sistema nada mais é do que muitos e muitos algoritmos pequenos juntos. Isso porque um sistema completo realiza centenas ou milhares de pequenas tarefas que serão construídas parte a parte.

Então, imagine que você está construindo um sistema comercial completo. Ele terá facilmente centenas de módulos e possivelmente milhares de linhas. Isso também significa que o sistema possuirá centenas de variáveis.

Se todas elas forem criadas como globais, será muito fácil o programador perder o controle sobre elas. Em algum momento ele esquecerá que já criou uma variável para uma determinada finalidade e que deve ser usada em vários pontos do programa, fazendo com que ele crie mais variáveis do que precisaria. Isso irá gerar diversos problemas como começar usando uma variável para armazenar um valor, mas usando outra variável na hora que for precisar desse mesmo valor mais adiante.

Oposto a esse problema, é muito fácil acidentalmente usar uma mesma variável para diferentes propósitos e apagar o valor inicial que deveria ser usado mais adiante.

São deslizes do programador que você pode achar desleixo. Mas isso é extremamente comum. Uma vez que você enxergou este problema, a pergunta é: então, se não devo usar variáveis globais indiscriminadamente, o que devo fazer?

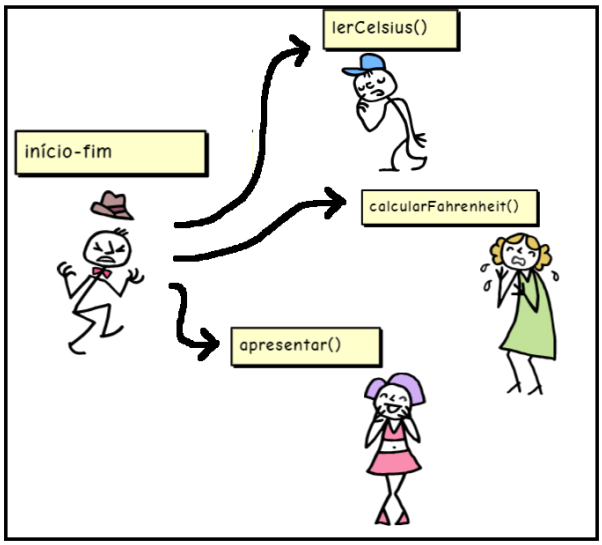
Usar variáveis locais!

Mas aí, como farei para passar um valor de um módulo para o outro, se variáveis locais só existem enquanto o módulo está sendo executado?

É nesse ponto que apresento dois recursos importantes dos módulos!

- Os módulos têm a capacidade de retornar um valor para quem os chamou! Isso se chama retorno;
- Quando um módulo é chamado, é possível enviar valores para ele! Isso se chama passagem de argumentos.

Figura 17 - Personificando as funções



Antes de entrar em detalhes, a ideia abstrata é: imagine que um módulo é um personagem. Então, no lugar da Figura 16, podemos fazer a mesma representação, como na Figura 17.

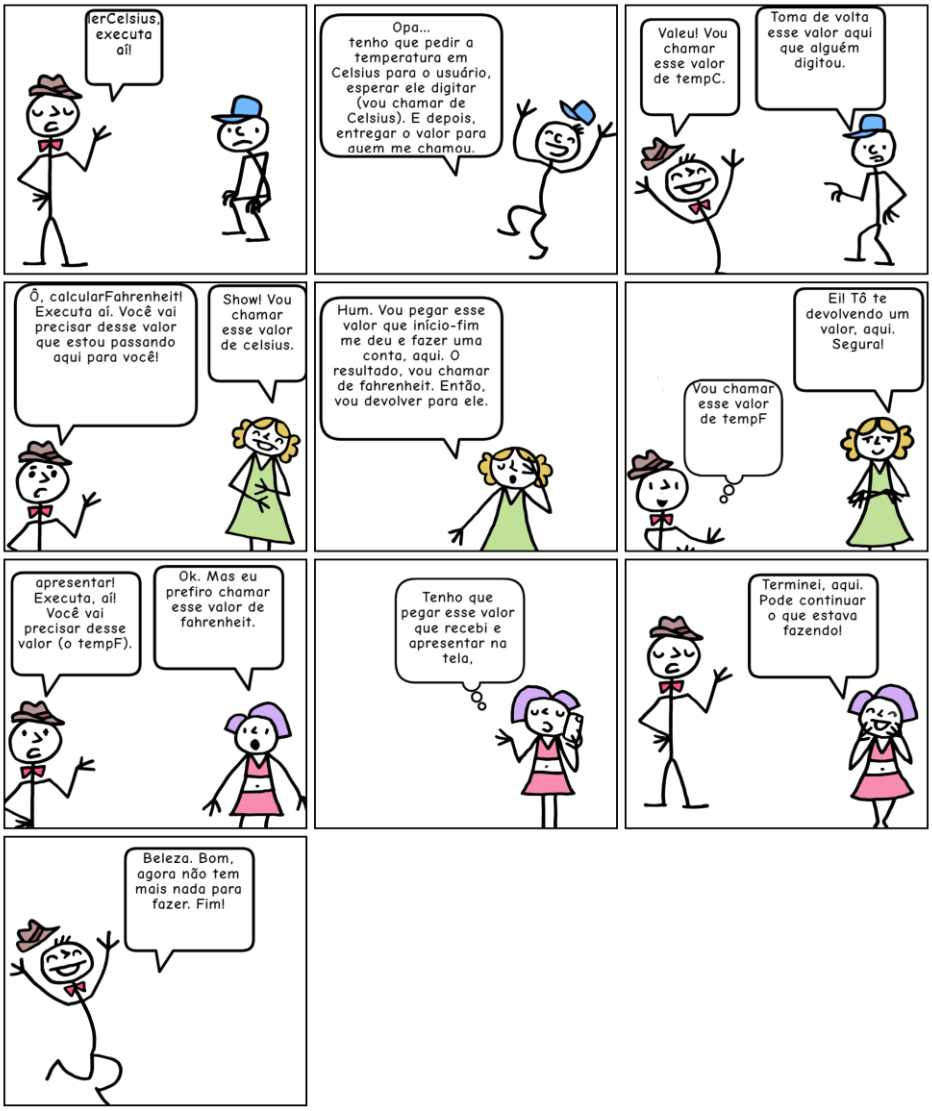
A relação entre os personagens é a mesma. Só que estamos animando a relação. Então, o personagem ***início-fim*** e os personagens que representam os módulos, cada um, tem sua sequência de instruções que deve realizar.

O primeiro personagem irá inicialmente chamar o personagem lerCelsius().

Esse personagem realizará suas tarefas conforme sua sequência de instruções. E quando terminar, o primeiro personagem continuará suas tarefas, chamando o personagem calcularFahrenheit() e assim por diante.

Agora, com esses dois recursos que citei anteriormente, podemos imaginar a seguinte sequência de instruções e interações entre esses personagens (Figura 18):

Figura 18 - Abstraindo a relação entre funções



Vamos discutir um pouco sobre as tiras que acabamos de ver.

Repare que:

- Tudo começa pelo ***início-fim***. E ele inicia chamando o lerCelsius(), como na primeira tira;

- lerCelsius() pede para o usuário digitar algo e usa a variável local `celsius` para armazenar o valor (Tira 2). Essa variável poderia ter o nome que o programador quisesse. Na história, o nome desta variável é `celsius`. Então, lerCelsius() RETORNA o valor para quem o chamou (que foi o bloco ***início-fim***), como na Tira 3;

- Agora, ***início-fim*** continua sua execução. Ele pegará o valor retornado e armazenará em sua variável local chamada `tempC` (que poderia ter qualquer outro nome, até mesmo celsius), como na Tira 4. Ele chama calcularFahrenheit() e, como calcularFahrenheit() precisa de um valor para ser executado (o valor em `tempC`), durante a chamada, ***início-fim*** precisará passar o valor de `tempC` para calcularFahrenheit(), como na Tira 5;

- Então, calcularFahrenheit() começa a execução recebendo o valor que ***início-fim*** passou e armazena em sua variável local chamada `celsius`, também na Tira 5. Em seguida, realiza a conversão e armazena o resultado em uma variável local chamada `fahrenheit` (novamente, a escolha do nome da variável é livre, à escolha do programador), como na Tira 6. Quando terminar a execução, esse módulo retorna o valor que está em fahrenheit (Tira 7);

- ***início-fim*** continua sua execução, primeiramente armazenando o valor retornado por calcularFahrenheit() em uma variável local sua chamada `tempF` (Tira 8). Então, ele chama o módulo apresentar() e passa o valor que acabou de armazenar para ele (Tira 9);

- apresentar() começa sua execução, armazenando o valor recebido em uma variável local sua que o programador resolveu chamar de `fahrenheit` (Tira 9). Em seguida, ele apresenta o valor recebido na tela (Tira 10) e, como não há nada para ser retornado, ele simplesmente "passa a bola", isto é, termina sua execução para que ***início-fim*** volte a ser executado (Tira 11);

- Como ***início-fim*** não tem mais instruções para serem executadas, o algoritmo termina.

É muito importante, neste ponto, você ter em mente a relação entre os módulos. Então, aqui vai um checklist para você pensar a respeito antes de continuar:

- Entenda como é a relação de chamada entre ***início-fim*** e cada módulo. No nosso exemplo, que é bem simples, ***início-fim*** praticamente delega todas as ações para cada módulo, em sequência. Como cada ação será realizada está em cada módulo;

- Entenda, de forma abstrata, como que os dados de um módulo ou ***início-fim*** é enviado:

    - Se um módulo cria/define/calcula um valor que precisa ser usado em outro lugar, então o módulo precisa retornar esse valor;

    - Se um módulo precisa de um valor que foi definido fora dele, então quem o chamar vai precisar enviar um valor para ele quando for chamado (o que chamamos de passagem de argumento);

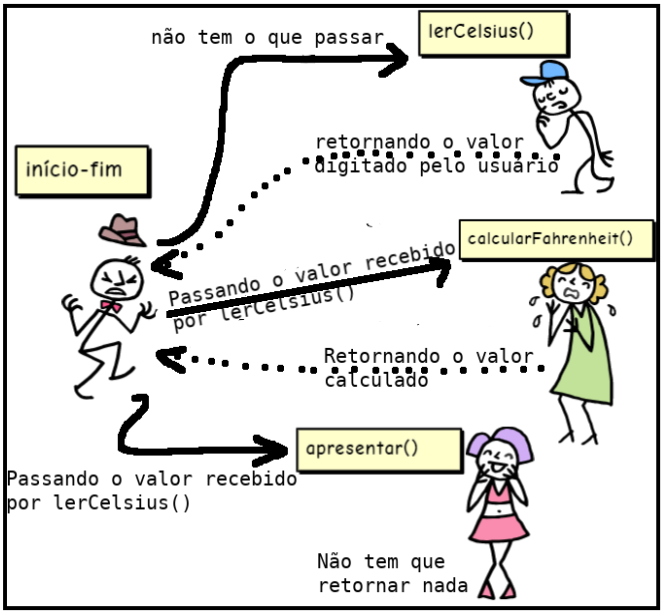
Imaginar cada módulo como um personagem é interessante porque nos facilita a entender que, assim como cada pessoa possui seu próprio cérebro e precisa armazenar algum valor em seu próprio cérebro, os módulos também possuem suas próprias variáveis locais onde os dados que precisam manipular deverão estar armazenados.

Agora, podemos melhorar a figura simplificada, considerando as passagens de argumento e os retornos (Figura 19).

Neste ponto, esta forma de programar parece mais "burocrática" do que usar variáveis globais. Com variáveis globais não preciso me preocupar em retornar valores para quem chamou ou passar algum valor para um módulo. Mas lembre-se do inconveniente de se usar variáveis globais.

E, repare também, que se usarmos variáveis locais, cada módulo passará a ter suas variáveis que são importantes apenas para elas. E não correremos os mesmos riscos do uso de variáveis globais discutidos anteriormente.

Figura 19 - Passagem de argumentos e retornos



Agora, pelo menos três questões devem estar na nossa cabeça:

- Questão 1: Quando sei que um módulo deve retornar um valor?
- Questão 2: Quando sei que um módulo precisa receber um valor?
- Questão 3: Como faço isso no meu algoritmo?

Se você não está se questionando sobre isso, faça uma pausa e volte ao começo da seção para lê-la novamente.

Quando Passar Argumentos e Quando Retornar Valores

Para responder à questão 1, sempre se pergunte: existe algum valor criado/calculado/digitado neste módulo que precisará ser utilizado fora deste módulo? Se a resposta for sim, é porque você deve retornar o valor. Uma coisa importante é: um módulo só deve retornar um valor. Se você sente que precisa retornar mais de um valor, é porque provavelmente precisará dividir este módulo em dois ou mais módulos. Há exceções, mas use essa regra como ponto de partida.

Para responder à questão 2, sempre se pergunte: existe algum valor que precisarei usar dentro deste módulo que foi criado/calculado/digitado fora dele? Se a resposta for sim, é porque esse módulo precisa receber um valor (ou mais de um valor).

Como Retornar Valores em um Algoritmo

Para responder às questões 3 e 1, vamos analisar o módulo lerCelsius():

```
1. lerCelsius()
2.     escreva("Digite uma temperatura em Celsius:")
3.     leia(celsius)
4. fim
```

Primeiramente, lembre-se que o objetivo aqui é utilizar apenas variáveis locais.

Então, repare que a variável *celsius*, para ser local, precisa ser declarada dentro do módulo, como abaixo:

```
1. lerCelsius()
2.     real celsius
3.     escreva("Digite uma temperatura em Celsius:")
4.     leia(celsius)
5. fim
```

Pronto, agora *celsius* é uma variável local.

Então, faremos aquela primeira pergunta: existe algum valor criado/calculado/digitado neste módulo que precisará ser utilizado fora deste módulo?

E a resposta é SIM! Precisaremos do valor digitado pelo usuário (e que estará armazenado na variável *celsius*) mais tarde para converter em Fahrenheit. Perceba que eu nem preciso me importar agora exatamente onde esse valor será usado depois. Preciso apenas saber se vai ou não ser usado.

Como a resposta é sim, então temos que retornar. No nosso código, sinalizaremos o retorno como abaixo:

```
1. real lerCelsius()
2.     real celsius
3.     escreva("Digite uma temperatura em Celsius:")
4.     leia(celsius)
5.     retornar celsius
6. fim
```

Nós fizemos duas alterações em relação ao código anterior. Primeiro, sinalizamos o retorno com a instrução **retornar** e o identificador da variável. Em linguagens como C e Java, a instrução é justamente **return** seguido do identificador da variável. Lembre-se de um detalhe: a função termina depois de retornar. Deste modo, não podemos colocar nenhuma instrução em sequência de retornar, porque ela terminará ao retornar.

Mas se prestar atenção, verá que fizemos uma outra alteração ainda na linha 1!

Uma vez que identifiquemos o valor que precisa ser retornado, precisamos também identificar qual o tipo do valor deverá ser retornado. Como queremos retornar um valor real, então, precisamos, antes do nome do módulo, colocar o tipo do valor que este módulo irá retornar.

Então, é simples! Precisamos indicar o que será retornado (linha 5) e o tipo que será retornado deverá ser indicado antes do nome do módulo (linha 1).

Há apenas um detalhe que não podemos esquecer: sempre que um módulo retornar um valor, esse valor será retornado para quem o chamou. E quem o chamou precisará armazenar o valor retornado, como nos Quadros 4 e 8.

E como se faz isso? Atribuindo o retorno do módulo a uma variável local de quem o chamou (no caso, o bloco início-fim). Considerando que o código original era:

```
1. início
2.     lerCelsius()
3.     calcularFahrenheit()
4.     apresentar()
5. fim
```

Devemos perceber que a chamada do módulo lerCelsius() deverá retornar um valor para a linha 2. E como já sabemos que esse valor retornado deverá ser armazenado em uma variável local do bloco ***início-fim***, então precisaremos criar uma variável local para receber o retorno e usá-la para tal fim, como no código abaixo:

```
1. início
2.     real tempC
3.     tempC <- lerCelsius()
4.     calcularFahrenheit()
5.     apresentar()
6. fim
```

Deste modo, quando a linha 3 for executada, ***início-fim*** irá parar de ser executado, esperará lerCelsius() ser executado e quando lerCelsius() terminar sua execução e retornar um valor, este valor retornado será armazenado na variável tempC de ***início-fim***.

Como Passar Argumentos em um Algoritmo

Para responder às questões 3 e 2, vamos analisar o módulo apresentar():

```
1. apresentar()
2.     escreva("Em Fahrenheit é: ", fahrenheit)
3. fim
```

Repare que a variável *fahrenheit* está sendo usada dentro de apresentar() e, por ser uma variável local, ela precisa ser declarada dentro deste módulo. Mas não vamos nos precipitar.

Antes de mais nada, faça a pergunta 2: existe algum valor que precisarei usar dentro deste módulo que foi criado/calculado/digitado fora dele?

E a resposta é sim!

Quando identificamos que o valor de uma variável que vamos usar vem de fora do módulo, declaramos esta variável em um lugar especial, dentro dos parênteses do módulo. E essa variável especial será chamada de parâmetro.

Adivinha qual o objetivo do parâmetro? Ele recebe o valor que será passado por quem chamar esse módulo!

Alterando o módulo original, o código fica assim:

```
1. nada apresentar(real fahrenheit)
2.     escreva("Em Fahrenheit é: ", fahrenheit)
3. fim
```

Deste modo, a variável local *fahrenheit* foi criada dentro do parênteses do módulo. Isso confere a ela o *status* de parâmetro. Então, quando essa função for chamada, o valor passado para esse módulo será automaticamente armazenado na variável *fahrenheit*!

Se fizermos aquela outra pergunta (existe algum valor criado/calculado/digitado neste módulo que precisará ser utilizado fora deste módulo?) veremos que a resposta é não! Este é um módulo que não precisa (e não deve) retornar valores. Quando isso acontece, colocaremos a palavra **nada** antes do nome do módulo para sinalizar que ele não retorna nada.

A dúvida agora é: como passo o valor para esse módulo? É só passar o valor desejado entre parênteses na hora da chamada, como abaixo:

```
1. início
2.     ...
3.     ...
4.     ...
5.     apresentar(tempF)
6. fim
```

Na linha 5, o valor que estiver na variável *tempF* será enviado para o módulo apresentar() e automaticamente o parâmetro *fahrenheit* do módulo apresentar() armazenará o valor passado.

Agora vamos analisar o módulo calcularFahrenheit(). Tente identificar se este módulo precisa retornar um valor e se ele precisará de um valor vindo de fora. O código original é:

```
1. calcularFahrenheit()
2.     fahrenheit <- celsius * 9 / 5 + 32
3. fim
```

Existe algum valor criado/calculado/digitado neste módulo que precisará ser utilizado fora deste módulo? Sim! *fahrenheit* será usado mais tarde para ser apresentado na tela! Portanto, precisamos retornar *fahrenheit*.

Existe algum valor que precisarei usar dentro deste módulo que foi criado/calculado/digitado fora dele? Sim! O valor de *celsius* foi definido em outro lugar, quando o usuário digitou um valor! Então, precisamos tornar essa variável um parâmetro.

Com estas observações, o novo código fica como abaixo:

```
1. real calcularFahrenheit(real celsius)
2.     real fahrenheit
3.     fahrenheit ← celsius * 9 / 5 + 32
4.     retornar fahrenheit
5. fim
```

Em um primeiro momento, em resposta à questão 1, adicionamos a instrução de retornar na linha 4, adicionamos a indicação do tipo que será retornado pelo módulo na linha 1 (o real antes do nome) e, como *fahrenheit* é uma variável local, nós a declaramos na linha 2.

Em um segundo momento, em resposta à questão 2, declaramos a variável *celsius* dentro do parênteses do módulo, conferindo a ela o status de parâmetro.

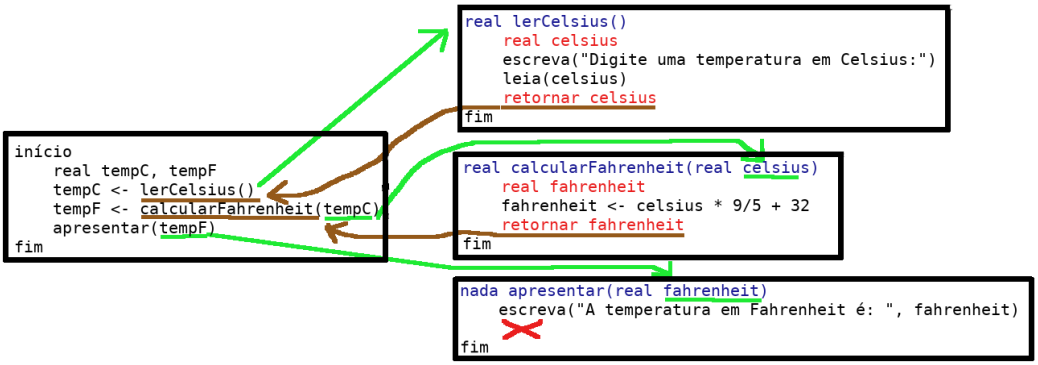
Por fim, precisamos adaptar o bloco ***início-fim*** para que realize as chamadas, passando os valores necessários para cada módulo e recebendo o retorno daqueles módulos que devem retornar.

```
1. início
2.     real tempC, tempF
3.     tempC <- lerCelsius()
4.     tempF <- calcularFahrenheit(tempC)
5.     apresentar(tempF)
6. fim
```

As linhas 3 e 5 já foram discutidas. Faltou apenas a linha 4 para adaptarmos. Como o módulo calcularFahrenheit() precisa do valor da temperatura em Celsius armazenado na variável `tempC`, então nós a passamos para este módulo. E como este módulo retorna um valor, que representa a conversão da temperatura em Fahrenheit, criamos uma variável `tempF` para receber o valor retornado.

Para visualizarmos a dinâmica entre os módulos, observe a Figura 20:

Figura 20 - Dinâmica entre módulos



- Tudo se inicia do ***início-fim***;
- ***início-fim*** chama lerCelcius(), que pede um valor para o usuário e retorna o valor;
- O valor retornado será atribuído à variável `tempC` de ***início-fim***;
- Então, ***início-fim*** chama calcularFahrenheit() e passa o valor que acabou de receber;
- calcularFahrenheit() recebe o valor e atribui à sua variável local `celsius` (que é um parâmetro);
- calcularFahrenheit() faz o cálculo e o devolve para quem o chamou;

- ***início-fim*** recebe o valor retornado e atribui à variável `tempF`;
- ***início-fim*** chama apresentar() e passa o valor calculado para esse módulo;
- apresentar() recebe o valor passado e o armazena em sua variável local `fahrenheit`;
- apresentar() mostra uma mensagem na tela e termina. Não há um valor a retornar;
- ***início-fim*** não recebe um valor de apresentar();
- Como não há mais nada, o algoritmo termina.

Procedimentos e Funções

Agora que entendemos o conceito de módulo, podemos ir para o próximo passo: um módulo pode ser um procedimento ou uma função. A diferença entre ambos é que procedimentos não retornam valores e funções retornam valores.

Assim, no exemplo usado, apresentar() é chamado de procedimento; já, lerCelsius() e calcularFahrenheit() são funções.

Em C, é comum chamarmos qualquer módulo de função, mesmo que não retorne valores. **Por isso, a partir deste ponto, usaremos a palavra função como sinônimo de módulo, apesar de sabermos que há uma distinção conceitual entre funções e procedimentos.**

Passagem de Mais de um Valor

E se quisermos passar mais de um valor para uma função?

Imagine, por exemplo, que a função apresentar() deva mostrar na tela os valores em Celsius e Fahrenheit. Então, um primeiro rascunho seria:

```
1. nada apresentar(???)
2.     escreva("Em Fahrenheit é: ", fahrenheit)
3.     escreva("Em Celsius é: ", celsius)
4. fim
```

Se respondermos à pergunta 2 (existe algum valor que precisarei usar dentro deste módulo que foi criado/calculado/digitado fora dele?) perceberemos que agora há dois valores sendo usados e que foram criados fora desta função (a temperatura em Celsius foi digitada pelo usuário em outro momento e a temperatura em Fahrenheit foi calculada em outro momento).

Então, é simples. Precisamos criar dois argumentos! Um para Fahrenheit e um para Celsius, como no código abaixo:

```
1. nada apresentar(real celsius, real fahrenheit)
2.     escreva("Em Fahrenheit é: ", fahrenheit)
3.     escreva("Em Celsius é: ", celsius)
4. fim
```

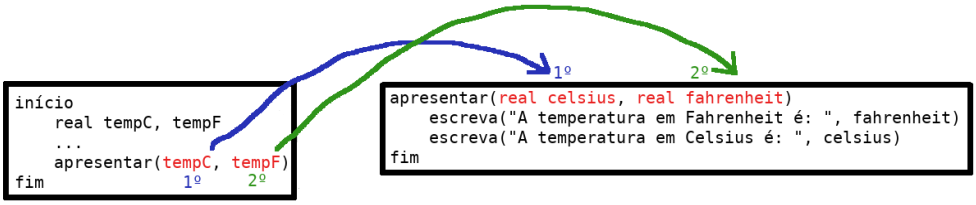
É importante notar que os parâmetros são separados por vírgula e que os tipos devem ser repetidos. Este é um padrão adotado pelas linguagens C e Java, e por isso iremos adotá-lo aqui.

Por fim, precisamos descobrir como os valores serão passados na chamada da função:

```
1. início
2.     real tempC, tempF
3.     tempC <- lerCelsius()
4.     tempF <- calcularFahrenheit(tempC)
5.     apresentar(tempC, tempF)
6. fim
```

Os valores passados na linha 5 do código anterior são atribuídos automaticamente para os argumentos da função apresentar() pela ordem em que são digitadas. Nesse exemplo, na linha 5 é passado primeiro o valor da variável *tempC* e depois o valor da variável *tempF*. Então, a variável *celsius* da função apresentar() vai receber o valor de *tempC* e a variável *fahrenheit* vai receber o valor de *tempF* (Figura 21).

Figura 21 - Passagem de mais de um parâmetro

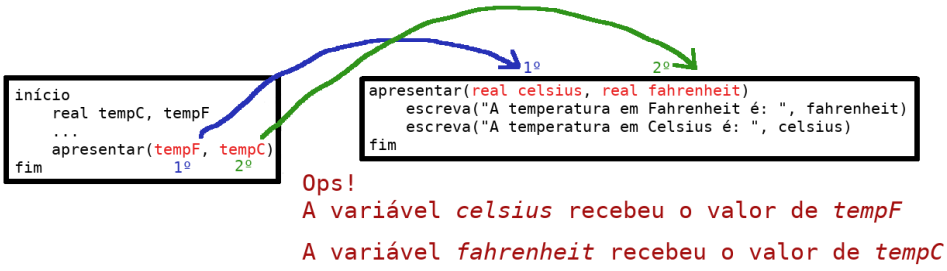


Esse é um cuidado que precisamos ter: para "o computador", o nome de uma variável não significa nada. Então, se por acidente invertermos a ordem das variáveis ao passá-las, como abaixo:

```
1. início
2.     real tempC, tempF
3.     tempC <- lerCelsius()
4.     tempF <- calcularFahrenheit(tempC)
5.     apresentar(tempF, tempC)
6. fim
```

Quem receberá o valor da variável *tempF*, na função apresentar será a variável *celsius*. O mesmo ocorrerá com a variável *tempC* e *fahrenheit*. Aí, o que será apresentado na tela será trocado (Figura 22).

Figura 22 - Cuidado com a ordem dos argumentos



Por fim:

- E se quisermos passar três valores? A ideia é a mesma: crie três parâmetros.
- E se quisermos passar valores de tipos diferentes? Sem problema, crie parâmetros com os respectivos tipos.

Então, se quisermos passar 3 valores, sendo dois inteiros e um real, faremos como a seguir:

```
1. início
2.     inteiro idade, nascimento
3.     real salario
4.     ...
5.     ...
6.     apresentar(idade, nascimento, salario)
7. fim
```

O importante é que, no momento de se criar a função apresentar(), os parâmetros sejam planejados na ordem em que os valores da linha 6 foram passados:

```
1. nada apresentar(inteiro idade, inteiro nasc, real salario)
2.     escreva("A idade é: ", idade)
3.     escreva("O ano de nascimento é: ", nasc)
4.     escreva("O salário é: ", salario)
5. fim
```

A variável *nasc* (que é um parâmetro) da linha 1 da função apresentar() foi propositalmente chamada de *nasc* para lembrarmos que o identificador dos parâmetros não precisam ser iguais aos das variáveis que estão sendo passadas na chamada da função. E as variáveis *idade* e *salario* foram criadas com o mesmo identificador para lembrarmos que elas podem.

E por quê? Porque os parâmetros são variáveis locais da função apresentar() e as variáveis passadas (*idade*, *nascimento* e *salário*, do bloco ***início-fim***) também são variáveis locais. E uma variável local de uma função só existe naquela função e pode ter o mesmo nome (identificador) de outra variável de outra função.

Exemplo de Reusabilidade

O exemplo anterior é bastante simples, mas muito poderoso porque cada função tem características diferentes: a função de leitura tinha retorno, mas não tinha

parâmetro; a função de apresentação tinha parâmetro mas não tinha retorno; e, a função de cálculo tinha parâmetro e retorno.

Uma vez que tenhamos entendido o básico, vamos ver um outro exemplo de algoritmo que pode se aproveitar do reúso. Aqui, veremos uma mesma função ser chamada mais de uma vez no algoritmo (portanto, sendo reusada). O enunciado é:

Faça um algoritmo que peça 2 números, calcule a soma destes números e apresente a soma na tela.

Se formos dividir o problema em partes, podemos de início identificar as seguintes tarefas:

- Pedir dois números;
- Calcular a soma;
- Apresentar a soma na tela.

Ao rascunhar o algoritmo, podemos pensar em criar uma função para cada uma dessas ações. Vamos focar na primeira função de ler os dois valores do usuário.

```
1. lerNumeros()
2.     real num1, num2
3.     escreva("Digite um valor:")
4.     leia(num1)
5.     escreva("Digite um valor:")
6.     leia(num2)
7. fim
8.
```

Se fizermos a pergunta: Existe algum valor criado/calculado/digitado neste módulo que precisará ser utilizado fora deste módulo? A resposta será sim. O problema é que são dois valores que precisarão ser usados futuramente.

É nesse ponto que percebemos que essa função está realizando tarefas demais. Uma função deve retornar apenas um valor! Aqui, podemos, então, pensar em dividir esta função em duas: uma para cada número a ser digitado:

```
1. real lerNum1()
2.     real num1
3.     escreva("Digite:")
4.     leia(num1)
5.     retornar num1
6. fim
```

```
7. real lerNum2()
8.     real num2
9.     escreva("Digite:")
10.    leia(num2)
11.    retornar num2
12. fim
```

```
1. início
2.     real n1, n2
3.     n1 <- lerNum1()
4.     n2 <- lerNum2()
5.     ...
6.     ...
7. fim
```

Ok, esta é uma solução. Mas olhe atentamente as duas funções. Reparou que elas são iguais? Se elas são iguais, eu posso usar uma função só, mas duas vezes!

```
1. real lerNum()
2.     real num
3.     escreva("Digite:")
4.     leia(num)
5.     retornar num
6. fim
```

```
7. início
8.     real n1, n2
9.     n1 <- lerNum()
10.    n2 <- lerNum()
11.    ...
12.    ...
13. fim
```

Aí está a mágica do reúso! Uma mesma função pode ser chamada mais de uma vez!

Vamos entender a execução:

- Quando o algoritmo for executado, começará pelo bloco ***início-fim***;
- Neste bloco, na linha 9, a função lerNum() será chamada;
- A função lerNum() será executada. Pedirá um número para o usuário e retornará este número para quem o chamou;
- Então, o valor retornado, desta vez, será enviado para a linha 9! Porque foi lá que a função lerNum() foi chamada. O valor retornado, então, será atribuído para a variável *n1*;
- Em seguida, a linha 10 será executada. Novamente a função lerNum() será chamada;
- A função será executada, pedirá um número para o usuário e o valor digitado será retornado para quem chamou a função;

- Desta vez, o valor retornado será enviado para a linha 10! Porque foi lá que a função lerNum() foi chamada da última vez! Então, o valor que o usuário digitou agora será atribuído para a variável *n2*;
- O bloco ***início-fim*** continuará sua execução nas próximas linhas, com os dois valores digitados. Um na variável *n1* e outro na variável *n2*.

Uma mesma função foi usada duas vezes.

É interessante notar que, às vezes, a gente pode não perceber este cenário em que é possível reusar uma única função em mais de um ponto do algoritmo. Para enxergarmos o reúso, devemos pensar: há duas funções que realizam uma mesma ação? Então, só preciso de uma!

Uma Mesma Função, Pequena Variação

E se quisermos que a mensagem apresentada na tela pela função lerNum() seja diferente, dependendo da ocasião? Por exemplo, em vez do algoritmo apresentar a mensagem: "Digite um valor" nas duas chamadas, não seria possível fazê-la apresentar mensagens diferentes? Como "Digite um número" e "Digite outro número"?

Sim! Quando há pequenas variações que gostaríamos de inserir em uma função, podemos ainda assim usar uma função, apenas.

