14

Lidando com arrays

Foge Foge, um jogo baseado no Pacman

Definindo a base do jogo e o mapa

Queremos desenvolver um jogo baseado no famoso Pacman, onde o herói foge de fantasmas que assombram sua vida. Nosso jogo, brasileiro, se chama foge-foge. Como de costume, começamos o jogo pedindo o nome de nosso usuário. Já sabemos que devemos isolar a parte de interface com o usuário, portanto criamos nosso `ui.rb` :

```
def da_boas_vindas
    puts "Bem vindo ao Foge-foge"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end
```

A lógica do jogo em `fogefoge.rb` :

```
require_relative 'ui'

def joga(nome)
    # nosso jogo aqui
end

def inicia_fogefoge
    nome = da_boas_vindas
    joga nome
end
```

E nosso `main.rb` , que inicia o jogo:

```
require_relative 'logic'

inicia_fogefoge
```

Precisamos definir o nosso mapa. Nosso jogo possui muros onde o herói não pode andar, fantasmas que caçam nosso jogador, o jogador em si e caminhos onde ele pode passar - trechos sem nada.

Chamando o muro de X o mapa a seguir é um exemplo de quatro linhas e quatro colunas:

```
XXXX
X  X
X  X
XXXX
```

Se chamarmos o herói de H, o mapa a seguir é um mapa de quatro linhas e quatro colunas, com o herói no meio:

```
XXXX
XH X
X  X
XXXX
```

Já chamando de `F` um fantasma, temos o seguinte mapa - difícil de ganhar - válido:

```
XXXX
XH X
X FX
XXXX
```

Usando essas definições, o primeiro mapa válido que utilizaremos é um com um único fantasma, criamos o arquivo `mapa1.txt` a seguir. Não esqueça de colocar os espaços em branco nas linhas cuja borda é um espaço em branco.

```
XXXXX
X H X
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
```

Um mapa mais interessante com dois fantasmas, `mapa2.txt` seria:

```
XXXXXXXXX
X H     X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXXF  F X
XXX XXX X
```

```
XXX XXX X
XXX     X
```

Isto é, estamos usando letras para representar um mapa que nós, como seres humanos, conseguimos compreender. Poderíamos usar 0s e 1s, mas seria mais trabalhoso para entendermos o que estamos fazendo nesse instante, o que não é nosso foco.

Array de array: matriz

Baseados então no segundo mapa, podemos escrever uma função de lógica (`logic.rb`) que lê tudo em uma `string` e quebra cada linha em um array de `string` s:

```
def le_mapa(numero)
    texto = File.read("mapa#{numero}.txt")
    mapa = texto.split("\n")
end
```

Mas que porcaria é um `"mapa#{numero}.txt"` . Sim, eu sou chato e gosto de dar nome a muitos bois. Nesse caso específico, se uma `string` tem um significado (uma semântica) outro que não seja meramente um texto, gosto de deixar isso bem claro. Parece ser um gosto bobo, mas deixar claro o que algo é facilita muito sua compreensão. Por exemplo qual o que significa uma variável ::pessoa::? E uma variável ::usuarioLogado::? Ambos são pessoas, mas uma é claramente algo especial para meu programa, ela possui um significado para mim. Portanto ::extract variable:: nele:

```
def le_mapa(numero)
    arquivo = "mapa#{numero}.txt"
    texto = File.read(arquivo)
    mapa = texto.split("\n")
end
```

Carregamos o mapa durante o jogo:

```
def joga(nome)
    mapa = le_mapa(1)
end
```

E mostramos o mesmo:

```
mapa = le_mapa(1)
desenha mapa
```

Claro, a cada nova rodada o usuário poderá andar. Para a direita, esquerda, cima ou baixo. Portanto devemos sempre perguntar para onde ele quer ir, em nosso `ui.rb` :

```
def pede_movimento
    puts "Para onde deseja ir?"
    movimento = gets.strip
end
```

E durante o jogo fazemos um laço, mostrando o mapa e perguntando:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
    end
end
```

Precisamos implementar a função `desenha` em nosso `ui.rb`, que é bem simples:

```
def desenha(mapa)
    puts mapa
end
```

Até aqui usamos somente recursos que já conhecíamos para ler o mapa e o movimento de nosso jogador. Testamos o jogo e a leitura está adequada:

```
Bem vindo ao jogo do Pacman
...
Começaremos o jogo para você, Guilherme
XXXXX
X H X
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
Para onde deseja ir?
```

Chamamos um array de arrays, um array de 2 dimensões, de uma matriz. Por mais que na matemática um array de uma dimensão (comumente chamado vetor) seja também chamado de matriz, no desenvolvimento de software costumamos utilizar o termo ::array:: para uma dimensão e ::matriz:: para duas ou mais dimensões. Mas tome cuidado... o que temos é um array de array ou um array de `String`?

Movimento

Nossa primeira funcionalidade do jogo envolve permitir ao jogador movimentar seu personagem. Queremos utilizar as mesmas teclas que jogos modernos utilizam: `WASD` para cima, esquerda, baixo e direita respectivamente.

Mas como encontrar o jogador no mapa? Vamos varrer o mapa e encontrar ele:

```
def encontra_jogador(mapa)
    for linha in 0..(mapa.size-1)
        if mapa[linha].include? "H"
            # achei!
        end
    end
    # não achei!
end
```

Caso encontremos o caracter `H` dentro da linha do mapa, precisamos procurar em qual coluna, qual posição, ele está lá dentro:

```
def encontra_jogador(mapa)
    for linha = 0..(mapa.size-1)
        if mapa[linha].include? "H"
            for coluna = 0..(mapa[linha].size-1)
                if mapa[linha][coluna] == "H"
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

Refatorando

Já vi código feio, mas nosso código está horrendo. Cada linha faz duas, três, quatro ou até mesmo cinco coisas (`if` , `[]` , `[]` , `==` , definir o caracter mágico `"H"` na mesma linha).

Comecemos extraindo o `"H"` , que é o herói:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        if mapa[linha].include? caracter_do_heroi
            for coluna = 0..(mapa[linha].size-1)
                if mapa[linha][coluna] == caracter_do_heroi
                    # achei!
                end
            end
        end
    end
```

```
    # não achei!
end
```

Agora podemos extrair a `linha_atual`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        if linha_atual.include? caracter_do_heroi
            for coluna = 0..(linha_atual.size-1)
                if linha_atual[coluna] == caracter_do_heroi
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

Extraímos as condições:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        heroi_esta_nessa_linha = linha_atual.include? caracter_do_heroi
        if heroi_esta_nessa_linha
            for coluna = 0..(linha_atual.size-1)
                heroi_esta_aqui = linha_atual[coluna] == caracter_do_heroi
                if heroi_esta_aqui
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

Agora paramos para pensar um pouco mais. Como funciona o método `include?`? Ele passa caractere a caractere, verificando se o `"H"` está lá. Se é isso que ele faz, e refazemos isso dentro do `if`, não precisamos do `if`, afinal já fazemos o nosso próprio `for` para encontrar a posição adequada. Com isso a variável `heroi_esta_nessa_linha` vai embora:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        for coluna = 0..(linha_atual.size-1)
```

```
            heroi_esta_aqui = linha_atual[coluna] == caracter_do_heroi
            if heroi_esta_aqui
                # achei!
            end
        end
    end
    # não achei!
end
```

O vazio, o nulo

Se já existe um método como o ::include?:: que diz se um caractere (ou ::String::) está dentro de uma ::String::, será que não existe um que já diz em qual posição ele aparece pela primeira vez? Procuramos na documentação e encontramos o método ::index:: que faz exatamente isso: devolve a posição onde o caractere está, ou nada (::nil::) caso não encontre o mesmo. Podemos usar então a função `index` para encontrar nosso herói:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            # achei!
        end
    end
    # não achei!
end
```

Vazio, ou nulo, é definido como ::nil:: em Ruby. Não devemos nos confundir com uma ::String:: vazia:

```
string_vazia = ""
nada = nil
nulo = nil
vazio = nil
```

Podemos fazer um `if` para verificar o valor vazio tanto usando a comparação:

```
        if coluna_do_heroi != nil
            # achei!
        end
```

Quanto com um `if` simples. Em Ruby, tudo que não é nulo nem falso é considerado verdadeiro:

```
        if coluna_do_heroi
            # achei!
        end
```

Temos que tomar cuidado quando temos um `nil` em nossas mãos. Suponha por exemplo que uma função retorne uma `String` ou `nil`. Nesse caso, se tentarmos calcular seu tamanho:

```
def pega_nome
  nil
end

nome = pega_nome
puts nome.size
```

Temos um erro que, por padrão, para toda a aplicação:

```
NoMethodError: undefined method `size' for nil:NilClass
```

Afinal, um valor vazio, nulo, não tem o método `size` como uma `String` tem. Cuidado sempre que permitir a existência de um `nil` em seu código, e sempre verifique o retorno de uma função cuja documentação indica que pode retornar `nil`.

Laço funcional básico

Nosso código já está mais compreensível, mas ainda podemos melhorar. Será que não existe um laço que já nos traz tanto o contador (`linha`), quanto o valor daquele laço? Se não existisse, eu não perguntaria aqui? Talvez. Nesse caso, existe. Existe um método de `Array` (que `String` também tem) que permite passar por cada (::each::) elementos dele. Basta dizermos como queremos chamar essas variáveis. Cada elemento é nossa ::linha_atual:::

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each do |linha_atual|
        # cade a linha?
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi != -1
            # achei!
        end
    end
    # não achei!
end
```

Mas qual o número da linha? Usamos o método `each_with_index` para nos dar a posição (::index::) de cada elemento, que chamaremos de ::linha:::

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
```

```
            # achei!
        end
    end
    # não achei!
end
```

Essa maneira de programar, onde chamamos um método e passamos para ele um bloco de código (o conteúdo dentro do ::do::) é o laço mais básico de uma maneira funcional de programar em Ruby.

Extraindo a posição

Agora que encontramos nosso herói, podemos retornar sua posição, tanto a linha quanto a coluna. Tome bastante cuidado pois estamos usando o formato linha/coluna, e não coluna/linha. Colocamos nosso código então no `fogefoge.rb` :

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            return [linha, coluna_do_heroi]
        end
    end
    # não achei!
end
```

O próximo passo é invocar a função em nosso laço principal:

```
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
```

E movimentar de acordo. Para isso, se o usuário digitar `W` , devemos subir, diminuindo em `1` a linha atual:

```
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        case direcao
            when "W"
                heroi[0] -= 1
        end
```

Se ele digitar `S` , ele desce, aumentando em `1` a linha atual:

```
        desenha mapa
        direcao = pede_movimento
```

```
heroi = encontra_jogador mapa
case direcao
    when "W"
        heroi[0] -= 1
    when "S"
        heroi[0] += 1
end
```

Já no movimento horizontal, aumentamos um quando vamos pra direita e diminuimos um para a esquerda:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
case direcao
    when "W"
        heroi[0] -= 1
    when "S"
        heroi[0] += 1
    when "A"
        heroi[1] -= 1
    when "D"
        heroi[1] += 1
end
```

Falta agora reposicionar nosso jogador no mapa. Para isso colocamos o herói na nova posição:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
case direcao
    when "W"
        heroi[0] -= 1
    when "S"
        heroi[0] += 1
    when "A"
        heroi[1] -= 1
    when "D"
        heroi[1] += 1
end
mapa[heroi[0]][heroi[1]] = "H"
```

E antes de movimentá-lo, tiramos ele de sua posição, colocando um espaço em branco lá. Ficamos com a funcão `joga`

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
```

```
        case direcao
            when "W"
                heroi[0] -= 1
            when "S"
                heroi[0] += 1
            when "A"
                heroi[1] -= 1
            when "D"
                heroi[1] += 1
        end
        mapa[heroi[0]][heroi[1]] = "H"
    end
end
```

Testamos o jogo e ele funciona!

```
XXXXX
X H X
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
Para onde deseja ir?
D

XXXXX
X  HX
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
Para onde deseja ir?
```

Lembre-se: estamos usando a letra maiúscula para mover-nos, não minúscula!

Refatorando

A complexidade da função `joga` aumentou rapidamente. Nosso `case` complica muito a compreensão dela, portanto extrairemos esse código. O que ele faz? Ele define a nova posição do herói? Então definimos ele:

```
def calcula_nova_posicao(heroi, direcao)
    case direcao
        when "W"
```