No exemplo a seguir, veremos como criar uma função que varie as mensagens a serem apresentadas. Primeiro, vamos lembrar que uma mensagem de texto é armazenada em uma variável do tipo cadeia, como a seguir:

```
cadeia msg
msg = "Eu sou uma mensagem"
escreva(msg)
```

Temos uma variável do tipo cadeia. O texto a ser atribuído deve estar entre aspas. Se quisermos apresentar a mensagem na tela, basta usar a instrução com a variável entre parênteses.

Agora, vamos imaginar que, na hora de chamar uma função, passamos para ela um texto entre aspas. Se estamos passando um valor para a função (o texto), então esta função precisa ser criada com um parâmetro. Como o que foi enviado é um texto, então o parâmetro será um texto (o tipo de variável que armazena texto é cadeia).

Por fim, da primeira vez que a função for chamada, passaremos uma mensagem. E da segunda vez que for chamada, passaremos uma segunda mensagem, como abaixo:

```
1. real lerNum(cadeia msg)
2.     real num
3.     escreva(msg)
4.     leia(num)
5.     retornar num
6. fim
```

```
7. início
8.     real n1, n2
9.     n1<-lerNum("Digite um número:")
10.    n2<-lerNum("Digite outro número:")
11.    ...
12.fim
```

Observe que, na linha 9, foi passado o texto "Digite um número". Este texto é o valor que será armazenado na variável *msg* (que é o parâmetro da função lerNum()).

Quando a função for executada da primeira vez, o valor de *msg* será "Digite um número:". Na linha 3, quando *msg* for apresentado na tela, mostrará "Digite um número:".

Mas, depois que a execução de lerNum() terminar e voltar para o bloco ***início-fim*** e a linha 10 for executada, a função lerNum() será chamada novamente, mas com um texto diferente!

Então, o parâmetro *msg* de lerNum() desta vez receberá o valor "Digite outro número:" e é essa a mensagem que ele apresentará na tela, desta segunda vez que a linha 3 for executada!

Diferentes Retornos em uma Mesma Função

É muito comum criar funções que retornem valores diferentes (mas de mesmo tipo!), dependendo da situação. E isso é facilmente solucionado colocando-se uma condição dentro da função.

Por exemplo, imagine que você queira criar uma função que "decida" se um valor é par ou ímpar. Então, podemos criar essa função de forma que ela receba um número inteiro, verifique se é par ou ímpar e retorne a respectiva mensagem correta. Uma forma de implementar essa função segue abaixo:

```
1. cadeia decidirParOuImpar(inteiro n)
2.     se(n%2 == 0) então
3.         retornar "par"
4.     senão
5.         retornar "ímpar"
6.     fimse
7. fim
```

```
8.   início
9.       cadeia resposta
10.      inteiro num
11.      escreva("Digite um número")
12.      leia(num)
13.      resposta<-decidirParOuImpar(num)
14.      escreva(resposta)
15.  fim
```

Vamos às considerações:

- Primeiro: o bloco ***início-fim*** poderia ser simplificado se criássemos uma função para ler um número e outra para apresentar a mensagem, como nos exemplos anteriores. Sugiro que faça isso como treino;
- Quando o bloco ***início-fim*** iniciar a execução, o usuário será orientado a digitar um número e este número será atribuído à variável *num*;
- Em seguida, a função decidirParOuImpar() será chamada e o valor digitado será passado para ela;
- O valor passado será atribuído à variável *n* desta função;
- Dentro da função, haverá uma decisão: se o número for par (tecnicamente, se o resto da divisão de 0 por 2 for 0) então a função retornará a mensagem "par". Do contrário, a função retornará a mensagem "ímpar";
- O código voltará a ser executado do ponto onde essa função foi chamada, isto é, na linha 13. A mensagem retornada será atribuída à variável *resposta*;
- E o valor da resposta será apresentado na tela.

É interessante notas que podemos colocar as instruções que quisermos dentro de uma função. Ela pode ter laços de repetição, SE-SENÃO, escreva, leia, ou quaisquer outras instruções.

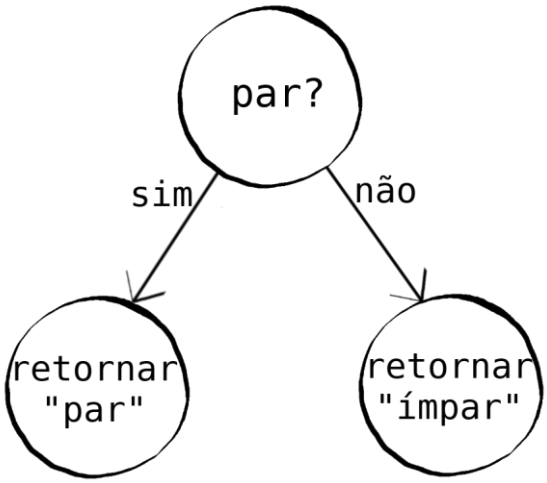
Mas temos que ter dois cuidados:

- Lembre-se de que a função só deve retornar um único valor. Apesar de a função anterior ter duas instruções retornar, apenas uma delas será executada quando a função for chamada. Ou será retornado "par" ou será

retornado “ímpar”, então a regra está sendo cumprida (Figura **Erro! Fonte de referência não encontrada.**);

- A instrução retornar deve ser a última a ser executada e não deverá haver instruções em sequência a ela. E essa regra também está sendo respeitada. Porque, uma vez que a instrução ***se-senão*** decida o caminho a ser tomado, não haverá instruções em sequência a retornar. A Figura 23 ilustra a explicação.

Figura 23 - Retorno dentro de blocos se-senão



Na figura, é possível perceber que, se o número for par, a última instrução será o retorno da mensagem “par”. E se o número for ímpar, a última instrução será o retorno da mensagem “ímpar”. Podem haver instruções em sequência ANTES de retornar, mas não depois.

Mas reparou que eu disse “em sequência”? Isso porque até pode haver mais instruções depois de retornar, contanto que haja um caminho para essas instruções sem que passe por este retornar.

Imagine que você queira criar uma função que retorna a divisão de um número por outro. E, como você sabe que não se pode dividir por 0, você decidiu que em caso de divisão por 0 a sua função deverá retornar 0. Do contrário, a função

deverá retornar o resultado da divisão. Uma possível solução de algoritmo segue abaixo:

```
1.  real dividir(real n1, real n2)
2.      real res
3.      se(n2 == 0) então
4.          retornar 0
5.      fimse
6.      res <- n1/n2
7.      retornar res
8.  fim

9.  início
10.     real n1, n2, res
11.     escreva("Digite um
12.              número")
13.     leia(num1)
14.     escreva("Digite outro
15.              número")
16.     leia(num2)
17.     res←dividir(num1, num2)
18.     escreva(res)
19. fim
```

No exemplo acima, se a condição for verdadeira, na linha 3, o algoritmo retornará 0 e terminará a função neste ponto. Mas, se a condição for falsa, o algoritmo continuará para a linha 6 e retornará um outro valor na linha 7.

Esta função está correta, porque apesar de haver instruções depois do retornar da linha 4, há um possível caminho para se chegar na linha 6 sem que se passe pelo retornar da linha 4: caso a condição da linha 3 seja falsa.

É importante entender que: uma vez que a instrução retornar seja executada, a função termina neste ponto; e que se a condição for falsa, a execução não passará por este retornar, e continuará até a próxima instrução retornar.

Isto nos leva a uma última regra importante: se a função foi criada de forma que ela deva retornar um valor, então ela SEMPRE precisa retornar um valor!!!

O código a seguir está errado!

```
1.  real dividir(real n1, real n2)
2.      real res
3.      se(n2 != 0) então
4.          res < - n1/n2
5.          retornar res
6.      fimse
7.      escreva(“Não pode haver divisão por zero!”)
8.  fim

9.  início
10.     real n1, n2, res
11.     escreva(“Digite um número”)
12.     leia(num1)
13.     escreva(“Digite outro número”)
14.     leia(num2)
15.     res<-dividir(num1, num2)
16.     escreva(res)
17. fim
```

No exemplo acima, o código está errado porque ela apenas retorna um valor se a condição da linha 3 for verdadeira. E não terá retorno se a condição for falsa. Então, ora esta função retorna valor e ora não retorna. Isto é um erro porque, lembre-se, uma função que foi definida para retornar um valor, sempre terá que retornar um valor.

Apesar de fazer muito sentido na nossa cabeça querer criar uma função que só retorne um valor se a regra for permitida, isso não faz sentido sob o ponto de vista de algoritmos. Pense: se a função retorna um valor, então quem chamou essa função está esperando um retorno (linha 15, no exemplo). Se o algoritmo não retornar um valor, o que a variável da linha 15 receberá? Percebe agora porquê precisa haver retorno?

Este é um momento em que precisamos parar e fazer exercícios para fixar.

Exercícios Básicos Sobre Funções

DICA: antes de começar o algoritmo, divida o problema em partes. Um bom começo é dividir o código em função de entrada, função de cálculo e função de apresentação.

PARTE 1: o clássico (entrada, cálculo, saída)

1. Crie um algoritmo que peça uma temperatura em Celsius, faça a conversão para Farenheit e apresente o valor convertido. F= (9*C/5) + 32

2. Crie um algoritmo que peça uma temperatura em Celsius, faça a conversão para Farenheit e apresente o valor em Celsius e em Farenheit. F= (9*C/5) + 32. Dica: se esqueceu, reveja a seção Passagem com Mais de Um Valor.

3. Crie um algoritmo que peça uma temperatura em Farenheit, faça a conversão para Celsius e apresente o valor convertido. C= 5*(F-32)/9

4. Crie um algoritmo que peça uma temperatura em Farenheit, faça a conversão para Celsius e apresente o valor em Farenheit e em Celsius. C= 5*(F-32)/9

5. Crie um algoritmo que peça duas notas, calcule a média e apresente a média

6. Crie um algoritmo que peça duas notas, calcule a média e informe se o aluno está aprovado (se sua nota for maior ou igual a 6) ou se está de recuperação (se sua nota for menor que 6).

PARTE 2: repare que nem sempre é preciso criar uma função para entrada, uma para cálculo e uma para saída. Nos exercícios abaixo, crie uma função para pedir o número, e uma outra função que vai decidir se é par ou ímpar e já apresentar o resultado.

7. Crie um algoritmo que peça um número inteiro e diga se é par ou ímpar.

8. Crie um algoritmo que receba dois números e apresente o maior. Dica: lembre-se da seção EXEMPLO DE REUSABILIDADE.

PARTE 3: mas dá! Vamos repetir os exercícios 7 e 8, mas desta vez, crie uma função para receber os dados de entrada, uma outra função para decidir se é par ou ímpar (no caso do exercício 7) ou qual o maior (no caso do exercício 8) e uma função para apresentar o resultado.

Como é em algumas linguagens

Português Estruturado

```
real lerCelsius()
    real celsius
    escreva(“Digite uma temperatura em Celsius:”)
    leia(celsius)
    retornar celsius
fim

real calcularFahrenheit(real celsius)
    real fahrenheit
    fahrenheit <- celsius * 9 / 5 + 32
    retornar fahrenheit
fim

nada apresentar(float celsius, float fahrenheit)
    escreva("Em Celsius: ", celsius)
    escreva("Em Fahrenheit: ", fahrenheit)
fim

início
    real tempC, tempF
    tempC <- lerCelsius()
    tempF <- calcularFahrenheit(tempC)
    apresentar(tempC, tempF)
fim
```

C

```c
#include <stdio.h>
#include <string.h>

float lerCelsius(){
    float celsius;
    printf("\nDigite uma temperatura em Celsius: ");
    scanf("%f",&celsius);
    return celsius;
}

float calcularFahrenheit(float celsius){
    float fahrenheit;
    fahrenheit = celsius * 9/ 5 + 32;
    return fahrenheit;
}

void apresentar(float celsius, float fahrenheit){
    printf("\nEm Celsius: %.2f", celsius);
    printf("\nEm Fahrenheit: %.2f", fahrenheit);
}

int main(){
    float tempC, tempF;
    tempC = lerCelsius();
    tempF = calcularFahrenheit(tempC);
    apresentar(tempC, tempF);
    return 0;
}
```

Obs: Em C/C++, da forma como foi mostrado, precisamos escrever a função antes dela ser chamada de dentro da main(). Do contrário, o compilador não conseguirá encontrar a função chamada e apresentará a mensagem de que ela não existe (apesar de existir). Deste modo, por exemplo, se escrevêssemos a função apresentar() depois da função main(), o código não compilaria.

Em programas pequenos, não é um problema criar uma função antes de chamá-la. Mas quando um programa passa a ter várias dezenas ou centenas de funções, pode ser inconveniente se preocupar se uma função está sendo chamada antes ou depois de ter sido escrita.

Se não quisermos nos preocupar com a ordem das funções, podemos "dizer ao compilador" que as funções existem, por meio de prototipação. Entenda prototipação como uma espécie de declaração de que uma função existe em algum lugar do código.

A prototipação de uma função e constituída do **tipo de retorno** + **nome da função** + **tipos de cada parâmetro**. Ela deve ser feita antes das funções serem escritas.

Para a função lerCelsius(), a linha de prototipação é:

```c
float lerCelsius(void);
```

ou

```c
float lerCelsius();
```

Quando a função não possui parâmetro, não é preciso indicar com a palavra void entre parênteses (mas pode).

Para a função calcularFahrenheit(), a prototipação é:

```
float calcularFahrenheit(float);
```

ou

```
float calcularFahrenheit(float celcius);
```

Colocar o nome do parâmetro é opcional e o compilador não verificará se o nome (identificador) é o mesmo usado na escrita da função. Portanto, é desnecessário.

E a prototipação da função apresentar() é:

```
void apresentar(float, float);
```

ou

```
void apresentar(float celsius, float fahrenheit);
```

Com estas linhas, no nosso exemplo, podemos criar as funções depois da main(). Abaixo, um código completo:

```
#include <stdio.h>
#include <string.h>

//Prototipação
float lerCelsius();
float calcularFahrenheit(float);
void apresentar(float, float);

int main(){
    float tempC, tempF;
    tempC = lerCelsius();
    tempF = calcularFahrenheit(tempC);
    apresentar(tempC, tempF);
    return 0;
}

//As funções podem ser escritas depois de serem chamadas
float lerCelsius(){
    float celsius;
    printf("\nDigite uma temperatura em Celsius: ");
    scanf("%f",&celsius);
    return celsius;
}

float calcularFahrenheit(float celsius){
    float fahrenheit;
    fahrenheit = celsius * 9/ 5 + 32;
    return fahrenheit;
}

void apresentar(float celsius, float fahrenheit){
    printf("\nEm Celsius: %.2f", celsius);
    printf("\nEm Fahrenheit: %.2f", fahrenheit);
}
```

Por fim, caso se deseje criar variáveis globais, basta declará-las fora das funções. C/C++ permite que variáveis globais sejam criadas em qualquer lugar fora das funções, mesmo depois de serem usadas. Mas deve-se criá-las logo no começo do programa para agrupá-las de forma a facilitar a manutenção do código.

C++

Pode ser o mesmo código do C, ou:

```
#include <iostream>

using namespace std;

float lerCelsius(){
    float celsius;
    cout << "\nDigite uma temperatura em Celsius: ";
    cin >> celsius;
    return celsius;
}

float calcularFahrenheit(float celsius){
    float fahrenheit;
    fahrenheit = celsius * 9/ 5 + 32;
    return fahrenheit;
}

void apresentar(float celsius, float fahrenheit){
    cout << "\nEm Celsius: " << celsius;
    cout << "\nEm Fahrenheit: " << fahrenheit;
}

int main(){
    float tempC, tempF;
    tempC = lerCelsius();
    tempF = calcularFahrenheit(tempC);
    apresentar(tempC, tempF);
    return 0;
}
```

Obs: as mesmas observações feitas em C se aplicam ao C++.

Go

```
package main

import (
    "fmt"
)

func lerCelsius() float32 {
    var celsius float32
    fmt.Print("Digite uma temperatura em Celsius: ")
    fmt.Scanf("%f", &celsius)
    return celsius
}

func calcularFahrenheit(celsius float32) float32 {
    var fahrenheit float32
    fahrenheit = celsius*9/5 + 32
    return fahrenheit
}

func apresentar(celsius float32, fahrenheit float32) {
    fmt.Print("Em Celsius: ", celsius)
    fmt.Print("\nEm Fahrenheit: ", fahrenheit)
}

func main() {
    var tempC, tempF float32
    tempC = lerCelsius()
    tempF = calcularFahrenheit(tempC)
    apresentar(tempC, tempF)
}
```

Obs: Go permite que as funções sejam criadas em qualquer ordem. Diferente de C/C++, pode-se criar uma função depois dela ser chamada, mesmo sem prototipação.

Assim como em C/C++, caso se deseje criar variáveis globais, basta declará-las fora das funções, em qualquer lugar, mesmo depois de serem usadas. Mas deve-se criá-las logo no começo do programa para agrupá-las de forma a facilitar a manutenção do código.

Ruby

```
def lerCelsius()
    print "\nDigite a temperatura em Celsius: "
    celsius = gets.chomp.to_f
    return celsius
end

def calcularFahrenheit(celsius)
    fahrenheit = celsius * 9/5 + 32
    return fahrenheit
end

def apresentar(celsius, fahrenheit)
    print "\nEm Celsius: ", celsius
    print "\nEm Fahrenheit: ", fahrenheit
end

tempC = lerCelsius()
tempF = calcularFahrenheit(tempC)
apresentar(tempC, tempF)
```

Obs: Em Ruby, as funções precisam ser criadas (definidas) antes de serem chamadas. Não há o recurso de prototipação, como em C/C++. Porém, caso o programa esteja grande o suficiente para ser um incômodo se preocupar com a ordem das funções, pode-se colocar um grupo de funções em um arquivo a parte com extensão .rb (como minhasfuncoes.rb). E, no arquivo em que a função for chamada, colocar a seguinte instrução no início do código:

require 'minhasfuncoes'

Se quisermos criar variáveis globais, elas devem possuir o símbolo $ antes de seu identificador, como no exemplo a seguir:

$idade = 1

Python

```
def lerCelsius():
    celsius =int(input('Digite a temperatura em Celsius: '))
    return celsius

def calcularFahrenheit(celsius):
    fahrenheit = celsius * 9/5 + 32
    return fahrenheit

def apresentar(celsius, fahrenheit):
    print ('Em Celsius: ' + str(celsius))
    print ('Em Fahrenheit: ' + str(fahrenheit))

tempC = lerCelsius()
tempF = calcularFahrenheit(tempC)
apresentar(tempC, tempF)
```

Obs: Assim como em Ruby, não é possível criar uma função depois de ser chamada. Mas também é possível criar as funções em outro arquivo e importá-los. Por exemplo, cria-se um arquivo minhasfuncoes.py e, no arquivo principal, importa-se o arquivo como segue:

```
from minhasfuncoes import *
```

O asterisco da linha acima significa “todas as funções”.

Para criar variáveis globais, basta que estejam fora de uma função, como em C/C++ e Go.

Exemplos Maiores

Os algoritmos criados até o momento são excelentes para começarmos a entender as vantagens da modularização e como fazê-la. Mas em um sistema real, um programa terá mais de 3 funções (muito mais!).

Nesta seção, começaremos a rascunhar algoritmos para problemas um pouco maiores. Isso expandirá nossos horizontes além da divisão de um algoritmo em função de entrada, cálculo e saída.

Vamos começar, então, por algo familiar: conversão de temperatura!

Imagine que queiramos criar um algoritmo para o problema abaixo.

Um algoritmo deve apresentar o seguinte menu:

1-Converter um valor de Celsius para Farenheit
2-Converter um valor de Farenheit para Celsius
0-Sair

Uma vez que o usuário escolha a opção 1 ou 2, o algoritmo deve pedir a temperatura, fazer a conversão de Celsius para Farenheit ou Farenheit para Celsius (dependendo da opção escolhida) e mostrar o valor original e o valor convertido.
Por fim, o menu deve ser apresentado novamente, caso a opção digitada seja diferente de 0.

Como este algoritmo é familiar, fica mais fácil começarmos o nosso rascunho:

1. Bom, preciso mostrar um menu e esperar alguém digitar algo. Vou criar uma função para isso;

2. Hum... o algoritmo deve converter uma temperatura de Celsius para Fahrenheit. Vou criar uma função para isso;

3. E deve converter uma temperatura de Fahrenheit para Celsius. Também vou criar uma função para isso;

4. Opa, e o algoritmo precisa pedir uma temperatura em Celsius ou em Fahrenheit. Vou criar uma função para pedir temperatura em Celsius e uma função para pedir em Fahrenheit. Eita, mas eu li que na seção *Uma Mesma Função, Pequena Variação* que eu posso aproveitar uma mesma função para realizar tarefas praticamente idênticas que variem apenas em relação às mensagens apresentadas! Então vou criar uma função só!

5. Preciso de uma função para apresentar o resultado;

6. E agora tenho que pensar em como vou fazer para chamar essas funções. Como vou organizar suas chamadas dentro do bloco principal?

7. E como vou fazer para executar tudo de novo enquanto a opção digitada não for igual a 0?

   Vamos começar pelo que a gente já sabe! Vamos criar as funções para cada uma das tarefas que separamos até o item 5. Depois nos preocuparemos com os itens 6 e 7.

   Criaremos, então, as funções:

   - apresentarMenu()
   - lerTemperatura()
   - converterCparaF()
   - converterFparaC()
   - apresentarResultado()

Depois de darmos nomes às funções, vamos pensar no código que vai dentro de cada uma. Em seguida, fazemos aquelas duas perguntas (há algum valor criado/definido na função que será usado em outro lugar? Há algum valor que estou usando nesta função e que veio de fora dela?). Tente planejar cada função antes de continuar a leitura, para comparar a sua versão com a que eu irei apresentar.

Com base nestas perguntas, podemos criar as funções como a seguir:

```
inteiro apresentarMenu()
    inteiro opt
    escreva(“MENU:”)
    escreva(“1-Converter de
          Celsius para Farenheit”)
    escreva(“2-Converter de
          Farenheit para Celsius”)
    escreva(“0-Sair”)
    leia(opt)
    retornar opt
fim
```

Esta função primeiro apresenta um menu. Em seguida espera o usuário digitar algo. **Pergunta 1:** há algum valor criado nesta função e que será usado em outro lugar? Sim! A opção digitada pelo usuário será usada mais tarde para se decidir qual conversão deverá ser realizada. Então precisa retornar um valor.
Pergunta 2: há algum valor que estou usando nesta função e que veio de fora dela?
Não! Então, não há parâmetros.

```
real lerTemperatura(cadeia msg)
    real temp
    escreva(“Digite a temp. em “, msg, “:”)
    leia(temp)
    retornar temp
fim
```

Esta função recebe uma mensagem (que deve ser a cadeia de caracteres “Celsius” ou “Fahrenheit”). Em seguida, apresenta a mensagem “Digite a temp. em <Celsius ou Fahrenheit>:” e espera o usuário digitar. Se tiver dúvida aqui, revise a seção *Uma Mesma Função, Pequena Variação*.
Pergunta 1: há algum valor criado nesta função e que será usado em outro lugar? Sim! A temperatura digitada pelo usuário. Então precisa retornar esse valor.
Pergunta 2: há algum valor que estou usando nesta função e que veio de fora dela?
Sim! A mensagem “Celsius” ou “Fahrenheit”. Então, criaremos um parâmetro do tipo cadeia para receber a mensagem.

```
real converterCparaF(real cel)
    real fah
    fah <- (9*cel/5) + 32
    retornar fah
fim
```

Esta função recebe um valor que representa uma temperatura em Celsius e realiza a conversão para Fahrenheit.
Pergunta 1: há algum valor criado nesta função e que será usado em outro lugar?
Sim! A temperatura convertida. Então precisa retornar esse valor.
Pergunta 2: há algum valor que estou usando nesta função e que veio de fora dela?
Sim! Precisa da temperatura em Celsius previamente digitada. Então, cria-se um parâmetro para que o valor seja recebido quando esta função for chamada.

```
real converterFparaC(real fah)
    real cel
    cel <- 5*(fah-32)/9
    retornar cel
fim
```

Esta função recebe um valor que representa uma temperatura em Fahrenheit e realiza a conversão para Celsius.
Pergunta 1: há algum valor criado nesta função e que será usado em outro lugar?
Sim! A temperatura convertida. Então precisa retornar esse valor.
Pergunta 2: há algum valor que estou usando nesta função e que veio de fora dela?
Sim! Precisa da temperatura em Fahrenheit previamente digitada. Então, cria-se um parâmetro para que o valor seja recebido quando esta função for chamada.

```
nada apresentarResuldado(real cel, real fah)
    escreva("Em Celsius: ", cel)
    escreva("Em Fahrenheit: ", fah)
fim
```

Esta função recebe dois valores que representam as respectivas temperaturas e, em seguida, as apresenta na tela.
Pergunta 1: há algum valor criado nesta função e que será usado em outro lugar?

Não! Então não há retorno.
Pergunta 2: há algum valor que estou usando nesta função e que veio de fora dela?
Sim! Duas! Precisa da temperatura em Celsius e Fahrenheit. Então, cria-se dois parâmetros para que cada valor seja recebido quando esta função for chamada.

Com as funções criadas, precisamos pensar em como criar o bloco ***início-fim***.

Vamos lembrar: o algoritmo deve apresentar o menu e esperar o usuário digitar a opção; deve pedir uma temperatura (em Celsius ou Fahrenheit, dependendo da opção) e esperar o usuário digitar a temperatura; converter o valor; e, apresentar na tela. Rascunhando a ordem!

```
1.  início
2.     inteiro opc
3.     real fah, cel
4.     opc <- apresentarMenu()
5.     se(opc==1)então
6.         cel <- lerTemperatura("Celsius")
7.         fah <- converterCparaF(cel)
8.         apresentarResultado(cel, fah)
9.     senão se(opc==2)então
10.        fah <- lerTemperatura("Fahrenheit")
11.        cel <- converterFparaC(fah)
12.        apresentarResultado(cel, fah)
13.    senão se(opc==0)então
14.        escreva("Tchau!")
15.    senão
16.        escreva("Opção inválida")
17.    fimse
18. fim
```

Este algoritmo está maior do que estamos acostumados. Mas leia com calma e perceberá que a leitura do que está sendo pedido é fácil! A linha 4 apresenta o menu. As linhas 6, 7 e 8 realizam a conversão de Celsius para Fahrenheit (pedindo

a temperatura, convertendo e apresentando). As linhas 10, 11 e 12 realizam a conversão de Fahrenheit para Celsius.

Faltou realizarmos a repetição. Afinal, se a opção for diferente de 0, o código deve ser repetido. Mas primeiro, vamos alterar o rascunho, trocando *se-senão* encadeado por *escolha-caso*:

```
1.   início
2.     inteiro opc
3.     real fah, cel
4.     opc <- apresentarMenu()
5.     escolha(opc)
6.         caso 1:
7.             cel <- lerTemperatura("Celsius")
8.             fah <- converterCparaF(cel)
9.             apresentarResultado(cel, fah)
10.        caso 2:
11.            fah <- lerTemperatura("Fahrenheit")
12.            cel <- converterFparaC(fah)
13.            apresentarResultado(cel, fah)
14.        caso 0:
15.            escreva("Tchau!")
16.        caso padrão:
17.            escreva("Opção inválida")
18.    fimescolha
19.  fim
```

Agora vamos pensar qual instrução de repetição devemos escolher. Sabemos que a regra para escolhermos entre *para*, *enquanto* e *faça-enquanto* é:

para: quando sabemos a quantidade de vezes que as instruções devem ser repetidas. Ou, de outra forma: quando sabemos o intervalo de repetição.

equanto: quando não se sabe de antemão quantas vezes as instruções devem ser repetidas, podendo acontecer de 0 a infinitas repetições;

faça-enquanto: quando não se sabe de antemão quantas vezes as instruções devem ser repetidas, podendo acontecer de 1 a infinitas repetições;

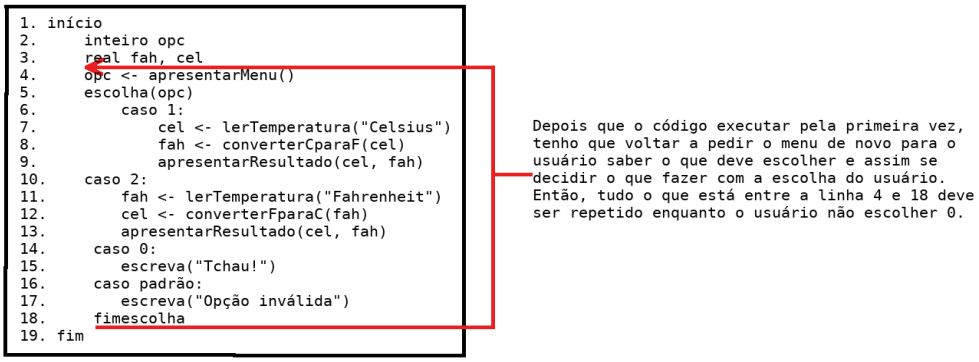
Como não sabemos em que momento o usuário escolherá a opção 0 (SAIR), então descartamos a instrução *para*.

Repare que o uso de ***enquanto*** e ***faça-enquanto*** é praticamente o mesmo. A diferença é: se precisar executar uma sequência de instruções pelo menos uma vez, devemos usar o ***faça-enquanto***.

E é aí que conseguimos decidir: o algoritmo deverá apresentar o menu e realizar o que for escolhido pelo menos uma vez! Nem que seja para o usuário digitar 0 logo de cara. Mas para ele digitar 0, é preciso apresentar o menu pelo menos uma vez e decidir o que fazer com o que o usuário digitar pelo menos uma vez! Então escolheremos ***faça-enquanto***.

Por último, devemos definir quais instruções precisam ficar dentro do bloco ***faça-enquanto*** e o que pode ficar de fora. Podemos decidir no rascunho (Figura 24):

Figura 24 - Escolhendo o código que vai repetir



Deste modo, o código final do bloco INÍCIO-FIM será:

```
início
    inteiro opc
    real fah, cel
    faça
         opc <- apresentarMenu()
             escolha(opc)
                 caso 1:
                     cel <- lerTemperatura(“Celsius”)
                       fah <- converterCparaF(cel)
                     apresentarResultado(cel, fah)
                caso 2:
                     fah <- lerTemperatura(“Fahrenheit”)
                     cel <- converterFparaC(fah)
                     apresentarResultado(cel, fah)
                caso 0:
                     escreva(“Tchau”)
```

```
                outro caso:
                     escreva("Valor inválido")
            fimescolha
    enquanto(opc!=0)
fim
```

Mais de uma Solução para um Mesmo Problema

Esta solução é boa, mas não é a única. Você poderia considerar criar uma única função de conversão que decide internamente qual conversão será realizada, de acordo com a opção enviada. Neste caso, seria enviada a temperatura a ser convertida e a opção escolhida, como abaixo:

```
real converter(real temp, inteiro opc)
    real conv
    se(opc==1)então
        conv <- (9*temp/5) + 32
    senão
        conv <- 5*(temp-32)/9
    fimse
    retornar conv
fim
```

Estamos supondo que a função será chamada apenas se o usuário tiver digitado 1 ou 2. O bloco ***início-fim*** precisaria ser alterado para:

```
início
    inteiro opc
    real fah, cel
    faça
        opc <- apresentarMenu()
            escolha(opc)
                caso 1:
                    cel <- lerTemperatura("Celsius")
                      fah <- converter(cel, opc)
                      apresentarResultado(cel, fah)
                caso 2:
                      fah <- lerTemperatura("Fahrenheit")
                      cel <- converter(fah, opc)
                      apresentarResultado(cel, fah)
                caso 0:
                     escreva("Tchau")
                outro caso:
                      escreva("Valor inválido")
             fimescolha
    enquanto(opc!=0)
fim
```

A pergunta é: qual a melhor solução? Usar uma função para cada conversão, ou uma única função que decida o que deve converter?

Em princípio, não há uma melhor solução. Qualquer uma que se chegue é uma boa solução.

Há momentos, futuramente, em que você irá considerar melhor agrupar vários cálculos inter-relacionados em uma única função mais robusta e situações em que considerará melhor separá-las de acordo com o nível de granularidade que irá querer dar às suas funções.

Por exemplo: você pode considerar que, ao criar uma função mais robusta, você tem que passar por instruções de comparação desnecessárias. Repare que neste último exemplo, temos um ***escolha-caso*** para decidir o que fazer no bloco ***início-fim***, mas temos um ***se-senão*** na função de conversão que precisará verificar novamente qual a opção digitada para realizar a conversão correta. Isso é desnecessário se você criar uma função para cada conversão e chamar a função correta para cada caso logo no bloco ***início-fim***.

Essa comparação a mais atrapalha algo? Não. Deixa o seu sistema mais lento? Não, é pouco demais para influenciar na performance. Então, no fim das contas, depende da sua percepção do que é melhor: agrupar cálculos inter-relacionados em uma única função ou deixá-los separados.

Para resumir a seção. Quando o problema pede várias tarefas:

- Primeiramente, tente identificar cada tarefa e imaginar uma função para cada. De tantos exercícios que já fez, vai começar a ficar cada vez mais fácil identificar as funções que precisará criar;
- Comece criando as funções independentemente de quando devem ser usadas;
- Terminadas as funções, pense como elas devem ser organizadas, isto é, em que ordem devem ser chamadas e em que condições devem ser chamadas;
- Rascunhe o bloco que irá controlar essas chamadas;

- Havendo a possibilidade/necessidade de se repetir o que deve ser feito, defina o começo e o fim da sequência de instruções que deverá ser repetida.

Claro que a sugestão acima é como a palavra diz: uma sugestão. Se você se sente mais confortável em planejar seu algoritmo de outra forma, vá em frente! Parte da diversão de se programar é que Algoritmos são “um jogo de mundo aberto”.

Exercícios Não Tão Básicos Sobre Funções

1. Faça um algoritmo que apresente o seguinte MENU:

MENU:
1-Converter quilômetros em milhas
2-Converter milhas em quilômetros
0-SAIR

Uma vez que o usuário escolha a opção 1 ou 2, o algoritmo deve pedir a distância, fazer a conversão de km para mi ou de mi para km (dependendo da opção escolhida) e mostrar o valor original e o valor convertido.

Por fim, o menu deve ser apresentado novamente, caso a opção digitada seja diferente de 0.

Este exercício é propositalmente semelhante ao do exemplo da seção Exemplos Maiores para termos uma base para começarmos o rascunho. Neste exemplo, além das funções de entrada e saída, crie uma função para a conversão de km para mi e uma outra função para a conversão de mi para km.

2. Faça um algoritmo que apresente o seguinte MENU:

MENU:
1-Converter quilômetros em milhas
2-Converter milhas em quilômetros
0-SAIR

Uma vez que o usuário escolha a opção 1 ou 2, o algoritmo deve pedir a distância, fazer a conversão de km para mi ou de mi para km (dependendo da opção escolhida) e mostrar o valor original e o valor convertido.

Por fim, o menu deve ser apresentado novamente, caso a opção digitada seja diferente de 0.

Neste exemplo, além das funções de entrada e saída, crie apenas uma função para a conversão de km para mi e a conversão de mi para km. Caso tenha dificuldade, recorde a seção Mais de uma Solução para Um Mesmo Problema.

3. Faça um algoritmo de pocketdex. O pocketdex deve poder armazenar 5 pocketmon. Cada pocketmon possui um campo nome, vida, ataque e defesa. O algoritmo, primeiramente deve apresentar o seguinte menu:

MENU:
1. Cadastrar um pocketmon no pocketdex.
2. Apresentar pocketmon.
0. Sair

Se o usuário escolher a opção 1, o algoritmo deverá pedir o nome, vida, ataque e defesa do pocketmon a ser cadastrado. Para isso, crie um registro pocketmon com os respectivos campos. Os valores digitados devem ser armazenados sempre em sequência no pocketdex (primeiramente no espaço 0, depois 1, 2, 3 e finalmente 4). Mas se o usuário tentar cadastrar um sexto pocketmon, o algoritmo não deve permitir.

Se o usuário escolher a opção 2, o algoritmo deverá pedir o número do pocketmon (um valor entre 0 e 4) e apresentar os dados do pocketmon já cadastrados. O valor precisa ser válido (entre 0 e 4). Do contrário, o algoritmo deve alertar que o pocketdex permite apenas valores entre 0 e 4.

Por fim, se o usuário não digitar 0, o algoritmo deve voltar a apresentar o menu e tudo se repete.

Este algoritmo é muito semelhante ao da seção de registros. Mas agora cada tarefa deve estar contida em uma função.

4. Faça um algoritmo de jogo da velha!

5. Invente seu próprio algoritmo.

Funções Chamam Funções

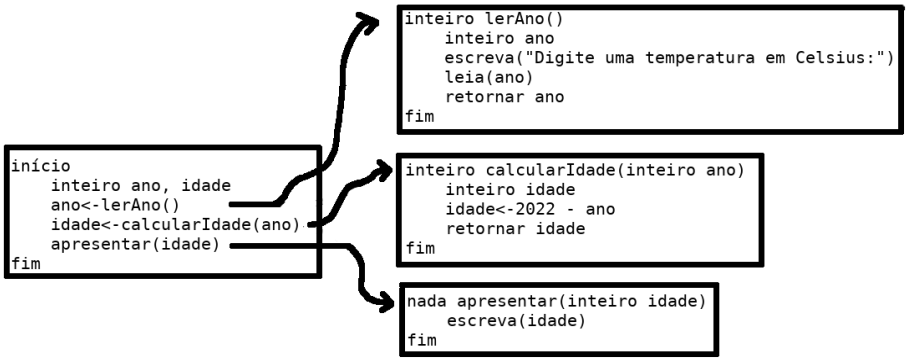
Quando começamos a aprender sobre funções, é muito comum surgir uma pergunta: uma função pode chamar outra função? Sim! E isso inevitavelmente irá acontecer durante o desenvolvimento de sistemas. Até o momento, não tivemos este tipo de situação porque os exemplos eram muito simples.

Simples para quem não está em fase de aprendizagem! Porque não é nada simples resolver as listas de exercícios propostos até o momento se você nunca teve contato com algoritmos. Por outro lado, se você pegar a primeira lista de exercícios proposta no início do livro, verá que aqueles exercícios com apenas escreva e leia se tornaram simples para você!

Mas, mesmo nos exemplos anteriores, seria possível adotar uma solução em que uma função chama a outra. Vamos pegar o exemplo do cálculo da idade. Nele, precisamos criar um algoritmo que pede o ano de nascimento do usuário, calcula a idade (com possível margem de erro de um ano) e apresenta na tela.

A nossa solução pode ser apresentada na Figura 25:

Figura 25 - Uma relação entre funções

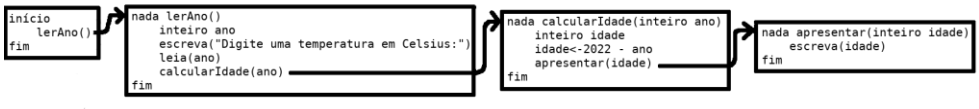


Com esta imagem, é possível notar que todo o controle sobre qual função será executada está no bloco ***início-fim***. Isto é, as funções são chamadas apenas por este bloco. E isso pode nos causar a impressão errada de que as funções só podem ser chamadas do bloco inicial.

Para começar, o próprio bloco ***início-fim*** é uma função em muitas linguagens de programação, como C/C++, Go e Java! Então, as funções já estão sendo chamadas de uma outra função.

Mas esse mesmo exemplo poderia ter sido pensado da seguinte forma (Figura 26):

Figura 26 - Outra possível relação entre funções



Nesta solução, o bloco ***início-fim*** chama apenas a função lerAno(). A função lerAno() pede para o usuário digitar algo. Em seguida, a própria função lerAno() chama a função calcularIdade() e passa o ano para ela. A função calcularIdade() será executada, calculando a provável idade do usuário e ela mesma chamará a função apresentar, passando a idade calculada. A função calcular apresentará a idade na tela.

O mais interessante é que esta solução parece muito intuitiva e mais simples! Repare que o bloco ***início-fim*** não precisou de variáveis e não precisamos nos preocupar com retorno de valores. Nenhuma função deste exemplo retorna valores!

Essa primeira impressão faz com que achemos essa solução bem melhor que a outra porque é bem linear no sentido de que "eu faço algo e mando para outro continuar; o outro continua o que tem que fazer e manda um outro continuar; e assim por diante".

MAS! Será que é melhor?

Vamos nos lembrar porque estamos modularizando um código: para torná-lo melhor organizado e para poder reusar código caso necessário.

Organizado esse último exemplo até que está. O problema recai no reúso.

Repare que no primeiro exemplo, a única tarefa da função lerAno() é ler o ano! Ela faz o que deve e entrega o resultado. O mesmo ocorre com as outras duas funções.

Mas repare que, no segundo exemplo, a função lerAno() não apenas lê o ano de nascimento do usuário, mas é responsável por chamar uma função que fará um cálculo de idade.

E se eu quiser reusar a função lerAno() em uma outra situação em que não será feito um cálculo de idade?

Percebe que a função lerAno() possui uma relação de dependência com calcularIdade() no segundo exemplo da Figura **Erro! Fonte de referência não encontrada.**? Se essa relação de dependência é algo que nem sempre deveria estar presente, então dificulta o reúso.

A regra que costumo usar é: sempre que possível, tente criar funções que não dependam de outra (que não chame outra função). Costumamos chamar essa propriedade de desacoplamento (quanto mais desacoplado, menos dependência uma função tem com outras funções). Uma segunda propriedade relacionada a esta é a coesão: quanto mais um módulo realiza uma única responsabilidade, mais coesa ela é. Quanto mais desacoplado e coeso é um módulo, mais fácil ele é de ser reusado.

No caso da nossa segunda solução, a função lerAno() é mais acoplada em relação à primeira solução porque ela chama uma função que não é necessária para ela realizar sua tarefa de ler o ano de nascimento. Da mesma forma, ela é menos coesa porque além da responsabilidade de pedir um ano de nascimento, ela também é responsável por chamar uma outra função que não tem relação direta com a tarefa dela.

Sempre é possível criar funções independentes como aquelas do primeiro exemplo? Não! Como exemplo: geralmente temos um grupo de funções que realizam tarefas independentes e outro grupo de funções que são responsáveis por gerenciar a chamada destas funções. De forma prática: funções responsáveis por cálculos e regras de negócio são mais fáceis de serem criadas de forma coesa e desacoplada; já funções que gerenciam ou controlam o fluxo do algoritmo são propositalmente mais acopladas.

Vamos a um exemplo em que temos este cenário que acabei de citar.

Imagine um algoritmo que apresente um menu para o usuário decidir se quer fazer conversão de temperatura ou conversão de distância. Se ele escolher a primeira opção, um novo menu com opções de conversão de temperatura deverá ser apresentado. Do contrário, um novo menu com opções de conversão de distância deverá ser apresentado:

- Apresentar menu com opções (1) de conversão de temperatura e (2) de distância.
- Se escolhida a opção de temperatura:
    - Apresentar menu com opções de (1) conversão de Celsius para Fahrenheit e (2) conversão de Fahrenheit para Celsius.
    - Se escolhida a opção 1:
        - Pedir temperatura em Celsius, converter para Fahrenheit e apresentar;
    - Se escolhida a opção 2:
        - Pedir temperatura em Fahrenheit, converter para Celsius e apresentar;
- Se escolhida a opção de distância:
    - Apresentar menu com opções de (1) conversão de quilômetros para milhas e (2) conversão de milhas para quilômetros.
    - Se escolhida a opção 1:
        - Pedir distância em km, converter para mi e apresentar;
    - Se escolhida a opção 2:
        - Pedir distância em mi, converter para km e apresentar;

Reparem que a primeira ação foi colocar o que deve ser feito de forma organizada, em Português, mesmo: em relação à ordem do que deve ser pedido, as condições para que cada tarefa seja realizada e uma tabulação (indentação) das tarefas em relação às outras de acordo com a relação de dependência (por exemplo, a relação entre digitar o 2 e a próxima tarefa de pedir a distância. Repare que pedir a distância está tabulada em relação à condição de escolher a opção 2).

Esse rascunho em Português nos ajuda a criar algoritmos quando não temos prática ou temos dificuldade em visualizar mentalmente o algoritmo.

Uma vez que organizamos o nosso código, podemos pensar nas funções que serão criadas:

- apresentarMenuPrincipal();
- apresentarMenuTemperatura();
- apresentarMenuDistancia();
- converterCparaF();
- converterFparaC();
- converterKmparaMi();
- converterMiparaKm();
- lerTemperatura();
- lerDistancia();
- apresentarTemperaturas();
- apresentarDistancias().

Uau! Pensamos em 11 funções! Isso é um recorde!

Apesar da quantidade, você provavelmente não teve dificuldade em identificar as funções listadas acima. Pode ter havido algumas diferenças. Por exemplo, é possível que tenha escolhido criar uma única função converterTemperatura() como discutido na seção *Mais de uma Solução para Um Mesmo Problema*.

Agora precisamos determinar como as funções serão chamadas. De acordo com o texto que organizamos anteriormente, poderíamos criar um bloco ***início-fim*** como a seguir:

```
início
    inteiro menuP, menuT, menuD
    real fah, cel, mi, km

    faça
        menuP <- apresentarMenuPrincipal()
        escolha(menuP)
```

```
            caso 1:
                menuT<-apresentarMenuTemperatura()
                escolha(menuT)
                    caso 1:cel←
                        cel<-lerTemperatura("Celsius")
                        fah<-converterCparaF(cel)
                        apresentarTemperatura(cel, fah)
                    caso 2:
                        fah<-lerTemperatura("Fahrenheit")
                        cel<-converterCparaF(fah)
                        apresentarTemperatura(cel, fah)
                fimescolha
            caso 2:
                menuD<-apresentarMenuDistancia()
                escolha(menuD)
                    caso 1:
                        km<-lerDistancia("quilômetros")
                        mi<-converterKmparaMi(km)
                        apresentarDistancia(km, mi)
                    caso 2:
                        mi<-lerDistancia("Milhas")
                        km<-converterMiparaKm(mi)
                        apresentarDistancia(mi, km)
                fimescolha
        fimescolha
    enquanto(menuP!=0)
fim
```

Você pode considerar que esse código do bloco ***início-fim*** está muito grande e de difícil entendimento, afinal, este bloco está gerenciando três grupos de tarefas: (i) definir qual tipo de conversão (temperatura ou distância) será escolhido; (ii) o que fazer se for temperatura; e, (iii) o que fazer se for distância. Repare que os grupos (ii) e (iii) dependem de qual tipo de conversão será escolhido (i).

Uma vez que identificamos que existe essa relação de dependência, podemos simplificar o bloco ***início-fim*** pensando o seguinte: vou deixar como única responsabilidade do bloco ***início-fim***, a decisão de qual tipo de conversão será escolhido. Os grupos (ii) e (iii) serão separados em duas funções. Deste modo, o bloco pode ser simplificado para:

```
início
    inteiro menuP
```

```
    faça
        menuP <- apresentarMenuPrincipal()
        escolha(menuP)
            caso 1:
                oferecerConversaoTemperatura()
            caso 2:
                oferecerConversaoDistancia()
        fimescolha
    enquanto(menuP!=0)
fim
```

Perceba como o bloco *início-fim* foi simplificado. A única responsabilidade dele é definir qual tipo de conversão será oferecido ao usuário. Fica mais fácil entender o que ele faz quando se tentar ler o código. O que precisamos agora, é criar as funções oferecerConversaoTemperatura() e oferecerConversaoDistancia().

```
nada oferecerConversaoTemperatura()
    inteiro menuT
    real cel, fah
    menuT<-apresentarMenuTemperatura()
    escolha(menuT)
        caso 1:
            cel<-lerTemperatura("Celsius")
            fah<-converterCparaF(cel)
            apresentarTemperatura(cel, fah)
        caso 2:
            fah<-lerTemperatura("Fahrenheit")
            cel<-converterCparaF(fah)
            apresentarTemperatura(cel, fah)
    fimescolha
fim
```

e

```
nada oferecerConversaoDistancia()
    inteiro menuD
    real mi, km
    menuD<-apresentarMenuDistancia()
    escolha(menuD)
        caso 1:
            km<-lerDistancia("quilômetros")
            mi<-converterKmparaMi(km)
```

```
            apresentarDistancia(km, mi)
        caso 2:
           mi<-lerDistancia("Milhas")
            km<-converterMiparaKm(mi)
            apresentarDistancia(mi, km)
    fimescolha
fim
```

Assim, criamos uma função cuja responsabilidade é oferecer as conversões de temperatura e uma outra função cuja responsabilidade é oferecer as conversões de distância.

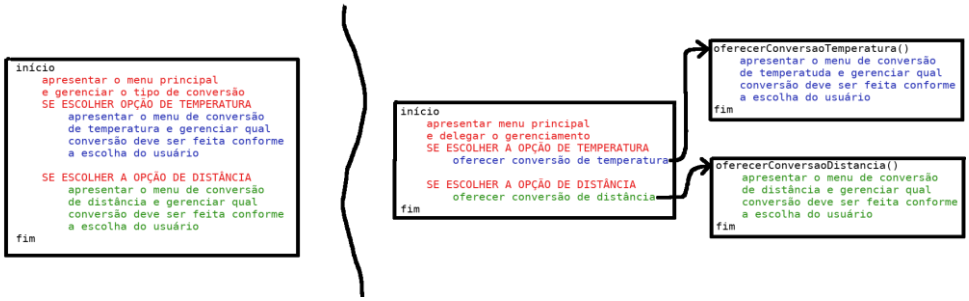
Estes são dois exemplos de funções que criamos sabendo que serão acopladas (porque precisam chamar diversas outras funções a partir delas). E isso nem sempre é um problema porque estas são funções que gerenciam tarefas. Se uma função tem como objetivo gerenciar a execução de outras funções então ela será naturalmente acoplada.

Já funções como as de conversão são desacopladas porque não precisam chamar outras funções. E isso é mais comum em funções que realizam cálculo ou que possuem alguma regra de negócio.

Mesmo que as duas novas funções sejam acopladas, ainda assim foi interessante criá-las porque originalmente o bloco ***início-fim*** estava com muitas responsabilidades diferentes, isto é, estava pouco coeso. Separando as responsabilidades, criamos funções mais coesas. E isso é bom porque a função faz apenas uma tarefa (ou poucas), fica mais simples e simplicidade ajuda na manutenção e na facilidade de leitura por parte do programador.

Podemos visualizar a diferença entre o código original e a última solução de maneira abstrata na Figura 27. Aqui nos concentramos apenas na relação entre as funções que controlam o fluxo da execução do algoritmo:

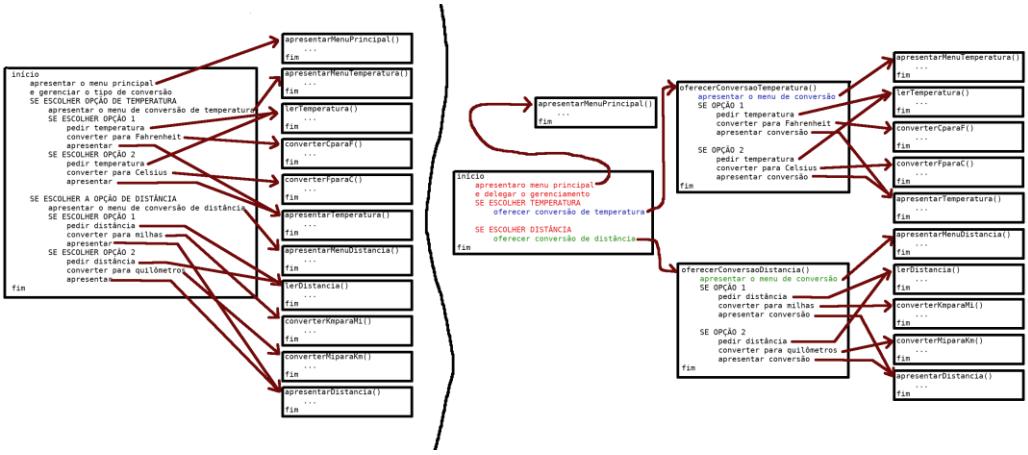
Figura 27 - Dividindo responsabilidades



Com as imagens anteriores, fica mais fácil identificar a divisão de tarefas em mais funções.

Agora, com as demais funções que apresentam mensagens e fazem cálculos, podemos visualizar o todo na Figura 28.

Figura 28 - Dividindo as responsabilidades das funções ainda mais



Com o diagrama da esquerda é fácil perceber que o bloco ***início-fim*** tem muitas responsabilidades. Ele tem que chamar (e gerenciar) 11 tarefas diferentes. Isto é, ele é muito acoplado em relação ao exemplo da direita porque depende de muitas funções.

No diagrama da direita, o bloco ***início-fim*** chama 3 tarefas. Deste modo, tornamos o bloco ***início-fim*** menos acoplado.

As funções:

```
oferecerConversaoTemperatura()
```

e

```
oferecerConversaoDistancia()
```

Gerenciam 5 tarefas cada uma.

Do mesmo modo, o que fizemos da esquerda para a direita foi modularizar melhor o código de forma a separar as responsabilidades de gerenciamento das tarefas em duas funções (oferecerConversaoTemperatura() e oferecerConversaoDistancia()). Esta modularização tornou as funções de controle de fluxo de programa mais coesas (***início-fim*** e as duas que oferecem conversão) porque cada uma tem responsabilidades de controlar grupos de tarefas específicos.

Exercício Sobre Funções que Chamam Funções

1. Faça um algoritmo que apresente um menu para o usuário decidir se quer exercitar fatos sobre (1) biologia ou (2) geografia.

- Se o usuário escolher a primeira opção, apresentar um menu com as opções: (1) Exercitar sobre artrópodes ou (2) sobre vertebrados.
  - Se o usuário escolher 1, o algoritmo deve perguntar: quantas patas tem um inseto? Se o usuário responder 6, o algoritmo deve apresentar a mensagem "Correto". Do contrário, o algoritmo deve apresentar a mensagem "Errado" e pedir para tentar de novo.
  - Se o usuário escolher 2, o algoritmo deve perguntar: qual destes animais não pertence aos vertebrados (1-Mamífero; 2-Réptil; 3-Ave; 4-Molusco). Se o usuário responder 4, o algoritmo deve apresentar a mensagem "Correto". Do contrário, o algoritmo deve apresentar a mensagem "Errado" e pedir para tentar de novo.
- Se o usuário escolher a segunda opção do primeiro menu, um novo menu com as opções: (1) Exercitar sobre países ou (2) exercitar sobre continentes;
  - Se o usuário escolher 1, o algoritmo deve perguntar: qual destes países não faz fronteira com o Brasil: (1) Colômbia; (2) Chile; (3) Venezuela. Se o usuário responder 2, o algoritmo deve apresentar a mensagem "Correto". Do contrário, o algoritmo deve apresentar a mensagem "Errado" e pedir para tentar de novo.
  - Se o usuário escolher 2, o algoritmo deve perguntar: qual destes não é um continente (1-Ásia; 2-África; 3-Atlântida). Se o usuário responder 3, o algoritmo deve apresentar a mensagem "Correto". Do contrário, o algoritmo deve apresentar a mensagem "Errado" e pedir para tentar de novo.

Capítulo 4: O Jogo da Velha

Agora, sim! Vamos pegar tudo aquilo que aprendemos e criar um projeto!

E por que um jogo da velha?

- Porque é legal!
- Porque é grande o suficiente para que o algoritmo precise ser planejado;
- Porque é grande o suficiente para aplicarmos todos aqueles conceitos que aprendemos;
- Porque não é tão grande que fique inviável escrevê-lo;
- Porque ainda podemos aprender algo novo;
- Porque há várias soluções possíveis para serem exploradas;
- Porque podemos começar do básico e tentar melhorar o código passo a passo;
- Porque é legal!!!

A beleza de se estudar Algoritmos é saber que há várias soluções possíveis. Começaremos este projeto planejando da forma mais intuitiva possível até criarmos nossa primeira versão. A partir dela, faremos algumas análises com o objetivo de encontrarmos soluções diferentes.

Esse exercício servirá para que você entenda duas coisas importantes na programação:

- O mais intuitivo é um bom ponto de partida!
- Nosso código pode ser melhorado, e isso é normal!
- Soluções alternativas nem sempre significam soluções melhores ou piores.

Planejamento das Jogadas

Quando planejamos um projeto um pouco maior do que aqueles propostos nas listas de exercícios, é muito interessante focarmos primeiro em “o que” devemos

fazer, e não no "como". Assim, quando formos planejar o jogo da velha, podemos pensar primeiro nos passos que os jogadores realizam durante a partida:

- O jogador 1 observa o tabuleiro;
- O jogador 1 escolhe uma casa vazia;
- Verifica se ganhou;
- O jogador 2 observa o tabuleiro;
- O jogador 2 escolhe uma casa vazia;
- Verifica se ganho;
- Repete os passos acima até alguém ganhar ou não ter mais casas vazias.

Os passos são bastante simples! A partir deles, podemos começar nosso primeiro rascunho das funções que criaremos e as suas ordens de chamada, no bloco principal. O rascunho abaixo faz muito sentido (mas está incompleto):

```
1. início
2.     logico vitoria
3.     vitoria <- falso
4.     faça
5.         apresentarTabuleiro()
6.         jogada1()
7.         vitoria <- verificarVitoria()
8.         apresentarTabuleiro()
9.         jogada2()
10.        vitoria <- verificarVitoria()
11.    enquanto(vitoria==falso)
12. fim
```

O algoritmo acima é praticamente uma transcrição do que definimos em Português. Para identificarmos se um jogador ganhou ou não, a função **verificarVitoria()** deverá retornar **verdadeiro** caso tenha ganhado ou **falso** em caso contrário. Desta forma, o laço apenas se repetirá se não houver vitória.

Mas há um problema de lógica. Supondo que o jogador 1 tenha vencido (o que é identificado na linha 7, após a execução da função verificarVitoria()), o algoritmo ainda permitirá que o jogador 2 faça a sua jogada, porque as linhas 8, 9 e 10 também serão executadas.

Então precisaríamos colocar uma condição ***se*** após a linha 7, para que as próximas instruções apenas sejam executadas se o jogador 1 não ganhou:

```
1. início
2.     logico vitoria
3.     vitoria <- falso
4.     faça
5.         apresentarTabuleiro()
6.         jogada1()
7.         vitoria <- verificarVitoria()
8.         se(vitoria==falso)então
9.             apresentarTabuleiro()
10.            jogada2()
11.            vitoria <- verificarVitoria()
12.        fimse
13.    enquanto(vitoria==falso)
14. fim
```

Repare que não é preciso colocar uma condição ***se*** para a primeira jogada, porque ou o jogo apenas começou e com certeza ninguém venceu na primeira jogada, ou o jogador 2 pode ter vencido, mas o algoritmo sairá do laço na linha 13 e o jogador 1 não correrá o risco de jogar caso o jogador 2 tenha vencido.

Aqui começamos a sentir a tal "intuição" de que algo pode ser melhorado, se estivermos dominando Algoritmos. Isso porque os passos a serem realizados para o jogador 1 são exatamente iguais para o jogador 2! Em ambos os casos, deve-se mostrar o tabuleiro, esperar a jogada do jogador da vez e ver se o jogador da vez ganhou!

Colocando no papel a nossa intuição, podemos pensar o seguinte: se a única coisa que varia nos passos é qual o jogador está jogando, então, podemos tentar eliminar a repetição de passos, enviando para a função qual é o jogador da vez! Para isso, podemos criar uma variável do tipo inteiro que armazena qual o jogador da vez! E, no final da jogada, precisamos trocar o valor da variável para o próximo jogador.

```
1. início
2.     logico vitoria
3.     inteiro jogador
4.     jogador <- 1
5.     vitoria <- falso
6.     faça
7.         apresentarTabuleiro()
```

```
8.          jogada(jogador)
9.          vitoria <- verificarVitoria(jogador)
10.         jogador < - vezDeQuem(jogador)
11.    enquanto(vitoria==falso)
12. fim
```

Repare como o código ficou mais simples:

- Inicialmente, dizemos que o jogador da vez é o 1 (isso é feito na linha 4);
- Ao se entrar no laço, o tabuleiro é apresentado;
- Agora, a jogada é feita, e é a vez do jogador 1, porque 1 é o valor da variável *jogador*, neste momento;
- Agora, a função chamada na linha 7 verifica se houve vitória do jogador 1 (por isso passamos o valor da variável *jogador* para a função);
- Na linha 8, a função vezDeQuem(jogador) retornará quem é o próximo jogador. Como isso será feito, nos preocuparemos mais tarde. Mas sabemos que é importante atualizar a vez do próximo jogador. Nesta vez, a função retornará 2 (porque o jogador 1 acabou de jogar e é a vez do jogador 2);
- Se a vitória não foi alcançada, voltamos no laço. A beleza é: os mesmos passos vão se repetir, mas agora para o jogador 2!

Essa solução que alcançamos não é a "única" certa. Mas parece uma boa solução. Por isso, nós a manteremos.

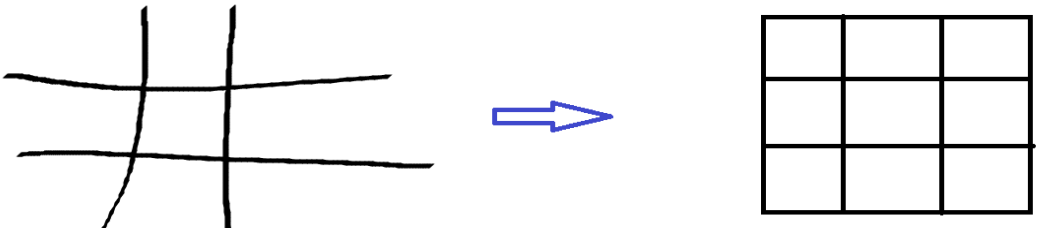
Agora planejaremos "como" cada tarefa (função) será realizada.

Sugestão: para algumas funções, serão apresentadas mais de uma versão. Você pode se ater apenas à primeira versão de cada e tentar implementá-las em uma linguagem de escolha. E, em um segundo momento, partir para as outras versões. É importante compreender bem uma versão, para seguir para a outra.

Planejamento do Tabuleiro

Antes de planejarmos como o tabuleiro será apresentado na tela, precisamos pensar em como o tabuleiro será representado pelo próprio algoritmo. Bem, vamos visualizar um tabuleiro de jogo da velha. Com que estrutura ele parece?

Figura 29 - Tabuleiro pode ser visto como uma matriz

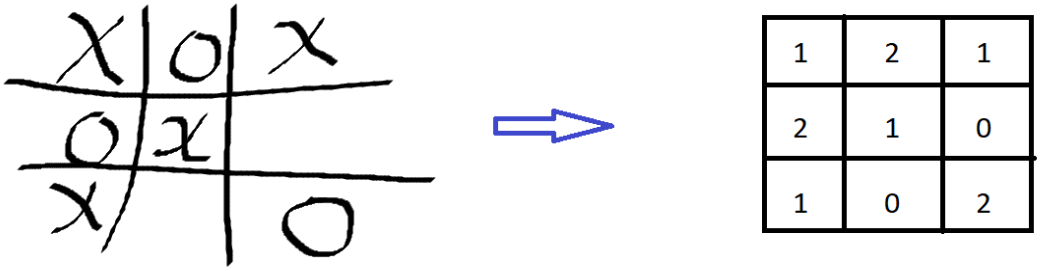


Yeap! Ele se parece muito com uma matriz de 3x3! Como na Figura 29.

Então, para representarmos o tabuleiro, criaremos uma matriz de 3x3. E o que iremos armazenar? Podemos usar o número 0 para representar uma casa vazia, o 1 para uma casa escolhida pelo jogador 1, e 2 para uma casa escolhida pelo jogador 2.

Como exemplo, um tabuleiro como o da Figura XXX pode ser representado pelo algoritmo como o da respectiva tabela à direita na Figura 30:

Figura 30 - Como representar as jogadas na matriz



Agora que escolhemos a matriz como representação, precisamos escolher onde ela será declarada. E este é um excelente exemplo de quando uma variável pode ser definida como global!

Pense bem! Esta variável sempre servirá para representar o tabuleiro. Não há risco de usarmos essa matriz para outra finalidade! E não há mais de um tabuleiro. Pela sua característica única, podemos declará-la, então, como **variável global**.

O problema do Lixo de Memória

Precisamos ficar atentos à linguagem de programação que usaremos para implementar nossa solução. Muitas linguagens, como Go e Java, iniciam as variáveis numéricas declaradas automaticamente com o valor 0.

Portanto, se declararmos uma variável numérica e pedirmos para seu valor ser apresentado na tela, este seria 0. Mas há linguagens que não fazem isso, como C/C++. Nestas linguagens, se uma variável for declarada, mas nenhum valor for atribuído a ela, o valor a ser apresentado será aquilo que estiver ocupando a memória naquele momento. Que pode muito bem ser 0 ou um número completamente aleatório sob o ponto de vista do usuário. E não queremos isso! Precisamos que, no começo da partida, todas as casas do nosso tabuleiro tenham o valor 0.

Deste modo, se usarmos linguagens como estas, precisamos atribuir valores assim que declararmos a variável. Isso pode não ser prático se a matriz ou vetor for grande. Nestes casos, podemos criar uma função para colocar o valor 0 em cada posição da matriz. Uma solução genérica, independente da linguagem seria:

```
nada zerarTabuleiro()
    inteiro i, j
    para(i <- 0; i < 3; i++)faça
        fimpara(j <- 0; j < 3; j++)faça
            tabuleiro[i][j] <- 0
        fimpara
    fimpara
fim
```

Não se esquecendo de que esta função precisaria ser chamada no início do bloco principal de nosso algoritmo.

Planejamento da função apresentarTabuleiro()

Precisamos, agora, criar uma função que apresente o tabuleiro e seu estado atual para o jogador da vez. Faremos da forma mais simples, que é apresentar na tela, os valores 0, 1 e 2 para cada casa. Como não estamos preocupados com gráficos, faremos a apresentação com caracteres normais de *prompt* de comando.

Então, se pegarmos o exemplo da Figura **Erro! Fonte de referência não encontrada.**, o tabuleiro será apresentado como segue:

```
1 | 2 | 1
----------
2 | 1 | 0
----------
1 | 0 | 2
```

Não é a tela de jogo dos sonhos, mas é bem funcional.