```
            heroi[0] -= 1
        when "S"
            heroi[0] += 1
        when "A"
            heroi[1] -= 1
        when "D"
            heroi[1] += 1
    end
    heroi
end
```

E invocamos ela:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        nova_posicao = calcula_nova_posicao heroi, direcao
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Passagem por referência ou valor?

Para verificar a soma que efetuamos no array `heroi`, retornando seu novo valor, vamos imprimir as duas variáveis que temos:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        nova_posicao = calcula_nova_posicao heroi, direcao
        puts "Antes: #{heroi}"
        puts "Depois: #{nova_posicao}"
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Rodamos o jogo e movemos para a direita:

```
...
D
Antes: [1, 3]
```

```
Depois: [1, 3]
...
```

Como assim? Os dois arrays tem o mesmo valor? O valor movido para a direita? O que está acontecendo aqui?

Vamos olhar com calma a simulação na memória de quando invocamos a função `calcula_nova_posicao`, afinal é ela quem devolve o array. O que acontece quando invocamos uma função com parâmetros? É criado uma nova variável, com o nome do parâmetro, e ela possui o mesmo valor que a variável que foi passada como argumento.

Isso significa que ao passarmos `heroi` como argumento, pegamos todos os valores de `heroi` e copiamos em um novo `array`? Não. Se fizéssemos isso, imagine invocar uma função com um array de 1 mega como argumento. A cada vez que chamamos uma função, ele teria que copiar esse 1 mega de alguma maneira (existem diversas técnicas de otimização), mas o conteúdo mais cedo ou mais tarde seria copiado! Megas e mais megas! Nada bom.

Se toda vez que passássemos um array como argumento esse array fosse copiado por inteiro, o tempo de processamento e o consumo de memória seria muito alto.

Ao mesmo tempo, o que é um array? É um "pedação" de memória que cabe vários valores. Mas o que é uma variável que referencia um array? Quando temos uma variável que referencia um array, ela possui um único valor: um apontador para onde está esse array. Isto é, toda variável possui um único valor. No caso de arrays, esse valor é um número que referencia onde na memória estão as casinhas para colocar diversos valores.

E é justamente esse valor que é passado como parâmetro. Isso significa que quando passamos um array como argumento, o que é copiado é o seu endereço, o apontador. O valor dele não é copiado, e tanto a variável anterior quanto a nova estão apontando para o mesmo - afinal só existe um! - array. Qualquer mudança que fizermos nesse array afeta as duas variáveis. E é isso que aconteceu. Nossa variável `heroi` está referenciando o mesmo array que o `nova_posicao`.

Passamos uma referência para nosso `array` como argumento, não passamos uma cópia de nosso array. Esse foi nosso erro.

Para corrigir ele, podemos pedir para o array ser duplicado (`dup`) assim que recebemos ele:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] -= 1
        when "S"
            heroi[0] += 1
        when "A"
            heroi[1] -= 1
        when "D"
            heroi[1] += 1
    end
    heroi
end
```

Agora sim:

```
...
D
Antes: [1, 2]
Depois: [1, 3]
...
```

Podemos já tirar os dois `puts` .

Detecção de colisão com o muro e o fim do mapa

Ainda existem algumas situações que não tratamos. A principal delas envolve bater com um muro. Não podemos permitir que nosso herói tente andar onde existe um muro. Para resolver isso podemos verificar se sua nova posição possui um muro e, se sim, cancelar o movimento.

Agora que temos a posição exata que iremos nos movimentar, podemos conferir se ela é um muro e, se for, não andar:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        nova_posicao = calcula_nova_posicao heroi, direcao
        if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
            next
        end

        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Mas cuidado! Só quero remover o herói de seu lugar caso ele tenha sido posicionado no lugar novo, isto é, somente se remarcarmos ele. Portanto somente limpamos a posição no mapa caso ele não tenha batido no muro:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = calcula_nova_posicao heroi, direcao
        if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
            next
        end

        mapa[heroi[0]][heroi[1]] = " "
```

```
            mapa[nova_posicao[0]][nova_posicao[1]] = "H"
        end
    end
```

Pronto! Nosso jogo já não permite mais movimentar para dentro de um muro. Falta verificarmos o fim do mapa. Como fazer isso?

O jogador não pode subir se estiver na linha `0`, nem ir para a esquerda se estiver na linha `0`. Isto é, não pode ir para a `nova_posicao[0] < 0`, nem `nova_posicao[1] < 0`:

```
            if nova_posicao[0] < 0
                next
            end
            if nova_posicao[1] < 0
                next
            end
            if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
                next
            end
```

Também não podemos deixar o jogador ir para após a última linha (`mapa.size`) nem para após a última coluna (`mapa[0].size`):

```
            if nova_posicao[0] < 0
                next
            end
            if nova_posicao[1] < 0
                next
            end
            if nova_posicao[0] >= mapa.size
                next
            end
            if nova_posicao[1] >= mapa[0].size
                next
            end
            if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
                next
            end
```

Pronto, evitamos a colisão com muros e sair do mapa.

Refatorando com || e &&

Já temos a primeira parte de nosso jogo funcionando, afinal o jogador é capaz de andar pelo mapa. Mas vamos melhorar nosso código ainda mais antes de continuar uma vez que nossa função `joga` novamente ficou complexa demais.

Após calcular a nova posição, temos diversos `ifs`, que é equivalente a um `switch`, muitas condições, algo complicado de entender rapidamente. Extraímos então uma função que diz se a posição é válida:

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0
        return false
    end
    if nova_posicao[1] < 0
        return false
    end
    if nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

Mas seria ainda melhor se pudéssemos agrupar algumas dessas condições. Por exemplo, ::se:: a posição da linha for menor que `0` ou maior ou igual ao número de linhas.

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0 OU nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] < 0 OU nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

No Ruby é possível fazer a condição ::OU:: com o operador ::||:::

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0 OU nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] < 0 OU nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

|| e &&

Assim como o operador `||` (::OR::) retorna verdadeiro se a primeira ou segunda condição for verdadeira, o operador `&&` (::AND::) retorna verdadeiro somente se ambas forem verdadeiras.

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0 || nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] < 0 || nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

Pronto, nossa função já está mais direta, se extrairmos algumas variáveis o código fica bem mais legível:

```
def posicao_valida?(mapa, nova_posicao)
    linhas = mapa.size
    colunas = mapa[0].size

    estourou_linha = nova_posicao[0] < 0 || nova_posicao[0] >= linhas
    estourou_coluna = nova_posicao[1] < 0 || nova_posicao[1] >= colunas

    if estourou_linha || estourou_coluna
        return false
    end

    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end

    true
end
```

Por fim, renomeamos `nova_posicao` para `posicao`:

```
def posicao_valida?(mapa, posicao)
    linhas = mapa.size
    colunas = mapa[0].size

    estourou_linha = posicao[0] < 0 || posicao[0] >= linhas
    estourou_coluna = posicao[1] < 0 || posicao[1] >= colunas