A função precisa apresentar o valor de cada célula, com caracteres de separação entre elas(|), e apresentar vários sinais de menos (-) entre as linhas. Como o tabuleiro é muito pequeno, não precisamos nos preocupar com laços de repetição. É realmente mais fácil usarmos várias instruções ***escreva***. Abaixo, apresentamos a declaração da variável global e a função apresentarTabuleiro().

```
inteiro tabuleiro[3][3]

nada apresentarTabuleiro()
    escreva(tabuleiro[0][0],"  |  ",tabuleiro[0][1],"  |  ",tabuleiro[0][2])
    escreva("------------------")
    escreva(tabuleiro[1][0],"  |  ",tabuleiro[1][1],"  |  ",tabuleiro[1][2])
    escreva("------------------")
    escreva(tabuleiro[2][0],"  |  ",tabuleiro[2][1],"  |  ",tabuleiro[2][2])
fim
```

Esse é o resultado que adotaremos como final! Mas vamos discutir um pouco outras soluções! Nada impede que usemos um laço ***para*** que apresente uma linha inteira por vez; ou mesmo laços ***para*** aninhados para apresentar cada célula por vez. Faria muito sentido em um tabuleiro maior.

Por outro lado, teríamos um certo trabalho para eliminarmos as exceções do desenho. Por exemplo, os traços entre as casas (|) não devem aparecer nem no começo nem no fim da linha. Ainda, os traços entre as linhas devem ser desenhadas apenas entre a linha 0 e 1, e a linha 1 e 2. Mas não devem aparecer antes da linha 0 ou depois da linha 2, como abaixo:

```
----------
1 | 2 | 1
----------
2 | 1 | 0
----------
1 | 0 | 2
----------
```

Para gerenciar esta última exceção, precisaríamos usar uma condição ***se***. O que deixaria o código praticamente do mesmo tamanho da nossa primeira solução. Usando uma solução com um laço ***para***, teríamos:

```
inteiro tabuleiro[3][3]

nada apresentarTabuleiro()
  inteiro i
  para(i <- 0; i < 3; i++)faça
    escreva(tabuleiro[i][0],"  |  ",tabuleiro[i][1],"  |  ",tabuleiro[i][2])
      se(i<2)então
        escreva("------------------")
      fimse
  fimpara
fim
```

No exemplo acima, desenhamos o tracejado apenas após as linhas, o que já evita o inconveniente do primeiro tracejado acima da primeira linha, mas temos ainda o problema de aparecer um tracejado após a última linha; a condição ***se*** evita justamente este problema. Observando as duas soluções, vemos que o uso de laço não facilita a apresentação do tabuleiro. Mas não é um crime ou um erro fazer desta forma.

Podemos apresentar um tabuleiro ainda mais simplificado, apenas com a jogada de cada casa, como abaixo:

1 2 1
2 1 0
1 0 2

Nesta situação, não precisamos nos preocupar com as exceções acima mencionadas e fica simples usarmos laço ***para*** aninhado:

```
1. inteiro tabuleiro[3][3]

2. nada apresentarTabuleiro()
3.     inteiro i
4.     para(i <- 0; i < 3; i++)faça
5.         escreva("\n")
6.         para(j <- 0; j < 3; j++)faça
7.             escreva(tabuleiro[i][j], "   ")
8.         fimpara
9.     fimpara
10. fim
```

Faça de conta, especificamente neste exemplo, que para pular uma linha, precisamos usar o marcador \n. Sem ele, as instruções ***escreva*** não pulam linha. Isto é comum às linguagens de programação.

A cada três casas apresentadas do tabuleiro, será pulada uma linha (5). Em 7, é apresentada cada casa, concatenada a um espaço para dar distância entre elas. Lembre-se de que estamos considerando que a instrução ***escreva*** não pula linha sem ter o marcador \n.

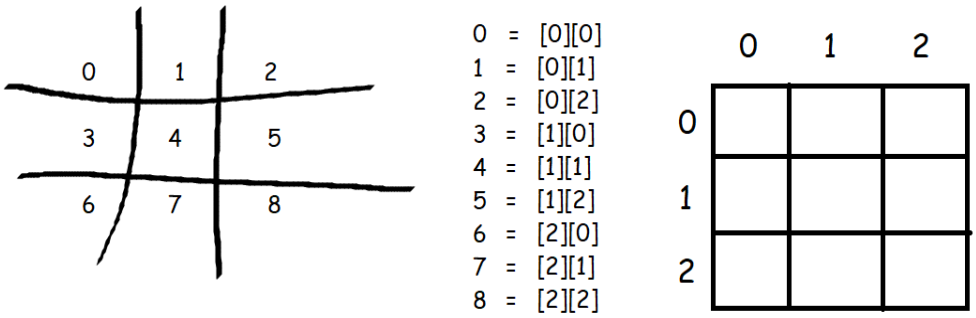
Esta é uma boa solução também! Ainda que colocar apenas três linhas com instruções **escreva** seja mais simples para nosso caso! E como descobrimos qual solução é a melhor? Depois que adquirimos bastante experiência com programação, muitas vezes a melhor solução já vem à cabeça. Mas se não vier, é só testarmos as soluções como fizemos aqui. Normal! É um trabalho de cinco ou dez minutos.

Planejamento da Função jogada() Versão 1

Agora que o jogador consegue enxergar como está o tabuleiro, ele precisa realizar a sua jogada. Para ele escolher a casa, poderíamos pensar em enumerá-las de 0 a 8 (ou de 1 a 9). E, para cada escolha, precisaríamos mapeá-las para a respectiva posição da matriz.

Então, teríamos o seguinte mapeamento de posições, como na Figura 31.

Figura 31 - Rascunho do mapeamento das posições do tabuleiro



Ou podemos pedir para o usuário digitar a linha e a coluna da jogada. Sob o ponto de vista de programação, essa solução é mais simples porque não precisamos fazer mapeamento. Então, vamos começar por ela (faremos como na Figura 31 na Versão 2, próxima seção).

Para facilitar a visualização do usuário, podemos mudar um pouco a função apresentarTabuleiro() original, enumerando as linhas e colunas, como abaixo:

```
nada apresentarTabuleiro()
   escreva(“ 0   1   2”)
   escreva(“ “)
   escreva(“0:“,tabuleiro[0][0],“ | “,tabuleiro[0][1],“ | “,tabuleiro[0][2])
   escreva(“  ------------------”)
   escreva(“1:“,tabuleiro[1][0],“ | “,tabuleiro[1][1],“ | “,tabuleiro[1][2])
   escreva(“  ------------------”)
   escreva(“2:“,tabuleiro[2][0],“ | “,tabuleiro[2][1],“ | “,tabuleiro[2][2])
fim
```

Esta alteração, fará com que o tabuleiro seja apresentado como abaixo:

```
   0   1   2
0: 0 | 0 | 0
    ———————
1: 0 | 0 | 0
    ———————
2: 0 | 0 | 0
```

Vamos voltar à nossa função de jogada(). Primeiro, vamos planejá-la. Ela precisa:

- Pedir para o usuário digitar a linha e a coluna que o jogador quer marcar;
- Atribuir à respectiva posição da matriz, o número 1 ou 2, dependendo de qual jogador está fazendo a jogada (então, precisamos passar para essa função, qual jogador está fazendo a jogada!).
- Mas não é só isso! É preciso verificar se a jogada é válida para que não corra o risco de jogar por cima de uma casa que já esteja ocupada! Ou mesmo se o usuário digitou um valor válido entre 0 e 2!

Podemos estruturar da seguinte forma: colocamos estes três passos dentro do bloco ***faça-enquanto***. Se, após a verificação, a jogada não for válida, começa de novo! Algo como:

```
1.    nada jogada(inteiro jogador)
2.        inteiro linha, coluna
3.        lógico válido
4.        escreva("Vez do jogador ", jogador, ": ")
5.        faça
6.            escreva("Digite a linha: ")
7.            leia(linha)
8.            escreva("Digite a coluna: ")
9.            leia(coluna)
10.           se(linha>2 || coluna>2 || linha<0 || coluna<0)ENTÃO
11.               valido = falso
12.           senão se(tabuleiro[linha][coluna]!=0)ENTÃO
13.               valido = falso
14.           senão
15.               valido = verdadeiro
16.          fimse
17.     enquanto(valido==falso)
18.     tabuleiro[linha][coluna] <- jogador
19. fim
```

Apesar de a função possuir 19 linhas, não é difícil entender o que faz:

- Quando a função for chamada, será enviado o número do jogador que está fazendo a jogada. Este número (1 ou 2) será atribuído ao parâmetro *jogador* da linha 1;
- As próximas linhas que pedem a jogada estão entre ***faça-enquanto*** porque precisarão ser repetidas caso a jogada seja inválida;
- As linhas de 6 a 9 pedem a linha e coluna a ser jogado;
- As linhas de 10 a 16 fazem a validação da jogada;
- Na linha 17, é verificado se o jogador terá que refazer a jogada (caso não seja válida);
- Ao sair do laço, a jogada é registrada no tabuleiro com o número do jogador que está jogando.

Este código pode ser escrito de várias outras formas. Por exemplo, a linha 16 poderia ser:

```
enquanto(valido!=verdadeiro)
```

Que é o mesmo que dizer: repetir enquanto NÃO for verdade que é válido.

Ainda, quando usamos variáveis lógicas, usualmente não precisamos nem escrever a palavra falso ou verdadeiro dentro de uma condição. Isso porque uma condição sempre espera como resultado um verdadeiro ou falso. Por exemplo, se a condição fosse:

```
enquanto(a==4)
```

O que interessa para a condição do ***enquanto*** é o resultado da comparação. Então, essa instrução está esperando o resultado falso ou verdadeiro! Sendo que verdadeiro é o valor esperado para que a condição seja satisfeita. Como variáveis lógicas já carregam o valor falso ou verdadeiro, poderíamos simplesmente escrever:

```
enquanto(!valido)
```

Lembre-se que o símbolo (!) significa NÃO.

Queremos que *válido* seja falso para repetir. Por outro lado, para repetir um bloco, a instrução ***enquanto*** espera que o resultado seja verdadeiro. Então, se o valor de *valido* for falso, e escrevemos *!valido*, estamos invertendo o resultado. *!falso* é o mesmo que dizer verdadeiro.

Outra mudança que podemos fazer é a de modularizar um pouco o código. Percebemos que nossa função realiza duas tarefas: pedir para o usuário digitar algo e validar a jogada. Não é um crime deixar como está, mas fica mais modularizado, organizado e legível se criarmos uma função para validação, como abaixo:

```
1.  logico validar(inteiro linha, inteiro coluna)
2.      se(linha>2 || coluna>2 || linha<0 || coluna<0)então
3.          retornar falso
4.      senão se(tabuleiro[linha][coluna]!=0)então
5.          retornar falso
6.      senão
7.          retornar verdadeiro
8.      fimse
9.  fim

10. nada jogada(inteiro jogador)
11.     inteiro linha, coluna
12.     lógico válido
13.     escreva("Vez do jogador ", jogador, ": ")
14.     faça
15.         escreva("Digite a linha: ")
16.         leia(linha)
17.         escreva("Digite a coluna: ")
18.         leia(coluna)
19.         valido <- validar(linha, coluna)
20.     enquanto(!valido)
21.     tabuleiro[linha][coluna] <- jogador
22. fim
```

Note que, apesar de aumentarmos um pouco o tamanho do código, a função jogada() fica mais fácil de ler. Podemos, ainda, diminuir a quantidade de linhas da

função validar, colocando as duas primeiras condições em apenas uma linha, já que ambas retornam o mesmo valor (falso):

```
1.  logico validar(inteiro linha, inteiro coluna)
2.      se(linha>2||coluna>2||linha<0||coluna<0
                    ||tabuleiro[linha][coluna]!=0)então
3.          retornar falso
4.      senão
5.          retornar verdadeiro
6.      fimse
7.  fim
```

As condições não couberam em apenas uma linha, mas considere como se a instrução ***se*** estivesse em apenas uma linha.

A lógica também poderia ser inversa! Se linha e coluna estiverem entre 0 e 2 e se o valor da posição da matriz for igual a 0, retorne verdadeiro. Do contrário, retorne falso.

Planejamento da Função jogada() Versão 2

Nesta seção, alteraremos a função para que o usuário digite uma posição de acordo com enumeração de 0 a 8 (ou 1 a 9). Como explicado anteriormente, precisamos dar um jeito de converter uma posição em duas coordenadas (linha e coluna) para podermos acessar o respectivo espaço da matriz que representa o tabuleiro.

Primeiramente, deveríamos alterar a função de apresentarTabuleiro() para que cada casa receba um número para o usuário entender o que deve escolher. Se fizermos isso, vai ficar ambíguo para os usuários 1 e 2 saberem onde já fizeram a jogada, já que existem números nas posições das casas para guiá-lo. Então, seria interessante apresentar X e O no lugar de 1 e 2, como normalmente se faz em um jogo.

Você já deve ter reparado que o planejamento inicial é importante porque pode influenciar em como devemos construir outras funções. Se quisermos alterar uma função apenas para deixá-la mais legível ou mais eficiente, sem alterar seu

resultado final, isso não vai influenciar em outras funções. Mas se quiser alterar a própria "funcionalidade", isso impactará nas demais funções que dependem dela.

Voltando à função jogada(), a maneira mais intuitiva de criá-la seria escrever um trecho de código que faça esse mapeamento, como na Figura **Erro! Fonte de referência não encontrada.**. Então, caso o usuário digite 0, a linha deve receber 0 e a coluna deve receber 0. Caso o usuário digite 1, a linha deve receber 0 e a coluna deve receber 1. É ao mesmo tempo intuitiva, mas "braçal", porque percebemos que temos que criar um caso para cada jogada. Um possível código seria como o abaixo:

```
1.  nada jogada(inteiro jogador)
2.      inteiro linha, coluna, jogada
3.      lógico válido
4.      escreva("Vez do jogador ", jogador, ": ")
5.      faça
6.          escreva("Digite uma jogada (de 0 a 8): ")
7.          leia(jogada)
8.          escolha(jogada)
9.              caso 0:
10.             linha <- 0
11.             coluna <- 0
12.         caso 1:
13.             linha <- 0
14.             coluna <- 1
15.         caso 2:
16.             linha <- 0
17.             coluna <- 2
18.         caso 3:
19.             linha <- 1
20.             coluna <- 0
21.         caso 4:
22.             linha <- 1
23.             coluna <- 1
24.         caso 5:
25.             linha <- 1
26.             coluna <- 2
27.         caso 6:
28.             linha <- 2
29.             coluna <- 0
30.         caso 7:
31.             linha <- 2
32.             coluna <- 1
33.         caso 8:
34.             linha <- 2
35.             coluna <- 2
36.     fimescolha
37.     valido <- validar(linha, coluna)
38.     ENQUANTO(valido==falso)
39. tabuleiro[linha][coluna] <- jogador
40. fim
```

O único inconveniente deste algoritmo é a quantidade de linhas. A função passou a ter 40 linhas. Não é um problema, mas isso leva a gente a pensar se não haveria uma forma mais curta de realizar o mapeamento.

Então, o primeiro passo é olhar para nosso tabuleiro e as casas (Figura **Erro! Fonte de referência não encontrada.**).

Podemos rascunhar em um papel:

Para as linhas:

- Se for a casa 0, 1 ou 2, a linha é 0;
- Se for a casa 3, 4 ou 5, a linha é 1;
- Se for a casa 6, 7 ou 8, a linha é 2.

E para colunas:

- Se for a casa 0, 3 ou 6, a coluna é 0;
- Se for a casa 1, 4 ou 7, a coluna é 1;
- Se for a casa 2, 5 ou 8, a coluna é 2.
-

Então, poderíamos melhorar nosso código para algo como:

```
1.  nada jogada(inteiro jogador)
2.      inteiro linha, coluna, jogada
3.      lógico válido
4.      escreva("Vez do jogador ", jogador, ": ")
5.      faça
6.          escreva("Digite uma jogada (de 0 a 8): ")
7.          leia(jogada)
8.          escolha(jogada)
9.              caso 0, 1, 2:
10.                 linha <- 0
11.             caso 3, 4, 5:
12.                 linha <- 1
13.             caso 6, 7, 8:
14.                 linha <- 0
15.         fimescolha
16.         escolha(jogada)
17.             caso 0, 3, 6:
18.                 coluna <- 0
19.             caso 1, 4, 7:
20.                 coluna <- 1
21.             caso 2, 5, 8:
22.                 coluna <- 2
23.         fimescolha
24.         valido <- validar(linha, coluna)
25.     enquanto(valido==falso)
26.     tabuleiro[linha][coluna] <- jogador
27. fim
```

Agora nossa função tem 27 linhas! Uma economia de 13 linhas!

Isso porque dividimos nosso *escolha-caso* em 2: um para definir linhas e outro para definir colunas, e agrupamos os casos que geram mesmos resultados.

Mas, olhando com um pouco mais de atenção aos casos, podemos chegar a uma solução ainda mais simples, se soubermos uma característica muito importante da computação: quando uma aritmética é realizada, o tipo do valor de retorno é o mesmo do maior tipo usado.

Como assim?

Se eu dividir um número inteiro por um número real, o valor calculado será do tipo real. Isso porque o conjunto dos números reais contém o conjunto dos números inteiros. Então, se:

```
inteiro a
real b
a <- 1
b <- 2
a / b
```

terá como resposta 0.5, porque o resultado é real.

E se:

```
a <- 2
b <- 2
a / b
```

terá como resposta 1.0, porque o resultado é real!!!

Mesmo que matematicamente 1 e 1.0 sejam a mesma coisa, internamente, para o computador, a representação, o espaço de memória e o próprio cálculo para chegar ao resultado podem ser diferentes entre ambos.

Até aí, você pode estar se perguntando o que isso tem a ver com jogo da velha. Não muito até agora, mas tem um detalhe que é. Se tivéssemos:

```
inteiro a, b
a <- 1
b <- 2
a / b
```

Qual será o resultado da divisão de *a* por *b*? Se por um lado, a aritmética entre tipos diferentes gera um resultado do tipo do maior operando, por outro lado,

se a aritmética for realizada entre valores de um mesmo tipo, irão gerar um resultado do mesmo tipo!!!

Por exemplo, se for realizada aritmética apenas entre valores do tipo inteiro, o resultado deverá ser um valor do tipo inteiro!!!

Portanto, se dividirmos `a` por `b` e ambos forem do tipo inteiro, como no caso anterior, o resultado deverá ser do tipo inteiro! Então, a resposta não será 0.5, mas será 0! Porque um tipo inteiro só consegue representar valores do tipo inteiro! Não há arredondamento. Se fosse 0.9, o resultado mostrado seria 0.

Como é um assunto novo, não há problema em ler novamente a explicação. Mas, uma vez entendido esse detalhe, você ainda está se perguntando: o que raios isso tem a ver com jogo da velha!!!

Repare a tabela a seguir. Nela, a variável `valor` é do tipo inteiro. Na segunda coluna, estamos dividindo o valor por 3, que é um número inteiro! Para representar como número decimal, deveríamos representar como 3.0 ou 3.f. Portanto, o resultado precisará ser um tipo inteiro!!!

Valor	Valor/3	Resultado se fosse na matemática básica	Resultado da divisão de um inteiro por um inteiro
0	0/3	0	0
1	1/3	0.33333	0
2	2/3	0.66666	0
3	3/3	1	1
4	4/3	1.33333	1
5	5/3	1.66666	1
6	6/3	2	2
7	7/3	2.33333	2
8	8/3	2.66666	2

Reparou que os números 0, 1 e 2 geram como resultado 0. E que 3, 4 e 5 geram como resultado 1. E que 6, 7 e 8 geram 2? Então, fazendo um cálculo, descobrimos como encontrar a respectiva linha da matriz! Se o usuário digitar a casa 8, por exemplo, e dividirmos 8 por 3, o resultado será 2, que é a linha onde a casa 8 fica!

Agora, falta descobrir a coluna! E já sabemos! Lembra do exercício para descobrir se um número é par ou ímpar? Usamos o símbolo %, que representa o resto da divisão de um número por outro.

Então, se fizermos a operação

0%3, o resultado será 0, porque 0 dividido por 3 dá 0 e resta 0.

1%3, o resultado será 1, porque 1 dividido por 3 dá 0 e resta 1.

2%3, o resultado será 2, porque 2 dividido por 3 dá 0 e resta 2.

3%3, o resultado será 0, porque 3 dividido por 3 dá 1 e resta 0.

4%3, o resultado será 1, porque 4 dividido por 3 dá 1 e resta 1.

5%3, o resultado será 2, porque 5 dividido por 3 dá 1 e resta 2.

E assim por diante. Colocando em uma tabela:

Valor	Valor%3	Resultado se fosse na matemática básica	Resto
0	0/3	0	0
1	1/3	0.33333	1
2	2/3	0.66666	2
3	3/3	1	0
4	4/3	1.33333	1
5	5/3	1.66666	2
6	6/3	2	0
7	7/3	2.33333	1
8	8/3	2.66666	2

E aí está a mágica! Os valores 0, 3 e 6 geram 0 como resto!. Os valores 1, 4 e 7 geram 1 como resto. E os valores 2, 5 e 8 geram 2 como resto. Que é exatamente o que estávamos querendo para encontrar a respectiva coluna em relação à casa que o jogador digitou.

Agora, podemos alterar nossa função para encontrar a linha e a coluna via cálculo, com base na casa que o usuário digitar. Em resumo:

Para encontrar a linha, basta dividir a casa por 3 (sendo ambos números inteiros);

Para encontrar a coluna, basta encontrar o resto da divisão da casa por 3.

E o código ficará bem mais enxuto:

```
1. nada jogada(inteiro jogador)
2.     inteiro linha, coluna, jogada
3.     lógico válido
4.     escreva(“Vez do jogador “, jogador, “: “)
5.     faça
6.         escreva(“Digite uma jogada (de 0 a 8): “)
7.         leia(jogada)
8.         linha <- jogada / 3
9.         coluna <- jogada % 3
10.        valido <- validar(linha, coluna)
11.    enquanto(valido==falso)
12.    tabuleiro[linha][coluna] <- jogador
13. fim
```

Se formos comparar, nossa solução inicial tinha 40 linhas. E conseguimos reduzir para 13. Um código mais enxuto não significa necessariamente um código mais fácil de entender. O código inicial era mais intuitivo. Para entender esta última solução, precisamos lembrar (ou aprender) conceitos que às vezes são pouco utilizados.

Mas esse é um excelente exemplo de uma função que, após escrita, olhamos para ela e ficamos pensando se não dá para reduzir. Isso acontecerá muito em nosso dia a dia. A sugestão durante o estudo de Algoritmo é: faça como vem à cabeça,

passe um tempo tentando descobrir soluções melhores por conta própria. Só depois de não haver mais como melhorar, vá procurar por outras soluções na internet ou com grupos de estudo.

Isso porque se esforçando e passando tempo para melhorar o código facilita o entendimento e fixação de outras soluções que você não pensou.

E se eu quisesse que o usuário digitasse valores de 1 a 9, em vez de 0 a 8? Fácil. Basta subtrair 1 do valor digitado. Por exemplo, se a primeira casa passa a ser a casa 1 (em vez de 0), então pegue este 1 e subtraia 1. O resultado será 0. A partir dele, faça os dois cálculos.

Planejando a função vezDeQuem()

Pela ordem do bloco principal do nosso algoritmo, deveríamos planejar a condição de vitória. Mas vamos planejar esta função primeiro por ser bem simples. Esta função precisa alternar o jogador. Então, se quem está jogando for o jogador 1, a função deve retornar o número 2 e vice-versa.

```
1.  inteiro vezDeQuem(inteiro jogador)
2.      se(jogador==1)então
3.          retornar 2
4.      senão
5.          retornar 1
6.      fimse
7.  fim
```

Bastante simples!

Agora vamos para a função de verificação de vitória.

Planejando a função verificarVitoria() Versão 1

Novamente, começaremos nossa primeira versão pelo mais intuitivo. Em um jogo no papel, como fazemos para verificar a condição de vitória? Verificamos se há alguma linha, alguma coluna ou alguma diagonal apenas com X ou com O.

Agora, vamos adaptar esta ideia para um algoritmo. Precisaremos criar um código que verifique se existe alguma linha, coluna ou diagonal apenas com números 1 ou números 2 (porque é o que registramos em nossa matriz quando os respectivos jogadores realizam suas jogadas, no lugar de X ou O). Havendo pelo menos uma destas situações, a função deverá mostrar o ganhador (para tanto, é necessário enviar para esta função, qual o jogador da vez) e retornar verdadeiro (alguém ganhou). Do contrário, retornará falso.

Como existem apenas 3 linhas, 3 colunas e 2 diagonais, podemos criar várias condições que verifiquem o que precisamos. Vamos começar pela verificação das linhas. Um rascunho que transcreva nossa ideia literalmente para o algoritmo seria:

```
lógico verificarVitoria(inteiro jogador)
    se(tabuleiro[0][0]==1 && tabuleiro[0][1]==1 && tabuleiro[0][2]==1 ||
       tabuleiro[0][0]==2 && tabuleiro[0][1]==2 && tabuleiro[0][2]==2 ||
       tabuleiro[1][0]==1 && tabuleiro[1][1]==1 && tabuleiro[1][2]==1 ||
       tabuleiro[1][0]==2 && tabuleiro[1][1]==2 && tabuleiro[1][2]==2 ||
       tabuleiro[2][0]==1 && tabuleiro[2][1]==1 && tabuleiro[2][2]==1 ||
       tabuleiro[2][0]==2 && tabuleiro[2][1]==2 && tabuleiro[2][2]==2) então
        escreva("Jogador ", jogador, " venceu!!!")
        retornar verdadeiro
    fimse
    retornar falso
fim
```

Este trecho acima faz a verificação apenas das linhas. A condição ***se*** foi quebrada em várias linhas para facilitar a visualização. Agora seria necessário fazer o mesmo para cada coluna e para as diagonais. Mas, vamos focar apenas nas linhas por enquanto.

Apesar de a lógica estar correta, dá para notar que nossa instrução ***se*** está muito grande. Então, este é um bom ponto para pensarmos se não podemos diminuir a quantidade de comparações com algum recurso lógico.

Por exemplo, da forma como transcrevemos nossa ideia inicial, verificamos se cada linha tem todas as casas como 1 ou 2. Mas é possível colocar de outra forma: verificar para cada linha se a primeira casa é igual à segunda casa; se a segunda casa é igual à terceira casa e se uma destas casas é 1 ou 2 (ou se uma destas casas é diferente de 0). Por que só precisamos verificar se uma casa é diferente de 0? Porque acabamos de verificar se elas são iguais!

É o mesmo princípio, mas que gera uma escrita mais enxuta. Então, aquele mesmo código é equivalente ao descrito a seguir:

```
1.  se(tabuleiro[0][0]==tabuleiro[0][1]&&tabuleiro[0][1]==tabuleiro[0][2]&&
       tabuleiro[0][0]!=0 ||
       tabuleiro[1][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[1][2]&&
       tabuleiro[1][0]!=0 ||
       tabuleiro[2][0]==tabuleiro[2][1]&&tabuleiro[2][1]==tabuleiro[2][2]&&
       tabuleiro[2][0]!=0) então
2.        escreva("Jogador ", jogador, " venceu!!!")
3.        retornar verdadeiro
4.  fimse
```

A condição é muito grande e precisou ser quebrada. Essa linha 1 do código verifica se a primeira célula é igual à segunda e se a segunda é igual à terceira, e se é diferente de 0, em cada linha da matriz (separadas pelo operador ou ||).

Neste ponto, é possível notar que as análises das linhas da tabela são praticamente iguais, variando apenas o primeiro índice (0, 1 e 2). Quando temos linhas idênticas, que apenas variam em algum número, podemos usar um laço para!

```
1.  para (i <- 0; i < 3;i++)faça
2.     se(tabuleiro[i][0]==tabuleiro[i][1]&&tabuleiro[i][1]==tabuleiro[i][2]
          && tabuleiro[i][0]!=0)então
3.            escreva("Jogador ", jogador, " venceu!!!")
4.            retornar verdadeiro
5.     fimse
6.  fimpara
```

O ***para*** verificará, a cada laço, se a respectiva linha *i* possui jogadas iguais!

Agora, precisamos verificar as colunas e diagonais. Vamos focar primeiro nas colunas. A ideia é exatamente a mesma do que fizemos com as linhas, mas precisamos apenas inverter os índices. A comparação entre apenas as colunas seria:

```
 se(tabuleiro[0][0]==tabuleiro[1][0] && tabuleiro[1][0]==tabuleiro[2][0] &&
     tabuleiro[0][0]!=0 ||
     tabuleiro[0][1]==tabuleiro[1][1] && tabuleiro[1][1]==tabuleiro[2][1] &&
     tabuleiro[0][1]!=0 ||
     tabuleiro[0][2]==tabuleiro[1][2] && tabuleiro[1][2]==tabuleiro[2][2] &&
     tabuleiro[0][2]!=0) então
     escreva("Jogador ", jogador, " venceu!!!")
     retornar verdadeiro
fimse
```

Repare que trocamos a ordem entre os índices porque precisamos verificar colunas!

Substituindo as três linhas de comparação por um laço *para*, teremos:

```
para (i <- 0; i < 3; i++)faça
    se(tabuleiro[0][i]==tabuleiro[1][i]&&tabuleiro[1][i]==tabuleiro[2][i]&&
       tabuleiro[0][i]!=0)então
           escreva("Jogador ", jogador, " venceu!!!")
           retornar verdadeiro
    fimse
fimpara
```

Temos agora dois laços *para*. Um que verifica linhas e outro que verifica colunas.

Se observarmos os dois laços, veremos que eles têm o mesmo cabeçalho! O valor inicial do índice é o mesmo, a condição é a mesma e o incremento é o mesmo! Isso é um sinal de que podemos juntar os dois laços em apenas um, como abaixo:

```
para (i<- 0;i<3;i++)faça
    se(tabuleiro[i][0]==tabuleiro[i][1]&&tabuleiro[i][1]==tabuleiro[i][2]&&
       tabuleiro[i][0]!=0 ||
       tabuleiro[0][i]==tabuleiro[1][i]&&tabuleiro[1][i]==tabuleiro[2][i]&&
       tabuleiro[0][i]!=0)então
           escreva("Jogador ", jogador, " venceu!!!")
           retornar verdadeiro
    fimse
fimpara
```

Pronto! Melhoramos nosso código mais um pouco!

Agora falta a verificação das diagonais:

```
se (tabuleiro[0][0]==tabuleiro[1][1] && tabuleiro[1][1]==tabuleiro[2][2] &&
    tabuleiro[1][1]!=0 ||
    tabuleiro[2][0]==tabuleiro[1][1] && tabuleiro[1][1]==tabuleiro[0][2] &&
    tabuleiro[1][1]!=0)então
    escreva("Jogador ", jogador, " venceu!!!")
    retornar verdadeiro
fimse
```

Legal! Fizemos todas as verificações para determinar se alguém ganhou. Provisoriamente, nossa função está no seguinte estado:

```
lógico verificarVitoria(inteiro jogador)
  inteiro i
  para (i< - 0; i < 3; i++)faça
    se(tabuleiro[i][0]==tabuleiro[i][1]&&tabuleiro[i][1]==tabuleiro[i][2]&&
       tabuleiro[i][0]!=0 ||
       tabuleiro[0][i]==tabuleiro[1][i]&&tabuleiro[1][i]==tabuleiro[2][i]&&
       tabuleiro[0][i]!=0)então
           escreva("Jogador ", jogador, " venceu!!!")
           retornar verdadeiro
    fimse
  fimpara

  se(tabuleiro[0][0]==tabuleiro[1][1] && tabuleiro[1][1]==tabuleiro[2][2] &&
     tabuleiro[1][1]!=0||
     tabuleiro[2][0]==tabuleiro[1][1] && tabuleiro[1][1]==tabuleiro[0][2] &&
     tabuleiro[1][1]!=0)então
         escreva("Jogador ", jogador, " venceu!!!")
         retornar verdadeiro
  fimse

  retornar falso
fim
```

Vamos lembrar um detalhe importante: a função deve retornar verdadeiro ou falso. Se um jogador ganhar, a função está retornando verdadeiro. Caso ninguém tenha ganhado, a função retorna falso.

Esse valor retornado é importante porque é ele que determinará se o jogo deve parar ou se deve continuar.

Mas há um problema. E se os jogadores já jogaram as nove casas do tabuleiro e ninguém ganhou? Do jeito que está, o jogo continuará eternamente a pedir para o próximo jogador fazer sua jogada. O pior, é que, uma vez que houve empate e todas as casas estão ocupadas, a partir deste ponto nosso algoritmo sempre dirá que a jogada é inválida, porque com certeza o jogador terá que escolher uma casa que já foi marcada.

Como resolvemos isso? Se houve empate, então, a função precisa avisar aos jogadores sobre o empate, mas também precisa retornar verdadeiro, para que o jogo termine. Então, precisamos escrever um trecho de código que verifique se houve empate.

Para isso, precisamos percorrer as casas do tabuleiro para verificar se não existe mais espaço vazio. Uma forma de fazer isso é:

```
    logico temzero
```

```
    inteiro i, j
    temzero <- falso
    para(i <- 0; i < 3; i++)faça
        para(j <- 0; j < 3; j++)faça
            se(tabuleiro[i][j]==0)então
                temzero<-1;
            fimse
        fimpara
    fimpara

    se(temzero){
        retornar falso
    senão
        escreva("Deu empate!")
        retornar verdadeiro
    fimse
```

No código anterior, criamos uma variável lógica (*temzero*) que representa se existe pelo menos uma casa com o número 0 (pelo menos uma casa sem jogada). A ideia é: se tem pelo menos uma casa sem jogada, esta variável receberá o valor verdadeiro (isto é, é verdade que tem pelo menos uma casa com 0). Então, o valor inicial dela será falso.

Os laços ***para*** aninhados percorrem o tabuleiro. E se for detectada pelo menos uma casa com zero, a variável *temzero* receberá verdadeiro.

No fim dos laços aninhados, se *temzero* for verdadeiro, é que ainda tem jogada a ser realizada, portanto não houve empate e o jogo deve continuar (por isso a condição seguinte retornará falso).

Mas se *temzero* for falso, é porque todas as casas estão ocupadas, portanto houve empate e o jogo deve parar (por isso a condição seguinte retornará verdadeiro.

O código completo de nossa primeira versão está pronto!

```
lógico verificarVitoria(inteiro jogador)
   inteiro i, j
   logico temzero

   //Verificando linhas e colunas
   para (i <- 0; i < 3; i++)faça
     se(tabuleiro[i][0]==tabuleiro[i][1]&&tabuleiro[i][1]==tabuleiro[i][2]&&
        tabuleiro[i][0]!=0 ||
        tabuleiro[0][i]==tabuleiro[1][i]&&tabuleiro[1][i]==tabuleiro[2][i]&&
        tabuleiro[0][i]!=0)então
            escreva("Jogador ", jogador, " venceu!!!")
            retornar verdadeiro
     fimse
   fimpara
```

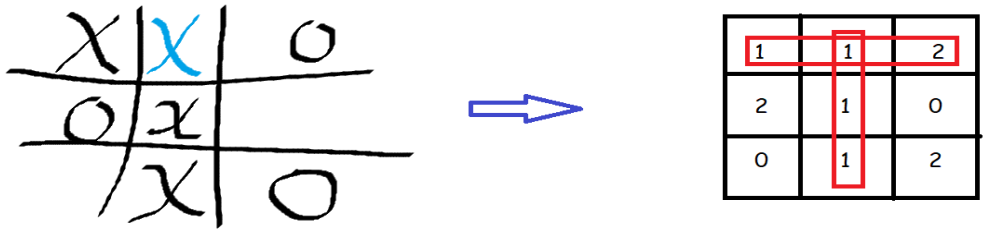
```
    //Verificando as diagonais
    se(tabuleiro[0][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[2][2]&&
       tabuleiro[1][1]!=0 ||
       tabuleiro[2][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[0][2]&&
       tabuleiro[1][1]!=0)então
          escreva("Jogador ", jogador, " venceu!!!")
          retornar verdadeiro
    fimse

    //Verificando se houve empate
    temzero < - falso
    para(i <- 0; i < 3 ; i++)faça
        para(j <- 0; j<3 ; j++)faça
            se(tabuleiro[i][j]==0)então
                temzero <- 1
            fimse
        fimpara
    fimpara

    se(temzero)então
        retornar falso
    senão
        escreva("Deu empate!")
        retornar verdadeiro
    fimse
fim
```

Planejando a função verificarVitoria() Versão 2

Esta próxima versão é fortemente baseada na anterior.

A diferença é que, em vez de verificarmos todas as possibilidades de vitória, verificaremos apenas aquelas baseadas na última jogada. Observe o tabuleiro da Figura 32.

Figura 32 - Reduzindo verificações de vitória



Supondo que o X em azul seja a última jogada realizada, então, neste caso, basta verificar se a linha em que a jogada foi feita ou se a coluna em que a jogada

foi feita acarretará uma vitória. Se a jogada fosse feita em um canto, então sua respectiva diagonal também precisaria ser verificada. Mas vamos focar apenas na linha e coluna da jogada.

Repare que vamos melhorar o código porque poderemos eliminar o laço ***para*** da versão anterior e verificar apenas a linha e coluna da jogada, e não todas as linhas e colunas. Um primeiro rascunho da alteração que faremos na condição ***se*** da versão anterior segue abaixo:

```
se(tabuleiro[linha][0] == tabuleiro[linha][1] &&
   tabuleiro[linha][1] == tabuleiro[linha][2] &&
   tabuleiro[linha][0] != 0 ||
   tabuleiro[0][coluna] == tabuleiro[1][coluna] &&
   tabuleiro[1][coluna] == tabuleiro[2][coluna] &&
   tabuleiro[0][coluna] !=0)então
      escreva("Jogador ", jogador, " venceu!!!")
      retornar verdadeiro
fimse
```

Não se engane. Apesar desta instrução ***se*** parecer maior que a anterior, é apenas porque trocamos a variável `i` por `linha` ou `coluna`, que são palavras maiores!

Para que este código funcione, precisamos alterar o código da função jogada(), para que as variáveis `linha` e `coluna` sejam globais para que, depois da jogada, a função verificarVitoria() consiga acessar os valores lá definidos. Também poderíamos mantê-las como locais, mas teríamos que alterar o código para que haja uma função de leitura da linha e uma função para leitura da coluna, para que cada função retorne um valor (já que uma função não pode retornar mais de um valor ao mesmo tempo) o que não foi feito. Também poderíamos retornar um vetor com os valores linha e coluna (mas nem todas as linguagens permitem o retorno de vetores de forma simples, com o que aprendemos até o momento, como C).

Por outro lado, se quisermos apenas usar variáveis locais, a segunda versão da função jogada(), na qual usamos um valor de 0 a 8 para escolhermos a casa do tabuleiro se adequaria melhor a essa nossa versão de verificarVitoria() porque bastaria que ela fosse retornada.

Como o objetivo aqui é mostrar a alteração da função verificarVitoria(), consideraremos que as variáveis `linha` e `coluna` estão declaradas como globais. No fim do capítulo, apresentaremos a versão completa com as modificações necessárias.

Voltando às melhorias na segunda versão, também não precisamos verificar as diagonais, se já tivermos encontrado um vencedor na primeira verificação de linha e coluna. Então, podermos juntar os dois ***se*** separados em um ***se-senão*** encadeado. E só precisamos verificar o empate, se não houve vitória.

Segue a versão 2 completa:

```
lógico verificarVitoria(inteiro jogador)
   inteiro i, j
   logico temzero

   //Verificando linhas e colunas
   se(tabuleiro[linha][0] == tabuleiro[linha][1] &&
      tabuleiro[linha][1] == tabuleiro[linha][2] &&
      tabuleiro[linha][0]!=0 ||
      tabuleiro[0][coluna] == tabuleiro[1][coluna] &&
      tabuleiro[1][coluna] == tabuleiro[2][coluna] &&
      tabuleiro[0][coluna]!=0)então
         escreva("Jogador ", jogador, " venceu!!!")
         retornar verdadeiro
   senão se(tabuleiro[0][0] == tabuleiro[1][1] &&
            tabuleiro[1][1] == tabuleiro[2][2] &&
            tabuleiro[1][1] != 0 ||
            tabuleiro[2][0] == tabuleiro[1][1] &&
            tabuleiro[1][1] == tabuleiro[0][2] &&
            tabuleiro[1][1]!=0)então
                escreva("Jogador ", jogador, " venceu!!!")
                retornar verdadeiro
   senão
                //Verificando se houve empate
                temzero < - falso

                para(i <- 0; i < 3; i++)faça
                    para(j <- 0; j < 3 ;j++)faça
                        se(tabuleiro[i][j]==0)então
                            temzero=1;
                        fimse
                    fimpara
                fimpara

                se(temzero)então
                    retornar falso
                senão
                    escreva("Deu empate!")
                    retornar verdadeiro
                fimse
    fimse
fim
```

Basicamente melhoramos o código eliminando excesso de verificações desnecessárias.

Planejando a função verificarVitoria() Versão 3

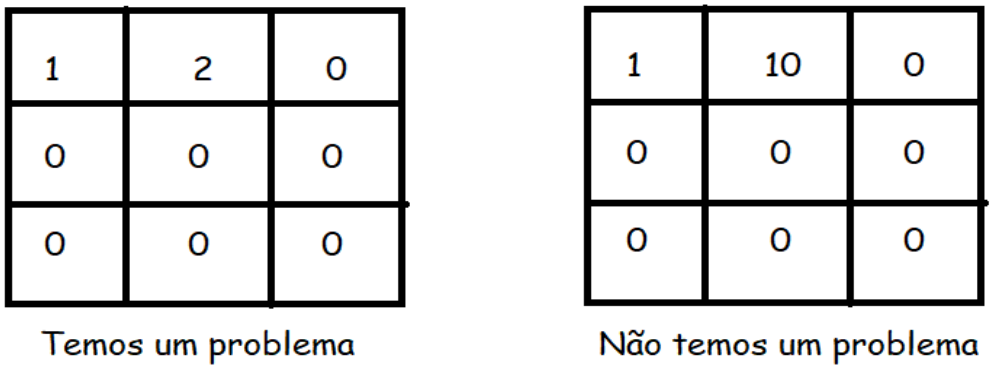
Ainda com o objetivo de explorar diferentes soluções e diminuir a quantidade de comparações, podemos mudar um pouco a forma como analisamos as condições de vitória. Pense da seguinte forma: em vez de atribuirmos 1 ou 2 na matriz, podemos atribuir valores como 1 (quando o jogador 1 fizer sua jogada) e 10 (quando for o jogador 2).

Vamos supor que estamos analisando apenas a primeira linha do tabuleiro. O que fizemos nas versões anteriores foi verificarmos se a primeira casa é igual à segunda e se a segunda é igual à terceira e se elas são diferentes de 0.

Em vez disso, podemos somar o valor da primeira casa com o da segunda e com o da terceira. Se o resultado for 3, é porque o jogador 1 venceu. Se o resultado for 30, é porque o jogador 2 venceu. Qualquer outro resultado, implica que ninguém venceu.

Para garantirmos que esta estratégia funcione, temos que relacionar valores aos jogadores, de forma que sejam bem distantes um do outro para que não haja ambiguidade. Por exemplo, se mantivéssemos os valores 1 e 2, poderíamos ter uma situação em que a soma da linha daria 3, mas isso não significa vitória do jogador 1.

Figura 33 - Cuidado com os valores para não dar vitória falsa



No exemplo de jogada do tabuleiro da esquerda na Figura 33, a soma das jogadas na primeira linha é 3, o que será um problema se considerarmos que a condição de vitória para o jogador 1 é que a soma dos valores das casas seja 3.

Eliminamos este problema criando valores distantes, como no exemplo da direita da Figura **Erro! Fonte de referência não encontrada.**. Também funcionaria se escolhêssemos um valor positivo para o jogador 1 e um mesmo valor, mas negativo, para o jogador 2.

Supondo que alteremos o código da função jogada(), para que valores distantes sejam atribuídos ao tabuleiro, conforme o jogador da vez, uma possível solução é apresentada a seguir:

```
inteiro verificarVitoria(inteiro jogador)
   inteiro i, j, temzero

   se(tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2]==3 ||
      tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna]==3 ||
      tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2]==30 ||
      tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna]==30)
      então
          mostrarGanhador(jogador)
          retornar 1
      senão se(tabuleiro[0][0]+tabuleiro[1][1]+tabuleiro[2][2]==3 ||
               tabuleiro[2][0]+tabuleiro[1][1]+tabuleiro[0][2]== 3 ||
               tabuleiro[0][0]+tabuleiro[1][1]+tabuleiro[2][2]== 30 ||
               tabuleiro[2][0]+tabuleiro[1][1]+tabuleiro[0][2]==30)então
          mostrarGanhador(jogador)
         retornar 1
     senão
         temzero <- 0;
         para(i <- 0; i < 3; i++)faça
             para(j <- 0; j < 3; j++)faça
                 se(tabuleiro[i][j]==0)
                     temzero <- 1
                 fimse
             fimpara
         fimpara

         se(temzero==1)
             retornar 0
         senão
             mostrarGanhador(0)
             retornar 1
         fimse
     fimse
fim
```

O inconveniente desta solução, é que a condição de vitória para o jogador 1 é diferente da condição de vitória para o jogador 2. Na versão anterior, as

condições eram as mesmas: bastava que três casas horizontais ou verticais ou diagonais fossem iguais, e ao mesmo tempo não fossem 0.

Nesta nova versão, as condições são diferentes porque para o jogador 1 ganhar, a soma das casas precisa ser 3 e para o jogador 2 ganhar, a soma das casas precisa ser 30. Isso nos fez duplicar a quantidade de comparações (para verificar tanto o jogador 1 quanto o 2).

Este inconveniente faz a gente pensar se não há uma maneira de encontrar uma condição para ambos. E há! Se usarmos os valores 1 e -1 para os respectivos jogadores. Neste caso, a condição de vitória para o jogador 1 continua sendo a soma 3, e a condição de vitória do jogador 2 passa a ser soma -3. E podemos tentar fazer a comparação com o número absoluto deste resultado.

Como não queremos depender de instruções prontas e específicas de uma linguagem, vamos ter que pensar como chegar a um valor igual sem usar uma instrução pronta que retorne um valor absoluto. Para isso é só lembrarmos de uma regra básica da aritmética: quando multiplicamos dois valores positivos o resultado é positivo; e quando multiplicamos dois valores negativos, o resultado também é positivo! No exemplo acima, se o resultado for 3, sabemos que 3 vezes 3 é 9. E se o resultado for -3, sabemos que -3 vezes -3 também será 9!

Então, primeiro fazemos as somas das casas (na linha e coluna da jogada, e nas diagonais principal e secundária). Depois verificamos se ele multiplicado por ele mesmo é igual à 9!

```
inteiro verificarVitoria(int jogador)
   inteiro i, j, temzero, somaL, somaC, somaDiagP, somaDiagS;

   somaL <- tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2]
   somaC <- tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna]
   somaDiagP <- tabuleiro[0][0]+tabuleiro[1][1] + tabuleiro[2][2]
   somaDiagS <- tabuleiro[2][0] + tabuleiro[1][1] + tabuleiro[0][2]

   se(somaL*somaL == 9 || somaC*somaC == 9|| somaDiagP*somaDiagP == 9 ||
      somaDiagS*somaDiagS == 9)então
         mostrarGanhador(jogador)
         retornar 1
   senão
         temzero = 0;
         para(i <- 0; i < 3 ; i++)faça
             para(j <- 0; j < 3; j++)faça
                 se(tabuleiro[i][j]==0)então
                     temzero <- 1
                 fimse
             fimpara
```

```
        fimpara
        se(temzero==1)então
            retornar 0
        senão
            mostrarGanhador(0)
            retornar 1
        fimse
   fimse
fim
```

O legal desta solução é que as nossas condições de vitórias se tornam simples o suficiente para colocarmos todas em uma mesma instrução ***se***. A partir deste código, poderíamos fazes várias alterações para chegarmos ao mesmo resultado. E aí está a graça de Algoritmos!

E qual a melhor versão? Todas. A que chegarmos. Em termos de complexidade computacional dos códigos, a diferença não é relevante. Em termos de leitura de código, há quem prefira a segunda versão por ser intuitiva e enxuta (em relação à primeira versão) e há quem prefira a terceira versão por ter a condição ***se*** mais enxuta (em relação às duas versões anteriores).

Planejando a função verificarVitoria() Versão 4, 5, 6...

Existem muitas outras soluções possíveis. Muitas delas exigem um pouco mais de conhecimento em Algoritmos, como com diferentes formas de representar o tabuleiro ou as jogadas, ou com uso de estruturas de dados.

Por exemplo, em vez de representarmos o tabuleiro com uma matriz, poderíamos representá-lo com 9 bits. Cada bit representa uma casa, na ordem da primeira para a última linha. Ao mesmo tempo, as jogadas de cada jogador também poderiam ser representadas com 9 bits:

000 000 000

Os três primeiros bits representam as três casas da primeira linha, as três próximas representam as casas da segunda linha e as três últimas representam as casas da última linha. Ainda, cada jogador possuirá sua representação de 9 bits de suas jogadas.

Com essa nova representação, novas portas se abrem! Uma solução interessante seria com operações bit a bit (*bitwise*). Operadores comumente usados são & e | (E e OU bit a bit). Estes operadores funcionam da seguinte maneira: imagine que temos dois números binários, como 001010110 e 001000101. As operações & e | (E e OU bit-a-bit) entre estes valores serão:

```
001 010 110 &
011 000 101
-----------
001 000 100
```

```
001 010 110 |
011 000 101
-----------
011 010 111
```

Analisando a primeira operação (E bit-a-bit), faremos operações E para cada unidade (cada bit) do número de cima com a respectiva unidade (respectivo bit) do número de baixo. Lembrando que a tabela verdade para operação E é como abaixo:

0 & 0 = 0 (0 e 0 é igual a 0)

0 & 1 = 0 (0 e 1 é igual a 0)

1 & 0 = 0 (1 e 0 é igual a 0)

1 & 1 = 0 (1 e 1 é igual a 1)

Ressaltamos, no exemplo, apenas três operações bit-a-bit para exemplificar, mas as operações serão realizadas entre todas elas. A primeira operação dentro do retângulo vermelho mostra o resultado de uma operação E bit-a-bit do 0 (número de cima) com o 0 (número de baixo). E o resultado é 0, conforme a tabela verdade. A segunda operação de exemplo, mostra o resultado de 1 & 1, que é 1, conforme a tabela verdade.

A operação OU bit-a-bit é similar, mas usando-se a tabela verdade OU, conforme abaixo:

0 | 0 = 0 (0 ou 0 é igual a 0)

0 | 1 = 1 (0 ou 0 é igual a 1)

1 | 0 = 1 (0 ou 0 é igual a 1)

1 | 1 = 1 (0 ou 0 é igual a 1)

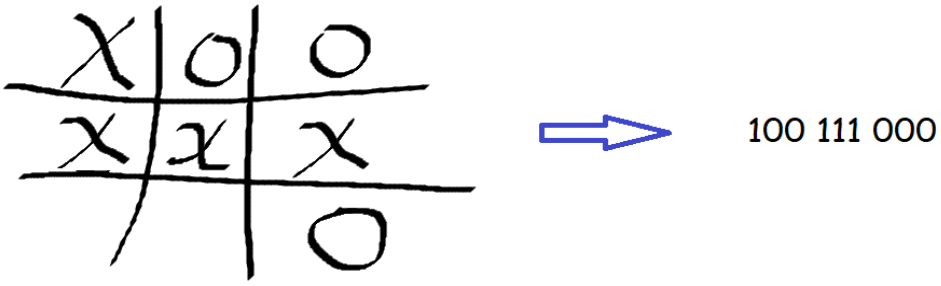
E o que operações bit-a-bit têm a ver com encontrar o resultado de uma jogada? Para usarmos estes conhecimentos para descobrir se houve vitória, além de representarmos as jogadas com 9 bits, criaremos um vetor com todas as soluções possíveis:

```
condição_de_vitória = [ 111 000 000,        ← Primeira linha
                        000 111 000,        ← Segunda linha
                        000 000 111,        ← Terceira linha
                        100 100 100,        ← Primeira coluna
                        010 010 010,        ← Segunda coluna
                        001 001 001,        ← Terceira coluna
                        100 010 001,        ← Diagonal principal
                        001  010 100]       ← Diagonal secundária
```

Por fim, precisamos fazer uma operação E bit-a-bit para verificar se o jogador 1 ganhou e outra para verificarmos se o jogador 2 ganhou.

Para exemplificar, vamos supor o seguinte cenário de vitória do jogador 1 da Figura 34.

Figura 34 - Tabuleiro representado com 9 bits



O número binário à direita representa as jogadas do jogador 1 no tabuleiro.

Agora, faremos a operação E bit-a-bit das jogadas do jogador 1 com cada uma das condições de vitória. Se pelo menos uma das operações der como resultado um valor exatamente igual à própria condição de vitória, então o jogador 1 venceu.

Vamos mostrar passo a passo. Na tabela abaixo, a primeira linha representa cada condição de vitória, a segunda linha representa as jogadas do jogador 1 e a terceira linha representa o resultado da operação E bit-a-bit:

Condições de vitória:

111000000	000111000	000000111	100100100	010010010	001001001	100010001	001010100
100111000	100111000	100111000	100111000	100111000	100111000	100111000	100111000
100000000	000111000	000000000	100100000	000010000	000001000	100010000	000010000

Olha que sensacional! Vimos que, na segunda coluna, o resultado da operação é exatamente igual à condição de vitória! Assim, conseguimos identificar que o jogador 1 ganhou.

O mesmo procedimento deve ser feito para o jogador 2, mas com a representação de suas jogadas.

Além desta versão, também é possível criar códigos menores e mais fáceis de entender com o uso de funções e operações que sejam específicas de alguma linguagem de programação.

Este capítulo serviu para mostrar a vocês aquilo que tenho explicado ao longo deste livro: existem várias soluções possíveis para um problema. E para cada solução, podemos sempre pensar em melhorar o código. Quando nos deparamos com um problema novo, não há necessidade de nos cobrarmos que a melhor solução venha à nossa cabeça. Podemos começar com o que parece mais intuitivo e trabalhar em cima dessas ideias.

Se fizermos isso por diversão, estaremos também treinando. E isso certamente nos ajudará em nossa profissão, seja por meio de intuição ao nos depararmos com novos problemas, seja por estarmos acostumados a pensar por meio de Algoritmo.

Como é em algumas linguagens

Português Estruturado

As identações foram adaptadas (maior ou menor recuo) para que as linhas não sejam quebradas.

```
Versão 1

inteiro tabuleiro[3][3]

nada apresentarTabuleiro()
  escreva("  0     1     2")
  escreva(" ")
  escreva("0 ",tabuleiro[0][0]," | ",tabuleiro[0][1]," | ",tabuleiro[0][2])
  escreva("  -----------------")
  escreva("1 ",tabuleiro[1][0]," | ",tabuleiro[1][1]," | ",tabuleiro[1][2])
  escreva("  -----------------")
  escreva("2 ",tabuleiro[2][0]," | ",tabuleiro[2][1]," | ",tabuleiro[2][2])
fim

logico validar(inteiro linha, inteiro coluna)
 se(linha>2||coluna>2||linha<0||coluna<0||tabuleiro[linha][coluna]!=0)então
     retornar falso
 senão
     retornar verdadeiro
 fimse
fim

nada jogada(inteiro jogador)
    inteiro linha, coluna
    lógico válido
    escreva(“Vez do jogador “, jogador, “: “)
    faça
        escreva(“Digite a linha: “)
        leia(linha)
        escreva(“Digite a coluna: “)
        leia(coluna)
        valido <- validar(linha, coluna)
    enquanto(valido==falso)
    tabuleiro[linha][coluna] = jogador
fim

inteiro vezDeQuem(inteiro jogador)
    se(jogador==1)então
        retornar 2
    senão
        retornar 1
    fimse
fim


nada mostrarGanhador(inteiro ganhador)
    se(ganhador==0)então
        escreva(“Empatou!!!”)
    senão
        escreva(“O jogador “, ganhador , “ venceu!!!”)
    fimse
    apresentarTabuleiro()
```

```
fim

lógico verificarVitoria(inteiro jogador)
  inteiro i, j
  logico temzero

  //Verificando linhas e colunas
  para(i <- 0; i < 3; i++)faça
    se(tabuleiro[i][0]==tabuleiro[i][1]&&tabuleiro[i][1]==tabuleiro[i][2]&&
       tabuleiro[i][0]!=0 ||
       tabuleiro[0][i]==tabuleiro[1][i]&&tabuleiro[1][i]==tabuleiro[2][i]&&
       tabuleiro[0][i]!=0)então
         mostrarGanhador(jogador)
         retornar verdadeiro
    fimse
  fimpara

  //Verificando as diagonais
  se(tabuleiro[0][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[2][2]&&
     tabuleiro[1][1]!=0 ||
     tabuleiro[2][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[0][2]&&
     tabuleiro[1][1]!=0)então
       mostrarGanhador(jogador)
       retornar verdadeiro
  fimse

  //Verificando se houve empate
  temzero < - falso
  para(i <- 0; i < 3 ;i++)faça
    para(j <- 0; j < 3; j++)faça
      se(tabuleiro[i][j]==0)então
        temzero <- 1;
      fimse
    fimpara
  fimpara

  se(temzero)então
    retornar falso
  senão
    mostrarGanhador(0)
    retornar verdadeiro
  fimse
fim

início
    logico vitoria
    inteiro jogador
    jogador <- 1
    vitoria <- falso
    faça
        apresentarTabuleiro()
        jogada(jogador)
        vitoria <- verificarVitoria(jogador)
        jogador < - vezDeQuem(jogador)
    enquanto(vitoria==falso)
fim
```

Versão 2

```
inteiro tabuleiro[3][3]

nada apresentarTabuleiro()
  escreva(“Número das casas do tabuleiro”)
  escreva(“  0 | 1 | 2”)
  escreva("  ---------")
  escreva(“  3 | 4 | 5”)
  escreva("  ---------")
  escreva(“  6 | 7 | 8”)

  escreva(“Jogadas Realizadas:”)
  escreva(tabuleiro[0][0], " | ",tabuleiro[0][1]," | ",tabuleiro[0][2])
  escreva("  ---------")
  escreva( tabuleiro[1][0]," | ",tabuleiro[1][1]," | ",tabuleiro[1][2])
  escreva("  ---------")
  escreva(tabuleiro[2][0]," | ",tabuleiro[2][1]," | ", tabuleiro[2][2])
fim

logico validar(inteiro linha, inteiro coluna)
 se(linha>2||coluna>2||linha<0||coluna<0||tabuleiro[linha][coluna]!=0)então
   retornar falso
 senão
   retornar verdadeiro
 fimse
fim

inteiro jogada(inteiro jogador)
  inteiro linha, coluna, posicao
  lógico válido
  escreva(“Vez do jogador “, jogador, “: “)
  faça
      escreva(“Digite uma jogada (de 0 a 8): “)
      leia(posicao)
      linha <- posicao / 3
      coluna <- posicao % 3
      valido <- validar(linha, coluna)
  enquanto(valido==falso)

  se(jogador==1)então
      tabuleiro[linha][coluna] = 1
  senão
      tabuleiro[linha][coluna] = -1
  fimse
  retornar posicao
fim

inteiro vezDeQuem(inteiro jogador)
  se(jogador==1)então
     retornar 2
  senão
     retornar 1
  fimse
fim




nada mostrarGanhador(inteiro ganhador)
```

```
  se(ganhador==0)então
      escreva(“Empatou!!!”)
  senão
      escreva(“O jogador “, ganhador , “ venceu!!!”)
  fimse
  apresentarTabuleiro()
fim

inteiro verificarVitoria(inteiro jogador, inteiro posicao)
  inteiro i, j, temzero, somaL, somaC, somaDiagP, somaDiagS
  inteiro linha, coluna

  linha  <- posicao/3
  coluna <- posicao%3
  somaL <- tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2]
  somaC <- tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna]
  somaDiagP <- tabuleiro[0][0]+tabuleiro[1][1] + tabuleiro[2][2]
  somaDiagS <- tabuleiro[2][0] + tabuleiro[1][1] + tabuleiro[0][2]

  se(somaL*somaL == 9 || somaC *somaC==9 || somaDiagP *somaDiagP == 9 ||
     somaDiagS *somaDiagS == 9)então
        mostrarGanhador(jogador)
        retornar 1
  senão
        temzero = 0;
        para(i <- 0; i < 3; i++)faça
            para(j <- 0; j < 3; j++)faça
                se(tabuleiro[i][j]==0)então
                    temzero <- 1
                fimse
            fimpara
        fimpara
        se(temzero==1) então
            retornar 0
        senão
            mostrarGanhador(0)
            retornar 1
        fimse
 fimse
fim

início
  logico vitoria
  inteiro jogador, posicao
  jogador <- 1
  vitoria <- falso
  faça
      apresentarTabuleiro()
      posicao <- jogada(jogador)
      vitoria <- verificarVitoria(jogador, posicao)
      jogador < - vezDeQuem(jogador)
  enquanto(!vitoria)
fim
```

C

Obs: C não possui tipo "lógico" (como bool). Então, usamos um número inteiro e consideramos 0 como falso e 1 como verdadeiro. Ainda, quando uma variável é declarada em C, seu valor não é automaticamente zerado. Há lixo de memória. Por isso, na declaração, atribuímos valores iniciais.

Versão 1

```
#include <stdio.h>
#include <string.h>

int tabuleiro[3][3]={{0,0,0}, {0,0,0}, {0,0,0}};

int validar(int linha, int coluna){
    if(linha<0 || linha>2 || coluna<0 || coluna>2){
        printf("\nValor invalido! Jogue novamente:");
        return 0;
    }else if(tabuleiro[linha][coluna]!=0){
        printf("\nValor invalido! Jogue novamente:");
        return 0;
    }else
        return 1;
}

void apresentarTabuleiro(){
    printf("\n      0     1     2\n");
    printf("\n0     %d  |  %d  |  %d", tabuleiro[0][0], tabuleiro[0][1]
                                                    , tabuleiro[0][2]);
    printf("\n     ------------------");
    printf("\n1     %d  |  %d  |  %d", tabuleiro[1][0], tabuleiro[1][1]
                                                    , tabuleiro[1][2]);
    printf("\n     ------------------");
    printf("\n2     %d  |  %d  |  %d", tabuleiro[2][0], tabuleiro[2][1]
                                                    , tabuleiro[2][2]);
}

void jogada(int jogador){
    int valido, linha, coluna;
    printf("\n\nVez do jogador %d: ",jogador);
    do{
        printf("\nDigite a linha: ");
        scanf("%d",&linha);
        printf("\nDigite a coluna: ");
        scanf("%d",&coluna);
        valido = validar(linha, coluna);
    }while(valido==0);
    tabuleiro[linha][coluna]=jogador;
}
```

```
int vezDeQuem(int jogador){
    if(jogador==1)
            return 2;
        else
            return 1;
}

void mostrarGanhador(int ganhador){
    if(ganhador==0)
        printf("\n\nEmpatou!!!");
    else
        printf("\n\nO jogador %d venceu!!!",ganhador);
    apresentarTabuleiro();
}

int verificarVitoria(int jogador){
    int i, j, temzero;
     for(i=0;i<3;i++)
        if(tabuleiro[i][0] == tabuleiro[i][1] &&
           tabuleiro[i][1] == tabuleiro[i][2] && tabuleiro[i][0] !=0 ||
           tabuleiro[0][i] == tabuleiro[1][i] &&
           tabuleiro[1][i] == tabuleiro[2][i] && tabuleiro[0][i]!=0){
            mostrarGanhador(jogador);
            return 1;
        }


    //Verificando as diagonais
    if(tabuleiro[0][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[2][2]&&
       tabuleiro[1][1]!=0 ||
       tabuleiro[2][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[0][2]&&
       tabuleiro[1][1]!=0){
        mostrarGanhador(jogador);
        return 1;
    }

    //Verificando se houve empate
    temzero = 0;
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            if(tabuleiro[i][j]==0)
                temzero=1;

    if(temzero==1)
        return 0;
    else{
        mostrarGanhador(0);
        return 1;
    }
}
```

```
int main(){
    int vitoria=0;
    int jogador=1;
    do{
        apresentarTabuleiro();
        jogada(jogador);
        vitoria = verificarVitoria(jogador);
        jogador = vezDeQuem(jogador);
    }while(vitoria==0);
    return 0;
}
```

Versão 2

```
int tabuleiro[3][3]={{0,0,0}, {0,0,0}, {0,0,0}};

void apresentarTabuleiro(){
    printf("\nNúmero das casas do tabuleiro");
    printf("\n  0  |  1  |  2");
    printf("\n------------------");
    printf("\n  3  |  4  |  5");
    printf("\n------------------");
    printf("\n  6  |  7  |  8\n");

    printf("\nJogadas realizadas:");
    printf("\n  %d  |  %d  |  %d", tabuleiro[0][0], tabuleiro[0][1]
                                                  , tabuleiro[0][2]);
    printf("\n------------------");
    printf("\n  %d  |  %d  |  %d", tabuleiro[1][0], tabuleiro[1][1]
                                                  , tabuleiro[1][2]);
    printf("\n------------------");
    printf("\n  %d  |  %d  |  %d", tabuleiro[2][0], tabuleiro[2][1]
                                                  , tabuleiro[2][2]);
}

int validar(int linha, int coluna){
    if(linha<0 || linha>2 || coluna<0 || coluna>2){
        printf("\nValor invalido! Jogue novamente:");
        return 0;
    }else if(tabuleiro[linha][coluna]!=0){
        printf("\nValor invalido! Jogue novamente:");
        return 0;
    }else
        return 1;
}

int jogada(int jogador){
    int valido, linha, coluna, posicao;
    printf("\n\nVez do jogador %d: ",jogador);
    do{
```

```
        printf("\nDigite uma jogada (de 0 a 8): ");
        scanf("%d",&posicao);
        linha = posicao/3;
        coluna = posicao%3;
        valido = validar(linha, coluna);
    }while(valido==0);

    if(jogador==1)
        tabuleiro[linha][coluna]=1;
    else
        tabuleiro[linha][coluna]=-1;
    return posicao;
}

void mostrarGanhador(int ganhador){
    if(ganhador==0)
        printf("\n\nEmpatou!!!");
    else
        printf("\n\nO jogador %d venceu!!!",ganhador);
    apresentarTabuleiro();
}

int verificarVitoria(int jogador){
    int i, j, temzero;
     for(i=0;i<3;i++)
        if(tabuleiro[i][0]==tabuleiro[i][1] &&
           tabuleiro[i][1]==tabuleiro[i][2] && tabuleiro[i][0]!=0 ||
           tabuleiro[0][i]==tabuleiro[1][i] &&
           tabuleiro[1][i]==tabuleiro[2][i] && tabuleiro[0][i]!=0){
            mostrarGanhador(jogador);
            return 1;
        }

    //Verificando as diagonais
    if(tabuleiro[0][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[2][2]&&
       tabuleiro[1][1]!=0 ||
       tabuleiro[2][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[0][2]&&
       tabuleiro[1][1]!=0){
        mostrarGanhador(jogador);
        return 1;
    }

    //Verificando se houve empate
    temzero = 0;
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            if(tabuleiro[i][j]==0)
                temzero=1;

    if(temzero==1)
        return 0;
    else{
```

```
        mostrarGanhador(0);
        return 1;
    }
}

int main(){
    int vitoria=0;
    int jogador=1;
    do{
        apresentarTabuleiro();
        jogada(jogador);
        vitoria = verificarVitoria(jogador);
        jogador = vezDeQuem(jogador);
    }while(vitoria==0);
    return 0;
}
```

C++

Pode ser o mesmo código do C, ou:

Quando uma variável é declarada em C++, seu valor não é automaticamente zerado. Há lixo de memória. Por isso inicializamos a matriz assim que a declaramos.