    if estourou_linha || estourou_coluna
        return false
    end
```

```
    if mapa[posicao[0]][posicao[1]] == "X"
        return false
    end

    true
end
```

E a nossa função `joga` invoca a `posicao_valida?` :

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = calcula_nova_posicao heroi, direcao
        if !posicao_valida? mapa, nova_posicao
            next
        end

        mapa[heroi[0]][heroi[1]] = " "
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Duck typing na prática

Até agora usamos um array de `String` s para representar nosso mapa. Poderíamos usar um array de arrays. Como no nosso caso utilizamos letras, não parece existir nenhuma vantagem em utilizar números (ou caracteres soltos em um array). Mas também não há nenhum método de `String` que estamos utilizando. Tudo indica que a única coisa que utilizaremos é extrair o valor na posição de uma `String` , com o `[]` , e o tamanho de uma linha com o `size` . Como tanto array quanto `String` respondem a esses métodos, tanto faz por enquanto nossa abordagem.

Tanto faz se é um `Array` ou `String` só me preocupo se ele responde ao comportamento que preciso. No mundo animal existe uma analogia famosa (e um tanto estranha) com patos: se ele faz ::quack:: como um pato, não me importa se é um pato, ele faz ::quack:: como um pato.

Por um lado o artifício de invocar um comportamento, independente do tipo que estamos utilizando (chamado de ::duck typigin::), é poderoso, mas pode trazer problemas. O método `size` de um `Array` pode trazer o número de elementos dele, o de `String` também. Portanto qualquer coisa que tem `size` me satisfaz. Será?

Mas e se eu tenho um tipo chamado `Product` (::Produto::), que o método `size` devolve seu tamanho físico? `size` em um contexto tem um significado, em outro tem outro. Boa sorte.

A frase original do ::duck typing:: era "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.", em tradução livre "Quando vejo um pássaro que anda como um pato, nada como um pato e grasna como um pato, eu chamo esse pássaro de pato" (James Whitcomb Riley). O problema é que muito desenvolvedor (inclusive o wikipedia) interpreta isso como uma abstração total da tipagem, em uma frase resumida a "Se anda como um pato, e grassa como um pato, é um pato". Repare a diferença entre a original e a resumida. Na frase

original ainda nos preocupamos com a tipagem do que estamos trabalhando: ainda tem que ser um ser vivo, mais ainda, um animal, mais ainda, um pássaro. Se um homem grasna, anda e nada como um pato, ele não é um pato. E é ai que mora o risco e vantagem do ::duck typing:: em Ruby, ele não verifica nada da tipagem, ele trabalha com a versão reduzida da frase, e não a original. Filosoficamente, não existe "verdade" ou "mentira" na frase original ou resumida. Existem consequências positivas e negativas de toda funcionalidade de uma linguagem, é importante sempre aprendermos todas elas.

Linguagens com ::duck typing:: nos alegram ao permitir tais invocações e ao mesmo tempo pecam em não nos protegerem de erros como esses. Não só um programa com um `Product` rodaria, como o seu resultado seria totalmente inesperado - boa sorte.

Tome cuidado com a ilusão que o ::duck typing:: as vezes nos passa. Não seja enganado quando ensinarem somente o lado positivo de uma funcionalidade de uma linguagem. Sempre aprendemos o bom e o ruim para tomar nossas decisões conscientemente. Esse é o foco desse material. Neste caso atual, você não quer saber o ::tamanho:: de qualquer coisa, você quer saber o tamanho de seu mapa, e isso implica em um significado (uma semântica) bem específica, que no nosso caso, a linguagem Ruby não fornece. Por outro lado, ela permite trocar um tipo por outro "sem nos preocupar" muito.

for i x for linha

Uma prática muito comum no mundo de programação é a abreviação e padronização do nome de variáveis. São diversos os `tot` s que são totais, ou os `s_nome` que são `string` nome. O padrão usado por desenvolvedores ruby é o de não utilizar um caractere no começo de uma variável para indicar qual o tipo daquela variável.

Por outro lado, existem nomes de métodos e variáveis abreviados e outros não. Como boa prática e regra geral, não abrevie. Quanto mais curto mais ambíguo e maior a chance de conflito. Como citado anteriormente, `size` é ambíguo, mas `tamanho` e `quantidade_de_elementos` não. Tudo depende do que você precisa, mas como regra geral evite a ambiguidade e o possível erro do desenvolvedor justamente por não entender o que está fazendo.

Um exemplo clássico de abreviação é o uso da variável `i` para um laço:

```
for i = 0..(mapa.size-1)
    puts mapa[i]
end
```

Se o código dentro de seu laço utilizar a variável que criou para algo importante, não deixe seu nome como um mero `i`, defina ela com o valor real que ela tem, com seu significado dentro do contexto atual:

```
for linha = 0..(mapa.size-1)
    puts mapa[linha]
end
```

Não foi sem querer que até agora não vimos nenhum `for i`. Em todos os instantes que fizemos um laço, as variáveis possuiam algum significado real para nossa aplicação, e ao darmos um nome real, deixamos claro para o próximo desenvolvedor o que aquela variável representa.

Resumindo

A definição de um formato de entrada e saída é um passo fundamental para todo programa que vai gravar dados em algum lugar. Esses arquivos de texto são simples mas já implicam em entendermos como funciona o processo de ::input:: e ::output:: (::IO::).

Vimos como definir arrays de arrays, apesar de usarmos até agora um array de `String` s. Fomos capazes de movimentar nosso jogador, entender o que significa o vazio, o nulo (::nil::), utilizar um laço funcional básico (::each::).

Outra base da programação foi apresentada aqui: a passagem de parâmetros por valor e por referência. Refatoramos nosso código para usar os operadores chamados de ::short-circuit:: ::&&:: e `||` . Por fim, vimos quais são as implicações positivas até agora e o cuidado que devemos tomar com ::duck typing::.