Versão 1

```
#include <iostream>
using namespace std;

int tabuleiro[3][3]={{0,0,0}, {0,0,0}, {0,0,0}};

bool validar(int linha, int coluna){
    if(linha<0 || linha>2 || coluna<0 || coluna>2){
        cout << "\nValor invalido! Jogue novamente:";
        return false;
    }else if(tabuleiro[linha][coluna]!=0){
        cout << "\nValor invalido! Jogue novamente:";
        return false;
    }else
        return true;
}

void apresentarTabuleiro(){
    cout << "\n      0     1     2\n";
    cout << "\n0     "<<tabuleiro[0][0]<<"  |  "<<tabuleiro[0][1]<<"  |  "
         << tabuleiro[0][2];
    cout << "\n    ------------------";
    cout << "\n1     "<<tabuleiro[1][0]<<"  |  "<<tabuleiro[1][1]<<"  |  "
         << tabuleiro[1][2];
    cout << "\n    ------------------";
    cout << "\n2     "<<tabuleiro[2][0]<<"  |  "<<tabuleiro[2][1]<<"  |  "
         << tabuleiro[2][2];
}

void jogada(int jogador){
    int valido, linha, coluna;
    cout << "\n\nVez do jogador " << jogador << ": ";
    do{
        cout << "\nDigite a linha: ";
        cin >> linha;
        cout << "\nDigite a coluna: ";
        cin >> coluna;
        valido = validar(linha, coluna);
    }while(valido==0);
    tabuleiro[linha][coluna]=jogador;
}
```

```
int vezDeQuem(int jogador){
    if(jogador==1)
        return 2;
    else
        return 1;
}

void mostrarGanhador(int ganhador){
    if(ganhador==0)
        cout << "\n\nEmpatou!!!";
    else
        cout << "\n\nO jogador “ << ganhador << “ venceu!!!";
    apresentarTabuleiro();
}

bool verificarVitoria(int jogador){
    int i, j;
    bool temzero;
     for(i=0;i<3;i++)
        if(tabuleiro[i][0] == tabuleiro[i][1] &&
           tabuleiro[i][1] == tabuleiro[i][2] &&
           tabuleiro[i][0]!=0 ||
           tabuleiro[0][i] == tabuleiro[1][i] &&
           tabuleiro[1][i]==tabuleiro[2][i] &&
           tabuleiro[0][i]!=0){
            mostrarGanhador(jogador);
            return true;
        }


    //Verificando as diagonais
    if(tabuleiro[0][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[2][2]&&
       tabuleiro[1][1]!=0 ||
       tabuleiro[2][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[0][2]&&
       tabuleiro[1][1]!=0){
        mostrarGanhador(jogador);
        return true;
    }

    //Verificando se houve empate
    temzero = false;
    for(i=0; i<3 ;i++)
        for(j=0; j<3 ; j++)
            if(tabuleiro[i][j]==0)
                temzero=true;

    if(temzero)
        return false;
    else{
        mostrarGanhador(0);
        return true;
    }
```

```
}

int main(){
    bool vitoria=false;
    int jogador=1;
    do{
        apresentarTabuleiro();
        jogada(jogador);
        vitoria = verificarVitoria(jogador);
        jogador = vezDeQuem(jogador);
    }while(!vitoria);
    return 0;
}
```

Versão 2

```
#include <iostream>

using namespace std;

int tabuleiro[3][3]={{0,0,0}, {0,0,0}, {0,0,0}};

bool validar(int linha, int coluna){
    if(linha<0 || linha>2 || coluna<0 || coluna>2){
        cout << "\nValor invalido! Jogue novamente:";
        return false;
    }else if(tabuleiro[linha][coluna]!=0){
        cout << "\nValor invalido! Jogue novamente:";
        return false;
    }else
        return true;
}

void apresentarTabuleiro(){
    cout << "\nNúmero das casas do tabuleiro";
    cout << "\n  0  |  1  |  2";
    cout << "\n-----------------";
    cout << "\n  3  |  4  |  5";
    cout << "\n-----------------";
    cout << "\n  6  |  7  |  8\n";

    cout << "\nJogadas realizadas:\n";
    cout << "\n "<<tabuleiro[0][0] << "  |  " << tabuleiro[0][1] << "  |  "
         << tabuleiro[0][2];
    cout << "\n---------------";
    cout << "\n "<<tabuleiro[1][0] << "  |  " << tabuleiro[1][1] << "  |  "
         << tabuleiro[1][2];
    cout << "\n---------------";
    cout << "\n "<<tabuleiro[2][0] << "  |  " << tabuleiro[2][1] << "  |  "
         << tabuleiro[2][2];
}
```

```
int jogada(int jogador){
    int linha, coluna, posicao;
    bool valido;
    cout << "\n\nVez do jogador " << jogador << ": ";
    do{
        cout << "\nDigite uma jogada (de 0 a 8): ";
        cin >> posicao;
        linha = posicao/3;
        coluna = posicao%3;
        valido = validar(linha, coluna);
    }while(!valido);

    if(jogador==1)
        tabuleiro[linha][coluna]=1;
    else
        tabuleiro[linha][coluna]=-1;
    return posicao;
}

int vezDeQuem(int jogador){
    if(jogador==1)
        return 2;
    else
        return 1;
}

void mostrarGanhador(int ganhador){
    if(ganhador==0)
        printf("\n\nEmpatou!!!");
    else
        printf("\n\nO jogador %d venceu!!!",ganhador);
    apresentarTabuleiro();
}

bool verificarVitoria(int jogador, int posicao){
    int i, j, somaL, somaC, somaDiagP, somaDiagS;
    int linha, coluna;
    bool temzero;

    linha = posicao/3;
    coluna = posicao%3;
    somaL = tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2];
    somaC = tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna];
    somaDiagP = tabuleiro[0][0]+tabuleiro[1][1]+tabuleiro[2][2];
    somaDiagS = tabuleiro[2][0]+tabuleiro[1][1]+tabuleiro[0][2];

    if(somaL * somaL==9 || somaC * somaC==9 || somaDiagP*somaDiagP==9 ||
       somaDiagS*somaDiagS==9){
            mostrarGanhador(jogador);
            return 1;
```

```
    }else{
        temzero = false;
        for(i=0; i<3 ;i++)
            for(j=0; j<3 ; j++)
                if(tabuleiro[i][j]==0)
                    temzero=true;

        if(temzero)
            return false;
        else{
            mostrarGanhador(0);
            return true;
        }
    }
}

int main(){
    bool vitoria=false;
    int jogador=1, posicao;
    do{
        apresentarTabuleiro();
        posicao = jogada(jogador);
        vitoria = verificarVitoria(jogador, posicao);
        jogador = vezDeQuem(jogador);
    }while(!vitoria);
    return 0;
}
```

Go

Versão 1

```
package main

import (
    "fmt"
)

var tabuleiro [3][3]int

func validar(linha int, coluna int) bool {
    if linha < 0 || linha > 2 || coluna < 0 || coluna > 2 {
        fmt.Print("\nValor invalido! Jogue novamente:")
        return false
    } else if tabuleiro[linha][coluna] != 0 {
        fmt.Print("\nValor invalido! Jogue novamente:")
        return false
    } else {
        return true
    }
}

func apresentarTabuleiro() {
    fmt.Print("\n      0     1     2\n")
    fmt.Print("\n0     ",tabuleiro[0][0],"  |  ", tabuleiro[0][1], "  |  ",
              tabuleiro[0][2])
    fmt.Print("\n     ------------------")
    fmt.Print("\n1     ",tabuleiro[1][0],"  |  ", tabuleiro[1][1], "  |  ",
              tabuleiro[1][2])
    fmt.Print("\n     ------------------")
    fmt.Print("\n2     ",tabuleiro[2][0],"  |  ", tabuleiro[2][1], "  |  ",
              tabuleiro[2][2])
}

func jogada(jogador int) {
    var linha, coluna int
    var valido bool
    fmt.Print("\n\nVez do jogador ", jogador, ": ")
    for {
        fmt.Print("\nDigite a linha: ")
        fmt.Scanf("%d\n", &linha)
        fmt.Print("\nDigite a coluna: ")
        fmt.Scanf("%d\n", &coluna)
        valido = validar(linha, coluna)
        if valido {
            break
        }
    }
```

```
    tabuleiro[linha][coluna] = jogador
}

func vezDeQuem(jogador int) int {
    if jogador == 1 {
        return 2
    } else {
        return 1
    }
}

func mostrarGanhador(ganhador int) {
    if ganhador == 0 {
        fmt.Print("\n\nEmpatou!!!")
    } else {
        fmt.Print("\n\nO jogador ", ganhador, " venceu!!!")
    }
    apresentarTabuleiro()
}

func verificarVitoria(jogador int) bool {
    var i, j int
    var temzero bool
    for i = 0; i < 3; i++ {
        if tabuleiro[i][0]==tabuleiro[i][1] &&
           tabuleiro[i][1]==tabuleiro[i][2] &&
           tabuleiro[i][0]!=0 ||
           tabuleiro[0][i]==tabuleiro[1][i] &&
           tabuleiro[1][i]==tabuleiro[2][i] && tabuleiro[0][i] != 0 {
            mostrarGanhador(jogador)
            return true
        }
    }

    //Verificando as diagonais
    if tabuleiro[0][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[2][2]&&
       tabuleiro[1][1] != 0 ||
       tabuleiro[2][0]==tabuleiro[1][1]&&tabuleiro[1][1]==tabuleiro[0][2]&&
       tabuleiro[1][1] != 0 {
        mostrarGanhador(jogador)
        return true
    }

    //Verificando se houve empate
    temzero = false
    for i = 0; i < 3; i++ {
        for j = 0; j < 3; j++ {
            if tabuleiro[i][j] == 0 {
                temzero = true
            }
        }
    }
```

```
    if temzero {
        return false
    } else {
        mostrarGanhador(0)
        return true
    }
}

func main() {
    var vitoria bool
    jogador := 1
    for {
        apresentarTabuleiro()
        jogada(jogador)
        vitoria = verificarVitoria(jogador)
        jogador = vezDeQuem(jogador)
        if vitoria {
            break
        }
    }
}
```

Versão 2

```
package main

import (
    "fmt"
)

var tabuleiro [3][3]int

func validar(linha int, coluna int) bool {
    if linha < 0 || linha > 2 || coluna < 0 || coluna > 2 {
        fmt.Print("\nValor invalido! Jogue novamente:")
        return false
    } else if tabuleiro[linha][coluna] != 0 {
        fmt.Print("\nValor invalido! Jogue novamente:")
        return false
    } else {
        return true
    }
}

func apresentarTabuleiro() {
    fmt.Print("\nNúmeros das casas do tabuleiro\n")
    fmt.Print("\n 0  |  1  |  2")
    fmt.Print("\n---------------")
    fmt.Print("\n 3  |  4  |  5")
```

```
    fmt.Print("\n--------------")
    fmt.Print("\n 6  |  7  |  8\n")

    fmt.Print("\nJogadas Realizadas:\n")
    fmt.Print("\n ", tabuleiro[0][0], "  |  ", tabuleiro[0][1], "  |  ",
              tabuleiro[0][2])
    fmt.Print("\n---------------")
    fmt.Print("\n ", tabuleiro[1][0], "  |  ", tabuleiro[1][1], "  |  ",
              tabuleiro[1][2])
    fmt.Print("\n---------------")
    fmt.Print("\n ", tabuleiro[2][0], "  |  ", tabuleiro[2][1], "  |  ",
              tabuleiro[2][2])
}

func jogada(jogador int) int {
    var linha, coluna, posicao int
    var valido bool
    fmt.Print("\n\nVez do jogador ", jogador, ": ")
    for {
        fmt.Print("\nDigite uma jogada (de 0 a 8): ")
        fmt.Scanf("%d\n", &posicao)
        linha = posicao / 3
        coluna = posicao % 3
        valido = validar(linha, coluna)
        if valido {
            break
        }
    }
    if jogador == 1 {
        tabuleiro[linha][coluna] = 1
    } else {
        tabuleiro[linha][coluna] = -1
    }
    return posicao
}

func vezDeQuem(jogador int) int {
    if jogador == 1 {
        return 2
    } else {
        return 1
    }
}

func mostrarGanhador(ganhador int) {
    if ganhador == 0 {
        fmt.Print("\n\nEmpatou!!!")
    } else {
        fmt.Print("\n\nO jogador ", ganhador, " venceu!!!")
    }
    apresentarTabuleiro()
}
```

```
func verificarVitoria(jogador int, posicao int) bool {
    var i, j, somaL, somaC, somaDiagP, somaDiagS int
    var linha, coluna int
    var temzero bool

    linha = posicao / 3
    coluna = posicao % 3

    somaL = tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2]
    somaC = tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna]
    somaDiagP = tabuleiro[0][0]+tabuleiro[1][1]+tabuleiro[2][2]
    somaDiagS = tabuleiro[2][0]+tabuleiro[1][1]+tabuleiro[0][2]

    if somaL*somaL == 9 || somaC*somaC == 9 || somaDiagP*somaDiagP == 9 ||
       somaDiagS*somaDiagS == 9 {
        mostrarGanhador(jogador)
        return true
    } else {
        temzero = false
        for i = 0; i < 3; i++ {
            for j = 0; j < 3; j++ {
                if tabuleiro[i][j] == 0 {
                    temzero = true
                }
            }
        }
        if temzero {
            return false
        } else {
            mostrarGanhador(0)
            return true
        }
    }
}

func main() {
    var vitoria bool
    var posicao int
    jogador := 1
    for {
        apresentarTabuleiro()
        posicao = jogada(jogador)
        vitoria = verificarVitoria(jogador, posicao)
        jogador = vezDeQuem(jogador)
        if vitoria {
            break
        }
    }
}
```

Ruby

Em Ruby, variáveis globais precisam ser representadas pelo símbolo $ no início de seus identificadores. Ainda, em linguagens tipadas em tempo de execução, temos que tomar cuidado onde as variáveis são usadas pela primeira vez. Por exemplo, em Ruby, na função jogada(), se não usássemos as variáveis linha e coluna logo no começo da função, e a usássemos pela primeira vez dentro do laço ***do***, estas variáveis apenas existiriam dentro do laço ***do***! E não poderiam ser usadas após o ***do*** terminar (isso já não aconteceria em Python, apesar de seu tipo também ser definido em tempo de execução.

Versão 1

```
$tabuleiro  = [
    [0,0,0],
    [0,0,0],
    [0,0,0]
]

def apresentarTabuleiro()
    print "\n    0     1     2\n"
    print "\n0    ", $tabuleiro[0][0], "  |  ", $tabuleiro[0][1], "  | "
                                          , $tabuleiro[0][2]
    print "\n   ------------------"
    print "\n1    ", $tabuleiro[1][0], "  |  ", $tabuleiro[1][1], "  |  "
                                          , $tabuleiro[1][2]
    print "\n   ------------------"
    print "\n2    ", $tabuleiro[2][0], "  |  ", $tabuleiro[2][1], "  |  "
                                          , $tabuleiro[2][2]
end

def validar(linha, coluna)
    if linha<0 || linha>2 || coluna<0 || coluna>2
        print "\nValor invalido! Jogue novamente:"
        return false
    elsif $tabuleiro[linha][coluna]!=0
        printf "\nValor invalido! Jogue novamente:"
        return false
    else
        return true
    end
end

def jogada(jogador)
    linha=0
    coluna=0
    print "\n\nVez do jogador ", jogador, ": "
    loop do
        print "\nDigite a linha: "
        linha = gets.chomp.to_i
        print "\nDigite a coluna: "
        coluna = gets.chomp.to_i
        valido = validar(linha, coluna);
        if valido
```

```
            break
        end
    end
    $tabuleiro[linha][coluna]=jogador;
end

def vezDeQuem(jogador)
    if jogador==1
        return 2
    else
        return 1
    end
end

def mostrarGanhador(ganhador)
    if ganhador==0
        print "\n\nEmpatou!!!"
    else
        print "\n\nO jogador ", ganhador, " venceu!!!"
    end
    apresentarTabuleiro()
end

def verificarVitoria(jogador)
     for i in 0..2 do
        if $tabuleiro[i][0]==$tabuleiro[i][1] &&
           $tabuleiro[i][1]==$tabuleiro[i][2] &&
           $tabuleiro[i][0]!=0 || $tabuleiro[0][i]==$tabuleiro[1][i] &&
           $tabuleiro[1][i]==$tabuleiro[2][i] && $tabuleiro[0][i]!=0
            mostrarGanhador(jogador);
            return true
        end
    end

    if $tabuleiro[0][0]==$tabuleiro[1][1] &&
       $tabuleiro[1][1]==$tabuleiro[2][2] &&
       $tabuleiro[1][1]!=0 || $tabuleiro[2][0]==$tabuleiro[1][1] &&
       $tabuleiro[1][1]==$tabuleiro[0][2] && $tabuleiro[1][1]!=0
        mostrarGanhador(jogador)
        return true
    end

    temzero = false
    for i in 0..2 do
        for j in 0..2 do
            if $tabuleiro[i][j]==0
                temzero=true
            end
        end
    end

    if temzero
```

```
        return false
    else
        mostrarGanhador(0)
        return true
    end
end

jogador=1
loop do
    apresentarTabuleiro()
    jogada(jogador)
    vitoria = verificarVitoria(jogador)
    jogador = vezDeQuem(jogador)
    if vitoria
        break
    end
end
```

Versão 2

```
$tabuleiro  = [
    [0,0,0],
    [0,0,0],
    [0,0,0]
]

def apresentarTabuleiro()
    print "\nNúmero das casas do tabuleiro"
    print "\n  0  |  1  |  2"
    print "\n-----------------"
    print "\n  3  |  4  |  5"
    print "\n-----------------"
    print "\n  6  |  7  |  8\n"

    print "\nJogadas realizadas:"
    print "\n  ", $tabuleiro[0][0], "  |  ", $tabuleiro[0][1], "  |  "
                                        , $tabuleiro[0][2]
    print "\n-----------------"
    print "\n  ", $tabuleiro[1][0], "  |  ", $tabuleiro[1][1], "  |  "
                                        , $tabuleiro[1][2]
    print "\n-----------------"
    print "\n  ", $tabuleiro[2][0], "  |  ", $tabuleiro[2][1], "  |  "
                                        , $tabuleiro[2][2]
end

def validar(linha, coluna)
    if linha<0 || linha>2 || coluna<0 || coluna>2
        print "\nValor invalido! Jogue novamente:"
        return false
    elsif $tabuleiro[linha][coluna]!=0
        printf "\nValor invalido! Jogue novamente:"
```

```
        return false
    else
        return true
    end
end

def jogada(jogador)
    linha=0
    coluna=0
    posicao=0
    print "\n\nVez do jogador ", jogador, ": "
    loop do
        print "\nDigite uma jogada (de 0 a 8): "
        posicao = gets.chomp.to_i
        linha = posicao/3
        coluna = posicao%3
        valido = validar(linha, coluna)
        if valido
            break
        end
    end

    if jogador==1
        $tabuleiro[linha][coluna]=1
    else
        $tabuleiro[linha][coluna]=-1
    end
    return posicao
end

def vezDeQuem(jogador)
    if jogador==1
        return 2
    else
        return 1
    end
end

def mostrarGanhador(ganhador)
    if ganhador==0
        print "\n\nEmpatou!!!"
    else
        print "\n\nO jogador ", ganhador, " venceu!!!"
    end
    apresentarTabuleiro()
end

def verificarVitoria(jogador, posicao)
    linha=posicao/3
    coluna=posicao%3
    somaL=$tabuleiro[linha][0]+$tabuleiro[linha][1]+$tabuleiro[linha][2]
    somaC=$tabuleiro[0][coluna]+$tabuleiro[1][coluna]+$tabuleiro[2][coluna]
```

```
    somaDiagP=$tabuleiro[0][0]+$tabuleiro[1][1]+$tabuleiro[2][2]
    somaDiagS=$tabuleiro[2][0]+$tabuleiro[1][1]+$tabuleiro[0][2]

    if somaL * somaL==9 || somaC * somaC==9 || somaDiagP*somaDiagP==9
                                            || somaDiagS*somaDiagS==9
            mostrarGanhador(jogador)
            return true
    else
        temzero = false
        for i in 0..2 do
            for j in 0..2 do
                if $tabuleiro[i][j]==0
                    temzero=true
                end
            end
        end

        if temzero
            return false
        else
            mostrarGanhador(0)
            return true
        end
    end
end

jogador=1
loop do
    apresentarTabuleiro()
    posicao = jogada(jogador)
    vitoria = verificarVitoria(jogador, posicao)
    jogador = vezDeQuem(jogador)
    if vitoria
        break
    end
end
```

Python

OBS: em Python, a indentação é importante, É ela que demarca o início e fim de um bloco. O nosso código abaixo possui algumas poucas condições grandes e que não cabem em uma linha só. Para evitar erros de indentação por parte do leitor, caso queira reproduzir o código, coloquei em verde as condições que devem ser escritas em apenas uma linha.

Versão 1

```
import numpy as np

tabuleiro = np.zeros((3,3), int)

def apresentarTabuleiro():
    print('\n   0     1     2\n')
    print('0 ',tabuleiro[0][0],' | ',tabuleiro[0][1],' | ',tabuleiro[0][2])
    print('   ------------------')
    print('1 ',tabuleiro[1][0],' | ',tabuleiro[1][1],' |
',tabuleiro[1][2])
    print(' ------------------')
    print('2 ',tabuleiro[2][0],' | ',tabuleiro[2][1],' | ',tabuleiro[2][2])

def validar(linha, coluna):
    if linha<0 or linha>2 or coluna<0 or coluna>2:
        print ('\nValor invalido! Jogue novamente:')
        return False
    elif tabuleiro[linha][coluna]!=0:
        print  ('\nValor invalido! Jogue novamente:')
        return False
    else :
        return True


def jogada(jogador):
    print ('\n\nVez do jogador ', jogador, ': ')
    while True:
        linha = int(input('\nDigite a linha: '))
        coluna =int(input('\nDigite a coluna: '))
        valido = validar(linha, coluna)
        if bool(valido):
            break
    tabuleiro[linha][coluna]=jogador

def vezDeQuem(jogador):
    if jogador==1:
        return 2
    else:
        return 1
```

```
def mostrarGanhador(ganhador):
    if ganhador==0:
        print ('\n\nEmpatou!!!')
    else:
        print ('\n\nO jogador ', ganhador, ' venceu!!!')
    apresentarTabuleiro()

def verificarVitoria(jogador):
    for i in range(3):
        if tabuleiro[i][0]==tabuleiro[i][1] and
tabuleiro[i][1]==tabuleiro[i][2] and tabuleiro[i][0]!=0 or
tabuleiro[0][i]==tabuleiro[1][i] and tabuleiro[1][i]==tabuleiro[2][i] and
tabuleiro[0][i]!=0:
            mostrarGanhador(jogador)
            return True


    if tabuleiro[0][0]==tabuleiro[1][1] and
tabuleiro[1][1]==tabuleiro[2][2] and tabuleiro[1][1]!=0 or
tabuleiro[2][0]==tabuleiro[1][1] and tabuleiro[1][1]==tabuleiro[0][2] and
tabuleiro[1][1]!=0:
        mostrarGanhador(jogador)
        return True

    temzero = False
    for i in range(3):
        for j in range(3):
            if tabuleiro[i][j]==0:
                temzero=True

    if temzero:
        return False
    else:
        mostrarGanhador(0)
        return True

jogador=1
while True:
    apresentarTabuleiro()
    jogada(jogador)
    vitoria = verificarVitoria(jogador)
    jogador = vezDeQuem(jogador)
    if bool(vitoria):
        break
```

Versão 2

```
import numpy as np

tabuleiro = np.zeros((3,3), int)

def apresentarTabuleiro():
    print ('\nNúmero das casas do tabuleiro')
    print (' 0  |  1  |  2')
    print ('---------------')
    print (' 3  |  4  |  5')
    print ('---------------')
    print (' 6  |  7  |  8\n')

    print ('\nJogadas realizadas:')
    print ('',tabuleiro[0][0],' | ',tabuleiro[0][1],' | ',tabuleiro[0][2])
    print ('---------------')
    print ('',tabuleiro[1][0],' | ',tabuleiro[1][1],' |
',tabuleiro[1][2])
    print ('---------------')
    print ('',tabuleiro[2][0],' | ',tabuleiro[2][1],' | ',tabuleiro[2][2])

def validar(linha, coluna):
    if linha<0 or linha>2 or coluna<0 or coluna>2:
        print ('\nValor invalido! Jogue novamente:')
        return False
    elif tabuleiro[linha][coluna]!=0:
        print  ('\nValor invalido! Jogue novamente:')
        return False
    else :
        return True

def jogada(jogador):
    print ('\n\nVez do jogador ', jogador, ': ')
    while True:
        posicao = int(input('\nDigite uma jogada (de 0 a 8): '))
        linha = int(posicao/3)
        coluna = posicao%3
        valido = validar(linha, coluna)
        if valido:
            break

    if jogador==1:
        tabuleiro[linha][coluna]=1
    else:
        tabuleiro[linha][coluna]=-1

    return posicao
```

```
def vezDeQuem(jogador):
    if jogador==1:
        return 2
    else:
        return 1

def mostrarGanhador(ganhador):
    if ganhador==0:
        print ('\n\nEmpatou!!!')
    else:
        print ('\n\nO jogador ', ganhador, ' venceu!!!')
    apresentarTabuleiro()


def verificarVitoria(jogador, posicao):
    linha = int(posicao/3)
    coluna = posicao%3
    somaL = tabuleiro[linha][0]+tabuleiro[linha][1]+tabuleiro[linha][2]
    somaC = tabuleiro[0][coluna]+tabuleiro[1][coluna]+tabuleiro[2][coluna]
    somaDiagP = tabuleiro[0][0] + tabuleiro[1][1] + tabuleiro[2][2]
    somaDiagS = tabuleiro[2][0] + tabuleiro[1][1] + tabuleiro[0][2]

    if somaL * somaL==9 or somaC * somaC==9 or somaDiagP*somaDiagP==9 or
somaDiagS * somaDiagS==9:
        mostrarGanhador(jogador)
        return True
    else:
        temzero = False
        for i in range(3):
            for j in range(3):
                if tabuleiro[i][j]==0:
                    temzero=True

        if temzero:
            return False
        else:
            mostrarGanhador(0)
            return True

jogador=1
while True:
    apresentarTabuleiro()
    posicao = jogada(jogador)
    vitoria = verificarVitoria(jogador, posicao)
    jogador = vezDeQuem(jogador)
    if vitoria:
        break
```

Exercícios Resolvidos

Exercícios de Instruções Escreva, Leia e Variáveis

1.

```
início
    escreva(“Olá, mundo!!!”)
fim
```

2.

```
início
    escreva(“Eu estou “, “fazendo “, “concatenação!”)
fim
```

3.

```
início
    escreva(“Olá, amigo! “, “Este texto tem uma vírgula dentro das
aspas. ”, “Mas eu sei que quando está dentro das aspas ela representa apenas
uma vírgula. “, “E quando está fora representa o operador de concatenação.”)
fim
```

4.

```
início
    cadeia nome
    escreva(“Digite o seu nome: “)
    leia(nome)
    escreva(“Seja bem vindo, “, nome)
fim
```

5.

```
início
    inteiro idade
    escreva(“Digite a sua idade: “)
    leia(idade)
    escreva(“Você tem “, idade, “ anos!”)
fim
```

6.

```
início
    cadeia nome
    inteiro idade
    escreva(“Digite o seu nome: “)
    leia(nome)
    escreva(“Digite a sua idade: “)
    leia(idade)
    escreva(“Legal, “, nome, “! Você tem “, idade, “ anos!”)
fim
```

7.

```
início
    inteiro idade, ano
    escreva(“Digite a sua idade: “)
    leia(idade)
    escreva(“Digite o seu ano de nascimento: ”)
    leia(ano)
    escreva(“Você tem “, idade, “ anos e nasceu em “, ano, “!”)
fim
```

8.

```
início
    inteiro ano, idade
    escreva(“Digite o seu ano de nascimento: “)
    leia(ano)
    idade <- 2022 - ano
    escreva(“Você tem entre “, idade-1 , “ e “, idade, “ anos”)
fim
```

9.

```
início
    inteiro ano, idade
    escreva(“Digite a sua idade: “)
    leia(idade)
    ano <- 2022 - idade
    escreva(“Você nasceu entre “, ano-1 , “ e “, ano)
fim
```

10.

```
início
    real celsius, fahrenheit
    escreva(“Digite a temperatura em Celsius: “)
    leia(celsius)
    fahrenheit <- 9 * celsius / 5 + 32
    escreva(“Em Fahrenheit: “, fahrenheit)
fim
```

11.

```
início
    real celsius, fahrenheit
    escreva(“Digite a temperatura em Fahrenheit: “)
    leia(fahrenheit)
    celsius <- 5 * (fahrenheit – 32) /9
    escreva(“Em Celsius: “, celsius)
fim
```

12.

```
início
    inteiro a, b, aux
    escreva(“Digite um valor: “)
    leia(a)
    escreva(“Digite outro valor: “)
    leia(b)
    aux <- a
    a <- b
    b <- aux
    escreva(“Novo valor de a: “, a, “. Novo valor de b: “, b)
fim
```

Exercícios de se-senão

1.

```
início
    real salario
    escreva(“Digite o seu salário: “)
    leia(salario)
    se(salario > 10000)então
        escreva(“Você ganha bem!”)
    fimse
fim
```

2.

```
início
    real salario
    escreva(“Digite o seu salário: “)
    leia(salario)
    se(salario < 1045)então
        escreva(“Você ganha menos do que um salário mínimo”)
    senão
        escreva(“Você ganha pelo menos um salário mínimo”)
    fimse
fim
```

3.

```
início
    real temperatura
    escreva(“Digite a sua temperatura: “)
    leia(temperatura)
    se(temperatura < 37)então
        escreva(“Você está sem febre”)
    senão
        escreva(“Você está com febre!”)
    fimse
fim
```

4.

```
início
    inteiro senha
    escreva(“Digite sua senha: “)
    leia(senha)
    se(senha != 123456)então
        escreva(“Senha não confere!”)
    fimse
fim
```

5.

```
início
    inteiro senha, confirmacao
    escreva(“Digite sua senha: “)
    leia(senha)
    escreva(“Digite a confirmação: “)
    leia(confirmacao)
    se(senha == confirmacao)então
        escreva(“Senha criada com sucesso”)
    senão
        escreva(“Senha não confere”)
    fimse
fim
```

6.

```
início
    real n1, n2, media
    escreva(“Digite a nota 1: “)
    leia(n1)
    escreva(“Digite a nota 2: “)
    leia(n2)
    media <- (n1 + n2) / 2
    se(media >= 6)então
        escreva(“Aprovado”)
    senão
        escreva(“Reprovado”)
    fimse
fim
```

7.

```
início
    cadeia nome
    caractere genero
    escreva(“Digite o seu nome: “)
    leia(nome)
    escreva(“Digite o seu gênero (m/f): ”)
    leia(genero)
    se(genero == ‘f’)então
        escreva(“Olá, senhora “, nome)
    senão
        escreva(“Olá, senhor “, nome)
    fimse
fim
```

8.

```
início
    inteiro num1, num2
    escreva(“Digite o primeiro valor: “)
    leia(num1)
    escreva(“Digite o segundo número: “)
    leia(num2)
    se(num1 < num2)então
        escreva(num1, “ e “, num2)
    senão
        escreva(num2, “ e “, num1)
    fimse
fim
```

9.

```
início
    inteiro total, defeito
    real limite
    escreva(“Digite o total de parafusos: “)
    leia(total)
    escreva(“Digite a quantidade de peças com defeito: “)
    leia(defeito)
    limite <- total * 0.1
    se(defeito > limite)então
        escreva(“Produção está ruim”)
    senão
        escreva(“Produção está boa”)
    fimse
fim
```

10.

```
início
    inteiro num
    escreva(“Digite um número: “)
    leia(num)
    se(num % 2 == 0)então
        escreva(“É par”)
    senão
        escreva(“É ímpar”)
    fimse
fim
```

Exercícios de se-senão Encadeado

1.

```
início
    inteiro num
    escreva(“Digite um número: “)
    leia(num)
    se(num > 0)então
        escreva(“Valor positivo“)
    senão se(num < 0)então
        escreva(“Valor negativo”)
    senão
        escreva(“Valor neutro”)
    fimse
fim
```

2.

```
início
    real salario
    escreva(“Digite o seu salário: “)
    leia(salario)
    se(salario > 18000)então
        escreva(“Você é classe A”)
    senão se(salario > 9000)então
        escreva(“Você é classe B”)
    senão
        escreva(“Você é classe C ou menos”)
    fimse
fim
```

3.

```
início
    real temperatura
    escreva(“Digite a temperatura da água”)
    leia(temperatura)
    se(temperatura > 100)então
        escreva(“A água está em estado de vapor”)
    senão se(temperatura > 0)então
        escreva(“A água está em estado líquido”)
    senão
        escreva(“A água está em estado sólido”)
    fimse
fim
```

4.

```
início
    inteiro idade
    escreva(“Digite a sua idade: “)
    leia(idade)
    se(idade >= 65)então
        escreva(“Você está na melhor idade”)
    senão se(idade >= 40)então
        escreva(“Você está na meia idade”)
    senão se(idade >= 30)então
        escreva(“Você é adulto”)
    senão se(idade >= 18)então
        escreva(“Você é adulto jovem”)
    senão se(idade >= 12)então
        escreva(“Você é adolescente”)
    senão se(idade >= 2)então
        escreva(“Você é criança”)
    senão
        escreva(“Você é bebê”)
    fimse
fim
```

5.

```
início
    real altura, massa, imc
    escreva(“Digite sua altura em metros: “)
    leia(altura)
    escreva(“Digite sua massa corporal: “)
    leia(massa)
    imc <- massa/(altura * altura)
    se(imc < 17)então
        escreva(“Muito abaixo do peso”)
    senão se(imc < 18.5)então
        escreva(“Abaixo do peso”)
    senão se(imc < 25)então
        escreva(“Peso normal”)
    senão se(imc < 30)então
        escreva(“Acima do peso”)
    senão se(imc < 35)então
        escreva(“Obesidade I”)
    senão se(imc < 40)então
        escreva(“Obesidade II (severa)”)
    senão
        escreva(“Obesidade III (mórbida)”)
    fimse
fim
```

Exercício de escolha-caso

1.
```
início
    inteiro opc
    escreva(“MONSTRODEX: ”)
    escreva(“1- Zikachu”)
    escreva(“2- Zulbassauro”)
    escreva(“3- Zharmander”)
    leia(opc)
    escolha(opc)
        caso 1:
            escreva(“Monstrinho elétrico da categoria rato”)
        caso 2:
            escreva(“Monstrinho de grama da categoria semente”)
        caso 3:
            escreva(“Monstrinho de fogo da categoria lagarto”)
        caso padrão:
            escreva(“Monstrinho não cadastrado. Há 8900 monstrinhos!”)
            escreva(“Temos que pegar!”)
    fimescolha
fim
```

2.
```
início
    caractere opc
    escreva(“MENU:”)
    escreva(“A- Avião”)
    escreva(“B- Carro”)
    escreva(“C- Cruzeiro”)
    leia(opc)
    escolha(opc)
        caso ‘A’:
            escreva(“É mais rápido!”)
        caso ‘B’:
            escreva(“É mais barato!”)
        caso ‘C’:
            escreva(“É mais bonito!”)
    fimescolha
fim
```

3.
```
início
    inteiro opc
    real area, altura, base, raio
    escreva(“MENU:”)
    escreva(“1- Calcular a área de um retângulo”)
    escreva(“2- Calcular a área de um círculo”)
    escreva(“3- Calcular a área de um triângulo”)
    leia(opc)
```

```
        escolha(opc)
            caso 1:
                escreva(“Digite a altura do retângulo”)
                leia(altura)
                escreva(“Digite a base do retângulo”)
                leia(base)
                area <- altura * base
                escreva(“A área do retângulo é “, area)
            caso 2:
                escreva(“Digite o raio do círculo”)
                leia(raio)
                area <- 3.14 * raio
                escreva(“A área do círculo é “, area)
            caso 3:
                escreva(“Digite a altura do triângulo”)
                leia(altura)
                escreva(“Digite a base do triângulo”)
                leia(base)
                area <- altura * base / 2
                escreva(“A área do círculo é “, area)
        fimescolha
    fim
```

```
4.
    início
        inteiro mes
        escreva(“Digite o número do mês (1 a 12):”)
        leia(mes)
        escolha(mes)
            caso 1, 3, 5, 7, 8, 10, 12:
                escreva(“O mês tem 31 dias”)
            caso 4, 6, 9, 11:
                escreva(“O mês tem 30 dias”)
            caso 2:
                escreva(“O mês tem 28 dias”)
            caso padrão:
                escreva(“Mês inválido”)
        fimescolha
    fim
```

Exercícios de Operadores Lógicos e Condições Compostas

```
1.
    início
        inteiro num
        escreva(“Digite um número entre 1 e 6”)
        leia(num)
        se(num >= 1 && num <= 6)então
            escreva(“Valor digitado com sucesso”)
        senão
            escreva(“Valor fora do intervalo permitido”)
        fimse
    fim
```

2.

```
início
    inteiro num
    escreva(“Digite um número inferior a 1 ou superior a 6”)
    leia(num)
    se(num < 1 || num > 6)então
        escreva(“Valor digitado com sucesso”)
    senão
        escreva(“Valor não pode estar entre 1 e 6”)
    fimse
fim
```

3.

```
início
    inteiro idade
    escreva(“Digite a sua idade: “)
    leia(idade)
    se(idade >= 18 && idade <=65)então
        escreva(“Você é obrigado a votar”)
    senão
        escreva(“Você não é obrigado a votar”)
    fimse
fim
```

4.

```
início
    cadeia usuario
    inteiro senha
    escreva(“Digite o usuário: “)
    leia(usuario)
    escreva(“Digite a senha: “)
    leia(senha)
    se(usuario == “chefe” && senha == 123456)então
        escreva(“Login realizado”)
    senão
        escreva(“Usuário e/ou senha incorretos”)
    fimse
fim
```

5.

```
início
    inteiro idade, tempo
    caractere genero
    escreva(“Digite a sua idade: “)
    leia(idade)
    escreva(“Digite o seu tempo de trabalho: “)
    leia(tempo)
    escreva(“Digite o seu gênero (f/m)”)
    leia(genero)
    se(genero == ‘m’ && tempo >= 30 && idade >= 65 ||
       genero == ’f’ && tempo >= 25 && idade >= 60)então
        escreva(“Você pode se aposentar”)
    senão
        escreva(“Você não pode se aposentar”)
    fimse
fim
```

6.

```
início
    real lado1, lado2, lado3
    escreva(“Digite um lado do triângulo: ”)
    leia(lado1)
    escreva(“Digite o segundo lado do triângulo: “)
    leia(lado2)
    escreva(“Digite o terceiro lado do triângulo: “)
    leia(lado3)
    se(lado1 < lado2 + lado3 && lado2 < lado1 + lado3 &&
       lado3 < lado1 + lado2)então
        escreva(“É um triângulo”)
    senão
        escreva(“Não é um triângulo”)
    fimse
fim
```

Exercícios de se-senão Aninhado

1.

```
início
    real altura
    inteiro idade
    escreva(“Digite a sua altura”)
    leia(altura)
    escreva(“Digite a sua idade”)
    leia(idade)
    se(altura < 1.6)então
        se(idade < 5)então
            escreva(“Pode brincar no pula-pula e Casinha”)
        senão se(idade <= 8)então
            escreva(“Pode brincar na prancha do pirata e piscina de
                    bolinhas”)
        senão
            escreva(“Pode brincar de pebolim, ping-pong e basquete”)
        fimse
    senão
        escreva(“Você é muito grande para entrar no parque”)
    fimse
fim
```

2.

```
início
    inteiro semestre, opc
    escreva(“Digite o semestre em que cursa a faculdade”)
    leia(semestre)
    se(semestre >= 7)então
        escreva(“MENU:”)
        escreva(“1-Matricular na disciplina de Jogos Digitais”)
        escreva(“2-Matricular na disciplina de Design de Jogos”)
        escreva(“3-Matricular na disciplina de Realidade Virtual”)
        leia(opc)
        escolha(opc)
            caso 1:
                escreva(“Jogos Digitais confirmado”)
            caso 2:
                escreva(“Design de Jogos confirmado”)
            caso 3:
                escreva(“Realidade Virtual confirmado”)
        fimescolha
    senão
        escreva(“Você não pode se matricular em disciplinas optativas”)
    fimse
fim
```

3.

```
início
    cadeia nacionalidade, regiao
    escreva(“Digite sua nacionalidade”)
    leia(nacionalidade)
    se(nacionalidade == “brasileiro”)então
        escreva(“Em que região você nasceu?”)
        leia(regiao)
        se(regiao == “sul”)então
            escreva(“Você está acostumado com frio”)
        senão se(regiao == “sudeste”)então
            escreva(“Você está acostumado com chuva”)
        senão se(regiao == “centro-oeste”)então
            escreva(“Você está acostumado com clima abafado”)
        senão se(regiao == “nordeste”)então
            escreva(“Você está acostumado com praias bonitas”)
        senão
            escreva(“Você é acostumado com chuvas no começo da tarde”)
        fimse
    senão
        escreva(“Seja bem-vindo ao Brasil”)
    fimse
fim
```

4.

```
início
    inteiro senha, opc
    escreva(“Digite a senha de espera que pegou: “)
    leia(senha)
    escreva(“MENU:”)
    escreva(“1-Prioridade”)
    escreva(“2-Aposentado”)
    escreva(“3-Comum”)
    leia(opc)
    se(senha < 100)então
        escreva(“Aguarde para ser atendido”)
    senão
        se(opc == 1)então
            escreva(“Você será reagendado para amanhã”)
        senão se(opc == 2)então
            escreva(“Você será reagendado para depois de amanhã”)
        senão
            escreva(“Você deve tentar outro dia”)
        fimse
    fimse
fim
```

Obs: o exercício acima seria melhor escrito se utilizarmos escolha-caso no lugar do se-senão aninhado. Foi feito com se-senão aninhado para mostrar que é possível.

5.

```
início
    inteiro opc1, opc2
    real num1, num2
    escreva("1-Conversão de temperatura")
    escreva("2-Conversão de distância")
    leia(opc1)
    escolha(opc1)
        caso 1:
            escreva("1-De Celsius para Fahrenheit")
            escreva("2-De Fahrenheit para Celsius")
            escreva("3-De Celsius para Kelvin")
            leia(opc2)
            escolha(opc2)
                caso 1:
                    escreva("Digite a temperatura em Celsius:")
                    leia(num1)
                    num2 <- 9 * num1 / 5 + 32
                    escreva("Em Fahrenheit: ", num2)
                caso 2:
                    escreva("Digite a temperatura em Fahrenheit:")
                    leia(num1)
                    num2 <- 5 * (num1 – 32) / 9
                    escreva("Em Celsius: ", num2)
                caso 3:
                    escreva("Digite a temperatura em Celsius:")
                    leia(num1)
                    num2 <- num1 + 273
                    escreva("Em Kelvin: ", num2)
            fimescolha
        caso 2:
            escreva("1-De quilômetros para milhas")
            escreva("2-De milhas para quilômetros")
            leia(opc2)
            escolha(opc2)
                caso 1:
                    escreva("Digite a distância em quilômetros:")
                    leia(num1)
                    num2 <- num1 * 0.62137
                    escreva("Em milhas: ", num2)
                caso 2:
                    escreva("Digite a distância em milhas:")
                    leia(num1)
                    num2 <- num1 / 0.62137
                    escreva("Em quilômetros: ", num2)
            fimescolha
    fimescolha
fim
```

Exercícios de Laço para

1.
```
início
    inteiro i
    para(i <- 1; i <= 10; i <- i + 1)faça
        escreva(i)
    fimpara
fim
```

2.
```
início
    inteiro i
    para(i <- 10; i >=1; i <- i – 1)faça
        escreva(i)
    fimpara
fim
```

3.
```
início
    inteiro i
    para(i <- 5; i <= 15; i++)faça
        escreva(i)
    fimpara
fim
```

4.
```
início
    inteiro i
    para(i <- 15; i >= 5; i--)faça
        escreva(i)
    fimpara
fim
```

5.
```
início
    inteiro i
    para(i <- 2; i <= 10; i <- i + 2)faça
        escreva(i)
    fimpara
fim
```

6.
```
início
    inteiro i
    para(i <- 1; i <= 9; i <- i + 2)faça
        escreva(i)
    fimpara
fim
```

7.

```
início
    inteiro i, max
    escreva(“Digite o valor máximo: “)
    leia(max)
    para(i <- 1; i <= max; i++)faça
        escreva(i)
    fimpara
fim
```

8.

```
início
    inteiro min, i
    escreva(“Digite o valor inicial: “)
    leia(min)
    para(i <- min; i <= 10; i++)faça
        escreva(i)
    fimpara
fim
```

9.

```
início
    inteiro i, min, max
    escreva(“Digite o valor inicial: “)
    leia(min)
    escreva(“Digite o valor final: “)
    leia(max)
    para(i <- min; i <= max; i++)faça
        escreva(i)
    fimpara
fim
```

10.

```
início
    inteiro i, comeco, fim
    escreva(“Digite o valor inicial: “)
    leia(comeco)
    escreva(“Digite o valor final: “)
    leia(fim)
    se(comeco < fim)então
        para(i <- comeco; i <= fim; i++)faça
            escreva(i)
        fimpara
    senão
        para(i <- comeco; i >= fim; i--)faça
            escreva(i)
        fimpara
    fimse
fim
```

OU

```
início
    inteiro i, comeco, fim, fator <- 1
    escreva("Digite o valor inicial: ")
    leia(comeco)
    escreva("Digite o valor final: ")
    leia(fim)
    se(comeco > fim)então
        fator <- -1
    fimse

    para(i <- começo * fator; i <= fim * fator; i++)faça
        escreva(i * fator)
    fimpara
fim
```

Exercícios de enquanto e faça-enquanto

1.

```
início
    inteiro senha
    escreva("Digite sua senha: ")
    leia(senha)
    enquanto(senha != 123456)faça
        escreva("Senha inválida. Tente novamente:")
        leia(senha)
    fimenquanto
    escreva("Senha correta. Seja bem-vindo.")
fim
```

2.

```
início
    inteiro num
    escreva("Digite um número entre 1 e 6: ")
    leia(num)
    enquanto(num < 1 || num > 6)faça
        escreva("Valor inválido. Tente novamente:")
        leia(num)
    fimenquanto
fim
```

3.

```
início
    inteiro i
    i <- 1
    enquanto(num <= 6)faça
        escreva(i)
        i <- i + 1
    fimenquanto
fim
```

A opção mais fácil deve ser com o laço para porque toda a informação sobre o laço está concentrada em um cabeçalho. Imagine que dentro do laço há dezenas de linhas de instrução. O programador deverá buscar entre todas estas linhas onde está a variável contadora. O mesmo para sua inicialização. No nosso exemplo, ela está logo antes da instrução enquanto. Mas poderia estar várias linhas antes, dependendo do tamanho do código.

4.

```
início
    inteiro jogador1, jogador2
    escreva(“Jogador 1, digite um número: “)
    leia(jogador1)
    faça
        escreva(“Jogador 2, digite um número: “)
        leia(jogador2)
    enquanto(jogador1 != jogador2)
    escreva(“Parabéns! Você acertou!”)
fim
```

5.

```
início
    inteiro opc
    faça
        escreva(“MENU:”)
        escreva(“1-Dizer Oi”)
        escreva(“2-Dizer Olá”)
        escreva(“0-Sair”)
        leia(opc)
        escolha(opc)
            caso 1:
                escreva(“Oi!!!”)
            caso 2:
                escreva(“Olá!!!”)
        fimescolha
    enquanto(opc != 0)
fim
```

6.

```
início
    inteiro opc
    faça
        escreva(“Monstrodex: “)
        escreva(“1-Zikachu”)
        escreva(“2-Zulbassauro”)
        escreva(“3-Zharmander”)
        escreva(“0-Sair”)
        leia(opc)
        escolha(opc)
            caso 1:
                escreva(“Monstro elétrico da categoria rato”)
            caso 2:
                escreva(“Monstro de grama da categoria semente”)
            caso 3:
                escreva(“Monstro de fogo da categoria lagarto”)
            caso 0:
                escreva(“Até logo!”)
            caso padrão:
                escreva(“Monstro não cadastrado”)
                escreva(“Há 8900 monstros!”)
                escreva(“Temos que pegar!”)
        fimescolha
    enquanto(opc != 0)
fim
```

Exercícios de Teste de Mesa

1.

Instrução	**tela**	**x**	**dobro**
escreva(“Digite um valor”)	Digite um valor	?	?
leia(x)	4	4	?
dobro <- x * 2		4	8
escreva(“O resultado é “, dobro)	O resultado é 8	4	8

2.

Instrução	**tela**	**x**	**res1**	**y**	**res2**
escreva(“Digite um valor”)	Digite um valor	?	?	?	?
leia(x)	4	4	?	?	?
res1 <- x * x		4	16	?	?
escreva(x, “ ao quadrado é “, res1)	4 ao quadrado é 16	4	16	?	?
escreva(“Digite outro valor”)	Digite outro valor	4	16	?	?
leia(y)	2	4	16	2	?
res2 <- y * 10		4	16	2	20
Escreva(y, “ vezes 10 é “, res2)	2 vezes 10 é 20	4	16	2	20

3.

Instrução	**tela**	**x**	**res**
escreva(“Digite um valor”)	Digite um valor	?	?
leia(x)	4	4	?
res1 <- x * x		4	16
escreva(x, “ ao quadrado é “, res1)	4 ao quadrado é 16	4	16
escreva(“Digite outro valor”)	Digite outro valor	4	16
leia(y)	2	2	16
res2 <- y * 10		2	20
Escreva(y, “ vezes 10 é “, res2)	2 vezes 10 é 20	2	20

4.

Instrução	**tela**	**i**
escreva(“Digite um valor entre 1 e 3”)	Digite um valor entre 1 e 3	?
leia(i)	4	4
enquanto(i<1 || i>3)faça (F || V) (V)		4
escreva(“Valor inválido. Tente novamente”)	Valor inválido. Tente novamente	4
leia(i)	2	2
enquanto(i<1 || i>3)faça (F || F) (F)		2
escolha(i)		2
caso 1: (F)		2
caso 2: (V)		2
escreva(“Você procura o equilíbrio”)	Você procura o equilíbrio	2

5.

Instrução	**Tela**	**i**
escreva(“Digite um valor entre 1 e 3”)	Digite um valor entre 1 e 3	?
leia(i)	4	4
enquanto(!(i>=1 && i<=3))faça (!(V && F)) (!(F)) (V)		4
escreva(“Valor inválido. Tente novamente”)	Valor inválido. Tente novamente	4
leia(i)	2	2
enquanto(!(i>=1 && i<=3))faça (!(V && V)) (!(V)) (F)		2
escreva(“Eita!”)	Eita!	2

6.

Instrução	**Tela**	**i**
escreva(“Digite um valor entre 1 e 3”)	Digite um valor entre 1 e 3	?
leia(i)	4	4
escolha(i)		4
caso 1: (F)		4
caso 2: (F)		4
caso 3: (F)		4
caso padrão: (V)		4
escreva(“Nah... valor inválido”)	Nah... valor inválido. Tente de novo	4
enquanto(i<1 || i>3) (F || V) (V)		4
escreva(“Digite um valor entre 1 e 3”)	Digite um valor entre 1 e 3	4
leia(i)	2	2
escolha(i)		2
caso 1: (F)		2
caso 2: (V)		2
escreva(“Você procura por equilíbrio”)	Você procura por equilíbrio	2
Enquanto(i<1 || i>3) (F || F) (F)		2

7. Ambos servem para o mesmo propósito e estão corretos. Particularmente, prefiro a primeira opção (exercício 4) pois acho que seria mais simples de entender se eu fosse realizar a manutenção do código e outro programador o tivesse criado.

8.

Instrução	**Tela**	**i**
I <- 0		0
I<4 (V)		0
escreva(i)	0	0
i <- i + 1		1
i<4 (V)		1
escreva(i)	1	1
i <- i + 1		2
i<4 (V)		2
escreva(i)	2	2
i <- i + 1		3
i<4 (V)		3
escreva(i)	3	3
i <- i + 1		4
i<4 (F)		4

9.

```
    início
        real c, f
        escreva(“Digite a temperatura em Celsius: “)
        leia(c)
        f <- 9 * c / 5 + 32
        escreva(“Em Fahrenheit: “, f)
    fim
```

Instrução	**Tela**	**c**	**f**
escreva(“Digite a temperatura em Celsius: “)	Digite a temperatura em Celsius:	?	?
leia(c)	100	100	?
f <- 9 * c / 5 + 32 f <- 9 * 100 / 5 + 32 f <- 212		100	212
Escreva(“Em Fahrenheit: “, f)	Em Fahrenheit: 212	100	212

10.

Instrução	**Tela**	**x**	**i**
leia(x)	3	3	?
i <- 0		3	0
i<x (V)		3	0
escreva(“#”)	#	3	0
i<-i+1		3	1
i<x (V)		3	1
escreva(“#”)	#	3	1
i<-i+1		3	2
i<x (V)		3	2
escreva(“#”)	#	3	2
i<-i+1		3	3
i<x (F)		3	3

O algoritmo pede um valor e apresenta a mesma quantidade em #.

11.

Instrução	**Tela**	**x**	**y**	**i**	**r**
r<-1		?	?	?	1
escreva(“Digite um número: “)	Digite um número:				
leia(x)	2	2	?	?	1
Escreva(“Digite outro número: “)	Digite outro número:	2	?	?	1
leia(y)	3	2	3	?	1
i<-0		2	3	0	1
i<y (V)		2	3	0	1
r<-r*x r<-1*2		2	3	0	2
i<-i+1		2	3	1	2
i<y (V)		2	3	1	2
r<-r*x r<-2*2		2	3	1	4
i<-i+1		2	3	2	4
i<y (V)		2	3	2	4
r<-r*x r<-1*2		2	3	2	8
i<-i+1		2	3	3	8
i<y (F)		2	3	3	8
escreva(“Resultado: “, r)	Resultado: 8	2	3	3	8

O algoritmo eleva o valor de x à potência y (x^y)

Exercícios de Vetores

1.

```
    início
        inteiro a, i
        para(i <- 0; i < 6; i++)faça
            escreva(“Digite um valor: “)
            leia(a)
            escreva(“Você digitou: “, a)
        fimpara
    fim
```

OU

```
    início
        inteiro v[6], i
        para(i <- 0; i < 6; i++)faça
            escreva(“Digite um valor: “)
            leia(v[i])
            escreva(“Você digitou: “, v[i])
        fimpara
    fim
```

Não há necessidade de usar vetores se não for usar os valores em outro momento. Neste exercício, era apenas necessário apresentar o valor após digitado.

2.

```
    início
        inteiro v[6], i
        para(i <- 0; i < 6; i++)faça
            escreva(“Digite um valor: “)
            leia(v[i])
        fimpara

        para(i <- 0; i < 6; i++)faça
            escreva(v[i])
        fimpara
    fim
```

Para este exercício, como os valores precisam ser apresentados após todos terem sido previamente digitados, então usa-se o vetor para armazená-los (para serem usados em um segundo momento de apresentação).

3.

```
    início
        inteiro v[5], i, maior

        escreva(“Digite um valor: “)
        leia(v[0])
        maior <- v[0]

        para(i <- 1; i < 5; i++)faça
            escreva(“Digite um valor: “)
            leia(v[i])
            se(v[i] > maior)então
                maior <- v[i]
            fimse
        fimpara

        escreva(maior)
    fim


4.
    início
        inteiro v[5], i, menor

        escreva(“Digite um valor: “)
        leia(v[0])
        menor <- v[0]

        para(i <- 1; i < 5; i++)faça
            escreva(“Digite um valor: “)
            leia(v[i])
            se(v[i] < menor)então
                menor <- v[i]
            fimse
        fimpara

        escreva(menor)
    fim


5.
    início
        inteiro v[5], i, aux

        para(i <- 0; i < 5; i++)faça
            escreva(“Digite um valor: “)
            leia(v[i])
        fimpara


        aux <- v[0]
        v[0] <- v[4]
        v[4] <- aux

    fim
```

6.

```
    início
        inteiro vetor[6], i
        para(i <- 0; i < 6; i++)faça
            escreva(“Digite um número: “)
            leia(vetor[i])
        fimpara

        para(i <- 5; i >= 0; i--)faça
            escreva(vetor[i])
        fimpara
    fim
```

O segundo laço para pode ser substituído por:

```
        para(i <- 0; i < 6; i++)faça
            escreva(vetor[5-i])
        fimpara
```

7.

```
    início
        inteiro vetor[6], i
        para(i <- 0; i < 6; i++)faça
            escreva(“Digite um número: “)
            leia(vetor[5-i])
        fimpara
```

8.

```
    início
        inteiro vetor[6], i, aux
        para(i <- 0; i < 6; i++)faça
            escreva(“Digite um número: “)
            leia(vetor[i])
        fimpara


        para(i <- 0; i < 3; i++)faça
            aux <- vetor[i]
            vetor[i] <- vetor[5-i]
            vetor[5-i] <- aux
        fimpara
```

Desafio 1:

Precisamos representar o que significa ser mais próximo da média.
Então, vamos supor que a média entre 5 números seja 6, como nos números: 1, 2, 5, 8 e 14. Repare que a média destes números é 6 porque (1 + 2 + 5 + 8 + 14)/5 é 6.

Uma maneira de verificar quem é mais próximo da média é verificar qual número gera a menor diferença da média. De um jeito mais formal: subtraia cada elemento pela média (e transforme-o em um número positivo, caso a subtração gere um valor negativo) e escolha qual gerou o número de menor valor:
O número absoluto de 6 – **1** é 5
O número absoluto de 6 – **2** é 4
O número absoluto de 6 – **5** é 1
O número absoluto de 6 – **8** é 3
O número absoluto de 6 – **14** é 8

Como o menor resultado foi 1, então o elemento mais próximo da média é 5.

Agora precisamos criar um algoritmo que:

1. Peça os cinco valores;
2. Calcule a média;

3. Considere o primeiro elemento como sendo o mais próximo da média;
4. Calcule a diferença da média pelo primeiro elemento do vetor e considere-a como a menor diferença calculada;
5. Se o resultado for um valor negativo, converta-o para positivo;

6. Para cada outro elemento do vetor:
7. Calcule a diferença da média pelo elemento da vez;
8. Se o resultado for um valor negativo, converta-o para positivo;
9. Se o resultado calculado for menor que o resultado considerado o menor até o momento, então:
10. Considere-o como o de menor diferença até o momento;
11. Considere o elemento da vez como o mais próximo da média.

Se repetirmos os passos de 6 a 11 para todos os elementos do vetor, no final teremos qual elemento é o mais próximo da média. O algoritmo transcrito do rascunho acima é:

```
início
    real v[5], media, soma <- 0, diferenca, diferencamenor
    inteiro i, maisproximo

    para(i <- 0; i < 5; i++)faça
        escreva("Digite um valor: ")
        leia(v[i])
        soma <- soma + v[i]
    fimpara
    media <- soma / 5

    maisproximo <- 0
    diferencamenor <- media – v[0]
    se(diferencamenor < 0)então
        diferencamenor <- diferencamenor * -1
    fimse

    para(i <- 1; i < 5; i++)faça
        diferenca <- media – v[i]
        se(diferenca < 0)então
            diferenca <- diferenca * -1
        fimse
        se(diferenca < diferencamenor)então
            diferencamenor <- diferença
            maisproximo <- i
        fimse
    fimpara

    escreva("Média: ", media)
    escreva("Valor mais próximo da média: ", v[maisproximo])
fim
```

Desafio 2:

Existem diversos algoritmos de ordenação. Será usado, aqui, o bubble sort.
Este algoritmo realiza os seguintes passos:

-Peça todos os elementos e os coloque no vetor.
-Compare o primeiro elemento com o segundo. Se o primeiro for maior que o segundo, troque-os de lugar.
-Compare o segundo elemento com o terceiro. Se o segundo for maior que o terceiro, troque-os de lugar.
-Compare o terceiro elemento com o quarto. Se o terceiro for maior que o quarto, troque-os de lugar.
-Compare o quarto elemento com o quinto. Se o quarto for maior que o quinto, troque-os de lugar.
-Houve pelo menos uma troca acima? Se houve, repita tudo de novo.
-Quando não houver mais trocas, é porque o vetor está ordenado.

O algoritmo é:

```
início
    inteiro v[5], i, aux
    lógico troca
    para(i <- 0; i < 5; i++)faça
        escreva("Digite um número: ")
        leia(v[i])
    fimpara

    faça
        troca <- falso
        para(i <- 0; i < 4; i++)faça
            se(v[i] > v[i+1])então
                aux <- v[i]
                v[i] <- v[i+1]
                v[i+1] <- aux
                troca <- verdade
        fimpara
    enquanto(troca)

    para(i <- 0; i < 5; i++)faça
        escreva(v[i])
    fimpara
fim
```

Exercícios de Matrizes

1.

```
início
    inteiro v[2][2], i, j
    para(i <- 0; i < 2; i++)faça
        para(j <- 0; j < 2; j++)faça
            escreva("Digite um número: ")
            leia(v[i][j])
        fimpara
    fimpara
    para(i <- 0; i < 2; i++)faça
        para(j <- 0; j < 2; j++)faça
            escreva(V[i][j])
        fimpara
    fimpara
fim
```

2.

```
    início
        inteiro v[3][3], i, j
        para(i <- 0; i < 3; i++)faça
            para(j <- 0; j < 3; j++)faça
                escreva("Digite um número: ")
                leia(v[i][j])
            fimpara
        fimpara


        para(i <- 0; i < 3; i++)faça
            escreva("\n")
            para(j <- 0; j < 3; j++)faça
                escreva(v[i][j], " ")
            fimpara
        fimpara
    fim
```

3.

```
    início
        inteiro v[3][2], i, j
        para(i <- 0; i < 3; i++)faça
            para(j <- 0; j < 2; j++)faça
                escreva("Digite um número: ")
                leia(v[i][j])
            fimpara
        fimpara

        para(i <- 0; i < 2; i++)faça
            escreva("\n")
            para(j <- 0; j < 3; j++)faça
                escreva(v[j][i], " ")
            fimpara
        fimpara
```

4.

```
    início
        inteiro v[2][3], i, j, soma
        para(i <- 0; i < 2; i++)faça
            para(j <- 0; j < 3; j++)faça
                escreva("Digite um número: ")
                leia(v[i][j])
            fimpara
        fimpara


        para(i <- 0; i < 2; i++)faça
            soma <- 0
            para(j <- 0; j < 3; j++)faça
                soma += v[i][j]
            fimpara
            escreva("Soma: %d", soma)
        fimpara
    fim
```

5.

```
início
    inteiro mat1[3][3], mat2[3][3], mult[3][3], i, j
    para(i <- 0; i < 3; i++)faça
        para(j <- 0; j < 3; j++)faça
            escreva("Digite um número para a primeira matriz: ")
            leia(mat1[i][j])
            escreva("Digite um número para a segunda matriz: ")
            leia(mat2[i][j])
            mult[i][j] <- mat1[i][j] * mat2[i][j]
        fimpara
    fimpara


    para(i <- 0; i < 3; i++)faça
        escreva("\n")
        para(j <- 0; j < 3; j++)faça
            escreva(mult[i][j], " ")
        fimpara
    fimpara
```

Exercícios de Estruturas

1.

```
estrutura aluno
    inteiro ra
    cadeia nome, endereço
fimestrutura


início
    aluno a
    escreva("Digite o nome do aluno: ")
    leia(a.nome)
    escreva("Digite o endereço do aluno: ")
    leia(a.endereco)
    escreva("Digite o RA do aluno: ")
    leia(a.ra)

    escreva("Nome: ", a.nome)
    escreva("Endereço: ", a.endereco)
    escreva("RA: ", a.ra)
fim
```

2.

```
estrutura dvd
```

```
        cadeia titulo, genero
        inteiro prateleira
        real duracao
    fimestrutura


    início
        dvd meudvd
        escreva("Digite o título: ")
        leia(meudvd.titulo)
        escreva("Digite o genero: ")
        leia(meudvd.genero)
        escreva("Digite a prateleira: ")
        leia(meudvd.prateleira)
        escreva("Digite a duração: ")

        escreva("Título: ", meudvd.titulo)
        escreva("Gênero: ", meudvd.genero)
        escreva("Prateleira: ", meudvd.prateleira)
        escreva("Duração: ", meudvd.duracao)
    fim


3.

    estrutura monstro
        cadeia nome
        inteiro vida, ataque, defesa
    fimestrutura

    início
        monstro mon
        escreva("Digite o nome do monstrinho: ")
        leia(mon.nome)
        escreva("Digite a vida do monstrinho: ")
        leia(mon.vida)
        escreva("Digite o ataque do monstrinho: ")
        leia(mon.ataque)
        escrva("Digite a defesa do monstrinho: ")
        leia(mon.defesa)

        escreva("Nome: ", mon.nome)
        escreva("Vida: ", mon.vida)
        escreva("Ataque: ", mon.ataque)
        escreva("Defesa: ", mon.defesa)
    fim


4.
    estrutura monstro
        cadeia nome
        inteiro vida, ataque, defesa
    fimestrutura

    estrutura treinador
        cadeia nome
        inteiro idade, insígnias
    fimestrutura

    início
```

```
    treinador t
    monstro mon

    escreva("Digite o nome do monstrinho: ")
    leia(mon.nome)
    escreva("Digite a vida do monstrinho: ")
    leia(mon.vida)
    escreva("Digite o ataque do monstrinho: ")
    leia(mon.ataque)
    escrva("Digite a defesa do monstrinho: ")
    leia(mon.defesa)

    escreva("Digite o nome do treinador: ")
    leia(t.nome)
    escreva("Digite a idade do treinador: ")
    leia(t.idade)
    escreva("Digite a quantidade de insígnias do treinador: ")
    leia(t.insignias)


    escreva("Nome do monstrinho: ", mon.nome)
    escreva("Vida do monstrinho: ", mon.vida)
    escreva("Ataque do monstrinho: ", mon.ataque)
    escreva("Defesa do monstrinho: ", mon.defesa)

    escreva("Nome do treinador: ", t.nome)
    escreva("Idade do treinador: ", t.idade)
    escreva("Insígnias do treinador: ", t.insignias)
```

Exercícios de Vetores de Variáveis Compostas Heterogêneas

```
1.
    estrutura funcionario
        cadeia nome, endereço, cargo
        real salario
    fimestrutura

    início
        funcionario func[100]
        inteiro i
        para(i <- 0; i < 100; i++)faça
            escreva("Digite o nome do funcionário ", i + 1, ": ")
            leia(func[i].nome)
            escreva("Digite o endereço do funcionário ", i + 1, ": ")
            leia(func[i].endereco)
            escreva("Digite o cargo do funcionário ", i + 1, ": ")
            leia(func[i].cargo)
            escreva("Digite o salário do funcionário ", i + 1, ": ")
            leia(func[i].salario)
        fimpara


        para(i <- 0; i < 100; i++)faça
```

```
        escreva(“Nome do funcionário “, i+1, “: “, func[i].nome)
        escreva(“Endereço do funcionário “, i+1, “: “, func[i].endereco)
        escreva(“Cargo do funcionário “, i + 1, “: “, func[i].cargo)
        escreva(“Salário do funcionário “, i + 1, “: “, func[i].salario)
    fimpara
fim
```

2.

```
estrutura pocketmon
    cadeira nome
    inteiro vida, ataque, defesa
fimestrutura

início
    pocketmon pocketdex[5]
    inteiro i

    para(i <- 0; i < 5; i++)faça
        escreva(“Digite o nome do pocketmon: “)
        leia(pocketdex[i].nome)
        escreva(“Digite a vida do pocketmon: “)
        leia(pocketdex[i].vida)
        escreva(“Digite o ataque do pocketmon: “)
        leia(pocketdex[i].ataque)
        escreva(“Digite a defesa do pocketmon: “)
        leia(pocketdex[i].defesa)
    fimpara


    para(i <- 0; i < 5; i++)faça
        escreva(“Nome do pocketmon: “, pocketdex[i].nome)
        escreva(“Vida do pocketmon: “, pocketdex[i].vida)
        escreva(“Ataque do pocketmon: “, pocketdex[i].ataque)
        escreva(“Defesa do pocketmon: “, pocketdex[i].defesa)
    fimpara
fim
```

3.

```
estrutura pocketmon
    cadeira nome
    inteiro vida, ataque, defesa
fimestrutura

início
    pocketmon pocketdex[5]
    inteiro i, opc
    para(i <- 0; i < 5; i++)faça
        escreva(“Digite o nome do pocketmon: “)
        leia(pocketdex[i].nome)
        escreva(“Digite a vida do pocketmon: “)
        leia(pocketdex[i].vida)
        escreva(“Digite o ataque do pocketmon: “)
        leia(pocketdex[i].ataque)
        escreva(“Digite a defesa do pocketmon: “)
        leia(pocketdex[i].defesa)
    fimpara
    escreva(“Digite uma opção de 0 a 4:”)
    leia(opc)
```

```
        escreva(“Nome do pocketmon: “, pocketdex[opc].nome)
        escreva(“Vida do pocketmon: “, pocketdex[opc].vida)
        escreva(“Ataque do pocketmon: “, pocketdex[opc].ataque)
        escreva(“Defesa do pocketmon: “, pocketdex[opc].defesa)
    fim
```

4.

```
    estrutura pocketmon
        cadeira nome
        inteiro vida, ataque, defesa
    fimestrutura

    início
        pocketmon pocketdex[5]
        inteiro i, opc
        para(i <- 0; i < 5; i++)faça
            escreva(“Digite o nome do pocketmon: “)
            leia(pocketdex[i].nome)
            escreva(“Digite a vida do pocketmon: “)
            leia(pocketdex[i].vida)
            escreva(“Digite o ataque do pocketmon: “)
            leia(pocketdex[i].ataque)
            escreva(“Digite a defesa do pocketmon: “)
            leia(pocketdex[i].defesa)
        fimpara

        faça
            escreva(“Digite uma opção de 0 a 4:”)
            leia(opc)
            se(opc >= 0 && opc <=4)então
                escreva(“Nome do pocketmon: “, pocketdex[opc].nome)
                escreva(“Vida do pocketmon: “, pocketdex[opc].vida)
                escreva(“Ataque do pocketmon: “, pocketdex[opc].ataque)
                escreva(“Defesa do pocketmon: “, pocketdex[opc].defesa)
            fimse
        enquanto(opc >= 0 && opc <= 4)
    fim
```

5.

```
    estrutura pocketmon
        cadeira nome
        inteiro vida, ataque, defesa
    fimestrutura

    início
        pocketmon pocketdex[5]
        inteiro i, opc
        para(i <- 0; i < 5; i++)faça
            escreva(“Digite o nome do pocketmon: “)
            leia(pocketdex[i].nome)
            escreva(“Digite a vida do pocketmon: “)
            leia(pocketdex[i].vida)
            escreva(“Digite o ataque do pocketmon: “)
            leia(pocketdex[i].ataque)
            escreva(“Digite a defesa do pocketmon: “)
            leia(pocketdex[i].defesa)
        fimpara
        faça
```

```
            escreva(“1-Alterar dados do pocketmon”)
            escreva(“2-Ver dados do pocketmon”)
            escreva(“Outro valor para sair”)
            leia(opc)
            escolha(opc)
                caso 1:
                    faça
                        escreva(“Digite qual pocketmon deseja alterar:”)
                        leia(i)
                    enquanto(i < 0 || i > 4)
                    escreva(“Digite o nome do pocketmon: “)
                    leia(pocketdex[i].nome)
                    escreva(“Digite a vida do pocketmon: “)
                    leia(pocketdex[i].vida)
                    escreva(“Digite o ataque do pocketmon: “)
                    leia(pocketdex[i].ataque)
                    escreva(“Digite a defesa do pocketmon: “)
                    leia(pocketdex[i].defesa)
                caso 2:
                    faça
                        escreva(“Digite qual pocketmon deseja ver:”)
                        leia(i)
                    enquanto(i < 0 || i > 4)
                    escreva(“Nome do pocketmon: “, pocketdex[opc].nome)
                    escreva(“Vida do pocketmon: “, pocketdex[opc].vida)
                    escreva(“Ataque do pocketmon: “, pocketdex[opc].ataque)
                    escreva(“Defesa do pocketmon: “, pocketdex[opc].defesa)
            fimescolha
        enquanto(opc == 1 || opc == 2)
    fim
```

Exercícios de Estruturas com vetores como Campos

```
1.
    estrutura pessoa
        cadeia nome, endereco
        inteiro rifa[5]
    fimestrutura


    início
        pessoa p
        inteiro i
        escreva(“Digite o nome da pessoa: “)
        leia(p.nome)
        escreva(“Digite o endereço da pessoa: “)
        leia(p.endereco)
        para(i <- 0; i < 5; i++)faça
            escreva(“Digite um número de rifa: “)
            leia(p.rifa[i])
        fimpara


        escreva(“Nome: “, p.nome)
```

```
        escreva(“Endereço: “, p.endereco)
        para(i <- 0; i < 5; i++)faça
            escreva(“Número da rifa: “, p.rifa[i])
        fimpara
    fim


2.
    estrutura cliente
        cadeia nome, endereco, cor[4]
    fimestrutura


    início
        cliente c
        inteiro i
        escreva(“Digite o nome do cliente: “)
        leia(c.nome)
        escreva(“Digite o endereço do cliente: “)
        leia(c.endereco)
        para(i <- 0; i < 4; i++)faça
            escreva(“Digite uma cor: “)
            leia(c.cor[i])
        fimpara

        escreva(“Nome: “, c.nome)
        escreva(“Endereço: “, c.endereco)
        para(i <- 0; i < 4; i++)faça
            escreva(“Cor: “, c.cor[i])
        fimpara
    fim


3.
    estrutura personagem
        cadeia nome, habilidade[3]
        inteiro idade, vida, ataque, defesa
    fimestrutura

    início
        personagem p
        inteiro i
        escreva(“Digite o nome do personagem: “)
        leia(p.nome)
        escreva(“Digite a idade do personagem: “)
        leia(p.idade)
        escreva(“Digite a vida do personagem: “)
        leia(p.vida)
        escreva(“Digite o ataque do personagem: “)
        leia(p.ataque)
        escreva(“Digite a defesa do personagem: “)
        leia(p.defesa)
        para(i <- 0; i < 3; i++)faça
            escreva(“Digite a habilidade: “)
            leia(p.habilidade[i])
        fimpara

        escreva(“Nome do personagem: “, p.nome)
        escreva(“Idade do personagem: “, p.idade)
        escreva(“Vida do personagem: “, p.vida)
```

```
        escreva(“Ataque do personagem: “, p.ataque)
        escreva(“Defesa do personagem: “, p.defesa)
        para(i <- 0; i < 3; i++)faça
            escreva(“Habilidade: “, p.habilidade[i])
        fimpara
    fim
```

4.

```
    estrutura personagem
        cadeia nome, habilidade[3]
        inteiro idade, vida, ataque, defesa
    fimestrutura

    início
        personagem p[5]
        inteiro i, j

        para(i <- 0; i < 5; i++)faça
            escreva(“Digite o nome do personagem: “)
            leia(p[i].nome)
            escreva(“Digite a idade do personagem: “)
            leia(p[i].idade)
            escreva(“Digite a vida do personagem: “)
            leia(p[i].vida)
            escreva(“Digite o ataque do personagem: “)
            leia(p[i].ataque)
            escreva(“Digite a defesa do personagem: “)
            leia(p[i].defesa)
            para(j <- 0; j < 3; j++)faça
                escreva(“Digite a habilidade: “)
                leia(p[i].habilidade[j])
            fimpara
        fimpara

        para(i <- 0; i < 5; i++)faça
            escreva(“Nome do personagem: “, p[i].nome)
            escreva(“Idade do personagem: “, p[i].idade)
            escreva(“Vida do personagem: “, p[i].vida)
            escreva(“Ataque do personagem: “, p[i].ataque)
            escreva(“Defesa do personagem: “, p[i].defesa)
            para(j <- 0; j < 3; j++)faça
                escreva(“Habilidade: “, p[i].habilidade[j])
            fimpara
        fimpara
    fim
```

Exercícios de Variável Composta Heterogênea dentro de Variável Composta Heterogênea

1.

```
estrutura pessoa
    cadeia nome, endereco
    inteiro cpf
fimestrutura

estrutura funcionario
    pessoa p
    real salario
fimestrutura

estrutura aluno
    pessoa p
    inteiro ra
fimestrutura

início
    funcionário f
    aluno a

    escreva("Digite o nome do funcionário: ")
    leia(f.p.nome)
    escreva("Digite o endereço do funcionário: ")
    leia(f.p.endereco)
    escreva("Digite o CPF do funcionário: ")
    leia(f.p.cpf)
    escreva("Digite o salário do funcionário")
    leia(f.salario)

    escreva("Digite o nome do aluno: ")
    leia(a.p.nome)
    escreva("Digite o endereço do aluno: ")
    leia(a.p.endereco)
    escreva("Digite o CPF do aluno: ")
    leia(a.p.cpf)
    escreva("Digite o RA do aluno: ")
    leia(a.ra)

    escreva("Nome do funcionário: ", f.p.nome)
    escreva("Endereço do funcionário: ", f.p.endereco)
    escreva("CPF do funcionário: ", f.p.cpf)
    escreva("Salário do funcionário: ", f.salario)

    escreva("Nome do aluno: ", a.p.nome)
    escreva("Endereço do aluno: ", a.p.endereco)
    escreva("CPF do aluno: ", a.p.cpf)
    escreva("Salário do aluno: ", a.ra)
fim
```

2.

```
estrutura pocketmon
    cadeia espécie
    inteiro vida, ataque, defesa
fimestrutura
```

```
    estrutura treinador
        cadeia nome
        inteiro idade
        pocketmon monstro
    fimestrutura

    início
        treinador t
        escreva(“Digite o nome do treinador: “)
        leia(t.nome)
        escreva(“Digite a idade do treinador: “)
        leia(t.idade)
        escreva(“Digite a espécie do pocketmon do treinador: “)
        leia(t.monstro.especie)
        escreva(“Digite a vida do pocketmon do treinador: “)
        leia(t.monstro.vida)
        escreva(“Digite o ataque do pocketmon do treinador: “)
        leia(t.monstro.ataque)
        escreva(“Digite a defesa do pocketmon do treinador: “)
        leia(t.monstro.defesa)

        escreva(“Nome do treinador: “, t.nome)
        escreva(“Idade do treinador: “, t.idade)
        escreva(“Espécie do monstro do treinador: “, t.monstro.especie)
        escreva(“Vida do monstro do treinador: “, t.monstro.vida)
        escreva(“Ataque do monstro do treinador: “, t.monstro.ataque)
        escreva(“Defesa do monstro do treinador: “, t.monstro.defesa)
    fim
```

3.

```
    estrutura pocketmon
        cadeia espécie
        inteiro vida, ataque, defesa
    fimestrutura

    estrutura treinador
        cadeia nome
        inteiro idade
        pocketmon monstro[5]
    fimestrutura

    início
        treinador t
        inteiro i
        escreva(“Digite o nome do treinador: “)
        leia(t.nome)
        escreva(“Digite a idade do treinador: “)
        leia(t.idade)
        para(i <- 0; i < 5; i++)faça
            escreva(“Digite a espécie do pocketmon do treinador: “)
            leia(t.monstro[i].especie)
            escreva(“Digite a vida do pocketmon do treinador: “)
            leia(t.monstro[i].vida)
            escreva(“Digite o ataque do pocketmon do treinador: “)
            leia(t.monstro[i].ataque)
            escreva(“Digite a defesa do pocketmon do treinador: “)
            leia(t.monstro[i].defesa)
        fimpara
```

```
        escreva(“Nome do treinador: “, t.nome)
        escreva(“Idade do treinador: “, t.idade)
        para(i <- 0; i < 5; i++)faça
          escreva(“Espécie do monstro do treinador: “, t.monstro[i].especie)
          escreva(“Vida do monstro do treinador: “, t.monstro[i].vida)
          escreva(“Ataque do monstro do treinador: “, t.monstro[i].ataque)
          escreva(“Defesa do monstro do treinador: “, t.monstro[i].defesa)
        fimpara
    fim
```

4.

```
    estrutura pocketmon
        cadeia espécie
        inteiro vida, ataque, defesa
    fimestrutura

    estrutura treinador
        cadeia nome
        inteiro idade
        pocketmon monstro[5]
    fimestrutura

    início
        treinador t[10]
        inteiro i, j

        para(i <- 0; i < 10; i++)faça
            escreva(“Digite o nome do treinador: “)
            leia(t[i].nome)
            escreva(“Digite a idade do treinador: “)
            leia(t[i].idade)
            para(j <- 0; j < 5; j++)faça
                escreva(“Digite a espécie do pocketmon do treinador: “)
                leia(t[i].monstro[j].especie)
                escreva(“Digite a vida do pocketmon do treinador: “)
                leia(t[i].monstro[j].vida)
                escreva(“Digite o ataque do pocketmon do treinador: “)
                leia(t[i].monstro[j].ataque)
                escreva(“Digite a defesa do pocketmon do treinador: “)
                leia(t[i].monstro[j].defesa)
            fimpara
        fimpara

        para(i <- 0; i < 10; i++)faça
            escreva(“Nome do treinador: “, t[i].nome)
            escreva(“Idade do treinador: “, t[i].idade)
            para(j <- 0; j < 5; j++)faça
                escreva(“Espécie do monstro: “, t[i]monstro[j].especie)
                escreva(“Vida do monstro: “, t[i].monstro[j].vida)
                escreva(“Ataque do monstro: “, t[i].monstro[j].ataque)
                escreva(“Defesa do monstro: “, t[i].monstro[j].defesa)
            fimpara
        fimpara
    fim
```

Exercícios Básicos sobre Funções

1.

```
real lerTemperatura()
    real cel
    escreva(“Digite a temperatura em Celsius: “)
    leia(cel)
    retornar cel
fim

real calcularFahrenheit(real cel)
    real fah
    fah <- 9 * cel / 5 + 32
    retornar fah
fim

nada apresentar(real fah)
    escreva(“Temperatura em Fahrenheit: “, fah)
fim

início
    real c, f
    c <- lerTemperatura()
    f <- calcularFahrenheit(c)
    apresentar(f)
fim
```

2.

```
real lerTemperatura()
    real cel
    escreva(“Digite a temperatura em Celsius: “)
    leia(cel)
    retornar cel
fim

real calcularFahrenheit(real cel)
    real fah
    fah <- 9 * cel / 5 + 32
    retornar fah
fim

nada apresentar(real cel, real fah)
    escreva(“Temperatura em Celsius: “, cel)
    escreva(“Temperatura em Fahrenheit: “, fah)
fim

início
    real c, f
    c <- lerTemperatura()
    f <- calcularFahrenheit(c)
    apresentar(c, f)
fim
```

3.

```
real lerTemperatura()
    real fah
    escreva(“Digite a temperatura em Fahrenheit: “)
```

```
        leia(fah)
        retornar fah
    fim

    real calcularCelsius(real fah)
        real cel
        cel <- 5 * (fah – 32) / 9
        retornar cel
    fim

    nada apresentar(real cel)
        escreva(“Temperatura em Celsius: “, cel)
    fim

    início
        real c, f
        f <- lerTemperatura()
        c <- calcularCelsius(f)
        apresentar(c)
    fim
```

4.

```
    real lerTemperatura()
        real fah
        escreva(“Digite a temperatura em Fahrenheit: “)
        leia(fah)
        retornar fah
    fim

    real calcularCelsius(real fah)
        real cel
        cel <- 5 * (fah – 32) / 9
        retornar cel
    fim

    nada apresentar(real cel, real fah)
        escreva(“Temperatura em Celsius: “, cel)
        escreva(“Temperatura em Fahrenheit: “, fah)
    fim

    início
        real c, f
        f <- lerTemperatura()
        c <- calcularCelsius(f)
        apresentar(c, f)
    fim
```

5.

```
    real lerNota()
        real nota
        escreva(“Digite uma nota: “)
```

```
        leia(nota)
        retornar nota
    fim

    real calcularMedia(real n1, real n2)
        real media
        media <- (n1 + n2)/2
        retornar media
    fim

    nada apresentar(real media)
        escreva("Média: ", media)
    fim

    início
        real n1, n2, media
        n1 <- lerNota()
        n2 <- lerNota()
        media <- calcularMedia(n1, n2)
        apresentar(media)
    fim
```

6.

```
    real lerNota()
        real nota
        escreva("Digite uma nota: ")
        leia(nota)
        retornar nota
    fim

    real calcularMedia(real n1, real n2)
        real media
        media <- (n1 + n2)/2
        retornar media
    fim

    nada apresentar(real media)
        se(media >= 6)então
            escreva("Aprovado")
        senão
            escreva("Recuperação")
        fimse
    fim

    início
        real n1, n2, media
        n1 <- lerNota()
        n2 <- lerNota()
        media <- calcularMedia(n1, n2)
        apresentar(media)
    fim
```

7.

```
    inteiro lerNumero()
        inteiro num
```

```
        escreva(“Digite um número: “)
        leia(num)
        retornar num
    fim

    nada apresentar(inteiro num)
        se(num%2 == 0)então
            escreva(“Par”)
        senão
            escreva(“Ímpar”)
        fimse
    fim

    início
        real num
        num <- lerNumero()
        apresentar(num)
    fim
```

8.

```
    real lerNumero()
        real num
        escreva(“Digite um número: “)
        leia(num)
        retornar num
    fim

    nada maior(real num1, real num2)
        se(num1 > num2)então
            escreva(“Maior: ”, num1)
        senão
            escreva(“Maior: ”, num2)
        fimse
    fim

    início
        real num1, num2
        num1 <- lerNumero()
        num2 <- lerNumero()
        maior(num1, num2)
    fim
```

9.

```
    inteiro lerNumero()
        inteiro num
        escreva(“Digite um número: “)
```

```
        leia(num)
        retornar num
    fim

    cadeia parOuImpar(inteiro num)
        se(num%2 == 0)então
            retornar “Par”
        senão
            retornar “Ímpar”
        fim
    fim

    nada apresentar(cadeia res)
        escreva(“O número é “, res)
    fim

    início
        inteiro num
        cadeia res
        num <- lerNumero()
        res <- parOuImpar(num)
        apresentar(res)
    fim
```

10.

```
    real lerNumero()
        real num
        escreva(“Digite um número: “)
        leia(num)
        retornar num
    fim

    real maior(real num1, real num2)
        se(num1 > num2)então
            retornar num1
        senão
            retornar num2
        fimse
    fim

    nada apresentar(real m)
        escreva(“O número maior é “, m)
    fim

    início
        real num1, num2, m
        num1 <- lerNumero()
        num2 <- lerNumero()
        m <- maior(num1, num2)
        apresentar(m)
    fim
```

Exercícios Não Tão Básicos sobre Funções

1.

```
inteiro apresentarMenu()
    inteiro opc
    escreva(“MENU:”)
    escreva(“1-Converter quilômetros para milhas”)
    escreva(“2-Converter milhas para quilômetros”)
    escreva(“0-SAIR”)
    leia(opc)
    retornar opc
fim

real lerDistancia()
    real dist
    escreva(“Escreva a distância: ”)
    leia(dist)
    retornar dist
fim

real deKmparaMi(real km)
    real mi
    mi <- km * 0.62137
    retornar mi
fim

real deMiparaKm(real mi)
    real km
    km <- mi / 0.62137
    retornar km
fim

nada apresentar(real km, real mi)
    escreva(“Em Km: “, km)
    escreva(“Em Mi: “, mi)
fim

início
    real mi, km
    inteiro opc

    faça
        opc <- apresentarMenu()
        escolha(opc)
            caso 1:
                km <- lerDistancia()
                mi <- deKmparaMi(km)
                apresentar(km, mi)
            caso 2:
                mi <- lerDistancia()
                km <- deMiparaKm(mi)
                apresentar(km, mi)
        fimescolha
    enquanto(opc != 0)
fim
```

2.

```
inteiro apresentarMenu()
    inteiro opc
```

```
        escreva(“MENU:”)
        escreva(“1-Converter quilômetros para milhas”)
        escreva(“2-Converter milhas para quilômetros”)
        escreva(“0-SAIR”)
        leia(opc)
        retornar opc
    fim

    real lerDistancia()
        real dist
        escreva(“Escreva a distância: ”)
        leia(dist)
        retornar dist
    fim

    real conversao(real original, inteiro opc)
        se(opc == 1)
            retornar original * 0.62137
        senão
            retornar original / 0.62137
    fim

    nada apresentar(real km, real mi)
        escreva(“Em Km: “, km)
        escreva(“Em Mi: “, mi)
    fim

    início
        real mi, km
        inteiro opc
        faça
            opc <- apresentarMenu()
            escolha(opc)
                caso 1:
                    km <- lerDistancia()
                    mi <- converter(km, 1)
                    apresentar(km, mi)
                caso 2:
                    mi <- lerDistancia()
                    km <- converter(mi, 2)
                    apresentar(km, mi)
            fimescolha
        enquanto(opc != 0)
    fim
```

3.

```
    estrutura pocketmon
        cadeia nome
        inteiro vida, ataque, defesa
    fimestrutura




    inteiro menu()
        inteiro opc
```

```
    escreva("MENU:")
    escreva("1- Cadastrar pocketmon no pocketdex")
    escreva("2- Apresentar pocketmon")
    escreva("0- Sair")
    leia(opc)
    retornar opc
fim

pocketmon cadastrar()
    pocketmon p
    escreva("Digite o nome do pocketmon: ")
    leia(p.nome)
    escreva("Digite a vida do pocketmon: ")
    leia(p.vida)
    escreva("Digite o ataque do pocketmon: ")
    leia(p.ataque)
    escreva("Digite a defesa do pocketmon: ")
    leia(p.defesa)
    retornar p
fim

inteiro qualPocketmon()
    inteiro opc
    escreva("Digite qual pocketmon deseja ver: ")
    leia(opc)
    enquanto(opc < 0 || opc > 4)faça
        escreva("O valor precisa estar entre 0 e 4")
        leia(opc)
    fimenquanto
    retornar opc
fim

nada apresentarPocketmon(inteiro qual, pocketmon p[])
    escreva("Nome do pocketmon: ", p.nome)
    escreva("Vida do pocketmon: ", p.vida)
    escreva("Ataque do pocketmon: ", p.ataque)
    escreva("Defesa do pocketmon: ", p.defesa)
fim

início
    pocketmon pocketdex[5]
    inteiro opc,qual, i <- 0

    faça
        opc <- menu()
        escolha(opc)
            caso 1:
                se(i<5)então
                    pocketdex[i++] <- cadastrar()
                fimse
            caso 2:
                qual <- qualPocketmon()
                apresentarPocketmon(qual, pocketdex)
        fimescolha
    enquanto(opc != 0)
fim
```

Exercícios de Funções que Chamam Funções

1.

```
inteiro menuPrincipal()
    inteiro opc
    escreva(“1- Biologia”)
    escreva(“2- Geografia”)
    leia(opc)
    retornar opc
fim

inteiro menuBiologia()
    inteiro opc
    escreva(“1- Exercitar sobre artrópodes”)
    escreva(“2- Exercitar sobre vertebrados”)
    leia(opc)
    retornar opc
fim

inteiro perguntaArtropode()
    inteiro opc
    escreva(“Quantas patas tem um inseto?”)
    leia(opc)
    retornar opc
fim

inteiro perguntaVertebrado()
    inteiro opc
    escreva(“Qual destes animais não pertence aos vertebrados?”)
    escreva(“1- Mamífero; 2- Réptil; 3- Ave; 4- Molusco”)
    leia(opc)
    retornar opc
fim

inteiro menuGeografia()
    inteiro opc
    escreva(“1- Exercitar sobre países”)
    escreva(“2- Exercitar sobre continentes”)
    leia(opc)
    retornar opc
fim

inteiro perguntaPaises()
    inteiro opc
    escreva(“Qual destes países não faz fronteira com o Brasil?”)
    escreva(“1- Colômbia; 2- Chile; 3- Venezuela”)
    leia(opc)
    retornar opc
fim

inteiro perguntaContinentes()
    inteiro opc
    escreva(“Qual destes não é um continente?”)
    escreva(“1- Ásia; 2- África; 3- Atlântida”)
    leia(opc)
    retornar opc
fim

nada certoOuErrado(inteiro resposta, inteiro gabarito)
    se(resposta == gabarito)então
        escreva(“Correto”)
```

```
        senão
            escreva(“Errado”)
    fim


    nada gerenciarBiologia()
        inteiro opc, resposta
        opc <- menuBiologia()
        escolha(opc)
            caso 1:
                resposta <- perguntaArtropode()
                certoOuErrado(resposta, 6)
            caso 2:
                resposta <- perguntaVertebrado()
                certoOuErrado(resposta, 4)
        fimescolha
    fim

    nada gerenciarGeografia()
        inteiro opc, resposta
        opc <- menuGeografia()
        escolha(opc)
            caso 1:
                resposta <- perguntaPaises()
                certoOuErrado(resposta, 2)
            caso 2:
                resposta <- perguntaContinentes()
                certoOuErrado(resposta, 3)
        fimescolha
    fim


    início
        inteiro opc
        opc <- menuPrincipal()
        escolha(opc)
            caso 1:
                gerenciarBiologia()
            caso 2:
                gerenciarGeografia()
        fimescolha
    fim


OU


    inteiro menuGenerico(cadeia msg1, cadeia msg2)
        inteiro opc
        escreva(msg1)
        escreva(msg2)
        leia(opc)
        retornar opc
    fim



    nada certoOuErrado(inteiro resposta, inteiro gabarito)
        se(resposta == gabarito)então
            escreva(“Correto”)
```

```
        senão
            escreva("Errado")
    fim


    nada gerenciarBiologia()
        inteiro opc, resposta
        opc <- menuGenerico("1- Exercitar sobre artrópodes",
                            "2- Exercitar sobre vertebrados")
        escolha(opc)
            caso 1:
              resposta<-menuGenerico("Quantas patas tem um inseto?","")
              certoOuErrado(resposta, 6)
            caso 2:
              resposta<-menuGenerico("Qual destes animais não pertence aos
vertebrados?", "1- Mamífero; 2- Réptil; 3- Ave; 4- Molusco")
              certoOuErrado(resposta, 4)
        fimescolha
    fim

    nada gerenciarGeografia()
        inteiro opc, resposta
        opc <- menuGenerico("1- Exercitar sobre países",
                           "2- Exercitar sobre continentes")
        escolha(opc)
            caso 1:
                resposta <- menuGenerico("Qual destes países não faz
fronteira com o Brasil?", "1- Colômbia; 2- Chile; 3- Venezuela")
                certoOuErrado(resposta, 2)
            caso 2:
                resposta <- menuGenerico("Qual destes não é um continente?",
                                         "1- Ásia; 2- África; 3- Atlântida")
                certoOuErrado(resposta, 3)
        fimescolha
    fim

    início
        inteiro opc
        opc <- menuGenerico("1- Biologia", "2- Geografia")
        escolha(opc)
            caso 1:
                gerenciarBiologia()
            caso 2:
                gerenciarGeografia()
        fimescolha
    fim




OU




inteiro menuGenerico(cadeia msg1, cadeia msg2)
        inteiro opc
        escreva(msg1)
```

```
    escreva(msg2)
    leia(opc)
    retornar opc
fim

nada gerenciarGenerico(cadeia mnu1, cadeia mnu2, cadeia msg1,
                       cadeia msg2, cadeia msg3, cadeia msg4,
                       inteiro gab1, inteiro gab2)
    inteiro opc, resposta
    opc <- menuGenerico(mnu1, mnu2)
    escolha(opc)
        caso 1:
          resposta<-menuGenerico(msg1, msg2)
          certoOuErrado(resposta, gab1)
        caso 2:
          resposta<-menuGenerico(msg3, msg4)
          certoOuErrado(resposta, gab2)
    fimescolha
fim

nada certoOuErrado(inteiro resposta, inteiro gabarito)
    se(resposta == gabarito)então
        escreva("Correto")
    senão
        escreva("Errado")
fim

início
    inteiro opc
    opc <- menuGenerico("1- Biologia", "2- Geografia")
    escolha(opc)
        caso 1:
            gerenciarGenerico("1- Exercitar sobre artrópodes",
                    "2- Exercitar sobre vertebrados",
                    "Quantas patas tem um inseto?",
                    "",
                    "Qual destes animais não pertence aos vertebrados?",
                    "1- Mamífero; 2- Réptil; 3- Ave; 4- Molusco",
                    6, 4)
        caso 2:
            gerenciarGenerico("1- Exercitar sobre países",
                   "2- Exercitar sobre continentes",
                   "Qual destes países não faz fronteira com o Brasil?",
                   "1- Colômbia; 2- Chile; 3- Venezuela",
                   "Qual destes não é um continente?",
                   "1- Ásia; 2- África; 3- Atlântida",
                   2, 4)
    fimescolha
fim
```

OBS: qual das três versões você prefere? Na primeira, criamos uma função específica para cada escolha. Na segunda, procuramos reusar algumas funções. E, na terceira, reusamos praticamente todas as funções que podem ser reusadas.

Quanto mais tentamos reusar estas funções, mais genéricas temos que torná-las e mais o algoritmo será dependente dos argumentos que passamos para as funções (é um código em que nosso foco está mais nos dados que iremos enviar para as funções). Na terceira versão, por exemplo, a função gerenciarGenerico() precisa de 8 parâmetros!

O que prefere?

Particularmente, minha resposta é: depende do contexto. Se o algoritmo for exatamente este, eu prefiro a segunda versão porque ela é um meio termo. Não tenho que me preocupar tanto em como posso reusar um código. E aqueles com que me preocupei precisam de poucos parâmetros. Para mim, o código fica enxuto e fácil o suficiente para compreender se fosse feito por terceiros (ou por mim, mas se eu ficasse um tempo afastado deste código e precisasse fazer a manutenção).

Já, se eu estivesse fazendo um algoritmo que pudesse variar as perguntas e menus, onde as mensagens estivessem em um banco de dados e pudessem ser escolhidas aleatoriamente por um outro trecho de código, então é interessante criar funções que sejam o mais genéricas possível e que dependam principalmente dos dados (argumentos) que iremos passar.